Common 68000 opcodes

Basics: Data is in 32-bit data registers `D0`–`D7`. Addresses are in 32-bit registers `A0`–`A6` (`A7` is the stack).

Most opcodes will work with, for example, `D3` [the value of register D3], `(A1)` [value of location pointed to by A1], `(A4)+` [value at A4, then increment A4], `-(A1)` [decrement A1, then value at A1], `32(A6)` [value at A6+32], `12(A0,A1)` [value at 12+A0+A1], `#15` [the immediate value 15] and `$EE1057` [value at address $\mathrm{EE1057}_{\mathrm{hexadecimal}}$].

Data should be qualified with the suffix `.B` [byte, 8 bits], `.W` [word, 16 bits] or `.L` [long word, 32 bits]. In 'binary' operations the source is the first argument, the destination is the second. Nearly all data processing operations will set the status flags (C [carry], N [negative], Z [equal to zero], etc.).

For number bases, use the prefices `$` [hexadecimal] or `%` [binary] otherwise base 10 [denary] will be assumed.

Common opcodes: These are the commonly used 68000 opcode mnemonics — most simple programs can be written with just these.

Code	**Purpose**
`ADD`	Add two numbers. E.g., `ADD.L #12,D4` [add 12 to the value of register D4]
`AND`	Part of the bitwise logical binary operators, AND, OR, EOR. E.g., `AND.B %00000100,PORTA1` [mask out all the bits except bit 2 from location PORTA1, and write the value back to PORTA1]
`ASL`	[arithmetic shift left] Shifts binary bits left n times (multiplies by 2^n). E.g., `ASL.L #2,D0` [multiplies D0 by 4].
`ASR`	[arithmetic shift right] divides by 2^n, filling any 'empty' bits with the correct twos complement sign.
`Bxx`	[branch on condition code] There are lots of condition codes, but the most useful are `GT` [greater than], `LT` [less than], `GE` [greater than or equal, $\geq$], `LE` [less than or equal, $\leq$], `EQ` [equal to], `NE` [not equal to], `RA` [branch always], `PL` [plus, positive] and `MI` [minus, negative]. E.g., `SUB.L #1,D2; BNE loop` [branch if D2 is not yet zero.] E.g., `CMP.B #100,D0; BGT toobig` [branch if D0 is greater than 100.]
`BSR`	[branch to subroutine] Calls the named routine. The routine must end with an `RTS`.
`BCHG`	[bit change (exclusive or)] See BSET.
`BCLR`	[bit clear (set to zero) See BSET. E.g., `BCLR.L #7,D0` [clear bit 7 of D0 to a zero] E.g., `BCLR.B D5,D0` [clear bit D5 of D0 to a zero, note the `.B`]
`BSET`	[bit set (set to one)] This set of 4 opcodes tests the named bit, then sets, clears, exclusive-ors or leaves it. Useful for I/O.
`BTST`	[bit test (leave the bit in its previous state)] See BSET. E.g., `BTST.B #3,PORTB2; BEQ off` [test bit 3 of port B2 and branch away if it was a zero]
`CLR`	[clear] Sets the destination to zero. E.g., `CLR.L (A2)` [clears the 32 bit value at location A2 to zero]
`CMP`	[compare] Subtracts the source from the destination but doesn't store the result anywhere. It's useful because it sets the status flags. See the example in BGT.
`EOR`	See AND.
`MOVE`	Moves one value into register. E.g., `MOVE.B (A1)+,PORTA2` [moves the value at the location pointed to by A1 into location PORTA2 then increments A1] E.g., `MOVE.L #$2000,D0` [moves the value 8192 into D0]
`NOP`	[no operation] Does nothing except waste time (2 cycles) and space (one byte).
`OR`	See AND.
`RTS`	[return from subroutine] Passes control back to the location that called the subroutine (normally a BSR operation). If RTS is called from the main routine, the program will finish.
`SUB`	Subtract a number from another. E.g., `SUB.L #10,D4` [subtract 10 from D4]