68000 Assembly

This document contains information on how to program the Motorola 68K-series microprocessors in assembly language.

The Motorola 68K series of microprocessors was used by many manufacturers:

- Apple in their Macintosh computers until they switched to the PowerPC series of microprocessors, and in their LaserWriter printers until they switched to the AMD 29000 series of microprocessors.
- The Commodore Amiga
- Atari ST/STE/TT/Falcon line of personal computers.
- Sinclair for the Sinclair QL (which was also re-badged as the ICL one-per-desk).
- NeXT before they became a software-only company.
- Palm in their handheld PDAs until they switched to the ARM series of microprocessors. Almost all applications run in emulated 68K mode, however.
- The 680x0 was popular in arcade machines until the middle of the 1990s. It can be found in many Capcom games.
- The Sega Mega Drive (Genesis in the U.S.) used a 68000 as its main processor. Its successor, the Saturn, used one as its sound processor.
- The Atari Jaguar had a 68000 as the central CPU among many dedicated processors.
- The Texas Instruments Calculators TI-89, TI-92, Voyage 200 and TI-89 Titanium
- Many embedded computer systems use 68K family microprocessors, often running real-time operating systems.

One thing to note is that the PowerPC is *not* binary compatible with the 68K processor. Their assembly languages are completely different. However, Apple has written an emulator (in PowerPC assembly language) which allows PowerPC microprocessors to interpret machine language code written for 68K microprocessors, albeit with a substantial performance decrease versus native PowerPC machine language.

The Motorola 68K is a CISC-based CPU that operates on memory organized in a Big-Endian fashion.

Registers

With the exception of the 16-bit status register, all 68000 registers are 32-bits wide.

There are eight data registers: d0, d1, d2, d3, d4, d5, d6, and d7. Which are intended to hold numbers that are having various mathematical and logical operations performed on them.

There are seven address registers: a0, a1, a2, a3, a4, a5, and a6. Which are typically used as pointers.

There is one active stack pointer: SP, also called a7. Normally the processor is in user mode. In user mode, SP refers to the User Stack Pointer (USP) register. (During interrupts, the active stack pointer SP is another register called the Interrupt Stack Pointer or the System Stack Pointer. The 68020 and higher processors have a third register called the Master Stack Pointer. Neither the ISP nor the MSP can be accessed in user mode).

The 68K includes special addressing modes that make it easy to manipulate a data stack structure using *any* address register.

The Program Counter (PC) points to the current instruction. On the 68000, only the lower 24 bits output to any pins, giving a maximum addressing range of 16MiB. The Program Counter is changed automatically when a new instruction is loaded or when a BRA, Bcc, BSR, JMP, Jcc, JSR, RTS, or RTE instruction is used. It can also be used as a pointer in PC relative addressing modes.

The Condition Code Register (CCR) consists of the lower byte of the Status Register (SR). Only the lower byte is accessible in user mode, and of this, only the first five bits are useful. In supervisor mode, the entire 16-bit register is accessible. The register looks like this:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T1	T0	S	M	0	I2	I1	I0	0	0	0	X	N	Z	V	C

Here's the decoded register contents:

Bit	Description
C	Carry
V	Overflow
Z	Zero
N	Negative
X	Extend
I0	Interrupt priority mask bit 1
I1	Interrupt priority mask bit 2
I2	Interrupt priority mask bit 3
M	Master/Interrupt switch. Determines which stack mode to use if S is set. If M is clear, SP refers to ISP, else SP refers to MSP. This bit is always clear on processor models lower than 68020.
S	Supervisor Mode flag. If clear, SP refers to USP. If set, look at M to determine what stack SP points to.
T0	Trace bit 1. If set, trace on change of program flow. This bit is always cleared on processor models lower than 68020.
T1	Trace bit 2. If set, trace is allowed on any instruction. **DO NOT SET BOTH TRACE BITS AT THE SAME TIME!**

Addressing Modes

n is a number between 0 and 7 denoting which register to use.

Immediate addressing with data registers

Assembler syntax

- Dn

Directly operate on the contents of a data register. Example:

```
MOVE.L D1,D0
```

Copies the contents of D1 to D0. When the instruction is executed, both registers will contain the same information. When moving a byte or a word, the upper part of the register will remain unchanged.

```
              ;lets assume
              ;D0=FFFFFFFF D1=01234567
 MOVE.B D1,D0 ;copies a Byte from source
              ;D0=FFFFFF67 D1=01234567
....
              ;lets assume
              ;D0=FFFFFFFF D1=01234567
 MOVE.W D1,D0 ;copies a Word from source
              ;D0=FFFF4567 D1=01234567
```

Immediate addressing with address registers

Assembler syntax:

- An

Directly operate on the contents of an address register.Example:

```
MOVE.L A1,D0
```

Copies whole A1 to D0. After the instruction, both registers contain the same information. When transferring with ADDRESS registers you must use word or longword. When a word is transferred to an address register, bit 15 (the sign bit) will be copied through the whole upper word (bit 16-31). If it wasn't so, a negative number would become positive.

```
              ;lets assume
              ;D0=FFFFFFFF A1=01234567
 MOVE.W A1,D0 ;copies a Word from source
              ;D0=FFFF4567 A1=01234567
....
              ;lets assume
              ;D0=01234567 A1=FFFFFFFF
 MOVE.W D0,A1 ;copies a Word from source
              ;D0=01234567 A1=00004567
              ; sign to A1, changed
....
              ;lets assume
              ;D0=0000FFFF A1=00000000
 MOVE.W D0,A1 ;copies a Word from source
              ;D0=0000FFFF A1=FFFFFFFF
              ; sign to A1, changed
```

Indirect addressing

Assembler syntax:

- (An)

Operate on the memory location pointed to by An. For example:

```
lea $1234,A1
move.w D0,(A1)
```

will move the first 16 bits of D0 into the word starting at $1234. Another example:

```
MOVE.L (A0),D0
```

Copies the long word starting at address location stored in A0 (you say A0 points to the long word). If you refer to a word or a long word, the address in the address register must be an EVEN number. Take care with this!!!

```
               ;lets assume
               ;D0=FFFFFFFF A1=00001000
               ;mem.addr. $1000=01234567
 MOVE.L (A1),D0 ;copies Long word starting at memory address stored in A1, to D0
               ;D0=01234567 A1=00001000
               ;mem.addr. $1000=01234567
```

Indirect addressing with postincrement

Assembler syntax:

- (An)+

Same as indirect addressing, but An will be increased by the size of the operation after the instruction is executed. The only exception is byte operations on A7 - this register must point to an even address, so it will always increment by at least 2. Example:

```
MOVE.L (A1)+,D0
```

Copies to D0 the longword which A1 points, and increases A1 with 4 (because of Long).

```
                ;lets assume
                ;D0=FFFFFFFF A1=00001000
                ;mem.addr. $1000=01234567
MOVE.L (A1)+,D0 ;copies the Long word starting at address stored in A1, to D0
                ;then increment A1 by 4
                ;D0=01234567 A1=00001004
                ;mem.addr. $1000=01234567
```

Indirect addressing with predecrement

Assembler syntax:

- -(An)

Same as indirect addressing, but An will be decrememted by the size of the operation BEFORE the instruction is executed. The only exception is byte operations on A7 - this register must point to an even address, so it will always decrement by at least 2. Note that there is no postdecrement or preincrement addressing mode. Example:

```
MOVE.L -(A0),D2
```

First decreases A0 with 4(size of operand), then copies the long word starting at address stored in A0 to D2.

```
                ;lets assume
                ;D0=FFFFFFFF A1=000010A8
                ;mem.addr. $10A4=01234567
MOVE.L -(A1),D0 ;first decrements A1 by 4, A1=000010A4
                ;then copies the Long word starting at address stored in A1, to D0
                ;D0=01234567
                ;mem.addr. $10A4=01234567
```

Indirect addressing with shifting

Assembler syntax:

- x(An)
- (x)(An)
- (x,An)

Operate on the location pointed to by x + An, where x is a 16-bit immediate value. All listed syntaxes are equivalent, but some assemblers won't accept them all.

Indirect addressing with index

Assembler syntax:

- x(An,Dn.W)
- x(An,Dn.L)
- x(An,An.W)
- x(An,An.L)
- (x)(An,Dn.W)
- (x)(An,Dn.L)
- (x)(An,An.W)
- (x)(An,An.L)
- (x,An,Dn.W)
- (x,An,Dn.L)
- (x,An,An.W)
- (x,An,An.L)

Same as above, but another register will also be added. Not all assemblers will take all listed syntaxes.

Absolute near addressing

Assembler syntax:

- (xxx).W
- xxx.W

Operate on the location pointed to by xxx. xxx is sign-extended by the assembler. You can write this either with or without the parentheses, and most assemblers can take either one. Which you choose is largely a matter of personal preference, but most people find (xxx).W easier to read.

Absolute far addressing

Assembler syntax:

- (xxx).L
- xxx.L

Operate on the location pointed to by xxx. xxx is sign-extended by the assembler. Some instructions only accept one or the other of near or far absolute addresses, thus the separation. Like absolute near, you can include the parentheses at your discretion.

Program Counter with displacement

Assembler syntax:

- x(PC)
- (x)(PC)
- (x,PC)

Operate on the memory value at x + PC, where x is a 16-bit immediate value. Note that PC is the address of the extension word that x is stored in (right after the instruction's word). All syntaxes are equivalent, but some assemblers won't take them all.

Program counter with index

Assembler syntax:

- x(PC,Dn.W)
- x(PC,Dn.L)
- x(PC,An.W)
- x(PC,An.L)
- (x)(PC,Dn.W)
- (x)(PC,Dn.L)
- (x)(PC,An.W)
- (x)(PC,An.L)
- (x,PC,Dn.W)
- (x,PC,Dn.L)
- (x,PC,An.W)
- (x,PC,An.L)

Like PC with displacement, but another register is added as well. Some assemblers won't take certain syntaxes.

Immediate addressing

Assembler syntax:

- #xxx

Operate on xxx.

Status Register addressing

Assembler syntax:

- SR
- CCR

SR is the entire status register, including the system byte. CCR is just the flags. Other than that, I don't know how this works. SR is only available in supervisor mode.

The only instructions that are allowed to use this addressing mode are: MOVE, ANDI (AND immediate), EORI (exclusive OR immediate) and ORI (OR immediate) known as: MOVE to/from CCR, MOVE to/from SR, ANDI to CCR, ORI to CCR, EORI to CCR, ANDI to SR, ORI to SR, EORI to SR.

```
           ;Example ORI to CCR
           ;lets assume CCR=$00
ORI #5,CCR ;sets both the carryflag (C) and the zeroflag (Z)
           ;CCR = $05 -> 00000101
           ;most assemblers recognize both SR and CCR
           ;so you don't have to specify the length of operand.
```

Conditional tests

Wherever you see "cc" in an instruction, you should replace it with the appropriate conditional test code. Refer to this table for what each test does.

Code	Description
T*	True, always tests true. Not available for Bcc or Jcc.
F	False, always tests false. Not available for Bcc or Jcc.
HI	High. True if Carry and Zero are both clear.
LS	Low or same. True if either Carry or Zero are set.
CC	Carry Clear. True if Carry is clear.
CS	Carry Set. True if Carry is set.
NE	Not Equal true if zero flag is set
EQ	Equal, true if zero flag is clear
VC	Overflow Clear. True if Overflow is clear.
VS	Overflow set. True if Overflow is set.
PL	Plus. True if Negative is clear.
MI	Minus. True if Negative is set.
GE	Greater or Equal
LT	Less Than
GT	Greater Than
LE	Less than or Equal

*As a compromise, most 68k compilers use BRA (BRanch Always) instruction for this value of "cc" field.

Labels

Labels are simply names for lines. You can have as many labels as you want. Typically, there are only a few places you'll want to refer to, for example the starting points of functions, loop starts and loop ends, and certain data storage locations.

The assembler handles labels as aliases for numbers. When it encounters one, it assigns it the current value of the assembler's PC. (I will refer to this as "declaring" the label.) This label can then be used as an operand anywhere a number can. They are usually used in Jcc or Bcc instructions.

Note that you can reference labels before they're actually declared. This is known as *forward referencing*, and is handled differently depending on the assembler. Usually, it just uses a known safe value (like the current PC), flags the location, and makes a second pass to substitute the real value. This may change the size of the label, in which case a third pass will be needed, and so on. Some assemblers require you to explicitly define the size of a jump/branch to keep from having to make a third pass. The assembler you use may have different behavior.

Labels and storage space:

```
MYDATA:

ds.b 16              ; Declare 16 bytes of space, MYDATA label refers to this location.

global Main          ; Inform assembler that the Main label should be exported (made visible to code not in this file)


Main:                ; Label for the main function

MOVEA.L #MYDATA, A0  ; Set address register A0 to refer to MYDATA space
```

```
MOVE.B #0, 0(A0)     ; Copy 0 into the first byte of MYDATA

MOVE.B #1, 1(A0)     ; Copy 1 into the second byte of MYDATA

rts                  ; Return
```

Instruction Set

The 68K instruction set is very orthogonal. Most instructions can operate on all data sizes, and very few are restricted to less than three addressing modes. 68K instructions are rather slow, but they do a lot more than instructions for the Z80 or x86 processors.

Detailed descriptions of every instruction in the MC68000 family can be found in the Programmer's Reference Manual. See External Links below.

```
Insert instruction table here - this version will probably have far more
columns than the tables that were here.
```

External Links

MC68000 Programmer's Reference Manual [1]

References

[1] http://www.freescale.com/files/archives/doc/ref_manual/M68000PRM.pdf

Article Sources and Contributors

68000 Assembly *Source*: http://en.wikibooks.org/w/index.php?oldid=2265030 *Contributors*: -OOPSIE-, Adamantix, Adrignola, Aurochs, Avicennasis, Chuckhoffmann, Darklama, DavidCary, Hyperwiz, JamesCrook, Jomegat, Jwhitehorn, Krischik, Richiez, Robert Horning, Thereen, WarrenWilkinson, Webaware, Whiteknight, 38 anonymous edits

License