Chapter 6 Assembly Language

Operand Size

Computers operate on chunks of data composed of a particular number of bits.

The 68K has a 32-bit architecture and a 16-bit organization.
- Internal registers hold 32 bits
- Can perform operations on 32 bits
- Has a 16 bit external data bus
  - operands must be moved in and out of memory as two 16 bit words
- Some internal data paths are 16 bit.
- 68K can handle 16 bit and 8 bit values.

The 68K has a byte-addressable architecture—
- Successive bytes are stored at consecutively numbered byte addresses
- Any byte can be individually written to or read from.
- 16-bit words are composed of 2 bytes and are stored at consecutive even addresses: 0, 2, 4, . . .
- 32-bit longwords are stored at addresses 0, 4, 8, 12. . .

When we store a 32 bit longword, where do the bytes go?
Consider the example of storing $12345678 into memory location $1000.
The 68K would store it as:

| 1000 | 12 | 34 | 96 | 78 |

Where the most significant byte is at the lowest address. This order is called **Big-Endian**

If the bytes are stored in the opposite order, with the most significant byte at the highest address like:

| 1000 | 78 | 96 | 34 | 12 |

Where the least significant bytes at the lowest address, we call the order **Little-Endian**.

When we encounter a command like

**MOVE.W DO,1234**

The following bit copy is performed:

<table>
<thead>
<tr>
<th>1232</th>
<th>d15</th>
<th>d16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1224</td>
<td>1225</td>
<td></td>
</tr>
</tbody>
</table>

In the 68K all addresses are 32 bit values. This should allow for an address space of

\[ 2^{32} \text{ bytes} = 4 \text{ G of memory.} \]

Here

\[ 2^{10} = 1024 = 1 \text{ K} \]
\[ 2^{20} = 1,048,576 = 1 \text{ M} \]
\[ 2^{30} = 1,073,741,824 = 1 \text{ G} \]

However, the original design was for a 64-pin package which limited it somewhat. The address bits A24-A31 are not connected to pins and cannot access memory. This effectively forces the address bus to 24 bits and the original 68K can only access

\[ 2^{24} = 2^{8} \times 2^{16} \] or 16 M of memory.

The internal registers

The 68K has a register-to-memory architecture. This means typical instructions will specify one operand in memory and the others as a register like:

**ADD $1234,D0**

which causes the contents of location $1234 to be added to D0

The 68K consists of 8 32-bit data registers, 8 32-bit address registers, an 8 bit condition code register, an 8 bit status byte (used by operating system to define the operating mode of the 68K) and a 32-bit program counter.
The programmer's view of the 68K:

<table>
<thead>
<tr>
<th>d15</th>
<th>d14</th>
<th>d13</th>
<th>d12</th>
<th>d11</th>
<th>d10</th>
<th>d9</th>
<th>d8</th>
<th>d7</th>
<th>d6</th>
<th>d5</th>
<th>d4</th>
<th>d3</th>
<th>d2</th>
<th>d1</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a15</th>
<th>a14</th>
<th>a13</th>
<th>a12</th>
<th>a11</th>
<th>a10</th>
<th>a9</th>
<th>a8</th>
<th>a7</th>
<th>a6</th>
<th>a5</th>
<th>a4</th>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>a0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PC | X | N | Z | V | C | CCR

Some C code:

```c
int int_power(register int m, register int e)
{
    register int temp;
    temp = 1;
    for (; e > 0;)
        temp = temp * m;
    return temp;
}
```

The number of actual register variables permitted depends on the environment and implementation of C. For the sake of portability if no registers are available the compiler will automatically transform register variables to non register variables, if needed.

All of the data registers are general-purpose, commands which can be applied to D0 can be applied to any data register.

We know one advantage of multiple data registers is to store frequently used data on chip for quicker access. The number of internal data registers is a function of economics. Most instructions store the op-code and operand register in a 16-bit word. It requires 3 bits to describe the 8 data registers. More registers would require more bits, leaving fewer bits dedicated to op-code and hence fewer instructions capable of being described in this length word.

Memory requires a 32 bit specification. Instructions that access memory are longer than instructions that access registers.

Most high level languages do not allow for the programmer to specify the use of registers for variable allocation – this is the compiler's job.

C is one of the exceptions:

In C there is a storage class specifier called `register`. It can be applied to local variables and formal parameters in functions. The class specifier is intended to be used where many references will be made to the same variable. The register specifier originally applied only to variables of type int or char (and indeed the C standard permits the compiler to ignore the specifier – the current standards state “access to the object be as fast as possible”)

```c
int int_power(register int m, register int e)
{
    register int temp;
    temp = 1;
    for (; e > 0;)
        temp = temp * m;
    return temp;
}
```

When referring to the bits of a data register (for purposes of RTL) we will denote consecutive bits i to j in register Dn by Dn(i:j). If we wanted to refer to bits 16 through 31 of register D5 we would write D5(16,31)

Recall we specify operations as either byte, word or longword operations by appending the .B, .W or .L to the mnemonic. Each affects a certain sequence of bits of the operand:

- `.B` D0
- `.W` D0
- `.L` D0

The remaining bits would remain unchanged.

```assembly
CLR.W D1
```

Would force the low 16 bits of D1 to 0s

D1 = XXXXXXXXXXXXXXXXXXX0000000000000000000000

Where X stands for whatever was already there.

If [D1] = $12345678 then after

CLR.W D1

we have [D1] = $12340000

### Assembly form | RTL form
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.L D0, D1</td>
<td>[D1(0:31)] ← ([D1(0:31)] + [D0(0:31)])</td>
</tr>
<tr>
<td>ADD.W D0, D1</td>
<td>[D1(0:15)] ← ([D1(0:15)] + [D0(0:15)])</td>
</tr>
<tr>
<td>ADD.B D0, D1</td>
<td>[D1(0:7)] ← ([D1(0:7)] + [D0(0:7)])</td>
</tr>
</tbody>
</table>

We can omit the slice notation unless we are meaning to emphasize what bits we are working on.
The affect on the carry bit of the .B, .W or .L is determined only by the corresponding bits for that size operation.

Consider the following example:

\[ [D0] = \$12345678 \quad [D1] = \$ABCDEF98 \]

In hexadecimal:

<table>
<thead>
<tr>
<th>12345678</th>
<th>ABCDEF98</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>BE024610</td>
<td>78 78</td>
</tr>
</tbody>
</table>

Assemble code | Register D1 | CCR carry bit C
---|---|---
ADD.B D0,D1 | [D1] = ABCDEF10 | [C] = 1
ADD.W D0,D1 | [D1] = ABCD4610 | [C] = 1
ADD.L D0,D1 | [D1] = BE024610 | [C] = 0

It is easy to make an error if we do not pay attention to using consistent size operations on data registers.

Consider the following:

MOVE.B XYZ,D0  
SUB.B #5,D0  
CMP.W #12,D0  
BGT Test

Notice on the MOVE.B we are only changing bytes \(d_{00} \cdots d_{07}\). The same is true when we subtract 5 from D0. When we get to the third step, however we are comparing \(d_{00} \cdots d_{15}\) to the word representation of 12. The problem is we don't have any idea what is in \(d_{08} \cdots d_{15}\). We may or may not wind up with the desired result.

### 68K Address Registers

The 68K's eight 32-bit address registers A0–A7 act as pointer registers, since they hold the address of some memory location. This will be important when dealing with arrays, matrices, tables and vectors.

A0 – A6 are identical in the way they are treated. A7 has the additional responsibility of being a stack pointer for storing subroutine addresses.

Many of the operations we did with data registers can be done with address registers with a few differences:

- Operations on address registers do not normally affect the CCR's bits
- All operations on the address registers are longword operations (Ai's are treated as single entities.
- Any .W operation is automatically extended to a .L operation
- .B operations aren't permitted

The contents of an address are treated as a signed 2's complement value. Performing a .W operation causes the sign bit to be extended from A15 to A31 – A15.

MOVE.W #$8022,A1  
(802216 = 10000000001000102)

has the effect  

\[ [A1] = \$00008022 \]

Why do we allow for negative addresses?

We can interpret a negative address as a move backwards.

Example:

Suppose A1 contains 1280 and A2 contains -40. Adding A1 and A2 results in a composite address 1240, 40 units in address lower than A1.

The 68K has special mnemonics for operations that modify addresses. Some examples (destination operand is address register)

- ADDA.L D1,A3  
- MOVEA.L D1,A2  
- SUBA.W D1,A3

Example: Address register A0 points to the beginning of a data structure in memory. The structure is made up of 50 items numbered 0 through 49. Each of the 50 items is 12 bytes long. D0 contains the number of the item w wish to access. We wish to put the address of the item in A1. We will use the operation

\[ \text{MULU } #n, D0 \]

which multiplies the 16-bit low order word in D0 by n and puts a 32-bit product in D0.

Using an address register to access a data element:

\[ \text{MULU } #12, D0 \]

\[ \text{MOVEA.L } A0, A1 \]

\[ \text{ADD.A. } D0, A1 \]

As you can see the instruction set is rather primitive compared to high level languages. Any type of complex operation will be implemented by a sequence of primitive instructions.
An Introduction to the 68K instruction set

Data Movement

The largest set of instructions in most programs are for data movement. When we omit the .W, .L or .B qualifier we will assume a default of .W, where it makes sense.

Two-operand instructions will have the format

```
Operation source,destination
```

Example: MOVE D3,D4 copies the contents of D3 into D4

Some 68K movement instructions

Assembly

<table>
<thead>
<tr>
<th>Operation</th>
<th>RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE Dl,Dj</td>
<td>[Di] ← [Dj]</td>
</tr>
<tr>
<td>MOVE P,Di</td>
<td>[Di] ← [M(P)]</td>
</tr>
<tr>
<td>MOVE Di,P</td>
<td>[M(P)] ← [Di]</td>
</tr>
<tr>
<td>EXG Dl,Dj</td>
<td>[TEMP] ← [Di], [Di] ← [Dj], [Dj] ← [TEMP]</td>
</tr>
<tr>
<td>SWAP Di</td>
<td>[Di(0:15)] ← [Di(16:31)], [Di(16:31)] ← [Di(0:15)]</td>
</tr>
<tr>
<td>LEA P,Ai</td>
<td>[Ai] ← P</td>
</tr>
</tbody>
</table>

LEA Load Effective Address

In the next example we will demonstrate these instructions using our assembler:

Create a text file: samp.x68

```
ORG $1000
MOVE.L #$12345678,D0
MOVE.B D0,D1
MOVE.W D0,D2
MOVE.L D0,D3
EXG D0,A0
SWAP D3
MOVEA.L data,A1
LEA data,A1
STOP #$2700
DATA DC.L $ABCDDCBA
END $1000
```

From the DOS prompt type (and see the response):

C:\WINDOWS\Desktop\assem\68Ksim>x68k samp -L

No errors reported.

Two Files should have been created in your directory:

Samp.bin, Samp.lis

First we will look at Samp.lis to see what the assembler did

```
Samp.lis (you should be able to open this with any text editor)
Source file: SAMP.X68
Assembled on: 01-10-30 at: 07:01:50
by: X68K PC -2.2 Copyright (c) University of Teesside 1989,99
Defaults: ORG $0/FORMAT/OPT A,BRL,CEX,CL,FRL,MC,MD,NOMEX,NOPCO

1  00001000                        ORG       $1000
2  00001000 203C12345678           MOVE.L    #$12345678,D0
3  00001006 1200                   MOVE.B    D0,D1
4  00001008 3400                   MOVE.W    D0,D2
5  0000100A 2600                   MOVE.L    D0,D3
6  0000100C C188                   EXG       D0,A0
7  0000100E 4843                   SWAP      D3
8  00001010 227900001020           MOVEA.L   DATA,A1
9  00001016 43F900001020           LEA       DATA,A1
10  0000101C 4E722700               STOP      #$2700
11  00001020 ABCDDCBA     DATA:     DC.L      $ABCDDCBA
12           00001000               END       $1000

Lines: 12, Errors: 0, Warnings: 0.
```

Next we will use the simulator to run our assembled code:

C:\WINDOWS\Desktop\assem\68Ksim>e68k samp

```
Address space 0 to ^10485759 (10240 kbytes).
Loading binary file "SAMP.BIN".
Start address: 001000, Low: 00001000, High: 00001023
```

At this point we will create a log of our interaction with the simulator at the '>' prompt:

```
> log fun.log
```

I will step through the program, and when done show y'all the Log file.

(remember typing "quit" at the simulators prompt will quit the simulator, df – display formatted registers and tr- trace, to exit trace mode type a ")
Another point: Computers execute instructions even if the instructions don’t make sense. Higher level languages are often strongly typed and would prevent the programmer from doing such things. Consider the following assembly language program that adds the ASCII representation for character ‘A’ to that of character ‘B’.

```
DIVU D1,Dj
MULU D1,Dj
```

Here [X] is the contents of the X bit of the CCR (usually a copy of the carry bit)

The multiplication and division are specific with regards to what they operate on. The other operations can be designated .B, .L or .W

#### Arithmetic and Logical operations

<table>
<thead>
<tr>
<th>Assembly Form</th>
<th>RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD D1,Dj</td>
<td>Di ← [Di] + [Dj]</td>
</tr>
<tr>
<td>ADDX D1,Dj</td>
<td>Di ← [Di] + [Dj] + [X]</td>
</tr>
<tr>
<td>SUB D1,Dj</td>
<td>Di ← [Di] - [Dj]</td>
</tr>
<tr>
<td>SUBX D1,Dj</td>
<td>Di ← [Di] - [Dj] - [X]</td>
</tr>
<tr>
<td>MULU D1,Dj</td>
<td><a href="31">Dj</a> ← <a href="">Dj</a> × <a href="">Dj</a></td>
</tr>
<tr>
<td>DIVU D1,Dj</td>
<td><a href="">Dj</a> ← remainder</td>
</tr>
</tbody>
</table>

The code:

```
ORG $1000
MOVE.B Char1,D0
MOVE.B Char2,D1
ADD.B D0,D1
STOP #$2700
Char1 DC.B 'A'
Char2 DC.B 'B'
END $1000
```

The .lis file after assembled:

```
END $1000
Char2 DC.B 'B'
Char1 DC.B 'A'
STOP #$2700
ADD.B D0,D1
```

The Log file for a run:

```
Logging to "FUN.LOG" started.

Lines: 9, Errors: 0, Warnings: 0.
```

The code:

```
ORG $1000
MOVE.B Char1,D0
MOVE.B Char2,D1
ADD.B D0,D1
STOP #$2700
Char1 DC.B 'A'
Char2 DC.B 'B'
END $1000
```

The Log file for a run:

```
Logging to "FUN.LOG" started.

Lines: 9, Errors: 0, Warnings: 0.
```

The code:

```
ORG $1000
MOVE.B Char1,D0
MOVE.B Char2,D1
ADD.B D0,D1
STOP #$2700
Char1 DC.B 'A'
Char2 DC.B 'B'
END $1000
```

The Log file for a run:

```
Logging to "FUN.LOG" started.

Lines: 9, Errors: 0, Warnings: 0.
```
The assembly code:

ORG $1000
X DC.W 50 ; value for X
Y DC.W 12 ; value for Y
Z DS.W 1 ; create space for result
ORG $1200
MOVE.W X,D0 ; Copy X to D0
MULU D0,D0 ; compare X
MOVE.W Y,D1 ; Copy Y to D1
MULU D1,D1 ; compare Y
ADD.L D0,D1 ; X^2 + Y^2 into D1
MOVEL X,D0 ; Copy X to D0
SUB.W Y,D0 ; X - Y into D0
DIVU DO,DO ; put(X^2+Y^2)/(X-Y) into D0
MOVE.W D1,Z ; put result out to memory Z
STOP $12700 ; End with address entry point

The assembled code:

Source file: SAMP.X68
Assembled on: 01-16-30 at: 14:16:33
by X68C PC-2.2 Copyright (c) University of Twente 1989
Defaults: ORG 0 FORMAT/OPT
A,RKU,CXU,FLU,ME,MD,HOMUX,HOPCO

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001000</td>
<td>DC.W 12</td>
<td>Copy X to D0</td>
</tr>
<tr>
<td>00001002</td>
<td>DC.W 12</td>
<td>Copy Y to D1</td>
</tr>
<tr>
<td>00001004</td>
<td>ORG $1200</td>
<td>Create space for result</td>
</tr>
<tr>
<td>00001012</td>
<td>DC.W 50</td>
<td>Value for X</td>
</tr>
<tr>
<td>00001014</td>
<td>DC.W 12</td>
<td>Value for Y</td>
</tr>
<tr>
<td>00001200</td>
<td>ORG $1100</td>
<td>End with address entry point</td>
</tr>
<tr>
<td>00001206</td>
<td>MOV.L X,D0</td>
<td>Copy X to D0</td>
</tr>
<tr>
<td>0000120C</td>
<td>MOV.L Y,D1</td>
<td>Copy Y to D1</td>
</tr>
<tr>
<td>00001212</td>
<td>SUB.L D0,D0</td>
<td>X - Y into D0</td>
</tr>
<tr>
<td>00001214</td>
<td>DIVU D0,DO</td>
<td>Put(X^2+Y^2)/(X-Y) into D0</td>
</tr>
</tbody>
</table>

The log file (with comment)
Logging to "FUN.LOG" started.

Subtracted 12

The log file (with comment)
Logging to "FUN.LOG" started.

What is the meaning of what is in D1?

Here is a program to calculate $Z = \frac{X^2 + Y^2}{X - Y}$

For X=50, Y=12
Answer should be: $\frac{50^2 + 12^2}{50 - 12} = 69.5789$

Or = 69 R 22
Trace>
PC=00121C SR=0200 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 M=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000000 D1=00160045 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
========>STOP    #$2700

Trace>
Processor halted at: 001220
PC=001220 SR=0270 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 M=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000000 D1=00160045 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
========>ORI.B        #$00,D0
MD - Memory Display command
Trace>MD 1000
001000 00 00 0C 00 45 00 00 00 00 00 00 00 00 00 00
001010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00q
>quit