Chapter 3
M68000 Instruction Set and Basic Programming

3.1. Introduction

In this chapter we begin to ‘‘peel the onion.’’ This ‘‘onion’’ consists of the different components of an embedded computer system. We begin by describing some very simple elements of the toolkit of the programmer of a microprocessor. The goal is to remove any abstractions and allow understanding of the machine ‘‘at it’s own level.’’ As we attempt to do more and more complex things with a microprocessor and various I/O devices, we will need to understand more and more of the sophisticated aspects of a system. Until then, however, we will just peel one layer of the ‘‘onion’’ at a time.

3.2. The Basic Programmer’s Model

The basic programmer’s model consists of the components shown in Figure 3.1.

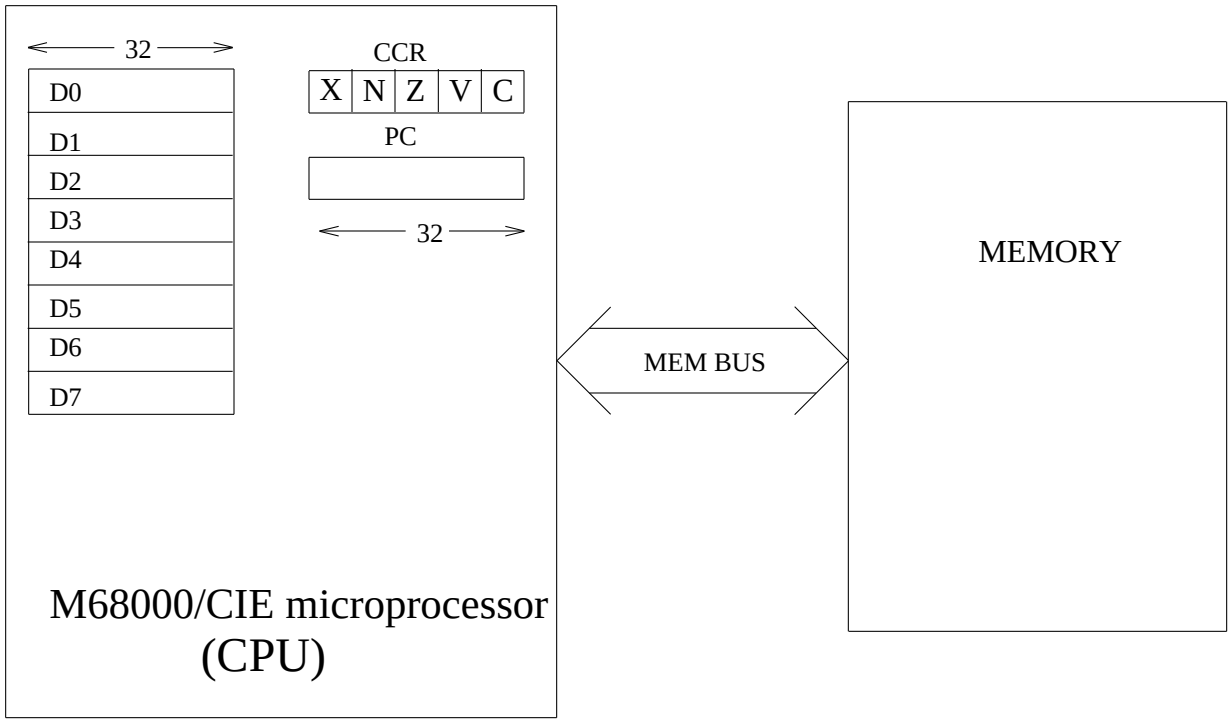

Figure 3.1. Simplified M68000 Programmers Model.

The CPU executes instructions that operate upon data stored in the Memory. All data must travel the path from memory to the CPU to be operated upon, and then make the return trip back to memory to be stored until needed for future operations. Not only does the memory store data between operations, but the memory also contains the *instructions* themselves which get executed by the CPU. This aspect is what we referred to in Chapter 1 as the *stored program*

concept.

While in the CPU, data need a place to be stored. The name we use for these places is **registers**. In our model of the M68000, there are 3 types of registers.

1. First are a set of **data registers**. These have very specific names: D0, D1, D2, D3, D4, D5, D6, D7. There are exactly eight of them. They are used to hold copies of data from the memory. It is these copies that may be operated upon by the instructions. Each data register is 32-bits in length.
2. Because instructions must be stored in memory, the CPU must have some idea of where the next instruction (to execute) is. This ''address'' is stored in the **program counter** register (PC). The PC is 32 bits in length. Normally, the program counter is incremented by one after each instruction. I.e., the next instruction to execute usually immediately follows the current one being executed. This is what is commonly referred to as *sequential* instruction execution. We will discuss ways to alter this sequential behavior later.
3. The third class of register is one which saves information about ''special'' conditions that may arise during the execution of an instruction. In the M68000, we call this register the **condition code register** (CCR), and it consists of 5 bits, each having a special meaning. A simple example of one of these is the 'Z' bit which is used to determine whether the result of a subtraction operation was zero or not.

In addition to the above-mentioned components of the CPU, the memory also consists of a very large number of components. *Luckily*, the components are all identical in purpose and function. We say that the memory consists of a set of **locations**. Each such location has a unique **address**. The addresses are simply numbers, and each address refers to an 8-bit, or one byte, unit of storage. If we have a 1 kilobyte (1 KB) memory, then there are exactly $2^{10} = 1024$ locations. Their addresses range from 0 through 1023. Be sure not to confuse registers and memory locations. Registers are a very small amount of temporary storage within the CPU, and memory is a vast array of storage external to the CPU.

3.3. NEWS FLASH! Computers Don't Eat Quiche.

I hate to break it to you, but you have been hoaxed in previous courses. Computers don't really execute things like:

```
...
y = x + z;
...
```

You may well recognize this as a C statement that sums the values of variables `x` and `z` and stores the result in the variable `y`. First of all, all variables are stored in memory locations, and as we have just seen, memory locations are numbered. They do not have arbitrary symbolic names like `x`, `y`, or `z`. To begin to understand the way a computer actually performs computation, let us see how a *real macho computer* would execute a few example C statements.

3.3.1. Sequential Flow; A Single Assignment Statement

Let us consider the single C statement shown in Figure 3.2. This statement is commonly referred to as an *assignment* statement because it *assigns* the value computed on the right side to the variable shown on the left side of the '='.

```
y = x + z;
```

Figure 3.2. A C Addition and Assignment Statement.

Since all data must be stored in memory, we must first decide where to store the variables `x`, `y`, and `z`. Let us assume the following correspondence of variables to memory locations.

```
x : memory location 1000
y : memory location 1001
z : memory location 1002
```

Now, since all data must be present in the CPU before being operated upon, we must first ''move'' or ''copy'' the data from memory to registers. This is done with the following 2 instructions,

```
MOVE      1000,D0
MOVE      1002,D1
```

which `MOVE` the data from their respective memory locations to data registers D0 and D1. Note, the word ''move'' does **not** imply that the source of the move is altered in any way. ''Copy'' is probably a better word to use, but as you will soon see, the term ''move'' will prevail. `MOVE` is referred to as an **instruction**. For now, assume that all instructions operate on byte-sized units of data. Later we will see that there is actually a choice of sizes for most instructions. The objects on the right side are called **operands**. We call the left-hand operand the **source** operand, and the right-hand operand is called the **destination**. We also sometimes refer to the operand on the left as the *first* operand, and the one on the right as the *second* operand. The first of these two instructions moves the contents of memory location 1000 into CPU data register D0. The second one moves the contents of memory location 1002 into CPU data register D1. Now that the data are present in the CPU, we can operate on these *copies* of `x` and `z`. The following instruction will perform the required addition.

```
ADD       D0,D1
```

This instruction sums the contents of registers D0 and D1, and leaves the resulting sum in register D1. In this case, both the first operand and the second operand are called *source* operands, but the second operand is also called the *destination*. Now, although the desired result has been computed, we are not finished with the C statement until the result is stored back to the memory location reserved for `y`. This is accomplished with the following instruction.

```
MOVE      D1,1001
```

As should be obvious at this point, this instruction moves the value stored in CPU data register D1 out to memory location 1001. Recall that we assumed `y` was allocated the byte of storage at address 1001. The program segment illustrating the entire statement shown in Figure 3.2 appears in Figure 3.3.

```
MOVE      1000,D0
MOVE      1002,D1
ADD       D0,D1
MOVE      D1,1001
```

Figure 3.3. M68000 Code Corresponding to Figure 3.2.

In addition to being an example of how a single C assignment statement would actually be executed on a computer, this example also illustrates what is known as the ‘‘Von Neumann Bottleneck’’. This bottleneck refers to the fact that data has to constantly be shuffled back and forth along the Bus shown in Figure 3.1. This bottleneck is the source of the primary limitation to performance achievable on 99% of computers today. Advanced research in Computer Architecture has been hard at work trying to eliminate this bottleneck for the past 15 or 20 years, but still today there has been no *widely accepted* alternative to the basic structure shown in Figure 3.1.

3.3.2. Alternation

The example in Section 3.3.1 illustrates how the simplest form of C statement can be executed on a computer. Obviously, we need to do more than assignments of values to memory locations. The second basic program structure we look at is **alternation**. Alternation simply refers to the situation in which a program reaches a point at which it will perform one of several alternatives. This is most easily illustrated by an example.

```
if (x < y) then
          z = y;
else
          z = x;
```

Figure 3.4. C If-Then-Else Statement.

Let us assume the same set of memory locations are reserved for `x`, `y`, and `z` as were assumed in Section 3.3.1. Here, we again first need to move the principal data values from memory to the CPU.

```
MOVE      1000,D0
MOVE      1001,D1
```

Now, to determine which half of the `if-then-else` statement is to be executed, we need to evaluate the conditional `(x < y)`. This is done with the M68000 `CMP` instruction as follows.

```
CMP       D1,D0
```

Be sure to refer to the M68000/CIE manual to understand precisely what the effect of this instruction is, but basically, the difference of D0 and D1 (i.e., D0−D1) is computed, but not stored in either D0 nor D1. Instead, the condition codes in the CCR are set as they would have been if the `SUB` instruction had been executed. For example, if the values in D0 and D1 had been equal before executing the `CMP`, the resulting difference would be zero, and as a result, the Z flag of the CCR would be set, however, neither D0 or D1 would themselves change. The importance of this effect comes from its coupling with the ‘‘Branch on Condition Code’’, or

`Bcc` instruction. For the `Bcc` instruction, selected bits of the CCR may be tested in a variety of ways to alter the selection of the next instruction to be executed. If it was desired to test to see if D0 and D1 were equal, we would use `BEQ`, because it bases its decision about what to do next on the value stored in the Z flag of the CCR. However, for the C statement of Figure 3.4, we need to test to see if D0 is less than D1. We do this with the following instruction:

```
        BLT         1100
```

A simple way to think about this instruction is that if D0 < D1, then we ‘‘go to’’ the instruction at the line labeled ‘1100’. Otherwise execute the instruction immediately following the `BLT`. Therefore, the instruction(s) immediately following the `BLT` should implement the `else` clause of Figure 3.4. This done with the following:

```
        MOVE        D0,1002
```

This moves the value of `x` (temporarily being stored in D0) into the memory location allocated to `z`. Obviously, to execute the `if` clause of Figure 3.4, we need to do the following:

```
1100    MOVE        D1,1002
```

This moves the value of `y` (temporarily being stored in D1) into the memory location allocated to `z`. Furthermore, notice that we label this line with ‘1100’. This was the ‘‘target’’ of the `BLT` instruction shown above.

Now, if we try to put this sequence of statements together into a program segment as we did in Figure 3.3 for our example of Section 3.3.1, we get the **incorrect** version shown in Figure 3.5.

```
                MOVE        1000,D0
                MOVE        1001,D1
                CMP         D1,D0
                BLT         1100
                MOVE        D0,1002
1100            MOVE        D1,1002
```

Figure 3.5. Incorrect M68000 Code Corresponding to Figure 3.4.

The reason this is incorrect is illustrated by the fact that no matter what `x` and y are, `z` will always end up with the value from `y`. This is graphically illustrated by the flowchart of Figure 3.6 (A).

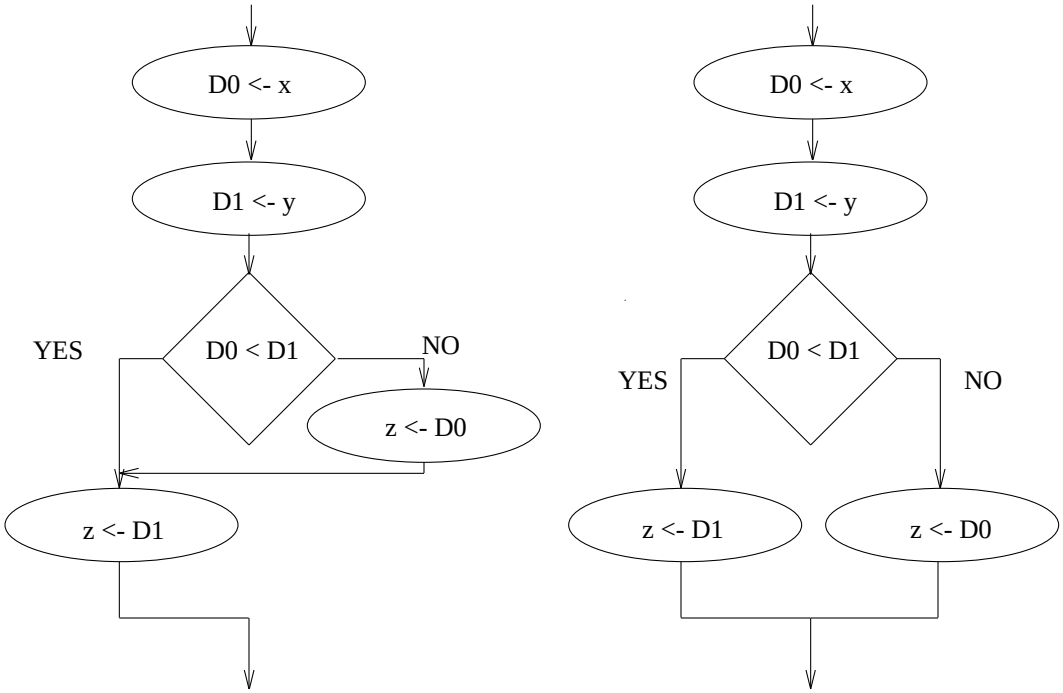

Figure 3.6. Flowcharts for Example of Figure 3.4.
(A) - Incorrect Flow; (B) - Correct Flow.

Note that the assignment of D0 to `z` will be overwritten by the assignment of D1 to `z`. What is desired is the control flow shown in Figure 3.6 (B). Referring back to our program segment of Figure 3.5, what is needed is a way to ‘‘jump’’ over the instruction on the line labeled ‘1100’ to the line following line ‘1100’. Let’s call this line ‘1200’. Furthermore, this ‘‘jump’’ must be **unconditional**. I.e., no condition need be tested before executing the ‘‘jump’’. The result is shown in Figure 3.7.

```
                MOVE      1000,D0
                MOVE      1001,D1
                CMP       D1,D0
                BLT       1100
                MOVE      D0,1002
                JMP       1200
1100            MOVE      D1,1002
1200            ...... program continues here
```

Figure 3.7. Correct M68000 Code Corresponding to Figure 3.4.

Believe it or not, there is really only one more control flow construct needed to provide the same capability as most high-level languages such as C. This is the *looping* construct.

3.3.3. Iterating or Looping

Let us now consider an example of a very common program control-flow construct — the loop.

```
x = 0;
for (i=1; i<=20; i++)
          x = x + i;
```

Figure 3.8. C For-loop Statement.

Let us assume the following correspondence of the variable `x` to a memory location.

```
x : memory location 1000
```

Because `i` is only used as a control variable within the context of the loop, there is no need to assign a memory location to it. For the code segment which implements this loop, we will let data register D1 hold the value of `i`.

First, we assign 0 (zero) to `x` as required in the first line of Figure 3.8.

```
          MOVE      #0,1000
```

Next, we will make a copy of `x` in a data register.

```
          MOVE      1000,D0
```

Now, we initialize data register D1, which is being used to hold the loop index `i`.

```
          MOVE      #1,D1
```

In addition to the loop index itself, we need to have a value to compare with, to determine whether the loop is finished or not. For this loop limit value we will use data register D2.

```
          MOVE      #20,D2
```

We now begin the main body of the loop, and let's simply label this line '1100', so we can refer to it later. This line will simply add the loop index to the accumulated value of `x` being held in D0.

```
1100      ADD       D1,D0
```

The line above comprises *all* of the code needed to execute the main body of the loop. What remains is the ''overhead'' code to manipulate and test the loop index, and branch back to '1100' if necessary.

```
          ADD       #1,D1
          CMP       D2,D1
          BLE       1100
```

the first line increments the loop counter, the second line compares the loop counter to the loop limit, and the last line will branch back to '1100' if the value stored in D1 is less than or equal to that stored in D2. Note, if we had used `BLT`, then we would fail to execute the loop for `i`

`= 20`. Finally, we have to store the resulting value of `x` back to memory location *$*1000 upon termination from the loop.

```
                    MOVE        D0,1000
```

Figure 3.9 shows the complete M68000 implementation of the C `for`-loop shown in Figure 3.8.

```
                    MOVE        #0,1000
                    MOVE        1000,D0
                    MOVE        #1,D1
                    MOVE        #20,D2
1100                ADD         D1,D0
                    ADD         #1,D1
                    CMP         D2,D1
                    BLE         1100
                    MOVE        D0,1000
```

Figure 3.9. M68000 Code Corresponding to Figure 3.8.

3.4. Control, Operands, and Operations

With the three brief examples so far, we have covered quite a few concepts. This section will briefly summarize the main points covered so far.

The first general idea is that of **control flow**. We have illustrated M68000 implementations of three fundamental control flow constructs; *sequencing*, *alternation*, and *iteration.* These are depicted graphically in Figure 3.10.

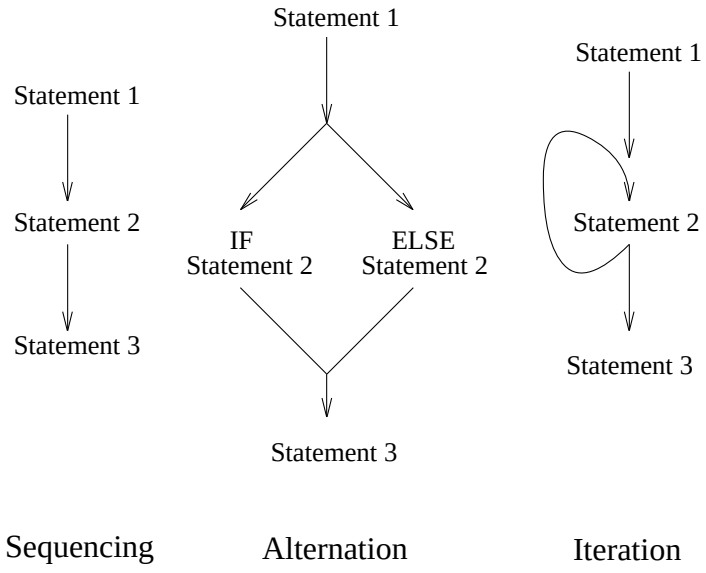

Figure 3.10. Three Primary Control Flow Structures.

The second general concept is that of an **addressing mode**. We have seen three addressing modes so far. They are:

1. **Data Register Direct.** The operand is in a register.

   e.g., `MOVE  D0,D1  ;`*both operands*

2. **Immediate Data.** The operand appears in the instruction.

   e.g., `MOVE  #1,D1  ;`*left operand only*

3. **Absolute.** The memory address of the operand appears in the instruction.

   e.g., `MOVE  1000,2000 ;`*both operands*

We have also already encountered a small subset of the M68000 instruction set. So far these instructions include:

- `MOVE` — Move from source to destination
- `ADD` — Add source to destination
- `CMP` — Compare source and destination
- `BLT` — Branch if Less Than to target address
- `BLE` — Branch if Less than or Equal to target address
- `JMP` — Jump unconditionally to target address

Finally, we have seen how we refer to one instruction from within another instruction. So far, we have arbitrarily assigned ‘‘address labels’’ to lines that were the target of a branch or jump. For example, we saw:

```
                .....
1100        ADD         D1,D0
                .....
                BLE         1100
```

which indicated that if the ‘LE’ condition held, then a branch was to occur to the line labeled ‘1100’. Later, we will see that address labels like ‘1100’ cannot be selected at random. However, if we simply think of them as symbolic labels, we will be fine for now.

3.5. Some Notation

When writing programs at the level we are, it is necessary to make very clear what base of arithmetic we are using. Therefore the following conventions will be observed. Normally, if we simply write a number, such as ‘9472’, it will be assumed to be a decimal quantity. If we precede the number with the ‘*$*’ symbol, such as ‘*$*2500’, it implies the number is hexadecimal. Finally, if we precede a number with the ‘%’ sign, such as ‘%0010010100000000’, it implies the number is binary. Therefore, the following all represent the same value.

9472 = *$*2500 = %0010010100000000

3.6. A More Complete View of Memory

Until now, we have only said that memory is addressed by associating a number with each location. In fact, there are several ways to view the organization of memory. For our purposes, we can think of memory as being organized in three different ways.

As shown in Figure 3.11, memory can be viewed as consisting of **bytes, words,** or **longwords**. A byte is 8-bits long, a word is 16 bits, and a longword is 32 bits.

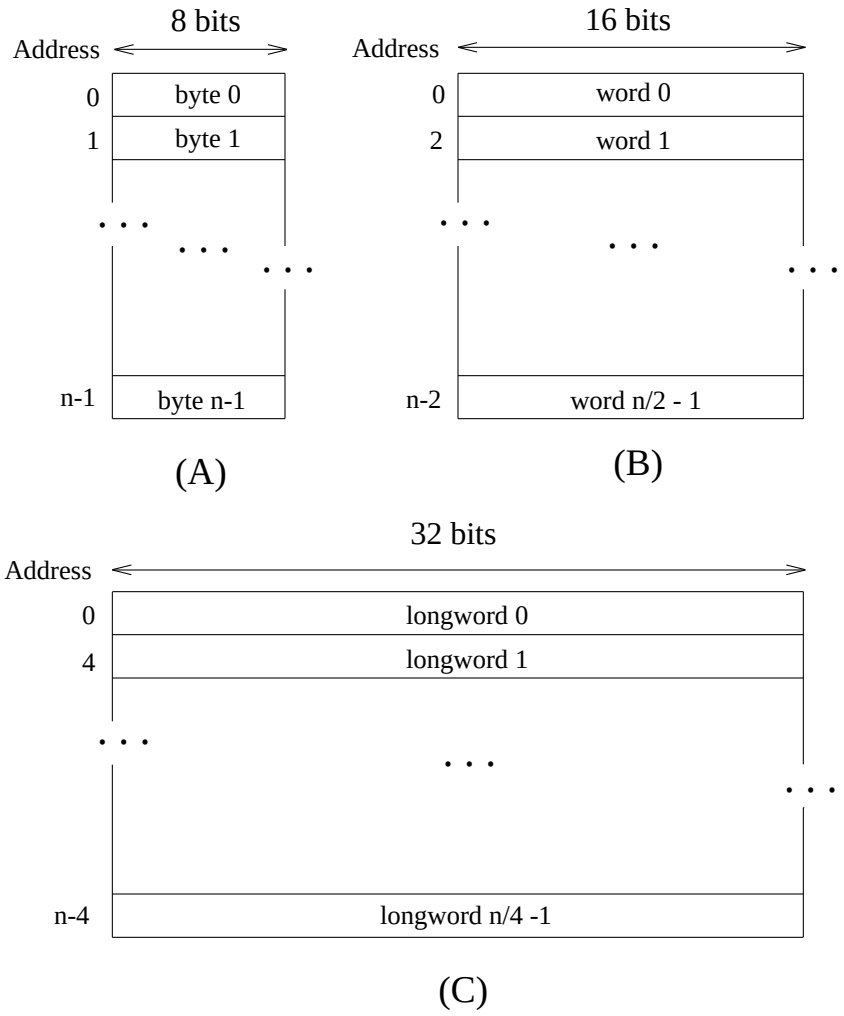

Figure 3.11. Three Separate Views of Memory.

Figure 3.11(A) shows a memory organization of *n* bytes. A byte is the smallest addressable unit of memory, therefore, the addresses of bytes are simply consecutive integers. Figure 3.11(B) shows the same *quantity* of memory as (A), but organized as a sequence of 16-bit words. Since words are each two bytes long, the total number of words is half that of the number of bytes (i.e., n/2 words). Furthermore, each address is a multiple of 2 — i.e. 0, 2, 4, etc. Figure 3.11(C) again shows the same quantity of information as (A) and (B), but organized as 32-bit long-words. Therefore, there are only n/4 longwords, and each has an address which is a multiple of 4 — 0, 4, 8, etc.

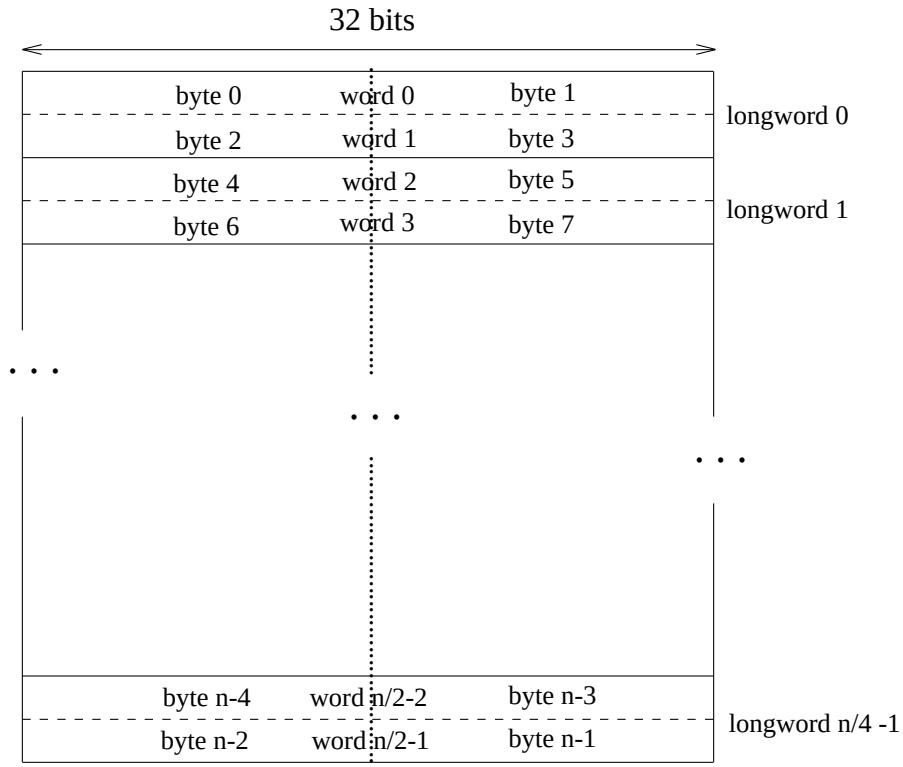

Figure 3.12. One Unified View of Memory Organization.

Figure 3.12 is a convenient way to think of memory all in one organization containing the others. One interesting thing to note is that byte 0 is shown in the most significant portion of word 0, and byte 2 is in the most significant portion of word 2. Byte 0 is also the most significant portion of longword 0. For historical reasons, this byte ordering is referred to as *Big Endian* ordering. Some other CPUs (e.g., Intel processors) use the opposite byte ordering. This inconsistency can make use of systems containing both kinds of processors difficult.

Now that we have clearly defined what the quantities byte, word and longword are, we more precisely state how instructions operate on these different sized units. Most instructions permit operations on operands of different sizes. This is denoted by appending a one character suffix to the instruction operation code. The three suffixes are **.B, .W,** and **.L**, for byte, word and longword respectively. For example, to move the byte from location *$*10023 to the low order byte of data register D0, the following instruction should be used.

```
MOVE.B     $10023,D0
```

Note that only the low order byte of D0 will be changed. The higher-order 24 bits will remain unchanged. Also, if the condition codes would be altered as the result of an instruction, that result is only based on the size of operand specified.

For example, if the following intitial condition is satisfied,

```
D0 = $12345678
D1 = $98765678
```

and the following instructions were executed,

```
CMP.W     D0,D1
BEQ       TARGET
```

then the `BEQ` would take the branch to `TARGET`. In other words, since the low-order 16 bits of D0 and D1 are equal, the Z bit of the CCR will be set, and the test of the Z bit in the `BEQ` instruction will succeed. The fact that the high-order 16 bits of D0 and D1 are not equal does not affect the `CMP` instruction.

3.7. More Instructions

3.7.1. Bcc

Previously, we saw two examples of the **conditional branch**. These were the `BLT`, and the `BLE` instructions. These two instructions are examples of the general `Bcc` instruction. The 'LT' and 'LE' portions of these instructions are termed *condition specifiers*. In all, the `Bcc` instruction permits 14 different condition specifiers. They are listed in the M68000 Programmer's Manual. Each one tests a different combination of the bits of the CCR — Condition Code Register. For example, the `BEQ` instruction tests the Z bit of the CCR, and if it is a '1', then the branch will be taken, and the next instruction to be executed is the one identified by the label in the operand field of the `BEQ` instruction. If the Z bit is a '0', then the branch will not be taken, and the next instruction to be executed will be the one immediately following the `BEQ` instruction in a sequential manner. Often, we say that a bit of the CCR is ''set'', if it is equal to '1', and we say that a bit is ''cleared'' (or simply ''clear'') if it is equal to '0'. Thus, we might say that ''the `BEQ` will succeed if the Z bit is set''.

While instructions like the `BEQ` are simple to understand in terms of the CCR bits themselves, it is sometimes easier to think of some of the `Bcc` instructions in terms of their mneumonic meanings. For example, the `BLT` is easiest to think of in terms of whether the operands of a `CMP` instruction (immediately preceding the `Bcc`) obeyed the '$<$' relation or not.

Finally, while it may very often be the case that a `Bcc` instruction is directly related to the way the CCR bits were set by the instruction immediately preceding it, this is not necessarily always the case. Not all instructions modify all CCR bits, and a particular usage of a `Bcc` instruction may be obeying the way the CCR bits were set or cleared during an instruction execution several instructions earlier. In other words, the `Bcc` instruction looks at the *current state* of the CCR. How the bits of the CCR became set the way they are at any point in time may be a function of instructions at any time in the past.

3.7.2. SUB

The `SUB` instruction is basically the same as the `ADD` instruction, except it subtracts. It is important to remember that the operation subtracts the source from the destination and leaves the result in the destination operand. It operates on byte, word, and longword entities.

3.7.3. JSR/RTS

A very useful mechanism for writing a segment of a program once and then utilizing it in several other places within a program is the **subroutine**. The basic idea is the same as that used in a high level language. Two instructions are available to provide a convenient way to make subroutine calls — `JSR` - Jump to SubRoutine, and `RTS` - ReTurn from Subroutine.

3.8. Logical Operators: AND, OR, NOT

One aspect of programming at this level which is foreign to high-level language programming (C is a notable exception) is that of setting, clearing, and testing single bits of a variable. This has particular usefulness when manipulating the status and control bits of a device interface register, such as for a terminal keyboard or screen. We will talk more about device registers later, but for now, we describe the operation commonly called **masking** in terms of three **logical operators**. They are called ‘‘logical’’ because they perform ‘‘logic’’ operations. First we summarize the effect of the three operators — **NOT, AND, and OR**. This summary is shown in Figure 3.13.

A	NOT A
0	1
1	0

(A)

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

(B)

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

(C)

Figure 3.13. Summary of Logical Operators NOT, AND, and OR.

The NOT operator is further described as a **unary** operator because it only has one operand, and the AND and OR operators are *binary* because they each have two operands. Figure 3.13(A) shows that the NOT operator simply changes 0’s to 1’s and 1’s to 0’s. 3.13(B) shows that the AND operator will produce a 1 only when its two input operands are 1’s. 3.13(C) shows that the OR operator will produce a 1 when either, or both, of its two input operands are 1.

3.8.1. AND, OR, and NOT in Assembly Language

The `AND`, `OR`, and `NOT` instructions of the M68000 are based directly on the three logical operators described in Figure 3.13. However, they operate simultaneously on all the bits of a byte, word, or longword. For example, if the following initial condition was satisfied,

```
D0 = %10011101
```

and the following instruction was executed,

```
NOT.B      D0
```

then immediately following execution, D0 would contain `%01100010` because of the bit by

bit NOT of D0 as defined in Figure 3.13. As a second example, if the following intitial conditions were satisfied,

```
D0 = %10011101
D1 = %01010110
```

and the following instruction was executed,

```
AND.B     D0,D1
```

then immediately following execution, D1 would contain `%00010100` because of the bit by bit AND of D0 and D1 as defined in Figure 3.13. As a final example, if the following initial conditions were satisfied,

```
D0 = %10011101
D1 = %01010110
```

and the following instruction was executed,

```
OR.B      D0,D1
```

then immediately following execution, D1 would contain `%11011111` because of the bit by bit OR of D0 and D1 as defined in Figure 3.13.

The preceding examples of the `AND`, `OR`, and `NOT` instructions demonstrate the basic functionality of these instructions. The usefulness of these instructions comes about when we need to test a single bit of a memory location. Consider the hypothetical example of needing to perform two different tasks based on whether bit position 4 of the byte at memory location $2000 is set or cleared.

```
          MOVE.B    $2000,D0
          AND.B     #%00010000,D0
          BNE       1150
1100      ...
...       ...
1150      ...
```

In this example, we assume the two locations, 1100 and 1150, are the locations of the "two different tasks" respectively. The "mask" value of `%00010000` only has bit position 4 set, and by AND'ing this value with the copy of the value stored in memory location `$2000`, all bits of the low order byte of D0 will be cleared except for bit position 4. Bit position 4 will be exactly the same as the corresponding bit in D0. Now, if bit 4 of D0 were a 0, then the *entire* low order byte of D0 will result in being `%00000000`, and therefore, the Z bit of the CCR will be set (to 1), and the `BNE` instruction will fail to branch to 1150, and execution will continue at 1100. If bit 4 of D0 had been a 1, then D0 will result in being `%00010000` — i.e., non-zero. In this case, the Z bit of the CCR will be cleared (i.e., set to 0), and the `BNE` instruction will succeed and execution will continue at 1150.

3.8.2. AND, OR and NOT (Masking) in C

The AND, OR, and NOT operators of C are also based on the three logical operators described in Figure 3.13. Like the 68000 instructions, they too operate simultaneously on all the bits of a byte, word, or longword. However, the data types in C have different names as we will

see. In this course, we will accomplish most masking and device interfacing in C. We now consider the examples of the previous section, but in hexadecimal and in C.
For example, if the following declaration was made,

```
unsigned char a = 0x9d; /* 10011101 */
```

and the following C statement was executed,

```
a = !a;
```

then immediately following execution, `a` would contain `$62` or `%01100010` because of the bit by bit NOT of `a` as defined in Figure 3.13. As a second example, if the following declarations were made,

```
unsigned char a = 0x90 /* 10011101 */
unsigned char b = 0x56 /* 01010110 */
```

and the following C statement was executed,

```
b = a & b;
```

then immediately following execution, `b` would contain `$DF` or `%00010100` because of the bit by bit AND of `a` and `b` as defined in Figure 3.13. As a final example, if the following declarations were made,

```
unsigned char a = 0x90 /* 10011101 */
unsigned char b = 0x56 /* 01010110 */
```

and the following C statement was executed,

```
b = a | b;
```

then immediately following execution, `b` would contain `$DF` or `%11011111` because of the bit by bit OR of `a` and `b` as defined in Figure 3.13

The preceding examples of the C `AND`, `OR`, and `NOT` operators (&, |, and !) demonstrate the basic functionality of these operators. Their actual usefulness comes about when we need to test a single bit of a memory location or device register. Consider the hypothetical example of needing to perform two different tasks based on whether bit position 4 of the byte at memory location $2000 is set or cleared.

```
unsigned char *p, mask, test; (1)
p = 0x2000;                   (2)
mask = 0x10;                  (3)
test = *p & mask;             (4)
if (test == 0)                (5)
          /* then clause */   (6)
else                          (7)
          /* else clause */   (8)
```

In this example, we assume that the `then` and `else` clauses of the `if` statement on line 5 contain the "two different tasks" respectively. The "mask" value of `%00010000` only has bit position 4 set, and by AND'ing this value with the value stored in memory location `$2000` in line 4, all bits of `test` will be cleared except for bit position 4. Bit position 4 will be exactly the same as the corresponding bit in `$2000`. Now, if bit 4 of `$2000` were a 0, then the *entire*

byte of `test` will result in being `%00000000`, and therefore, the `if` conditional in line 5 will be TRUE and the `then` clause will be executed. If bit 4 of `$2000` had been a 1, then `test` would result in being `%00010000` — i.e., non-zero. In this case, the `if` conditional in line 5 will evaluate to FALSE, and the `else` clause will be executed.

3.9. Arrays and Address Registers

Consider, for example, summing the elements of a 100-byte array as shown in Figure 3.14. This example assumes that the array `ARR` is stored in the 100 bytes of memory beginning at address `$1000`. Obviously, this solution has at least one weakness — carpal tunnel syndrome [2] for the programmer! Clearly, what is needed is a way to embed a single `ADD.B` instruction in a loop to process one element of the array at a time. However, as yet, we do not have an addressing mode that will allow us to step through the array elements one at a time as the loop index increases. Our solution is the use of another addressing mode in the M68000.

```
MOVE.B    #0,D0
ADD.B     $1000,D0
ADD.B     $1001,D0
ADD.B     $1002,D0
ADD.B     $1003,D0
...
ADD.B     $1062,D0
ADD.B     $1063,D0
END
```

Figure 3.14. Summing the Elements of a 100-byte Array.
(The Hard Way)

Address Registers and Indirect Addressing

Recall, our current programmers model contains memory, a memory bus, and a CPU containing 8 data registers, a program counter (PC), and a condition code register (CCR) as was shown in Figure 3.1. We now add to this model a set of **Address Registers** as shown in Figure 3.15.

---

2. An arthritis-like condition caused by repetitive motion (like typing).

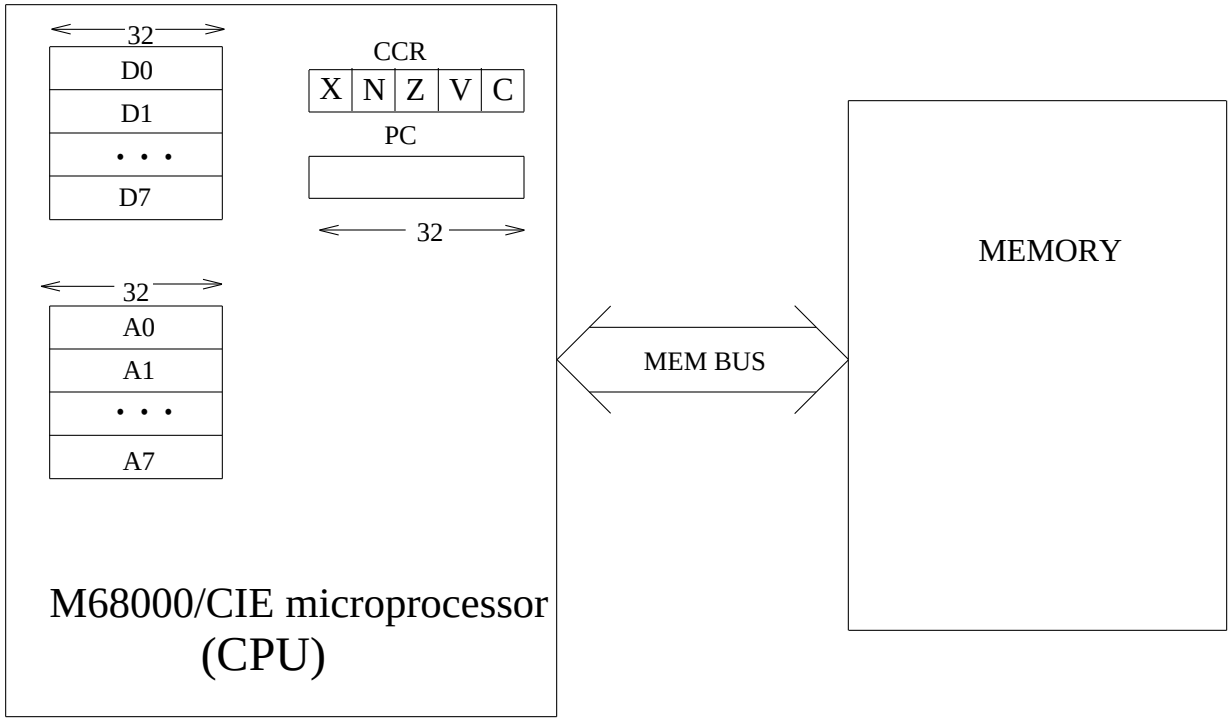

Figure 3.15. First Revision of Programmer's Model.

Address registers are, in many ways, similar to data registers. There are 8 of them, named `A0` through `A7`, and each is 32 bits long. However, there are differences in the instructions which may manipulate them, and also in the types of addressing modes which may employ them. While data registers are intended to contain data values, address registers are thought of as containing values which are themselves addresses. Since addresses are all 32-bit values, there are no byte or word operations defined for address registers. All operations which modify the value of an address must operate on a longword quantity.

Before returning to the array example, a simpler example will be used to illustrate the basic use of address registers.

```
MOVE.L    #$1000,A0       <=>
MOVE.B    #0,(A0)         <=>  MOVE.B    #0,$1000
```

The code segments on the left and the right accomplish exactly the same thing. They assign the value 0 (zero) to the byte at memory location $1000. The right side uses the immediate addressing mode for the source operand, and absolute addressing for the destination operand. The code segment on the left first moves the value $1000 into address register `A0` by using immediate addressing for the source, and register direct addressing for the destination. In the second instruction on the left side, the immediate operand 0 (zero) is moved to a destination specified by ‘‘`(A0)`’’. This addressing mode is referred to as **Address Register Indirect**. The effect of this mode is summarized by saying that ‘‘the operand is found in the memory location whose address is found in the address register — `A0`’’. The advantage of this mode is only seen when we point out that arithmetic may be performed at program *run-time* on address registers.

We now return to our array example. In Figure 3.16 we see a program containing a loop which terminates conditioned on the value of address register `A0`.

```
                MOVE.B    #0,D0              (1)
                MOVE.L    #$1000,A0          (2)
$1100           ADD.B     (A0),D0            (3)
                ADD.L     #1,A0              (4)
                CMP.L     #$1064,A0          (5)
                BLT       $1100              (6)
                END                          (7)
```

Figure 3.16. Summing the Elements of a 100-byte Array.
(The Easy Way)

The first line of this program is identical to that of Figure 3.14. In line 2, `A0` is initialized with the beginning address of the array `ARR`. Then, through each iteration of the loop, `A0` is incremented in line 4 and tested to see if the end of the loop has been reached in lines 5 and 6. In line 5, the address register indirect mode is being used to access successive elements of the array on successive iterations of the loop. Each time through the loop, `A0` will contain the address of some element of the array, and the instruction in line 3 says ‘‘add the byte in the memory location whose address is contained in address register `A0` to data register `D0`.’’ Note the use of ‘‘`$1064`’’ in line 5. This is how we specify the address of the location immediately following the end of the array to be used as a loop termination test.

Chapter 9
Information Coding II

9.1. Hand Coding of 68000 Instructions

Computers can only store information in binary format. This goes for both the data and the instructions. In order for the CPU to understand what the instruction is telling it to do, there is a unique pattern for each and every instruction. These patterns are different for different CPUs. For example, the encoding for the Intel 8088 chip is different from the encoding for the Motorola 68000 chip.

You have seen some of the encoding before. It appears in your list (.lst) file generated by the as68 assembler. The M68000 chip uses word length instructions or multiples thereof.

This chapter covers the encoding of the M68000 instructions introduced in this course. This is in no way complete, but is sufficient for the amount of material covered in the course.

ADD

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	0	1	Register	OpMode	EA.Mode	EA.Register

SUB

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	0	1	Register	OpMode	EA.Mode	EA.Register

The *Register*, *EA.Mode* and *EA.Register* fields are common for many instructions and are detailed after the `NOT` instruction. The *OpMode* field is different for the different instructions, and hence is presented after each group of instructions for which the *OpMode* is the same. EA refers to the effective address. I.e., the ultimate address of the operand to be accessed or modified.

OpMode

Byte	**Word**	**Long**	*Operation* *
000	001	010	<ea> Op <Dn> → <Dn>
100	101	110	<Dn> Op <ea> → <ea>
	011	111	<ea> Op <An> → <An>

* "Op" stands for whatever operation is being done in the instruction being assembled.

AND

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	0	0	Register	OpMode	EA.Mode	EA.Register

OR

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	0	0	Register	OpMode	EA.Mode	EA.Register

OpMode

Byte	**Word**	**Long**	*Operation* *
000	001	010	<ea> Op <Dn> → <Dn>
100	101	110	<Dn> Op <ea> → <ea>

* "Op" stands for whatever operation is being done in the instruction being assembled.

CMP

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	1	1	Register	OpMode	EA.Mode	EA.Register

OpMode

Byte	**Word**	**Long**	*Operation*
000	001	010	<ea> - <Dn> → CCR
	011	111	<ea> - <An> → CCR

MOVE

15	14	13 12	11 10 9	8 7 6	5 4 3	2 1 0
			Destination		Source	
0	0	Size	EA.Register	EA.Mode	EA.Mode	EA.Register

Size

Byte	**Word**	**Long**
01	11	10

NOT

15	14	13	12	11	10	9	8	7 6	5 4 3	2 1 0
0	1	0	0	0	1	1	0	Size	EA.Mode	EA.Register

Size

Byte	**Word**	**Long**
00	01	10

Register

Number of the address or data register which will be involved in the register operand.

EA.Mode and *EA.Register*

Addressing Mode		**EA.Mode**	**EA.Register**
Dn	(Data Register Direct)	000	Register number
An	(Address Register Direct)	001	Register number
(An)	(Register Indirect)	010	Register number
#<data>	(Immediate) (only for source)	111	100
xxx.W	(Absolute with address of Word size)	111	000
xxx.L	(Absolute with address of Long size)	111	001

Note: In case of immediate and absolute addressing modes, the word(s) following the first word of the instruction contain the data for that instruction. These additional word(s) are referred to as ‘‘Extension Word(s)’’. For example, if the source operand is of immediate type, the first extension word(s) of the instruction will contain the immediate data, and the number of such words will depend upon the size of the operation. If an operand is of absolute type, the number of extension words data can be one or two, depending on the length of the address (this is independent of the operation size). If both the source and the destination need extension words, the source information will be placed before the destination information.

JMP

15	14	13	12	11	10	9	8	7	6	5 4 3	2 1 0
0	1	0	0	1	1	1	0	1	1	EA.Mode	EA.Register

JSR

15	14	13	12	11	10	9	8	7	6	5 4 3	2 1 0
0	1	0	0	1	1	1	0	1	0	EA.Mode	EA.Register

EA.Mode and *EA.Register*

Addressing Mode		**EA.Mode**	**EA.Register**
xxx.W	(Absolute with address of Word size)	111	000

xxx.L (Absolute with address of Long size) 111 001

For JMP and JSR, put the memory location to which the JMP or JSR would take place in the extension word(s). Use 1 word if the first mode is chosen, 2 if the second is chosen.

Bcc

15	14	13	12	11 10 9 8	7 6 5 4 3 2 1 0
0	1	1	0	Condition	8-bit displacement
16-bit displacement if 8-bit = $00					
32-bit displacement if 8-bit = $FF					

Condition

Use the code corresponding to the instruction as given in the M68000/CIE Programmer's Manual (2nd edition) Instruction Set Reference.

RTE

$4E73

RTS

$4E75

RTE and RTS codes are supplied in hexadecimal format for sake of brevity. Notice that these instructions do not have any operands, and hence have a unique code with no variable fields.

9.2. Examples

Here, a section of a code will be encoded, as given below.

```
Address   Code              Label    Instruction
                                     ORG       $1000
$1000     D154              LOOP     ADD.W     D0,(A4)
$1002     67FC                       BEQ       LOOP
$1004     21FC000ADE124000           MOVE.L    #ADE12,$4000
```

For the `ADD` instruction, the *Register* field is 0 (as the register operand is D0) and the *OpMode* field is 101 (register operand is a data register, and is the source and the instruction is a word operation). The *EA.Mode* is 010 (A4 register indirect) and the *EA.Register* field is 4 (the register is A4). Hence, the first instruction will be coded as $D154.

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	0	1	Register	OpMode	EA.Mode	EA.Register
1	1	0	1	0 0 0	1 0 1	0 1 0	1 0 0

For the `BEQ` instruction, the condition is EQ, which from the Instruction Set Reference, is determined to have a code of 0111. The 8-bit displacement is now determined. This

displacement is a 2's complement representation of the number that will be added to the PC if the condition is true. As the branch is to a location very close to the instruction, only 8 bits will be required to represent the displacement. If 16 bits were required, the 8-bit displacement would be set to $00 and the second word would contain the displacement. If 32 bits were required, the 8-bit field would be set to $FF and the following longword would contain the displacement.

The target address of the `BEQ` is calculated by adding the displacement to the value of the PC immediately after the displacement is fetched. I.e. the displacement is computed as the difference between the target address and the address of the instruction immediately following the last word of the `BEQ` instruction. Hence, -4 needs to be added to the PC in order to execute the `ADD` instruction if the condition is true (two bytes for the `ADD` instruction and two bytes for the `BEQ` instruction). Therefore the displacement is $FC. The complete coding for `BEQ` instruction is $67FC.

15	14	13	12	11 10 9 8	7 6 5 4 3 2 1 0
0	1	1	0	Condition	8-bit displacement
0	1	1	0	0 1 1 1	1 1 1 1 1 1 0 0

For the `MOVE` instruction, the *Size* field is 10 (operation size is long). The destination is an absolute address which can be represented in a word size, so the *EA.Register* is 000 and *EA.Mode* is 111. The source is immediate data, so the *EA.Register* is 100 and the *EA.Mode* is 111. There are 2 extension words to provide the immediate data and 1 extension word to denote the memory location being used in the destination. Therefore the complete coding for this instruction is $21FC000ADE124000.

15	14	13 12	11 10 9	8 7 6	5 4 3	2 1 0
			Destination		Source	
0	0	Size	EA.Register	EA.Mode	EA.Mode	EA.Register
0	0	1 0	0 0 0	1 1 1	1 1 1	1 0 0
Immediate data (high word) = $000A						
Immediate data (low word) = $DE12						
Address location = $4000						

- THIS PAGE INTENTIONALLY BLANK -