Address Modes

CIS*2030
Fall 2024

Neatly print your name, student identification number, and lab information below:

Name:

ID:

Section:

Overview

The efficacy of any ISA is not solely defined by the types of instructions it supports, but also by the types of address modes that are available. Address modes are all of the different ways of specifying the effective address (i.e., location) of the operand(s) that an instruction is to use to carry out its operation. In this lab, you will gain experience working with some of the 68000's address modes. It is worth noting that these address modes are not unique to the 68000 but are commonly available in other ISAs.

Objectives

Upon completion of this lab, you will:

- Understand how the fundamental address modes (e.g., direct, absolute, and immediate addressing) affect both the *size* and *runtime* of instructions,
- Understand how *address registers* are used to implement pointers,
- Understand the many variations of the *address register indirect address mode*, and
- Understand how PC-relative addressing enables *position-independent code* to be written.

Preparation

Prior to attending your lab section, you must complete the following reading assignments from your textbook, and understand how each address mode (or instruction) works to the degree that you can use them correctly and confidently in an actual program:

- Section 2.2.2 (Address Registers)

- Instructions that operate on address registers: `LEA` (pages 80 and 312), `MOVEA` (pages 79 and 318), `ADDA` (page 269) and `SUBA` (page 354).
- Sections 2.4-2.4.11 (Data Register Direct, Absolute Short, Absolute Long, Register Indirect, Post-Increment Register Indirect, Pre-decrement Register Indirect, Register Indirect with Offset, Register Indirect with Index and Offset, PC-Relative with Offset, and PC-Relative with Index and Offset)

You should also review your lecture notes related to the above topics. <u>You will not have time to perform these reading assignments during your lab session and complete the lab questions</u>. Therefore, it is your responsibility to come prepared!

Evaluation

This lab has 5 questions all of which contain multiple parts. Questions *with* an asterisk '*' beside them can be completed prior to the lab. However, all other questions must be performed during your scheduled lab session.

Question	**1***	**2***	**3**	**4**	**5**	**Total**
Max points	6	3	13	4	5	31
Your Points						

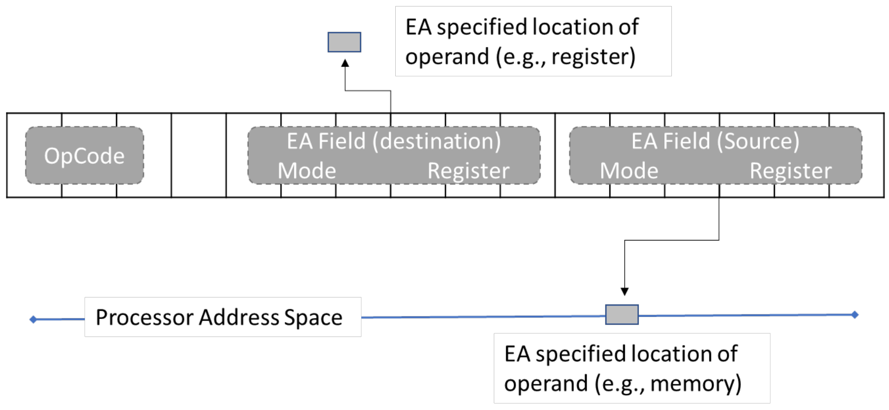
All marking is performed in real time *during* the lab. Therefore, when you complete a question raise your hand to let the lab instructor know that you are prepared to be marked. Marks will be given based on the written answer you provide and/or your ability to orally answer related questions asked by the lab instructor. The key to obtaining full marks is to demonstrate comprehension.

Introduction

The following table summarizes the available address modes.

Address Mode	Mode	Register	Assembler Syntax
Data register direct	000	000-111	Dn
Address register direct	001	000-111	An
Address register indirect	010	000-111	(An)
Address register indirect with post-increment	011	000-111	(An)+
Address register indirect with pre-decrement	100	000-111	-(An)
Address register indirect with offset	101	000-111	d16(An)
Address register indirect with index	110	000-111	d8(An,Xn)
Absolute short	111	000	<…>.W
Absolute long	111	001	<…>.L
PC-relative with offset	111	010	d16(PC)
PC-relative with offset and immediate	111	011	d8(PC,Xn)
Immediate	111	100	#<…>

The instruction data sheets in Appendix B of your textbook identify which address modes are available to access the operand(s) required by an instruction. At assemble time, the source/destination address mode is encoded in the Effective-Address (EA) field of the instruction, as illustrated below. The mode and register bits for each address mode are indicated in the table above.


EA specified location of operand (e.g., register)
OpCode
EA Field (destination)
Mode
Register
EA Field (Source)
Mode
Register
Processor Address Space
EA specified location of operand (e.g., memory)

Question 1: Fundamental Address Modes – Instruction Length (*)

Download the sample program called **1.X68** from the course website and open it in Easy68K. Examine the source code and make sure that you understand what the program is doing.

In the table below, print the names of the address modes that are being used to specify the location of the source and destination operands. In each case, also include the binary values of the mode and register bits that make up the EA field.

Instruction	Source Operand	Destination Operand
`MOVE   DATA,D0`		
`MOVE   D0,DATA+8`		

Now, assemble the program, then write down the machine code for both instructions in the table below. Specify the operation word in the middle column, and any extension word(s) in the right-most column.

Instruction	Operation Word	Extension Word(s)
`MOVE   DATA,D0`		
`MOVE   D0,DATA+8`		

How many extension words does each instruction have? Why?

What do these extension words refer to? Be specific.

Now make the following change to the program:

```
DATA     EQU     $6000
```

Re-assemble the program, then write down the machine code for both instructions in the table below.

Instruction	Operation Word	Extension Word(s)
`MOVE   DATA,D0`		
`MOVE   D0,DATA+8`		

How many extension words does each instruction have? Why?

How many total bytes of memory does the current implementation (i.e., instructions and data) require? Is this more or less than the original program? Explain.

Question 2: Fundamental Address Modes – Execution Time (*)

Download the sample program called **2.X68** from the course website and open it in Easy68K. Examine the source code and make sure that you understand what the program is doing.

Clearly, all three program instructions use the same address mode for the source operands and the same address mode for the destination operands. Below, identify the name of the address modes, as well as the mode and register bits.

Source Address Mode:

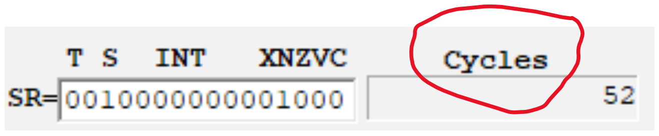
Destination Address Mode:

Assemble the program, then write down the machine code for all three instructions in the table below.

Instruction	Operation Word	Extension Word(s)
`MOVE.B   #‘J’,D0`		
`MOVE.W   #‘Ja’,D0`		
`MOVE.L   #‘Jays’,D0`		

How many extension words does each instruction have? Why?

The Easy68K simulator displays the *total* cycle time required for (all) previously executed instructions, as illustrated below. (In actual hardware, each 68000 instruction takes a specific number of clock cycles to execute. Instructions that require more clock cycles have longer execution times compared to instructions with fewer clock cycles.) By tracing through a program one instruction at a time, it is possible to determine the number of clock cycles for each instruction (by observing the current cycle time, executing the next instruction, and then subtracting the new cycle time from the previous cycle time).



Fill in the table below, by tracing through the program and determining the number of clock cycles required to execute each instruction:

Instruction	No. of Clock Cycles
`MOVE.B   #‘J’,D0`	
`MOVE.W   #‘Ja’,D0`	
`MOVE.L   #‘Jays’,D0`	

The previous table provides a sense of how the number of extension words associated with an instruction affects the number of clock cycles required to execute the instruction. In the 68000’s ISA, reading/writing a byte/word from/to memory takes 4 clock cycles, while reading or writing a long word takes 8 clock cycles. An instruction like `MOVE.L #’JAYS’,D0` consists of three memory-read cycles, as the instruction is three words long. The first memory-read cycle retrieves the operation word for the instruction. Then, the long word data is read from memory. This takes two more memory read cycles: one for each extension word. The total number of clock cycles required to read, decode and execute the instruction is 12. In general, instructions that use address modes that result in fewer (or no) extension words execute faster than instructions that use address modes that require one or more extension words. They also result in instructions that take up less memory. We will consider these other address modes in the remainder of this lab.

Question 3: Address Registers and Indirect Addressing Variants

Download the sample program called **3.X68** from the course website and open it in Easy68K. Examine the source code and make sure that you understand what the program is doing.

In the table below, print the names of the address modes that are being used to specify the location of the source and destination operands, along with the binary values of the mode and register bits.

Instruction	Source Operand	Destination Operand
`ADD.L (A0),D0`		
`MOVE.L  D0,(A1)`		

Clearly, both address registers (`A0` and `A1`) are being used as pointers. What do each of the pointers point at? Be specific.

What does the program do?

Assemble and run the program. Verify to yourself that the program does what you just said it does.

In the current implementation, a `LEA` instruction is being used to initialize both pointers (`A0` and `A1`). Show how the pointers could also have been initialized using the `MOVE` instruction along with an appropriate address mode. Assemble and run your code to verify that the changes you made are correct. Save your program in a file called **3a.X68**.

Modify the original program so that it now sums all (four) of the longwords in the array before writing the sum to the location after the end of the array. To do this, you will need to manually update the first pointer (`A0`) to point to the next item in the array as you traverse the array. You should do this using the `ADDA` instruction. Save your program in a file called **3b.X68**.

Modify the original program so that it sums all (four) of the longwords in the array before writing the sum to the end of the list. Again, you will need to update the first pointer (`A0`) to point to the next item in the array. However, this time, instead of manually updating the pointer using the `ADDA` instruction, use address register indirect with post-increment addressing to automatically update the pointer (thus saving an instruction!). Also, in this implementation, there is no need for the second pointer (`A1`), so it should be removed. Save your program in a file called **3c.X68**.

Modify the original program so that it sums all (four) of the longwords in the array before writing the sum to the end of the array. However, this time use address register indirect with pre-decrement addressing to automatically update the pointer. Since you are using pre-decrement rather than post-increment, you should use the second pointer (`A1`) to access the items in the array in the reverse order; that is, you should start with the last item in the array and move the pointer towards the first item at the start of the array. In this implementation, there is no need for the first pointer (`A0`), so it should be removed. Save your program in a file called **3d.X68**.

Modify the original program so that it sums all (four) of the longwords in the array before writing the sum to the end of the array. This time use address register indirect with offset to access the items in the array. (Remember, this address mode does not change the original address in the address register.) Use the first pointer (`A0`) to point to the start (i.e., base) of the array. Then access each array item by specifying its offset from the base of the array. The end of the array can also be specified as an offset from the base of the array. Hence, there is no need for the second pointer (`A1`), so remove it from the code. Save your program in a file called **3e.X68**.

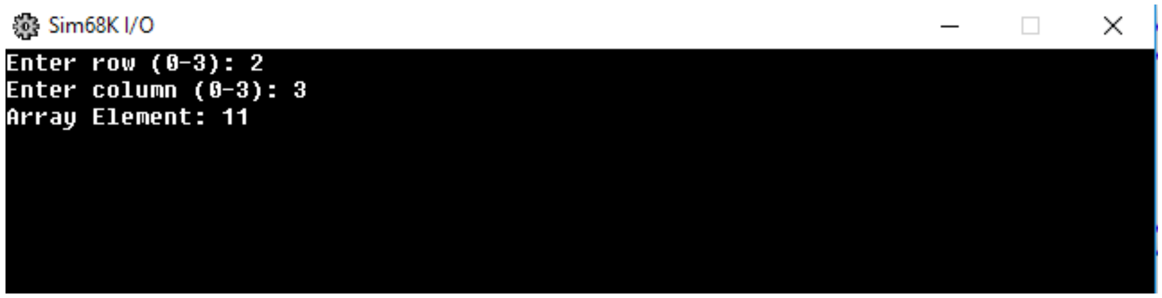
Question 4: Address Register Indirect with Index and Offset

Download the sample program called **4.X68** from the course website, then examine it carefully.

Notice that the program contains a statically allocated 4 x 4 array, called `MATRIX`, which contains the following 8-bit values:

$$\mathrm{MATRIX}_{ij} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

During execution, the program prompts the user to enter a row number *i* and a column number *j*. (Both *i* and *j* should be in the range of 0 to 3.) The program then displays `MATRIX[i][j]` – the decimal value of the byte at row *i* and column *j*, as shown below.



```
Enter row (0-3): 2
Enter column (0-3): 3
Array Element: 11
```

Assemble the program, then run the program multiple times, each time trying different values for *i* and *j*. Make sure you understand how the program functions, before proceeding.

Modify the program's data so that each matrix element is now a longword (32 bits) rather than a byte. Then, update the program's instructions so that the program works correctly with longwords. (Note: You do not have to change the input and output code – it will continue to function correctly.) But, you will need to modify the code on lines 46 through 50 to take into account that each array element is 4 bytes rather than a single byte. Save the new program under the file name **4a.X68.**

Question 5: PC-Relative Addressing with Offset

Download the sample program called **5.X68** from the course website and open it in Easy68K. Examine the source code and make sure that you understand what the program is doing.

In the table below, print the names of the address modes that are being used to specify the location of the source and destination operands. Also, in each case specify the mode and register bits.

Instruction	Source Operand	Destination Operand
`MOVE.B   VALUE,D0`		
`MOVE.B  VALUE(PC),D0`		

Now, assemble the program, then write down the machine code for both instructions in the table below. Specify the operation word in the middle column and any extension word(s) in the right-most column.

Instruction	Operation Word	Extension Word(s)
`MOVE.B  VALUE,D0`		
`MOVE.B VALUE(PC),D0`		

What do these extension words refer to? Be specific.

Show how the assembler calculates the displacement required by the `MOVE.B VALUE(PC),D0` instruction. Do the calculation by hand.

If the program (i.e., data and instructions) was moved to a completely different part of the computer's memory at runtime, perhaps by the operating system, would the `MOVE.B VALUE,D0` work correctly? Explain.

If the program (i.e., data and instructions) was moved to a completely different part of the computer's memory, perhaps by the operating system, would the `MOVE.B VALUE(PC),D0` work correctly? Explain.