Freescale Semiconductor

AN1716

Using M68HC12 Indexed Indirect Addressing

By Marlan Winter

Introduction

Indexed indirect addressing (IIA) is an addressing mode infrequently found in CPU instruction sets. The IIA mode adds an additional level of indirection beyond standard indexed addressing modes. This application note presents some techniques for making the IIA mode more useful. The efficient instruction set and the addressing mode features of the M68HC12 allow it to compete effectively with RISC (reduced instruction set computer) processors with faster cycle times.

While microelectronic-controlled systems are usually thought of as large and complex, many new systems are small and have high production volumes. Applications engineers are, therefore, becoming more sensitive to system cost. As long as cost is an important consideration in a microelectronic-controlled system, then assembly language and code size will continue to be important. Memory management and usage are prime factors in determining the cost of a microelectronic-controlled system.

Since the IIA mode allows a programmer to compress more function in a single instruction, the assembly code is quite efficient. Fewer instructions mean smaller programs and fewer memory accesses for the system. The result is faster execution times and decreased size of the programs in memory. The IIA mode can also shrink code size because it permits more efficient algorithms.

freescale™
semiconductor

How IIA Works

Typical IIA mode assembly source code syntax appears very much like standard M68HC11 indexed addressing with the addition of brackets around the operands. Converting assembly code to use brackets instead of parentheses is a trivial exercise.

The valid source forms are:

```
inst       [16 bit offset data, index register or pc]
inst       [D, index register, or pc]
```

Examples:

```
staa       [D,x]
staa       [D,pc]
addd       [$103f,y]
ldd        [_TEMPVAR,sp]
```

Invalid Forms:

```
ldaa       [NUM,x+] ;post and preincrement are not allowed
                     inside the brackets
ldaa       [NUM,x]+ ;post and preincrement are not allowed
                     outside the brackets either
ldaa       (NUM,pc) ;we use the brackets, not the
                     paranthesis
```

Valid index registers that can be used are the X, Y, or the stack pointer. The PC register can also be used when referencing table data that is embedded in the same memory used by the executing program.

AN1716

2

Freescale Semiconductor, Inc.

16-Bit Offset IIA Mode

The 16-bit offset indexed indirect addressing mode adds a 16-bit constant to the content of the index register, then uses the result as an intermediate address to the operand data. The 16-bit offset is provided to the CPU in the program. This means that the 16-bit offset mode takes more bytes to implement than the D accumulator mode.

Examples:

```
Given: Address 10 contains 20, Address 20 contains 40,
       index register X contains 5
           inc   5,x     ;Address 10 will contain 21
Given: Address 10 contains 20, Address 20 contains 40,
       index register X contains 5
           inc   [5,x]  ;Address 20 will contain 41
```

In the above example, the following steps took place in the increment instruction which used the IIA mode:

1. The content of the index register X is added to the 16-bit constant. In this case, X contains a 5, and the 16-bit constant is 5. These are added together to provide a result of 10.

2. The result of 10 is used as an address. The content of address 10 is 20.

3. The content of address 10 is then loaded into a temporary internal register. In this case, the internal register now will contain 20 (the content of address 10).

4. The content of the internal register (which is 20) is used as an address. This address is 20 and that is the address in which the inc[rement] function will be performed.

5. The content of address 20 is operated on according to the indicated instruction. In this example, the instruction is an increment instruction so the content of address 20 is incremented. The end result is that the 40 is changed to 41 in memory location 20.

AN1716

3

Freescale Semiconductor, Inc.

Freescale Semiconductor, Inc.

D Accumulator Offset IIA Mode

The D accumulator offset indexed indirect addressing mode adds a 16-bit value stored in the D accumulator to the content of the index register, then uses the result as an intermediate address to the operand data.

Examples:

```
Given: Address 10 contains 20, Address 20 contains 40,
       index register X contains 5, d accumulator contains 5
          inc  [d,x]  ;Address 20 will contain 41
```

In the above example the following steps took place in the increment instruction which used the indexed indirect addressing mode.

1. The content of the index register X is added to the D accumulator. In this case, the content of X is 5, and the content of the D accumulator is 5. These are added together to provide a result of 10.

2. The result of 10 is used as an address. The content of address 10 is 20.

3. The content of address 10 is then loaded into a temporary internal register. In this case the internal register now will contain 20 (the content of address 10).

4. The content of the internal register (which is 20) is used as an address. This address is 20 and that is the address in which the inc[rement] function will be performed.

5. The content of address 20 is operated on according to the indicated instruction. In this example, the instruction is an increment instruction so the content of address 20 is incremented. In this example, the end result is that 40 is changed to 41 in memory location 20.

AN1716

Techniques

Jump Tables and Data Tables

Jump tables are blocks of data that cannot be executed, but instead represent a list of addresses that point to executable segments of code. Data tables are also blocks of addresses that point to segments of data such as data types that are created by the C language "struct" operator.

This is a valid use of JUMP tables appropriate for JSR and JMP instructions:

```
my_table:
        fdb         add_element
        fdb         del_element
        fdb         mov_element
        fdb         rot_element
end_my_table:
add_element:
        ...                 ;code here
        rts
del_element:
        ...                 ;code here
        rts
etc.
```

This example is a valid use of a JUMP table appropriate for the CALL instruction:

```
my_table:
        fdb         add_element
        fcb         mod_database_page
        fdb         search_for_element
        fcb         manipulate_db_page
end_my_table:
add_element:
        ...                 ;code here
        rtc
search_for_element: ;this code is located on a different page
                    ;from the add_element routine
        ...                 ;code here
        rtc
```

AN1716

This next example is a valid use of a DATA table. This data table should be stored in RAM since updating its elements is a fundamental element of a technique for managing databases without moving the data.

```
current_element ds 2            ; pointer to the data,
                                ; not the data itself!

first_element   ds 2
last_element    ds 2
some_code:
   stx   current_element       ;store a valid pointer in the
                               ;current_element
   ...                         ;several lines of code in which X
                               ;is used
   clrx
   ldd   [current_element,x]   ;loads the d register
                               ;with first part of the data
```

Where Can I Use Them?

Jump tables are useful anywhere in a user program where the routines can be referred to numerically. This could be an index of a dipswitch, the result of a calculation, a routine number such as is commonly used in real-time operating systems, a system level priority, or the results of case statements. Jump tables might also be implemented in software which has the capability of being expanded in the field with system plug-ins or the equivalent of dynamic link libraries (DLLs) used in windowing programs. In these applications, the software has pointers to "do-nothing" routines and when a new function is implemented, the "do-nothing" address is overwritten with the address of the new function.

Jump tables are a fundamental part of computed GOTOs. An organized design of the data structures in a program will make assembly much easier and more efficient. Jump tables aid in organization and they also make expansion of programs much easier.

AN1716

6

Computed GOTOs   The computed GOTO is a method of jumping to a series of subroutines based on the result of a calculation. The calculation may be as simple as telling the program to execute routine No. 4. Routine No. 4 might be the routine number because of a bit pattern in a control byte or any other construct that could generate the number 4.

Computed GOTOs can be used in any piece of code that needs to call a subroutine and an index or numerical result exists that enables selection of the routine.

An example is a routine based upon the setting of a 2-position dipswitch. There are four possible combinations, each determining a different course of action for the microcontroller. The following code is an example of how a selection might be accomplished on the M68HC12 microcontroller.

```
# assume porta is defined and it has the 2 position dipswitch
table:
    fdb       all_switches_off
    fdb       option_1_on
    fdb       option_2_on
    fdb       all_switches_on

computed_GOTO:
    ldab      porta
    aslb                 ;2 byte increments
    exg       b, x       ;new to the hc12 (zero extend)
    jsr       [table,x]  ;new to the hc12
    ...                  ;continue from here

all_switches_off:
    ...                  ;your code here
    rts
option_1_on:
    ...                  ;your code here
    rts
option_2_on:
    ...                  ;your code here
    rts
all_switches_on:
    ...                  ;your code here
    rts
```

Step-by-Step
Example

Make Subroutines

First, it is important to make the subroutines without too much concern about what they will be called later. In this example, the assumption is that the routines will not pass any variables.

Put the subroutines anywhere it is convenient, since they will be called from a computed GOTO. Make sure that every subroutine has meaningful labels, using as many characters as needed, at every possible point of entry. Another excellent practice is to have a single exit point from every routine.

Put Data in a Table

After all the building blocks (subroutines) are present so that enough code can be executed to provide a meaningful task, collect all entry labels into a table. It is important to remember that the arrangement of the labels in this table influences the GOTO calculations. In the current example, if the labels in the table are arranged differently, it will be difficult to perform an easy calculation to get the index into the jump table.

On a microcontroller that does not have indexed indirect addressing, the table would have to include instructions. This means that the same construct would execute much slower because of the additional program fetch. The table would also require an additional byte-per-entry for the instruction.

Do Calculations

Frequently, it is possible to get the index to the subroutines in the form 0,1, 2, . . . x. In the current example, a dipswitch can be read directly and actions can be directly branched to without needless compare and branch operations. After the index is gathered, a calculation to correct for the size of the address is needed. Since we have 16-bit addresses, it takes two bytes to describe the location of the subroutine relative to a starting point. The current example has routines in these locations: table+0, table+2, table+4 and table+6. It is necessary to multiply the dipswitch settings 0, 1, 2 and 3 by 2 to get the 0, 2, 4, and 6 numbers that will be added to the label "table." Of course, a multiply by 2 can be accomplished simply by an ASLB

AN1716

8

instruction. If the application uses 24-bit addresses, the M68HC12's multiply instruction is so fast that a corrected index can be generated in three cycles. In this example, the calculation correction is by 3, and not by 2:

```
ldab    porta
ldaa    #3              ;2 byte increments
mul                     ;only 3 cycles in the cpu12
tfr     d, x            ;new to the hc11
jsr     [table,x]       ;new to the cpu12
```

Check Boundaries

Although this example does not include it, there is a chance that the calculations would point to a subroutine that was not defined in the table. In the next example, there is now a 3-position dipswitch. When the switch is read, there is a possibility that 0, 1, 2, 3, 4, 5 will be read. Since there are no subroutines defined for positions 4 and 5, the course of action may depend on the kind of error handling that is desired.

One way to handle this error condition would be to use the AND instruction and force the unused bits to zeros. Another way would be to check the value against an upper limit and force it to the maximum allowed value or to the minimum allowed value.

In the current example, the cost of adding additional code is more expensive than adding two more subroutine labels. The new table is:

```
table:
    fdb     all_switches_off
    fdb     option_1_on
    fdb     option_2_on
    fdb     all_switches_on
    fdb     dipswitch_error_handler
    fdb     dipswitch_error_handler
```

AN1716

9

The M68HC12 has instructions that are very useful for boundary checking. In the current example, it is now assumed that the table is not extended, but instead includes instructions for boundary checking. The computed GOTO section would look like this:

```
table:
   fdb  all_switches_off
   fdb  option_1_on
   fdb  option_2_on
   fdb  all_switches_on
table_end:
   fdb  dipswitch_error_handler
   fdb  (table_end-table) ;largest legal offset
computed_GOTO:
   ldx  #table
   ldab porta
   aslb                    ;2 byte increments
   clra                    ;fast
   emind (table_end+2),x   ;check bounds - new to HC12
   jsr  [d,x]              ;new to the hc12
   ...                     ;continue from here
```

Here, the calculation section works a little differently than before because the EMIND instruction uses indexed addressing modes.

Make Jump

Several types of jumps need to be considered. While most can accomplish the same thing in one way or another, good program design requires planning, since each implementation has effects in both the generation of the computed GOTOs as well as with the subroutines the GOTOs will call. There are also the usual trade-offs among code complexity, speed of execution, and modularity.

- JMP — Using JMP indexed indirect is the fastest of all jumps that can be used. Advantages include speed of execution, no stacking, and no additional jump manager code after the JMP instruction. Disadvantages are lack of modularity and no chance to have useful exit code after the JMP instruction. The exit software is important since JMP relies on the returns of the subroutines it calls. It cannot be used both for interrupts and in-line code at the same time. Careful use of exit code can aid in reuse and reliability.

AN1716

- JSR — Using JSR indexed indirect is probably the best of all choices. It is fast, allows modularity, allows exit code, and the stacking it does provides ample opportunities for sophisticated stack operations.

- CALL — The call instruction can be used best when the M68HC12 is in a memory expansion scheme. In memory expansion routines, CALL can be immensely useful, since its indexed indirect addressing mode allows tabling of the expansion page value as well as the 16-bit address. This allows the M68HC12 to use multiple pages in a single jump (in this case call) table.

- BRA, BSR, LBRA, and LBSR — Since these instructions do not have indexed addressing modes, they are not valid candidates for computed GOTOs. However, there is a relationship between LBRA, LBSR, JMP and JSR, which is shown here:

```
jmp    destination-*,pc  ;same as LBRA
jsr    destination-*,pc  ;same as LBSR
(LBSR not a standard HC12 mnemonic)
```

An Important Variation

In this variation, the table is placed directly after an indexed indirect JMP using the program counter relative mode of addressing. This has the advantage of not using an index register, thus avoiding the loads required. The trade-off is that flexibility is not as great, as JMP is the central GOTO mechanism.

```
computed_GOTO:
    ldab    porta
    rolb                    ;2 byte increments
    clra                    ;fast
    jmp     [d,pc]          ;new to the hc12
table:
    fdb     all_switches_off
    fdb     option_1_on
    fdb     option_2_on
    fdb     all_switches_on
table_end:
    ...                     ;continue from here
```

Stack Operations

**Subroutine
Variable Passing**

In indexed indirect addressing mode, since the program works with the pointer to the data and not the data itself, this mode can easily be used from the stack. This is efficient because the programmer only has to pass an index which can be used to represent large amounts of data. By using indexing indirect, the data does not have to be removed from the stack or loaded into an index register. An example of using the indexed indirect addressing mode to access variables on the stack is:

```
calling_code:
    movw    #a_variable_address 2,-sp a way of making a push
                        ;immediate
    movw    #a_subroutine_address 2,-sp
    jsr     called_routine
    leas    4,sp
    ...                 ; code continuation

called_routine:
    jsr     [2,sp]  ;will execute the routine at
                    ;"a_subroutine_address"
    addd    [4,sp]  ;will add the value pointed to by
                    ;"a_subroutine_address"
    rts
```

Notice that an index register was not used and there were no unnecessary loads to use these variables after they were passed to the stack. Stack cleanup is quick and efficient by using the LEAS instruction to update the stack.

AN1716

Freescale Semiconductor, Inc.

Data after Call Technique (Use Indirect to Fetch Data Located after the Call)

If the time it takes to stack arguments is a problem, indexed indirect addressing can be used to access an operand. This technique has some disadvantages; the necessity for stack cleanup is one, but programming is a constant trade-off among speed, size, modularity, and other performance characteristics. An example of this technique is:

```
calling_code:
    jsr    called_routine
    fcb    parameter_value
    ...                        ;code continuation
called_routine:
    ldaa   [0,sp]
    staa   porta               ;we can get to this store quickly
    ldd    0,sp                ;load the return address
    addd   #1                  ;correct for space taken by
“parameter_value”
    std    0,sp                ;restore on stack
    rts                        ;now its ok to return
```

The CALL Instruction

The CALL instruction and its complementary instruction RTC (return from call) are special instructions which operate differently from JSR and RTS in useful ways. The primary purpose of the CALL instruction is to enable system designers to use the memory expansion on the M68HC12 while retaining code compatibility with the M68HC11, including the stack frame. The CALL instruction essentially is a 24-bit JSR instruction which updates a memory expansion register allowing the MCU to access additional memory via expansion address bits. Without the CALL instruction, 16-bit machines resorted to complex memory expansion schemes with significant software overhead. If an MCU does not have external memory or peripherals connected to the memory expansion chip selects, the CALL instruction may still be used, but is less efficient than simple JSRs.

Indirect Addressing Special

When designing the CALL instruction, the indexed indirect mode will not work efficiently without making a modification. When using the indexed indirect addressing, the final result pointed to by the indirect pointer is not a 16-bit value, but rather a 24-bit value.

The 24-bit addressing of the M68HC12 is stored in a way which makes it code-compatible with the M68HC11. The stack image for a CALL looks exactly like that of an M68HC11 with an extra byte stacked. All a programmer has to do is pull a byte off of the stack to get a stack image identical to that of an M68HC11. This feature is also convenient for changing the return page with a simple pull/push combination. Remember that the M68HC12 has a 64-Kbyte CPU with memory expansion that offers many new possibilities to the programmer.

Memory Management

There is a another form of memory management that is easily implemented with indexed indirect addressing that potentially could be very useful. This form of memory management is not an original idea, but is used by the Apple Macintosh operating system.

In this scheme, system memory is divided into two sections: The first section is a small area, which is in a fixed location and never moves; the second location is the memory that can be allocated, deallocated, and moved by the memory manager.

When a program needs memory, it makes a request to a firmware routine. The firmware routine then looks in the large second area for a block the size that was requested. When it finds the block, it places the address of the block into the next available location in the first small section. Then, the address in the first small section is handed to the program which originally made the request.

The real power of this technique is that when the memory manager moves blocks to de-fragment memory, it simply moves the memory and then updates its location in the first small area. As long as the main program only refers to allocated memory via the pointer, it will never lose the block.

AN1716

Complementary Information

LEA Indirect Addressing

A quick look at the LEA instruction reveals that it does not have an indexed indirect addressing mode. While it seems that this is an omission, there are equivalent instructions. These are:

```
ldy     0,x         ;y gets the content of address 0+x
leay    0,x         ;y gets the value 0+x
ldy     [0,x]       ;y gets the content of address pointed to
                    ;by 0+x
leay    [0,x]       ;y gets the content of 0+x
                    ;(but this doesn't exist)
ldy     0,x         ;y gets the content of address 0+x (this
                    ;is the equivalent)
```

Another example:

```
leax    [0,y]       ;doesn't exist
ldx     0,y         ;equivalent
```

The EMAXD Genre (Bounds Checking)

Several new instructions in the CPU12 can aid in boundary checking. These max and min instructions (EMAXD, EMAXM, EMIND, EMINM, MAXM, MAXA, MINM, and MINA) can be used to perform boundary checks when doing computed GOTOs. These instructions are the equivalent of a series of instructions.

An example of using the EMIND instruction is:

```
emind (table_end-table),x ;check bounds - new to HC12
```

Before using this instruction, the D accumulator should have the index value of the routine that it should execute selected from a table of possible routines. Since there is a possibility of calculating the index to a routine which puts the value outside the range of allowable values, the EMIND instruction will compare the index to the maximum allowable index and then replace it with the maximum allowable, if necessary. This will keep the CPU12 program from going into an error condition.

AN1716

15

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

AN1716/D

For More Information On This Product,
Go to: www.freescale.com