AN1218

HC05 to HC08 Optimization

By Mark Glenewinkel
CSIC Applications
Austin, Texas

Introduction

Freescale's HC05 Family of microcontrollers contains the world's most popular 8-bit microcontroller units (MCUs). In keeping pace with technology and the changing needs of the customer, Freescale has designed the HC08 Family of MCUs. The HC08 Family CPU is a performance extension to the HC05 Family of low cost MCUs. This application note will describe the differences and advantages of the HC08 Family CPU: the CPU08.

CPU08 is fully opcode and object code compatible with the HC05 CPU. Any HC05 code will execute directly on the HC08 without instruction set differences. As this application note will show, there are many improvements to the speed and capability in the CPU08.

CPU08 is a faster processor. The basic execution speed of the CPU08 has been increased with advanced high performance CMOS technology. Execution cycles of most instructions have been improved with an advanced computer architecture.

CPU08 has more programming capability. It has more addressing modes, better math support, and much improved data manipulation, accessing, and moving capabilities. Looping and branching instructions have also been optimized.

This application note will help inform and educate the reader concerning the differences between the HC05 and HC08 CPUs. Detailed examples illustrating the added features found with the CPU08 are given to help optimize software design with the CPU08.

Scope of this Application Note

This note assumes the reader has a background in MCU software and hardware design and is also familiar with the HC05. It was written for the engineering manager and the design engineer. As a reference, the application note overviews the basic differences between the two CPUs so that one can fit the right CPU for a specific application. As a tutorial, the application note gives the designer the means to understand and utilize the HC08 enhancements. Software is given to illustrate and compare the performance of the CPUs.

HC08 Features

The following is a list of major features of the HC08 CPU (CPU08) that differentiate it from the HC05 CPU (CPU05).

- Fully upward object code compatible with the MC6805, MC146805, and the MC68HC05 Family
- 64 KByte program/data memory space
- Enhanced HC05 programming model
- 8 MHz CPU bus frequency
- 16 addressing modes, 5 more than the HC05
- Expandable internal bus definition for addressing range extension beyond 64 KBytes
- 16-bit index register with manipulation instructions
- 16-bit stack pointer with manipulation instructions
- Memory to memory data moves without using the accumulator

- Fast 8-bit multiply and integer/fractional divide instructions
- Binary coded decimal (BCD) instruction enhancements
- Internal bus flexibility to accommodate CPU enhancing peripherals such as a DMA controller
- Fully static low voltage/low power design

CPU05/CPU08 Programmer's Model Comparison

The CPU05 and the CPU08 programmer's model differences are illustrated in **Figure 1**.

H Index Register

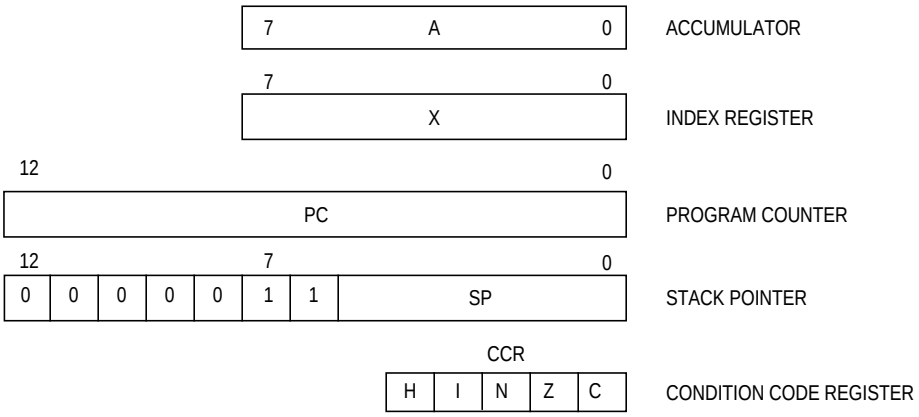
The index register of the CPU08 has been extended to 16 bits, allowing the user to index or address a 64 KByte memory space without any offset. The upper byte of the index register is called the H index register. The concatenated 16-bit register is called the H:X register. Source code written for CPU05 will not affect the H register and it will remain in its reset state of $00. There are seven new instructions that allow the user to manipulate the H:X index register. These instructions are covered in detail later.

Stack Pointer

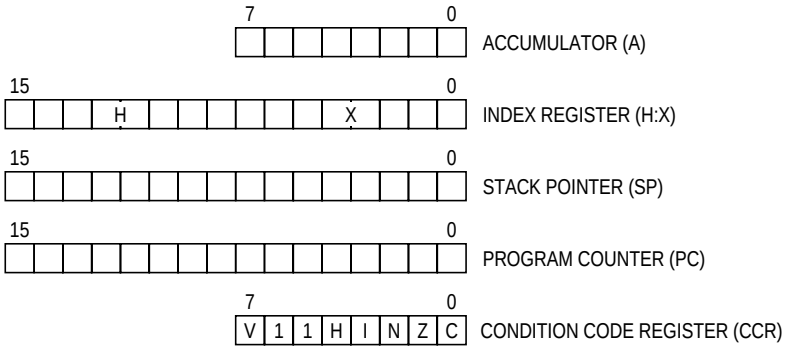
The stack pointer (SP) has been extended from its 6-bit CPU05 version to a full 16-bit SP on the CPU08. SPH:SPL refers to the 16-bit stack pointer by naming the high byte, SPH, and the low byte, SPL. To maintain HC05 compatibility, the reset state is $00FF.

New instructions and new addressing modes greatly increase the utility of the CPU08 stack pointer over the CPU05 stack pointer. Nine new CPU08 instructions allow the user to easily manipulate the SP and the stack.

CPU08 also has relative addressing modes that allow the SP to be used as an index register to access temporary variables on the stack. These addressing modes and new instructions are discussed later in this application note.



(a) CPU05



(b) CPU08

Figure 1. CPU05 and CPU08 Comparison

Program Counter Expanded

The CPU08 program counter (PC) has been expanded to 16 bits which allows the CPU08 to address 64 KBytes of memory. Not all HC05 devices have a 16-bit program counter.

New Addressing Modes, Comparison

CPU08 has 16 addressing modes, 8 more than the HC05. **Table 1** lists these addressing modes and the CPUs that use them. A brief discussion of these modes is given below.

Table 1. Addressing Mode Comparison Table

Addressing Mode	HC05	HC08
Inherent	X	X
Immediate	X	X
Direct	X	X
Extended	X	X
Indexed, no offset	X	X
Indexed, 8-bit offset	X	X
Indexed, 16-bit offset	X	X
Relative	X	X
Stack Pointer, 8-bit offset		X
Stack Pointer, 16-bit offset		X
Memory to memory (4 modes)		X
Indexed w/post increment		X
Indexed, 8-bit offset, w/post increment		X

HC05 and HC08 Addressing Modes

Inherent instructions such as reset stack pointer (RSP) and multiply (MUL) have no operand. Inherent instructions require no memory address and are one byte long.

Immediate instructions contain a value that is used in an operation with the index register or accumulator. Immediate instructions require no memory address and are two bytes long. The operand is found in the byte immediately following the opcode.

Direct instructions can access any of the first 256 memory addresses with only two bytes. The first byte contains the opcode followed by the low byte of the operand address. The CPU automatically uses $00 for the high byte of the operand address. Most direct instructions are two bytes long.

Extended instructions can access any address in the memory map. Extended instructions are three bytes long and contain the opcode and the two-byte operand address.

Indexed instructions with no offset are one-byte instructions that utilize the index register of the CPU. CPU08 also uses the H:X register containing the high byte of the address operand.

Indexed, 8-bit offset instructions are two-byte instructions that utilize the index register of the CPU to access data at any location in memory. The 8-bit unsigned offset following the opcode is added to the 16-bit unsigned index register (H:X). The sum is the address used to access data.

Indexed, 16-bit offset instructions are like the 8-bit offset instructions except that they are three bytes long and add a 16-bit unsigned number to the 16-bit index register (H:X).

Relative addressing is only used for branch instructions. If the branching condition is true, the CPU finds the branch destination by adding the offset operand to the PC counter. The offset is a two's complement byte that gives a branching range of –128 to +127 bytes. This instruction is two bytes long.

New HC08 Addressing Modes

Stack pointer, 8-bit offset instructions operate like indexed, 8-bit offset instructions except that they add the offset to the 16-bit SP. This mode is available only on the CPU08. If interrupts are disabled, this addressing mode allows the SP to be used as a second index register. This instruction is three bytes long.

Stack pointer, 16-bit offset instructions are only available on the CPU08. They are like the stack pointer, 8-bit offset instructions except that they add a 16-bit value to the SP. This instruction is four bytes long.

Memory to memory instructions utilize four different modes available only to the CPU08.

1. The move, immediate to direct, is a three-byte mode generally used to initialize RAM and register values in page 0 of the memory map. The operand in the second byte is immediately stored to the direct page location found in the third byte.
2. The move, direct to direct, is a three-byte instruction. The operand following the opcode is the direct page location that is stored to the second operand direct page location.
3. The move, indexed to direct, post increment, is a two-byte instruction. The operand addressed by the 16-bit index register (H:X) is stored to direct page location address by the byte following the opcode. The index register is then incremented.
4. The move, direct to indexed, post increment, is a two-byte instruction. The operand in the direct page location addressed by the byte following the opcode is stored in the location addressed by the 16-bit index register (H:X). The index register is then incremented.

In the CPU08, four instructions address operands with the index register and then increment the index register afterwards. This is called indexed with post increment mode. These instructions include CBEQ indexed, CBEQ indexed with offset, MOV IX+Dir, and MOV DirIx+.

Table 2 gives examples to illustrate these different addressing modes.

Table 2. Addressing Mode Examples

Addressing Mode	Example	
Inherent	RSP	
Immediate	LDA	#$FF
Direct	LDA	$50
Extended	LDA	$1000
Indexed, no offset	LDA	,X
Indexed, 8-bit offset	LDA	$50,X
Indexed, 16-bit offset	LDA	$0150,X
Relative	BRA	$20
Stack Pointer, 8-bit offset*	LDA	$50,SP
Stack Pointer, 16-bit offset*	LDA	$0150,SP
Memory to memory ImmDir* DirDir* Ix+Dir* DirIx+*	 MOV MOV MOV MOV	 #$30,$80 $80,$90 X+,$90 $80,X+
Indexed w/post increment*	CBEQ	X+,LOOP
Indexed, 8-bit offset, w/post increment*	CBEQ	$20,X+,LOOP

* New CPU08 addressing modes

Condition Code Register with Overflow Bit V

A summary of the condition code register (CCR) is given below. Unless otherwise stated, all bits correspond to both CPUs.

Overflow Bit V

This bit is set when a two's-complement overflow has occurred as the result of an operation. The V bit has been added to the CPU08 condition code register to support two's-complement arithmetic.

Half-Carry Bit H

The half-carry bit is set when a carry has occurred between bits 3 and 4 of the accumulator because of the last ADD or ADC operation. This bit is required for BCD operations.

Interrupt Mask Bit I

All timer and external interrupts are disabled when this bit is set. Interrupts are enabled when the bit is cleared. This bit is automatically set after any CPU reset.

Negative Bit N

This bit is set after any arithmetic, logical, or data manipulation operation was negative. In other words, bit 7 of the result of the operation was a logical one.

Zero Bit Z

The zero bit is set after any arithmetic, logical, or data manipulation operation was zero.

Carry/Borrow Bit C

The carry/borrow bit is set when a carry out of bit 7 of the accumulator occurred during the last arithmetic, logical, or data manipulation operation. The bit is also set or cleared during bit test and branch instructions and shifts and rotates.

Description of the Clock

In the CPU08, the CPU clock rate is twice that of the address/data bus rates. The internal CPU08 clock rate is 16 MHz for an 8 MHz HC08. To maintain a 50% duty cycle CPU clock, the oscillator clock, OSC CLK, must run twice the rate of the CPU clock. Therefore a 32 MHz OSC clock is needed to drive an 8 MHz HC08.

The flagship member of the CPU08 family has a phase locked loop (PLL) synthesizer to generate the 32 MHz signal. It is derived from a suggested crystal frequency of 4.9152 MHz.

Address/Data Rate	=	Z	=	8 MHz
CPU Clock Rate	=	2Z	=	16 MHz
OSC Clock Rate	=	4Z	=	32 MHz

Index Registers

CPU08 has the additional H index register which is the high byte extension to the X index register. Together, the two index registers formulate the concatenated 16-bit H:X index register. Five new instructions are introduced on the CPU08 to allow manipulation of the H:X index register. Source code written for the HC05 will not effect the H register and it will remain in its reset state of $00.

The TSX and the TXS instructions also utilize the H:X index register. These instructions are covered in more detail in the stack pointer section.

Five New Indexing Instructions, Detail

The new CPU08 instructions that affect the index registers are listed below. Examples for these instructions are given in **Appendix A — New CPU08 Indexing Instruction Examples**.

AIX Add Immediate to Index Register

Operation: X ← (H:X) + (M)

Description: AIX adds an immediate value to the 16-bit index register formed by the concatenation of the H and X registers. The immediate operand is an 8-bit two's complement signed offset. Prior to addition to H:X, the offset is sign extended to 16 bits.

CLRH Clear Index High

Operation: H ← $00

Description: The contents of H are replaced with zeros.

CPHX Compare 16-bit Index Register

Operation: (H:X) – (M:M+1)

Description: CPHX compares the 16-bit index register H:X with the 16-bit value in memory and sets the condition code register accordingly.

LDHX Load 16-bit Index Register

Operation: H:X ← (M:M+1)

Description: Loads the contents of the speci ed memory location into the 16-bit index register H:X. The condition codes are set according to the data.

STHX Store 16-bit Index Register

Operation: (M:M+1) ← (H:X)

Description: Stores the 16-bit index register H:X to the speci ed memor y location. The condition codes are set according to the data.

Software Techniques Using Indexed Addressing, Tables

The CPU08 index register has some distinct advantages over the CPU05 index register. Even though the CPU05 has 16-bit index offset, the 8-bit index register restricts indexing to a maximum of 256 bytes. CPU08 with its H register extension allows full 16-bit index addressing equaling 65,536 bytes of memory access. Proper 16-bit pointers allow efficient compiling of C code and other higher level languages. Maximum table lengths in the CPU08 which can be accessed in a single instruction are therefore 64 KByte. An optional address extension module can extend the data space beyond 64 KBytes, but the maximum offset remains 64 KBytes. Index addressing modes include 8- and 16-bit offsets.

Many programmers like to use calculated addressing. CPU08 has a new instruction, AIX, that allows the addition of a two's complement number. Table access is easier and more flexible.

The H:X index register can also be used as an auxiliary 16-bit accumulator. Sixteen-bit data comparisons are easier with the CPHX instruction.

The following section illustrates the advantage of using a 16-bit index register.

Code Example

We will now illustrate the added benefit of the CPU08 16-bit index register. The index will be used to address a 512 byte table. In the CPU05, the table must be broken up into sections of memory consisting of 256 bytes per section. Our table has 512 bytes, so we will be using two sections, section 0 and section 1, for the CPU05. The address to look up on the table will be found in RAM. Notice that the CPU05 code is longer. If your table was larger, you would require more sections of memory to handle your table. A subroutine might be written to make the job more modular. In the HC08 example, the 512 byte table can be handled directly. A comparison between CPU05 and CPU08 code is shown in **Appendix B — CPU05 and CPU08 512-Byte Table Indexing Code**.

Stack Pointer

CPU08 has a full 16-bit stack pointer. To maintain compatibility with the CPU05, it is initialized to $00FF out of reset.

Stack manipulation is from high to low memory. The SP is decremented each time data is pushed on the stack and incremented each time data is pulled from the stack. The SP points to the next available stack address rather than the latest stack entry address.

Nine new instructions have been added for the user to manipulate the stack. These instructions allow the direct push and pull of any register to the stack. The SP can be changed with a transfer of the H:X register to the SP or the SP can be augmented by the add immediate instruction.

Stack manipulation can be a very powerful programming technique. With the CPU08, the assembly programmer can pass parameters and store local or temporary variables when using subroutines and/or interrupts.

New addressing modes were added to address these variables on the stack. Using the stack pointer as an index register with 8- or 16-bit offsets, the user may access variables on the stack. These instructions greatly cut cycle count by not having to load/store the variable. RAM

requirements are also reduced. Significant C code efficiency can be gained when utilizing these new stack pointer addressing modes.

If interrupts are disabled, the stack pointer can be used as a second 16-bit index register with 8- or 16-bit offsets.

Nine New Stack Manipulation Instructions, Detail

All the new CPU08 instructions that affect the stack pointer are listed below. Examples for these instructions are given in **Appendix C — New CPU08 Stack Pointer Instructions**.

AIS Add Immediate to Stack Pointer

Operation: SP ← (SP) + (M)

Description: Adds the immediate operand to the stack pointer SP. The immediate value is an 8-bit two's complement signed operand. Prior to addition to the SP, the operand is sign extended to 16 bits. This instruction can be used to create and remove a stack frame buffer which is used to store temporary variables.

PSHA Push Accumulator onto Stack

Operation: ⇓ (A); SP ← (SP-$01)

Description: The contents of the accumulator are pushed onto the stack at the address contained in the stack pointer. The stack pointer is then decremented to point at the next available location in the stack. The contents of the accumulator remain unchanged.

PSHH Push Index Register H onto Stack

Operation: ⇓ (H); SP ← (SP-$01)

Description: The contents of the 8-bit high order index register H are pushed onto the stack at the address contained in the stack pointer. The stack pointer is then decremented to point at the next available location in the stack. The contents of the H register remain unchanged.

PSHX Push Index Register X onto Stack

Operation: ⇓ (X); SP ← (SP-$01)

Description: The contents of the 8-bit low order index register X are pushed onto the stack at the address contained in the stack pointer. The stack pointer is then decremented to point at the next available location in the stack. The contents of the X register remain unchanged.

PULA Pull Accumulator from Stack

Operation: SP ← (SP+$01); ⇑ (A)

Description: The stack pointer is incremented to address the last operand on the stack. The accumulator is then loaded with the contents of the address pointed to by SP.

PULH Pull Index Register H from Stack

Operation: SP ← (SP+$01); ⇑ (H)

Description: The stack pointer is incremented to address the last operand on the stack. The 8-bit index register H is then loaded with the contents of the address pointed to by SP.

PULX Pull Index Register X from Stack

Operation: SP ← (SP+$01); ⇑ (X)

Description: The stack pointer is incremented to address the last operand on the stack. The 8-bit index register X is then loaded with the contents of the address pointed to by SP.

TSX Transfer Stack Pointer to Index Register

Operation: H:X ← (SP) + $0001

Description: Loads the index register H:X with one plus the contents of the 16-bit stack pointer SP. The contents of the stack pointer remain unchanged. After a TSX instruction, the

index register H:X points to the last value that was stored on the stack.

TXS Transfer Index Register to Stack Pointer

Operation: SP ← (H:X) - $0001

Description: Loads the stack pointer SP with the contents of the index register H:X minus one. The contents of the index register H:X remain unchanged.

Software Techniques Using the SP

The CPU05 and the CPU08 use the stack for two primary purposes. First, every time the CPU executes an interrupt service routine, the register contents are saved on the stack. After the execution of a return from interrupt (RTI) instruction, the register contents on the stack are restored to the CPU. Second, every time a jump to subroutine (JSR) or a branch to subroutine (BSR) occurs, the return address is saved on the stack. The address is restored to the program counter after a return from subroutine (RTS) instruction is executed.

The CPU08 with its new stack manipulation instructions allows the user to pass parameters to the subroutine and store local or temporary values within the subroutine. Two major benefits are derived from using the stack for parameters and temporary values:

1. A subroutine will allocate RAM storage for its variables and release this memory when the subroutine is finished. Therefore, global variables are not needed for these routines. This saves RAM memory space.
2. The allocation of new local variables for each subroutine makes the subroutine recursive and reentrant. This allows the programmer to easily modularize his code.

Let's look at the stacking operation of the CPU05 and the CPU08. The stack is located in RAM. Since stacking occurs from high memory to low memory, the SP usually points to the highest RAM memory address. Both the CPU05 and the CPU08 reset the SP at $00FF. The CPU08 instruction set allows the programmer to move the stack out of Page 0 memory if needed.

When an interrupt occurs, the contents of all the CPU registers are pushed onto the stack, the interrupt vector is fetched, and the program begins execution at the start of the interrupt routine. The stack contents before and after an interrupt are shown in **Figure 2**. For the CPU08 to remain upward compatible with the CPU05, the H index register is not pushed onto the stack.

NOTE: *If the H register is used in the interrupt service routine or if indexed addressing modes are used, the H register must be pushed onto the stack.*

This is accomplished by using the PSHH instruction. Before returning from the interrupt, the PULH instruction must be used to extract the H index register off the stack.

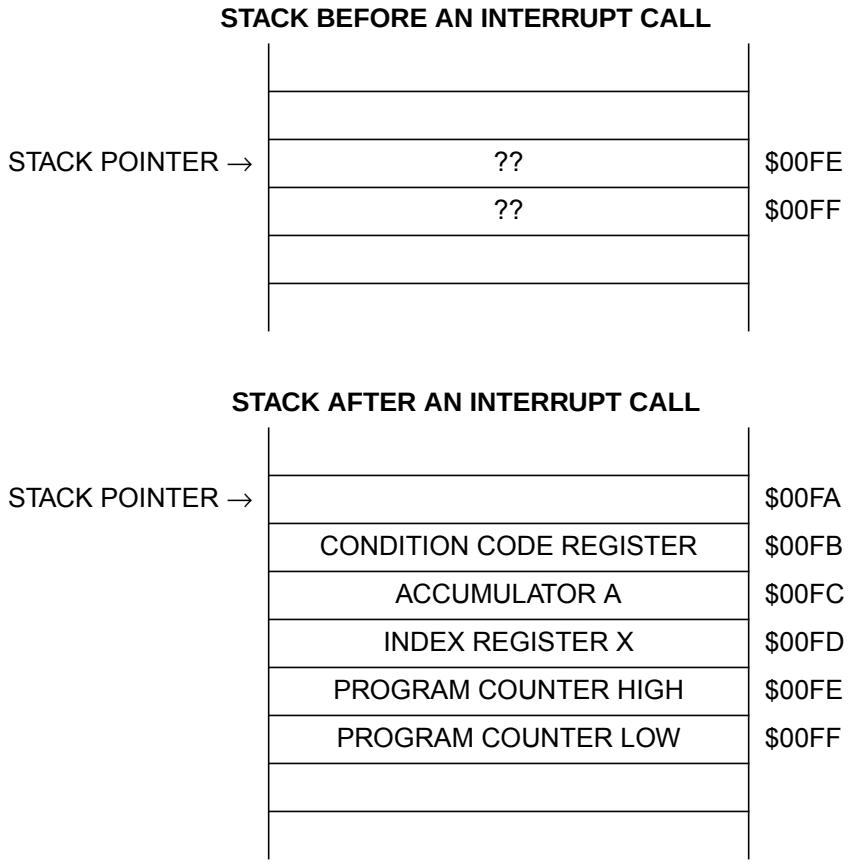


Figure 2. Stack Before and After an Interrupt Call

Figure 3 illustrates the stack before and after a subroutine is called when the stack pointer is at $00FF. When a subroutine is called, the 16-bit program counter is pushed onto the stack and the execution of code begins at the start of the subroutine. The program counter is split into its 8-bit high and low bytes.

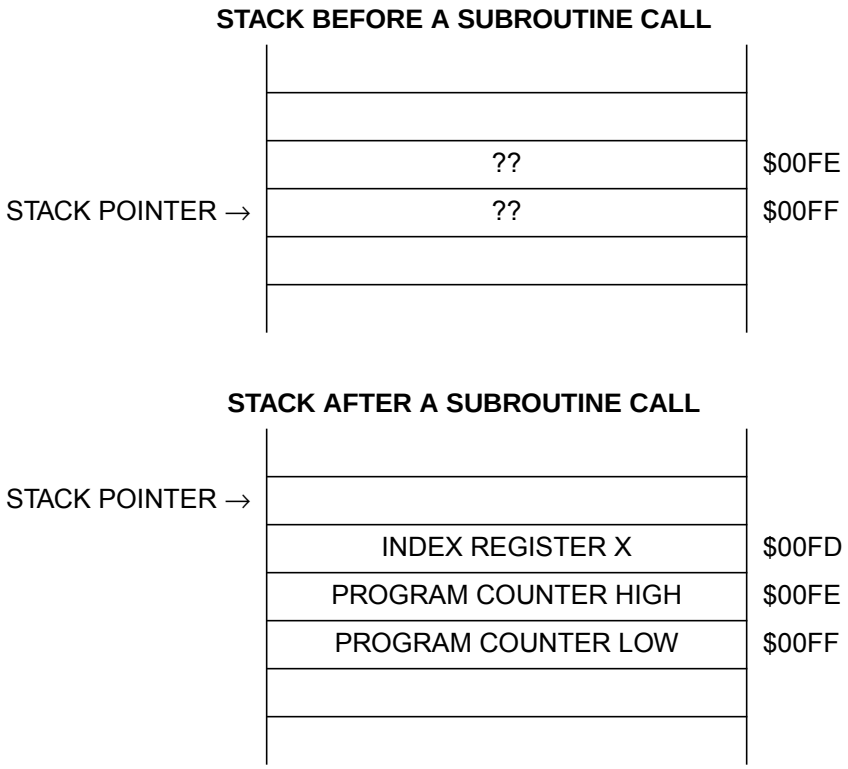


Figure 3. Stack Before and After a Subroutine Call

If the values in the X register and the accumulator are needed within a subroutine, they will need to be saved somehow before the subroutine uses them. If using the CPU05, you would have to allocate global RAM space for saving these CPU registers. Your code would look something like that in Figure 4.

```
*       Subroutine XX for CPU05       *

START   STX     $50             ;4 store X away to RAM
        STA     $51             ;4 store A away to RAM

        XX      XX              ;actual subroutine code
        XX      XX

        LDX     $50             ;3 load X from RAM
        LDA     $51             ;3 load A from RAM
```

Figure 4. CPU05 Subroutine Code

The CPU05 code will use 14 cycles to store and load registers. Also, two bytes of global RAM space are allocated for this subroutine. If we were to use the CPU08, the code could utilize the stack. Global RAM space and six cycles would be saved. Refer to **Figure 5**.

```
*       Subroutine XX for CPU08       *

START   PSHX                    ;2 push X onto stack
        PSHH                    ;2 push H onto stack
        PSHA                    ;2 push A onto stack

        XX      XX              ;actual subroutine code
        XX      XX

        PULA                    ;2 pull A off of stack
        PULH                    ;2 pull H off of stack
        PULX                    ;2 pull X off of stack
```

Figure 5. CPU08 Subroutine Code

The stack helps in efficiently utilizing parameters, local variables, and subroutine return values. Parameters are variables that are passed to the subroutine. Local variables are variables that are only used within the scope of the subroutine. A subroutine return value is the output of the subroutine. An example of a subroutine and its variables are given below in equation form:

$$Y = (X)3$$

If we were to write a subroutine that calculates the cube of the value X, X would be the parameter passed to the subroutine. Y would be the subroutine return value, and any variable used to calculate Y would be

a local variable. The stack of these complex subroutines follow the generalized structure shown in **Figure 6**. **Figure 6** shows the stack before the subroutine initialization, before entering the subroutine, and during the subroutine. The actual cube subroutine is written in the following section of code. A diagram of the stack during its execution is given within the code listing.

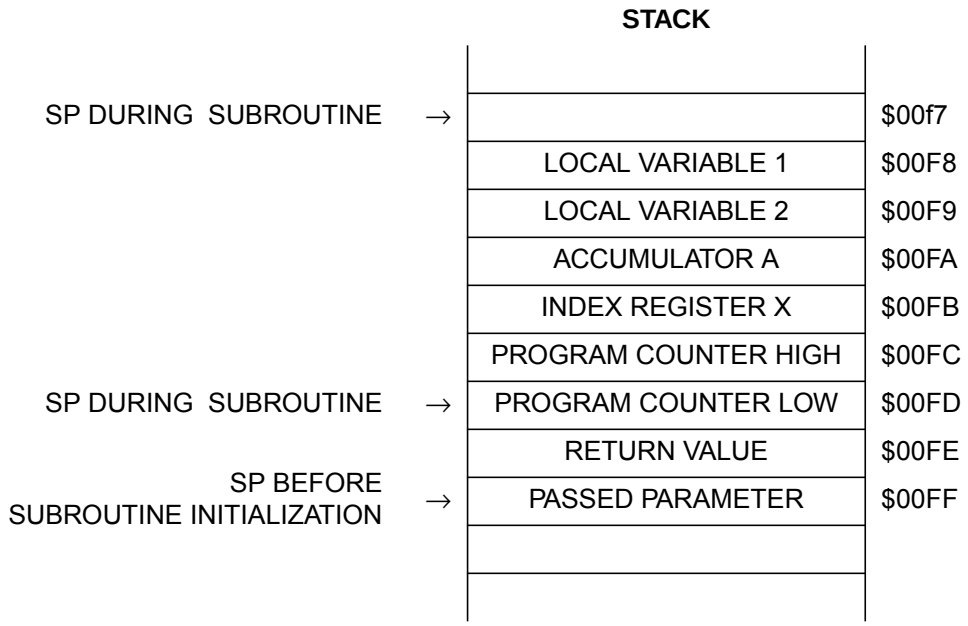


Figure 6. Stack Structure of a Complex Subroutine

Code Example

Refer to **Appendix D — Using the Stack in a Subroutine to Compute a Cube** for an example of modular subroutine code that efficiently computes the cube of an 8-bit positive number.

Data Movement

Why Improve the Movement of Data in the CPU05?

The most common CPU function is the transfer of data. Most microcontroller-based systems spend the majority of their time moving data from one location to the other. Many different addressing modes are used to access and transfer bytes of data. If there was a way to decrease the time it takes to transfer data, then the overall performance of the system would be improved.

CPU05 moves data from one location to the next by first loading the accumulator with the byte from the transfer source. Next, CPU05 stores the byte from the accumulator to the transfer's destination. In this manner all data must pass through the accumulator, thus making the accumulator a bottleneck in data movement. The movement of the contents of location $40 to location $60 with the CPU05 is illustrated in Figure 7.

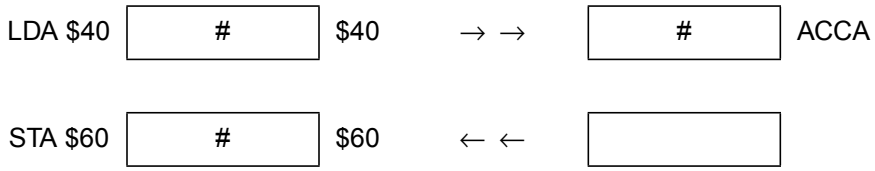


Figure 7. Accumulator as a Bottleneck

CPU08 provides the new MOV instruction which bypasses the accumulator. Using the MOV instruction, the CPU is instructed to take the contents of the source location and directly place the data in the destination. This is illustrated in Figure 8. There are four different addressing modes special to the MOV instruction. Details of this instruction are given below. .

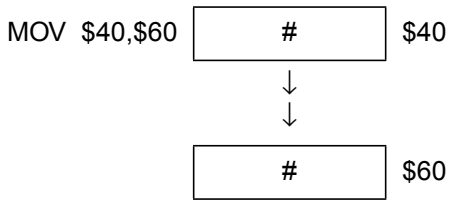


Figure 8. No Accumulator Bottleneck

New MOVE Instruction, Detail

The new CPU08 Move instruction is detailed below. Examples for this instruction and all four of its addressing modes are given in **Appendix E — New CPU08 MOV Instruction Examples**. The examples in **Appendix E — New CPU08 MOV Instruction Examples** also compare the CPU05 and the CPU08 bus cycles and memory requirements for the algorithm to execute the movement of data.

MOV Move

Operation: (M)destination ← (M)source

Description: Moves a byte of data from a source address to a destination address. Data is examined as it is moved, and condition codes are set. Source data is not changed. Internal registers (other than CCR) are not affected. There are four addressing modes for the MOV instruction. A discussion of these modes was given in an earlier section.

Software Techniques

The MOV command will cut cycle time and code space when moving data. The most obvious advantage of the MOV instruction is when the configuration registers are being initialized along with other RAM variables at the start of the program.

Code Example

A user wants to start his application one of two different ways. The user initializes the application on the MCU based on the logic level of port D bit 4. Once the part is out of reset, it reads port D and moves data from ROM into the RAM configuration registers according to the logic level of bit 4. Refer to **Appendix F — CPU05 and CPU08 Data Movement Code** for code comparing the CPU05 and the CPU08.

New Branch Instructions

Description

Six new unsigned branch instructions were added to the instruction set of the CPU08 to improve looping and table searching capabilities. These instructions are CBEQ, CBEQA, CBEQX, DBNZ, DBNZA, and DBNZX. The CBEQ instructions combine the compare (CMP and CPX) instructions and the branch if equal (BEQ) instruction . The DBNZ instructions combine the decrement (DEC, DECA, and DECX) instructions and the branch if not equal (BNE) instruction. These new instructions improve cycle time and decrease code space. More detail is given below on each instruction.

Six New Branch Instructions, Detail

All the new CPU08 instructions that affect branching are listed below. Examples for these instructions are given in **Appendix G — New Branch Instruction Examples**. The examples in **Appendix G — New Branch Instruction Examples** also compare the CPU05 and the CPU08 bus cycles and memory requirements for the algorithm to execute the branch.

CBEQ Compare and Branch if Equal

Operation: A) - (M); PC ← (PC) + $0003 + Rel if result is $00

For IX+ mode: (A) - (M);

PC ← (PC) + $0002 + Rel, if result is $00

Description: CBEQ compares the operand from memory with the accumulator and causes a branch if the result is zero. This function combines CMP and BEQ for faster table look-up routines.

The addressing mode CBEQ_IX+ compares the operand addressed by the 16-bit index register H:X to the accumulator and causes a branch if the result is zero. The 16-bit index register is then incremented regardless of whether a branch is taken. CBEQ_IX1+ operates the same way except

an 8-bit offset is added to the effective address of the operand.

CBEQA Compare and Branch if Equal

Operation: (A) - (M); PC ← (PC) + $0003 + Rel if result is $00

Description: CBEQA compares an immediate operand in memory with the accumulator and causes a branch if the result is zero. This instruction combines CMP and BEQ for faster table look-up routines.

CBEQX Compare and Branch if Equal

Operation: (IX) - (M); PC ← (PC) + $0003 + Rel if result is $00

Description: CBEQX compares an immediate operand in memory with the lower order index register X and causes a branch if the result is zero. This instruction combines CPX and BEQ for faster loop counter control.

DBNZ Decrement and Branch if Not Zero

Operation: M ← (M) - $01;

PC ← (PC) + $0003 + Rel, if result _ $00 for Direct, IX1, and SP1

PC ← (PC) + $0002 + Rel, if result _ $00 for IX

Description: DBNZ subtracts one from the operand M in memory and causes a branch if the result is not zero. This instruction combines DEC and BNE for faster loop counter control.

DBNZA Decrement and Branch if Not Zero

Operation: A ← (A) - $01; PC ← (PC) + $0002 + Rel, if result _ $00

Description: DBNZA subtracts one from the accumulator and causes a branch if the result is not zero. This instruction combines DECA and BNE for faster loop counter control.

DBNZX Decrement and Branch if Not Zero

Operation: X ← (X) - $01; PC ← (PC) + $0002 + Rel, if result _ $00

Description: DBNZX subtracts one from the lower index register and causes a branch if the result is not zero. This instruction combines DECX and BNE for faster loop counter control.

Code Example

The use of these new instructions can cut cycle time in looping or counting routines. Compare and branch routines can be used to search for specific values in tables or variable locations. Decrement and branch routines can be used for keeping count in loops.

The following piece of code shows how the compare and branch instruction searches a table for a match. As an example, let's say that you recently read in a table of 80 A/D data bytes. You would like to know if the signal was saturated above the rails of the A/D converter. You would then search the table for the value $FF. If found, your code would branch out and execute some control algorithm to attenuate the analog signal. Refer to **Appendix H — CPU05 and CPU08 Search Code** for a comparison of CPU05 and CPU08 code.

Mathematical Operations

V Bit, DIV, DAA, and the NSA Instruction

New features and instructions added to the CPU08 have made some mathematical computations easier. The V bit is added to the CCR to support signed arithmetic. CPU08 has the capability of 16-bit division. The DIV instruction will divide a 16-bit dividend by an 8-bit divisor. For binary coded decimal operations, the CPU08 has a decimal adjust accumulator, DAA, instruction and a nibble swap accumulator, NSA, instruction.

Signed Math and Signed Branches

The V bit in the CCR adds greater programming flexibility to the user. The addition of two's complement comparisons can aid in the branching

operations of high level languages such as C. Also, the representation of signed numbers and their operations can easily be computed. This can be especially helpful with digital signal processing algorithms and the proper storage of signed analog to digital readings.

Four New Signed Branch Instructions, Detail

All the new CPU08 instructions that affect signed branching are listed below. Examples for these instructions are given in **Appendix I — New CPU08 Signed Branch Instruction Examples**.

BGE Branch if Greater Than or Equal (signed operands)

Operation: PC ← (PC) + $0002 + Rel, if (N⊕V)=0, i.e., if (A) _ (M), ("signed" numbers)

Description: If the BGE instruction is executed immediately after execution of any of the compare or subtract instructions, the branch will occur if and only if the two's complement number represented by the appropriate internal register (A, X, or H:X) was greater than or equal to the two's complement number represented by M.

BGT Branch if Greater Than (signed operands)

Operation: PC ← (PC) + $0002 + Rel, if Z+(N⊕V)=0, i.e., if (A) > (M), ("signed" numbers)

Description: If the BGT instruction is executed immediately after execution of any of the compare or subtract instructions, the branch will occur if and only if the two's complement number represented by the appropriate internal register (A, X, or H:X) was greater than the two's complement number represented by M.

BLE Branch if Less Than or Equal (signed operands)

Operation: PC ← (PC) + $0002 + Rel, if Z+(N⊕V)=1
i.e., if (A) _ (M), ("signed" numbers)

Description: If the BLE instruction is executed immediately after execution of any of the compare or subtract instructions, the branch will occur if and only if the two's complement number represented by the appropriate internal register (A, X, or H:X) was less than or equal to the two's complement number represented by M.

BLT Branch if Less Than (signed operands)

Operation: PC ← (PC) + $0002 + Rel, if (N⊕V)=1
i.e., if (A) < (M), ("signed" numbers)

Description: If the BLT instruction is executed immediately after execution of any of the compare or subtract instructions, the branch will occur if and only if the two's complement number represented by the appropriate internal register (A, X, or H:X) was less than the two's complement number represented by M.

New DIV Instruction

The Divide instruction on the CPU08 does not require the lengthy code needed to divide numbers on the CPU05. A description of the Divide instruction is given below. **Appendix J — Five Miscellaneous CPU08 Instructions Including BCD, Divide, and CCR Operations** shows a short example of using the new Divide instruction. **Appendix K — CPU08 Averaging Code** illustrates an averaging routine implementing the Divide instruction.

DIV Divide

Operation: (H:A) / X → A; Remainder → H

Description: Divides a 16-bit unsigned dividend contained in the concatenated registers H and A by an 8-bit divisor contained in index

register X. The quotient is placed in the accumulator A, and the remainder is placed in the high order index register H. The divisor is left unchanged.

New DAA and the NSA instruction

The decimal adjust accumulator, DAA, and the nibble swap accumulator, NSA, are new instructions to help with binary coded decimal (BCD) operations. The DAA instruction allows the user to adjust the accumulator so that the number represents a BCD number. Swapping nibbles is needed for packing BCD numbers into memory. One use of BCD is data instrumentation. It is easier to store and manipulate these numbers in BCD rather than convert or decode numbers from hexadecimal. Packing is used to store decimal numbers into memory. Instead of one byte storing one decimal, the NSA instruction easily swaps nibbles in the accumulator so that two decimal numbers can be stored in one byte. **Appendix J — Five Miscellaneous CPU08 Instructions Including BCD, Divide, and CCR Operations** gives examples using the DAA instruction and the NSA instruction. Refer to **Appendix L — CPU08 BCD Example Code** for an example of BCD code.

DAA Decimal Adjust Accumulator

Operation: (A)10

Description: Adjusts the contents of the accumulator and the state of the CCR carry bit after binary coded decimal operations so that there is a correct BCD sum and an accurate carry indication. The state of the CCR half carry bit affects operation.

NSA Nibble Swap Accumulator

Operation: A ← (A[3:0]:A[7:4])

Description: Swaps upper and lower nibbles (4 bits) of the accumulator. This is used for more ef cient stor age and use of binary coded operands.

New TAP and TPA instructions

The transfer accumulator to the condition code register, TAP, and the transfer condition code register to accumulator, TPA, are new instructions to modify or manipulate the condition code register, CCR. These instructions are detailed below. Code examples can be found in **Appendix J — Five Miscellaneous CPU08 Instructions Including BCD, Divide, and CCR Operations**.

TAP Transfer Accumulator to Condition Code Register

Operation: CCR ← (A)

Description: Transfers the contents of the Accumulator to the Condition Code Register.

TPA Transfer Condition Code Register to Accumulator

Operation: A ← (CCR)

Description: Transfers the contents of the Condition Code Register to the Accumulator.

Instruction Cycle Improvements

The CPU08 instruction set not only has new instructions but many of the old instructions are faster. The CPU08 gathers data in a pipeline fashion. Instead of waiting for the instruction to be finished to gather the next opcode or operand, the CPU will fetch the next address byte during the execution of the current instruction. This pipelining overlaps execution of most instructions and thus increases the performance of the CPU08. A list of instructions that were improved is given in **Table 3**. Please refer to the CPU08 opcode map for further details.

Table 3. Instruction List (Sheet 1 of 4)

Opcode Mnemonic	Address Mode	HC05 Cycles	HC08 Cycles
ADC	IX	3	2
ADC	IX1	4	3
ADC	IX2	5	4
ADD	IX	3	2
ADD	IX1	4	3
ADD	IX2	5	4
AND	IX	3	2
AND	IX1	4	3
AND	IX2	5	4
ASR	DIR	5	4
ASR	IX	5	3
ASR	IX1	6	4
ASRA	INH	3	1
ASRX	INH	3	1
BCLR0	DIR	5	4
BCLR1	DIR	5	4
BCLR2	DIR	5	4
BCLR3	DIR	5	4
BCLR4	DIR	5	4
BCLR5	DIR	5	4
BCLR6	DIR	5	4
BCLR7	DIR	5	4
BIT	IX	3	2
BIT	IX1	4	3
BIT	IX2	5	4
BSET0	DIR	5	4
BSET1	DIR	5	4
BSET2	DIR	5	4
BSET3	DIR	5	4
BSET4	DIR	5	4
BSET5	DIR	5	4
BSET6	DIR	5	4
BSET7	DIR	5	4
BSR	REL	6	4
CLC	INH	2	1

Table 3. Instruction List (Sheet 2 of 4)

Opcode Mnemonic	Address Mode	HC05 Cycles	HC08 Cycles
CLR	DIR	5	3
CLR	IX	5	2
CLR	IX1	6	3
CLRA	INH	3	1
CLRX	INH	3	1
CMP	IX	3	2
CMP	IX1	4	3
CMP	IX2	5	4
COM	DIR	5	4
COM	IX	5	3
COM	IX1	6	4
COMA	INH	3	1
COMX	INH	3	1
CPX	IX	3	2
CPX	IX1	4	3
CPX	IX2	5	4
DEC	DIR	5	4
DEC	IX	5	3
DEC	IX1	6	4
DECA	INH	3	1
DECX	INH	3	1
EOR	IX	3	2
EOR	IX1	4	3
EOR	IX2	5	4
INC	DIR	5	4
INC	IX	5	3
INC	IX1	6	4
INCA	INH	3	1
INCX	INH	3	1
JSR	DIR	5	4
JSR	EXT	6	5
JSR	IX	5	4
JSR	IX1	6	5
JSR	IX2	7	6
LDA	IX	3	2

Table 3. Instruction List (Sheet 3 of 4)

Opcode Mnemonic	Address Mode	HC05 Cycles	HC08 Cycles
LDA	IX1	4	3
LDA	IX2	5	4
LDX	IX	3	2
LDX	IX1	4	3
LDX	IX2	5	4
LSL	DIR	5	4
LSL	IX	5	3
LSL	IX1	6	4
LSLA	INH	3	1
LSLX	INH	3	1
LSR	DIR	5	4
LSR	IX	5	3
LSR	IX1	6	4
LSRA	INH	3	1
LSRX	INH	3	1
MUL	INH	11	5
NEG	DIR	5	4
NEG	IX	5	3
NEG	IX1	6	4
NEGA	INH	3	1
NEGX	INH	3	1
NOP	INH	2	1
ORA	IX	3	2
ORA	IX1	4	3
ORA	IX2	5	4
ROL	DIR	5	4
ROL	IX	5	3
ROL	IX1	6	4
ROLA	INH	3	1
ROLX	INH	3	1
ROR	DIR	5	4
ROR	IX	5	3
ROR	IX1	6	4
RORA	INH	3	1
RORX	INH	3	1

Table 3. Instruction List (Sheet 4 of 4)

Opcode Mnemonic	Address Mode	HC05 Cycles	HC08 Cycles
RSP	INH	2	1
RTI	INH	9	7
RTS	INH	6	4
SBC	IX	3	2
SBC	IX1	4	3
SBC	IX2	5	4
SEC	INH	2	1
STA	DIR	4	3
STA	EXT	5	4
STA	IX	4	2
STA	IX1	5	3
STA	IX2	6	4
STOP	INH	2	1
STX	DIR	4	3
STX	EXT	5	4
STX	IX	4	2
STX	IX1	5	3
STX	IX2	6	4
SUB	IX	3	2
SUB	IX1	4	3
SUB	IX2	5	4
SWI	INH	10	9
TAX	INH	2	1
TST	DIR	4	3
TST	IX	4	2
TST	IX1	5	3
TSTA	INH	3	1
TSTX	INH	3	1
TXA	INH	2	1
WAIT	INH	2	1

Conclusion

This application note has covered the differences between the HC05 and the HC08 CPU architecture. Please refer to the *M68HC05 Applications Guide* for further study of the CPU05. The *CPU08 Reference Manual* is a valuable resource for studying the CPU08 in more detail.

Please consult your local Freescale sales office or your authorized Freescale distributor for applications support, literature, and specific part information.

The MCU BBS is also available with free software for use with HC05 and HC08 MCUs. The BBS number is (512) 891-3733. The code examples used in this application note can be found on the BBS. The file name is HC08OPT.ARC.

Appendix A — New CPU08 Indexing Instruction Examples

```
*************************************************************
*
*       File :  INDEX.ASM
*       Description :
*               Shows examples for new CPU08 indexing
*                instructions - AIX, CLRH, CPHX, LDHX, STHX
*               Not all addressing modes are shown.
*       Note :  Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*
*************************************************************

        ORG     $200

*****   AIX - add immediate to index register

START   LDHX    #$1010          ; H:X ← $1010
        AIX     #-10            ; H:X = $1010 + (-$10)
                                ;     = $1000

*****   CLRH - clear index high

        LDHX    #$1290          ; H:X ← $1290
        CLRH                    ; H:X ← $0090

*****   CPHX - compare 16-bit index register

        LDHX    #$1290          ; H:X ← $1290
                                ; CCR = %0110,1000
                                ; CCR before CPHX, Z=0
        CPHX    #$1290          ; H:X ← $1290
                                ; CCR = %0110,1010
                                ; CCR after CPHX, Z=1

*****   LDHX - load 16-bit index register

        LDHX    #$1290          ; H:X ← $1290

*****   STHX - store 16-bit index register

        LDHX    #$1290          ; H:X ← $1290
        STHX    $50             ; ($50) ← (H:X)
                                ; ($50) ← $12
                                ; ($51) ← $90

DONE    NOP
        BRA     DONE

*****   Initialize the reset vector
        ORG     $FFFE
        DW      START
```

Appendix B — CPU05 and CPU08 512-Byte Table Indexing Code

```
*************************************************************
*
*       File :  INDEXX.ASM
*       Description :
*               The following code illustrates the
*                different instructions used to address
*                a 512 byte table in memory.  HC05 and HC08
*                code is compared.
*       Notes:  Comments to the right of some instructions
*                give numbers.
*               CPU05 - 1st # is CPU05 cycle count
*                       2nd # is instruction byte count
*               CPU08 - 1st # is CPU08 cycle count
*                       2nd # is instruction byte count
*               Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*
*************************************************************


*       For the purpose of this example, the table address
*       will be predefined in RAM.
*       TBL_A = $120


TBL_ST0 EQU     $400            ;  start of table, section 0
TBL_ST1 EQU     TBL_ST0+256T    ;  start of table, section 1

        ORG     $50             ;  start of RAM variables
TBL_A   RMB     2               ;  address for table to be
                                ;   accessed by the code


        ORG     $200

*****   Address a 512 byte table with the index register
*****   The table starts at $400 and ends at $5FF
```

```
*********************************
*       HC05 code               *
*       CPU05 has to address the table in a section-like
*         fashion.  Section 0 is between $400 and $4FF.
*         Section 1 is between $500 and $5FF.
*       The 16-bit address is stored in RAM location TBL_A.
*         This is the offset to the table starting
*         at $400, TBL_ST0.
*       Example: Address is $520 = $400 + $120
*                TBL_A   = $01
*                TBL_A+1 = $20


START   LDX     TBL_A+1         ;3,2 X ← (TBL_A+1)
        LDA     TBL_A           ;3,2 A ← (TBL_A)
        BEQ     TBL0            ;3,2 branch to section 0 if 0
        LDA     TBL_ST1,X       ;5,3 A ← (X+TBL_ST1)
        BRA     NEXT            ;3,2 branch when done to
                                ;    the CPU08 example

TBL0    LDA     TBL_ST0,X       ;5,3 A ← (X+TBL_ST0)

*       Total # CPU05 cycles    = 17 (max)
*       Total # bytes           = 11 (max)
*********************************

*********************************
*       HC08 code               *
*       CPU08 has full 16-bit indexed addressing so the
*         table address is loaded from TBL_A in RAM.  No
*         memory table sectioning is needed.

NEXT    LDHX    TBL_A           ;4,2 H:X ← (TBL_A)
        LDA     TBL_ST0,X       ;4,3 A ← (X+TBL_ST0)

*       Total # CPU08 cycles    = 8
*       Total # bytes           = 5
*********************************


DONE    NOP
        BRA     DONE


*****   Initialize the reset vector
        ORG     $FFFE
        DW      START
```

Appendix C — New CPU08 Stack Pointer Instructions

```
*************************************************************
*
*       File :  SP.ASM
*       Description :
*               Shows examples for new CPU08 stack pointer
*                instructions - AIS, PSHA, PSHH, PSHX
*                               PULA, PULH, PULX, TSX, TXS
*               Not all addressing modes are shown.
*       Note :  Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*
*************************************************************


        ORG     $200

*****   AIS - add immediate to stack pointer
*       SP is predefined at $0FE0

START   AIS     #$1F            ; SP ← $0FE0 + $1F
                                ; SP = $0FFF

*****   PSHA - push accumulator onto stack
*       SP is predefined at $0FFF
*       A = $80

        PSHA                    ; ($0FFF) ← $80
                                ; SP ←  SP-$01
                                ; SP = $0FFE

*****   PSHH - push index register H onto stack
*       SP is predefined at $0FFE
*       H:X = $2050

        PSHH                    ; ($0FFE) ← $20
                                ; SP ←  SP-$01
                                ; SP = $0FFD

*****   PSHX - push index register X onto stack
*       SP is predefined at $0FFD
*       H:X = $2050

        PSHX                    ; ($0FFD) = $50
                                ; SP ←  SP-$01
                                ; SP = $0FFC
```

```
*****   PULX - pull index register X from stack
*       SP is predefined at $0FFC
*       $0FFD = $50
*       H:X = $0000

        PULX                    ; SP ← SP+$01
                                ; SP = $0FFD
                                ; X ← ($0FFD)
                                ; H:X = $0050

*****   PULH - pull index register H from stack
*       SP is predefined at $0FFD
*       $0FFE = $20
*       H:X = $0050

        PULH                    ; SP ← SP+$01
                                ; SP = $0FFE
                                ; H ← ($0FFE)
                                ; H:X = $2050

*****   PULA - pull accumulator from stack
*       SP is predefined at $0FFE
*       $0FFF = $80
*       A = $00

        PULA                    ; SP ← SP+$01
                                ; SP = $0FFF
                                ; A ← ($0FFF)
                                ; A = $80

*****   TSX - transfer stack pointer to index register
*       SP is predefined at $0FF5
*       H:X = $1290

        TSX                     ; H:X ← SP+$01
                                ; H:X = $0FF6

*****   TXS - transfer index register to stack pointer
*       SP is predefined at $0FF5
*       H:X = $1290

        TXS                     ; SP ← H:X-$01
                                ; SP = $128F

DONE    NOP
        BRA     DONE

*****   Initialize the reset vector
        ORG     $FFFE
        DW      START
```

Appendix D — Using the Stack in a Subroutine to Compute a Cube

```
*************************************************************
*
*       File :  CUBE.ASM
*       Description :
*               This program takes an 8-bit positive
*                number, X_IN, and cubes it.  The answer,
*                Y_IN, is in a 24-bit format.
*               This program also illustrates the
*                value of using the stack for complex
*                subroutines that use parameter passing,
*                local variables, and return values.
*       Stack Description:
*               Given below is a diagram of the stack
*                during the subroutine
*               The numbers on the right specify the
*                number of bytes above the stack pointer
*
*               -------------------------
*               SP →   ??
*                      VAR1            1
*                      VAR2            2
*                      A               3
*                      H               4
*                      X               5
*                      PC_HIGH         6
*                      PC_LOW          7
*                      Y_HIGH          8
*                      Y_MED           9
*                      Y_LOW          10
*                      X_IN           11
*               -------------------------
*
*       Note :  Please consult the CPU08 Reference Manual
*                for further HC08 instruction details
*               Code is written for educational
*                purposes only
*
*************************************************************


        ORG     $80

X_IN    RMB     1               ;8-bit number to be cubed

        ORG     $200
```

```
*****   Load up stack before entering the subroutine
*       Stack is given the 8-bit number to be cubed, X_IN
*       Next, 3 bytes must be made available to the stack
*         for the 24 bit output of the routine
*       3 pushes are made to illustrate this point

START   LDA     X_IN            ;A ← (X_IN)
        PSHA                    ;push parameter X_IN onto stack
        CLRA                    ;zero must be pushed on stack
                                ; allocation for return answer
        PSHA                    ;push Y_Low byte onto stack
        PSHA                    ;push Y_Med byte onto stack
        PSHA                    ;push Y_High byte onto stack

*****   Jump to the cube subroutine
        JSR     CUBE            ;jump sub to CUBE, Y = X_IN^3

*****   When subroutine is over, reset stack pointer to original
*       location.  Pull the answers off the stack when needed.

        AIS     #$04            ;SP ← (SP) + $04

        BRA     DONE            ;branch to the end of this
                                ;example

*****   CUBE subroutine
*****   Given X_IN, find Y = X^3

*       Save X,H, and A on stack
*       Decrement stack for 2 bytes
CUBE    PSHX                    ;push X onto stack
        PSHH                    ;push H onto stack
        PSHA                    ;push A onto stack
        AIS     #-2             ;decrement stack for local var

*       Run the math routine
*       Square X_IN, answer is X:A
        LDA     11T,SP          ;A = X_IN
        LDX     11T,SP          ;X = X_IN
        MUL                     ;X:A = (X)*(A)

*       Store away the high byte answer, X, to var1
        STX     1,SP            ;store high answ to var1

*       Multiply 16 bit result by X_IN
*       Multiply X_IN by low byte of 16-bit square
        LDX     11T,SP          ;X = X_IN
        MUL                     ;X:A = (X)*(A)
```

```
*       Store away low byte of 16-bit result
*         to Y_LOW
*       Store high byte of 16-bit result to var2
        STA     10T,SP          ;store low answ to Y_LOW
        STX     2,SP            ;store high answ to var2

*       Multiply high byte of 16-bit result by X_IN
        LDA     11T,SP          ;A ←  X_IN
        LDX     1,SP            ;load X with var1
        MUL                     ;X:A = X_IN * var1

*       Store high byte of answer to Y_HIGH
        STX     8T,SP           ;store high byte to Y_HIGH

*       ADD var2 to the low byte answer to get Y_MED
*       If there is a carry, add one bit to Y_HIGH
        ADD     2T,SP           ;A = var2 + A
        BCS     CS              ;branch if C bit set in CCR
        BRA     FIN             ;C bit is 0, branch to FIN

CS      INC     8T,SP           ;add 1 to Y_HIGH

FIN     STA     9T,SP           ;store A to Y_MED


*       Save X,H, and A on stack
*       Increment stack for 2 bytes
*       Restore X,H, and A
*       Return from the subroutine
        AIS     #$02
        PULA
        PULH
        PULX
        RTS


DONE    NOP
        BRA     DONE


*****   Initialize the reset vector
        ORG     $FFFE
        DW      START
```

Appendix E — New CPU08 MOV Instruction Examples

```
*************************************************************
*
*       File :  MOVE.ASM
*       Description :
*               Shows examples for the MOV instruction
*               All four addressing modes are illustrated
*               05 and 08 code is compared
*       Notes:  Comments to the right of some instructions
*                give numbers.
*               CPU05 - 1st # is CPU05 cycle count
*                       2nd # is instruction byte count
*               CPU08 - 1st # is CPU08 cycle count
*                       2nd # is instruction byte count
*               Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*
*************************************************************


        ORG     $200

*****   Move Immediate value to Direct memory location

*       HC05                    *
START   LDA     #$30            ;2,2 A ← $30
        STA     $80             ;4,2 ($80) ← (A)

*       HC08                    *
        MOV     #$30,$80        ;4,3 ($80) ← $30

*       Total CPU05 cycles, bytes       = 6,4
*       Total CPU08 cycles, bytes       = 4,3

*****   Move Direct mem value to Direct mem location

*       HC05                    *
        LDA     $80             ;3,2 A ← ($80)
        STA     $90             ;4,2 ($90) ← (A)


*       HC08                    *
        MOV     $80,$90         ;5,3 ($90) ← ($80)

*       Total CPU05 cycles, bytes       = 7,4
*       Total CPU08 cycles, bytes       = 5,3
```

```
*****   Move contents of Indexed to Direct mem location, post inc
        Xreg

*       HC05                    *
        LDX     #$80            ;    X ← $80
        LDA     ,X              ;3,1 A ← (X)
        STA     $90             ;4,2 ($90) ← (A)
        INCX                    ;3,1 X ← X + 1

*       HC08                    *
        LDX     #$80            ;    X ← $80
        MOV     X+,$90          ;4,2 ($90) ← (X)
                                ;    X ← X + 1

*       Total CPU05 cycles, bytes       = 10,4
*       Total CPU08 cycles, bytes       =  4,2

*****   Move Direct mem contents to Indexed location, post inc
        Xreg

*       HC05                    *
        LDX     #$90            ;    X ← $90
        LDA     $80             ;3,2 A ← ($80)
        STA     ,X              ;4,1 (X) ← (A)
        INCX                    ;3,1 X ← X + 1

*       HC08                    *
        LDX     #$90            ;    X ← $90
        MOV     $80,X+          ;4,2 (X) ← ($80)
                                ;    X ← X + 1

*       Total CPU05 cycles, bytes       = 10,4
*       Total CPU08 cycles, bytes       =  4,2


*****   Initialize the reset vector
        ORG     $FFFE
        DW      START
```

Appendix F — CPU05 and CPU08 Data Movement Code

```
*************************************************************
*
*       File :  MOVEX.ASM
*       Description :
*               A user wants to start an application one of
*                two different ways.  The user sets the
*                application on the MCU by the logic level
*                of Port D, bit 3.  Once out of reset, the
*                MCU reads Port D and moves data from ROM
*                into the RAM configuration registers
*                according to the logic level of bit 3.
*       Notes:  Comments to the right of some instructions
*                give numbers.
*               CPU05 - 1st # is CPU05 cycle count
*                       2nd # is instruction byte count
*               CPU08 - 1st # is CPU08 cycle count
*                       2nd # is instruction byte count
*               Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*
*************************************************************

*       For the purpose of this example, we will be using
*       random ctrl registers for the code.  They are listed
*       below in an equate table

TBL     EQU     $1000           ; start of table
PORTD   EQU     $03             ; port D data register
PORTADR EQU     $04            ; port A data direction register
PORTBDR EQU     $05            ; port B data direction register
SPICTRL EQU     $0A             ; SPI control register
SCICTRL EQU     $0E             ; SCI control register
TIMCTRL EQU     $12             ; Timer control register

        ORG     $200

*****   If bit 3 = 0 when read, then the table
*       starts at $1000
*       If bit 3 = 1 when read, then the table
*       starts at $1008

*********************************
*       HC05 code               *

START05 LDA     PORTD           ;3,2 A ← (PORTD)
        AND     #$08            ;2,2 clear A except bit 3
                                ;    A = 0 or 8
        TAX                     ;2,1 X ← (A)
                                ;    set the offset of X
```

```
        LDA     TBL,X           ;5,3 A ← (TBL+X)
        STA     PORTADR         ;4,2 (PORTADR) ← (A)
        INCX                    ;3,1 X ← X + 1
        LDA     TBL,X           ;5,3 A ← (TBL+X)
        STA     PORTBDR         ;4,2 (PORTBDR) ← (A)
        INCX                    ;3,1 X ← X + 1
        LDA     TBL,X           ;5,3 A ← (TBL+X)
        STA     SPICTRL         ;4,2 (SPICTRL) ← (A)
        INCX                    ;3,1 X ← X + 1
        LDA     TBL,X           ;5,3 A ← (TBL+X)
        STA     SCICTRL         ;4,2 (SCICTRL) ← (A)
        INCX                    ;3,1 X ← X + 1
        LDA     TBL,X           ;5,3 A ← (TBL+X)
        STA     TIMCTRL         ;4,2 (TIMCTRL) ← (A)

*       Total # CPU05 cycles    = 64
*       Total # bytes           = 34
*********************************

*********************************
*       HC08 code               *

START08 LDHX    #TBL            ;3,3 H:X ← TBL
        LDA     PORTD           ;3,2 A ← (PORTD)
        AND     #$08            ;2,2 clear A except bit 3
                                ;    A = 0 or 8
        TAX                     ;1,1 X ← (A)
                                ;    set the offset of X

        MOV     X+,PORTADR      ;4,2 (PORTADR) ← (H:X)
                                ;    X ← X + 1
        MOV     X+,PORTBDR      ;4,2 (PORTBDR) ← (H:X)
                                ;    X ← X + 1
        MOV     X+,SPICTRL      ;4,2 (SPICTRL) ← (H:X)
                                ;    X ← X + 1
        MOV     X+,SCICTRL      ;4,2 (SCICTRL) ← (H:X)
                                ;    X ← X + 1
        MOV     X+,TIMCTRL      ;4,2 (TIMCTRL) ← (H:X)
                                ;    X ← X + 1

*       Total # CPU08 cycles    = 29
*       Total # bytes           = 18
*********************************

DONE    NOP
        BRA     DONE

*****   Initialize the reset vector
        ORG     $FFFE
        DW      START05
```

Appendix G — New Branch Instruction Examples

```
*************************************************************
*
*       File :  BRANCH.ASM
*       Description :
*               Shows examples for new CPU08 branch
*               instructions - CBEQ, CBEQA, CBEQX
*                              DBNZ, DBNZA, DBNZX
*       Notes:  Comments to the right of some instructions
*                give numbers.
*               CPU05 - 1st # is CPU05 cycle count
*                       2nd # is instruction byte count
*               CPU08 - 1st # is CPU08 cycle count
*                       2nd # is instruction byte count
*               Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*
*************************************************************

        ORG     $200

*****   CBEQ - compare and branch if equal, direct
*       A is predefined at $40
*       Memory location $80 contains $40

*       HC05 code               *
LPA     CMP     $80             ;3,2 (A) - ($80)
        BEQ     LP1             ;3,2 if (A) = ($80) then
                                ;    branch to LP1
        BRA     LPA             ;    go to LPA, try again!

*       HC08 code               *
LP1     CBEQ    $80,LPB         ;5,3 if (A)-($80)=0,
                                ;    then branch to LPB
        BRA     LP1             ;    go to LP1

*       Total CPU05 cycles, bytes       = 6,4
*       Total CPU08 cycles, bytes       = 5,3

*****   CBEQA - compare and branch if equal, immediate
*       A is predefined at $50

*       HC05 code               *
LPB     CMP     #$50            ;2,2 (A) - $50
        BEQ     LP2             ;3,2 if (A) = $50, then LP2
        BRA     LPB             ;    go to LPB
```

```
*       HC08 code               *
LP2     CBEQA   #$50,LPC        ;4,3 if #$50 = (A), then LPC
        BRA     LP2             ;    go to LP2

*       Total CPU05 cycles, bytes       = 5,4
*       Total CPU08 cycles, bytes       = 4,3


*****   CBEQX - compare and branch if equal, index
*       Index register X is predefined at $60

*       HC05 code               *
LPC     CPX     #$60            ;2,2 X - $60
        BEQ     LP3             ;3,2 if X = $60, then LP3
        BRA     LPC             ;    go to LPC

*       HC08 code               *
LP3     CBEQX   #$60,LPD        ;4,3 if X = $60, then LPD
        BRA     LP3             ;    go to LP3

*       Total CPU05 cycles, bytes       = 5,4
*       Total CPU08 cycles, bytes       = 4,3


*****   DBNZ - decrement and branch if not zero

*       HC05 code               *
*       Memory location $A0 is predefined at $08
LPD     NOP                     ;    used here to represent any
                                ;    number of instructions
        DEC     $A0             ;5,2 decrement ($A0)
        BNE     LPD             ;3,2 if ($A0) not zero, then LPD

*       HC08 code               *
*       Memory location $A0 is predefined at $08
LP4     NOP                     ;    used here to represent any
                                ;    number of instructions
        DBNZ    $A0,LP4         ;5,3 ($A0) = ($A0) - 1
                                ;    if ($A0) not zero, then LP4

*       Total CPU05 cycles, bytes       = 8,4
*       Total CPU08 cycles, bytes       = 5,3
```

```
*****   DBNZA - decrement acca and branch if not zero

*       HC05 code               *
*       A is predefined at $06
LPE     NOP                     ;    used here to represent any
                                ;    number of instructions
        DECA                    ;3,1 (A) = (A) - 1
        BNE     LPE             ;3,2 if (A) not zero, then LPE


*       HC08 code               *
*       A is predefined at $06
LP5     NOP                     ;    used here to represent any
                                ;    number of instructions
        DBNZA   LP5             ;3,2 (A) = (A) - 1
                                ;    if (A) not zero, then LP5

*       Total CPU05 cycles, bytes       = 6,3
*       Total CPU08 cycles, bytes       = 3,2


*****   DBNZX - decrement x and branch if not zero

*       HC05 code               *
*       Index register X is predefined at $04
LPF     NOP                     ;    used here to represent any
                                ;    number of instructions
        DECX                    ;3,1 (X) = (X) - 1
        BNE     LPF             ;3,2 if (X) not zero, then LPF

*       HC08 code               *
*       Index register X is predefined at $04
LP6     NOP                     ;    used here to represent any
                                ;    number of instructions
        DBNZX   LP6             ;3,2 (X) = (X) - 1
                                ;    if (X) not zero, then LP6

*       Total CPU05 cycles, bytes       = 6,3
*       Total CPU08 cycles, bytes       = 3,2


DONE    NOP
        BRA     DONE


*****   Initialize the reset vector
        ORG     $FFFE
        DW      LPA
```

Appendix H — CPU05 and CPU08 Search Code

```
*************************************************************
*
*       File :  BRANCHX.ASM
*       Description :
*               This code shows an example of using branch
*                algorithms to search for a number in a
*                table.  The code will search for $FF in
*                a table.  This would signify that in a
*                table of A/D values, an A/D reading
*                was saturated.
*       Notes:  Comments to the right of some instructions
*                give numbers.
*               CPU05 - 1st # is CPU05 cycle count
*                       2nd # is instruction byte count
*               CPU08 - 1st # is CPU08 cycle count
*                       2nd # is instruction byte count
*               Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*
*************************************************************


TABLE   EQU     $400            ; starting address of the
                                ;  A/D table


        ORG     $50
TBL_LEN RMB     1               ; memory value containing
                                ; the number of values in
                                ; a the A/D table


        ORG     $200

*****   Search for $FF (saturation) in a table of A/D values
*         TBL_LEN is predefined at $08 for this example
*         Therefore the table is defined from $400 to $407
*       The values given for the total # of cycles and bytes
*         reflect an absolute count with no looping involved
*       An accurate account of the cycle count would involve
*         the table length and whether or not a comparison
*         was made.
```

```
*       HC05 code                   *
SRCH    LDX     TBL_LEN             ;3,2 X ← (TBL_LEN)
LOOP3   LDA     TABLE-1,X           ;5,3 A ← (TABLE-1+X)
        CMP     #$FF                ;2,2 (A) - $FF
        BEQ     NEXT                ;3,2 if Z=1, then goto NEXT
                                    ;    this signifies that a
                                    ;    saturation value has been
                                    ;    found in the table
        DECX                        ;3,1 X ← X -1
        BNE     LOOP3               ;3,2 if Z=0, then goto LOOP3
                                    ;    go look at another value
                                    ;    in the table

*       Total # CPU05 cycles    = 19
*       Total # bytes           = 12


*       HC08 code                   *
NEXT    LDX     TBL_LEN             ;3,2 X ← (TBL_LEN)
LOOP4   LDA     TABLE-1,X           ;4,3 A ← (TABLE-1+X)
        CBEQA   #$FF,DONE           ;4,3 (A) - $FF
                                    ;    if Z=1, then goto DONE
                                    ;    this signifies that a
                                    ;    saturation value has been
                                    ;    found in the table
        DBNZX   LOOP4               ;3,2 X ← X -1
                                    ;    if Z=0, then goto LOOP4
                                    ;    go look at another value
                                    ;    in the table

*       Total # CPU08 cycles    = 14
*       Total # bytes           = 10


DONE    NOP
        BRA     DONE

*****   Initialize the reset vector
        ORG     $FFFE
        DW      SRCH
```

Appendix I — New CPU08 Signed Branch Instruction Examples

```
*************************************************************
*
*       File :  SIGNBRA.ASM
*       Description :
*               Shows examples for new CPU08 signed branch
*                instructions - BGE, BGT, BLE, BLT
*               The examples demonstrate two's complement
*                math with branching.
*       Note :  Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*
*************************************************************

        ORG     $200

*****   BGE - branch if greater than or equal
*       A is predefined at $FF

LP_BGE  CMP     #$FF            ; (A) - $FF, [ -1 - (-1) ]
        BGE     LP_BGT          ; if A >= $FF, then
                                ; branch to LP_BGT
        BRA     LP_BGE          ; go to LP_BGE

*****   BGT - branch if greater than
*       A is predefined at $07

LP_BGT  CMP     #$FF            ; (A) - $FF, [ 7 - (-1) ]
        BGT     LP_BLE          ; if A > $FF, then
                                ; branch to LP_BLE
        BRA     LP_BGT          ; go to LP_BGT

*****   BLE - branch if less than or equal
*       A is predefined at $FF

LP_BLE  CMP     #$FF            ; (A) - $FF, [ -1 - (-1) ]
        BLE     LP_BLT          ; if A <= $FF, then
                                ; branch to LP_BLT
        BRA     LP_BLE          ; go to LP_BLE

*****   BLT - branch if less than
*       A is predefined at $FF

LP_BLT  CMP     #$07            ; $FF - $07, [ -1 - (7) ]
        BLT     DONE            ; if A < $FF, then
                                ; branch to DONE
        BRA     LP_BLT          ; go to LP_BLT

DONE    NOP
        BRA     DONE


*****   Initialize the reset vector
        ORG     $FFFE
        DW      LP_BGE
```

Appendix J — Five Miscellaneous CPU08 Instructions Including BCD, Divide, and CCR Operations

```
*************************************************************
*
*       File :  MISCINST.ASM
*       Description :
*               Shows examples for 5 misc CPU08 instructions
*               that include BCD, Divide, and CCR operations
*               They are DAA, NSA, DIV, TAP, TPA
*       Notes:  Comments to the right of some instructions
*                give numbers.
*               CPU05 - 1st # is CPU05 cycle count
*                       2nd # is instruction byte count
*               CPU08 - 1st # is CPU08 cycle count
*                       2nd # is instruction byte count
*               Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*
*************************************************************

        ORG     $200

*****   DAA - decimal adjust accumulator

START   LDA     #$26            ; A ← $26, a BCD #
        ADD     #$37            ; A ← $37 + (A)
                                ; A = $5D, a hex #
        DAA                     ; (A) = 63 = (26 + 37)
                                ; the hex #, 5D, has been
                                ; adjusted to the BCD #, 63

*****   NSA - nibble swap accumulator
*       A is predefined at $37
*       When finished A will be at $73

*       HC05 code               *
        TAX                     ;2,1 X ← (A)
        ROLX                    ;3,1 rotate left X
        ROLA                    ;3,1 rotate left A
        ROLX                    ;3,1 rotate left X
        ROLA                    ;3,1 rotate left A
        ROLX                    ;3,1 rotate left X
        ROLA                    ;3,1 rotate left A
        ROLX                    ;3,1 rotate left X
        ROLA                    ;3,1 rotate left A
```

```
*       HC08 code                *
        NSA                      ;3,1 swap the nibbles of A

*       Total CPU05 cycles, bytes       = 26,9
*       Total CPU08 cycles, bytes       = 3,1

*****   DIV - divide 16 bit by 8 bit
*     The immediate addressing mode is used to load the registers
*       to illustrate the components needed to execute
*       a DIV instruction.

        LDHX    #$0200           ; H ← $02
        LDX     #$80             ; X ← $80
        LDA     #$00             ; A ← $00
        DIV                      ; H:A / X = A rem H
                                 ; Answer is $04 rem 0


*****   TAP - transfer accumulator to ccr
*       A is predefined at $E2
*       CCR = %0110,0000

        TAP                      ; CCR ← (A)
                                 ; CCR = %1110,0010


*****   TPA - transfer ccr to accumulator
*       A is predefined at $00
*       CCR = %1110,0010

        TPA                      ; A ← (CCR)
                                 ; A = $E2

DONE    NOP
        BRA     DONE


*****   Initialize the reset vector
        ORG     $FFFE
        DW      START
```

Appendix K — CPU08 Averaging Code

```
*************************************************************
*       File :  AVERAGE.ASM
*       Description :
*               This code demonstrates an average routine
*                showing the use of the CPU08's DIV inst.
*               8-bit values are read from a table in
*                memory and the average of those numbers
*                is computed.
*       Notes:  Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*************************************************************

TBL_STR EQU     $400            ; starting address of
                                ; table in memory

        ORG     $50
LENGTH  RMB     1               ; length of table
TOT_H   RMB     1               ; high byte of total
TOT_L   RMB     1               ; low byte of total

        ORG     $200

*       The length of this table is predefined as 3 for
*        this example
*       The values in the table start at $401
*       $401 = 50
*       $402 = 60
*       $403 = 70

START   CLR     TOT_H           ; clear TOT_H in mem
        CLR     TOT_L           ; clear TOT_L in mem
        LDX     LENGTH          ; X ← length of table
NEXT    LDA     TBL_STR,X       ; A ←(X+TBL_STR)
        ADD     TOT_L           ; A ← (A)+(TOT_L)
        STA     TOT_L           ; TOT_L ← (A)
        BCS     CS              ; if carry bit is set,
                                ; branch to CS
        BRA     NEXT2           ; branch to next table entry

CS      INC     TOT_H           ; inc the high byte of total

NEXT2   DBNZX   NEXT            ; dec X, if X not 0, then
                                ;   branch to next table entry
        LDHX    TOT_H           ; H ← high byte of dividend
        TXA                     ; A ← low byte of dividend
        LDX     LENGTH          ; X ← load divisor
        DIV                     ; H:A / X
                                ; answer in A with H rem

*       Answer can be found in A, remainder in H
*       Answer is equal to $60 with no remainder

DONE    NOP
        BRA     DONE

*****   Initialize the reset vector
        ORG     $FFFE
        DW      START
```

Appendix L — CPU08 BCD Example Code

```
*************************************************************
*
*       File :  BCD.ASM
*       Description :
*               This code demonstrates a BCD routine to
*                be used on the CPU08.
*               Two 16-bit BCD numbers are added together
*                and the result is 16-bit BCD number
*               BCD1 + BCD2 = BCDT
*       Notes:  Please consult the CPU08 Reference Manual
*                for further details on these instructions
*               Code is written for educational
*                purposes only
*
*************************************************************

        ORG     $50
BCD1_H  RMB     1               ; high byte of bcd #1
BCD1_L  RMB     1               ; low byte of bcd #1
BCD2_H  RMB     1               ; high byte of bcd #2
BCD2_L  RMB     1               ; low byte of bcd #2
BCDT_H  RMB     1               ; high byte of bcd total
BCDT_L  RMB     1               ; low byte of bcd total

        ORG     $200

*       Predefine values for the example
*       BCD1 = 150,   BCD1_H = 01 & BCD1_L = 50
*       BCD2 = 250,   BCD2_H = 02 & BCD2_L = 50

*       First, add the low bytes of the 16-bit BCD #s
START   LDA     BCD1_L          ; A ← (bcd #1 low byte)
        ADD     BCD2_L          ; A ← (A)+(bcd #2
                                ;  low byte)
        DAA                     ; decimal adjust accumulator
        STA     BCDT_L          ; store away result to total low

*       Second, add the high bytes of the 16-bit BCD #s
*       Add the carry bit from the previous addition
        LDA     BCD1_H          ; A ← (bcd #1 high byte)
        ADC     BCD2_H          ; A ← (A)+(bcd #2
                                ;  high byte)+C
        DAA                     ; decimal adjust accumulator
        STA     BCDT_H          ; store away result to total high

*       Answer is in BCDT_H and BCDT_L
*       BCDT_H = 04
*       BCDT_L = 00

DONE    NOP
        BRA     DONE

*****   Initialize the reset vector
        ORG     $FFFE
        DW      START
```

Application Note

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