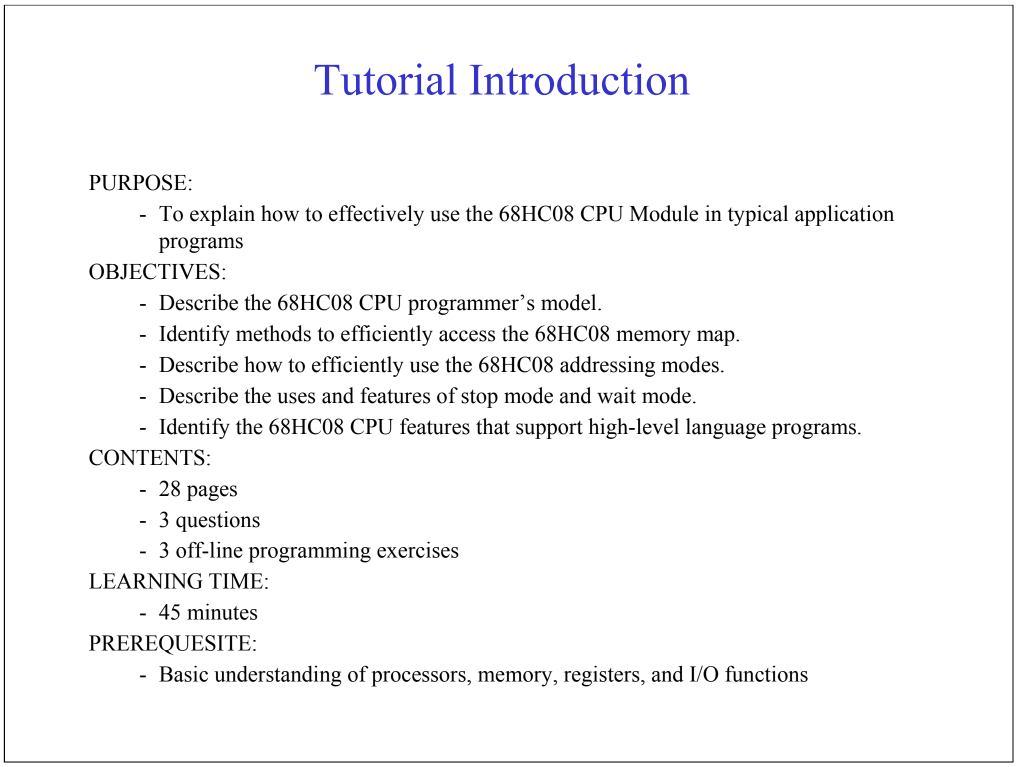


Tutorial Introduction

PURPOSE:
- To explain how to effectively use the 68HC08 CPU Module in typical application programs

OBJECTIVES:
- Describe the 68HC08 CPU programmer's model.
- Identify methods to efficiently access the 68HC08 memory map.
- Describe how to efficiently use the 68HC08 addressing modes.
- Describe the uses and features of stop mode and wait mode.
- Identify the 68HC08 CPU features that support high-level language programs.

CONTENTS:
- 28 pages
- 3 questions
- 3 off-line programming exercises

LEARNING TIME:
- 45 minutes

PREREQUESITE:
- Basic understanding of processors, memory, registers, and I/O functions

Welcome to this tutorial on the 68HC08 CPU. This tutorial describes the features and configuration of the CPU. Please note that on subsequent pages, you will find reference buttons in the upper right of the content frame that access additional content.

Upon completion of this tutorial, you'll be able to describe the 68HC08 CPU programmer's model and the features that support high-level language programs. You'll also be able to write efficient code using memory mapping and the 68HC08 CPU instruction set and addressing modes.

This tutorial assumes that you have a basic understanding of processors, memory, registers, and I/O functions. Click the Forward arrow when you're ready to begin the tutorial.

68HC08 CPU Architecture

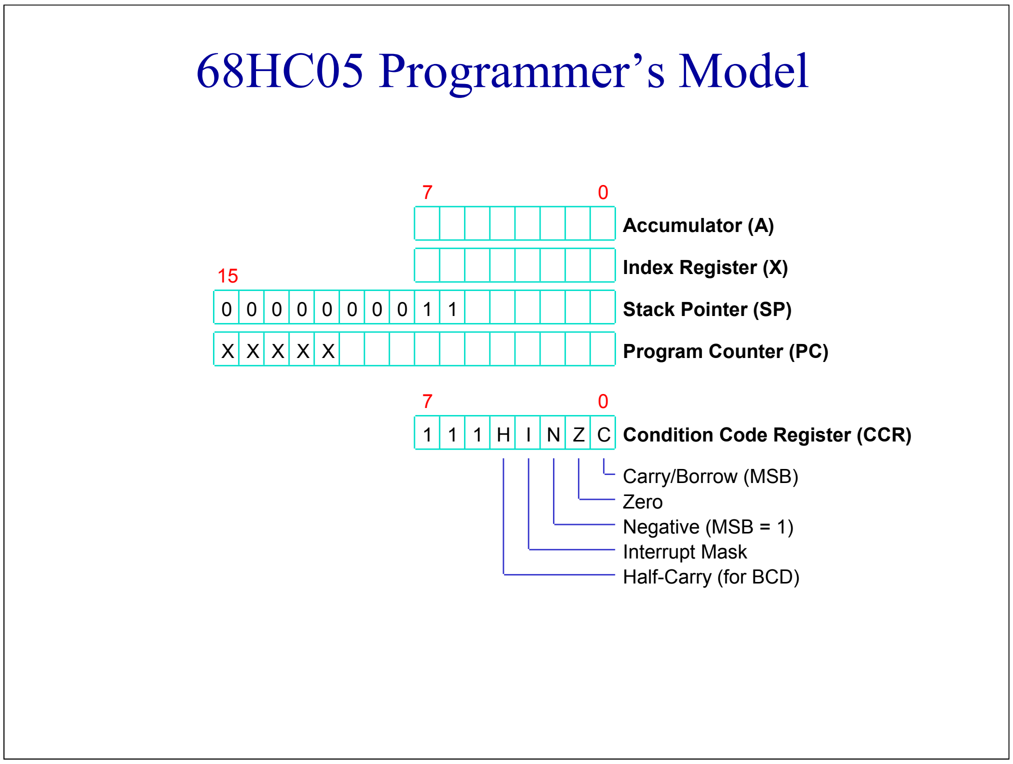
- Based on 68HC05 architecture
- Memory mapped registers and peripherals
- Enhanced instruction set
- Enhanced addressing modes:
  – New indexed addressing modes
  – Stack pointer relative addressing modes
  – Memory to memory addressing modes
- Low-power modes
  – WAIT mode
  – STOP mode

Let’s begin this tutorial with an overview of the 68HC08 CPU architecture. The 68HC08 CPU is based on the 68HC05 CPU architecture. Like the 68HC05 CPU, the 68HC08 CPU maps all registers and peripherals into a single linear address space, providing efficient memory access.

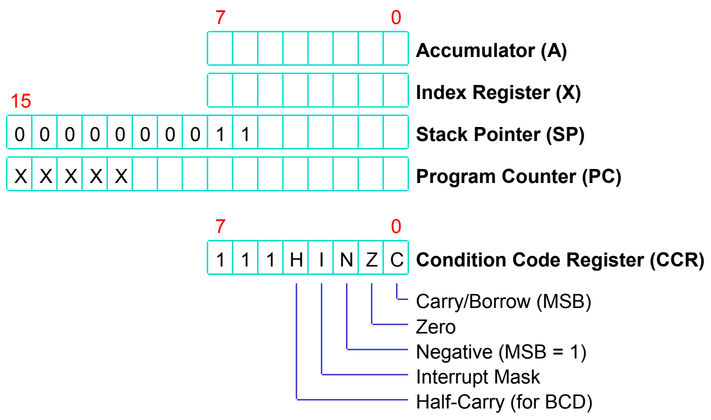
The 68HC08 CPU instruction set extends the 68HC05 instructions for data movement, data manipulation, and branching and control logic. In addition, the 68HC08 model adds several new addressing modes for a total of sixteen addressing modes.

The 68HC08 CPU has two low-power modes, WAIT mode and STOP mode. Later in the tutorial, we’ll discuss the uses and features of these two modes. We’ll also examine a number of features included in the 68HC08 CPU for efficiently implementing code written in a high-level language.

Next, let’s take a closer look at the 68HC08 CPU architecture with a review of the programmer’s model.



68HC05 Programmer's Model



Recall that the 68HC08 CPU is based on the 68HC05 CPU. We'll first look at the 68HC05 model and then show the evolution to the 68HC08 model.

The programmer's model describes what the 68HC05 CPU looks like to the assembly language programmer. This simple model shows a practical, 8-bit accumulator-based, Von Neumann architecture. It includes an accumulator, an index register, a stack pointer, a program counter, and a condition code register. The 68HC05 CPU is very efficient in code size and execution.

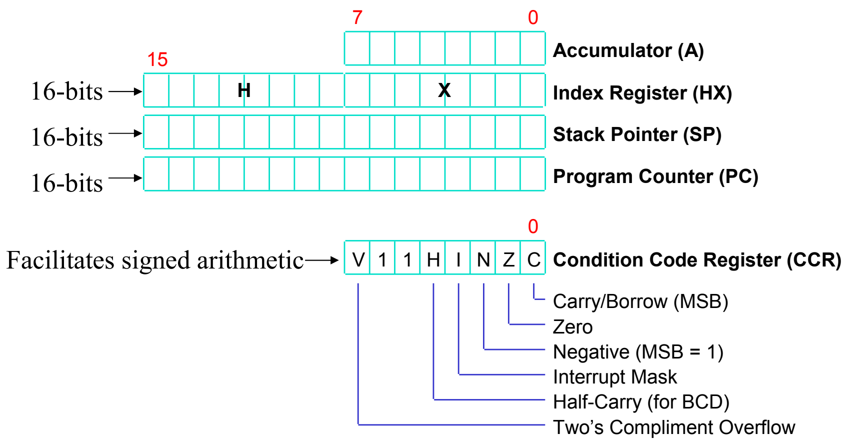
Note that every 68HC05 CPU uses this model. This greatly simplifies supporting existing code and transporting code between projects that may use different variants of the 68HC05 family. This scheme is often referred to as "design reuse" and represents a great advantage over competing architectures that don't maintain this level of compatibility between MCU families.

Note that the stack pointer size is only 6-bits, limiting the available stack space to 64 bytes. Using 2 bytes per subroutine call, and 5 bytes per interrupt, the stack size is sufficient to accommodate most applications.

Upon power-on reset, the 68HC05 CPU will reset the stack pointer to $00FF. Later, we'll compare this behavior to the 68HC08 model.

Although the program counter is defined as 16-bits wide, it is optimized to satisfy only the address space implemented on the particular MCU. There's no need to have the program counter point to an address that is beyond the available memory map.

68HC08 Programmer’s Model



The 68HC08 CPU programmer's model is an evolutionary extension of the 68HC05 model. Note that every 68HC08 CPU, regardless of MCU size or feature set, implements this CPU model.

Let’s review the specific enhancements that are included in the 68HC08 architecture.

Note that the index register has been extended to 16-bits, which helps to more efficiently handle table or data structures that are larger than 256 bytes.

The stack pointer and the program counter are both defined and implemented as 16-bit registers, regardless of the available memory on-chip. We’ll discuss the stack in more detail later.

Upon power-on reset, the 68HC08 CPU looks exactly like the 68HC05. During reset, the upper 8-bits of the 16-bit HX index register are cleared and the stack pointer is initialized to $00FF, as in the 68HC05. Considering the extra RAM available on many 68HC08s, it is likely that the user will relocate the stack pointer. However, this feature helps to maintain operational compatibility with existing 68HC05 software.

In the 68HC08 CPU, the V-bit in the condition code register facilitates signed arithmetic calculations. This enhancement allows assembly writers and compilers to perform addressing calculations more efficiently.

68HC05/08 Code Compatibility

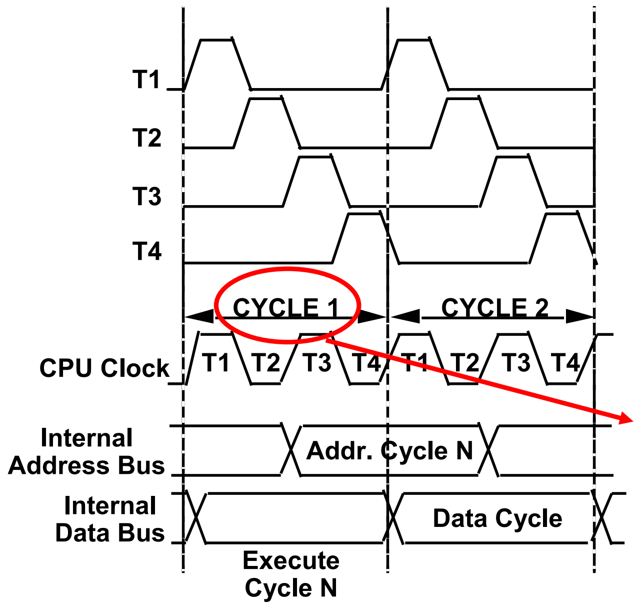
- Object code is upward compatible from 68HC05 to 68HC08
  - Identical binary opcode patterns
  - State machine, mathematical, and flow control algorithms port transparently
- Important differences to consider:
  - Addressing and peripheral differences
  - 68HC08 is much faster than 68HC05
  - 68HC08 includes additional instructions and addressing modes

The 68HC08 CPU is object code compatible with the 68HC05 CPU. It uses the same binary opcode patterns. Note that you don't need 68HC05 source code to port to the 68HC08 CPU. You can simply dump the existing 68HC05 S-Record and expect the software to execute correctly.

There are two important differences between 68HC05 and 68HC08 MCUs that you must consider when porting code. First, there are some address and peripheral differences but this should amount to a small percentage of the code. The important state machine, mathematical, and flow control algorithms port transparently. Second, the 68HC08 will run 68HC05 code about 25% faster, running at the same bus speed and the 68HC08 can typically run at a bus speed four times faster than the 68HC05. Together this averages 5X faster performance running 68HC05 code. Since the 68HC08 is much faster than the 68HC05, you may need to update tight timing loops implemented in 68HC05 software.

The 68HC08 CPU includes an enhanced instruction set and addressing modes that can provide even larger performance gains. Later in the tutorial, we'll take a more detailed look at these features and how they can be utilized to increase performance with smaller code size.

68HC08 Internal Timing



- Four phase internal clock
- Bus frequency = clock reference / 4
- All instructions specified in bus cycles
- Most 68HC08s include PLL to generate clock reference

bus frequency = 32 Mhz / 4 = 8 MHz
instruction cycle = 125 nsec bus cycle

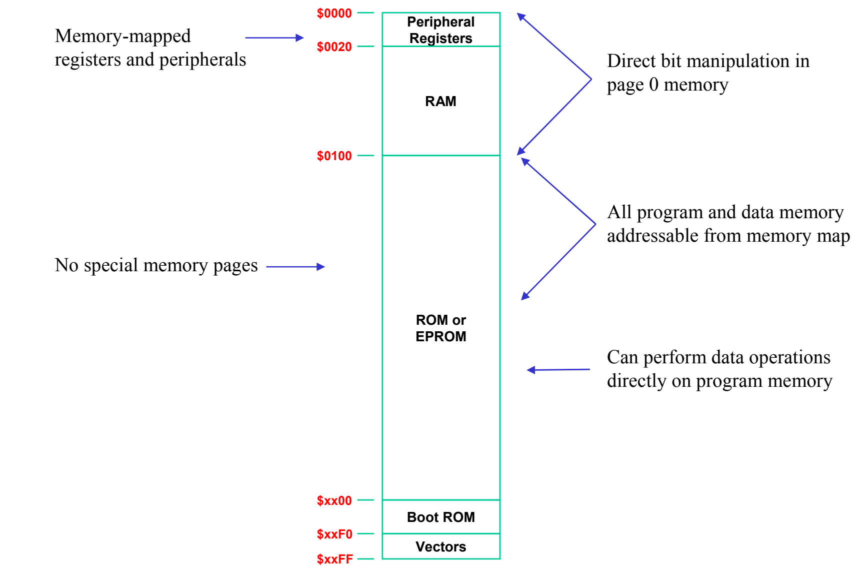
Next, let's take a closer look at the 68HC08 CPU internal timing.

The 68HC08 utilizes a four phase internal clocking scheme. A 68HC08 CPU execution cycle is made up of these four phases. If the 68HC08 is being driven by a crystal, the CPU execution cycle is one fourth the crystal frequency. This is referred to as the bus cycle. In the 68HC08, all instruction timing is specified in bus cycles.

For example, a 32 MHz clock input would result in an 8 MHz bus frequency. A single cycle instruction would execute in 125 ns, or 1 divided by 8 MHz.

Most 68HC08s include an on-chip phase locked loop (PLL) circuit which takes a lower speed crystal and internally generates the four phases at much higher clock frequencies than the crystal. For more information about clock generation and PLL circuits, see the Clock Generation Module (CGM) tutorial.

68HC05 Memory Organization


Memory-mapped registers and peripherals
No special memory pages
$0000
$0020
$0100
$xx00
$xxF0
$xxFF
Peripheral Registers
RAM
ROM or EPROM
Boot ROM
Vectors
Direct bit manipulation in page 0 memory
All program and data memory addressable from memory map
Can perform data operations directly on program memory

Now, let's take a detailed look at the 68HC05 memory map. All 68HC05 family members use this memory model.

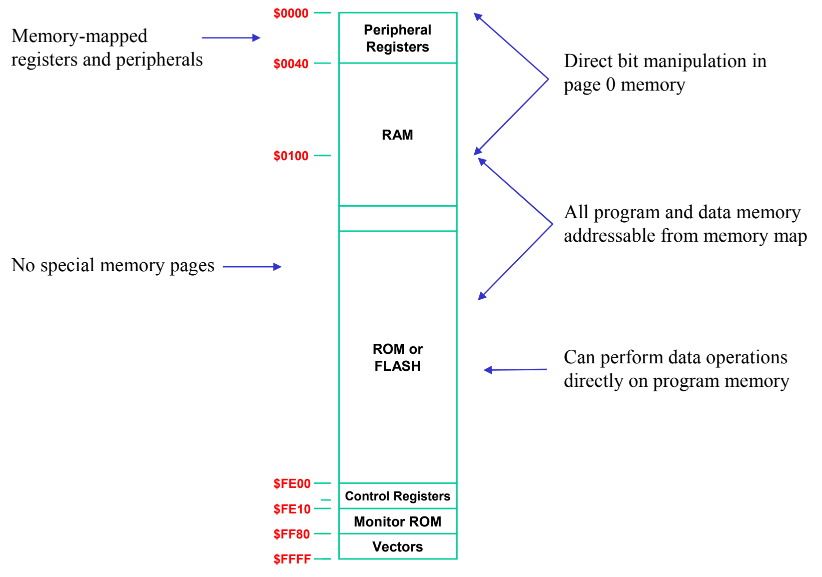
All storage and input/output resources are memory mapped into a single linear address space. The memory map is organized as bytes, beginning at $0000 with a variable end location ($xxFF) depending on factors such as ROM or EPROM array size. The address space on a 68HC05 device is usually sized just large enough to contain the integrated ROM or EPROM, RAM, and control registers. For example, the MC68HC705KJ1 has a 2-kbyte memory map containing 1240 bytes of EPROM, 64 bytes of RAM, and 14 bytes of to support other processor accessible resources.

I/O registers are memory mapped starting at address $0. No special I/O instructions are required, allowing easy and very efficient addressing of peripherals. The I/O registers reside in direct page which is the first 256 bytes of memory. Direct page is just a naming convention as the address space is not paged. Directly addressing locations in the first 256 bytes is a very efficient means of accessing I/O peripherals since only an 8-bit address operand is required. The 68HC05s ability to directly manipulate any bit in direct page also contributes to efficient I/O processing.

Typically, RAM begins in direct page immediately after the I/O registers. ROM or EPROM usually starts right after the RAM. While RAM is typically used for data and ROM or EPROM is typically used for program code, the 68HC05 makes no distinction. Because all memory types are addressable from the same memory map, you can execute code out of RAM. You can also store data in ROM or EPROM without having to use inefficient table instructions to return the data from code space back to RAM for processing. If you want to process data stored in EPROM, you can access the data as you would in RAM.

The 68HC05 is also a big-endian machine. A 16-bit piece of data, or word, is stored in memory with the high byte at address N and the low byte at address N+1. This byte order applies whether the word is part of an assembled instruction, such as an index register offset, a return address residing on the stack, or the address of a service routine contained in an interrupt vector. This can provide easier debugging of assembly language programs.

68HC08 Memory Organization


Memory-mapped registers and peripherals
No special memory pages
$0000
$0040
$0100
$FE00
$FE10
$FF80
$FFFF
Peripheral Registers
RAM
ROM or FLASH
Control Registers
Monitor ROM
Vectors
Direct bit manipulation in page 0 memory
All program and data memory addressable from memory map
Can perform data operations directly on program memory

The figure shows the memory map of the 68HC08.

The 68HC08 memory map is identical in employment to the 68HC05, except that it is always implemented in 64 kbytes regardless of the available memory sizes. In the 68HC08, the peripheral space is larger (64 bytes) with RAM following immediately after. But the ROM or FLASH actually occupies the upper memory boundary preceding $FE00. This leaves a “hole” in the middle of the memory map between the RAM and the ROM/FLASH which are trapped by the illegal address checking.

All 68HC08 Family members use this model, though some models have a different start and end addresses for specific areas like monitor ROM or control registers.

68HC05 Instruction Set

Data Movement TXA	LDA, LDX, STA, STX, TAX,
Arithmetic	ADD, ADC, SUB, SUBC, MUL
Data Manipulation (Inc/Dec/Neg/Clr)	INCA, INCX, INC, DECA, DECX, DEC, CLR,NEGA, NEGX, NEG
Data Manipulation (Rotate/Shift)	ROLA, ROLX, ROL, RORA, RORX, ROR, LSLA, LSLX, LSL, LSRA, LSRX, LSR, ASRA, ASRX, ASR
Data Manipulation	BSET, BCLR
Logic	AND, ORA, EOR, COMA, COMX, COM
Data Test	CMP, CPX, BIT, TSTA,TSTX, TST, BRCLR, BRSET
Branch BNE, BIH	BRA, BRN, BSR, BHI, BLO, BHS, BLS, BPL, BMI, BEQ, BCC, BCS, BHC, BHCC, BHCS, BMC, BMS, BIL,
Jump/Return	JMP, JSR, RTS
Control STOP	SEC, CLC, SEI, CLI, SWI, RTI, RSP, NOP, WAIT,

Now that we've reviewed the CPU programmer's model and memory map, let's look next at the CPU instruction set. The table shows the 68HC05 instruction set. The 68HC05 has a total of 85 instructions that form a basic yet functional and powerful instruction set. The instruction set can be grouped into ten groups by instruction operation.

The data movement instructions include load, store, and transfer instructions.

The arithmetic instructions include a single byte 8 by 8 multiply instruction.

There are data manipulation instructions that increment, decrement, negate, or clear. In addition, there are instructions that rotate or shift either logically or arithmetically.

The instruction set includes many logic and data test instructions. The logic instructions include and, or, and complement instructions. The data test instructions enable you to compare or test any byte in the memory map, or test any bit in direct page, and even branch if a bit is set or cleared.

In addition there is a wide range of relative branching and control instructions. Absolute jump and return instructions enable you to direct the program flow to any location in the 64 kbyte map without paging. The control instructions work with the condition code register, exceptions, and special CPU low-power modes.

The 68HC08 instruction set builds on this base.

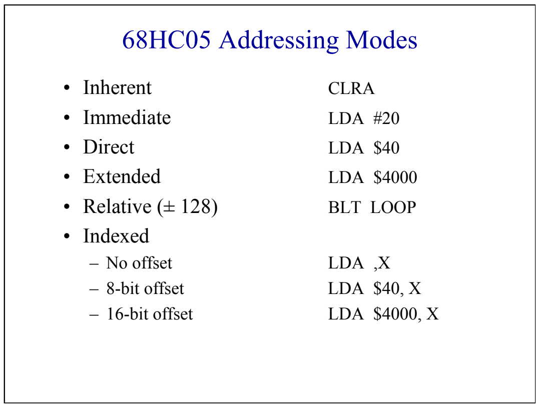
68HC08 Instruction Set

Data Movement PSHH,	LDA, LDX, STA, STX, TAX, TXA, LDHX, MOV, PSHA, PSHX, PULA, PULH, PULX, STHX
Arithmetic	ADD, ADC, SUB, SUBC, MUL, DAA, DIV
Data Manipulation (Inc/Dec/Neg/Clr)	INCA, INCX, INC, DECA, DECX, DEC, CLR,NEGA, NEGX, NEG AIS, AIX, CLRH
Data Manipulation (Rotate/Shift)	ROLA, ROLX, ROL, RORA, RORX, ROR, LSLA, LSLX, LSL, LSRA, LSRX, LSR, ASRA, ASRX, ASR
Data Manipulation	BSET, BCLR
Logic	AND, ORA, EOR, COMA, COMX, COM, NSA
Data Test	CMP, CPX, BIT, TSTA,TSTX, TST, BRCLR, BRSET, CPHX
Branch BNE, BIH,	BRA, BRN, BSR, BHI, BLO, BHS, BLS, BPL, BMI, BEQ, BCC, BCS, BHC, BHCC, BHCS, BMC, BMS, BIL, BGE, BGT, BLE, BLT, CBEQ, CBEQA, CBEQX, DBNZ
Jump/Return	JMP, JSR, RTS
Control STOP	SEC, CLC, SEI, CLI, SWI, RTI, RSP, NOP, WAIT, TAP, TPA, TSX, TXS

The 68HC08 CPU adds 28 instructions to the 68HC05 instruction set. Many of these instructions support the lengthened index register and the fully re-locatable stack pointer. However, a number of additional conditional branch instructions have been added. Later in the training, we'll take a detailed look at a few of these new instructions.

Other additions include the decimal adjust accumulator instruction, DAA, and the nibble swap accumulator instruction, NSA, to support BCD. The 68HC08 even adds a 16 by 8 unsigned divide instruction, DIV, that executes in only 7 bus cycles. Note that the multiply instruction (MUL) has been enhanced to execute in only 5 bus cycles instead of the eleven cycles required by the 68HC05.

The enhanced 68HC08 instruction set improves code efficiency and performance by taking multiple instructions and performing the same task with a single instruction.



68HC05 Addressing Modes

• Inherent	CLRA
• Immediate	LDA #20
• Direct	LDA $40
• Extended	LDA $4000
• Relative (± 128)	BLT LOOP
• Indexed	
– No offset	LDA ,X
– 8-bit offset	LDA $40, X
– 16-bit offset	LDA $4000, X

Next, let's look at the CPU addressing modes, beginning with the 68HC05 modes. The figure shows the different modes with example instructions. These modes include inherent, immediate, direct, extended, relative, and indexed addressing modes. These addressing modes can be used efficiently in both the 68HC05 and the 68HC08.

Inherent instructions have no operand fetch as the operand is defined in the 8-bit opcode. Clear accumulator, CLRA, is a single cycle instruction for the HC08; it takes 3 cycles for the HC05.

Immediate instructions have the 8-bit or 16-bit operand immediately following the opcode. Load accumulator with 20 is a two-byte instruction that executes in 2 bus cycles.

Direct addressing instructions have the 8-bit address of the operand immediately following the opcode. Therefore, direct instructions access the first 256 bytes of memory. Earlier, we referred to this type of access as direct page. Load accumulator with the operand at memory location 40 is a two-byte instruction that executes in 3 bus cycles.

Extended addressing instructions provide absolute addressing to any location in the 64K memory map without paging. They require three bytes total for the opcode plus the 16-bit address of the operand. Most assemblers will automatically use the shorter direct mode for any access to the first 256 bytes of the memory map.

All conditional branch instructions use relative addressing. If the branch condition is true, the PC is added to the signed byte immediately following the branch opcode. This gives a -128 to +127 byte range for branching. The jump or jump to subroutine instructions can be used to move the PC anywhere in the 64K memory map.

The indexed addressing modes are key to efficiently addressing tables and other data structures. Indexed addressing with no offset is what most other architectures refer to as indirect pointer addressing. The value in the index register is the address, or pointer, of the operand. The offset addressing modes provide the 68HC08 with a sometimes dramatic improvement in code efficiency compared to architectures with just indirect addressing or indirect address with another offset register.

Let's look at some examples of indexed addressing using some of the new control instructions of the 68HC08 instruction set.

Indexed Addressing with 8-bit Offset

A + B = C, where each is table of 5 values

```
        LDX     #5                2 cycles
LOOP    LDA     A_Data - 1, X     3 cycles
        ADD     B_Data - 1, X     3 cycles
        STA     C_Result - 1, X   3 cycles
        DBNZX   LOOP              3 cycles
                                  -----------------
                                  12 cycles/loop
                                  + 2 overhead

                                  10 bytes total code
```

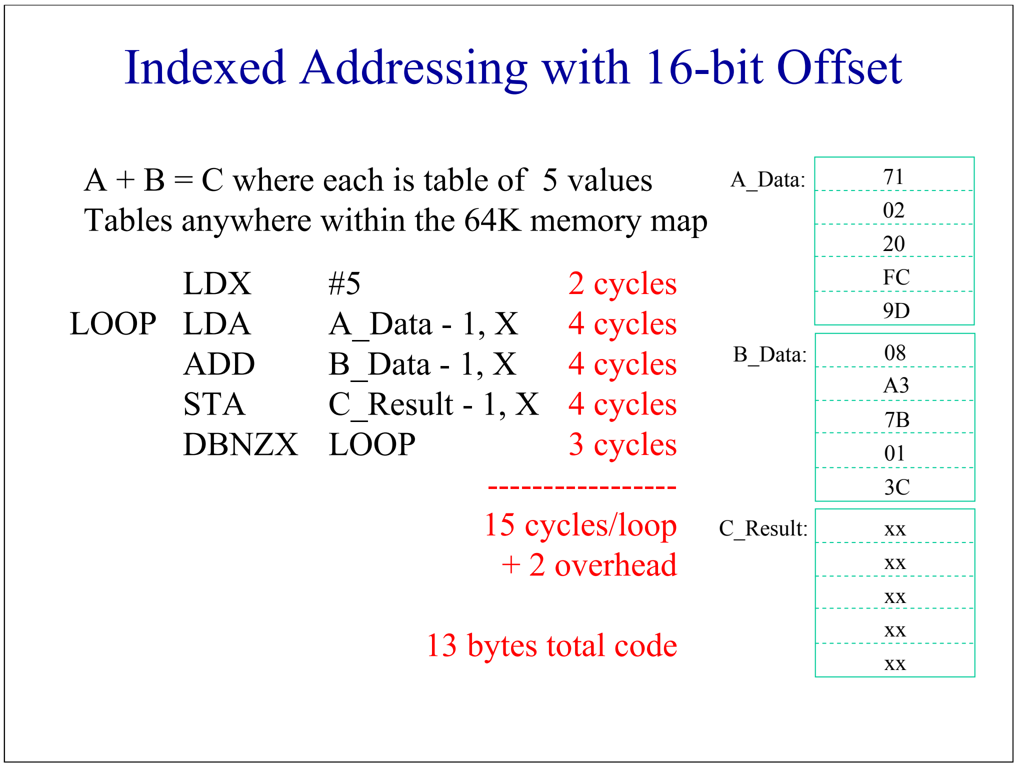
A_Data:	71
	02
	20
	FC
	9D
B_Data:	08
	A3
	7B
	01
	3C
C_Result:	xx
	xx
	xx
	xx
	xx

This first example is very basic but shows how powerful indexed addressing with 8-bit offset can be. This example will also use one of the new 68HC08 looping control instructions. We have two tables of data and would like to add each entry of table A to its corresponding entry in table B and then store the result in table C.

In this example, we have three tables each containing five entries. The first instruction loads the X index register with 5. This will be the offset into each entry of the tables as well as our counter for looping though every entry. Next, the LDA instruction loads the accumulator with the fifth entry of table A, 9D. The address pointer for this instruction is calculated first by the assembler subtracting 1 from the address label A_Data. Then the CPU adds the index register value 5 to point to the 5th entry of the table. The ADD instruction will add 9D in the accumulator with the fifth entry of table B, 3C. The result is placed in the accumulator. The STA instruction stores the result to the fifth entry of table C. Offset indexed addressing allows a single pointer to efficiently address three tables without time and code size consuming pointer reloading.

The DBNZX instruction is a new 68HC08 instruction that decrements the X register and branches if the result is not zero. This will decrement our pointer to 4 and branch back to the LOOP instruction to perform the same operation on the fourth entry of each table. This continues until all five entries are complete with X = 0. Later, we'll discuss some new 68HC08 addressing instructions that enable you to decrement the accumulator or an operand anywhere in the 64K map.

This code takes just 12 cycles per loop, two overhead cycles, and only 10 bytes total code space. This is much better than loading and reloading pointers, maintaining a separate loop count, and checking for the loop count.



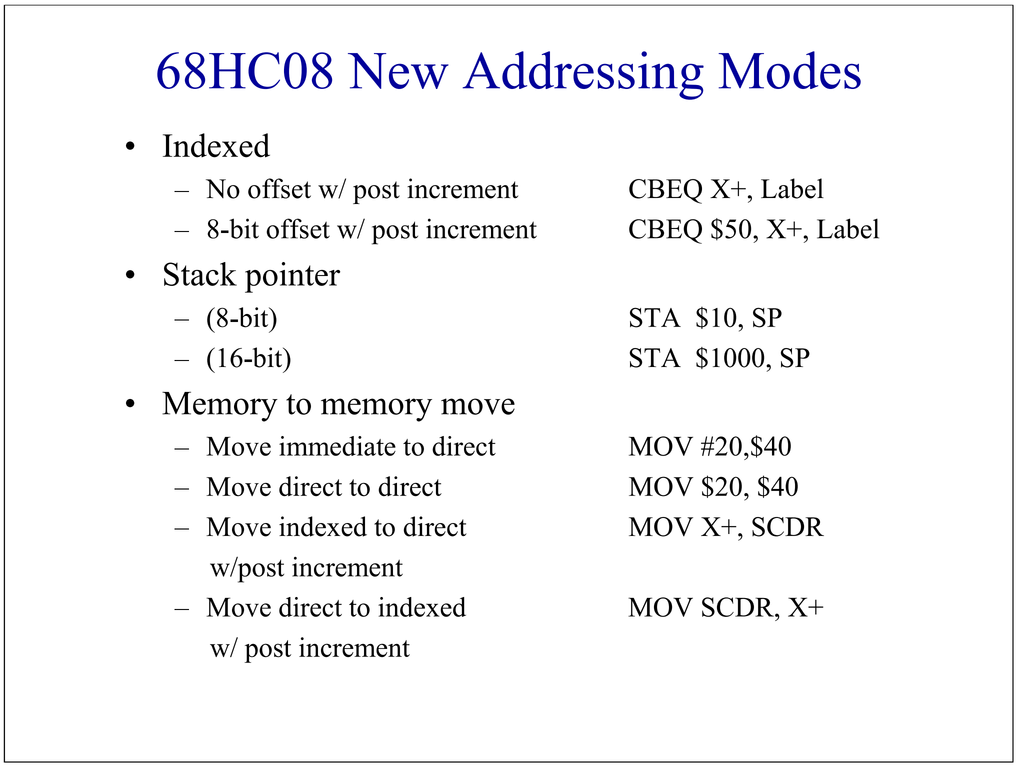
Indexed Addressing with 16-bit Offset

A + B = C where each is table of 5 values
Tables anywhere within the 64K memory map

```
        LDX     #5               2 cycles
LOOP    LDA     A_Data - 1, X    4 cycles
        ADD     B_Data - 1, X    4 cycles
        STA     C_Result - 1, X  4 cycles
        DBNZX   LOOP             3 cycles
                                 -----------------
                                 15 cycles/loop
                                 + 2 overhead

                                 13 bytes total code
```

A_Data:	71
	02
	20
	FC
	9D
B_Data:	08
	A3
	7B
	01
	3C
C_Result:	xx
	xx
	xx
	xx
	xx

The previous example assumed the tables could be addressed with only an 8-bit offset. Let's perform the same function on tables that are located anywhere within the 64K memory map. Notice that there are no changes to the assembly code! The assembler would automatically use the 16-bit offset indexed addressing mode instead of the 8-bit offset. For each instruction, this adds one more byte and one more bus cycle as shown. This example holds true regardless of table position; they can be anywhere in the 64 kbyte memory map regardless of whether the tables are in RAM or FLASH. Compared to many 8-bit architectures, this eliminates the need for additional code to handle paging, special instructions to first access data stored in the program code space table, or additional code to handle the case when the table crosses a 256 byte boundary.



68HC08 New Addressing Modes

• Indexed	
– No offset w/ post increment	CBEQ X+, Label
– 8-bit offset w/ post increment	CBEQ $50, X+, Label
• Stack pointer	
– (8-bit)	STA $10, SP
– (16-bit)	STA $1000, SP
• Memory to memory move	
– Move immediate to direct	MOV #20,$40
– Move direct to direct	MOV $20, $40
– Move indexed to direct w/post increment	MOV X+, SCDR
– Move direct to indexed w/ post increment	MOV SCDR, X+

The 68HC08 adds several new addressing modes, including two new indexed addressing modes, two stack pointer addressing modes, and four memory to memory move addressing modes. The figure shows the different modes with example instructions.

The two indexed addressing modes automatically post increment the index pointer.

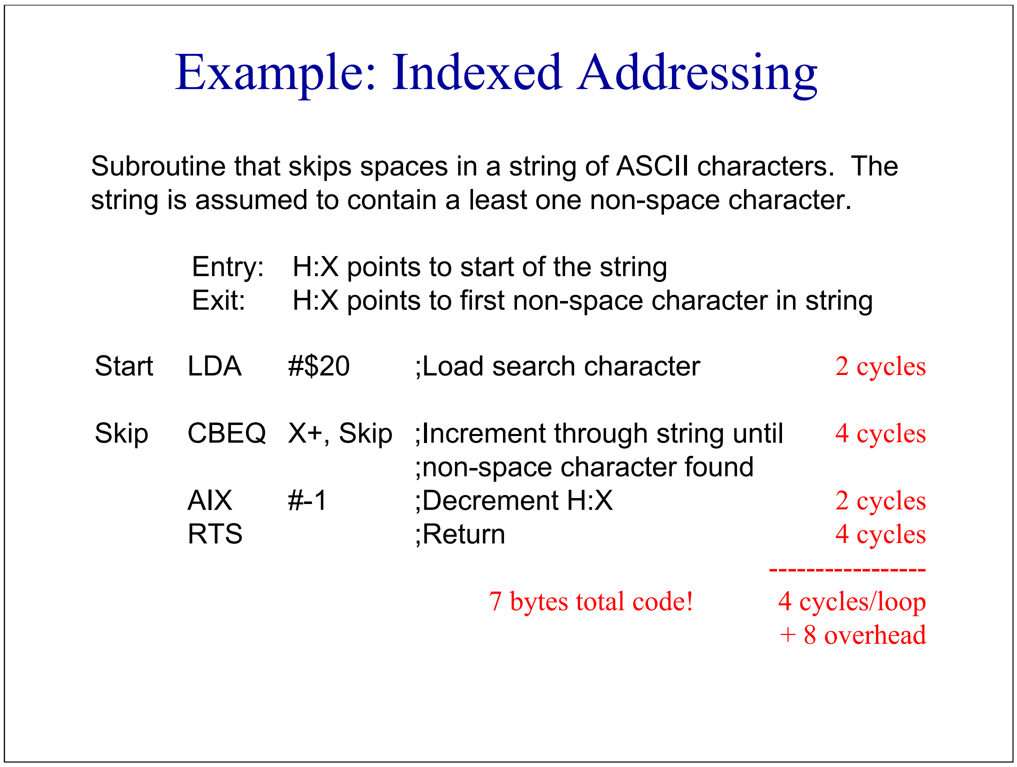
The new stack pointer relative addressing modes further improve C code efficiency. There are two types, stack pointer relative with an 8-bit offset and stack pointer relative with a 16-bit offset. They work similarly to the indexed addressing modes but use the stack pointer instead of the H:X index register. Note that you could use the stack pointer as an additional index register when interrupts are disabled.

In assembly programs and C compilers, the stack is commonly used to pass operands to subroutines. Typically the subroutine would pull the operands as needed into the accumulator. Stack relative addressing permits easy access of data on the stack, providing direct access to the operands, and eliminating the code and time required to push and pop values to and from the stack.

The four memory to memory addressing modes use the new move instruction, MOV, to directly move data without using the accumulator. The move immediate to direct mode is typically used to initialize variables and registers in the direct page. By eliminating the instructions required to save the accumulator, load the accumulator with the transfer data, store the accumulator with the transfer data, and then restore the accumulator, we reduce the number of execution cycles from nine to four.

The move direct to direct mode is typically used for register to register moves of data from the direct page. The move indexed to direct with post increment mode is typically used to transfer tables anywhere in the 64K memory map to a register in the direct page. For example, you can use this mode to transfer a buffer in RAM to the SCI transmitter. A similar mode is the move direct to indexed with post increment. You can use this mode to transmit data from the SCI to a receive buffer.

Let's look at a few examples of the new 68HC08 addressing modes.



Example: Indexed Addressing

Subroutine that skips spaces in a string of ASCII characters. The string is assumed to contain a least one non-space character.

Entry: H:X points to start of the string
Exit: H:X points to first non-space character in string

```
Start  LDA    #$20       ;Load search character             2 cycles

Skip   CBEQ   X+, Skip   ;Increment through string until    4 cycles
                         ;non-space character found
       AIX    #-1        ;Decrement H:X                     2 cycles
       RTS               ;Return                            4 cycles
                                                    -----------------
                         7 bytes total code!          4 cycles/loop
                                                      + 8 overhead
```

This example demonstrates indexed addressing with no offset and post increment. The subroutine uses the new 68HC08 instruction, compare and branch on equal, or CBEQ. This example searches a string of ASCII characters that is located anywhere in the 64K memory map to find the first non-blank character.

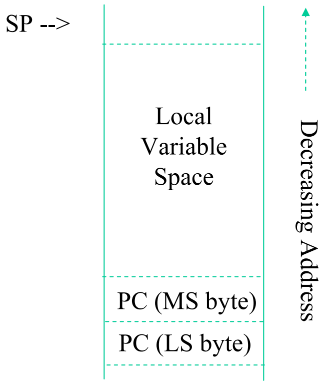
First, we load the accumulator with the ASCII blank character $20. The CBEQ instruction compares the accumulator with the operand pointed to by the H:X index register. If the accumulator matches the operand, this indicates a blank character and it branches back to itself

Note that the H:X register has been automatically incremented as part of the instruction. When it branches back to itself, it will compare the next character in the string. If the accumulator does not match the operand, we have found the first non-space character. In this case, the H:X register is still incremented but the subroutine doesn't branch to the Skip instruction. Instead, the subroutine decrements the H:X register with the add immediate to X (AIX) instruction, and returns with H:X pointing to the first non-space character.

This examples executes with 4 bus cycles per character that is searched and an additional eight cycles for overhead. This subroutine represents only 7 bytes of code! The CBEQ instruction can also use the 8-bit offset addressing mode with post increment. This allows comparing operands in multiple strings or data structures using a single index register as we saw in an earlier example. There is also a variation of this instruction that compares the 8-bit X index register with operands in memory called CBEQX.

Example: Stack Pointer Addressing

```
Sub1   AIS    # -16      ;Create 16 bytes
                         ;of local space
.
.
Skip   AIS    #16        ;Clean up stack

       RTS               ;Return
```



Note that SP must always point to the next unused byte. Do not use this byte (0,SP) for storage.

The second example demonstrates stack pointer addressing. The subroutine uses the new add immediate to stack instruction, or AIS.

The AIS instruction adds an immediate 8-bit signed value to the stack. It can be used to create and remove a stack frame buffer for storing temporary variables. This is commonly referred to as dynamically allocating RAM within the code or subroutine. Instead of each subroutine requiring dedicated RAM locations for temporary variables, the routines use the stack only as needed. This saves RAM space in applications that have numerous subroutines requiring temporary storage. Note that the AIS instruction does not set condition codes, so it will not affect any pending condition codes.

For example, let's assume you have ten subroutines and that each subroutine requires ten bytes for local temporary storage. Instead of allocating 100 bytes, you can use the stack and dynamically allocate RAM. This method uses only 10 bytes. These techniques enable you to write highly efficient C code and modular assembly code.

Note that the stack pointer must always point to the next unused byte. Therefore, do not use this byte, SP offset by 0, for storage.

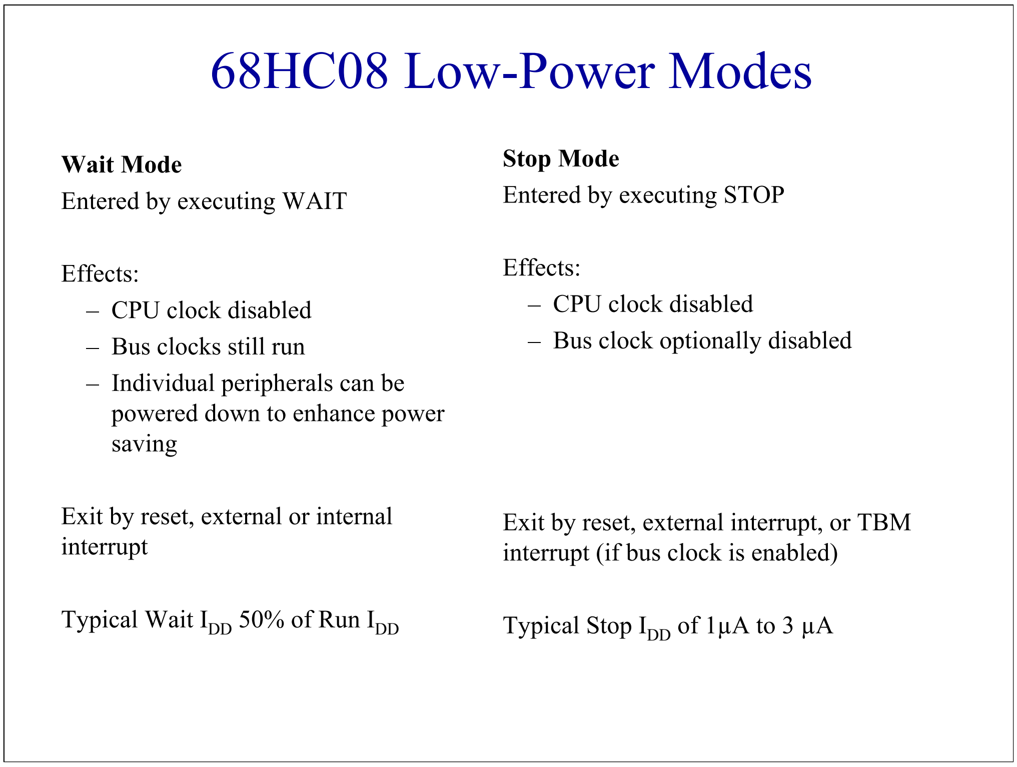
Example: Memory to Memory Move

```
;; Move indexed to direct

SIZE      EQU   16             ; TX circular buffer length
SCSR1     EQU   $16            ; SCI status register
SCDR      EQU   $18            ; SCI transmit data register

          ORG   $50
PTR_OUT   RMB   2              ; Circular buffer data out pointer
TX_B      RMB   SIZE           ; Circular buffer

          ORG   TX_IRQ         ; SCI transmit empty interrupt service
TX_INT    LDHX  PTR_OUT        ; Load pointer
          LDA   SCSR1          ; Dummy read as part of SCTE reset
          MOV   X+, SCDR       ; Move new byte to SCI and increment H:X
          CPHX  #TX_B + SIZE   ; Gone past end of circular buffer?
          BLS   NOLOOP         ; If note, continue
          LDHX  #TX_B          ; Else reset to start of new buffer
NOLOOP    STHX  PTR_OUT        ; Save new pointer value
          RTI                  ; Return
```

The next example is an interrupt-driven, asynchronous, serial port transmit routine that supports a circular buffer. The routine services the SCI transmitter empty interrupt flag. Note that the SCI Module is discussed in detail in the SCI tutorial. Let's examine the CPU code required to implement the interrupt service routine. The routine illustrates the move indexed to direct, memory to memory addressing mode.

This example uses a serial transmit buffer of 16 bytes. Two bytes have been allocated in RAM for the transmit buffer pointer and sixteen bytes have been allocated for the buffer itself. When the SCI is ready to transmit a byte, it will generate an interrupt when interrupts are enabled. When this occurs, the interrupt service routine first loads the pointer to the circular transmit buffer from the previously allocated RAM. This is followed by a dummy read of the SCI status register, which is the first step to clear the transmitter empty flag. The transmitter empty flag is finally cleared after the MOV instruction transfers the current byte of the transmit buffer to the SCI transmit data register. At the same time, the pointer H:X is automatically incremented to point to the next location in the transmit buffer.

After the MOV instruction, the CPHX instruction checks if the increment is past the end of the buffer. If not, the routine saves the updated H:X pointer and returns. If the pointer has extended past the end of the buffer, the pointer is reset to the beginning, TX_B, to form a circular buffer.



68HC08 Low-Power Modes

Wait Mode

Entered by executing WAIT

Effects:
- CPU clock disabled
- Bus clocks still run
- Individual peripherals can be powered down to enhance power saving

Exit by reset, external or internal interrupt

Typical Wait I_{DD} 50% of Run I_{DD}

Stop Mode

Entered by executing STOP

Effects:
- CPU clock disabled
- Bus clock optionally disabled

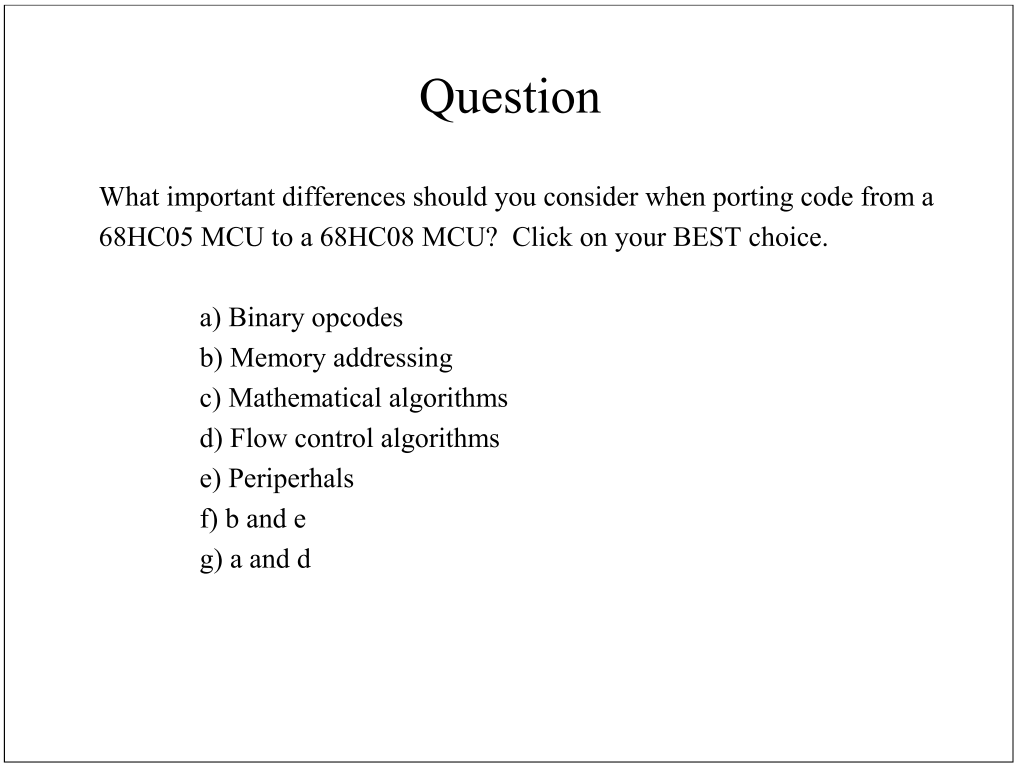
Exit by reset, external interrupt, or TBM interrupt (if bus clock is enabled)

Typical Stop I_{DD} of 1µA to 3 µA

As we discussed earlier, the 68HC08 CPU supports two basic low-power modes, wait mode and stop mode. Let's take a closer look at the uses and features of these two CPU modes.

Wait mode is entered when the WAIT instruction is executed. This instruction disables the CPU while keeping the clocks distributed to any running peripherals. When the CPU is in wait mode, each peripheral has the option to continue running or be disabled to further reduce power consumption. Wait mode power consumption is typically 50% of the run mode power consumption. Wait mode is exited by a reset, an internal interrupt, or an external interrupt. After exiting wait mode, normal CPU operations is immediately resumed since the system clocks are still running.

Stop mode is entered when the STOP instruction is executed. Unlike wait mode, stop mode disables all system clocks. An exception to this is the Time Base Module (TBM) which can be configured to run in stop mode to auto wake-up the CPU. The TBM is covered in the TBM training module. Other than the TBM, stop mode is exited only by a reset or an external interrupt. Stop mode offers the lowest power consumption, typically in the micro-amp range. Upon exiting stop mode, normal CPU operations are typically delayed by 4096 bus cycles to give the clock circuitry time to stabilize after restart. This stop mode delay can be disabled on some 68HC08 models.



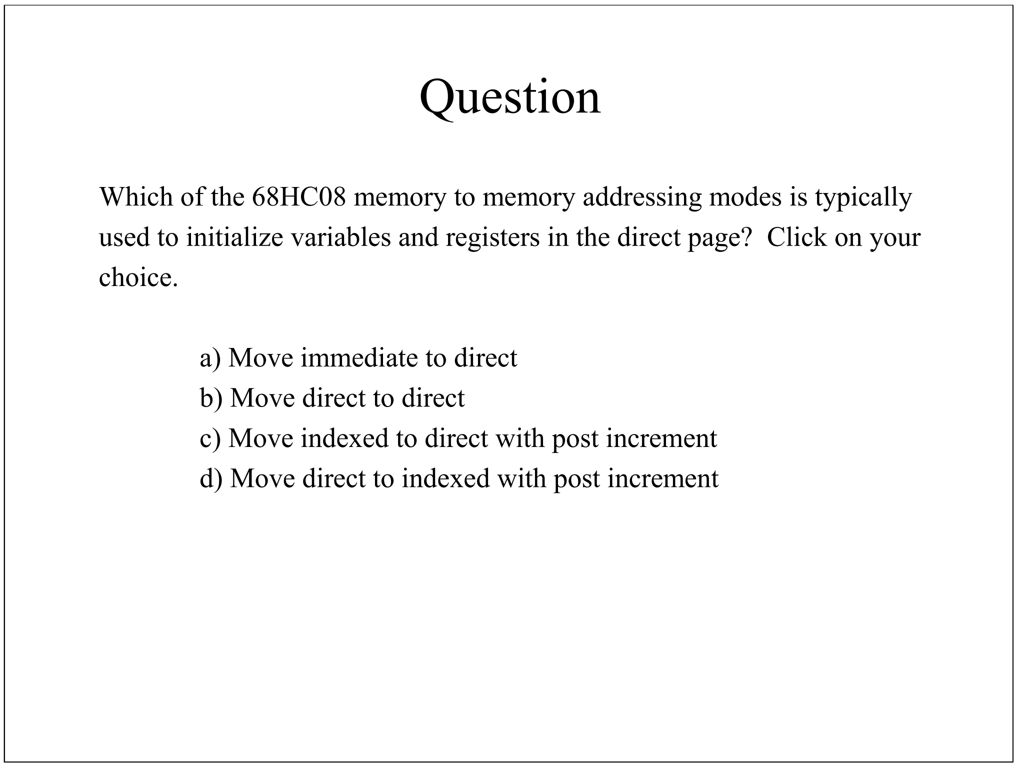
Question

What important differences should you consider when porting code from a 68HC05 MCU to a 68HC08 MCU? Click on your BEST choice.

a) Binary opcodes
b) Memory addressing
c) Mathematical algorithms
d) Flow control algorithms
e) Periperhals
f) b and e
g) a and d

Let’s review what we’ve discussed so far with a few questions to check your understanding of the material. What important differences should you consider when porting code from a 68HC05 MCU to a 68HC08 MCU? Click on your BEST choice.

Answer: When porting code from the 68HC05 MCU to the 68HC08 MCU, you must consider differences between addressing and peripherals. The other items in the list, binary opcodes, mathematical algorithms, and flow control algorithms, should port transparently.



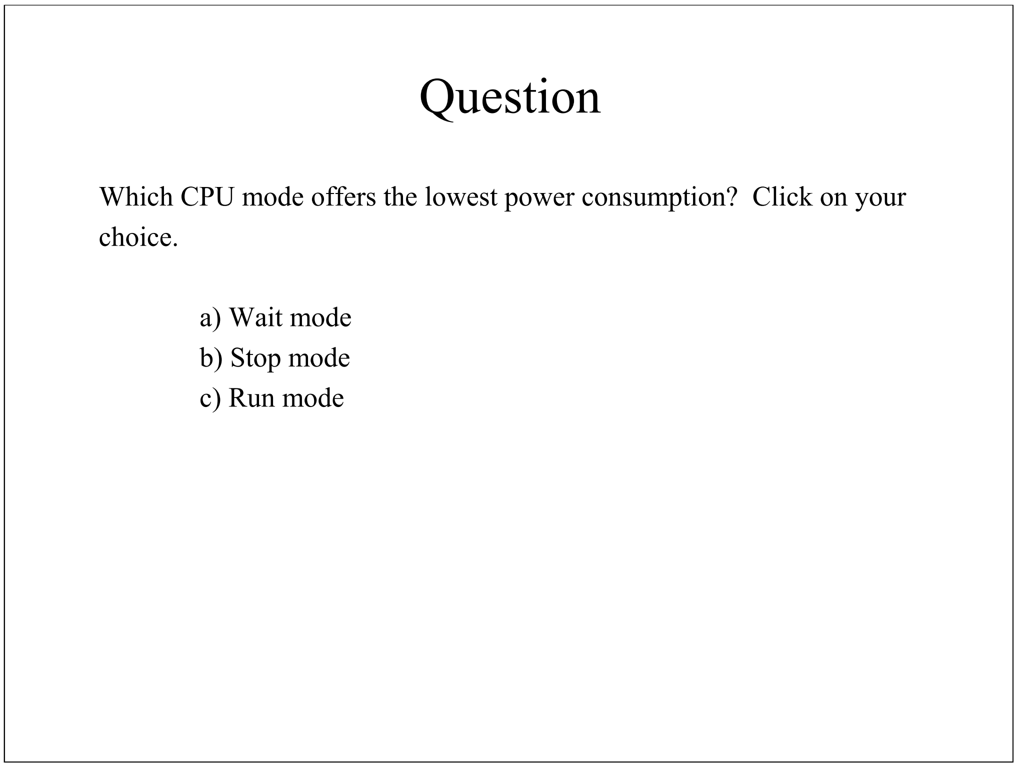
Question

Which of the 68HC08 memory to memory addressing modes is typically used to initialize variables and registers in the direct page? Click on your choice.

a) Move immediate to direct
b) Move direct to direct
c) Move indexed to direct with post increment
d) Move direct to indexed with post increment

Which of the 68HC08 memory to memory addressing modes is typically used to initialize variables and registers in the direct page? Click on your choice.

Answer: The move immediate to direct addressing mode is typically used to initialize variables and registers in the direct page.



Question

Which CPU mode offers the lowest power consumption? Click on your choice.

a) Wait mode
b) Stop mode
c) Run mode

Which CPU mode offers the lowest power consumption? Click on your choice.

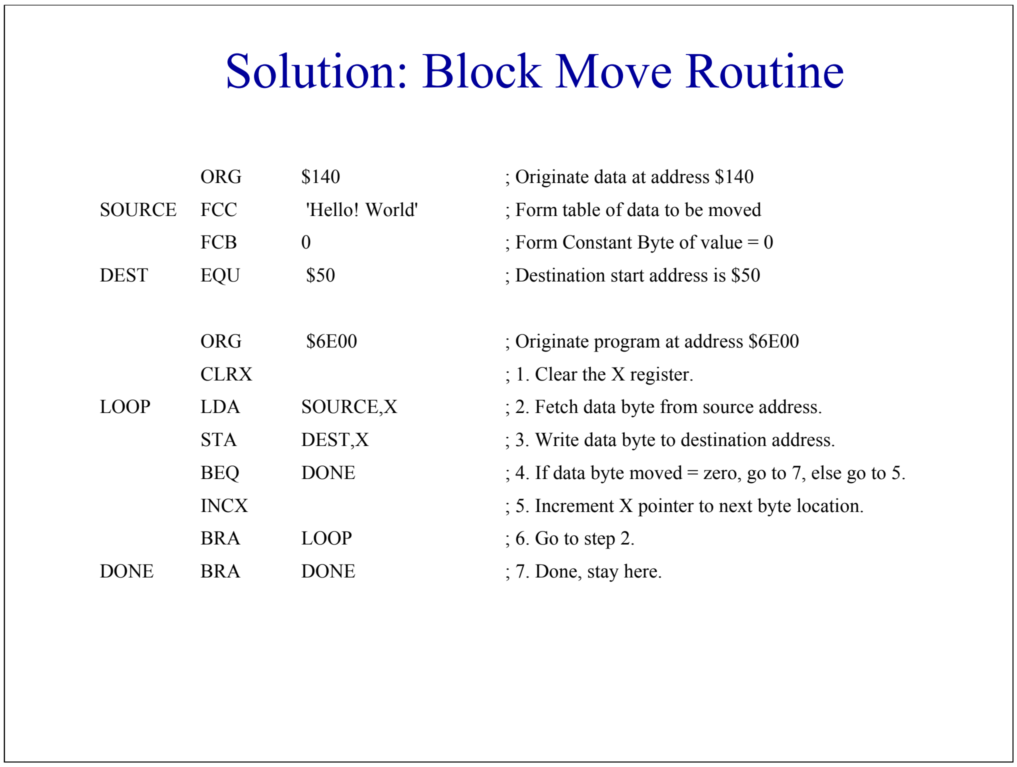
Answer: Stop mode offers the lowest power consumption, typically in the micro-amp range.

Exercise: Block Move Routine

- Write a block move routine to copy data.
- In the initialization instructions:
  - Originate the data starting at address $0140.
  - Copy this data to memory starting at address $0050.
  - Originate the program at address $6E00.
- In the main program loop:
  - Copy the data one byte at a time.
  - Include a check for the byte value 0.

Let's complete this tutorial with three programming exercises. In the first exercise, write a block move routine to copy data starting from memory location $1040 to memory location $0050. Copy the data one byte at a time. To detect the end of the data, use a check for the byte value 0.

Take a moment to review the exercise instructions. When you are finished writing your program, click the Forward arrow to continue the tutorial and review the exercise solution.



Solution: Block Move Routine

```
          ORG      $140           ; Originate data at address $140
SOURCE    FCC       'Hello! World' ; Form table of data to be moved
          FCB      0              ; Form Constant Byte of value = 0
DEST      EQU       $50           ; Destination start address is $50

          ORG       $6E00         ; Originate program at address $6E00
          CLRX                    ; 1. Clear the X register.
LOOP      LDA      SOURCE,X       ; 2. Fetch data byte from source address.
          STA      DEST,X         ; 3. Write data byte to destination address.
          BEQ      DONE           ; 4. If data byte moved = zero, go to 7, else go to 5.
          INCX                    ; 5. Increment X pointer to next byte location.
          BRA      LOOP           ; 6. Go to step 2.
DONE      BRA      DONE           ; 7. Done, stay here.
```

Compare your program with the one provided in the solution. When you are finished, click the Forward arrow to move to the next programming exercise.

Exercise: Find Maximum

- Write routine to find the maximum in a list of ten data bytes.
- In the initialization instructions:
  - Originate the data starting at address $130.
  - Use a table to store the ten data bytes.
  - Originate the program at address $6E00.
- In the main program loop:
  - Use pointers to search for the maximum value.
  - Include a check for the byte value 0.

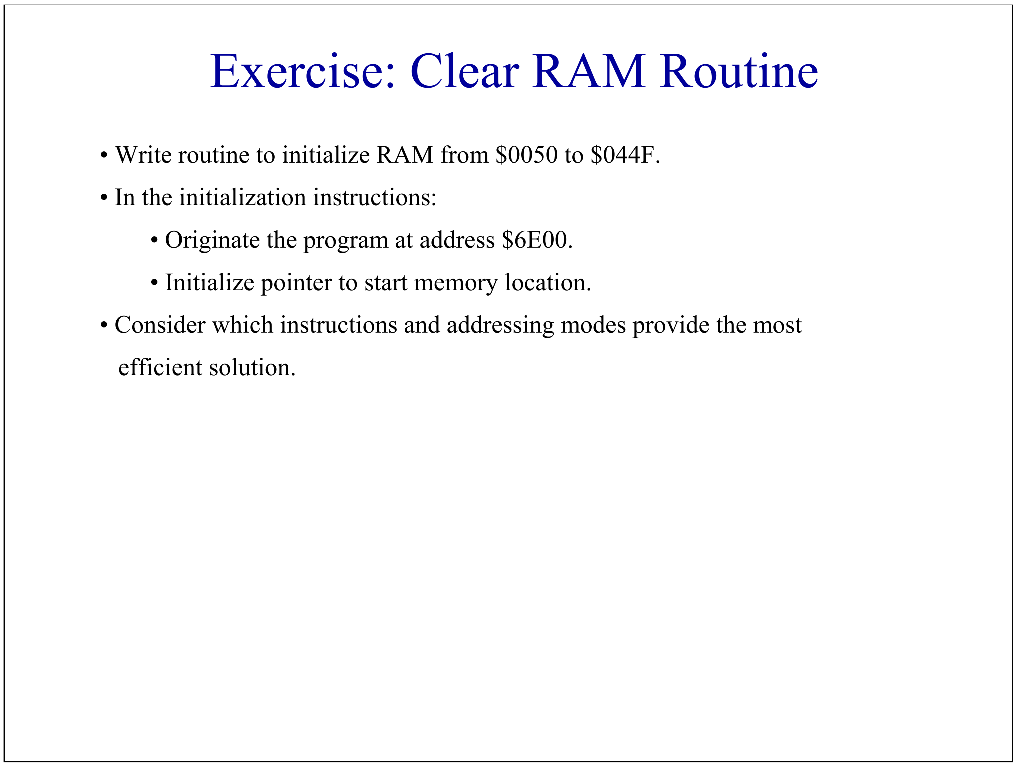
In the second exercise, write a routine that finds the maximum value in a list of ten data bytes. The list of data bytes starts at address $130. Use a table to store the data. Use pointers to search the table for the maximum value. To detect the end of the data list, use a check for the byte value 0.

Take a moment to review the exercise instructions. When you are finished writing your program, click the Forward arrow to continue the tutorial and review the exercise solution.

Solution: Find Maximum Value

```
          ORG       $130                ; Originate data at address $130
DATA      FCB       $C0,$40,$8B         ; Form table of data bytes
          FCB       $75,$A0,$60
          FCB       $DB,$25,$B0
          FCB        $50

          ORG       $6E00               ; Originate program at address $6E00
          LDX       #10                 ; 1. Load the number of bytes in the X register.
GETBYTE   LDA       DATA-1,X            ; 2. Load accumulator with byte from source address.
DECCNT    DECX                          ; 3. Decrement the X register.
          BEQ       DONE                ; 4. If pointer = 0, then go to step 8 (DONE)
          CMP       DATA-1,X            ; 5. Compare accumulator value to byte pointed
                                        ;      to by source pointer.
          BHS       DECCNT              ; 6. If accumulator value > byte value pointed to by
                                        ;     source pointer, go to step 3, else go to step
7.
          BRA       GETBYTE             ;  7. Loop back GETBYTE instruction
DONE      BRA       DONE                ;  8. Done, stay here.
```

Compare your program with the one provided in the solution. When you are finished, click the Forward arrow to move to the next programming exercise.

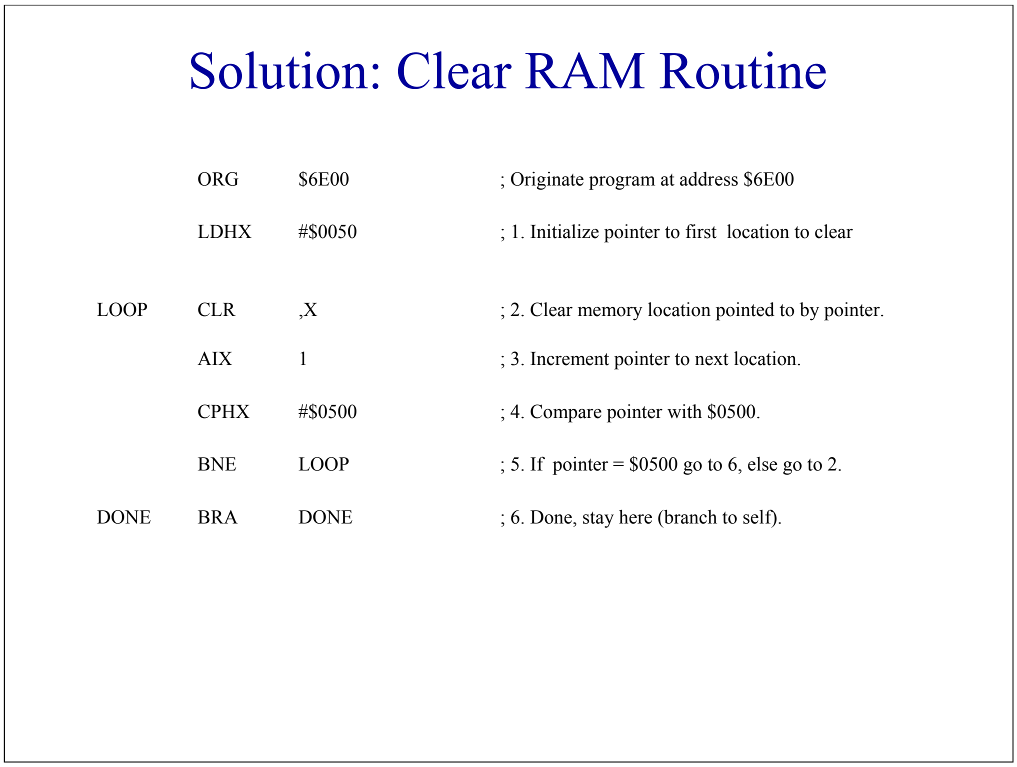


Exercise: Clear RAM Routine

- Write routine to initialize RAM from $0050 to $044F.
- In the initialization instructions:
  - Originate the program at address $6E00.
  - Initialize pointer to start memory location.
- Consider which instructions and addressing modes provide the most efficient solution.

In the third exercise, write clear RAM routine to copy data starting from memory location $1040 to memory location $0050. You should consider which instructions and addressing modes will produce the most efficient subroutine.

Take a moment to review the exercise instructions. When you are finished writing your program, click the Forward arrow to continue the tutorial and review the exercise solution.



Solution: Clear RAM Routine

```
        ORG     $6E00       ; Originate program at address $6E00
        LDHX    #$0050      ; 1. Initialize pointer to first  location to clear

LOOP    CLR     ,X          ; 2. Clear memory location pointed to by pointer.
        AIX     1           ; 3. Increment pointer to next location.
        CPHX    #$0500      ; 4. Compare pointer with $0500.
        BNE     LOOP        ; 5. If  pointer = $0500 go to 6, else go to 2.
DONE    BRA     DONE        ; 6. Done, stay here (branch to self).
```

Compare your program with the one provided in the solution.

Tutorial Completion

- Evolution from 68HC05 to 68HC08 Architecture
- CPU Programmer’s Model
- Memory Mapping
- Enhanced Instruction Set and Addressing Modes
- Stop mode and Wait mode

In this tutorial, you’ve had an opportunity to work with the 68HC08 CPU. You’ve learned about the evolution of the 68HC05 architecture to the 68HC08 model that extends the programmer’s model and provides a high-degree of upward code compatibility. You’ve also learned about the 68HC08 CPU features that enable you to write efficient code, including memory mapping, an enhanced instruction set, and enhanced addressing modes. You’ve demonstrated your knowledge by writing several programs to move and manipulate data.