

COP8

COP8 Instruction Set Performance Evaluation



Literature Number: SNOA377

COP8™ Instruction Set Performance Evaluation

National Semiconductor
Application Note 1042
July 1996



1.0 OVERVIEW

National offers two families of 8-bit COP8 microcontrollers: Basic family and Feature family. The Feature family offers more on-chip peripheral and nine additional instructions compared to the Basic family. In this report, the COP8 Basic family instruction set performance is evaluated versus that of three competitive microcontrollers, the Motorola M68HC05, the Intel 80C51, and the Microchip PIC16C5X. The architecture, addressing modes, instruction sets and salient features of these microcontrollers are compared. Eight benchmark programs are developed, with each microcontroller programmed with full documentation for each of the benchmarks. Summary tables compare the results relative to both code efficiency and execution time.

The report examines only the instruction set efficiency and speed of execution of the selected microcontrollers. Factors related to on-chip hardware features (RAM/ROM sizes, interrupts, etc.) are not considered. Manufacturers offer a variety of options of hardware features, but the instruction set efficiency and execution speed typically remain identical across a manufacturer's microcontroller product line.

When comparing a microcontroller to the competition, it is customary for manufacturers to select benchmark programs which particularly highlight the instruction set of their device. To avoid this phenomenon, general purpose commonly used benchmark routines were selected.

One word of caution—this benchmark report, like all others of its kind, relies on a set of small program fragments. The operations performed by each of these fragments may or may not correspond to the operations required for a particular application. When evaluating a microcontroller for a demanding application, it is important to examine how an individual microcontroller will perform in that particular case.

ARCHITECTURE

Three of the four microcontrollers have a modified Harvard architecture, while a fourth (the Motorola M68HC05) has a Von Neumann architecture (named after John Von Neumann, an early pioneer in the computer field at Princeton). With a Von Neumann architecture, a CPU (Central Processing Unit) and a memory are interconnected by a common address bus and a data bus. Positive aspects of this approach include convenient access to tables stored in ROM and a more orthogonal instruction set. The address bus is used to identify which memory location is being accessed, while the data bus is used to transfer information either from the CPU to the selected memory location or vice versa. Von Neumann was the first to realize that this architectural model could have the memory serve as either program memory or data memory. In earlier computers (both electronic and electromechanical), the program store (often a programmed patchboard) had been completely separate from the data store (often a bank of vacuum tubes or relays).

The single address bus of the Von Neumann architecture is used sequentially to access instructions from program memory and then execute the instructions by retrieving data from and/or storing data in data memory. This means that instruction fetch cannot overlap data access from memory.

A Harvard architecture (names after the Harvard Mark 1 and the early electromechanical computers developed at Harvard by Howard Aiken—another computer pioneer) has separate program memory and data memory with a separate address bus and data bus for each memory. One of the benefits of the Harvard architecture is that the operation of the microcontroller can be controlled more easily in the event of corrupted program counter. A modified Harvard architecture allows accessing data tables from program memory. This is very important with modern day computers, since the program memory is usually ROM (Read Only Memory) while the data memory is usually RAM (Random Access Memory). Consequently, data tables usually need to be in ROM so that they are not lost when the computer is powered down.

The obvious advantage of a Von Neumann architecture is the single address and single data bus linking memory with the CPU. A drawback is that code can be executed from data memory opening up the possibility for undesired operation due to corruption of the program counter or other registers. Alternatively, the advantage of a modified Harvard architecture is that instruction fetch and memory data transfers can be overlapped with a two stage pipeline, which means that the next instruction can be fetched from program memory while the current instruction is being executed using data memory. A drawback is that special instructions are required to access RAM and ROM data values making programming more difficult.

The instructions which cause the "Modified" Harvard architectures of the three microcontrollers are the instructions that provide a data path from program memory to the CPU. These instructions are listed below:

COP8	LAID	Load Accumulator indirect from program memory
80C51	MOVC A, @A + DPTR	Move Constant - Load Accumulator with a "fixed constant" from program memory
PIC16C5X	RETLW	Return Literal to W from program memory

TRI-STATE® is a registered trademark of National Semiconductor Corporation.
COP8™ and WATCHDOG™ are trademarks of National Semiconductor Corporation.

COP8 offers the single-byte LAID instruction which uses the contents of the accumulator to point to a data table stored in the program memory. The data accessed from the program memory is transferred to the accumulator. This instruction can be used for table lookup operations and to read the entire program memory contents for checksum calculations.

The 80C51 family offers a similar instruction. The MC68HC05 family has a Von Neumann architecture where CPU and program memory are interconnected by a common address and data bus.

Microchips's PIC16C5x family offers the RETLW instruction. To do table lookup, the table must contain a string of RETLW instructions. The first instruction just in front of the table of the RETLW instructions, calculates the offset into the table. The table can only be used as a result of a CALL. This instruction certainly does not offer the flexibility of the COP8 LAID instruction and cannot be used to perform a checksum calculation on the entire program memory contents.

The use these instructions is demonstrated in the fourth benchmark program, which entails a three byte table search from a data table in program memory.

2.0 SELECTED MICROCONTROLLERS

COP8

The COP8 has a modified Harvard architecture with memory mapped input/output. The CPU registers include an 8-bit accumulator (A), a 16-bit program counter (PC), two 8-bit data pointers (B and X), an 8-bit stack pointer (SP), an 8-bit processor status word (PSW), and an 8-bit control register (CNTRL). The data memory includes a bank of 16 registers (including the three pointers) which have special attributes. All RAM, I/O ports, and registers (except A and PC) are mapped into the data memory address space. The timer section includes a 16-bit timer, and an associated 16-bit autoreload register. The generic COP8 I/O section includes two 8-bit I/O ports, each with an associated 8-bit configuration register and an associated 8-bit data register. The I/O section also contains one dedicated 8-bit output port with an associated 8-bit data register, one dedicated 8-bit input register, and one special purpose 8-bit I/O port with associated 8-bit configuration register and 8-bit data register.

M68HC05

The M68HC05 has a Von Neumann architecture with memory mapped input/output. The CPU registers include an 8-bit accumulator (A), a 16-bit program counter (PC), an 8-bit index register (X), a stack pointer (SP), and an 8-bit condition code register (CCR). The timer section includes a 16-bit free running counter, two 16-bit counter registers to read the counter, a 16-bit timer input capture register, a 16-bit counter output compare register, an 8-bit timer control register, and an 8-bit timer status register. The I/O section includes three bidirectional 8-bit I/O ports, each with an associated 8-bit data register and an 8-bit direction register. The I/O section also includes a 7-bit input port with an associated input register.

80C51

The 80C51 has a modified Harvard architecture with memory mapped input/output. The CPU registers include an 8-bit accumulator (A), an 8-bit auxiliary register (B) for multiply and divide, a 16-bit program counter (PC), and 8-bit program status word (PSW), an 8-bit stack pointer (SP), and a 16-bit data pointer (DPTR). The register bank consists of eight special working registers R0–R7. Registers R0 and R1 can be used as indirect address pointers to data memory. The timer section includes two 16-bit timers, an 8-bit timer mode register (TMOD), and an 8-bit timer control register (TCON). The I/O section includes four 8-bit I/O ports, each with an associated 8-bit register.

PIC16C5X

The PIC16C5X has a modified Harvard architecture with memory mapped input/output. The PIC16C5X also has a RISC (Reduced Instruction Set Computer) type architecture in that there are only 33 single word basic instructions. Actually these 33 instructions should be expanded to a total of 47 for comparison purposes with the other microcontrollers. This is necessary since 14 of the 33 instructions have a programmable destination bit, which selects one of two destinations for the result of the instruction. Consequently, each of these 14 instructions should be counted as dual instructions. The CPU registers include an 8-bit working register (W) which serves as a pseudo accumulator in that it holds the second operand, receives the literal in the immediate type instructions, and also can be program selected as the destination register. However, a bank of 31 file registers serve as the primary accumulators in that they represent the first operand and also may be program selected as the destination register. The first eight file registers include the real time clock/counter register (RTCC) mapped as F1, the 9-bit program counter (PC) mapped as F2, the 8-bit status word register (SWR) mapped as F3, and the 8-bit I/O port registers mapped as F5–F7. The 8-bit File Select Register (FSR) is mapped as F4, whose low order 5 bits select one of the 31 file registers in the indirect addressing mode. Calling for file F0 in any of the file oriented instructions selects indirect addressing and will use the File Select Register (FSR). It should be noted that file register F0 is not a physically implemented register. The CPU also contains a two level 12-bit hardware push/pop stack for subroutine linkage. The PIC16C5X also has a WATCHDOG™ timer as well as the real time clock/counter register (RTCC), but it does not have any hardware interrupts. Consequently, the counter register RTCC must be program sampled for any overflow.

The number of instructions in a program must also be calibrated differently with the PIC16C5X since the instruction word is 12 bits in length. Consequently, twenty 12-bit word instructions will contain the byte equivalent of thirty COP8 8-bit byte instructions. The larger size instruction word is instrumental in implementing the RISC architecture. It should also be noted that the upper members of the PIC family expand to having a 16-bit instruction word.

3.0 INSTRUCTION SET ANALYSIS

Addressing Modes

COP8

1. Direct
2. Register Indirect
3. Register Indirect with Post Increment/Decrement
4. Immediate
5. Immediate Short
6. Indirect from Program Memory
7. Jump Relative
8. Jump Absolute
9. Jump Absolute Long
10. Jump Indirect

68HC05

1. Inherent
2. Immediate
3. Extended
4. Direct
5. Indexed with no offset
6. Indexed with 8-bit offset
7. Indexed with 16-bit offset
8. Relative

80C51

1. Register
2. Direct
3. Indirect
4. Immediate
5. Relative
6. Absolute
7. Long
8. Indexed

PIC16C5X

1. Data Direct
2. Data Indirect
3. Immediate
4. Program Direct
5. Program Indirect
6. Relative

Instruction Types

COP8

Total basic instructions: 49
Total instructions including addressing modes: 87

1. Arithmetic
2. Load and Exchange
3. Logical
4. Bit Manipulation
5. Conditional
6. Transfer of Control

68HC05

Total basic instructions: 62
Total instructions including addressing modes: 210

1. Register/Memory
2. Read/Modify/Write
3. Branch
4. Control

80C51

Total basic instructions: 51
Total instructions including addressing modes: 111

1. Arithmetic
2. Logical
3. Data Transfer
4. Boolean Variable
5. Program Branching

PIC16C5X

Total basic instructions: 33
Total instructions with 14 dual destination instructions counted: 47

1. Byte-oriented File Register
2. Bit-oriented File Register
3. Literal and Control

COP8 Instruction Set Features

1. Majority of single byte opcode instructions to minimize program size.
2. One instruction cycle for the majority of single byte instructions to minimize program execution time.
3. Many single byte multiple function instructions such as DRSZ.
4. Three memory mapped pointers: Two data pointers (B and X) for register indirect addressing, and one program memory stack pointer (SP) for the software stack.
5. Sixteen memory mapped registers which allow an optimized implementation of certain instructions.
6. Ability to set, reset, and test any individual bit in data memory address space, including the memory mapped I/O ports and associated registers.
7. Register indirect LOAD and EXCHANGE instructions with optional automatic post-incrementing or post-decrementing of the register pointers (Both B and X pointers). This allows for greater efficiency (both in throughput time and program code) in both loading and processing fields in data memory.
8. Unique instructions to optimize program size and throughput efficiency. Some of these instructions are:

DRSZ	IFBNE	DCOR
RETSK	RRC	LAI
9. Forty nine basic instructions.
10. Ten addressing modes provide great flexibility.

M68HC05 Instruction Set Features

1. Very flexible branch structure with 21 different branch instructions.
2. Twelve different read/modify/write instructions.
3. Five bit manipulation instructions (Clear, Set, Branch if Bit Clear, Branch if Bit Set, Bit Test Memory with Accumulator) provide great flexibility.
4. Indexed addressing with options of no offset, 8-bit offset, or 16-bit offset.
5. Data Tables located in page 0 address space (0000–00FF) take advantage of the direct addressing mode for optimal code.
6. Multiply instruction (unsigned, 8×8).
7. Sixty two basic instructions.
8. Eight addressing modes provide great flexibility.

80C51 Instruction Set Features

1. Optimized for 8-bit control applications.
2. Provides a variety of fast compact addressing modes for accessing data memory to facilitate operations on small data structures.
3. Offers extensive support for one bit variables, allowing direct bit manipulation in control and logic systems that require Boolean processing.
4. Eight working registers (R0–R7) selected by three bits allow a function code and register address to be combined in one single byte instruction.
5. Register R0 and R1 serve as indirect addressing data memory pointers.
6. Four banks of working registers, with only one bank active at a time, permit fast and effective “context switching”.
7. Several register specific instructions referring to the accumulator (A), the accumulator and auxiliary register (B), register pair (AB), the carry flag (C), the data pointer (DPTR), and the program counter (PC) provide great efficiency.
8. Indexed addressing using a base register (either the program counter (PC) or the data pointer (DPTR) and an offset in the accumulator (A) provide great flexibility.
9. Multiply and Divide instruction (using A and B registers).
10. Fifty one basic instructions.
11. Eight addressing modes provide great flexibility.

PIC16C5X Instruction Set Features

1. All single word (12-bit) instructions for compact code efficiency.
2. All instructions are single cycle except the jump type instructions (GOTO, CALL) and failed test instructions (DECFSZ, INCFSZ, BTFSC, BTFSS) which are two cycle.
3. 32 File registers can be addressed directly or indirectly, and serve as accumulators to provide first operand, Working register (W) serves as pseudo accumulator, providing second operand.
4. Working register (W) also serves as destination for literal from program memory in MOVLW (Move Literal to W) and RETLW (Return Literal to W) instructions.

5. Many instructions include a destination bit which selects either the register file or the accumulator as the destination for the result.
6. Four bit manipulation instructions (Set, Clear, Test and Skip if Set, Test and Skip if Clear) provide great flexibility.
7. Status word register (SWR) memory mapped as register file F3 allows testing of status bits (carry, digit carry, zero, power down, and time-out).
8. Program counter (PC) memory mapped as register file F2 allows W to be used as offset register for indirect addressing of program memory.
9. Indirect addressing mode data pointer FSR (file select register) memory mapped as register file F4. Addressing F0 causes FSR to be used to select file register.
10. Literal in RETLW (Return Literal to W) instruction combined with file register mapped program counter allow data tables to be accessed from program memory.
11. Two level 12-bit push/pop hardware stack for subroutine linkage using the CALL and RETLW instructions.
12. WATCHDOG timer.
13. Thirty three basic instructions.
14. Six addressing modes provide great flexibility.

COMPARISON OF SALIENT INSTRUCTION SET FEATURES

Addressing

The three types of indexed addressing (no offset, single byte offset, and dual byte offset) in the M68HC05 provide great flexibility in walking across two or three data fields (two operands, result) simultaneously. The three other microcontrollers achieve the same result with indirect addressing and pointers. The COP8 and 80C51 both have dual pointers for indirect data addressing (B and X pointers in the COP8, R0 and R1 in the 80C51). The PIC16C5X only has one indirect data pointer F4, which is called by using F0. Consequently, processing two multiple byte operands with the PIC16C5X requires alternate loading of the operand addresses into the F4 pointer as the multiple byte data fields are processed.

All four microcontrollers have some form of indirect addressing for program memory, which is very useful in producing jump tables. The COP8 uses the JID (jump indirect) instruction, while the PIC16C51X can address the program counter as file register F2 and add an offset to it. The M68HC05 and 80C51 both use their indexed addressing modes to achieve indirect addressing of program memory. Both jump tables and lookup tables are easily created with indexed addressing. The COP8 uses the LAID (load accumulator indirect) instruction to implement lookup tables, while the PIC16C5X uses the RETLW (return literal to W) instruction as the table entries preceded by a table header which produces the table offset address.

Bit Manipulation

All four microcontrollers have instructions to set, reset, and test individual bits in data memory. However, the 80C51 allows bit manipulation in only 210-bit addressable locations of data memory (16 general purpose byte locations 20–2F, and the rest in the special function registers including the I/O ports). The other three microcontrollers are capable of bit addressing anywhere in data memory. Three of the four microcontrollers have direct bit tests for a bit being either set or reset, while the COP8 test is for bit set only. The 80C51 has two instructions (MOV C, bit and MOV bit, C) to either transfer a bit of data memory to or from the carry. This is very handy for saving a carry and restoring it for later use. The two bit test instructions (BRCLT and BRSET) in the M68HC05 also set the carry to the state of the bit tested. The 80C51 also has four logical (two logical AND, two logical OR) bit manipulation instructions, with the carry and a selected memory bit serving as the two operands. These four instructions are described more fully with the logical instructions.

Input/Output

The four microcontrollers all have multiple I/O ports, with some of them (COP8, M68HC05) also having dedicated input and/or dedicated output ports. The COP8 and M68HC05 both have configuration registers for each I/O port, with each configuration bit determining whether the associated port bit is selected as input or output. The PIC16C5X contains a TRISTATE® instruction (TRIS) whose operand determines whether or not each associated port bit is put in the TRISTATE condition to serve as an input pin. The 80C51 configures a port by writing ones to TRISTATE the port bit positions selected as inputs. Otherwise the associated 80C51 port bit is selected as an output bit. The COP8 has three addresses associated with each I/O port. The first two addresses select the associated data and configuration register for the port, while the third address selects the actual port pins. If a COP8 port bit is configured as an input, then the associated bit in the data register selects whether the associated input is Hi-Z or weak pull-up. Input/Output manipulation with the I/O ports is demonstrated in the fifth benchmark program.

Increment/Decrement

All four microcontrollers have accumulator increment and decrement instructions. The register file of the PIC16C5X serves as multiple accumulators. The COP8 has a DRSZ (decrement register and skip if zero) instruction for its sixteen registers located at data memory addresses 0F0–0FF. This register bank includes the B and X data memory pointers and also the SP stack pointer. The PIC16C5X also has both INCFSZ (increment file and skip if zero) and DECFSZ (decrement file and skip if zero) instructions. The M68HC05 increment and decrement instructions can be selected for the accumulator, the index register, or any memory location. The 80C51 increment and decrement instructions can be selected for the accumulator, the eight R registers including the R0 and R1 data pointers, or any memory location (selected either directly or indirectly). The 80C51 also has a DJNZ (decrement and jump if not zero) instruction which can be selected for any of the eight R registers including the R0 and R1 data pointers, or for any memory location selected with a relative address.

Load Memory Immediate

Only the COP8 and the 80C51 have instructions that can load memory with an immediate value. The memory location to be loaded can be selected with either direct or indirect addressing for both microcontrollers.

Post Incrementation or Decrementation of Data Pointers

Only the COP8 has instructions that can post increment, post decrement, or leave the data pointer unchanged. This feature applies to both the COP8 load and exchange instructions, and can be used with either of the two pointers B and X. The feature is very useful when processing multiple byte fields, especially with two operands (additions, subtractions) or an operand and a result (block move). Examples of the usage of this feature can be found in the first three COP8 benchmark programs.

Loop Counting and Data Pointing Testing

Three of the four microcontrollers (excluding M68HC05) have specific instructions to facilitate loop counting. The COP8 DRSZ (decrement register and skip if zero) instruction tests one of sixteen registers (including the two data pointers B and X) and skips the next instruction (a branch back to loop) if the result is zero. The PIC16C5X DECFSZ (decrement file register and skip if zero) is analogous to the COP8 DRSZ instruction. The 80C51 DJNZ (decrement and jump if not zero) instruction combines both the test and jump in a single instruction. These three instructions (COP8 DRSZ, PIC16C5X DECFSZ, 80C51 DJNZ) all operate on a loop counter, but uses the standard BNE (branch if not equal) branch instruction following the decrement to test if the counter is zero. The BNE instruction tests the M68HC05 zero status flag which is active with the decrement instruction. Only the COP8 contains a test data pointer action (IFBNE—If B pointer not equal). This instruction is used to sense the end of a program loop where multiple byte fields are being processed.

The 80C51 CJNE (compare and jump if not equal) instruction used in the immediate addressing mode provides an alternative method of loop counting. This instruction is analogous to the COP8 IFBNE instruction but combines the test with the jump in a single instruction.

It should be noted that neither the COP8 IFBNE instruction nor the 80C51 CJNE instruction require the use of a counter, but rather depend on an end of field comparison for loop termination. Consequently these instructions are used in loops where a data pointer is being used to walk across a multiple byte field. Examples of the usage of these two instructions can be found in the first three benchmarks.

Branch and Subroutine Call Instructions

All four microcontrollers have absolute address branching, and three of the four (excluding the PIC16C5X) also have relative address branching. Only the COP8 and the M68HC05 have single byte branching. The COP8 single byte branching is with relative addressing, where six bits provide relative addressing of –31 to +32. The M68HC05 single byte branching is with no offset indexed addressing, where the index register X provides the absolute address. All of the myriad conditional branch instructions of the M68HC05 use two bytes, with the second bytes providing relative addressing of –127 to +128.

Three of the four microcontrollers have a software stack (excluding the PIC16C5X) where the return address is stored when a subroutine is called. Consequently, multiple levels of subroutine and interrupt nesting are possible with the software stack. The PIC16C5X has a two level hardware stack where the return address is stored with a subroutine call. Thus only two levels of subroutine nesting are possible with the PIC16C5X. The other three microcontrollers have absolute address subroutine calls, with the 80C51 also having a relative address call. Only the M68HC05 provides a single byte subroutine call, using no offset indexed addressing (with the index register providing the absolute address).

Return Instructions

All four microcontrollers have return from subroutine instructions and three of the four (excluding the PIC16C5X) have return from interrupt instructions. The PIC16C5X does not have an interrupt.

The COP8 has two return from subroutine instructions, RET and RETSK. The RET is the normal return from subroutine instruction, while the RETSK instruction returns from the subroutine and then skips the next instruction. These two instructions are very useful in passing back flag information from the subroutine in lieu of actually using a flag such as the carry. This is demonstrated in the fourth benchmark (table search) where a flag needs to be passed back from the subroutine to indicate whether or not the search was successful. The COP8 simply uses the RETSK instruction to indicate a successful search, whereas each of the other three microcontrollers need to set up the carry as the flag. The PIC16C5X return from subroutine instruction is RETLW (return and place literal in W). The literal serves as an immediate data value from memory. The use of this instruction is demonstrated in the fourth benchmark (table search), where the program memory data table consists of a block of RETLW instructions preceded by the table header.

Comparison and Conditional Branch Instructions

Three of the four microcontrollers (excluding the PIC16C5X) have comparison instructions. The COP8 has two comparison instructions, IFEQ (if equal) and IFGT (if greater than). These comparison instructions compare the accumulator versus an immediate value or a number from memory (selected with either direct or indirect addressing). The M68HC05 has two comparison instructions, CMP (compare) and BIT (bit test memory with accumulator). The CMP instruction compares the accumulator versus an immediate value or a number from memory (selected with either direct or indexed addressing). The BIT instruction performs a logical AND comparison between the accumulator and an immediate value or a number from memory (selected with either direct or indexed addressing). The 80C51 has one comparison instruction CJNE (compare and jump if not equal) which is combined with a branch in the same instruction. All four microcontrollers have bit test instructions (including test carry.)

The M68HC05 has a myriad of test conditions and branch instructions, including BCC (branch if carry clear), BCS (branch if carry set), BRCLR (branch if memory bit clear), BRSET (branch if memory bit set), BEQ (branch if equal), BHI (branch if higher), BHS (branch if higher or same), BLO (branch if lower), BLS (branch if lower or same), BMI (branch if minus), BNE (branch if not equal), and BPL (branch if plus).

The 80C51 includes seven conditional branch instructions, consisting of JC (jump if carry), JNC (jump if no carry), JNZ (jump if not zero), JZ (jump if zero), JB (jump if memory bit set), JNB (jump if memory bit not set), and JBC (jump if memory bit set and clear bit).

The COP8 includes seven conditional test instructions, IFBIT (test if memory bit is set), IFC (if carry), IFNC (if no carry), IFEQ (if equal), IFGT (if greater than), IFBNE (if B pointer not equal), and DRSZ (decrement register and skip if zero). These seven instructions will all cause the next instruction to be skipped if the test is successful. The skipped instruction can either be a branch or some function (such as setting a bit flag or returning from a subroutine) that can be contained in one instruction.

The PIC16C5X includes four test instructions, consisting of DECFSZ (decrement file register and skip if zero), INCFSZ (increment file register and skip if zero), BTFSC (bit test file register and skip if bit clear), and BTFSS (bit test file register and skip if bit set). These four instructions will all cause the next instruction to be skipped if the test is successful.

Logical Instructions (AND, OR, Exclusive OR)

All four microcontrollers have a full complement of the logical instructions, with the accumulator and a selected memory location (using either direct or indirect addressing) serving as the two operands. Three of the four microcontrollers (excluding the PIC16C5X) can also perform the logical instructions with the accumulator and an immediate value serving as the operands. The 80C51 contains four logical bit instructions which perform either the logical AND or logical OR between the carry bit and a selected memory bit, with either polarity of the memory bit being selectable. The M68HC05 contains a logical compare instruction, which performs a logical AND comparison between the accumulator and an immediate value or a number from memory (selected with either direct or indexed addressing).

Shift and Rotate Instructions

Only the M68HC05 has shift instructions, both logical left shift and logical right shift. All four microcontrollers have a right rotate instruction through carry (none bit loop), and three of the four (excluding the COP8) also have a left rotate through carry. The 80C51 also has both a right and left rotate direct (eight bit loop).

Complement Instructions

Three of the four microcontrollers (excluding the COP8) have a complement instruction for the accumulator. The 80C51 also has two complement bit instructions, with the first being used to toggle the carry and the second to complement selected accumulator bits.

BDC Decimal Correct

Only the COP8 and the 80C51 provide instructions to expedite BCD processing. The DCOR (decimal correct) instruction of the COP8 is used following an add or subtract instruction to achieve the correct BCD result. Note that a hex 66 must be added to the first operand to start the addition process. The DA (decimal adjust) instruction of the 80C51 is used following an add instruction to achieve the correct BCD result. The decimal adjust instruction does not work following subtraction. Consequently, BCD subtraction in the 80C51 must be implemented by adding the complement of the subtrahend (second operand) and then using the decimal adjust instruction. BCD subtraction is demonstrated in the third benchmark program.

Exchange and Swap

Only the COP8 and the 80C51 have exchange instructions between the accumulator and memory. Both microcontrollers can select either direct or indirect addressing for the associated exchange memory selection. The 80C51 also has a XCHD A, @Ri (exchange digit) instruction which exchanges the low order nibble (digit) of the accumulator with the low order nibble of the indirectly addressed memory location selected.

All of the microcontrollers except the M68HC05 have a SWAP instruction which interchanges the low and high order nibbles of the accumulator. With the PIC16C5X, the SWAPF instruction can be selected for any of the register bank accumulators.

Push and Pop Instructions

Only the 80C51 has PUSH and POP instructions to augment operations with the software stack. Consequently, only the 80C51 has the ability to store temporary data in the software stack as well as subroutine and interrupt return addresses.

4.0 BENCHMARK PROGRAMS

1. FIVE BYTE BLOCK MOVE

This benchmark program moves a block of five data byte from one location to another.

2. FOUR BYTE BINARY ADDITION

This benchmark subroutine adds two four byte binary numbers and replaces the first operand with the result, like an adding machine (A + B replaces A). The carry flag is used to indicate an overflow.

3. FOUR BYTE PACKED BCD SUBTRACTION

This benchmark subroutine subtracts two eight digit packed BCD numbers and replaces the first operand with the result, like an adding machine. (A - B replaces A). The carry flag is used to indicate a negative result.

4. THREE BYTE TABLE SEARCH

This benchmark subroutine searches a program memory data table for a three byte match. A flag indicates whether or not the search was successful, with the address of the first byte of a matched string being returned.

5. INPUT/OUTPUT MANIPULATION

This benchmark compares two 8-bit I/O ports P1 and P2. If they are equal, a nine is output as the least significant digit of a third port P3. If P1 is greater than P2, then P2 data is output on P1. If P1 is < P2, then the higher order digit of P1 is copied to the lower order digit position of P3.

6. SERIAL INPUT/OUTPUT WITH OFFSET TABLE

This benchmark subroutine inputs a sequence of byte couplets, each of which consists of an address followed by a data byte. The address is used to access a data table to produce an offset value which is added to the data byte. The updated data byte is output while the next couplet's address byte is input.

7. TIMEKEEPING

This timekeeping benchmark program emulates a real time clock, keeping track of hours, minutes, and seconds in packed BCD format. The program is interrupt driven, using a timer interrupt.

8. SWITCH ACTIVATED FIVE SECOND LED

This benchmark samples a switch input to activate a five second output for turning on an LED. The switch is debounced with a 50 ms program delay both on opening and closure.

5.0 BENCHMARK DATA

This section provides the benchmark programs for each microcontroller. For each instruction, the byte count and instruction cycle count is given. The total cycle count is multiplied by the fastest cycle time of the selected microcontroller to yield the total benchmark execution time.

Microcontrollers	Instruction Cycle Time	XTAL
COP8	1.0 μ s	10 MHz
M68HC05	0.5 μ s	4 MHz
80C51	1.0 μ s	12 MHz
PIC16C5X	0.5 μ s	8 MHz

COP8 BENCHMARK #1—FIVE BYTE BLOCK MOVE

This benchmark moves only a block of five data bytes from a specific source location to a specific destination location.

```
MOVE:   LD      B, #14    ; 1/1 Source address to B pointer
        LD      X, #50    ; 2/3 Destination address to X pointer
LOOP:   LD      A, [B+]   ; 1/2 Load A with source byte
        X      A, [X+]   ; 1/3 Source byte to destination
        IFBNE  #3        ; 1/1 Test (modulo 16) if block move finished (19-16=3)
        JP     LOOP      ; 1/3 Loop back if not finished
```

7 BYTES
47 CYCLES

68HC05 BENCHMARK #1—FIVE BYTE BLOCK MOVE

This benchmark moves only a block of five data bytes from a specific source location to a specific destination location.

```
MOVE:   LDX     #5        ; 2/2 Load X with block length
LOOP:   LDA     24, X     ; 2/4 Load A with source byte (start at block top)
        STA     54, X     ; 2/5 Store A at destination (start at block top)
        DECX    ; 1/3 Decrement X
        BNE    LOOP      ; 2/3 Loop back if block transfer not finished
```

9 BYTES
76 CYCLES

80C51 BENCHMARK #1—FIVE BYTE BLOCK MOVE

This benchmark moves only a block of five data bytes from a specific source location to a specific destination location.

```

MOVE: MOV    RO, #20      ; 2/1 Load RO pointer with source address
      MOV    R1, #50      ; 2/1 Load R1 pointer with destination address
LOOP: MOV    A, @RO       ; 1/1 Load A with source byte
      MOV    @R1, A       ; 1/1 Store source byte at destination
      INC    R1           ; 1/1 Increment destination pointer
      INC    RO           ; 1/1 Increment source pointer
      CJNE   RO, #25, LOOP; 3/2 Compare & loop back if block transfer not finished

```

11 BYTES
32 CYCLES

PIC16C5X BENCHMARK #1—FIVE BYTE BLOCK MOVE

This benchmark moves only a block of five data bytes from a specific source location to a specific destination location.

```

      TEMP   EQU F29      ;
      BASE   EQU F30      ;      Source @ F20-F24 Destination @ F25-F29
      CNTR   EQU F31      ;
MOVE: MOVLW  5            ; 1/1 Load W with length of block
      MOVWF          ; 1/1 Store block length in CNTR
      MOVLW  20          ; 1/1 Load W with address of source
LOOP: MOVF    BASE, W     ; 1/1 Source address to W
      MOVWF   F4         ; 1/1 Source address to F4 (indirect pointer)
      MOVF    FO, W      ; 1/1 Source byte to W
      MOVWF   TEMP       ; 1/1 Source byte to TEMP
      MOVLW  5           ; 1/1 Destination/source address difference to W
      ADDWF   F4         ; 1/1 Add to pointer to get destination address
      MOVF    TEMP, W    ; 1/1 Source byte to W
      MOVWF   FO         ; 1/1 Source byte to destination
      INCF   BASE       ; 1/1 Increment BASE
      DECFSZ CNTR       ; 1/1 Test if block move finished
      GOTO   LOOP       ; 1/2 Loop back if not finished

```

14 WORDS (Equivalent to 21 BYTES)
62 CYCLES

COP8 BENCHMARK #2—FOUR BYTE BINARY ADDITION

This benchmark adds two four byte binary numbers and replaces the first operand with the result. This emulates an adding machine addition, where $A + B$ replaces A. The benchmark is programmed as a subroutine, with the carry flag indicating an overflow.

```
ADDITION: LD      B, #10      ; 1/1 Set up address of 1st operand in B pointer
          LD      X, #20      ; 2/3 Set up address of 2nd operand in X pointer
          RC                          ; 1/1 Reset carry
LOOP:     LD      A, [X+]     ; 1/3 Load 2nd operand to A
          ADC     A, [B]     ; 1/1 Add 1st operand to 2nd operand
          X      A, [B+]    ; 1/2 Result replaces 1st operand
          IFBNE  #14        ; 1/1 Test if addition finished
          JP     LOOP        ; 1/3 Loop back if not finished
          RET                      ; 1/5 Return from subroutine
```

10 BYTES
48 CYCLES

M68HC05 BENCHMARK #2—FOUR BYTE BINARY ADDITION

This benchmark adds two four byte binary numbers and replaces the first operand with the result. This emulates an adding machine addition, where $A + B$ replaces A. The benchmark is programmed as a subroutine, with the carry flag indicating an overflow.

```
ADDITION: LD      X, #16      ; 2/2 Load index register with 1st operand address
          CLC                          ; 1/2 Clear carry
LOOP:     LDA     X           ; 1/3 Load A with 1st operand
          ADC     4, X        ; 2/4 Add 2nd operand to A
          STA     X           ; 1/4 Replace 1st operand with result
          INCX                      ; 1/3 Increment index register
          CPX     #20        ; 2/2 Compare X with end of field
          BNE    LOOP        ; 2/3 Loop back if addition not finished
          RTS                      ; 1/6 Return from subroutine
```

13 BYTES
85 CYCLES

80C51 BENCHMARK #2—FOUR BYTE BINARY ADDITION

This benchmark adds two four byte binary numbers and replaces the first operand with the result. This emulates an adding machine addition, where A + B replaces A. The benchmark is programmed as a subroutine, with the carry flag indicating an overflow.

```
ADDITION: MOV   R1, #20      ; 2/1 Load R1 pointer with address of 2nd operand
           MOV   R0, #16      ; 2/1 Load R0 pointer with address of 1st operand
           CLR   C           ; 1/1 Clear carry
LOOP:     MOV   A, @R0       ; 1/1 Load 1st operand to A
           ADD   A, @R1      ; 1/1 Add 2nd operand to A
           MOV   @R0, A      ; 1/1 Replace 1st operand with result
           INC   R1         ; 1/1 Increment 2nd operand pointer
           INC   R0         ; 1/1 Increment 1st operand pointer
           CJNE  R0, #20, LOOP ; 3/2 Test if finished and loop back if not
           RET                    ; 1/2 Return from subroutine
```

14 BYTES
33 CYCLES

PIC16C5X BENCHMARK #2—FOUR BYTE BINARY ADDITION

This benchmark adds two four byte binary numbers and replaces the first operand with the result. This emulates an adding machine addition, where A + B replaces A. The benchmark is programmed as a subroutine, with the carry flag indicating an overflow.

```
           STATUS EQU F3      ; Status register
           BASE  EQU F26      ; 1st operand @ F16-F19 2nd operand @ F20-F23
           CNTR  EQU F25      ;
           TEMP  EQU F24      ;
ADDITION: MOVLW 4             ; 1/1 Load W with byte count (length of field)
           MOVWF CNTR        ; 1/1 Store byte count in CNTR
           MOVLW 20          ; 1/1 Load W with address of 2nd operand
           MOVWF BASE        ; 1/1 Store address of 2nd operand in BASE
           BCF   STATUS, 0    ; 1/1 Clear carry (carry is bit 0 of F3 register)
LOOP:     MOVF  BASE, W       ; 1/1 2nd operand address to W
           MOVWF F4          ; 1/1 2nd operand address to F4 (indirect pointer)
           MOVF  F0, W        ; 1/1 2nd operand to W
           MOVWF TEMP        ; 1/1 2nd operand to TEMP
           MOVLW 4           ; 1/1 Byte count to W
           SUBWF F4          ; 1/1 Subtract from pointer to get 1st operand address
           MOVF  TEMP, W     ; 1/1 2nd operand to W
           ADDWF F0          ; 1/1 Add 2nd operand to 1st operand
           MOVLW 4           ; 1/1 Byte count to W
           ADDWF F4          ; 1/1 Add to pointer to restore 2nd operand address
           INCF  BASE        ; 1/1 Increment BASE
           DECFSZ CNTR       ; 1/1 Test if addition finished
           GOTO  LOOP        ; 1/2 Loop back if addition not finished
           RETLW 0           ; 1/2 Return from subroutine
```

19 WORDS (Equivalent to 28 1/2 BYTES)
62 CYCLES

COP8 BENCHMARK #3—FOUR BYTE PACKED BCD SUBTRACTION

This benchmark subtracts two eight digit packed BCD numbers (four bytes each) and replaces the minuend (first operand) with the result. This emulates an adding machine subtraction, where A - B replaces A. The benchmark is programmed as a subroutine, with the carry flag being used to indicate a positive or negative result (carry set for negative result). The BCD decimal correct (DCOR) command is used following the subtraction to achieve the correct BCD result.

```
BCDSUBT: LD      B, #14      ; 1/1 Set up address of 1st operand in B pointer
          LD      X, #20      ; 2/3 Set up address of 2nd operand in X pointer
          SC                      ; 1/1 Set carry for no input borrow to subtraction
LOOP:    LD      A, [X+]     ; 1/3 Load 2nd operand to A
          X                      ; 1/1 Exchange 1st and 2nd operands
          SUBC   A, [B]     ; 1/1 Subtract 2nd from 1st operand
          DCOR   A          ; 1/1 Decimal correct result of subtraction
          X      A, [B+]    ; 1/2 Result replaces 1st operand
          IFBNE #2          ; 1/1 Test (modulo 16) if subtraction finished (18-16=2)
          JP     LOOP       ; 1/3 Loop back if not finished
          IFNC                      ; 1/1 Test if result negative (borrow=no carry)
          JP     NEGR       ; 1/3 Jump if result negative
          RC                      ; 1/1 Reset carry to indicate positive result
          RET                      ; 1/5 Return from subroutine
NEGR:    SC                      ; 1/1 Set carry for no input borrow to subtraction
          LD      B, #14      ; 1/1 Reinitialize B pointer to start of result
LUP:     CLR     A          ; 1/1 Clear A to set up subtraction from zero
          SUBC   A, [B]     ; 1/1 Subtract result from zero
          DCOR   A          ; 1/1 Decimal correct new result
          X      A, [B+]    ; 1/2 Store new result
          IFBNE #2          ; 1/1 Test (modulo 16) if subtraction finished (18-16=2)
          JP     LUP        ; 1/3 Loop back if not finished
          SC                      ; 1/1 Set carry to indicate negative result
          RET                      ; 1/5 Return from subroutine
```

25 BYTES

97 CYCLES

M68HC05 BENCHMARK #3—FOUR BYTE PACKED BCD SUBTRACTION

This benchmark subtracts two eight digit packed BCD numbers (four bytes each) and replaces the minuend (first operand) with the result. This emulates an adding machine subtraction, where A - B replaces A. The benchmark is programmed as a subroutine, with the carry flag being used to indicate a positive or negative result (carry set for negative result). Note in the M68HC05 subtraction that the carry represents the borrow, unlike some microcontrollers where the carry represents the absence of borrow in subtraction. The M68HC05 does not have any decimal correct or decimal adjust instructions, so the BCD correction must be program implemented. The correction algorithm for BCD subtraction consists of subtracting six whenever a nibble borrow occurs from the BCD subtraction result for each digit.

```

        TEMP    EQU    $91        ;
        FLAG    EQU    $92        ;
BCDSUBT: JSR     INIT             ; 2/5 Call initialization subroutine
LOOP:   LDA     X                 ; 1/3 Load A with 1st operand
        SBC     4, X              ; 2/4 Subtract 2nd operand from A
        JSR     CORRECT          ; 2/5 Call correction subroutine
        BNE     LOOP            ; 2/3 Loop back if subtraction not finished
        BCS     NEGR            ; 2/3 Branch to NEGR (neg. result) if carry
        RTS                      ; 1/6 Return from subroutine
NEGR:   JSR     INIT             ; 2/5 Call initialization subroutine
LUP:    CLRA                    ; 1/3 Clear A
        SBC     X                 ; 1/3 Subtract result of previous subtraction from zero
        JSR     CORRECT          ; 2/5 Call correction subroutine
        BNE     LUP             ; 2/3 Loop back if subtraction from zero not finished
        SEC                      ; 1/2 Set carry to indicate negative result
        RTS                      ; 1/6 Return from subroutine
INIT:   LD      X, #16           ; 2/2 Load index register with 1st operand address
        CLC                      ; 1/2 Clear carry (reset borrow)
        CLRA                    ; 1/2 Clear A
        STA     FLAG            ; 2/4 Clear FLAG register
        RTS                      ; 1/6 Return from initialization subroutine
CORRECT: STA    X                ; 1/4 Replace 1st operand with result
        CLRA                    ; 1/3 Clear A
        BCC     BYP1            ; 2/3 Branch if carry set (no borrow)
        BSET    0, FLAG         ; 2/5 Set flag if carry (save carry value)
        LDA     #$60            ; 2/2 Load A with packed BCD 60
BYP1:   BHCC    BYP2            ; 2/3 Branch if half carry set (no half borrow)
        ADD     #$06            ; 2/2 Add packed BCD 06 to A
BYP2:   STA     TEMP            ; 2/4 Store A in TEMP
        LDA     X                ; 1/3 Load A with result of first subtraction
        SUB     A, TEMP          ; 2/3 Subtract TEMP from previous result
        BRSET   0, FLAG, NXT    ; 3/5 Restore carry (flag value to carry)
NXT:    STA     X                ; 1/4 Store new result
        INCX                   ; 1/3 Increment index register
        CPX     #20             ; 2/2 Compare X with end of field
        RTS                      ; 1/6 Return from correction subroutine
```

54 BYTES

567 CYCLES

80C51 BENCHMARK #3—FOUR BYTE PACKED BCD SUBTRACTION

This benchmark subtracts two eight digit packed BCD numbers (four bytes each) and replaces the minuend (first operand) with the result. This emulates an adding machine subtraction, where $A - B$ replaces A . The benchmark is programmed as a subroutine, with the carry flag being used to indicate a positive or negative result (carry set for negative result). It should be noted in the 80C51 subtraction that the carry represents the borrow, unlike some microcontrollers where the carry represents the absence of borrow in subtraction.

The BCD decimal adjust (DA) command only works following addition, not subtraction. Consequently, the BCD subtraction must be implemented as an addition by adding the complement of the subtrahend (2nd operand) to the 1st operand. This complement is achieved by subtracting the subtrahend from a packed BCD 99 (binary 10011001), and then adding one to the result. The following are examples of this complementation procedure:

$$1. 61-49 = 61 + (99-49) + 1 = 61 + 50 + 1 = 12 \pmod{100} \text{ (Output borrow shows positive result)}$$

$$2. 49-61 = 49 + (99-61) + 1 = 49 + 38 + 1 = 88 \text{ (No output borrow indicates negative result)}$$

$$\text{Negative result correction: } -(99-88 + 1) = -12$$

```
BCDSUBT: MOV  R2, #4          ; 2/1 Load R2 with byte count
          MOV  R1, #20       ; 2/1 Load R1 with address of 2nd operand
          MOV  R0, #16       ; 2/1 Load R0 with address of 1st operand
          CLR  C              ; 1/1 Clear carry for no input borrow to subtraction
LOOP:    MOV  A, #099H       ; 2/1 Load accumulator with packed BCD 99
          SUBB A, @R1        ; 1/1 Subtract 2nd operand from packed BCD 99
          ADD  A, #01        ; 2/1 Add one to result
          DA   A              ; 1/1 Decimal correct result
          ADD  A, @R0        ; 1/1 Add 1st operand to result
          DA   A              ; 1/1 Decimal correct new result
          MOV  @R0, A        ; 1/1 Replace 1st operand with result
          INC  R1             ; 1/1 Increment 2nd operand pointer
          INC  R0             ; 1/1 Increment 1st operand pointer
          DJNZ R2, LOOP      ; 2/2 Test if finished and loop back if not
          JC   NEGR          ; 2/2 Test and jump if negative result
          RET                ; 1/2 Return from subroutine (no carry: positive result)
NEGR:    CLR  C              ; 1/1 Clear carry for no input borrow to subtraction
          MOV  R0, #16       ; 2/1 Load R0 with address of result
LUP:     MOV  A, #099H       ; 2/1 Load accumulator with packed BCD 99
          SUBB A, @R0        ; 1/1 Subtract result from packed BCD 99
          ADD  A, #01        ; 2/2 Add one to result
          DA   A              ; 1/1 Decimal correct result
          MOV  @R0, A        ; 1/1 Store result
          INC  R0             ; 1/1 Increment pointer
          CJNZ R0, #20, LUP  ; 3/2 Test if finished and loop back if not
          SETB C              ; 1/1 Set carry to indicate negative result
          RET                ; 1/2 Return from subroutine (carry: negative result)
```

39 BYTES

91 CYCLES

PIC16C5X BENCHMARK #3—FOUR BYTE PACKED BCD SUBTRACTION PAGE 1 OF 2

This benchmark subtracts two eight digit packed BCD numbers (four bytes each) and replaces the minuend (first operand) with the result. This emulates an adding machine subtraction, where A - B replaces A. The benchmark is programmed as a subroutine, with the carry flag being used to indicate a positive or negative result (carry set for negative result). The PIC16C5X does not have any decimal adjust or decimal correct instructions, so the BCD correction must be program implemented. The correction algorithm for BCD subtraction consists of subtracting six whenever a nibble borrow occurs from the BCD subtraction result for each digit.

```

        STATUS EQU F3      ;      Status register
        FLAG EQU F27      ;
        BASE EQU F26      ;      1st operand @ F16-F19 2nd operand @ F20-F23
        CNTR EQU F25      ;
        TEMP EQU F24      ;
BCDSUBT: MOVLW 4           ; 1/1 Load W with byte count (length of field)
        MOVWF CNTR        ; 1/1 Store byte count in CNTR
        MOVLW 20          ; 1/1 Load W with address of 2nd operand
        MOVWF BASE        ; 1/1 Store address of 2nd operand in BASE
        BSF STATUS, 0     ; 1/1 Set carry (reset borrow) (STATUS register bit 0)
LOOP:   MOVF BASE, W      ; 1/1 2nd operand address to W
        MOVWF F4          ; 1/1 2nd operand address to F4 (indirect pointer)
        MOVF FO, W        ; 1/1 2nd operand to W
        MOVWF TEMP        ; 1/1 2nd operand to TEMP
        MOVLW 4           ; 1/1 Byte count to W
        SUBWF F4          ; 1/1 Subtract from pointer to get 1st operand address
        CALL CORRECT      ; 1/2 Call CORRECT subroutine
        MOVLW 4           ; 1/1 Byte count to W
        ADDWF F4          ; 1/1 Add to pointer to restore 2nd operand address
        DECFSZ CNTR       ; 1/1 Test if subtraction finished
        GOTO LOOP         ; 1/2 Loop back if subtraction not finished
        BITFSS STATUS, 0  ; 1/1 Test if output borrow (no carry)
        GOTO NEGR         ; 1/2 Branch to NEGR (neg. result) if output borrow
        BCF STATUS, 0     ; 1/1 Clear carry to indicate positive result
        RETLW 0           ; 1/2 Return from subroutine
NEGR:   MOVLW 4           ; 1/1 Load W with byte count (length of field)
        MOVWF CNTR        ; 1/1 Store byte count in CNTR
        MOVLW 16          ; 1/1 Load W with address of result (old 1st operand)
        MOVWF BASE        ; 1/1 Store address of result in BASE
        BSF STATUS, 0     ; 1/1 Set carry (reset borrow)
LUP:    MOVF BASE, W      ; 1/1 Result address to W
        MOVWF F4          ; 1/1 Result address to F4 (indirect pointer)
        MOVF FO, W        ; 1/1 Result to W
        MOVWF TEMP        ; 1/1 Result to TEMP
        CLRF FO           ; 1/1 Zero to result (old 1st operand)
        CALL CORRECT      ; 1/2 Call CORRECT subroutine
        DECFSZ CNTR       ; 1/1 Test if subtraction finished
        GOTO LUP         ; 1/2 Loop back if subtraction not finished
        BSF STATUS, 0     ; 1/1 Set carry to indicate negative result
        RETLW 0           ; 1/2 Return from subroutine
```

35 WORDS

PIC16C5X BENCHMARK #3—FOUR BYTE BCD SUBTRACTION (Continued)

```
CORRECT:  CLRf      FLAG          ; 1/1 Clear FLAG register
          MOVf      TEMP, W       ; 1/1 Move TEMP to W
          SUBWF    FO             ; 1/1 Subtract W from register
          CLRf      TEMP          ; 1/1 Clear TEMP
          BTFSC    STATUS, 0      ; 1/1 Test if carry (no borrow)
          GOTO     BYP1          ; 1/2 Branch if carry (no borrow)
          MOVLW    60H           ; 1/1 Load W with packed BCD 60
          ADDWF    TEMP          ; 1/1 Add W to TEMP
BYP1:     BSF      FLAG, 0        ; 1/1 Carry to FLAG bit 0
          BTFSC    STATUS, 1      ; 1/1 Test if digit carry (no digit borrow)
          GOTO     BYP2          ; 1/2 Branch if digit carry (no digit borrow)
          MOVLW    06H           ; 1/1 Load W with packed BCD 06
          ADDWF    TEMP          ; 1/1 Add W to TEMP
BYP2:     MOVf      TEMP, W       ; 1/1 Move TEMP to W
          BSF      STATUS, 0      ; 1/1 Set carry (reset borrow)
          SUBWF    FO             ; 1/1 Subtract W from previous result
          BTFSC    FLAG, 0        ; 1/1 Test if FLAG bit 0 reset
          BCF      STATUS, 0      ; 1/1 Clear carry (set borrow)
          INCF     BASE          ; 1/1 Increment BASE
          RETLW    0             ; 1/2 Return from CORRECT subroutine
```

20 WORDS

TOTAL: 55 WORDS (Equivalent to 82 1/2 BYTES)
274 CYCLES

COP8 BENCHMARK #4—THREE BYTE TABLE SEARCH

This benchmark searches a 200 byte table (resident in program memory) for a three byte character string, which may be resident anywhere in the lookup table (not necessarily on three byte boundaries). The type of return from subroutine (RETSK versus RET) indicates the success or failure of the search, with the address of the first byte of a matched string being returned in BASE. The benchmark is programmed as a subroutine, which should be located following the table since the LAID instructions in the subroutine must be located in the same 256 byte page of program memory.

```
CHAR1 = 00      ;      Three bytes of character string
CHAR2 = 01      ;      resident in registers 00, 01, 02
CHAR3 = 02      ;
SIZE = 0F0     ;
BASE = 0F1     ;
TEMP = 0F2     ;
TBLSRCH: LD     B, #SIZE ; 2/3 Address of SIZE to B pointer
LD     [B+], #198 ; 2/2 Table size (200) minus 2 to SIZE
LD     [B], #0 ; 2/2 Table base address of 0 to BASE
SEARCH: LD     A, [B+] ; 1/2 1st byte of table address to A
X     A, [B] ; 1/1 Save 1st byte address in TEMP
LD     A, [B] ; 1/1 Restore 1st byte table address to A
LAID ; 1/3 Load 1st of 3 test bytes from prog. memory table
IFEQ  A, CHAR1 ; 3/4 Test if 1st byte match
JP     SEARCH2 ; 1/3 First byte match
JP     FAIL ; 1/3 Fail if mismatch
SEARCH2: LD     A, [B] ; 1/1 Restore 1st byte address to A
INC    A ; 1/1 Increment address to get 2nd byte table address
X     A, [B] ; 1/1 Save 2nd byte address in TEMP
LD     A, [B] ; 1/1 Restore 2nd byte table address to A
LAID ; 1/3 Load 2nd of 3 test bytes from prog. memory table
IFEQ  A, CHAR2 ; 3/4 Test if 2nd byte match
JP     SEARCH3 ; 1/3 Second byte match
JP     FAIL ; 1/3 Fail if mismatch
SEARCH3: LD     A, [B] ; 1/1 Restore 2nd byte address to A
INC    A ; 1/1 Increment address to get 3rd byte table address
X     A, [B] ; 1/1 Save 3rd byte address in TEMP
LD     A, [B] ; 1/1 Restore 3rd byte table address to A
LAID ; 1/3 Load 3rd of 3 test bytes from prog. memory table
IFEQ  A, CHAR3 ; 3/4 Test if 3rd byte match
SUCCESS: RETSK ; 1/5 RETURN and SKIP if three byte match found
FAIL:  LD     A, [B-] ; 1/2 Decrement B pointer to select BASE
LD     A, [B] ; 1/1 Restore 1st byte address to A from BASE
INC    A ; 1/1 Increment 1st byte address
X     A, [B] ; 1/1 Save new 1st byte address in BASE
DRSZ  SIZE ; 1/3 Decrement SIZE and skip if table search finished
JMP   SEARCH ; 2/3 Continue table search
RET ; 1/5 RETURN if three byte match not found
```

*First search iteration fails with first byte 42 BYTES
mismatch, second search iteration successful 77 CYCLES*

68HC05 BENCHMARK #4—THREE BYTE TABLE SEARCH

This benchmark searches a 200 byte table (resident in program memory) for a three byte character string, which may be resident anywhere in the lookup table (not necessarily on three byte boundaries). The status of the carry bit indicates the success or failure of the search, with the address of the first byte of a matched string being returned in BASE. The benchmark is programmed as a subroutine.

```
TBLSRCH: LDA    #198      ; 2/2 Set up table size (200) minus 2
          STA    SIZE     ; 2/4 Save table size
          CLX                    ; 1/3 Set up table base address of 0
          STX    BASE     ; 2/4 Save table base address
SEARCH:  LDA    X         ; 2/4 Get first byte from table
          CMP    CHAR1    ; 2/3 Compare 1st byte with CHAR1
          BNE    FAIL     ; 2/3 Fail
          INCX                   ; 1/3 Set up address of 2nd byte from table
          LDA    X         ; 2/4 Get second byte from table
          CMP    CHAR2    ; 2/3 Compare 2nd byte with CHAR2
          BNE    FAIL     ; 2/3 Fail
          INCX                   ; 1/3 Set up address of 3rd byte from table
          LDA    X         ; 2/4 Get third byte from table
          CMP    CHAR3    ; 2/3 Compare 3rd byte with CHAR3
          BNE    FAIL     ; 2/3 Fail
SUCCESS: SEC                    ; 1/2 Set carry to indicate three byte match found
          RTS                    ; 1/6 Return
FAIL:   LDX    BASE     ; 2/3 Restore base address
          INCX                   ; 1/3 Increment base address to get new 1st byte address
          STX    BASE     ; 2/4 Save new base address
          CPX    SIZE     ; 2/3 Compare new base address with table size
          BNE    SEARCH    ; 2/3 Continue table search
          CLC                    ; 1/2 Reset carry to indicate no three byte match found
          RTS                    ; 1/6 Return
```

40 BYTES
80 CYCLES*

*First search iteration fails with first byte mismatch, second search iteration successful

80C51 BENCHMARK #4—THREE BYTE TABLE SEARCH

This benchmark searches a 200 byte table (resident in program memory) for a three byte character string, which may be resident anywhere in the lookup table (not necessarily on three byte boundaries). The status of the carry bit indicates the success or failure of the search, with the address of the first byte of a matched string being returned in DPTR. The benchmark is programmed as a subroutine.

```
TBLSRCH:  MOV      DPTR, #TBLBASE ; 3/2 Set up table starting address
          MOV      R2, #198      ; 2/1 Set up table size (200) minus 2
SEARCH:   CLR      A              ; 1/1 Clear accumulator
          MOVC     A, @A + DPTR  ; 1/2 Get 1st byte from program memory table
          CJNE    A, CHAR1, FAIL ; 3/2 Compare 1st byte with CHAR1
          MOV      A, #1         ; 2/1 Load accumulator with offset of 1
          MOVC     A, @A + DPTR  ; 1/2 Get 2nd byte from table
          CJNE    A, CHAR2, FAIL ; 3/2 Compare 2nd byte with CHAR2
          MOV      A, #2         ; 2/1 Load accumulator with offset of 2
          MOVC     A, @A + DPTR  ; 1/2 Get 3rd byte from table
          CJNE    A, CHAR3, FAIL ; 3/2 Compare 3rd byte with CHAR3
SUCCESS:  SET      C              ; 1/1 Set carry to indicate three byte match found
          RET                               ; 1/2 Return from subroutine
FAIL:    INC      DPTR          ; 1/2 Increment data pointer to get new
          ;                               1st byte address
          DJNZ    R2, SEARCH     ; 2/2 Decrement size and test if
          ;                               table search finished
          CLR     C              ; 1/1 Clear carry to indicate no three
          ;                               byte match found
          RET                               ; 1/2 Return from subroutine
```

29 BYTES
30 CYCLES*

*First search iteration fails with first byte mismatch, second search iteration successful

PIC16C5X BENCHMARK #4—THREE BYTE TABLE SEARCH

This benchmark searches a 200 word table (resident in program memory) for a three byte character string, which may be resident anywhere in the lookup table (not necessarily on three word boundaries). The status of the carry bit indicates the success or failure of the search, with the address of the first byte of a matched string being returned in OFFSET. The benchmark is programmed as a subroutine, which in turn calls the table header (TABLE) as a subroutine. This table header should immediately precede the 200 word table.

```
TBLSRCH:  MOVLW  198          ; 1/1 Set up table size (200) minus 2
           MOVWF  SIZE        ; 1/1 Save table size
           MOVLW  1           ; 1/1 Initialize table offset from table header
           MOVWF  OFFSET      ; 1/1 Save offset
SEARCH:   MOVF    W           ; 1/1 Load offset
           CALL   TABLE     ; 1/2 Get 1st byte from table
           SUBWF  CHAR1      ; 1/1 Subtract 1st test byte from CHAR1
           BTFSS STATUS, Z   ; 1/1 Test if result is zero (F3, bit 2)
           GOTO  FAIL1      ; 1/2 Fail if mismatch
           INCF  OFFSET      ; 1/1 Increment offset to set up 2nd byte
           MOVF  W           ; 1/1 Load offset
           CALL  TABLE     ; 1/2 Get 2nd byte
           SUBWF  CHAR2      ; 1/1 Subtract 2nd test byte from CHAR2
           BTFSS STATUS, Z   ; 1/1 Test if result is zero (F3, bit 2)
           GOTO  FAIL2      ; 1/2 Fail if mismatch
           INCF  OFFSET      ; 1/1 Increment offset to set up 3rd byte
           MOVF  W           ; 1/1 Load offset
           CALL  TABLE     ; 1/2 Get 3rd byte
           SUBWF  CHAR3      ; 1/1 Subtract 3rd test byte from CHAR3
           BTFSS STATUS, Z   ; 1/1 Test if result is zero (F3, bit 2)
           GOTO  FAIL3      ; 1/2 Fail if mismatch
SUCCESS:  DECF  OFFSET      ; 1/1 Decrement offset to return to 2nd byte
           DECF  OFFSET      ; 1/1 Decrement offset to return to 1st byte
           BSF  STATUS, C    ; 1/1 Set carry (F3, bit 0) to indicate match
                           found
           RETLW  0          ; 1/2 Return
FAIL3:   DECF  OFFSET      ; 1/1 Decrement offset to set up 2nd byte
           DECF  OFFSET      ; 1/1 Decrement offset to set up 1st byte
FAIL1:   INCF  OFFSET      ; 1/1 Increment offset to set up new 1st byte
FAIL2:   DECFSZ SIZE       ; 1/1 Decrement size and skip if table search
                           finished
           GOTO  SEARCH     ; 1/2 Continue table search
           BCF  STATUS, C   ; 1/1 Clear carry (F3, bit 0) to indicate no
                           match found
           RETLW  0          ; 1/2 Return

TABLE:   ADDWF  PC          ; 1/1 Add offset to PC
           RETLW  Data 1    ; 1/2 1st entry of data table
           RETLW  Data 2    ; -/- 2nd entry of data table
```

*First search iteration fails with first byte 34 WORDS (Equivalent to 51 BYTES)
mismatch, second search iteration successful 52 CYCLES*

```
RETLW  Program table entry 1 ; 1/2 This 200 word table contains 300 bytes
RETLW  Program table entry 2 ; 1/2
----  -----  ----
RETLW  Program table entry 200 ; 1/2
```

COP8 BENCHMARK #5—INPUT/OUTPUT MANIPULATION

This benchmark compares two 8-bit I/O ports P1 and P2. If they are equal, a nine is output as the least significant digit (lower nibble) of a third port P3. If port P1 is greater than port P2, then port P2 is output on port P1. If port P1 is less than port P2, then the most significant digit of Port P1 is copied to the least significant digit of Port P3.

```

PORTLD = ODO          ;
PORTLC = OD1          ;
PORTLI = OD2          ;
PORTGD = OD4          ;
PORTGC = OD5          ;
PORTGI = OD6          ;
PORTD = ODC           ;

PORTCMP: LD          B, #PORTLI    ; 2/3 Load B pointer (PORTL is P1)
          LD          X, #PORTGI    ; 2/3 Load X pointer (PORTG is P2)
          SC          ; 1/1 Initialize carry (no borrow) for subtraction
          LD          A, [X]        ; 1/1 Load P2
          SUBC        A, [B]        ; 1/1 Subtract P1 from P2
          IFC          ; 1/1 Test if P2 greater or equal to P1
          JP          POS          ; 1/3 Branch if result positive
NEG:     LD          A, [B-]        ; 1/2 Decrement B pointer
          LD          [B-], #OFF     ; 2/2 Configure PORTL (P1) as output port
          LD          A, [X]        ; 1/3 Load P2 to A
          X           A, [B]        ; 1/1 Output P2 to P1
          JP          FIN          ; 1/3 Jump to finish
POS:     IFEQ        A, #0          ; 2/2 Test if result zero
          JP          EQUAL         ; 1/3 Branch if zero
          LD          A, [B]        ; 1/1 Load P1 to A
          SWAP        A            ; 1/1 Swap nibbles of A
          X           A, PORTD      ; 2/3 Result to P3 (PORTD)
          JP          FIN          ; 1/3 Jump to finish
EQUAL:   LD          PORTD, #09     ; 3/3 Digit 9 to P3 (PORTD)
FIN:     ---

                                     26 BYTES

          P1 < P2          24 CYCLES
          P1 = P2          21 CYCLES
          P1 > P2          22 CYCLES
```

68HC05 BENCHMARK #5—INPUT/OUTPUT MANIPULATION

This benchmark compares two 8-bit I/O ports P1 and P2. If they are equal, a nine is output as the least significant digit (lower nibble) of a third port P3. If Port P1 is greater than port P2, then port P2 is output on Port P1. If Port P1 is less than Port P2, then the most significant digit of Port P1 is copied to the least significant digit of Port P3.

```

        PORTA      EQU $00      ;      Port P1
        PORTB      EQU $01      ;      Port P2
        PORTC      EQU $02      ;      Port P3
        DDRA       EQU $04      ;      PORTA configuration register
        DDRB       EQU $05      ;      PORTB configuration register
        DDRC       EQU $06      ;      PORTC configuration register
        TEMP       EQU $9F      ;      Temporary register

PORTCMP:  CLA                ; 1/3 Clear A
          STA      DDRA      ; 2/4 Configure Port A as input port (P1)
          STA      DDRB      ; 2/4 Configure Port B as input port (P2)
          DECA                ; 1/3 Decrement A to all ones
          STA      DDRC      ; 2/4 Configure Port C as output port (P3)
          LDA      PORTB     ; 2/3 Load A with Port P2 data
          CMP      A, PORTA  ; 2/3 Compare Port P1 data with Port P2 data
          BPL      POS       ; 2/3 Branch if comparison result (P2-P1) positive
NEG:      LDA      #$FF      ; 2/2 Load A with all ones
          STA      DDRA      ; 2/4 Configure Port A as output port (P1)
          LDA      PORTB     ; 2/3 Port P2 data to A
          STA      PORTA     ; 2/4 Output Port P2 data to Port P1
          BRA      FIN       ; 2/3 Branch to FIN
POS:      BEQ      EQUAL     ; 2/3 Branch if comparison result equal
          LDA      PORTA     ; 2/3 Port P1 data to A
          AND      A, #$F0   ; 2/2 Extract high order nibble of Port P1 data
          LSRA                ; 1/3 Logical shift A right one bit four times
          LSRA                ; 1/3 in order to shift upper nibble
          LSRA                ; 1/3 down into lower nibble position
          LSRA                ; 1/3
          STA      PORTC     ; 2/4 Output result to Port P3
          BRA      FIN       ; 2/3 Branch to FIN
EQUAL:    LDA      #9        ; 2/2 Load A with digit 9
          STA      PORTC     ; 2/4 Output digit 9 to Port P3
FIN:      ---                ; -/-

                                     42 BYTES

          P1 < P2              53 CYCLES
          P1 = P2              36 CYCLES
          P1 > P2              42 CYCLES
```

80C51 BENCHMARK #5—INPUT/OUTPUT MANIPULATION

This benchmark compares two 8-bit I/O ports P1 and P2. If they are equal, a nine is output as the least significant digit (lower nibble) of a third port P3. If Port P1 is greater than Port P2, then Port P2 is output on Port P1. If Port P1 is less than Port P2, then the most significant digit of Port P1 is copied to the least significant digit of Port P3.

```
PORTCMP:  MOV      P1, #0FF      ; 3/2 Configure Port P1 for input
          MOV      P2, #0FF      ; 3/2 Configure Port P2 for input
          CLR      C              ; 1/1 Clear carry
          MOV      A, P2          ; 2/1 Load A with Port P2 data
          SUBB     A, P1          ; 2/1 Subtract Port P1 data from A
          JNC     POS            ; 2/2 Jump TO POS if no carry from subtract
NEG:      MOV      A, P2          ; 2/1 Load A with Port P2 data
          MOV      P1, A          ; 2/1 Output Port P2 data to Port P1
          AJMP    FIN            ; 2/2 Jump to FIN
POS:     JZ       EQUAL          ; 2/2 Jump to EQUAL if subtraction result zero
          MOV      A, P1          ; 2/1 Load A with Port P1 data
          SWAP    A              ; 1/1 Swap nibbles of A
          ANL     A, #00F        ; 2/1 Extract lower nibble of A
          MOV      P3, A          ; 2/1 Output Port P1 upper nibble to Port P3
          AJMP    FIN            ; 2/2 Jump to FIN
EQUAL:   MOV      P3, #9         ; 3/2 Output digit 9 to Port P3
FIN:     ---                    ; -/-
```

33 BYTES

```
P1 < P2      17 CYCLES
P1 = P2      11 CYCLES
P1 > P2      13 CYCLES
```

PIC16C5X BENCHMARK # 5—INPUT/OUTPUT MANIPULATION

This benchmark compares two 8-bit I/O ports P1 and P2. If they are equal, a nine is output as the least significant digit (lower nibble) of a third Port P3. If Port P1 is greater than Port P2, then Port P2 is output on Port P1. If Port P1 is less than Port P2, then the most significant digit of Port P1 is copied to the least significant digit of Port P3.

```

        STATUS      EQU F3      ;      Status register
        PORTA       EQU F5      ;      Serves as Port P3
        PORTB       EQU F6      ;      Serves as Port P2
        PORTC       EQU F7      ;      Serves as Port P1
        TEMP        EQU F8      ;      Temporary register

PORTCMP:  MOVLW     FFH          ; 1/1 Load W with all ones
          TRIS     PORTB        ; 1/1 Tristate PORTB
          TRIS     PORTC        ; 1/1 Tristate PORTC
          MOVF     PORTB, W      ; 1/1 Load W with PORTB input
          MOVWF    TEMP         ; 1/1 Store PORTB input in TEMP
          MOVF     PORTC, W      ; 1/1 Load W with PORTC input
          BSF     STATUS, 0      ; 1/1 Set carry (bit 0 of STATUS)
          SUBWF   TEMP          ; 1/1 Subtract P1 from P2
          CLRW                    ; 1/1 Clear W
          TRIS     PORTA        ; 1/1 Configure PORTA as output port
          BTFSC   STATUS, 0      ; 1/1 Test if P2 less than P1
          GOTO    POS           ; 1/2 Branch to POS if test fails
NEG:      TRIS     PORTC        ; 1/1 Change PORTC to output port
          MOVF     PORTB, W      ; 1/1 Load W with PORTB (P2) input
          MOVWF    PORTC        ; 1/1 P2 input data output to PORTC (P1)
          GOTO    FIN           ; 1/2 Branch to FIN
POS:      BTFSC   STATUS, 2      ; 1/1 Test if subtraction result non-zero
          GOTO    EQUAL        ; 1/2 Branch to EQUAL if test fails
          MOVF     PORTC, W      ; 1/1 Load W with PORTC (P1) input
          MOVWF    TEMP         ; 1/1 P1 input to TEMP
          SWAPF   TEMP, W       ; 1/1 Swap nibbles of TEMP with result to W
          MOVWF    PORTA        ; 1/1 Output result from W to PORTA (P3)
          GOTO    FIN           ; 1/2 Branch to FIN
EQUAL:    MOVLW     9           ; 1/1 Digit 9 to W
          MOVWF    PORTA        ; 1/1 Digit 9 output to PORT A (P3)
FIN:      ---                    ; -/-
```

25 WORDS (Equivalent to 37 1/2 BYTES)

```

P1 < P2      21 CYCLES
P1 = P2      18 CYCLES
P1 > P2      17 CYCLES
```

COP8 BENCHMARK #6—SERIAL INPUT/OUTPUT WITH OFFSET TABLE

This benchmark inputs a sequence of byte couplets, with each couplet consisting of an 8-bit offset address followed by a data byte. The address is used to access a program memory data table to produce an offset value which is added to the second byte of the input couplet. The updated data byte is output while the address byte of the next couplet is being input. The benchmark is written as a subroutine, with the size of the input stream being an input parameter.

The program is written using the MICROWIRE/PLUS fast burst technique, which utilizes a 500 kHz burst clock SK (instruction cycle clock divided by 2). Each byte of the I/O stream utilizes nine burst clock periods, yielding an effective byte rate of 55.6 kHz.

*** Setting the Microwire busy bit to initialize the Microwire must occur at the same time in each alternating 18 cycle loop for this program to work. Alternate cycle definitions are as follows:

```

                Even cycle:   New address input for table lookup offset value
                               Previous data result output

                Odd cycle:   New data input added to offset value from table lookup
                               New data input returned as output

PORTGD = OD4      ; PORTG configuration register
PORTGC = OD5      ; PORTG data register
SIOR = OE9        ; Serial Input/Output register
CNTRL = OEE       ; Control register
PSW = OEF         ; Program Status Word
SIZE = OF0        ; Size of data stream
BUSY = 2          ; Microwire busy bit in PSW register

UPDATE: LD        PORTGC, #030 ; 3/3 Configure G4, G5 as outputs S0, SK to
LD        PORTGD, #0          ; 3/3 select uwire master mode (SI is G6)
INC       A                  ; 1/1 Increment byte count to compensate for one
X        A, SIZE             ; 2/3 byte throughput and store result in SIZE
LD        B, #CNTRL          ; 2/3 Set up B pointer for CNTRL register
LD        [B+], #8           ; 2/2 Select uwire master with divide by 2 clock
RC                               ; 1/1 Initialize carry
LOOP:   IFNC                ; 1/1 1 1 Test if no carry
JP        BYP1               ; 1/3 1 3 Branch if no carry
NOP                               ; 1/1 1 - NOP for delay compensation for 18 cycle loop
ADD       A, SIOR            ; 3/4 4 - Add uwire input data to table offset value
SBIT     BUSY, [B]          ; 1/1 1 - ***Set uwire BUSY bit to start microwire
RC                               ; 1/1 1 - Reset carry
JP        BYP2               ; 1/3 3 - Branch
BYP1:   X        A, SIOR      ; 2/3 - 3 New address input & previous result output
SBIT     BUSY, [B]          ; 1/1 - 1 ***Set uwire BUSY bit to start microwire
LAID                               ; 1/3 - 3 Offset table lookup from program memory
SC                               ; 1/1 - 1 Set carry
BYP2:   DRSZ     SIZE         ; 1/3 3 3 Decrement SIZE and test if result zero
JP        LOOP              ; 1/3 3 3 Loop back if zero test fails
RET                               ; 1/5 Return from subroutine

31 BYTES
18 CYCLES per Loop
```

M68HC05 BENCHMARK #6—SERIAL INPUT/OUTPUT WITH OFFSET TABLE

This benchmark inputs a sequence of byte couplets, with each couplet consisting of an 8-bit offset address followed by a data byte. The address is used to access a data table to produce an offset value which is added to the second byte of the input couplet. The updated data byte is output while the address byte of the next couplet is being input. The benchmark is written as a subroutine, with the size of the input stream being an input parameter.

The program is written using the SPI with the divide by 2 fast clock option selected, which yields a 1 MHz burst clock SCK (instruction cycle clock divided by 2). Each byte of the I/O stream utilizes 27 burst clock periods (equivalent to 13.5 μ s), yielding an effective byte rate of 74.1 kHz.

***Setting the SPI system enable bit SPE to initialize the SPI must occur at the same time in each alternating 27 cycle loop for this program to work. Alternate cycle definitions are as follows:

```

                Even cycle:  New address input for table lookup offset value
                               Previous data result output
                Odd cycle:   New data input added to offset value from table lookup
                               New data input returned as output

PORTD EQU $03      ; SPI Interface port
SPCR  EQU $0A      ; Serial Peripheral Control Register
SPSR  EQU $0B      ; Serial Peripheral Status Register
SPDR  EQU $0C      ; Serial Peripheral Data I/O Register

UPDATE: INCA      ; 1/3 Increment byte count to compensate for one
          STA A, SIZE ; 2/4 byte throughput and store result in SIZE
          LDA #$040 ; 2/2 Set up data for SPCR register
          STA SPCR ; 2/4 Enable SPI master with divide by 2 clock
          CLC      ; 1/2 Initialize carry
LOOP:    BCC BYP1 ; 2/3 2 3 Test and branch if no carry
          ADD SPDR ; 2/3 3 - Add SPI input data to table offset value in A
          STA SPDR ; 2/4 4 - Output result to SPDR for SPI output
          BSET 6, SPCR ; 2/5 5 - ***Turn on SPI system enable bit
          CLC      ; 1/2 2 - Clear carry
          BRA BYP2 ; 2/3 3 - Branch
BYP1:    LDA X      ; 1/3 - 3 NOP for delay compensation for 27 cycle loop
          LDX SPDR ; 2/3 - 3 SPI input data address to index register
          BSET 6, SPCR ; 2/5 - 5 ***Turn on SPI system enable bit
          LDA X      ; 1/3 - 3 Load A with table lookup offset value
          SEC      ; 1/2 - 2 Set carry
BYP2:    DEC SIZE   ; 2/5 5 5 Decrement SIZE
          BNE LOOP ; 2/3 3 3 Test and branch if result equal to zero
          RTS      ; 1/6 Return from subroutine

                31 BYTES
                27 CYCLES per Loop
```

80C51 BENCHMARK #6—SERIAL INPUT/OUTPUT WITH OFFSET TABLE

This benchmark inputs a sequence of byte couplets, with each couplet consisting of an 8-bit offset address followed by a data byte. The address is used to access a program memory data table to produce an offset value which is added to the second byte of the input couplet. The updated data byte is output while the address byte of the next couplet is being input. The benchmark is written as a subroutine, with the size of the input stream being an input parameter.

The program is written using the 8-bit Shift Register Mode 0, with the serial data being either transmitted or received (not both simultaneously) with the least significant bit first. The clock rate for this mode is 1/12 that of the on-chip oscillator frequency of 12 MHz, which equates to an I/O baud rate of 1 MHz. This is equivalent to the instruction cycle time of 1 μ s. In Shift Register Mode 0, the RXD line is used for both data input and output, while the TXD line is used for the output clock. Consequently, the terms "RXD" and "TXD" are misleading in this mode of serial I/O.

The program takes 52 instruction cycles per couplet loop, which equates to 26 instruction cycles (26 μ s) per byte. This yields an effective byte rate of 38.5 kHz.

```

                SIZE      EQU R2      ;

UPDATE:  MOV      SIZE, A      ; 1/1 Save and increment byte count to
         INC      SIZE        ; 1/1   compensate for one byte throughput
         MOV      DPTR, #TBLBASE ; 3/2 Set up table starting address
LOOP:    CLR      C           ; 1/1 Clear carry
SINP:    MOV      SCON, #010H  ; 3/2 Initialize SCON for Mode 0 and enable receive
STALL1: JNB      SCON.RI, STALL1 ; 3/2 Wait for receiving to finish
         JC       BYP         ; 2/2 Jump if carry
         MOV      A, SBUF      ; 2/1 Read SBUF to get data table address
         MOVC     A, @A + DPTR ; 1/2 Offset value from program memory
                                data table lookup
         SETB     C           ; 1/1 Set carry
         AJMP    SINP        ; 2/2 Jump to SINP
BYP:     ADD      A, SBUF      ; 2/1 Add input data from SBUF to offset value
         MOV      SCON, #0     ; 3/2 Reinitialize SCON for Mode and no receive
SOUT:    MOV      SBUF, A      ; 2/1 Result to SBUF starts serial output
STALL2: JNB      SCON.TI, STALL2 ; 3/2 Wait for transmitting to finish
         CLR      SCON.TI     ; 2/1 Clear TI (transmit interrupt) flag
         DJNZ    SIZE, LOOP   ; 2/2 Decrement & test SIZE and loop back
                                if not finished
         RET                    ; 1/2 Return from subroutine

                35 BYTES
                52 CYCLES per couplet Loop
```

PIC16C5X BENCHMARK #6—SERIAL INPUT/OUTPUT WITH OFFSET TABLE

This benchmark inputs a sequence of byte couplets, with each couplet consisting of an 8-bit offset address followed by a data byte. The address is used to access a program memory data table to produce an offset value which is added to the second byte of the input couplet. The updated data byte is output while the address byte of the next couplet is being input. The benchmark is written as a subroutine, with the size of the input stream being an input parameter.

The program is written using a transmit/receive subroutine and takes 214 instruction cycles per couplet loop. This equates to 107 instruction cycles (53.5 μ s) per byte, yielding an effective byte rate of 18.7 kHz.

```
UPDATE:  MOVWF    SIZE      ; 1/1 Save and increment byte couplet count to
         INCF    SIZE      ; 1/1 compensate for one byte throughput
         CLRF   XRDATA    ; 1/1 Clear data register
         MOVLW   2        ; 1/1 Set up bit select for input
         TRIS   PORTB     ; 1/1 Tristate bit 2 of PORTB
LOOP:    CALL   XMITREC    ; 1/2 Call XMITREC transmit/receive subroutine
         MOVF   XRDATA, W  ; 1/1 Move offset table address to W
         CALL   TABLE     ; 1/2 Call TABLE (table lookup subroutine)
         CALL   XMITREC    ; 1/2 Call XMITREC subroutine
         ADDWF  XRDATA     ; 1/1 Add offset to received data
         DECFSZ SIZE      ; 1/1 Decrement SIZE and test if zero
         GOTO  LOOP       ; 1/2 Branch back if not finished
         RETLW  0         ; 1/2 Return from subroutine

XMITREC: MOVLW   8        ; 1/1 Set up bit count
         MOVWF  BITCNT    ; 1/1 Bit count to BITCNT
LUP:    BCF    PORTB, 0    ; 1/1 Reset clock output bit
         BCF    PORTB, 1    ; 1/1 Reset data output bit
         RRF    XRDATA     ; 1/1 Rotate right through carry
         BTFSC  F3, 0      ; 1/1 Test carry bit and skip if clear
         BSF    PORTB, 0    ; 1/1 Set data output bit
         BSF    PORTB, 1    ; 1/1 Set clock output bit
         BCF    XRDATA, 7   ; 1/1 Clear upper bit of XRDATA
         BTFSC  PORTB, 2    ; 1/1 Test receive bit
         BSF    XRDATA, 7   ; 1/1 Set upper bit of XRDATA if receive bit high
         DECFSZ BITCNT     ; 1/1 Decrement count and skip if zero
         GOTO  LUP        ; 1/2 Branch back to XMITREC
         BCF    PORTB, 1    ; 1/1 Reset clock output bit
         RETLW  0         ; 1/2 Return from subroutine

TABLE:  ADDWF   PC        ; 1/1 Add offset to PC
         RETLW  Data 1    ; 1/2 1st entry of data table
         RETLW  Data 2    ; -/- 2nd entry of data table
         ----      ; -/- -----
```

30 WORDS (Equivalent to 45 BYTES)
214 CYCLES per couplet Loop

COP8 BENCHMARK #7—TIMEKEEPING

This timekeeping benchmark is interrupt driven, using 5 ms. timer cycle interrupts. The program emulates a real time clock, keeping track of hours, minutes, and seconds in packed BCD format.

TIMER SETUP ROUTINE:

```
      LO = 088           ;      5 ms low component for timer
      HI = 013           ;      5 ms high component for timer
      TMRLO = OEA       ;      Timer low byte
      TMRHI = OEB       ;      Timer high byte
      TAULO = OEC       ;      Autoreload register low byte
      TAUHI = OED       ;      Autoreload register high byte
      CNTRL = OEE       ;      CNTRL control register
      PSW = OEF         ;      PSW control register
      CNTR = OF0        ;      5 ms counter
      HOUR = OF1        ;      Hour register (packed BCD format)
      MIN = OF2         ;      Minute register (packed BCD format)
      SEC = OF3         ;      Second register (packed BCD format)
      TEMP = OF4        ;      Temporary register

TSETUP:  LD      B, #TAULO      ; 2/3 Load B pointer with address of low autoreload
         LD      [B+], #LO     ; 2/3 Load low autoreload reg with 5 ms component
         LD      [B+], #HI     ; 2/2 Load high autoreload reg with 5 ms component
         LD      [B+], #080    ; 2/2 Set up timer PWM mode in CNTRL register
         LD      [B+], #011    ; 2/2 Set up timer interrupt in PSW register
         LD      [B+], #200    ; 2/2 Initialize 5 ms counter to count one second
         LD      [B+], #1      ; 2/2 Initialize HOUR to 1
         LD      [B+], #0      ; 2/2 Initialize MIN to 0
         LD      [B], #0       ; 2/2 Initialize SEC to 0
         SBIT    4, CNTRL      ; 3/4 Start timer
STALL:   JP      STALL        ; -/- Loop back on self to simulate main program
```

21 BYTES

COP8 BENCHMARK #7—TIMEKEEPING (Continued)

TIMER INTERRUPT SERVICE ROUTINE:

```

TIMEKEEP: DRSZ      CNTR          ; 1/3  Decrement 5 ms counter
          RETI       ; 1/5  Return from timer interrupt
          LD        CNTR, #200    ; 2/3  Reload 5 ms counter
          LD        B, #TEMP      ; 2/3  Load B pointer with address of TEMP
          X         A, [B-]      ; 1/2  Save A in TEMP
          JSR       INCBBCD      ; 2/5  Call INCBBCD subroutine to increment SEC
          RETI       ; 1/5  Return from timer interrupt
          JSR       INCBBCD      ; 2/5  Call INCBBCD subroutine to increment MIN
          RETI       ; 1/5  Return from timer interrupt
          SC        ; 1/1  Set carry for BCD increment of HOUR
          LD        A, #066      ; 2/2  Set up BCD increment
          ADC       A, [B]       ; 1/1  Increment A in BCD
          DCOR      ; 1/1  Decimal correct result of BCD increment
          X         A, [B]       ; 1/1  Store result in HOUR
          IFEQ      A, #012      ; 2/2  Test if result was BCD 13 (previous result 12)
          LD        [B], #1      ; 2/2  Reset HOUR from BCD 13 to BCD 1
          LD        A, TEMP      ; 2/3  Restore A from TEMP
          RETI       ; 1/5  Return from timer interrupt

INCBBCD:  SC        ; 1/1  Set carry for BCD increment
          LD        A, #066      ; 2/2  Set up BCD increment
          ADC       A, [B]       ; 1/1  Increment A in BCD
          DCOR      ; 1/1  Decimal correct result of BCD increment
          X         A, [B]       ; 1/1  Store result
          IFEQ      A, #059      ; 2/2  Test if result was BCD 60 (previous result 59)
          JP        RESTORE      ; 1/3  Jump to RESTORE if test successful
          RET       ; 1/5  Return from subroutine

RESTORE:  LD        [B-], #0     ; 2/3  Reset to zero
          RETSK      ; 1/5  Return from subroutine and skip

```

39 BYTES

TOTAL: 60 BYTES
80 CYCLES*

*Case where 12:59:59 is advancing to 1:00:00

M68HC05 BENCHMARK #7—TIMEKEEPING

This timekeeping benchmark is interrupt driven, using the timer output compare interrupt (in conjunction with the 131.072 ms timer cycle). The program emulates a real time clock, keeping track of hours, minutes, and seconds in packed BCD format.

The timer clock is equivalent to the instruction cycle clock (2 MHz) divided by 4, yielding a timer clock period of 2 μ s. The 16-bit free running timer has a maximum of 65536 counts, which at 2 μ s per count equates to a timer cycle of 131.072 ms. One second divided by the timer cycle (1,000,000 divided by 131,072) yields 7 cycles with a residual of 82.496 ms. This residual equates to 41248 timer counts, which represents the value (A120 in hex) added to the output compare register during each timer interrupt service.

TIMER SETUP ROUTINE:

```

        OCIE  EQU 6      ;      Output compare interrupt bit in TCR (timer control reg)
        CNTR  EQU 10     ;
        TEMP  EQU 11     ;
        ATEMP EQU 12     ;
        SEC   EQU 20     ;
        MIN   EQU 21     ;
        HOUR  EQU 22     ;

TSETUP: CLR   SEC        ; 2/5 Initialize SEC to 0
        CLR   MIN        ; 2/5 Initialize MIN to 0
        CLR   HOUR       ; 2/5 Clear HOUR to 1
        INC   HOUR       ; 2/5 Initialize HOUR to 1
        LDA   #7         ; 2/2 Load counter initialization value
        STA   CNTR       ; 2/4 Initialize timer loop counter
        BSET  OCIE, TCR  ; 2/5 Enable Output Compare Interrupt
STALL:  JMP   STALL      ; -/- Loop back on self to simulate main program
```

14 BYTES

M68HC05 BENCHMARK #7—TIMEKEEPING (Continued)

```

TIMER INTERRUPT SERVICE ROUTINE:      ;      OCOMP = OUTPUT COMPARE

TIMEKEEP: DEC      CNTR      ; 2/5  Decrement 5 ms counter
          BNE      FINI      ; 2/3  Branch if result zero
          STA      TEMP      ; 2/4  Save A in TEMP
          LDA      #7        ; 2/2  Load A with counter reload value
          STA      CNTR      ; 2/4  Reload timer loop counter
          LDA      OCMPL0    ; 2/3  Read OCMPL0
          ADD      #20       ; 2/2  ADD low byte of offset count
          STA      ATEMP     ; 2/4  Store in ATEMP until OCMPHI is updated
          LDA      OCMPHI    ; 2/3  Read OCMPHI
          ADC      #A1       ; 2/2  Add high byte of offset count
          STA      OCMPHI    ; 2/4  Update OCMPHI with new offset
          LDA      TSR       ; 2/3  Read TSR to clear OCF flag
          LDA      ATEMP     ; 2/3  Retrieve update for OCMPL0 from ATEMP
          STA      OCMPL0    ; 2/4  Update OCMPL0 with new offset
          LDX      #20       ; 2/2  Load index register with address of SEC
          JSR      INCB CD   ; 2/5  Call INCB CD subroutine
          BCC      FIN       ; 2/3  Branch if carry clear
          JSR      INCB CD   ; 2/5  Call INCB CD subroutine
          BCC      FIN       ; 2/3  Branch if carry clear
          JSR      BCDFIX    ; 2/5  Call BCDFIX subroutine
          CMP      A, #013   ; 2/2  Compare A with BCD 13
          BNE      FIN       ; 2/3  Branch if not equal
          CLR      X         ; 1/5  Clear result
          INC      X         ; 1/5  Set HOUR to 1
FIN:      LDA      TEMP     ; 2/3  Restore A from TEMP
FINI:     RTI              ; 1/9  Return from interrupt

INCB CD:  JSR      BCDFIX    ; 2/5  Call BCDFIX subroutine
          CMP      A, #060   ; 2/2  Compare A with BCD 60
          BNE      BYP       ; 2/3  Branch if not equal
          CLR      X         ; 1/5  Clear result (SEC or MIN) to zero
          SEC          ; 1/2  Set carry
          INCX        ; 1/3  Increment index
BYP:      RTS              ; 1/5  Return from subroutine

BCDFIX:  LDA      X         ; 1/3  Load A with operand (SEC or MIN)
          ADD      #1        ; 2/2  Add 1 to A
          STA      X         ; 1/4  Store A in result
          AND      A, #0F    ; 2/2  Extract low order BCD digit
          CMP      A, #10    ; 2/2  Compare result with 10
          BNE      BYPASS    ; 2/3  Branch if not equal
          LDA      #010     ; 2/2  Load A with BCD 10
          STA      X         ; 1/4  Store BCD 10 in result
BYPASS:  RTS              ; 1/5  Return from subroutine

```

*Case where 12:59:59 is advancing to 1:00:00 73 BYTES

```

TOTAL:      87 BYTES
           218 CYCLES*

```

80C51 BENCHMARK #7—TIMEKEEPING

This timekeeping benchmark is interrupt driven, using 250 μ s timer cycle interrupts. The program emulates a real time clock, keeping track of hours, minutes, and seconds in packed BCD format.

TIMER SETUP ROUTINE:

```

        CNTR1    EQU R2        ;
        CNTR2    EQU R3        ;
        TEMP     EQU R4        ;
        SEC      EQU 20        ;
        MIN      EQU 21        ;
        HOUR     EQU 22        ;

TSETUP:  MOV      CNTR1, #20    ; 2/1 Initialize 250  $\mu$ s counter to 20
        MOV      CNTR2, #200   ; 2/1 Initialize 5 ms counter to 200
        MOV      SEC, #0       ; 2/1 Initialize SEC to 0
        MOV      MIN, #0       ; 2/1 Initialize MIN to 0
        MOV      HOUR, #1      ; 2/1 Initialize HOUR to 1
        MOV      TMOD, #02H    ; 3/2 Select Timer 0, Mode 2
        MOV      TH0, #-250    ; 3/2 Setup 250  $\mu$ s delay for Timer 0
        SETB     TRO           ; 2/1 Start Timer 0
        MOV      IE, #82H      ; 3/2 Enable Timer 0 interrupt
LOOP:    SJMP     LOOP         ; -/- Loop back on self to simulate main program
```

21 BYTES

80C51 BENCHMARK #7—TIMEKEEPING (Continued)

TIMER INTERRUPT SERVICE ROUTINE:

```

TIMEKEEP: DJNE      CNTRL, FIN      ; 2/2 Decrement 250 μs counter and jump if not zero
           MOV       CNTRL, #20     ; 2/1 Reload 250 μs counter
           DJNE      CNTR2, FIN     ; 2/2 Decrement 5 ms counter and jump if not zero
           MOV       CNTR2, #200    ; 2/1 Reload 5 ms counter
           MOV       TEMP, A        ; 1/1 Save A in TEMP
           MOV       RO, #20        ; 2/1 Load RO with address of SEC
           ACALL     INCBBCD        ; 2/2 Call INCBBCD subroutine
           JNC       FIN           ; 2/2 Jump if no carry
           ACALL     INCBBCD        ; 2/2 Call INCBBCD subroutine
           JNC       FIN           ; 2/2 Jump if no carry
           MOV       A, @RO        ; 1/1 Move HOUR to A
           CLR       C             ; 1/1 Initialize carry
           ADD       A, #1         ; 2/1 Add 1 to A
           DA        A            ; 1/1 Decimal adjust A for BCD correction
           MOV       @RO, A        ; 1/1 Return A to HOUR
           CJNE     A, #013H, FIN   ; 3/2 Compare and jump if hour not equal to 13
           MOV       @RO, #1       ; 2/1 Correct HOUR to 1
FIN:       MOV       A, TEMP       ; 1/1 Restore A from TEMP
           RETI      ; 1/2 Return from timer interrupt

INCBBCD:  MOV       A, @RO        ; 1/1 Move operand to A (SEC or MIN)
           ADD       A, #1         ; 2/1 Add 1 to A
           DA        A            ; 1/1 Decimal adjust A for BCD correction
           MOV       @RO, A        ; 1/1 Return A to result register (SEC or MIN)
           CJNE     A, #060H, BYPASS; 3/2 Compare and jump if result not equal to 60
           MOV       @RO, #0       ; 2/1 Correct result to zero (SEC or MIN)
           SETB     C             ; 1/1 Set carry to indicate corrected result
BYPASS:   INC       RO           ; 1/1 Increment indirect address data pointer
           RET        ; 1/2 Return from subroutine

```

45 BYTES

TOTAL: 66 BYTES
49 CYCLES*

*Case where 12:59:59 is advancing to 1:00:00

PIC16C5X BENCHMARK #7—TIMEKEEPING

This timekeeping benchmark uses a pseudo software 5 ms delay loop as the basic core timing for emulating a real time clock. Note that the PIC16C5X does not have any hardware interrupts. Consequently, the program must keep track of when the counter RTCC overflows for real time applications. The RTCC F1 file register is only 8 bits, but with an 8-bit prescaler selected (1 : 256 RTCC rate), the RTCC emulates a 16-bit timer. The program keeps track of hours, minutes, and seconds in packed BCD format.

5 ms timing analysis: $0.5 \mu\text{s}$ (instruction cycle time) \times 256 (prescaler) = 128 μs RTCC increment rate

5 ms emulation: $5000/128 = 39$ cycles + residue of 8 μs

```
RTCC      EQU 1      ; Register File F1
STATUS    EQU 3      ; Reg File F3
TEMP      EQU 20     ; Reg File F20
SEC       EQU 21     ; Reg File F21
MIN       EQU 22     ; Reg File F22
HOUR      EQU 23     ; Reg File F23
CNTR      EQU 24     ; Reg File F24

TIMEKEEP: MOVLW      7      ; 1/1 Set up data for option register
          OPTION     ; 1/1 Select a 1:256 prescaler with RTCC
          CLRF      SEC    ; 1/1 Initialize SEC to 0
          CLRF      MIN    ; 1/1 Initialize MIN to 0
          CLRF      HOUR   ; 1/1 Clear HOUR
          INCF      HOUR   ; 1/1 Initialize HOUR TO 1
          MOVLW     217    ; 1/1 256-39 = 217 (RTCC counter increments)
          MOVWF     RTCC   ; 1/1 Set up RTCC for 39 counts until overflow
STALL:    GOTO      STALL  ; -/- Loop back on self to simulate main program
```

8 WORDS

Main program must return within 5 ms to sense exactly when RTCC counter overflows !!!

PIC16C5X BENCHMARK #7—TIMEKEEPING (Continued)

```

TSTOVFLW:  BTFSC      RTCC, 7      ; 1/1 Test if RTCC has overflowed
           GOTO      TSTOVFLW     ; 1/2 Loop back if test fails
           MOVLW     217          ; 1/1 256-39 = 217
           MOVWF     RTCC         ; 1/1 Set up RTCC for 39 counts until overflow
           DECFSZ    CNTR         ; 1/1 Decrement CNTR and test for zero result
           GOTO      STALL        ; 1/2 Go to MAIN program
           MOVLW     200          ; 1/1 Set up 200 (200 × 5 ms = 1 sec)
           MOVWF     CNTR        ; 1/1 Initialize 5 ms counter
           MOVLW     SEC          ; 1/1 Address of SEC to W
           MOVWF     F4          ; 1/1 Address of SEC to F4
           CALL      INCBDCD      ; 1/2 Call INCBDCD subroutine to increment SEC
           BTFSS     F3, 2       ; 1/1 Test and skip if zero result (Z is F3 bit 2)
           GOTO      STALL        ; 1/2 Go to MAIN program
           CALL      INCBDCD      ; 1/2 Call INCBDCD subroutine to increment MIN
           BTFSS     F3, 2       ; 1/1 Test and skip if zero result
           GOTO      STALL        ; 1/2 Go to MAIN program
           INCF      HOUR        ; 1/1 Increment hour
           MOVLW     10          ; 1/1 Move 10 to W
           SUBWF     HOUR, W      ; 1/1 HOUR - 10 to W
           MOVLW     6           ; 1/1 Move 6 to W
           BTFSC     F3, 2       ; 1/1 Test if subtraction result not zero
           ADDWF     HOUR        ; 1/1 Add 6 to HOUR for BCD correction
           MOVLW     013H       ; 1/1 Move BCD 13 to W
           SUBWF     HOUR, W      ; 1/1 HOUR - BCD 13 to W
           MOVLW     1           ; 1/1 Move 1 to W
           BTFSC     F3, 2       ; 1/1 Test and skip if subtraction result not zero
           MOVWF     HOUR        ; 1/1 Move 1 to hour if subtraction result zero
           GOTO      STALL        ; 1/2 Go to MAIN program

INCBDCD:   INCF      FO          ; 1/1 Increment BCD data operand
           MOVF      FO, W       ; 1/1 Move data to W
           MOVWF     TEMP        ; 1/1 Move data to TEMP
           MOVLW     0FH        ; 1/1 Move hex 0F to W
           ANDWF     TEMP, W     ; 1/1 Extract low order BCD digit
           MOVLW     10         ; 1/1 Move 10 to W
           BSF      F3, 0       ; 1/1 Set carry (reset borrow for subtraction)
           SUBWF     TEMP, W     ; 1/1 TEMP - 10 to TEMP
           MOVLW     6          ; 1/1 Move 6 to W
           BTFSC     F3, 2       ; 1/1 Test if subtraction result zero (Z is F3 bit 2)
           ADDWF     FO          ; 1/1 Add 6 to correct BCD operand if test successful
           MOVLW     060H       ; 1/1 Move BCD 60 to W
           BSF      F3, 0       ; 1/1 Set carry (reset borrow for subtraction)
           SUBWF     FO, W       ; 1/1 BCD operand - BCD 60 to W
           BTFSC     F3, 2       ; 1/1 Test if subtraction result zero
           CLRF     FO          ; 1/1 Reset BCD operand to zero if test successful
           INCF     F4          ; 1/1 Increment indirect address pointer
           RETLW     0          ; 1/2 Return from subroutine
    
```

*Case where 12:59:59 is advancing to 1:00:00 46 WORDS

```

TOTAL:      54 WORDS (Equivalent to 81 BYTES)
            69 CYCLES*
    
```

COP8 BENCHMARK #8—SWITCH ACTIVATED FIVE SECOND LED

This benchmark samples a switch input to activate a five second output for turning on an LED. The switch is debounced with a 50 ms delay both on opening and closure. Once activated, the switch will turn on an LED output for five seconds and then turn it off, regardless of whether or not the switch is still activated. Once the switch is turned off, the procedure is repeated. Both the switch input and the LED output are low true.

```

    CNTR1 = OF0      ;      Three Counters in registers OF0, OF1, OF2
    CNTR2 = OF1      ;
    CNTR3 = OF2      ;
    PORTLD = OD1     ;      PORTL Data register
    PORTLC = OD2     ;      PORTL Configuration register
    PORTLI = OD3     ;      PORTL Input address

LED5:  LD      B, #PORTLD ; 2/3 PORTL data register address to B pointer
       LD      [B+], #OF0 ; 2/2 Set upper 4 bits of data register for low true output
       LD      [B+], #OF0 ; 2/2 Configure L port for upper nibble output and
                               ;      lower nibble input (incr B ptr to port input addr)

WSWON: IFBIT    0, [B]    ; 1/1 Sample input switch (low true)
       JP      WSWON     ; 1/3 Wait for Switch ON

SWON:  JSR     DLY50     ; 2/5 Debounce 50 ms
       RBIT    7, PORTLD ; 3/4 Turn on LED (low true)
       LD      CNTR1, #100 ; 2/3 Call DLY50 50 ms subroutine 100 times
SEC5:  JSR     DLY50     ; 2/5 to get 5 second LED ON time
       DRSZ   CNTR1     ; 1/3 Decrement CNTR1 and test for zero result
       JP      SEC5     ; 1/3 Loop back until count finished
       SBIT    7, PORTLD ; 3/4 Turn off LED (low true)

WSWOFF: IFBIT   0, [B]   ; 1/1 Sample input switch (low true)
        JP     SWOFF    ; 1/3 Jump to Switch OFF
        JP     WSWOFF   ; 1/3 Wait for Switch OFF

SOFF:  JSR     DLY50     ; 2/5 Debounce 50 ms
        JP     WSWON    ; 1/3 Repeat procedure (wait for Switch ON)

DLY50: LD      CNTR2, #33 ; 2/3 Set up outer loop count
       LD      CNTR3, #118 ; 2/3 Set up initial inner loop count
LOOP:  DRSZ   CNTR3     ; 1/3 Decrement inner loop count and test for zero
       JP     LOOP     ; 1/3 Loop back until inner count finished
       DRSZ   CNTR2     ; 1/3 Decrement outer loop count and test for zero
       JP     LOOP     ; 1/3 Loop back until outer count finished
       RET                      ; 1/5 Return from subroutine
```

*Cycle times without wait loops37 BYTES
76 CYCLES*

DLY50 TIMING ANALYSIS: CYCLES

```

Initial (2 X 3)      6
1 x (117 x 6 + 1 x 10)712
32 x (255 x 6 + 1 x 10)9280
Terminating (5 - 2)  3
```

TOTAL 50001 Equivalent to 50.001 ms @ 1 μ s per cycle

M68HC05 BENCHMARK #8—SWITCH ACTIVATED FIVE SECOND LED

This benchmark samples a switch input to activate a five second output for turning on an LED. The switch is debounced with a 50 ms delay both on opening and closure. Once activated, the switch will turn on an LED output for five seconds and then turn it off, regardless of whether or not the switch is still activated. Once the switch is turned off, the procedure is repeated. Both the switch input and the LED output are low true.

```

PORTB EQU $01 ;
DDRB EQU $05 ;
TEMP EQU $9F ;

LED5: LDA #$F0 ; 2/2 Output data and configuration to A
      STA PORTB ; 2/4 Set upper 4 bits of PORTB for low true output
      STA DDRB ; 2/4 Configure PORTB for upper nibble output
      ; and lower nibble input

WSWON: BRSET 0, PORTB, WSWON ; 3/5 Sample input switch (low true)
SWON: JSR DLY50 ; 3/6 Debounce 50 ms
      BCLR 7, PORTB ; 2/5 Turn on LED (low true)
      LDA #100 ; 2/2 Call DLY50 50 ms subroutine 100 times
SEC5: JSR DLY50 ; 3/6 to get 5 second LED ON time
      DECA ; 1/3 Decrement count
      BNE SEC5 ; 2/3 Loop back until count finished
      BSET 7, PORTB ; 2/5 Turn off LED (low true)
WSWOFF: BRCLR 0, PORTB, WSWOFF ; 3/5 Sample input switch (low true)
SWOFF: JSR DLY50 ; 3/6 Debounce 50 ms
      BRA WSWON ; 2/3 Repeat procedure (wait for Switch ON)

DLY50: STA TEMP ; 2/4 Save A
      LDA #65 ; 2/2 Set up outer loop count
      LDX #226 ; 2/2 Set up inner loop count
LOOP: DECX ; 1/3 Decrement inner loop count
      BNE LOOP ; 2/3 Loop back if non-zero
      DECA ; 1/3 Decrement outer loop count
      BNE LOOP ; 2/3 Loop back if non-zero
      LDA TEMP ; 2/3 Restore A
      RTS ; 1/6 Return from subroutine

```

47 BYTES
88 CYCLES*

*Cycle times without wait loops

```

DLY50 TIMING ANALYSIS:          CYCLES

Initial (4 + 2 + 2)            8
1 x (225 x 6 + 1 x 11)        1361
64 x (255 x 6 + 1 x 11)       98624
Terminating (3 + 6 - 1)        8

TOTAL                          100001 Equivalent to 50.000 ms @ 0.5 μs per cycle

```

80C51 BENCHMARK #8—SWITCH ACTIVATED FIVE SECOND LED

This benchmark samples a switch input to activate a five second output for turning on an LED. The switch is debounced with a 50 ms delay both on opening and closure. Once activated, the switch will turn on an LED output for five seconds and then turn it off, regardless of whether or not the switch is still activated. Once the switch is turned off, the procedure is repeated. Both the switch input and the LED output are low true.

```
LED5:  MOV    P1, #00F      ; 3/2 Configure lower 4 bits of port P1 for input
WSWON:  JB     P1.0, WSWON   ; 3/2 Sample input switch (low true)
SWON:   ACALL  DLY50        ; 2/2 Debounce 50 ms
        CLR    P1.7        ; 2/1 Turn on LED (low true)
        MOV    R2, #100     ; 2/1 Call DLY50 50 ms subroutine 100 times
SEC5:   ACALL  DLY50        ; 2/2 to get 5 second LED ON time
        DJNZ  R2, SEC5      ; 2/2 Decrement count and test and test for zero result
        SETB  P1.7        ; 2/1 Turn off LED (low true)
WSWOFF: JNB    P1.0, WSWOFF ; 3/2 Sample input switch (low true)
SWOFF:  ACALL  DLY50        ; 2/2 Debounce 50 ms
        AJMP  WSWON        ; 2/2 Repeat procedure (wait for Switch ON)

DLY50:  MOV    R3, #98      ; 2/1 Set up outer loop count
        MOV    R4, #68      ; 2/1 Set up inner loop count
LOOP:   DJNZ  R4, LOOP      ; 2/2 Decrement inner loop count and test for zero
        DJNZ  R3, LOOP      ; 2/2 Decrement outer loop count and test for zero
        RET                    ; 1/2 Return from subroutine
```

34 BYTES
27 CYCLES*

*Cycle times without wait loops

DLY50 TIMING ANALYSIS: CYCLES

```
INITIAL (2 x 1)           2
1 x (67 x 2 + 1 x 4)      138
97 x (255 x 2 + 1 x 4)   49858
Terminating (2)           2
```

TOTAL 50000 Equivalent to 50.000 ms @ 1 μ s per cycle

PIC16C5X BENCHMARK #8—SWITCH ACTIVATED FIVE SECOND LED

This benchmark samples a switch input to activate a five second output for turning on an LED. The switch is debounced with a 50 ms delay both on opening and closure. Once activated, the switch will turn on an LED output for five seconds and then turn it off, regardless of whether or not the switch is still activated. Once the switch is turned off, the procedure is repeated. Both the switch input and the LED output are low true.

```

PORTB      EQU 6          ; Register File F6
CNTR       EQU 21        ; Reg File F21
CNTR2      EQU 22        ; Reg File F22
CNTR3      EQU 23        ; Reg File F23

LED5:      MOVLW         FOH          ; 1/1 Output data to W
           MOVWF         PORTB       ; 1/1 Set upper 4 bits of port for low true output
           MOVLW         OFH          ; 1/1 Tri-state control to W
           TRIS         PORTB       ; 1/1 Tri-state lower 4 bits of port for input
WSWON:     BTFSC        PORTB, 0     ; 1/1 Sample input switch (low true)
           GOTO         WSWON        ; 1/2 Wait for Switch ON
SWON:      CALL         DLY50        ; 1/2 Debounce 50 ms
           BCF         PORTB, 7     ; 1/1 Turn on LED (low true)
           MOVLW         100         ; 1/1 Five sec count to W
           MOVWF         CNTR        ; 1/1 Call DLY50 50 ms subroutine 100 times
SEC5:      CALL         DLY50        ; 1/2 to get 5 second LED ON time
           DECFSZ       CNTR        ; 1/1 Decrement CNTR1 and test for zero result
           GOTO         SEC5         ; 1/2 Loop back until count finished
           BSF         PORTB, 7     ; 1/1 Turn off LED (low true)
WSWOFF:    BTFSS        PORTB, 0     ; 1/1 Sample input switch (low true)
           GOTO         WSWOFF       ; 1/2 Wait for Switch OFF
SWOFF:     CALL         DLY50        ; 1/2 Debounce 50 ms
           GOTO         WSWON        ; 1/2 Repeat procedure (wait for Switch ON)

DLY50:     MOVLW         130         ; 1/1 Outer loop count to W
           MOVWF         CNTR2      ; 1/1 Set up outer loop count
           MOVLW         221        ; 1/1 Inner loop count to W
           MOVWF         CNTR3      ; 1/1 Set up inner loop count
LOOP:      DECFSZ       CNTR3        ; 1/1 Decrement inner loop count and test for zero
           GOTO         LOOP        ; 1/2 Loop back until inner count finished
           DECFSZ       CNTR2      ; 1/1 Decrement outer loop count and test for zero
           GOTO         LOOP        ; 1/2 Loop back until outer count finished
           RETLW        0           ; 1/2 Return from subroutine

```

27 WORDS (Equivalent to 40 1/2 BYTES)

Cycle times without wait loops 37 CYCLES

DLY50 TIMING ANALYSIS: CYCLES

Initial (4 x 1)	4
1 x (220 x 3 + 1 x 5)	665
129 x (255 x 3 + 1 x 5)	99330
Terminating (2 - 1)	1
TOTAL	100000 Equivalent to 50.000 ms @ 0.5 μs/cycle

6.0 SUMMARY OF RESULTS (Both Positive (+) and Negative (-))

Benchmark #1 - BLOCK TRANSFER

COP8 Two indirect data pointers (+); IFBNE instruction (+)
M68HC05 Indexed addressing (+); Decrementing index to zero (+)
80C51 Two indirect data pointers (+)
PIC16C5X Only one indirect data pointer (-)

Benchmark #2 - BINARY ADDITION

COP8 Two indirect data pointers (+); IFBNE instruction (+)
M68HC05 Indexed addressing with offset (+)
80C51 Two indirect data pointers (+)
PIC16C5X Only one indirect data pointer (-)

Benchmark #3 - BCD SUBTRACTION

COP8 Two indirect data pointers (+); IFBNE instruction (+); DCOR (decimal correct) instruction (+)
M68HC05 Indexed addressing with offset (+); No BCD correction instruction (-)
80C51 Two indirect data pointers (+); DA instruction (BCD correction) only for add, not subtract (-)
PIC16C5X Only one indirect data pointer (-); No BCD correction instruction (-)

Benchmark #4 - TABLE SEARCH

COP8 LAID instruction (+); Lack of IFNE comparison instruction (-); RETSK instruction (+)
M68HC05 Indexed addressing (+); Lack of loading immediate values directly to memory (-)
80C51 16-bit DPTR (data pointer (+); MOV A @A + DPTR instruction (+); CJNE instruction combine comparison and branch (+)
PIC16C5X RETLW instruction (+)

Benchmark #5 - INPUT/OUTPUT MANIPULATION

COP8 I/O Port Configuration registers for individual bit control (+)
M68HC05 I/O Port Data Direction reg's for individual bit control (+); No SWAP to reverse nibbles (-)
80C51 Distinction between tristating for input versus output of 1 (?-); No configuration register (-)
PIC16C5X TRIS (tristate instruction) (+); No configuration register (-)

Benchmark #6 - SERIAL INPUT/OUTPUT WITH OFFSET TABLE

COP8 Microwire/Plus for serial I/O (+); LAID instruction (+)
M68HC05 SPI (Serial Peripheral Interface) for serial I/O (+)
80C51 Serial I/O with 8-bit shift register mode cannot input and output simultaneously (-); MOV A, @A + DPTR instruction (+)
PIC16C5X Lack of any dedicated serial I/O (-); RETLW instruction (+)

Benchmark #7 - TIMEKEEPING

COP8 Timer autoreload with interrupt very flexible (+)
M68HC05 Timer Output Compare interrupt structuring for real time cumbersome and confusing (-)
80C51 Lack of 16-bit timer autoreload (-)
PIC16C5X OPTION instruction to select timer prescaler (+); Lack of timer interrupt (-); No timer autoreload (-)

Benchmark #8 - SWITCH ACTIVATED 5 SEC LED

COP8 Lack of bit testing for bit clear (-)
M68HC05 Bit testing for both set and clear (+); Versatility of index register (+)
80C51 DJNZ instruction combines testing and branch (+); JB and JNB instructions combine bit testing and branch (+)
PIC16C5X Bit testing for both set and clear (+)

TABLE I. The Benchmark Results: Code Size Efficiency in Bytes

BENCHMARK BYTES	National COP8	Motorola 68HC05	Intel 80C51	Microchip PIC16C5X
Five Byte Block Move	7	9	11	21
Four Byte Binary Addition	10	13	14	28 1/2
Four Byte BCD Subtraction	25	54	39	82 1/2
Three Byte Table Search	42	40	29	51
Input/Output Manipulation	26	42	33	37 1/2
Serial I/O with Offset Table	31	31	35	45
Timekeeping	60	87	66	81
Switch Activated 5s LED	37	47	34	40 1/2
TOTAL	238	323	261	387
RATIO	1	1.36	1.10	1.63

TABLE II. The Benchmark Results: Code Execution Time Efficiency in Cycles and Microseconds

BENCHMARK CYCLE/ μ s	National COP8	Motorola 68HC05	Intel 80C51	Microchip PIC16C5X
Five Byte Block Move	47/47	76/38	32/32	62/31
Four Byte Binary Addition	48/48	85/42.5	33/33	62/31
Four Byte BCD Subtraction	97/97	567/283.5	91/91	274/137
Three Byte Table Search (Note 2)	77/77	80/40	30/30	52/26
Input/Output Manipulation (Note 3)	24/24	53/26.5	17/17	21/10.5
Serial I/O with Offset Table (Note 1)	36/36	54/27	52/52	214/107
Timekeeping (Note 5)	80/80	218/109	49/49	69/34.5
Switch Activated 5s LED (Note 4)	76/76	88/44	27/27	37/18.5
TOTAL	485/485	1221/610.5	331/331	791/395.5
RATIO	1/1	2.53/1.26	0.68/0.68	1.63/0.82

Note 1: Couplet (address, data) loop time: address and data input, updated data output.

Note 2: First search iteration fails with 1st byte mismatch, second search iteration successful.

Note 3: Case where $P1 < P2$.

Note 4: Cycle times without wait loops.

Note 5: Case where 12:59:59 is advancing to 1:00:00.

7.0 CONCLUSIONS

This report provides a detailed study of the instruction sets of popular 8-bit single chip microcontrollers. Eight benchmark programs, demonstrating data movement, arithmetic operations, I/O manipulation, and timekeeping, were coded for four current microcontrollers.

In terms of byte efficiency, the COP8 from National, uses 8% less program space than its nearest competitor, the Intel 80C51. The COP8 uses, 26% less program space than Motorola 68HC05, 39% less program space than the Microchip PIC16C5X. Code efficiency is important because it enables designers to pack more on-chip functionality into less program memory space. Selecting a microcontroller with smaller program memory size translates into lower system costs, and the added security of knowing that more code can be packed into the microcontroller.

In terms of code execution, Intel's 80C51 is the fastest, while COP8 executes these benchmark routines faster than Motorola 68HC05 and Microchip PIC16C5X.

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.



National Semiconductor Corporation
1111 West Bardin Road
Arlington, TX 76017
Tel: (800) 272-9959
Fax: (1800) 737-7018

<http://www.national.com>

National Semiconductor Europe

Fax: +49 (0) 180-530 85 86
Email: europe.support@nsc.com
Deutsch Tel: +49 (0) 180-530 85 85
English Tel: +49 (0) 180-532 78 32
Français Tel: +49 (0) 180-532 93 58
Italiano Tel: +49 (0) 180-534 16 80

National Semiconductor Hong Kong Ltd.

13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductor Japan Ltd.

Tel: 81-043-299-2308
Fax: 81-043-299-2408

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products

Audio	www.ti.com/audio
Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
OMAP Mobile Processors	www.ti.com/omap
Wireless Connectivity	www.ti.com/wirelessconnectivity

Applications

Communications and Telecom	www.ti.com/communications
Computers and Peripherals	www.ti.com/computers
Consumer Electronics	www.ti.com/consumer-apps
Energy and Lighting	www.ti.com/energy
Industrial	www.ti.com/industrial
Medical	www.ti.com/medical
Security	www.ti.com/security
Space, Avionics and Defense	www.ti.com/space-avionics-defense
Transportation and Automotive	www.ti.com/automotive
Video and Imaging	www.ti.com/video

TI E2E Community Home Page

e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2011, Texas Instruments Incorporated