Application Note

AN2627/D
12/2003

Cycle-by-Cycle Instruction Set Details for the M68HC08 Family of MCUs

By Jim Sibigtroth
8/16 Bit Systems/Applications Engineering
Austin, Texas

Introduction

This application note provides detailed information, not previously published, about the cycle-by-cycle behavior of CPU M68HC08 instructions. Although most applications do not require this level of detail, it can be very useful in unusual cases where it is important to carefully control the timing of control sequences or the relative timing of I/O events. This level of detail also helps users understand exactly how read-modify-write instructions work.

NOTE: *With the exception of mask set errata documents, if any other Motorola document contains information that conflicts with the information in the device data sheet, the device data sheet should be considered to have the most current and correct data.*

Cycle Codes

This document uses the shorthand notation that is used to document cycle-by-cycle details in the HCS08 and HCS12 instruction sets. This shorthand uses one character to mnemonically represent each bus cycle. For example, a lowercase `p` is used to represent a program fetch cycle. In the HC08 CPU, all bus cycles refer to 8-bit data so all of the mnemonic cycle codes use lowercase letters. In the HCS12, some bus cycles used uppercase letters to indicate 16-bit memory accesses. The cycle-by-cycle codes used for the HC08 CPU are explained in the following paragraphs.

All trademarks belong to their respective companies.
This product incorporates SuperFlash® technology licensed from SST.

Program Fetch Cycle – `p` – Used to fetch the next byte of object code from program memory. When any HC08 instruction starts, the first byte of object code for that instruction is already in the CPU’s instruction buffer. Each instruction includes enough `p` cycles to replace the number of bytes of object code for that instruction.

Figure 1 shows the timing diagram for internal clock and bus signals during a program fetch cycle.

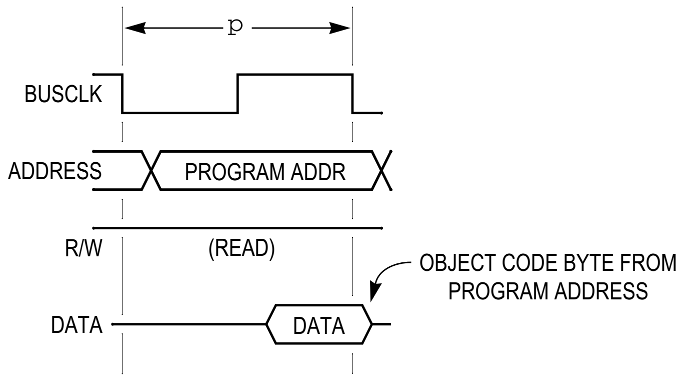


Figure 1. Timing Waveforms for a Program Fetch (`p`) Cycle

For example, when a 2-byte instruction such as ADD (direct addressing mode) is executed, the opcode ($BB) is already in the CPU’s instruction buffer. The instruction then performs one `p` cycle to fetch the low half of the direct address of the operand, uses this address to read the operand from memory, and then performs a second `p` cycle to fetch the opcode of the next instruction.

Byte Read Cycle – `r` – Used to read one byte of operand data from memory.

Figure 2 shows the timing diagram for internal clock and bus signals during a data read cycle. If the operand address corresponds to an input port pin, there will usually be a simple synchronizer circuit associated with the signal so that the value on the data bus does not change state for a setup-and-hold time near the falling edge of the bus clock. Because the internal BUSCLK signal is not visible outside the MCU, you should think of the read as taking place sometime during the last half of the BUSCLK cycle.

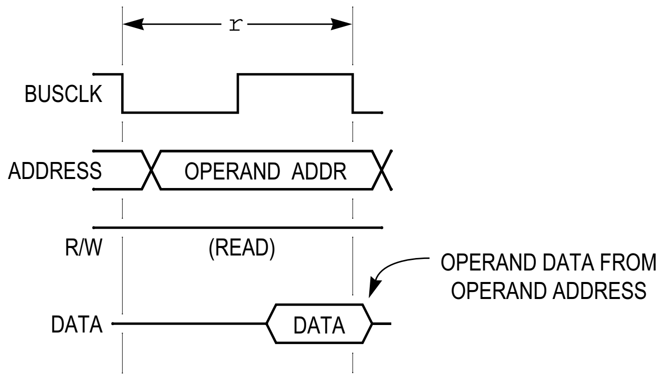


Figure 2. Timing Waveforms for a Data Read (`r`) Cycle

Byte Write Cycle – w – Used to write one byte of operand data to memory.

Figure 3 shows the timing diagram for internal clock and bus signals during a data write cycle. The write takes place during the last half of the bus cycle. In the case where the operand address corresponds to an output port pin, the pin changes state one propagation delay after the middle of the bus cycle.

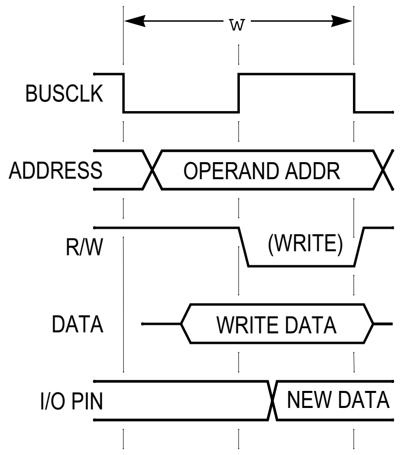


Figure 3. Timing Waveforms for a Write (w) Cycle

Stack Write (Push) Cycle – `s` – Used to write (push) one byte of data to the next available location on the system stack. The stack in the M68HC08 builds from higher addresses to lower addresses, and the stack pointer (SP) always points to the next available location on the stack.

Figure 4 shows the timing diagram for internal clock and bus signals during a stack write cycle.

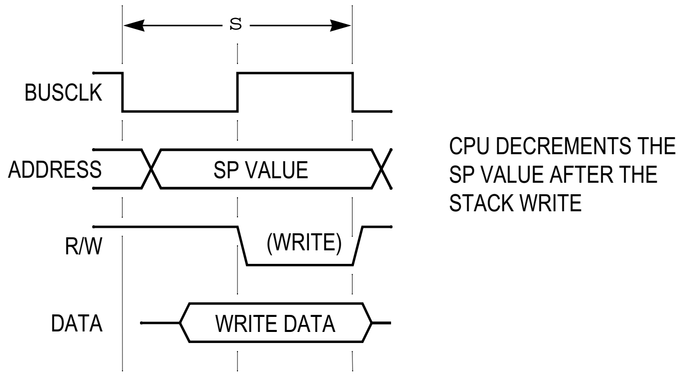


Figure 4. Timing Waveforms for a Stack Write (`s`) Cycle

For example, a PSHA instruction is a 2-cycle instruction with the shorthand code `ps` where the `p` cycle fetches a byte of object code to make up for the one byte of object code needed for the PSHA instruction. The `s` cycle is used to store the contents of the accumulator at the location pointed-to by SP. Then SP is decremented to point at the next available location on the stack.

Stack Read (Pop) Cycle – `u` – Used to unstack or read (pop) one byte of data from the system stack. For example a PULA instruction is a 2-cycle instruction with the shorthand code `pu`. The `p` cycle fetches a byte of object code to make up for the one byte of object code needed for the PULA instruction. The `u` cycle is used to get one byte of data from the stack by incrementing SP by one and then reading the value pointed-to by SP into the accumulator.

Figure 5 shows the timing diagram for internal clock and bus signals during a stack read cycle.

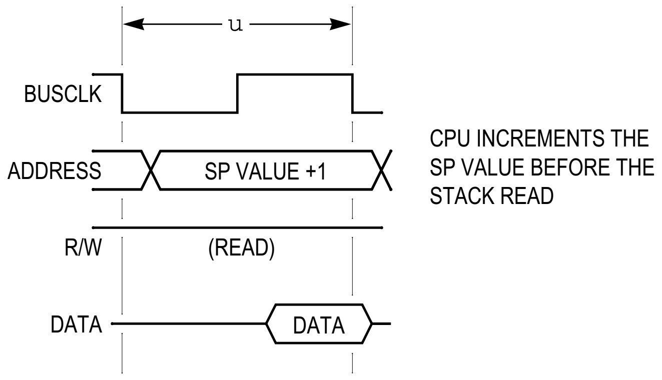


Figure 5. Timing Waveforms for a Stack Read (`u`) Cycle

Vector Fetch Cycle – `v` – Used to fetch one-half of a 16-bit interrupt or reset vector from memory. `v` cycles are always found in pairs to fetch the high and low bytes of a 16-bit vector, respectively.

Figure 6 shows the timing diagram for internal clock and bus signals during a pair of vector fetch cycles. During these two `v` cycles, the data that is read from the vector addresses is loaded directly into the program counter high and low halves (PCH and PCL), respectively. The next cycle after a pair of vector fetch cycles is always a program fetch cycle (not shown in this figure) using the address that was loaded into PCH and PCL by the two `v` cycles.

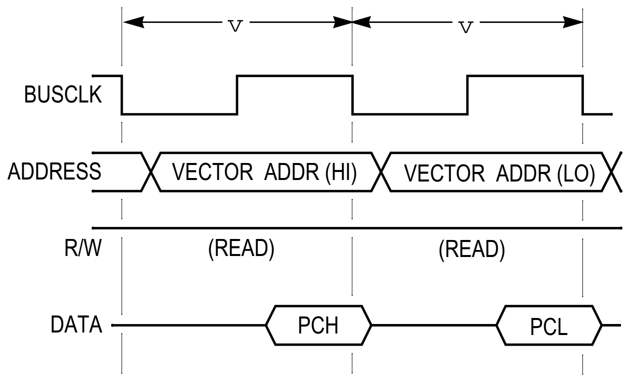


Figure 6. Timing Waveforms for Two Vector Fetch (`v`) Cycles

Dummy (Read) Cycles – `d` – Used to perform internal operations where no new information is read or written using system address and data buses. In these cases, a dummy read cycle is performed using the same address as the previous bus cycle. The data from a `d` cycle is ignored.

Figure 7 shows the timing diagram for internal clock and bus signals during a dummy cycle.

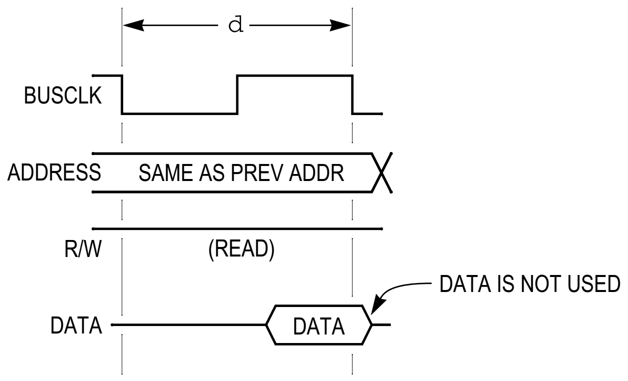


Figure 7. Timing Waveforms for a Dummy (`d`) Cycle

Interpreting Cycle-by-Cycle Code Sequences

This section will discuss an example timing diagram of internal bus and control signals during the execution of a BCLR instruction. After the meaning of the code letters is understood, you will not need to see cycle-by-cycle details as timing diagrams. The cycle-by-cycle code sequence for a BCLR 0,*opr8a* instruction is `prwp`. Because there are four code letters, the instruction takes four bus cycles. **Figure 8** shows this 4-cycle sequence for a BCLR instruction that will generate a falling edge on the port B, bit 0, pin.

```
"    "  "        "          "     "            "

8310 11 01       loop:      BCLR  Bit_0,PortB   ;[4] drive PTB0 low
8312 20 FC                  BRA   loop          ;[3] repeat
"    "  "        "          "     "            "
```

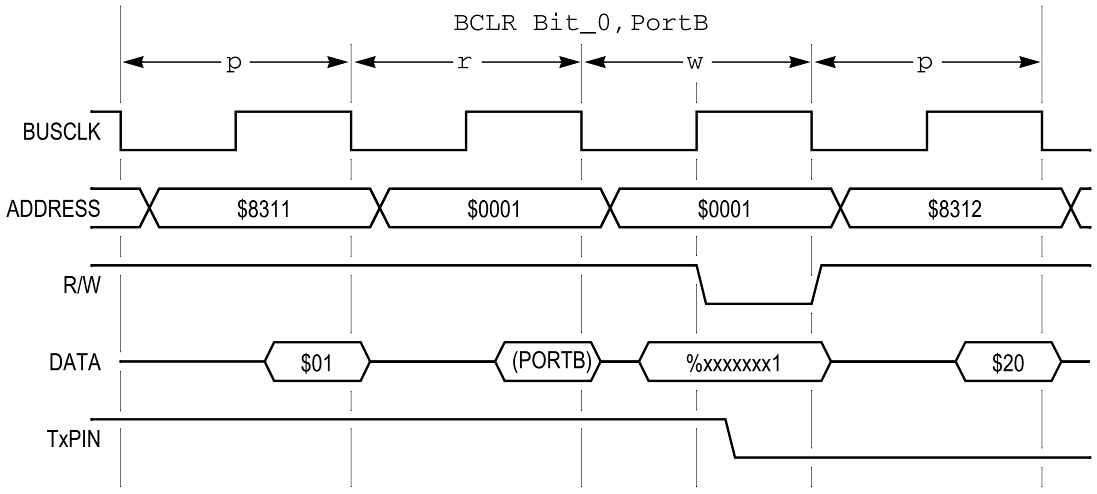


Figure 8. Timing Details for a BCLR Instruction

The top portion of this figure shows two program listing lines, including the BCLR instruction. Because the first byte of object code for the next instruction ($20) will be fetched during the BCLR instruction, the BRA instruction line is also shown. The 4-digit hexadecimal number at the beginning of each of these lines is the program address for the first byte of object code for the instruction. In the case of the BCLR 0,*oper8a* instruction, the opcode is $11 and the $01 is the low half of the address for the operand.

Remember that when an instruction starts to execute, the first byte of object code for the instruction is already in the CPU's instruction buffer because it was fetched during the previous instruction. So, the first `p` cycle fetches the second byte of object code ($01) for the BCLR instruction from address $8311. Between the first and second cycles of this instruction, the CPU constructs the address of the operand by using this $01 as the low half of an address in the range $0000–$00FF. The second cycle of this BCLR instruction (`r`) reads the current contents of the port B register from this constructed address ($0001). Between the second and third cycles, the CPU forces bit 0 of the value that was read from port B to a 1. During the third cycle (`w`), the CPU writes this modified data value back to port B at $0001. During the fourth cycle (`p`), the CPU reads the next byte of object code ($20) from address $8312. This is the BRA opcode and it is loaded into the instruction buffer so that the CPU will be ready to execute the first cycle of the next instruction immediately following the fourth cycle of the BCLR instruction.

Hardware Reset

Resets cause registers and systems inside the MCU to assume default values. The reset sequence includes several sequential events and tests before the CPU begins executing instructions. After the hardware reset sequence is completed, the CPU performs two `v` cycles to fetch the high byte of the reset vector from $FFFE and the low byte from $FFFF, respectively. It then performs one `p` cycle to pre-load the CPU's instruction buffer with the opcode of the first instruction. Execution then continues using the cycle-by-cycle sequences for each consecutive instruction. There are no gaps between the cycle-by-cycle sequences for reset, instructions, or interrupts. So if the first instruction of an application program was an LDA (immediate) instruction, the CPU would execute the sequence `vvp` for the reset, immediately followed by `pp` for the LDA (immediate) instruction.

Interrupts

Hardware interrupts are an exception to the sequential flow of program instructions. When an interrupt occurs, the CPU completes the instruction that is currently being executed and then responds to the interrupt. The interrupt sequence uses the same 9-cycle sequence as an SWI instruction (`psssssvvp`). The first `p` cycle is a fetch of the program byte that would have been fetched if the interrupt had not occurred. This data will not be used by the CPU, but the fetch was already scheduled before the interrupt occurred. The next five `s` cycles of the interrupt sequence store (push) the return address low, return address high, X, A, and CCR onto the stack so the CPU can resume the interrupted program at the point where it was interrupted (after completing the interrupt service routine (ISR)). The next two `v` cycles fetch the high and low halves of the interrupt vector for the highest priority source that caused the interrupt. Finally, a `p` cycle is executed to pre-fill the instruction buffer with the opcode of the first instruction of the ISR.

Usually, the ISR would end with a 7-cycle RTI instruction (`puuuuup`), which recovers the previously saved CPU state and resumes execution of the original program as if the interrupt had not occurred.

Conditional Branches

Conditional branch instructions execute one of two different sequences depending upon whether the branch condition was true. In the case of the BRN instruction, the branch condition is never true. Therefore, the sequence is `pdp` where the first `p` cycle fetches the second byte of object code for the branch instruction (the offset byte). The next `d` cycle is a dummy read from the same address. During the `d` cycle, the CPU adds the offset to the program counter. This gets the pointer to the instruction at the branch destination. Because the BRN instruction never branches, this calculated address is not used. Instead, the branch is not taken, and the third cycle (`p`) fetches the opcode of the next instruction after the BRN instruction.

When the branch condition for a branch instruction is true, the sequence is the same except the last `p` cycle fetches the opcode of the instruction at the branch destination rather than the opcode of the instruction immediately after the conditional branch instruction.

BRSET and BRCLR instructions have a 5-cycle sequence (`prpdp`). The first `p` cycle fetches the low half of the operand's direct address. The `r` cycle fetches the whole 8-bit operand from the direct memory location that contains the bit that will be tested. The second `p` cycle is used to fetch the branch offset while the specified bit is tested in the operand that was just fetched. The next `d` cycle is a dummy read from the same address while the CPU is adding the offset to the program counter to get the pointer to the instruction at the branch destination. The last `p` cycle fetches the opcode of the next instruction from either the destination address or the next address after the offset, depending on whether the branch condition was true or false, respectively.

Similarly, the last `p` cycle of a CBEQ or DBNZ instruction fetches the opcode of the next instruction from either the destination address or the next address after the offset, depending on whether the branch condition was true or false, respectively.

Cycle-Timed Code

Although you don't need to know the cycle-by-cycle details of instructions for most application programs, there are times when this information is critical. For example, suppose you want to write a program that transmits or receives serial data using software and general-purpose I/O pins to create an RS232 serial communications interface (SCI). In such a case, it may be possible to write a program that can send or receive at a slightly faster baud rate if you know the timing of reads and writes at the cycle level instead of the instruction level. For example, in direct and extended addressing mode variations of STA instructions, the write takes place in the next-to-last cycle of the instruction, but in the indexed addressing mode variations, the write occurs in the last cycle of the instruction. The position of the read cycle in LDA instructions is similar.

To demonstrate how you would use the detailed cycle-by-cycle information, we will study an example routine. The program segment in **Listing 1** is the working portion of a software SCI transmit routine. A general-purpose I/O pin at bit number TxBit, in the I/O port corresponding to the data direction register TxDDR, will be used as our TxD pin. The port bit corresponding to this pin was previously written to 0 and the associated pullup was enabled (or an external pullup resistor is connected). When the DDR bit is 0, the pin behaves as a high-impedance input that is pulled high. When the DDR bit is set to 1, the pin behaves as an output pin that is driven low. The 8-bit data value was previously pushed onto the stack, and because another value was also previously pushed, the data will be at `3,SP` after the PSHX at `putbyte3:`

```
"            "       "                 "
putbyte3: pshx                    ;store delay counter
          lda    #10              ;start, 8 data, stop = 10 loops
          sec                     ;becomes stop bit after 9 RORs
          bra    outLow           ;[3] Tx a low for start bit

PutLoop:  ror    3,SP             ;[5] LSB to C-bit, Tx that level
          bcc    outLow           ;[3] if C=0 Tx low, else Tx a hi
outHi:    bclr   TxBit,TxDDR      ;[4] PTA0 input pulls up to high
          bra    outDelay         ;[3] go to time 1 bit delay
outLow:   bset   TxBit,TxDDR      ;[4] PTA0 output makes pin drive low
          bra    outDelay         ;[3] time 1 bit delay (match time)
outDelay: dbnzx  *                ;[3] loop 3~ * (value in X)
          pulx                    ;[2]
          pshx                    ;[2]
          dbnza  PutLoop          ;[3] repeat for start, 8 data, stop
 "            "       "                 "
```

Listing 1. Partial Code Listing for SCI Transmit Routine

We will map out the cycle-by-cycle operation of this routine starting from the `bra outLow` instruction above `PutLoop:` until the program has generated the start bit and the first data bit of the transmit value. We will assume that the first data bit is a 1 so we can easily see the bit time boundaries. We also assume the I/O pin associated with TxBit was acting as an input and was pulled up before starting this routine.

Listing 2 shows the instructions in the order that they would be executed (starting from the `bra outLow` instruction just above the `PutLoop:` label in **Listing 1**). This listing shows instructions in execution order so some instructions and sequences are repeated (such as the DBNZX instruction that is used to form a bit-time delay). For this example, we will assume X is 2 at the start of the routine to simplify our drawings. In the actual SCI transmit routine, X would be set to make the delay for a bit time equal to the appropriate length for the desired baud rate and bus speed.

Listing 2 also shows the cycle-by-cycle details for each instruction. For example, the `outLow:   bset   TxBit,TxDDR` instruction is made up of the 4-cycle sequence `prwp`. The first `p` cycle fetches the direct address for the BSET instruction. The second cycle is a byte read of the operand (the current contents of the TxDDR register). Between the second and third cycles of the BSET instruction, the CPU sets the TxBit in this value. The third cycle is a write of the modified value back to the TxDDR register. And finally, the last cycle is a program fetch to refill the instruction buffer in preparation for the next instruction.

From **Listing 2**, we can see that the transmit data pin will go low during the third cycle of the `outLow:   bset   TxBit,TxDDR` instruction, and the start bit will end exactly 28 cycles later in the third cycle of the `outHi:   bclr TxBit,TxDDR` instruction. We can also see that the transmit data pin will go low again at the end of the LSB data bit exactly 28 cycles later during the `outLow:   bset   TxBit,TxDDR` instruction at the bottom of **Listing 2**.

```
_____________________________________________
_ p _          bra    outLow
_ d _
_ p __________________________________________
_ p outLow:    bset   TxBit,TxDDR
_ r _
_ w _  -------------------------  <-beginning of start bit
_ p __________________________________________
_ p _          bra    outDelay
_ d _
_ p __________________________________________
_ p outDelay: dbnzx  *
_ d _
_ p __________________________________________
_ p outDelay: dbnzx  *
_ d _
_ p __________________________________________
_ p _          pulx
_ u __________________________________________
_ p _          pshx
_ s __________________________________________
_ p _          dbnza  PutLoop
_ d _
_ p __________________________________________
_ p PutLoop:   ror    3,SP
_ p _
_ p _
_ r _
_ w __________________________________________
_ p _          bcc    outLow
_ d _
_ p __________________________________________
_ p outHi:     bclr   TxBit,TxDDR
_ r _
_ w _  -------------------------  <-end of start bit
_ p __________________________________________
_ p _          bra    outDelay
_ d _
_ p __________________________________________
_ p outDelay: dbnzx  *
_ d _
_ p __________________________________________
_ p outDelay: dbnzx  *
_ d _
_ p __________________________________________
_ p _          pulx
_ u __________________________________________
_ p _          pshx
_ s __________________________________________
_ p _          dbnza  PutLoop
_ d _
_ p __________________________________________
_ p PutLoop:   ror    3,SP
_ p _
_ p _
_ r _
_ w __________________________________________
_ p _          bcc    outLow
_ d _
_ p __________________________________________
_ p outLow:    bset   TxBit,TxDDR
_ r _
_ w _  -------------------------  <- end of LSB (bit-0)
_ p __________________________________________
```

Listing 2. Instruction Order Listing (with Cycle Details)

Conclusion

This application note explains the cycle-by-cycle details for each addressing mode of each instruction that is included in the instruction set summary for the M68HC08 CPU. Cycle-by-cycle details for each addressing mode of each instruction are provided in a new column near the right side of the instruction set summary table. A shorthand (code) was used to provide this detailed information in a compact form where each bus cycle is represented by a single mnemonic character. The code letters were explained. Several special operations including reset, interrupts, and branches were also explained using the same bus cycle codes. Finally, a code example was used to explain how this cycle-by-cycle information can be used while writing software routines that can control I/O pins with one-cycle precision.

This application note can also help users understand how the M68HC08 CPU executes instructions. For example, the cycle-by-cycle detail for a BSET or BCLR instruction shows that these instructions are read-modify-write instructions. This means these instructions read the entire 8-bit location, internally modify the selected bit within that value, and then re-write the modified value to the memory location. Without this level of detail, it could appear that the CPU somehow wrote to a single bit without reading or writing other bits in the memory location.

Appendix A — Instruction Set Summary

Table 1 provides a summary of the M68HC08 instruction set in all possible addressing modes. The table shows operand construction, execution time in internal bus clock cycles, and cycle-by-cycle details for each addressing mode variation of each instruction.

Table 1. Instruction Set Summary (Sheet 1 of 9)

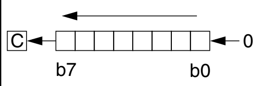
Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Affect on CCR	
						V 1 1 H	I N Z C
ADC #*opr8i*	Add with Carry A ← (A) + (M) + (C)	IMM	A9 ii	2	pp	↕ 1 1 ↕	– ↕ ↕ ↕
ADC *opr8a*		DIR	B9 dd	3	prp		
ADC *opr16a*		EXT	C9 hh ll	4	pprp		
ADC *oprx16*,X		IX2	D9 ee ff	4	pppr		
ADC *oprx8*,X		IX1	E9 ff	3	ppr		
ADC ,X		IX	F9	2	pr		
ADC *oprx16*,SP		SP2	9E D9 ee ff	5	ppppr		
ADC *oprx8*,SP		SP1	9E E9 ff	4	pppr		
ADD #*opr8i*	Add without Carry A ← (A) + (M)	IMM	AB ii	2	pp	↕ 1 1 ↕	– ↕ ↕ ↕
ADD *opr8a*		DIR	BB dd	3	prp		
ADD *opr16a*		EXT	CB hh ll	4	pprp		
ADD *oprx16*,X		IX2	DB ee ff	4	pppr		
ADD *oprx8*,X		IX1	EB ff	3	ppr		
ADD ,X		IX	FB	2	pr		
ADD *oprx16*,SP		SP2	9E DB ee ff	5	ppppr		
ADD *oprx8*,SP		SP1	9E EB ff	4	pppr		
AIS #*opr8i*	Add Immediate Value (Signed) to Stack Pointer SP ← (SP) + (M)	IMM	A7 ii	2	pp	– 1 1 –	– – – –
AIX #*opr8i*	Add Immediate Value (Signed) to Index Register (H:X) H:X ← (H:X) + (M)	IMM	AF ii	2	pp	– 1 1 –	– – – –
AND #*opr8i*	Logical AND A ← (A) & (M)	IMM	A4 ii	2	pp	0 1 1 –	– ↕ ↕ –
AND *opr8a*		DIR	B4 dd	3	prp		
AND *opr16a*		EXT	C4 hh ll	4	pprp		
AND *oprx16*,X		IX2	D4 ee ff	4	pppr		
AND *oprx8*,X		IX1	E4 ff	3	ppr		
AND ,X		IX	F4	2	pr		
AND *oprx16*,SP		SP2	9E D4 ee ff	5	ppppr		
AND *oprx8*,SP		SP1	9E E4 ff	4	pppr		
ASL *opr8a*	Arithmetic Shift Left  (Same as LSL)	DIR	38 dd	4	prwp	↕ 1 1 –	– ↕ ↕ ↕
ASLA		INH	48	1	p		
ASLX		INH	58	1	p		
ASL *oprx8*,X		IX1	68 ff	4	pprw		
ASL ,X		IX	78	3	prw		
ASL *oprx8*,SP		SP1	9E 68 ff	5	ppprw		

Table 1. Instruction Set Summary (Sheet 2 of 9)

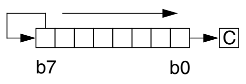
Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Affect on CCR	
						V 1 1 H	I N Z C
ASR *opr8a* ASRA ASRX ASR *oprx8*,X ASR ,X ASR *oprx8*,SP	Arithmetic Shift Right 	DIR INH INH IX1 IX SP1	37 dd 47 57 67 ff 77 9E 67 ff	4 1 1 4 3 5	prwp p p pprw prw ppprw	↕ 1 1 –	– ↕ ↕ ↕
BCC *rel*	Branch if Carry Bit Clear (if C = 0)	REL	24 rr	3	pdp	– 1 1 –	– – – –
BCLR *n*,*opr8a*	Clear Bit n in Memory (Mn ← 0)	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	11 dd 13 dd 15 dd 17 dd 19 dd 1B dd 1D dd 1F dd	4 4 4 4 4 4 4 4	prwp prwp prwp prwp prwp prwp prwp prwp	– 1 1 –	– – – –
BCS *rel*	Branch if Carry Bit Set (if C = 1) (Same as BLO)	REL	25 rr	3	pdp	– 1 1 –	– – – –
BEQ *rel*	Branch if Equal (if Z = 1)	REL	27 rr	3	pdp	– 1 1 –	– – – –
BGE *rel*	Branch if Greater Than or Equal To (if N ⊕ V = 0) (Signed)	REL	90 rr	3	pdp	– 1 1 –	– – – –
BGT *rel*	Branch if Greater Than (if Z I (N ⊕ V) = 0) (Signed)	REL	92 rr	3	pdp	– 1 1 –	– – – –
BHCC *rel*	Branch if Half Carry Bit Clear (if H = 0)	REL	28 rr	3	pdp	– 1 1 –	– – – –
BHCS *rel*	Branch if Half Carry Bit Set (if H = 1)	REL	29 rr	3	pdp	– 1 1 –	– – – –
BHI *rel*	Branch if Higher (if C I Z = 0)	REL	22 rr	3	pdp	– 1 1 –	– – – –
BHS *rel*	Branch if Higher or Same (if C = 0) (Same as BCC)	REL	24 rr	3	pdp	– 1 1 –	– – – –
BIH *rel*	Branch if IRQ Pin High (if IRQ pin = 1)	REL	2F rr	3	pdp	– 1 1 –	– – – –
BIL *rel*	Branch if IRQ Pin Low (if IRQ pin = 0)	REL	2E rr	3	pdp	– 1 1 –	– – – –
BIT #*opr8i* BIT *opr8a* BIT *opr16a* BIT *oprx16*,X BIT *oprx8*,X BIT ,X BIT *oprx16*,SP BIT *oprx8*,SP	Bit Test (A) & (M) (CCR Updated but Operands Not Changed)	IMM DIR EXT IX2 IX1 IX SP2 SP1	A5 ii B5 dd C5 hh ll D5 ee ff E5 ff F5 9E D5 ee ff 9E E5 ff	2 3 4 4 3 2 5 4	pp prp pprp pppr ppr pr ppppr pppr	0 1 1 –	– ↕ ↕ –
BLE *rel*	Branch if Less Than or Equal To (if Z I (N ⊕ V) = 1) (Signed)	REL	93 rr	3	pdp	– 1 1 –	– – – –
BLO *rel*	Branch if Lower (if C = 1) (Same as BCS)	REL	25 rr	3	pdp	– 1 1 –	– – – –
BLS *rel*	Branch if Lower or Same (if C I Z = 1)	REL	23 rr	3	pdp	– 1 1 –	– – – –
BLT *rel*	Branch if Less Than (if N ⊕ V = 1) (Signed)	REL	91 rr	3	pdp	– 1 1 –	– – – –

Table 1. Instruction Set Summary (Sheet 3 of 9)

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Affect on CCR	
						V 1 1 H	I N Z C
BMC *rel*	Branch if Interrupt Mask Clear (if I = 0)	REL	`2C rr`	3	`pdp`	– 1 1 –	– – – –
BMI *rel*	Branch if Minus (if N = 1)	REL	`2B rr`	3	`pdp`	– 1 1 –	– – – –
BMS *rel*	Branch if Interrupt Mask Set (if I = 1)	REL	`2D rr`	3	`pdp`	– 1 1 –	– – – –
BNE *rel*	Branch if Not Equal (if Z = 0)	REL	`26 rr`	3	`pdp`	– 1 1 –	– – – –
BPL *rel*	Branch if Plus (if N = 0)	REL	`2A rr`	3	`pdp`	– 1 1 –	– – – –
BRA *rel*	Branch Always (if I = 1)	REL	`20 rr`	3	`pdp`	– 1 1 –	– – – –

BRCLR *n,opr8a,rel*	Branch if Bit *n* in Memory Clear (if (Mn) = 0)	DIR (b0)
		DIR (b1)
		DIR (b2)
		DIR (b3)
		DIR (b4)
		DIR (b5)
		DIR (b6)
		DIR (b7)
			`01 dd rr`
			`03 dd rr`
			`05 dd rr`
			`07 dd rr`
			`09 dd rr`
			`0B dd rr`
			`0D dd rr`
			`0F dd rr`
				5
				5
				5
				5
				5
				5
				5
				5
					`prpdp`
					`prpdp`
					`prpdp`
					`prpdp`
					`prpdp`
					`prpdp`
					`prpdp`
					`prpdp`
						– 1 1 –	– – – ↕

BRN *rel*	Branch Never (if I = 0)	REL	`21 rr`	3	`pdp`	– 1 1 –	– – – –

BRSET *n,opr8a,rel*	Branch if Bit *n* in Memory Set (if (Mn) = 1)	DIR (b0)
		DIR (b1)
		DIR (b2)
		DIR (b3)
		DIR (b4)
		DIR (b5)
		DIR (b6)
		DIR (b7)
			`00 dd rr`
			`02 dd rr`
			`04 dd rr`
			`06 dd rr`
			`08 dd rr`
			`0A dd rr`
			`0C dd rr`
			`0E dd rr`
				5
				5
				5
				5
				5
				5
				5
				5
					`prpdp`
					`prpdp`
					`prpdp`
					`prpdp`
					`prpdp`
					`prpdp`
					`prpdp`
					`prpdp`
						– 1 1 –	– – – ↕

BSET *n,opr8a*	Set Bit *n* in Memory (Mn ← 1)	DIR (b0)
		DIR (b1)
		DIR (b2)
		DIR (b3)
		DIR (b4)
		DIR (b5)
		DIR (b6)
		DIR (b7)
			`10 dd`
			`12 dd`
			`14 dd`
			`16 dd`
			`18 dd`
			`1A dd`
			`1C dd`
			`1E dd`
				4
				4
				4
				4
				4
				4
				4
				4
					`prwp`
					`prwp`
					`prwp`
					`prwp`
					`prwp`
					`prwp`
					`prwp`
					`prwp`
						– 1 1 –	– – – –

BSR *rel*	Branch to Subroutine PC ← (PC) + \$0002 push (PCL); SP ← (SP) – \$0001 push (PCH); SP ← (SP) – \$0001 PC ← (PC) + *rel*	REL	`AD rr`	4	`pssp`	– 1 1 –	– – – –

CBEQ *opr8a,rel*
CBEQA *#opr8i,rel*
CBEQX *#opr8i,rel*
CBEQ *oprx8*,X+,*rel*
CBEQ ,X+,*rel*
CBEQ *oprx8*,SP,*rel*
	Compare and... Branch if (A) = (M)
	Branch if (A) = (M)
	Branch if (X) = (M)
	Branch if (A) = (M)
	Branch if (A) = (M)
	Branch if (A) = (M)
		DIR
		IMM
		IMM
		IX1+
		IX+
		SP1
			`31 dd rr`
			`41 ii rr`
			`51 ii rr`
			`61 ff rr`
			`71 rr`
			`9E 61 ff rr`
				5
				4
				4
				5
				4
				6
					`pprdp`
					`ppdp`
					`ppdp`
					`pprdp`
					`prdp`
					`ppprdp`
						– 1 1 –	– – – –

CLC	Clear Carry Bit (C ← 0)	INH	`98`	1	`p`	– 1 1 –	– – – 0
CLI	Clear Interrupt Mask Bit (I ← 0)	INH	`9A`	2	`pd`	– 1 1 –	0 – – –

Table 1. Instruction Set Summary (Sheet 4 of 9)

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Affect on CCR	
						V 1 1 H	I N Z C
CLR *opr8a*	Clear M ← $00	DIR	3F dd	3	pwp	0 1 1 –	– 0 1 –
CLRA	A ← $00	INH	4F	1	p		
CLRX	X ← $00	INH	5F	1	p		
CLRH	H ← $00	INH	8C	1	p		
CLR *oprx8*,X	M ← $00	IX1	6F ff	3	ppw		
CLR ,X	M ← $00	IX	7F	2	pw		
CLR *oprx8*,SP	M ← $00	SP1	9E 6F ff	4	pppw		
CMP #*opr8i*		IMM	A1 ii	2	pp	↕ 1 1 –	– ↕ ↕ ↕
CMP *opr8a*		DIR	B1 dd	3	prp		
CMP *opr16a*	Compare Accumulator with Memory	EXT	C1 hh ll	4	pprp		
CMP *oprx16*,X	A – M	IX2	D1 ee ff	4	pppr		
CMP *oprx8*,X	(CCR Updated But Operands Not Changed)	IX1	E1 ff	3	ppr		
CMP ,X		IX	F1	2	pr		
CMP *oprx16*,SP		SP2	9E D1 ee ff	5	ppppr		
CMP *oprx8*,SP		SP1	9E E1 ff	4	pppr		
COM *opr8a*	Complement M ← (M̄)= $FF – (M)	DIR	33 dd	4	prwp	0 1 1 –	– ↕ ↕ 1
COMA	(One’s Complement) A ← (Ā) = $FF – (A)	INH	43	1	p		
COMX	X ← (X̄) = $FF – (X)	INH	53	1	p		
COM *oprx8*,X	M ← (M̄) = $FF – (M)	IX1	63 ff	4	pprw		
COM ,X	M ← (M̄) = $FF – (M)	IX	73	3	prw		
COM *oprx8*,SP	M ← (M̄) = $FF – (M)	SP1	9E 63 ff	5	ppprw		
CPHX #*opr*	Compare Index Register (H:X) with Memory (H:X) – (M:M + $0001)	IMM	65 ii jj	3	ppp	↕ 1 1 –	– ↕ ↕ ↕
CPHX *opr*	(CCR Updated But Operands Not Changed)	DIR	75 dd	4	prrp		
CPX #*opr8i*		IMM	A3 ii	2	pp	↕ 1 1 –	– ↕ ↕ ↕
CPX *opr8a*		DIR	B3 dd	3	prp		
CPX *opr16a*	Compare X (Index Register Low) with	EXT	C3 hh ll	4	pprp		
CPX *oprx16*,X	Memory	IX2	D3 ee ff	4	pppr		
CPX *oprx8*,X	X – M	IX1	E3 ff	3	ppr		
CPX ,X	(CCR Updated But Operands Not Changed)	IX	F3	2	pr		
CPX *oprx16*,SP		SP2	9E D3 ee ff	5	ppppr		
CPX *oprx8*,SP		SP1	9E E3 ff	4	pppr		
DAA	Decimal Adjust Accumulator After ADD or ADC of BCD Values	INH	72	2	pp	U 1 1 –	– ↕ ↕ ↕
DBNZ *opr8a*,*rel*		DIR	3B dd rr	5	pprwp	– 1 1 –	– – – –
DBNZA *rel*		INH	4B rr	3	pdp		
DBNZX *rel*	Decrement A, X, or M and Branch if Not Zero	INH	5B rr	3	pdp		
DBNZ *oprx8*,X,*rel*	(if (result) ≠ 0)	IX1	6B ff rr	5	pprwp		
DBNZ ,X,*rel*	DBNZX Affects X Not H	IX	7B rr	4	prwp		
DBNZ *oprx8*,SP,*rel*		SP1	9E 6B ff rr	6	ppprwp		
DEC *opr8a*	Decrement M ← (M) – $01	DIR	3A dd	4	prwp	↕ 1 1 –	– ↕ ↕ –
DECA	A ← (A) – $01	INH	4A	1	p		
DECX	X ← (X) – $01	INH	5A	1	p		
DEC *oprx8*,X	M ← (M) – $01	IX1	6A ff	4	pprw		
DEC ,X	M ← (M) – $01	IX	7A	3	prw		
DEC *oprx8*,SP	M ← (M) – $01	SP1	9E 6A ff	5	ppprw		
DIV	Divide A ← (H:A)÷(X); H ← Remainder	INH	52	7	pdpdddd	– 1 1 –	– – ↕ ↕

Table 1. Instruction Set Summary (Sheet 5 of 9)

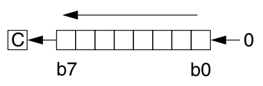
Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Affect on CCR	
						V 1 1 H	I N Z C
EOR *#opr8i*	Exclusive OR Memory with Accumulator	IMM	A8 ii	2	pp	0 1 1 –	– ↕ ↕ –
EOR *opr8a*	A ← (A ⊕ M)	DIR	B8 dd	3	prp		
EOR *opr16a*		EXT	C8 hh ll	4	pprp		
EOR *oprx16*,X		IX2	D8 ee ff	4	pppr		
EOR *oprx8*,X		IX1	E8 ff	3	ppr		
EOR ,X		IX	F8	2	pr		
EOR *oprx16*,SP		SP2	9E D8 ee ff	5	ppppr		
EOR *oprx8*,SP		SP1	9E E8 ff	4	pppr		
INC *opr8a*	Increment M ← (M) + $01	DIR	3C dd	4	prwp	↕ 1 1 –	– ↕ ↕ –
INCA	A ← (A) + $01	INH	4C	1	p		
INCX	X ← (X) + $01	INH	5C	1	p		
INC *oprx8*,X	M ← (M) + $01	IX1	6C ff	4	pprw		
INC ,X	M ← (M) + $01	IX	7C	3	prw		
INC *oprx8*,SP	M ← (M) + $01	SP1	9E 6C ff	5	ppprw		
JMP *opr8a*		DIR	BC dd	2	pp	– 1 1 –	– – – –
JMP *opr16a*	Jump	EXT	CC hh ll	3	ppp		
JMP *oprx16*,X	PC ← Jump Address	IX2	DC ee ff	4	ppdp		
JMP *oprx8*,X		IX1	EC ff	3	pdp		
JMP ,X		IX	FC	2	pp		
JSR *opr8a*	Jump to Subroutine	DIR	BD dd	4	pssp	– 1 1 –	– – – –
JSR *opr16a*	PC ← (PC) + *n* (*n* = 1, 2, or 3)	EXT	CD hh ll	5	ppssp		
JSR *oprx16*,X	Push (PCL); SP ← (SP) – $0001	IX2	DD ee ff	6	ppssdp		
JSR *oprx8*,X	Push (PCH); SP ← (SP) – $0001	IX1	ED ff	5	pssdp		
JSR ,X	PC ← Unconditional Address	IX	FD	4	pssp		
LDA *#opr8i*		IMM	A6 ii	2	pp	0 1 1 –	– ↕ ↕ –
LDA *opr8a*		DIR	B6 dd	3	prp		
LDA *opr16a*		EXT	C6 hh ll	4	pprp		
LDA *oprx16*,X	Load Accumulator from Memory	IX2	D6 ee ff	4	pppr		
LDA *oprx8*,X	A ← (M)	IX1	E6 ff	3	ppr		
LDA ,X		IX	F6	2	pr		
LDA *oprx16*,SP		SP2	9E D6 ee ff	5	ppppr		
LDA *oprx8*,SP		SP1	9E E6 ff	4	pppr		
LDHX *#opr*	Load Index Register (H:X)	IMM	45 ii jj	3	ppp	0 1 1 –	– ↕ ↕ –
LDHX *opr*	H:X ← (M:M + $0001)	DIR	55 dd	4	prrp		
LDX *#opr8i*		IMM	AE ii	2	pp	0 1 1 –	– ↕ ↕ –
LDX *opr8a*		DIR	BE dd	3	prp		
LDX *opr16a*		EXT	CE hh ll	4	pprp		
LDX *oprx16*,X	Load X (Index Register Low) from Memory	IX2	DE ee ff	4	pppr		
LDX *oprx8*,X	X ← (M)	IX1	EE ff	3	ppr		
LDX ,X		IX	FE	2	pr		
LDX *oprx16*,SP		SP2	9E DE ee ff	5	ppppr		
LDX *oprx8*,SP		SP1	9E EE ff	4	pppr		
LSL *opr8a*	Logical Shift Left	DIR	38 dd	4	prwp	↕ 1 1 –	– ↕ ↕ ↕
LSLA		INH	48	1	p		
LSLX		INH	58	1	p		
LSL *oprx8*,X		IX1	68 ff	4	pprw		
LSL ,X		IX	78	3	prw		
LSL *oprx8*,SP	(Same as ASL)	SP1	9E 68 ff	5	ppprw		

Table 1. Instruction Set Summary (Sheet 6 of 9)

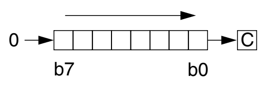
Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Affect on CCR							
						V	1	1	H	I	N	Z	C
LSR *opr8a*	Logical Shift Right 	DIR	34 dd	4	prwp	↕	1	1	–	–	0	↕	↕
LSRA		INH	44	1	p								
LSR*X*		INH	54	1	p								
LSR *oprx8*,X		IX1	64 ff	4	pprw								
LSR ,X		IX	74	3	prw								
LSR *oprx8*,SP		SP1	9E 64 ff	5	ppprw								
MOV *opr8a*,*opr8a*	Move $(M)_{destination} \leftarrow (M)_{source}$ In IX+/DIR and DIR/IX+ Modes, H:X ← (H:X) + $0001	DIR/DIR	4E dd dd	5	prpwp	0	1	1	–	–	↕	↕	–
MOV *opr8a*,X+		DIR/IX+	5E dd	4	prwp								
MOV #*opr8i*,*opr8a*		IMM/DIR	6E ii dd	4	ppwp								
MOV ,X+,*opr8a*		IX+/DIR	7E dd	4	prwp								
MUL	Unsigned multiply X:A ← (X) × (A)	INH	42	5	ppddd	–	1	1	0	–	–	–	0
NEG *opr8a*	Negate (Two's Complement) M ← – (M) = $00 – (M)	DIR	30 dd	4	prwp	↕	1	1	–	–	↕	↕	↕
NEGA	A ← – (A) = $00 – (A)	INH	40	1	p								
NEGX	X ← – (X) = $00 – (X)	INH	50	1	p								
NEG *oprx8*,X	M ← – (M) = $00 – (M)	IX1	60 ff	4	pprw								
NEG ,X	M ← – (M) = $00 – (M)	IX	70	3	prw								
NEG *oprx8*,SP	M ← – (M) = $00 – (M)	SP1	9E 60 ff	5	ppprw								
NOP	No Operation — Uses 1 Bus Cycle	INH	9D	1	p	–	1	1	–	–	–	–	–
NSA	Nibble Swap Accumulator A ← (A[3:0]:A[7:4])	INH	62	3	ppd	–	1	1	–	–	–	–	–
ORA #*opr8i*	Inclusive OR Accumulator and Memory A ← (A) ǀ (M)	IMM	AA ii	2	pp	0	1	1	–	–	↕	↕	–
ORA *opr8a*		DIR	BA dd	3	prp								
ORA *opr16a*		EXT	CA hh ll	4	pprp								
ORA *oprx16*,X		IX2	DA ee ff	4	pppr								
ORA *oprx8*,X		IX1	EA ff	3	ppr								
ORA ,X		IX	FA	2	pr								
ORA *oprx16*,SP		SP2	9E DA ee ff	5	ppppr								
ORA *oprx8*,SP		SP1	9E EA ff	4	pppr								
PSHA	Push Accumulator onto Stack Push (A); SP ← (SP) – $0001	INH	87	2	ps	–	1	1	–	–	–	–	–
PSHH	Push H (Index Register High) onto Stack Push (H); SP ← (SP) – $0001	INH	8B	2	ps	–	1	1	–	–	–	–	–
PSHX	Push X (Index Register Low) onto Stack Push (X); SP ← (SP) – $0001	INH	89	2	ps	–	1	1	–	–	–	–	–
PULA	Pull Accumulator from Stack SP ← (SP + $0001); Pull (A)	INH	86	2	pu	–	1	1	–	–	–	–	–
PULH	Pull H (Index Register High) from Stack SP ← (SP + $0001); Pull (H)	INH	8A	2	pu	–	1	1	–	–	–	–	–
PULX	Pull X (Index Register Low) from Stack SP ← (SP + $0001); Pull (X)	INH	88	2	pu	–	1	1	–	–	–	–	–

Table 1. Instruction Set Summary (Sheet 7 of 9)

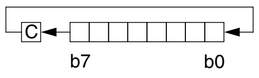
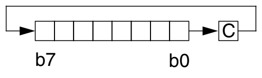
Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Affect on CCR							
						V	1	1	H	I	N	Z	C
ROL *opr8a* ROLA ROLX ROL *oprx8*,X ROL ,X ROL *oprx8*,SP	Rotate Left through Carry 	DIR INH INH IX1 IX SP1	39 dd 49 59 69 ff 79 9E 69 ff	4 1 1 4 3 5	prwp p p pprw prw ppprw	↕	1	1	–	–	↕	↕	↕
ROR *opr8a* RORA RORX ROR *oprx8*,X ROR ,X ROR *oprx8*,SP	Rotate Right through Carry 	DIR INH INH IX1 IX SP1	36 dd 46 56 66 ff 76 9E 66 ff	4 1 1 4 3 5	prwp p p pprw prw ppprw	↕	1	1	–	–	↕	↕	↕
RSP	Reset Stack Pointer (Low Byte) SPL ← $FF (High Byte Not Affected)	INH	9C	1	p	–	1	1	–	–	–	–	–
RTI	Return from Interrupt SP ← (SP) + $0001; Pull (CCR) SP ← (SP) + $0001; Pull (A) SP ← (SP) + $0001; Pull (X) SP ← (SP) + $0001; Pull (PCH) SP ← (SP) + $0001; Pull (PCL)	INH	80	7	puuuuup	↕	1	1	↕	↕	↕	↕	↕
RTS	Return from Subroutine SP ← SP + $0001; Pull (PCH) SP ← SP + $0001; Pull (PCL)	INH	81	4	puup	–	1	1	–	–	–	–	–
SBC #*opr8i* SBC *opr8a* SBC *opr16a* SBC *oprx16*,X SBC *oprx8*,X SBC ,X SBC *oprx16*,SP SBC *oprx8*,SP	Subtract with Carry A ← (A) – (M) – (C)	IMM DIR EXT IX2 IX1 IX SP2 SP1	A2 ii B2 dd C2 hh ll D2 ee ff E2 ff F2 9E D2 ee ff 9E E2 ff	2 3 4 4 3 2 5 4	pp prp pprp pppr ppr pr ppppr pppr	↕	1	1	–	–	↕	↕	↕
SEC	Set Carry Bit (C ← 1)	INH	99	1	p	–	1	1	–	–	–	–	1
SEI	Set Interrupt Mask Bit (I ← 1)	INH	9B	2	pd	–	1	1	–	1	–	–	–
STA *opr8a* STA *opr16a* STA *oprx16*,X STA *oprx8*,X STA ,X STA *oprx16*,SP STA *oprx8*,SP	Store Accumulator in Memory M ← (A)	DIR EXT IX2 IX1 IX SP2 SP1	B7 dd C7 hh ll D7 ee ff E7 ff F7 9E D7 ee ff 9E E7 ff	3 4 4 3 2 5 4	pwp ppwp pppw ppw pw ppppw pppw	0	1	1	–	–	↕	↕	–
STHX *opr*	Store H:X (Index Reg.) (M:M + $0001) ← (H:X)	DIR	35 dd	4	pwwp	0	1	1	–	–	↕	↕	–
STOP	Enable Interrupts: Stop Processing Refer to MCU Documentation I bit ← 0; Stop Processing	INH	8E	1	p	–	1	1	–	0	–	–	–

Table 1. Instruction Set Summary (Sheet 8 of 9)

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Affect on CCR	
						V 1 1 H	I N Z C

STX *opr8a*
STX *opr16a*
STX *oprx16*,X
STX *oprx8*,X
STX ,X
STX *oprx16*,SP
STX *oprx8*,SP
	Store X (Low 8 Bits of Index Register) in Memory
	M ← (X)
		DIR
		EXT
		IX2
		IX1
		IX
		SP2
		SP1
			`BF dd`
			`CF hh ll`
			`DF ee ff`
			`EF ff`
			`FF`
			`9E DF ee ff`
			`9E EF ff`
				3
				4
				4
				3
				2
				5
				4
					`pwp`
					`ppwp`
					`pppw`
					`ppw`
					`pw`
					`ppppw`
					`pppw`
						0 1 1 –	– ↕ ↕ –

SUB *#opr8i*
SUB *opr8a*
SUB *opr16a*
SUB *oprx16*,X
SUB *oprx8*,X
SUB ,X
SUB *oprx16*,SP
SUB *oprx8*,SP
	Subtract
	A ← (A) – (M)
		IMM
		DIR
		EXT
		IX2
		IX1
		IX
		SP2
		SP1
			`A0 ii`
			`B0 dd`
			`C0 hh ll`
			`D0 ee ff`
			`E0 ff`
			`F0`
			`9E D0 ee ff`
			`9E E0 ff`
				2
				3
				4
				4
				3
				2
				5
				4
					`pp`
					`prp`
					`pprp`
					`pppr`
					`ppr`
					`pr`
					`ppppr`
					`pppr`
						↕ 1 1 –	– ↕ ↕ ↕

SWI	Software Interrupt PC ← (PC) + $0001 Push (PCL); SP ← (SP) – $0001 Push (PCH); SP ← (SP) – $0001 Push (X); SP ← (SP) – $0001 Push (A); SP ← (SP) – $0001 Push (CCR); SP ← (SP) – $0001 I ← 1; PCH ← Interrupt Vector High Byte PCL ← Interrupt Vector Low Byte	INH	`83`	9	`psssssvvp`	– 1 1 –	1 – – –
TAP	Transfer Accumulator to CCR CCR ← (A)	INH	`84`	2	`pd`	↕ 1 1 ↕	↕ ↕ ↕ ↕
TAX	Transfer Accumulator to X (Index Register Low) X ← (A)	INH	`97`	1	`p`	– 1 1 –	– – – –
TPA	Transfer CCR to Accumulator A ← (CCR)	INH	`85`	1	`p`	– 1 1 –	– – – –

TST *opr8a*
TSTA
TSTX
TST *oprx8*,X
TST ,X
TST *oprx8*,SP
	Test for Negative or Zero (M) – $00
	(A) – $00
	(X) – $00
	(M) – $00
	(M) – $00
	(M) – $00
		DIR
		INH
		INH
		IX1
		IX
		SP1
			`3D dd`
			`4D`
			`5D`
			`6D ff`
			`7D`
			`9E 6D ff`
				3
				1
				1
				3
				2
				4
					`prp`
					`p`
					`p`
					`ppr`
					`pr`
					`pppr`
						0 1 1 –	– ↕ ↕ –

TSX	Transfer SP to Index Reg. H:X ← (SP) + $0001	INH	`95`	2	`pp`	– 1 1 –	– – – –
TXA	Transfer X (Index Reg. Low) to Accumulator A ← (X)	INH	`9F`	1	`p`	– 1 1 –	– – – –

Table 1. Instruction Set Summary (Sheet 9 of 9)

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Affect on CCR V 1 1 H	Affect on CCR I N Z C
TXS	Transfer Index Reg. to SP SP ← (H:X) – $0001	INH	94	2	pp	– 1 1 –	– – – –
WAIT	Enable Interrupts; Wait for Interrupt I bit ← 0; Halt CPU	INH	8F	1	p	– 1 1 –	0 – – –

Object Code:

dd	Direct address of operand
ee ff	High and low bytes of offset in indexed, 16-bit offset addressing
ff	Offset byte in indexed, 8-bit offset addressing
hh ll	High and low bytes of operand address in extended addressing
ii	Immediate operand byte
ii jj	16-bit immediate operand for H:X
rr	Relative program counter offset byte

Addressing Modes:

DIR	Direct addressing mode
EXT	Extended addressing mode
IMM	Immediate addressing mode
INH	Inherent addressing mode
IX	Indexed, no offset addressing mode
IX1	Indexed, 8-bit offset addressing mode
IX2	Indexed, 16-bit offset addressing mode
IX+	Indexed, no offset, post increment addressing mode
IX1+	Indexed, 8-bit offset, post increment addressing mode
REL	Relative addressing mode
SP1	Stack pointer, 8-bit offset addressing mode
SP2	Stack pointer 16-bit offset addressing mode

CCR Bits, Effects:

V	Overflow bit
H	Half-carry bit
I	Interrupt mask
N	Negative bit
Z	Zero bit
C	Carry/borrow bit
↕	Set or cleared
–	Not affected
U	Undefined

Operation Symbols:

A	Accumulator
CCR	Condition code register
H	Index register high byte
M	Memory location
n	Any bit
opr	Operand (one or two bytes)
PC	Program counter
PCH	Program counter high byte
PCL	Program counter low byte
rel	Relative program counter offset byte
SP	Stack pointer
X	Index register low byte
&	Logical AND
|	Logical OR
⊕	Logical EXCLUSIVE OR
( )	Contents of
–( )	Negation (two's complement)
#	Immediate value
«	Sign extend
←	Loaded with
?	If
:	Concatenated with

Cycle-by-Cycle Codes:

d	Dummy duplicate of the previous p, r, or s cycle. d is always a read cycle so sd is a stack write followed by a read of the address pointed-to by the updated stack pointer
p	Program fetch; read from next consecutive location in program memory
r	Read 8-bit operand
s	Push (write) one byte onto stack
u	Pop (read) one byte from stack
v	Read vector from $FFxx (high byte first)
w	Write 8-bit operand

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

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

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

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

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

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