RS08

Core Reference Manual

RS08RM
Rev 1.0
04/2006

freescale.com

freescale™ semiconductor

Revision History

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

http://freescale.com

The following revision history table summarizes changes contained in this document.

Revision Number	Revision Date	Description of Changes
1.0	4/2006	Initial public release version

This product incorporates SuperFlash® technology licensed from SST.

Freescale, and the Freescale logo are trademarks of Freescale Semiconductor, Inc.
© Freescale Semiconductor, Inc., 2006. All rights reserved.

List of Sections

Section 1. General Information and Block Diagram . . . 11

Section 2. Central Processor Unit (CPU) 17

Section 3. Development Support 55

Appendix A. Instruction Set Details 85

Appendix B. Code Examples 147

Appendix C. Assembler and Disassembler
Style Guide 151

Table of Contents

Section 1. General Information and Block Diagram

1.1 Introduction to the RS08 Family of Microcontrollers 11

1.2 Memory Map for the RS08 Family 12

1.3 RS08 Core Paging Scheme 14

Section 2. Central Processor Unit (CPU)

2.1 Introduction 17

2.2 Programmer's Model and CPU Registers 17
2.2.1 Accumulator (A) 18
2.2.2 Program Counter (PC) 19
2.2.3 Shadow Program Counter (SPC) 19
2.2.4 Condition Code Register (CCR) 20
2.2.5 Indexed Data Register (D[X]) 22
2.2.6 Index Register (X) 22
2.2.7 Indexed/Indirect Addressing 22
2.2.8 Page Select Register (PAGESEL) 23

2.3 Addressing Modes 24
2.3.1 Inherent Addressing Mode (INH) 25
2.3.2 Relative Addressing Mode (REL) 25
2.3.3 Immediate Addressing Mode (IMM) 26
2.3.4 Tiny Addressing Mode (TNY) 27
2.3.5 Short Addressing Mode (SRT) 27
2.3.6 Direct Addressing Mode (DIR) 28
2.3.7 Extended Addressing Mode (EXT) 28
2.3.8 Indexed Addressing Mode
(IX, Implemented by Pseudo Instructions) 28

2.4 Special Operations 29
2.4.1 Reset Sequence 29
2.4.2 Interrupts 30
2.4.3 Wait and Stop Mode 30
2.4.4 Active Background Mode 30

2.5 Instruction Set Description by Instruction Types 31

2.5.1 Data Movement Instructions 33
2.5.1.1 Loads and Stores 34
2.5.1.2 Bit Set and Bit Clear 35
2.5.1.3 Memory-to-Memory Moves 36
2.5.2 Math Instructions 36
2.5.2.1 Add, and Subtract 36
2.5.2.2 Increment, Decrement, and Clear 37
2.5.2.3 Compare 37
2.5.3 Logical Operation Instructions 38
2.5.3.1 AND, OR, Exclusive-OR, and Complement 38
2.5.4 Shift and Rotate Instructions 39
2.5.5 Jump, Branch, and Loop Control Instructions 39
2.5.5.1 Unconditional Jump and Branch 40
2.5.5.2 Simple Branches 41
2.5.5.3 Unsigned Branches 41
2.5.5.4 Bit Condition Branches 43
2.5.5.5 Loop Control 43
2.5.6 Subroutine-Related Instructions 43
2.5.7 Miscellaneous Instructions 44
2.6 Summary Instruction Table 45

Section 3. Development Support

3.1 Introduction 55
3.2 Features 56
3.3 RS08 Background Debug Controller (BDC) 56
3.3.1 BKGD Pin Description 58
3.3.2 Communication Details 59
3.3.3 SYNC and Serial Communication Timeout 62
3.4 BDC Registers and Control Bits 63
3.4.1 BDC Status and Control Register (BDCSCR) 64
3.4.2 BDC Breakpoint Match Register 65
3.5 RS08 BDC Commands 66
3.5.1 SYNC 69
3.5.2 BDC_RESET 70
3.5.3 BACKGROUND 70
3.5.4 READ_STATUS 71
3.5.5 WRITE_CONTROL 72
3.5.6 READ_BYTE 73
3.5.7 READ_BYTE_WS 74

3.5.8 WRITE_BYTE 74
3.5.9 WRITE_BYTE_WS 75
3.5.10 READ_BKPT 76
3.5.11 WRITE_BKPT 76
3.5.12 GO 77
3.5.13 TRACE1 77
3.5.14 READ_BLOCK 78
3.5.15 WRITE_BLOCK 78
3.5.16 READ_A 79
3.5.17 WRITE_A 79
3.5.18 READ_CCR_PC 79
3.5.19 WRITE_CCR_PC 80
3.5.20 READ_SPC 81
3.5.21 WRITE_SPC 81

3.6 BDC Hardware Breakpoint 81

3.7 BDM in Stop and Wait Modes. 82

3.8 BDC Command Execution 83

Appendix A. Instruction Set Details

A.1 Introduction 85
A.2 Nomenclature 85
A.3 Convention Definitions 89
A.4 Use of ‘X’, ‘,X’ and ‘D[X]’ as instruction operands. 89
A.5 Instruction Set. 90
ADC Add with Carry 91
ADD Add without Carry 92
AND Logical AND 93
ASLA Arithmetic Shift Left. 94
BCC Branch if Carry Bit Clear 95
BCLR *n* Clear Bit *n* in Memory .96
BCLR *n* Clear Bit *n* in Memory .97
BCS Branch if Carry Bit Set .98
BEQ Branch if Equal 99
BGND Background. 100
BHS Branch if Higher or Same 101
BLO Branch if Lower 102
(Pseudo Instruction, same as BCS) 102
BNE Branch if Not Equal 103
BRA Branch Always 104
BRA Branch Always 105

BRN Branch Never 106
BRCLR *n* Branch if Bit *n* in Memory Clear 107
BRCLR *n* Branch if Bit *n* in Memory Clear 108
BRSET *n* Branch if Bit *n* in Memory Set 109
BRSET *n* Branch if Bit *n* in Memory Set 110
BSET *n* Set Bit *n* in Memory. 111
BSET *n* Set Bit *n* in Memory. 112
BSR Branch to Subroutine 113
CBEQ Compare and Branch if Equal. 114
CLC Clear Carry Bit 115
CLR Clear 116
CMP Compare Accumulator with Memory. 117
COMA Complement (One’s Complement) 118
DBNZ Decrement and Branch if Not Zero 119
DEC Decrement 120
EOR Exclusive-OR Memory with Accumulator 121
INC Increment 122
JMP Jump 123
JSR Jump to Subroutine. 124
LDA Load Accumulator from Memory. 125
LDX Load X Index Register from Memory 126
LSLA Logical Shift Left 127
LSRA Logical Shift Right. 128
MOV Move 129
NOP No Operation. 130
ORA Inclusive-OR Accumulator and Memory 131
ROLA Rotate Left through Carry 132
RORA Rotate Right through Carry. 133
RTS Return from Subroutine. 134
SBC Subtract with Carry 135
SEC Set Carry Bit 136
SHA Swap Shadow PC High with A 137
SLA Swap Shadow PC Low with 138
STA Store Accumulator in Memory. 139
STOP Stop Processing 140
STX Store X (Index Register Low) in Memory 141
SUB Subtract. 142
TAX Transfer A to X 143
TST Test for Zero 144
TXA Transfer X to A 145
WAIT Stop CPU Clock 146

Appendix B. Code Examples

B.1 Illegal Table 147
B.2 lda 148
B.3 probe. 148
B.4 walk. 149

Appendix C. Assembler and Disassembler Style Guide

C.1 Support Notes for RS08 Tools 151
C.1.1 Pseudo Instructions 151
C.1.2 Tiny and Short Addressing 151
C.1.3 Forcing Tiny/Short and Direct Addressing 152
C.1.4 Unsupported Instructions 152
C.1.5 Tiny, Short, and Direct Address Usage Statistics. 153
C.2 Debugger and Disassembly 153
C.2.1 HC(S)08 Style 153
C.2.2 RS08 Style 154
C.2.3 Native RS08 Style 154
C.3 Compilers 154

Section 1. General Information and Block Diagram

1.1 Introduction to the RS08 Family of Microcontrollers

Freescale's RS08 Family of microcontrollers (MCUs) uses a reduced version of the HCS08 central processor unit (CPU). The RS08 Family is targeted for small embedded applications. When working with an individual member of the RS08 Family of MCUs, refer to the device data sheet for information specific to the MCU.

Each MCU device in the RS08 Family consists of the RS08 core plus several memory and peripheral modules. The RS08 core consists of:

- RS08 CPU (reduced HCS08)
- Background debug controller (BDC)
- Chip-level address decoder

The RS08 CPU executes a subset of HCS08 instructions, with minor extensions. See Section 8, "Central Processor Unit (RS08CPUV1)," for more information.

The background debug controller (BDC) is built into the CPU core to allow easier access to address generation circuits and CPU register information. The BDC includes one hardware breakpoint. The BDC allows access to internal register and memory locations via a single pin on the MCU. See Section Chapter 12, "Development Support," for more information.

The core includes support for various interrupts for wakeup after STOP/WAIT instruction execution and various reset sources. The peripheral modules provide local interrupt enable circuitry and flag registers. See Section Chapter 8, "Central Processor Unit (RS08CPUV1)," for more information.

1.2 Memory Map for the RS08 Family

A portion of the memory map has been standardized. The most frequently used registers for input/output (I/O) ports and control and status registers for peripheral modules are located between $0010 and $001E. The space from $0000 to $000E and $0020 to $01FF are reserved for static RAM memory. A space between $0200 to $03FF is reserved for high-page registers.

The RS08 has also introduced a standard paging scheme for the CPU to access the whole 16K memory space using a 64-byte page window located at $00C0 to $00FF. Details are described in the following section.

Figure 1-1 shows the memory map of a typical RS08 Family device.

Address	Region	PAGESEL REGISTER CONTENT
$0000–$000D	FAST ACCESS RAM 14 BYTES	$00
$000E	D[X]	
$000F	REGISTER X	
$0010–$001E	FREQUENTLY USED REGISTERS	
$001F	PAGESEL	
$0020–$004F	RAM 48 BYTES	
	UNIMPLEMENTED	
$00C0–$00FF	PAGING WINDOW	
	UNIMPLEMENTED	
$0200–$023F	HIGH PAGE REGISTERS	$08 (reset value)
	UNIMPLEMENTED	
$3800–$3FFB	FLASH 2044 BYTES	$E0
$3FFC	NVOPT	
$3FFD–$3FFF	FLASH	

Figure 1-1. Typical RS08 Memory Map

1.3 RS08 Core Paging Scheme

Because the RS08 core does not support extended memory access for data access beyond the first 256 bytes (commonly known as direct or page 0), a paging scheme has been implemented. This scheme segments the full 16-Kbyte address map of the RS08 core into 256 pages of 64 bytes each. The direct/page 0 address range ($0000–$00FF) is mapped into the first four 64-byte pages. The range of pages of typical interest for a 2-Kbyte FLASH device with FLASH located at $3800–3FFF is provided in Table 1-1:

Table 1-1. RS08 Paging Scheme

Page Number	Address
0	$0000–$003F
1	$0040–$007F
2	$0080–$00BF
3	$00C0–$00FF
4–7	$0100–$01FF
8	$0200–$023F
9–223	$0240–$37C0
224	$3800–$383F
225–255	$3840–$3FFF

To access the data in different pages, the 64-byte high page access window located at $00C0–$00FF must be used in the direct page. To position the high page access window to the desired address range within the 16K address space of the RS08, the PAGESEL register (location $001F) must be set with the appropriate value. As soon as the PAGESEL register has been written with the appropriate value, subsequent accesses to the high page access window will address the area determined by the PAGESEL register.

The following code example shows the procedure to access the first, eighth, and ninth byte of FLASH memory starting at location $3800 using the high page access window and indexed addressing mode.

```
PAGESEL:EQU     $001F  ;Page select reg. for High Page Access Window
HPAWS:  EQU     $00C0  ;Start address of High Page Access Window

;Set High Page Access Window to point to first page of flash
        MOV     #$E0,PAGESEL
        LDA     $00C0  ;Read contents of first byte in flash

;Use the Index register to access the 8th byte of the flash ($3807)
        LDX     #$C7
        LDA     ,X

;Move 9th byte of Flash to $0000 (location $3808)
        INCX
        LDA     ,X
        STA     $00
```

Section 2. Central Processor Unit (CPU)

2.1 Introduction

The RS08 CPU has been developed to target extremely low-cost embedded applications using a process-independent design methodology, allowing it to keep pace with rapid developments in silicon processing technology.

The main features of the RS08 core are:

- Streamlined programmer's model
- Subset of HCS08 instruction set with minor instruction extensions
- Minimal instruction set for cost-sensitive embedded applications
- New instructions for shadow program counter manipulation, SHA and SLA
- New short and tiny addressing modes for code size optimization
- 16K bytes accessible memory space
- Reset will fetch the first instruction from $3FFD
- Low-power modes supported through the execution of the STOP and WAIT instructions
- Debug and FLASH programming support using the background debug controller module
- Illegal address and opcode detection with reset

2.2 Programmer's Model and CPU Registers

Figure 2-1 shows the programmer's model for the RS08 CPU. These registers are not located in the memory map of the microcontroller. They are built directly inside the CPU logic.

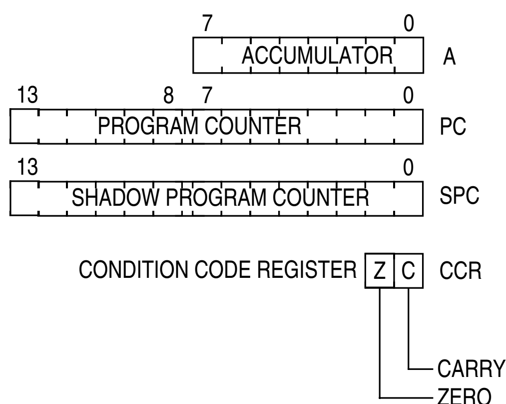


Figure 2-1. CPU Registers

In addition to the CPU registers, there are three memory mapped registers that are tightly coupled with the core address generation during data read and write operations. They are the indexed data register (D[X]), the index register (X), and the page select register (PAGESEL). These registers are located at $000E, $000F, and $001F, respectively.

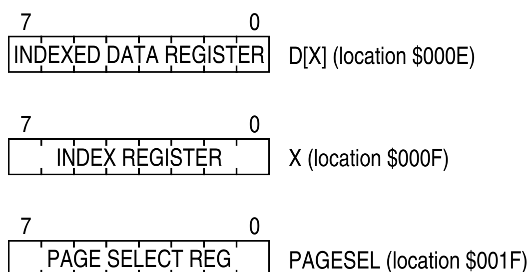


Figure 2-2. Memory Mapped Registers

2.2.1 Accumulator (A)

This general-purpose 8-bit register is the primary data register for RS08 MCUs. Data can be read from memory into A with a load accumulator (LDA) instruction. The data in A can be written into memory with a store accumulator (STA) instruction. Various addressing mode variations allow a great deal of flexibility in specifying the memory location involved in a load or store instruction. Exchange instructions allow values to be

exchanged between A and SPC high (SHA) and also between A and SPC low (SLA).

Arithmetic, shift, and logical operations can be performed on the value in A as in ADD, SUB, RORA, INCA, DECA, AND, ORA, EOR, etc. In some of these instructions, such as INCA and LSLA, the value in A is the only input operand and the result replaces the value in A. In other cases, such as ADD and AND, there are two operands: the value in A and a second value from memory. The result of the arithmetic or logical operation replaces the value in A.

Some instructions, such as memory-to-memory move instructions (MOV), do not use the accumulator. DBNZ also relieves A because it allows a loop counter to be implemented in a memory variable rather than the accumulator.

During reset, the accumulator is loaded with $00.

2.2.2 Program Counter (PC)

The program counter is a 14-bit register that contains the address of the next instruction or operand to be fetched.

During normal execution, the program counter automatically increments to the next sequential memory location each time an instruction or operand is fetched. Jump, branch, and return operations load the program counter with an address other than that of the next sequential location. This is called a change-of-flow.

During reset, the program counter is loaded with $3FFD and the program will start execution from this specific location.

2.2.3 Shadow Program Counter (SPC)

The shadow program counter is a 14-bit register. During a subroutine call using either a JSR or a BSR instruction, the return address will be saved into the SPC. Upon completion of the subroutine, the RTS instruction will restore the content of the program counter from the shadow program counter.

During reset, the shadow program counter is loaded with $3FFD.

2.2.4 Condition Code Register (CCR)

The 2-bit condition code register contains two status flags. The content of the CCR in the RS08 is not directly readable. The CCR bits can be tested using conditional branch instructions such as BCC and BEQ. These two register bits are directly accessible through the BDC interface. The following paragraphs provide detailed information about the CCR bits and how they are used. Figure 2-3 identifies the CCR bits and their bit positions.

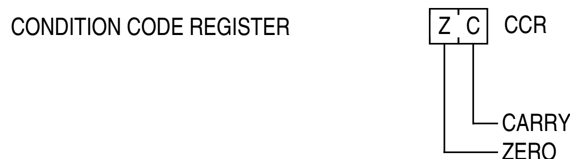


Figure 2-3. Condition Code Register (CCR)

The status bits (Z and C) are cleared to 0 after reset.

The two status bits indicate the results of arithmetic and other instructions. Conditional branch instructions will either branch to a new program location or allow the program to continue to the next instruction after the branch, depending on the values in the CCR status bit. Conditional branch instructions, such as BCC, BCS, and BNE, cause a branch depending on the state of a single CCR bit.

Often, the conditional branch immediately follows the instruction that caused the CCR bit(s) to be updated, as in this sequence:

```
         cmp    #5        ;compare accumulator A to 5
         blo    lower     ;branch if A smaller 5
more:    deca             ;do this if A not higher than or same as 5
lower:
```

Other instructions may be executed between the test and the conditional branch as long as the only instructions used are those which do not disturb the CCR bits that affect the conditional branch. For instance, a test is performed in a subroutine or function and the conditional branch

is not executed until the subroutine has returned to the main program. This is a form of parameter passing (that is, information is returned to the calling program in the condition code bits).

Z — Zero Flag

The Z bit is set to indicate the result of an operation was $00.

Branch if equal (BEQ) and branch if not equal (BNE) are simple branches that branch based solely on the value in the Z bit. All load, store, move, arithmetic, logical, shift, and rotate instructions cause the Z bit to be updated.

C — Carry

After an addition operation, the C bit is set if the source operands were both greater than or equal to $80 or if one of the operands was greater than or equal to $80 and the result was less than $80. This is equivalent to an unsigned overflow. A subtract or compare performs a subtraction of a memory operand from the contents of a CPU register so after a subtract operation, the C bit is set if the unsigned value of the memory operand was greater than the unsigned value of the CPU register. This is equivalent to an unsigned borrow or underflow.

Branch if carry clear (BCC) and branch if carry set (BCS) are branches that branch based solely on the value in the C bit. The C bit is also used by the unsigned branches BLO and BHS. Add, subtract, shift, and rotate instructions cause the C bit to be updated. The branch if bit set (BRSET) and branch if bit clear (BRCLR) instructions copy the tested bit into the C bit to facilitate efficient serial-to-parallel conversion algorithms. Set carry (SEC) and clear carry (CLC) allow the carry bit to be set or cleared directly. This is useful in combination with the shift and rotate instructions and for routines that pass status information back to a main program, from a subroutine, in the C bit.

The C bit is included in shift and rotate operations so those operations can easily be extended to multi-byte operands. The shift and rotate operations can be considered 9-bit shifts that include an 8-bit operand or CPU register and the carry bit of the

CCR. After a logical shift, C holds the bit that was shifted out of the 8-bit operand. If a rotate instruction is used next, this C bit is shifted into the operand for the rotate, and the bit that gets shifted out the other end of the operand replaces the value in C so it can be used in subsequent rotate instructions.

2.2.5 Indexed Data Register (D[X])

This 8-bit indexed data register allows the user to access the data in the direct page address space indexed by X. This register resides at the memory mapped location $000E. For details on the D[X] register, please refer to Section 2.3.8, "Indexed Addressing Mode (IX, Implemented by Pseudo Instructions)."

2.2.6 Index Register (X)

This 8-bit index register allows the user to index or address any location in the direct page address space. This register resides at the memory mapped location $000F. For details on the X register, please refer to Section 2.3.8, "Indexed Addressing Mode (IX, Implemented by Pseudo Instructions)."

2.2.7 Indexed/Indirect Addressing

Register D[X] and register X together perform the indirect data access. Register D[X] is mapped to address $000E. Register X is located in address $000F. The 8-bit register X contains the address that is used when register D[X] is accessed. Register X is cleared to zero upon reset. By programming register X, any location on the first page ($0000–$00FF) can be read/written via register D[X]. Figure 2-4 shows the relationship between D[X] and register X. For example, in HC08/S08 syntax lda ,x is comparable to lda D[X] in RS08 coding when register X has been programmed with the index value.

The physical location of $000E is in RAM. Accessing the location through D[X] returns $000E RAM content when register X contains $0E. The physical location of $000F is register X, itself. Reading the location

through D[X] returns register X content; writing to the location modifies register X.

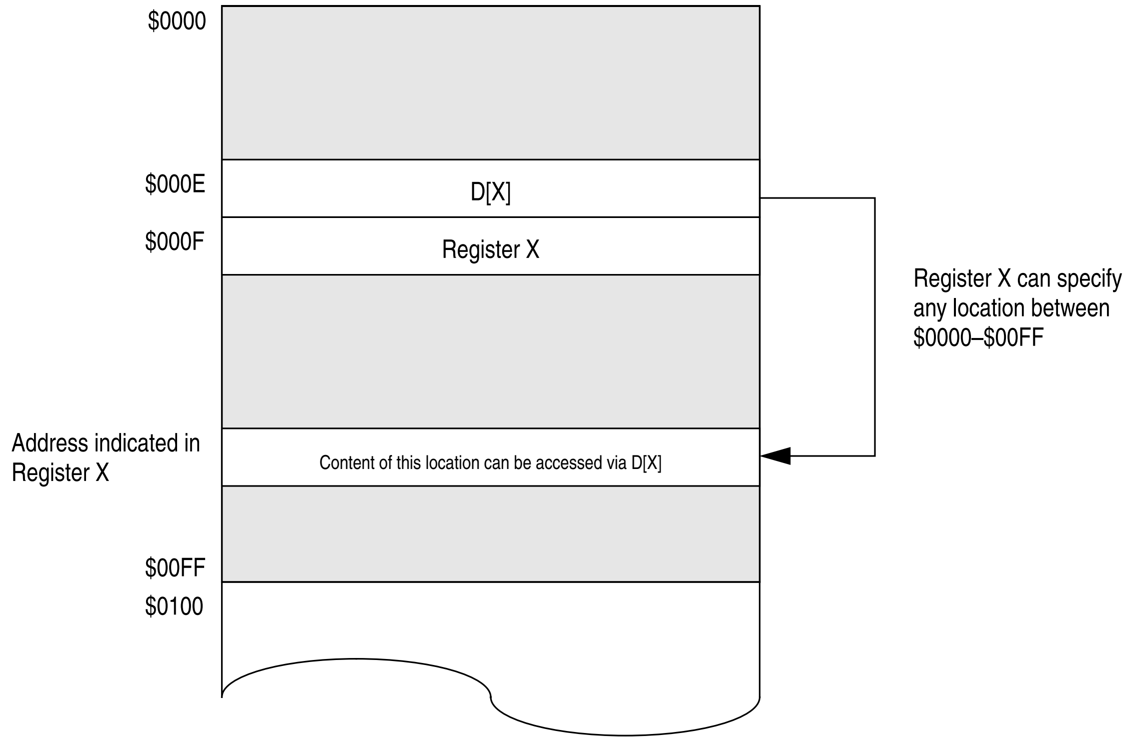


Figure 2-4. Indirect Addressing Registers

2.2.8 Page Select Register (PAGESEL)

This 8-bit page select register allows the user to access all memory locations in the entire 16K-byte address space through a page window located from $00C0 to $00FF. This register resides at the memory mapped location $001F.The paging scheme is shown in Figure 2-5.

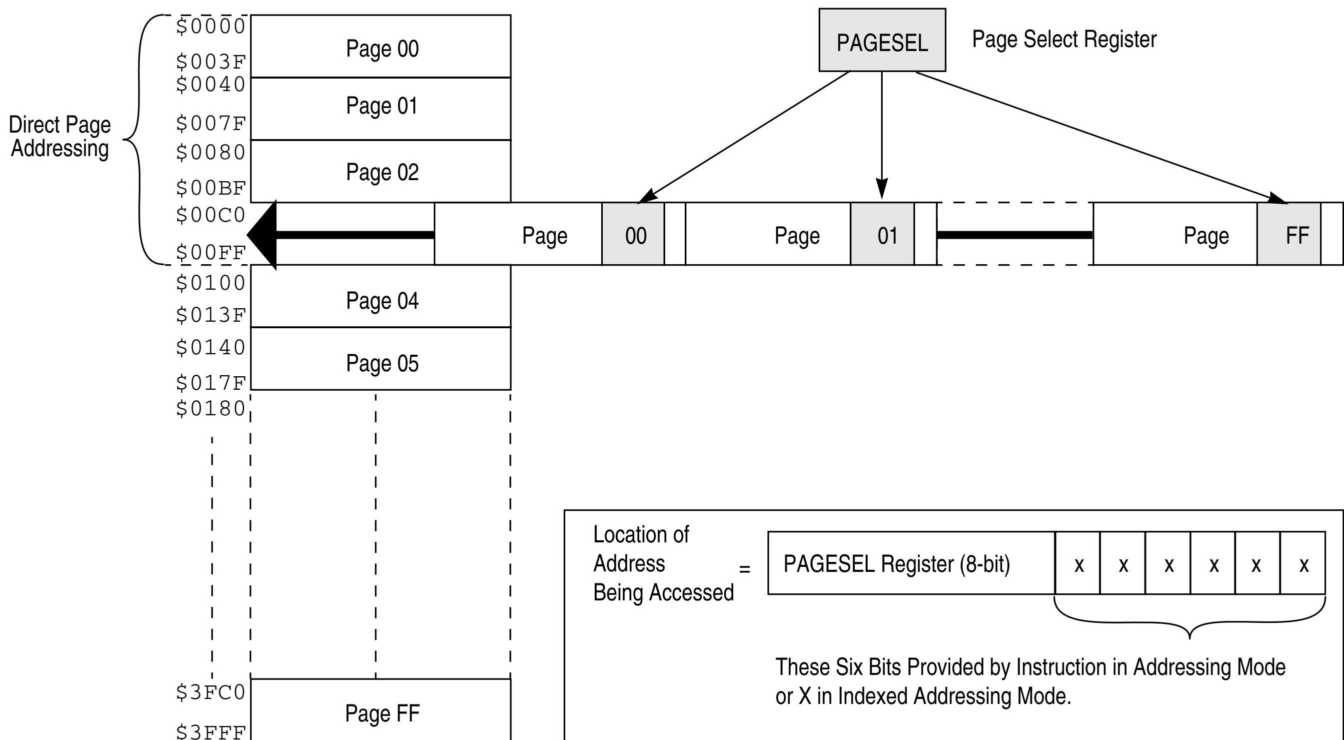


Figure 2-5. RS08 Paging Scheme

2.3 Addressing Modes

Whenever the MCU reads information from memory or writes information into memory, an addressing mode is used to determine the exact address where the information is read from or written to. This section explains several addressing modes and how each is useful in different programming situations.

Every opcode tells the CPU to perform a certain operation in a certain way. Many instructions, such as load accumulator (LDA), allow several different ways to specify the memory location to be operated on, and each addressing mode variation requires a separate opcode. All of these variations use the same instruction mnemonic, and the assembler knows which opcode to use based on the syntax and location of the operand field. In some cases, special characters are used to indicate a specific addressing mode (such as the # [pound] symbol, which indicates immediate addressing mode). In other cases, the value of the operand tells the assembler which addressing mode to use. For

example, the assembler chooses short addressing mode instead of direct addressing mode if the operand address is from $0000 to $001F. Besides allowing the assembler to choose the addressing mode based on the operand address, assembler directives can also be used to force direct or tiny/short addressing mode by using the “>” or “<” prefix before the operand, respectively.

Some instructions use more than one addressing mode. For example, the move instructions use one addressing mode to access the source value from memory and a second addressing mode to access the destination memory location. For these move instructions, both addressing modes are listed in the documentation. All branch instructions use relative (REL) addressing mode to determine the destination for the branch, but BRCLR, BRSET, CBEQ, and DBNZ also must access a memory operand. These instructions are classified by the addressing mode used for the memory operand, and the relative addressing mode for the branch offset is assumed.

The discussion in the following paragraphs includes how each addressing mode works and the syntax clues that instruct the assembler to use a specific addressing mode.

2.3.1 Inherent Addressing Mode (INH)

This addressing mode is used when the CPU inherently knows everything it needs to complete the instruction and no addressing information is supplied in the source code. Usually, the operands that the CPU needs are located in the CPU’s internal registers, as in LSLA, CLRA, INCA, SLA, RTS, and others. A few inherent instructions, including no operation (NOP) and background (BGND), have no operands.

2.3.2 Relative Addressing Mode (REL)

Relative addressing mode is used to specify the offset address for branch instructions relative to the program counter. Typically, the programmer specifies the destination with a program label or an expression in the operand field of the branch instruction; the assembler

calculates the difference between the location counter (which points at the next address after the branch instruction at the time) and the address represented by the label or expression in the operand field. This difference is called the offset and is an 8-bit two's complement number. The assembler stores this offset in the object code for the branch instruction.

During execution, the CPU evaluates the condition that controls the branch. If the branch condition is true, the CPU sign-extends the offset to a 14-bit value, adds the offset to the current PC, and uses this as the address where it will fetch the next instruction and continue execution rather than continuing execution with the next instruction after the branch. Because the offset is an 8-bit two's complement value, the destination must be within the range –128 to +127 locations from the address that follows the last byte of object code for the branch instruction.

A common method to create a simple infinite loop is to use a branch instruction that branches to itself. This is sometimes used to end short code segments during debug. Typically, to get out of this infinite loop, use the debug host (through background commands) to stop the program, examine registers and memory, or to start execution from a new location. This construct is not used in normal application programs except in the case where the program has detected an error and wants to force the COP watchdog timer to timeout. (The branch in the infinite loop executes repeatedly until the watchdog timer eventually causes a reset.)

2.3.3 Immediate Addressing Mode (IMM)

In this addressing mode, the operand is located immediately after the opcode in the instruction stream. This addressing mode is used when the programmer wants to use an explicit value that is known at the time the program is written. A # (pound) symbol is used to tell the assembler to use the operand as a data value rather than an address where the desired value should be accessed.

The size of the immediate operand is always 8 bits. The assembler automatically will truncate or extend the operand as needed to match the

size needed for the instruction. Most assemblers generate a warning if a 16-bit operand is provided.

It is the programmer's responsibility to use the # symbol to tell the assembler when immediate addressing should be used. The assembler does not consider it an error to leave off the # symbol because the resulting statement is still a valid instruction (although it may mean something different than the programmer intended).

2.3.4 Tiny Addressing Mode (TNY)

TNY addressing mode is capable of addressing only the first 16 bytes in the address map, from $0000 to $000F. This addressing mode is available for INC, DEC, ADD, and SUB instructions. A system can be optimized by placing the most computation-intensive data in this area of memory.

Because the 4-bit address is embedded in the opcode, only the least significant four bits of the address must be included in the instruction; this saves program space and execution time. During execution, the CPU adds 10 high-order 0s to the 4-bit operand address and uses the combined 14-bit address ($000x) to access the intended operand.

2.3.5 Short Addressing Mode (SRT)

SRT addressing mode is capable of addressing only the first 32 bytes in the address map, from $0000 to $001F. This addressing mode is available for CLR, LDA, and STA instructions. A system can be optimized by placing the most computation-intensive data in this area of memory.

Because the 5-bit address is embedded in the opcode, only the least significant five bits of the address must be included in the instruction; this saves program space and execution time. During execution, the CPU adds nine high-order 0s to the 5-bit operand address and uses the combined 14-bit address ($000x or $001x) to access the intended operand.

2.3.6 Direct Addressing Mode (DIR)

DIR addressing mode is used to access operands located in direct address space ($0000 through $00FF).

During execution, the CPU adds six high-order 0s to the low byte of the direct address operand that follows the opcode. The CPU uses the combined 14-bit address ($00xx) to access the intended operand.

2.3.7 Extended Addressing Mode (EXT)

In the extended addressing mode, the 14-bit address of the operand is included in the object code in the low-order 14 bits of the next two bytes after the opcode. This addressing mode is only used in JSR and JMP instructions for jump destination address in RS08 MCUs.

2.3.8 Indexed Addressing Mode (IX, Implemented by Pseudo Instructions)

Indexed addressing mode is sometimes called indirect addressing mode because an index register is used as a reference to access the intended operand.

An important feature of indexed addressing mode is that the operand address is computed during execution based on the current contents of the X index register located in $000F of the memory map rather than being a constant address location that was determined during program assembly. This allows writing of a program that accesses different operand locations depending on the results of earlier program instructions (rather than accessing a location that was determined when the program was written).

The index addressing mode supported by the RS08 Family uses the register X located at $000F as an index and D[X] register located at $000E as the indexed data register. By programming the index register X, any location in the direct page can be read/written via the indexed data register D[X].

These pseudo instructions can be used with all instructions supporting direct, short, and tiny addressing modes by using the D[X] as the operand.

2.4 Special Operations

Most of what the CPU does is described by the instruction set, but a few special operations must be considered, such as how the CPU starts at the beginning of an application program after power is first applied. After the program begins running, the current instruction normally determines what the CPU will do next. Two exceptional events can cause the CPU to temporarily suspend normal program execution:

- Reset events force the CPU to start over at the beginning of the application program, which forces execution to start at $3FFD.
- A host development system can cause the CPU to go to active background mode rather than continuing to the next instruction in the application program.

2.4.1 Reset Sequence

Processing begins at the trailing edge of a reset event. The number of things that can cause reset events can vary slightly from one RS08 derivative to another; however, the most common sources are: power-on reset, the external $\overline{\text{RESET}}$ pin, low-voltage reset, COP watchdog timeout, illegal opcode detect, and illegal address access. For more information about how the MCU recognizes reset events and determines the difference between internal and external causes, refer to the Resets and Interrupts chapter in the technical data sheet for the specific RS08 derivative.

Reset events force the MCU to immediately stop what it is doing and begin responding to reset. Any instruction that was in process will be aborted immediately without completing any remaining clock cycles. A short sequence of activities is completed to decide whether the source of reset was internal or external and to record the cause of reset. For the remainder of the time, the reset source remains active and the internal

clocks are stopped to save power. At the trailing edge of the reset event, the clocks resume and the CPU exits from the reset condition. The program counter is reset to $3FFD and an instruction fetch will be started after the release of reset.

For the device to execute code from the on-chip memory starting from $3FFD after reset, care should be taken to not force the BKDG pin low on the end of reset because this will force the device into active background mode where the CPU will wait for a command from the background communication interface.

2.4.2 Interrupts

The interrupt mechanism in RS08 is not used to interrupt the normal flow of instructions; it is used to wake up the RS08 from wait and stop modes. In run mode, interrupt events must be polled by the CPU. The interrupt feature is not compatible with Freescale's HC05, HC08, or HCS08 Families.

2.4.3 Wait and Stop Mode

Wait and stop modes are entered by executing a WAIT or STOP instruction, respectively. In these modes, the clocks to the CPU are shut down to save power and CPU activity is suspended. The CPU remains in this low-power state until an interrupt or reset event wakes it up. Please refer to the device specific documentation for the effects of wait and stop on other device peripherals.

2.4.4 Active Background Mode

Active background mode refers to the condition in which the CPU has stopped executing user program instructions and is waiting for serial commands from the background debug system. Refer to Section 3, "Development Support," for detailed information on active background mode.

2.5 Instruction Set Description by Instruction Types

In this section, the instruction is listed by type.

```
;This defines the labels used in the code examples
;
1000                                org      $1000
1000 0008    tinyaddr               equ      $08
1000 001a    shortaddr              equ      $1A
1000 008c    directaddr             equ      $8C
1000 005a    mask                   equ      $5a
```

Instruction Set Summary Nomenclature

The nomenclature listed here is used in the instruction descriptions in Table 2-12 through Table 2-13.

Operators

( )	=	Contents of register or memory location shown inside parentheses
←	=	Is loaded with (read: “gets”)
⇔	=	Exchange with
&	=	Boolean AND
|	=	Boolean OR
⊕	=	Boolean exclusive-OR
:	=	Concatenate
+	=	Add

CPU registers

A	=	Accumulator
CCR	=	Condition code register
PC	=	Program counter
PCH	=	Program counter, higher order (most significant) six bits
PCL	=	Program counter, lower order (least significant) eight bits
SPC	=	Shadow program counter
SPCH	=	Shadow program counter, higher order (most significant) six bits
SPCL	=	Shadow program counter, lower order (least significant) eight bits

Memory and addressing

M	=	A memory location or absolute data, depending on addressing mode
rel	=	The relative offset, which is the two's complement number stored in the last byte of machine code corresponding to a branch instruction
X	=	Pseudo index register, memory location $000F
,X or D[X]	=	Memory location $000E pointing to the memory location defined by the pseudo index register (location $000F)

Condition code register (CCR) bits

Z	=	Zero indicator
C	=	Carry/borrow

CCR activity notation

–	=	Bit not affected
0	=	Bit forced to 0
1	=	Bit forced to 1
↕	=	Bit set or cleared according to results of operation
U	=	Undefined after the operation

Machine coding notation

dd	=	Low-order eight bits of a direct address $0000–$00FF (high byte assumed to be $00)
ii	=	One byte of immediate data
hh	=	High-order 6-bit of 14-bit extended address prefixed with 2-bit of 0
ll	=	Low-order byte of 14-bit extended address
rr	=	Relative offset

Source form

Everything in the source forms columns, *except expressions in italic characters,* is literal information which must appear in the assembly source file exactly as shown. The initial 3- to 5-letter mnemonic is always a literal expression. All commas, pound signs (#), parentheses, and plus signs (+) are literal characters.

n — Any label or expression that evaluates to a single integer in the range 0–7.

opr8i — Any label or expression that evaluates to an 8-bit immediate value.

opr4a — Any label or expression that evaluates to a Tiny address (4-bit value). The instruction treats this 4-bit value as the low order four bits of an address in the 16-Kbyte address space ($0000–$000F). This 4-bit is embedded in the low order 4 bits in the opcode.

opr5a — Any label or expression that evaluates to a Short address (5-bit value). The instruction treats this 5-bit value as the low order five bits of an address in the 16-Kbyte address space ($0000–$001F). This 5-bit value is embedded in the low order 5 bits in the opcode.

opr8a — Any label or expression that evaluates to an 8-bit value. The instruction treats this 8-bit value as the low order eight bits of an address in the 16-Kbyte address space ($0000–$00FF).

opr16a — Any label or expression that evaluates to a 14-bit value. On the RS08 Family, the upper two bits are always 0s. The instruction treats this combined value as an address in the 16-Kbyte address space.

rel — Any label or expression that refers to an address that is within –128 to +127 locations from the next address after the last byte of object code for the current instruction. The assembler will calculate the 8-bit signed offset and include it in the object code for this instruction.

Address modes

INH = Inherent (no operands)
IMD = Immediate to Direct (in MOV instruction)
IMM = Immediate
DD = Direct to Direct (in MOV instruction)
DIR = Direct
SRT = Short
TNY = Tiny
EXT = Extended
REL = 8-bit relative offset

2.5.1 Data Movement Instructions

This group of instructions is used to move data between memory and CPU registers, or between memory locations. Load, store, and move

instructions automatically update the Z flag based on the value of the data. This allows conditional branching with BEQ and BNE immediately after a load, store, or move instruction without having to do a separate test or compare instruction.

2.5.1.1 Loads and Stores

Table 2-1. Load and Store Instructions

Source Form	Description	Operation	Effect on CCR		Address Mode	Opcode	Operand	Cycles
			Z	C				
LDA #*opr8i* LDA *opr8a* LDA *opr5a* LDA ,X [(1)]	Load Accumulator from Memory	A ← (M)	↕	–	IMM DIR SRT IX	A6 B6 C*x*/D*x* CE	ii dd	2 3 3 3
LDX #*opr8i* [(1)] LDX *opr8a* [(1)] LDX ,X [(1)]	Load Index Register from Memory	$0F ← (M)	↕	–	IMD DIR IX	3E 4E 4E	ii 0F dd 0F 0E 0F	4 5 5
STA *opr8a* STA *opr5a* STA ,X [(1)] STA X	Store Accumulator in Memory	M ← (A)	↕	–	DIR SRT IX SRT	B7 E*x* / F*x* EE EF	dd	3 2 2 2
STX *opr8a* [(1)]	Store Index Register in Memory	M ← (X)	↕	–	DIR	4E	0F dd	5
TAX[(1)]	Transfer A to X	X ← (A)	↕	–	INH	EF		2
TXA[(1)]	Transfer X to A	A ← (X)	↕	–	INH	CF		3

1. This is a pseudo instruction supported by the normal RS08 instruction set.

```
1000 a6 5a        lda     #mask          ;Immediate
1002 b6 8c        lda     directaddr     ;Direct address
1004 da           lda     shortaddr      ;Short address
1005 ce           lda     ,X             ;Indexed
1006 3e 5a 0f     ldx     #mask          ;MOVE Immediate,direct
1009 4e 8c 0f     ldx     directaddr     ;MOVE direct,direct
100c b7 8c        sta     directaddr     ;Direct address
100e fa           sta     shortaddr      ;Short address
100f ee           sta     ,X             ;Indexed
1010 4e 0f 8c     stx     directaddr     ;MOVE direct,direct
```

Table 2-2. BSET, BCLR, and Move Instructions

Source Form	Description	Operation	Effect on CCR Z	Effect on CCR C	Address Mode	Opcode	Operand	Cycles
BCLR *n,opr8a*	Clear Bit n in Memory	Mn ← 0	–	–	DIR (b0)	11	dd	5
					DIR (b1)	13	dd	5
					DIR (b2)	15	dd	5
					DIR (b3)	17	dd	5
					DIR (b4)	19	dd	5
					DIR (b5)	1B	dd	5
					DIR (b6)	1D	dd	5
					DIR (b7)	1F	dd	5
BCLR *n*,D[X]					IX (b0)	11	0E	5
					IX (b1)	13	0E	5
					IX (b2)	15	0E	5
					IX (b3)	17	0E	5
					IX (b4)	19	0E	5
					IX (b5)	1B	0E	5
					IX (b6)	1D	0E	5
					IX (b7)	1F	0E	5
BCLR n,X					DIR (b0)	11	0F	5
					DIR (b1)	13	0F	5
					DIR (b2)	15	0F	5
					DIR (b3)	17	0F	5
					DIR (b4)	19	0F	5
					DIR (b5)	1B	0F	5
					DIR (b6)	1D	0F	5
					DIR (b7)	1F	0F	5
BSET *n,opr8a*	Set Bit *n* in Memory	Mn ← 1	–	–	DIR (b0)	10	dd	5
					DIR (b1)	12	dd	5
					DIR (b2)	14	dd	5
					DIR (b3)	16	dd	5
					DIR (b4)	18	dd	5
					DIR (b5)	1A	dd	5
					DIR (b6)	1C	dd	5
					DIR (b7)	1E	dd	5
BSET *n*,D[X]					IX (b0)	10	0E	5
					IX (b1)	12	0E	5
					IX (b2)	14	0E	5
					IX (b3)	16	0E	5
					IX (b4)	18	0E	5
					IX (b5)	1A	0E	5
					IX (b6)	1C	0E	5
					IX (b7)	1E	0E	5
BSET *n*,X					DIR (b0)	10	0F	5
					DIR (b1)	12	0F	5
					DIR (b2)	14	0F	5
					DIR (b3)	16	0F	5
					DIR (b4)	18	0F	5
					DIR (b5)	1A	0F	5
					DIR (b6)	1C	0F	5
					DIR (b7)	1E	0F	5
MOV *opr8a,opr8a*	Move	$(M)_{destination} \leftarrow (M)_{source}$	↕	–	DD	4E	dd dd	5
MOV *#opr8i,opr8a*					IMD	3E	ii dd	4
MOV D[X],opr8a					IX/DIR	4E	0E dd	5
MOV opr8a,D[X]					DIR/IX	4E	dd 0E	5
MOV #opr8i,D[X]					IMM/IX	3E	ii 0E	4

2.5.1.2 Bit Set and Bit Clear

```
1013 11 8c          bclr    0,directaddr        ;Direct address
1015 1f 0f          bclr    7,X                 ;Indexed
1017 10 08          bset    0,tinyaddr          ;Direct address
1019 1e 0f          bset    7,X                 ;Indexed
```

2.5.1.3 Memory-to-Memory Moves

```
101b 3e 5a 1a     mov     #mask,shortaddr      ;MOVE Immediate,direct
101e 4e 08 1a     mov     tinyaddr,shortaddr   ;MOVE direct,direct
```

2.5.2 Math Instructions

Math instructions include the traditional add and subtract operations, a collection of utility instructions including increment, decrement, clear, and compare. The compare instruction is actually subtract operation where the CCR bits are affected but the result is not written back to a CPU register.

2.5.2.1 Add, and Subtract

Table 2-3. Add, and Subtract Instructions

Source Form	Description	Operation	Effect on CCR		Address Mode	Opcode	Operand	Cycles
			Z	C				
ADC #*opr8i* ADC *opr8a* ADC ,X [(1)] ADC X	Add with Carry	A ← (A) + (M) + (C) A ← (A) + (X) + (C)	↕	↕	IMM DIR IX DIR	A9 B9 B9 B9	ii dd 0E 0F	2 3 3 3
ADD #*opr8i* ADD *opr8a* ADD *opr4a* ADD ,X [(1)] ADD X	Add without Carry	A ← (A) + (M)	↕	↕	IMMDIR TNY IX DIR	AB BB 6*x* 6E 6F	ii dd	2 3 3 3 3
SBC #*opr8i* SBC *opr8a* SBC ,X [(1)] SBC X	Subtract with Carry	A ← (A) – (M) – (C) A ← (A) – (X) – (C)	↕	↕	IMM DIR IX DIR	A2 B2 B2 B2	ii dd 0E 0F	2 3 3 3
SUB #*opr8i* SUB *opr8a* SUB *opr4a* SUB ,X [(1)] SUB X	Subtract	A ← (A) – (M) A ← (A) – (X)	↕	↕	IMM DIR TNY IX DIR	A0 B0 7*x* 7E 7F	ii dd	2 3 3 3 3

1. This is a pseudo instruction supported by the normal RS08 instruction set.

```
1021 a9 5a            adc      #mask           ;Immediate
1023 b9 8c            adc      directaddr      ;Direct address
1025 b9 0e            adc      ,X              ;Indexed
1029 ab 5a            add      #mask           ;Immediate
102b bb 8c            add      directaddr      ;Direct address
102d 68               add      tinyaddr        ;Tiny address
102e 6e               add      ,X              ;Indexed
1030 a2 5a            sbc      #mask           ;Immediate
1032 b2 8c            sbc      directaddr      ;Direct address
1034 b2 0e            sbc      ,X              ;Indexed
1038 a0 5a            sub      #mask           ;Immediate
103a b0 8c            sub      directaddr      ;Direct address
103c 78               sub      tinyaddr        ;Tiny address
103d 7e               sub      ,X              ;Indexed
```

Table 2-4. Other Arithmetic Instructions

Source Form	Description	Operation	Effect on CCR		Address Mode	Opcode	Operand	Cycles
			Z	C				
INC *opr8a* INC *opr4a* INC ,X [(1)] INCA INCX [(1)]	Increment	M ← (M) + $01 A ← (A) + $01 X ← (X) + $01	↕	–	DIR TNY IX INH INH	3C 2*x* 2E 4C 2F	dd	5 4 4 1 4
DEC *opr8a* DEC *opr4a* DEC ,X [(1)] DECA DECX [(1)]	Decrement	M ← (M) – $01 A ← (A) – $01 X ← (X) – $01	↕	–	DIR TNY IX INH INH	3A 5*x* 5E 4A 5F	dd	5 4 4 1 4
CLR *opr8a* CLR *opr5a* CLR ,X [(1)] CLRA CLRX [(1)]	Clear	M ← $00 A ← $00 X ← $00	1	–	DIR SRT IX INH INH	3F 8*x* / 9*x* 8E 4F 8F	dd	3 2 2 1 2
CMP #*opr8i* CMP *opr8a* CMP ,X [(1)] CMP X [(1)]	Compare Accumulator with Memory	(A) – (M) (A) – (X)	↕	↕	IMM DIR IX INH	A1 B1 B1 B1	ii dd 0E 0F	2 3 3 3
TST opr8a [(1)] TST ,X [(1)] TSTA [(1)] TSTX [(1)]	Test for Zero	(M) – $00 (A) – $00 (X) – $00	↕	–	DD IX INH INH	4E 4E AA 4E	dd dd 0E 0E 00 0F 0F	5 5 2 5

1. This is a pseudo instruction supported by the normal RS08 instruction set.

2.5.2.2 Increment, Decrement, and Clear

```
103f 3c 8c        inc     directaddr        ;Direct address
1041 28           inc     tinyaddr          ;Tiny address
1042 2e           inc     ,X                ;Indexed
1043 4c           inca                      ;Inherent
1044 2f           incx                      ;Inherent
1045 3a 8c        dec     directaddr        ;Direct address
1047 58           dec     tinyaddr          ;Tiny address
1048 5e           dec     ,X                ;Indexed
1049 4a           deca                      ;Inherent
104a 5f           decx                      ;Inherent
104b 3f 8c        clr     directaddr        ;Direct address
104d 9a           clr     shortaddr         ;Short address
104e 8e           clr     ,X                ;Indexed
104f 4f           clra                      ;Inherent
1050 8f           clrx                      ;Inherent
```

2.5.2.3 Compare

```
1051 a1 5a        cmp     #mask             ;Immediate
1053 b1 8c        cmp     directaddr        ;Direct address
1055 b1 0e        cmp     ,X                ;Indexed
1057 b1 0f        cmpx                      ;Inherent
```

2.5.3 Logical Operation Instructions

These instructions perform eight bitwise Boolean operations in parallel. For the complement instruction, each bit of the register operand is inverted. The other logical instructions involve two operands, one in the accumulator (A) and the other in memory. Immediate, direct, or indexed addressing modes may be used to access the memory operand. Each bit of the accumulator is ANDed, ORed, or exclusive-ORed with the corresponding bit of the memory operand. The result of the logical operation is stored into the accumulator, overwriting the original operand.

Table 2-5. Logical Operation Instructions

Source Form	Description	Operation	Effect on CCR Z	Effect on CCR C	Address Mode	Opcode	Operand	Cycles
AND #*opr8i* AND *opr8a* AND ,X [1] AND X	Logical AND	A ← (A) & (M) A ← (A) & (X)	↕	–	IMM DIR IX DIR	A4 B4 B4 B4	ii dd 0E 0F	2 3 3 3
ORA #*opr8i* ORA *opr8a* ORA ,X [1] ORA X	Inclusive OR Accumulator and Memory	A ← (A) | (M) A ← (A) | (X)	↕	–	IMM DIR IX DIR	AA BA BA BA	ii dd 0E 0F	2 3 3 3
EOR #*opr8i* EOR *opr8a* EOR ,X [1] EOR X	Exclusive OR Memory with Accumulator	A ← (A ⊕ M) A ← (A ⊕ X)	↕	–	IMM DIR IX DIR	A8 B8 B8 B8	ii dd 0E 0F	2 3 3 3
COMA	Complement (One's Complement)	A ← ($\overline{A}$)	↕	1	INH	43		1

1. This is a pseudo instruction supported by the normal RS08 instruction set.

2.5.3.1 AND, OR, Exclusive-OR, and Complement

```
1059 a4 5a          and      #mask            ;Immediate
105b b4 8c          and      directaddr       ;Direct address
105d b4 0e          and      ,X               ;Indexed
1061 aa 5a          ora      #mask            ;Immediate
1063 ba 8c          ora      directaddr       ;Direct address
1065 ba 0e          ora      ,X               ;Indexed
1069 a8 5a          eor      #mask            ;Immediate
106b b8 8c          eor      directaddr       ;Direct address
106d b8 0e          eor      ,X               ;Indexed
1071 43             coma                      ;Inherent
```

2.5.4 Shift and Rotate Instructions

All of the shift and rotate instructions operate on a 9-bit field consisting of an 8-bit value in A and the C bit in the CCR. Drawings are provided in the instruction descriptions to show where the C bit fits into the shift or rotate operation. The logical shift instructions are simple shifts which shift a zero into the first bit of the value and shift the last bit into the carry bit.

The arithmetic left shift pseudo instruction is also available because its operation is identical to logical shift left.

Table 2-6. Shift and Rotate Instructions

Source Form	Description	Operation	Effect on CCR		Address Mode	Opcode	Operand	Cycles
			Z	C				
ASLA[1]	Arithmetic Left Shift	C ← b7 … b0 ← 0	↕	↕	INH	48		1
LSLA	Logical Shift Left	C ← b7 … b0 ← 0	↕	↕	INH	48		1
LSRA	Logical Shift Right	0 → b7 … b0 → C	↕	↕	INH	44		1
ROLA	Rotate Left through Carry	C ← b7 … b0 ← C	↕	↕	INH	49		1
RORA	Rotate Right through Carry	C → b7 … b0 → C	↕	↕	INH	46		1

1. This is a pseudo instruction supported by the normal RS08 instruction set.

```
1072 48          lsla          ;Inherent
1073 44          lsra          ;Inherent
1074 49          rola          ;Inherent
1075 46          rora          ;Inherent
```

2.5.5 Jump, Branch, and Loop Control Instructions

The instructions in this group cause a change of flow which means that the CPU loads a new address into the program counter so program

execution continues at a location other than the next memory location after the current instruction.

Jump instructions cause an unconditional change in the execution sequence to a new location in a program. Branch and loop control instructions cause a conditional change in the execution sequence. Branch and loop control instructions use relative addressing mode to conditionally branch to a location that is relative to the location of the branch. Processor status indicators in the CCR control whether a conditional branch or loop control instruction will branch to a new address or simply continue to the next instruction in the program. BRA is a special case because the branch always occurs, and BRN is special because the branch never taken (this is functionally equivalent to a 2-byte, 3-cycle NOP).

Table 2-7. Jump and Branch Instructions

Source Form	Description	Operation	Effect on CCR		Address Mode	Opcode	Operand	Cycles
			Z	C				
JMP *opr16a*	Jump	PC ← Effective Address	–	–	EXT	BC	hh ll	4
BCC *rel*	Branch if Carry Bit Clear	PC ← (PC) + $0002 + rel, if (C) = 0	–	–	REL	34	rr	3
BCS *rel*	Branch if Carry Bit Set (Same as BLO)	PC ← (PC) + $0002 + *rel*, if (C) = 1	–	–	REL	35	rr	3
BEQ *rel*	Branch if Equal	PC ← (PC) + $0002 + *rel*, if (Z) = 1	–	–	REL	37	rr	3
BHS *rel* [1]	Branch if Higher or Same (Same as BCC)	PC ← (PC) + $0002 + rel, if (C) = 0	–	–	REL	34	rr	3
BLO *rel* [1]	Branch if Lower (Same as BCS)	PC ← (PC) + $0002 + *rel*, if (C) = 1	–	–	REL	35	rr	3
BNE *rel*	Branch if Not Equal	PC ← (PC) + $0002 + *rel*, if (Z) = 0	–	–	REL	36	rr	3
BRA *rel*	Branch Always	PC ← (PC) + $0002 + *rel*	–	–	REL	30	rr	3
BRN *rel* [1]	Branch Never	PC ← (PC) + $0002	–	–	REL	30	00	3

. This is a pseudo instruction supported by the normal RS08 instruction set.

2.5.5.1 Unconditional Jump and Branch

Jump (JMP), and branch always (BRA) are unconditional and do not depend on the state of any CCR bits. Jump may be used to go to any memory location in the 16-Kbyte address space while branch instructions are limited to destinations within –128 to +127 locations from the address immediately after the branch offset byte.

```
1076 bc 10 b5            jmp      extended        ;Extended address
1079 30 fe               bra      *               ;Relative
```

2.5.5.2 Simple Branches

The simple branches depend only on the state of a single condition bit in the CCR.

Table 2-8. Simple Branch Summary

Branch Condition	Branch if True	Branch if False
Z	BEQ	BNE
C	BCS	BCC

```
107b 37 fe                  beq     *                  ;Relative
107d 36 fe                  bne     *                  ;Relative
107f 34 fe                  bcc     *                  ;Relative
1081 35 fe                  bcs     *                  ;Relative
```

2.5.5.3 Unsigned Branches

Branch if lower (BLO), and branch if higher or same (BHS) are used after operations involving unsigned numbers. The simple branches, branch if equal (BEQ) and branch if not equal (BNE), can also be used after operations involving unsigned numbers.

```
1083 34 fe                  bhs     *                  ;Relative
1085 35 fe                  blo     *                  ;Relative
```

Table 2-9. Bit Branches and Loop Control Instructions

Source Form	Description	Operation	Effect on CCR Z	Effect on CCR C	Address Mode	Opcode	Operand	Cycles
BRCLR *n,opr8a,rel* BRCLR *n*,D[X],*rel*[1] BRCLR *n*,X,*rel*[1]	Branch if Bit *n* in Memory Clear	PC ← (PC) + $0003 + *rel*, if (Mn) = 0	–	↕	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7) IX (b0) IX (b1) IX (b2) IX (b3) IX (b4) IX (b5) IX (b6) IX (b7) DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	01 03 05 07 09 0B 0D 0F 01 03 05 07 09 0B 0D 0F 01 03 05 07 09 0B 0D 0F	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr 0E rr 0E rr 0E rr 0E rr 0E rr 0E rr 0E rr 0E rr 0F rr 0F rr 0F rr 0F rr 0F rr 0F rr 0F rr 0F rr	5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
BRSET *n,opr8a,rel* BRSET *n*,D[X],*rel*[1] BRSET *n*,X,*rel*[1]	Branch if Bit *n* in Memory Set	PC ← (PC) + $0003 + *rel*, if (Mn) = 1	–	↕	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7) IX (b0) IX (b1) IX (b2) IX (b3) IX (b4) IX (b5) IX (b6) IX (b7) DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	00 02 04 06 08 0A 0C 0E 00 02 04 06 08 0A 0C 0E 00 02 04 06 08 0A 0C 0E	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr 0E rr 0E rr 0E rr 0E rr 0E rr 0E rr 0E rr 0E rr 0F rr 0F rr 0F rr 0F rr 0F rr 0F rr 0F rr 0F rr	5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
CBEQA #*opr8i,rel* CBEQ *opr8a,rel* CBEQ ,X,*rel*[1],[2] CBEQ X,*rel*[1]	Compare and Branch if Equal	PC ← (PC) + $0003 + *rel*, if (A) – (M) = $00 PC ← (PC) + $0003 + *rel*, if (A) – (M) = $00 PC ← (PC) + $0003 + *rel*, if (A) – (X) = $00	–	–	IMM DIR IX DIR	41 31 31 31	ii rr dd rr 0E rr 0F rr	4 5 5 5
DBNZ *opr8a,rel* DBNZ ,X,*rel*[1] DBNZA *rel* DBNZX *rel*	Decrement and Branch if Not Zero	A ← (A) – $01 or M ← (M) - $01 PC ← (PC) + $0003 + *rel* if (result) ≠ 0 for DBNZ direct PC ← (PC) + $0002 + *rel* if (result) ≠ 0 for DBNZA X ← (X) – $01 PC ← (PC) + $0003 + *rel* if (result) ≠ 0	–	–	DIR IX INH INH	3B 3B 4B 3B	dd rr 0E rr rr 0F rr	7 7 4 7

1. This is a pseudo instruction supported by the normal RS08 instruction set.

2.5.5.4 Bit Condition Branches

These branch instructions test a single bit in a memory operand in direct addressing space ($0000–$00FF) and BRSET branches if the tested bit is set while BRCLR branches if the bit was clear. Although this seems like a limited number of locations, it includes all of the I/O and control register space and a significant portion of the RAM where program variables may be located. By having separate opcodes for each bit position, these instructions are particularly efficient, requiring only three bytes of object code and five bus cycles.

```
1087 01 08 fd          brclr    0,tinyaddr,*       ;Direct address
108a 0f 0e fd          brclr    7,D[X],*           ;Indexed
108d 00 08 fd          brset    0,tinyaddr,*       ;Direct address
1090 0e 0e fd          brset    7,D[X],*           ;Indexed
```

2.5.5.5 Loop Control

The CBEQ instructions compare the contents of the accumulator to a memory location and branch if they are equal to each other. CBEQA allows A to be compared against an immediate operand.

The DBNZ instructions decrement A or a memory location and then branch if the decremented value is still not zero. This provides an efficient way to implement a loop counter.

```
1093 31 08 fd          cbeq     tinyaddr,*         ;Direct address
1096 31 0e fd          cbeq     ,X,*               ;Indexed
1099 31 0e fd          cbeq     [X],*              ;Indexed
109c 41 5a fd          cbeqa    #mask,*            ;Immediate
109f 3b 08 fd          dbnz     tinyaddr,*         ;Direct address
10a5 4b fe             dbnza    *                  ;Relative
```

2.5.6 Subroutine-Related Instructions

Jump-to-subroutine (JSR) and branch-to-subroutine (BSR) instructions are used to go to a sequence of instructions (a subroutine) somewhere else in a program. Normally, at the end of the subroutine, a return-from-subroutine (RTS) instruction causes the CPU to return to the next instruction after the JSR or BSR that called the subroutine.

Table 2-10. Subroutine-Related Instructions

Source Form	Description	Operation	Effect on CCR Z	Effect on CCR C	Address Mode	Opcode	Operand	Cycles
BSR *rel*	Branch Subroutine	PC ← (PC) + 2 Push PC to shadow PC PC ← (PC) + *rel*	–	–	REL	AD	rr	3
JSR *opr16a*	Jump to Subroutine	PC ← (PC) + 3 Push PC to shadow PC PC ← Effective Address	–	–	EXT	BD	hh ll	4
RTS	Return from Subroutine	Pull PC from shadow PC	–	–	INH	BE		3
SHA	Swap Shadow PC High with A	A ⇔ SPCH	–	–	INH	45		1
SLA	Swap Shadow PC Low with A	A ⇔ SPCL	–	–	INH	42		1

```
10a7 ad fe          bsr     *                 ;Relative
10a9 bd 10 b5       jsr     extended          ;Extended address
10ac be             rts                       ;Inherent
10ad 45             sha                       ;Inherent
10ae 42             sla                       ;Inherent
```

2.5.7 Miscellaneous Instructions

Table 2-11. Miscellaneous Instructions

Source Form	Description	Operation	Effect on CCR Z	Effect on CCR C	Address Mode	Opcode	Operand	Cycles
NOP	No Operation	None	–	–	INH	AC		1
SEC	Set Carry Bit	C ← 1	–	1	INH	39		1
CLC	Clear Carry Bit	C ← 0	–	0	INH	38		1
BGND	Background	Enter Background Debug Mode	–	–	INH	BF		5+
WAIT	Put MCU into wait mode		–	–	INH	AF		2+
STOP	Put MCU into stop mode		–	–	INH	AE		2+

```
10af ac                 nop                     ;Inherent
10b0 39                 sec                     ;Inherent
10b1 38                 clc                     ;Inherent
10b2 bf                 bgnd                    ;Inherent
10b3 af                 wait                    ;Inherent
10b4 ae                 stop                    ;Inherent
```

2.6 Summary Instruction Table

Instruction Set Summary Nomenclature

The nomenclature listed here is used in the instruction descriptions in Table 2-12 through Table 2-13.

Operators

( )	=	Contents of register or memory location shown inside parentheses
←	=	Is loaded with (read: “gets”)
⇔	=	Exchange with
&	=	Boolean AND
|	=	Boolean OR
⊕	=	Boolean exclusive-OR
:	=	Concatenate
+	=	Add

CPU registers

A	=	Accumulator
CCR	=	Condition code register
PC	=	Program counter
PCH	=	Program counter, higher order (most significant) six bits
PCL	=	Program counter, lower order (least significant) eight bits
SPC	=	Shadow program counter
SPCH	=	Shadow program counter, higher order (most significant) six bits
SPCL	=	Shadow program counter, lower order (least significant) eight bits

Memory and addressing

M	=	A memory location or absolute data, depending on addressing mode
rel	=	The relative offset, which is the two’s complement number stored in the last byte of machine code corresponding to a branch instruction

X = Pseudo index register, memory location $000F
,X or D[X] = Memory location $000E pointing to the memory location defined by the pseudo index register (location $000F)

Condition code register (CCR) bits

Z = Zero indicator
C = Carry/borrow

CCR activity notation

– = Bit not affected
0 = Bit forced to 0
1 = Bit forced to 1
↕ = Bit set or cleared according to results of operation
U = Undefined after the operation

Machine coding notation

dd = Low-order eight bits of a direct address $0000–$00FF (high byte assumed to be $00)
ii = One byte of immediate data
hh = High-order 6-bit of 14-bit extended address prefixed with 2-bit of 0
ll = Low-order byte of 14-bit extended address
rr = Relative offset

Source form

Everything in the source forms columns, *except expressions in italic characters,* is literal information which must appear in the assembly source file exactly as shown. The initial 3- to 5-letter mnemonic is always a literal expression. All commas, pound signs (#), parentheses, and plus signs (+) are literal characters.

n — Any label or expression that evaluates to a single integer in the range 0–7.
x — Any label or expression that evaluates to a single hexadecimal integer in the range $0–$F.
opr8i — Any label or expression that evaluates to an 8-bit immediate value.

opr4a — Any label or expression that evaluates to a Tiny address (4-bit value). The instruction treats this 4-bit value as the low order four bits of an address in the 16-Kbyte address space ($0000–$000F). This 4-bit value is embedded in the low order four bits in the opcode.

opr5a — Any label or expression that evaluates to a Short address (5-bit value). The instruction treats this 5-bit value as the low order five bits of an address in the 16-Kbyte address space ($0000–$001F). This 5-bit value is embedded in the low order 5 bits in the opcode.

opr8a — Any label or expression that evaluates to an 8-bit value. The instruction treats this 8-bit value as the low order eight bits of an address in the 16-Kbyte address space ($0000–$00FF).

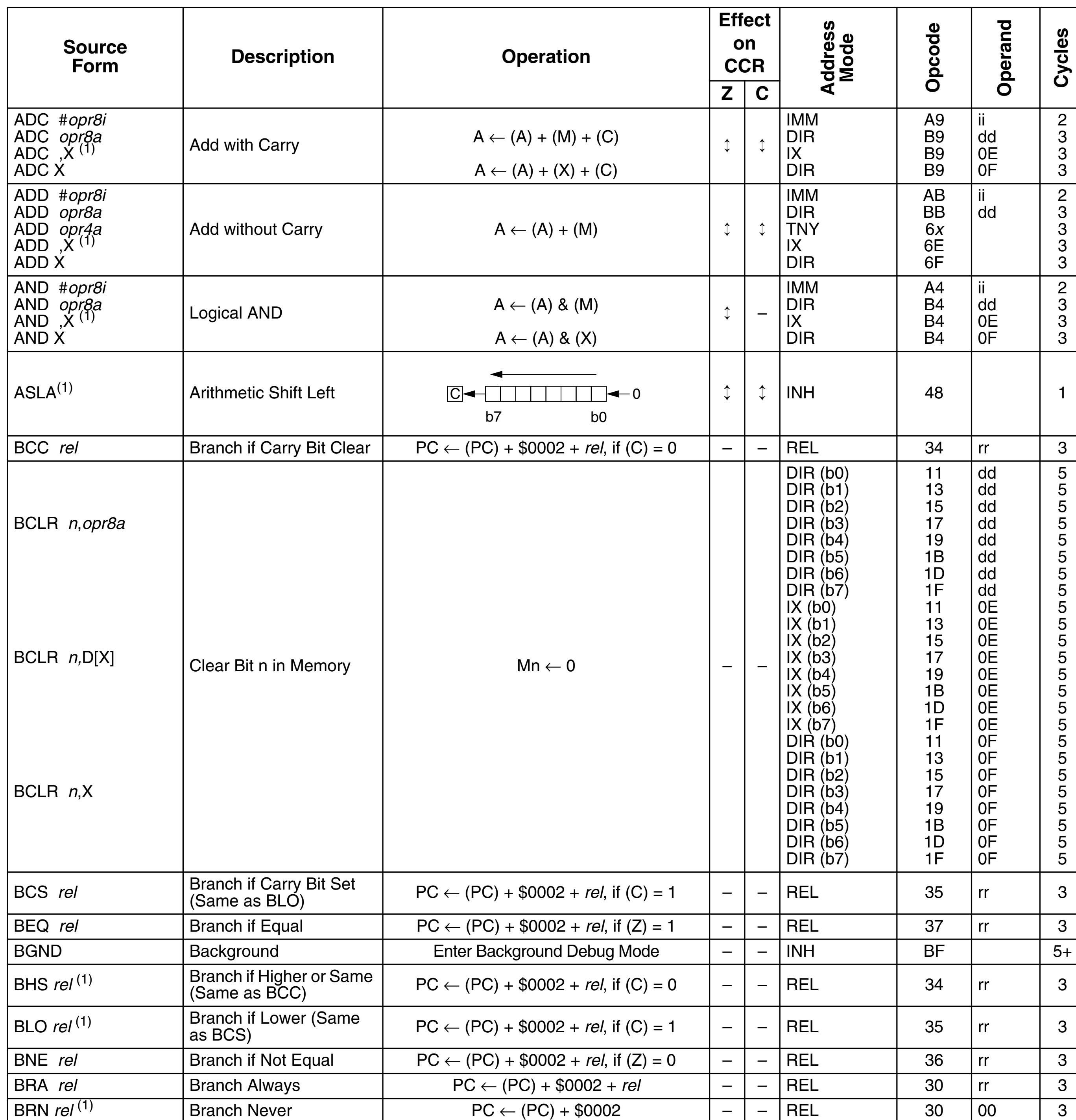
opr16a — Any label or expression that evaluates to a 14-bit value. On the RS08 core, the upper two bits are always 0s. The instruction treats this value as an address in the 16-Kbyte address space.

rel — Any label or expression that refers to an address that is within –128 to +127 locations from the next address after the last byte of object code for the current instruction. The assembler will calculate the 8-bit signed offset and include it in the object code for this instruction.

Address modes

INH = Inherent (no operands)
IMD = Immediate to Direct (in MOV instruction)
IMM = Immediate
DD = Direct to Direct (in MOV instruction)
DIR = Direct
SRT = Short
TNY = Tiny
EXT = Extended
REL = 8-bit relative offset

Table 2-12. Instruction Set Summary (Sheet 1 of 5)



Source Form	Description	Operation	Effect on CCR Z	Effect on CCR C	Address Mode	Opcode	Operand	Cycles
ADC #*opr8i* ADC *opr8a* ADC ,X (1) ADC X	Add with Carry	A ← (A) + (M) + (C) A ← (A) + (X) + (C)	↕	↕	IMM DIR IX DIR	A9 B9 B9 B9	ii dd 0E 0F	2 3 3 3
ADD #*opr8i* ADD *opr8a* ADD *opr4a* ADD ,X (1) ADD X	Add without Carry	A ← (A) + (M)	↕	↕	IMM DIR TNY IX DIR	AB BB 6*x* 6E 6F	ii dd	2 3 3 3 3
AND #*opr8i* AND *opr8a* AND ,X (1) AND X	Logical AND	A ← (A) & (M) A ← (A) & (X)	↕	–	IMM DIR IX DIR	A4 B4 B4 B4	ii dd 0E 0F	2 3 3 3
ASLA(1)	Arithmetic Shift Left	C ← b7 … b0 ← 0	↕	↕	INH	48		1
BCC *rel*	Branch if Carry Bit Clear	PC ← (PC) + $0002 + *rel*, if (C) = 0	–	–	REL	34	rr	3
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 13 15 17 19 1B 1D 1F	dd dd dd dd dd dd dd dd	5 5 5 5 5 5 5 5
BCLR *n*,D[X]					IX (b0) IX (b1) IX (b2) IX (b3) IX (b4) IX (b5) IX (b6) IX (b7)	11 13 15 17 19 1B 1D 1F	0E 0E 0E 0E 0E 0E 0E 0E	5 5 5 5 5 5 5 5
BCLR *n*,X					DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	11 13 15 17 19 1B 1D 1F	0F 0F 0F 0F 0F 0F 0F 0F	5 5 5 5 5 5 5 5
BCS *rel*	Branch if Carry Bit Set (Same as BLO)	PC ← (PC) + $0002 + *rel*, if (C) = 1	–	–	REL	35	rr	3
BEQ *rel*	Branch if Equal	PC ← (PC) + $0002 + *rel*, if (Z) = 1	–	–	REL	37	rr	3
BGND	Background	Enter Background Debug Mode	–	–	INH	BF		5+
BHS *rel* (1)	Branch if Higher or Same (Same as BCC)	PC ← (PC) + $0002 + *rel*, if (C) = 0	–	–	REL	34	rr	3
BLO *rel* (1)	Branch if Lower (Same as BCS)	PC ← (PC) + $0002 + *rel*, if (C) = 1	–	–	REL	35	rr	3
BNE *rel*	Branch if Not Equal	PC ← (PC) + $0002 + *rel*, if (Z) = 0	–	–	REL	36	rr	3
BRA *rel*	Branch Always	PC ← (PC) + $0002 + *rel*	–	–	REL	30	rr	3
BRN *rel* (1)	Branch Never	PC ← (PC) + $0002	–	–	REL	30	00	3

1. This is a pseudo instruction supported by the normal RS08 instruction set.
2. This instruction is different from that of the HC08 and HCS08 in that the RS08 does not auto-increment the index register.

Table 2-12. Instruction Set Summary (Sheet 2 of 5)

Source Form	Description	Operation	Effect on CCR Z	Effect on CCR C	Address Mode	Opcode	Operand	Cycles
BRCLR *n*,*opr8a*,*rel*	Branch if Bit *n* in Memory Clear	PC ← (PC) + $0003 + *rel*, if (Mn) = 0	–	↕	DIR (b0)	01	dd rr	5
					DIR (b1)	03	dd rr	5
					DIR (b2)	05	dd rr	5
					DIR (b3)	07	dd rr	5
					DIR (b4)	09	dd rr	5
					DIR (b5)	0B	dd rr	5
					DIR (b6)	0D	dd rr	5
					DIR (b7)	0F	dd rr	5
BRCLR *n*,D[X],*rel*					IX (b0)	01	0E rr	5
					IX (b1)	03	0E rr	5
					IX (b2)	05	0E rr	5
					IX (b3)	07	0E rr	5
					IX (b4)	09	0E rr	5
					IX (b5)	0B	0E rr	5
					IX (b6)	0D	0E rr	5
					IX (b7)	0F	0E rr	5
BRCLR *n*,X,*rel*					DIR (b0)	01	0F rr	5
					DIR (b1)	03	0F rr	5
					DIR (b2)	05	0F rr	5
					DIR (b3)	07	0F rr	5
					DIR (b4)	09	0F rr	5
					DIR (b5)	0B	0F rr	5
					DIR (b6)	0D	0F rr	5
					DIR (b7)	0F	0F rr	5
BRSET *n*,*opr8a*,*rel*	Branch if Bit *n* in Memory Set	PC ← (PC) + $0003 + *rel*, if (Mn) = 1	–	↕	DIR (b0)	00	dd rr	5
					DIR (b1)	02	dd rr	5
					DIR (b2)	04	dd rr	5
					DIR (b3)	06	dd rr	5
					DIR (b4)	08	dd rr	5
					DIR (b5)	0A	dd rr	5
					DIR (b6)	0C	dd rr	5
					DIR (b7)	0E	dd rr	5
BRSET *n*,D[X],*rel*					IX (b0)	00	0E rr	5
					IX (b1)	02	0E rr	5
					IX (b2)	04	0E rr	5
					IX (b3)	06	0E rr	5
					IX (b4)	08	0E rr	5
					IX (b5)	0A	0E rr	5
					IX (b6)	0C	0E rr	5
					IX (b7)	0E	0E rr	5
BRSET *n*,X,*rel*					DIR (b0)	00	0F rr	5
					DIR (b1)	02	0F rr	5
					DIR (b2)	04	0F rr	5
					DIR (b3)	06	0F rr	5
					DIR (b4)	08	0F rr	5
					DIR (b5)	0A	0F rr	5
					DIR (b6)	0C	0F rr	5
					DIR (b7)	0E	0F rr	5

1. This is a pseudo instruction supported by the normal RS08 instruction set.
2. This instruction is different from that of the HC08 and HCS08 in that the RS08 does not auto-increment the index register.

Table 2-12. Instruction Set Summary (Sheet 3 of 5)

Source Form	Description	Operation	Effect on CCR Z	Effect on CCR C	Address Mode	Opcode	Operand	Cycles
BSET *n,opr8a* BSET *n*,D[X] BSET *n*,X	Set Bit *n* in Memory	Mn ← 1	–	–	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7) IX (b0) IX (b1) IX (b2) IX (b3) IX (b4) IX (b5) IX (b6) IX (b7) DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	10 12 14 16 18 1A 1C 1E 10 12 14 16 18 1A 1C 1E 10 12 14 16 18 1A 1C 1E	dd dd dd dd dd dd dd dd 0E 0E 0E 0E 0E 0E 0E 0E 0F 0F 0F 0F 0F 0F 0F 0F	5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
BSR *rel*	Branch Subroutine	PC ← (PC) + 2 Push PC to shadow PC PC ← (PC) + *rel*	–	–	REL	AD	rr	3
CBEQA *#opr8i,rel* CBEQ *opr8a,rel* CBEQ *,X,rel* [1],[2] CBEQ X,*rel* [1]	Compare and Branch if Equal	PC ← (PC) + $0003 + *rel*, if (A) – (M) = $00 PC ← (PC) + $0003 + *rel*, if (A) – (M) = $00 PC ← (PC) + $0003 + *rel*, if (A) – (X) = $00	–	–	IMM DIR IX DIR	41 31 31 31	ii rr dd rr 0E rr 0F rr	4 5 5 5
CLC	Clear Carry Bit	C ← 0	–	0	INH	38		1
CLR *opr8a* CLR *opr5a* CLR ,X [1] CLRA CLRX [1]	Clear	M ← $00 A ← $00 X ← $00	1	–	DIR SRT IX INH INH	3F 8*x* / 9*x* 8E 4F 8F	dd	3 2 2 1 2
CMP *#opr8i* CMP *opr8a* CMP ,X [1] CMP X [1]	Compare Accumulator with Memory	(A) – (M) (A) – (X)	↕	↕	IMM DIR IX INH	A1 B1 B1 B1	ii dd 0E 0F	2 3 3 3
COMA	Complement (One's Complement)	$A \leftarrow (\overline{A})$	↕	1	INH	43		1
DBNZ *opr8a,rel* DBNZ ,X,rel [1] DBNZA *rel* DBNZX rel [1]	Decrement and Branch if Not Zero	A ← (A) – $01 or M ← (M) - $01 PC ← (PC) + $0003 + *rel* if (result) ≠ 0 for DBNZ direct PC ← (PC) + $0002 + *rel* if (result) ≠ 0 for DBNZA X ← (X) – $01 PC ← (PC) + $0003 + *rel* if (result) ≠ 0	–	–	DIR IX INH INH	3B 3B 4B 3B	dd rr 0E rr rr 0F rr	7 7 4 7
DEC *opr8a* DEC *opr4a* DEC ,X [1] DECA DEC X	Decrement	M ← (M) – $01 A ← (A) – $01 X ← (X) – $01	↕	–	DIR TNY IX INH DIR	3A 5*x* 5E 4A 5F	dd	5 4 4 1 4
EOR *#opr8i* EOR *opr8a* EOR ,X [1] EOR X	Exclusive OR Memory with Accumulator	A ← (A ⊕ M) A ← (A ⊕ X)	↕	–	IMM DIR IX DIR	A8 B8 B8 B8	ii dd 0E 0F	2 3 3 3

1. This is a pseudo instruction supported by the normal RS08 instruction set.
2. This instruction is different from that of the HC08 and HCS08 in that the RS08 does not auto-increment the index register.

Table 2-12. Instruction Set Summary (Sheet 4 of 5)

Source Form	Description	Operation	Effect on CCR Z	Effect on CCR C	Address Mode	Opcode	Operand	Cycles
INC *opr8a* INC *opr4a* INC ,X [(1)] INCA INCX [(1)]	Increment	M ← (M) + $01 A ← (A) + $01 X ← (X) + $01	↕	–	DIR TNY IX INH INH	3C 2*x* 2E 4C 2F	dd	5 4 4 1 4
JMP *opr16a*	Jump	PC ← Effective Address	–	–	EXT	BC	hh ll	4
JSR *opr16a*	Jump to Subroutine	PC ← (PC) + 3 Push PC to shadow PC PC ← Effective Address	–	–	EXT	BD	hh ll	4
LDA #*opr8i* LDA *opr8a* LDA *opr5a* LDA ,X [(1)]	Load Accumulator from Memory	A ← (M)	↕	–	IMM DIR SRT IX	A6 B6 C*x*/D*x* CE	ii dd	2 3 3 3
LDX #*opr8i* [(1)] LDX *opr8a* [(1)] LDX ,X [(1)]	Load Index Register from Memory	$0F ← (M)	↕	–	IMD DIR IX	3E 4E 4E	ii 0F dd 0F 0E 0E	4 5 5
LSLA	Logical Shift Left	C ← [b7 … b0] ← 0	↕	↕	INH	48		1
LSRA	Logical Shift Right	0 → [b7 … b0] → C	↕	↕	INH	44		1

MOV *opr8a,opr8a*
MOV #*opr8i,opr8a*
MOV D[X],*opr8a*
MOV *opr8a*,D[X]
MOV #*opr8i*,D[X]
	Move	$(M)_{destination} \leftarrow (M)_{source}$	↕	–	DD
					IMD
					IX/DIR
					DIR/IX
					IMM/IX
						4E
						3E
						4E
						4E
						3E
							dd dd
							ii dd
							0E dd
							dd 0E
							ii 0E
								5
								4
								5
								5
								4

NOP	No Operation	None	–	–	INH	AC		1
ORA #*opr8i* ORA *opr8a* ORA ,X [(1)] ORA X	Inclusive OR Accumulator and Memory	A ← (A) | (M) A ← (A) | (X)	↕	–	IMM DIR IX DIR	AA BA BA BA	ii dd 0E 0F	2 3 3 3
ROLA	Rotate Left through Carry	C ← [b7 … b0] ← C	↕	↕	INH	49		1
RORA	Rotate Right through Carry	C → [b7 … b0] → C	↕	↕	INH	46		1
RTS	Return from Subroutine	Pull PC from shadow PC	–	–	INH	BE		3
SBC #*opr8i* SBC *opr8a* SBC ,X [(1)] SBC X	Subtract with Carry	A ← (A) – (M) – (C) A ← (A) – (X) – (C)	↕	↕	IMM DIR IX DIR	A2 B2 B2 B2	ii dd 0E 0F	2 3 3 3
SEC	Set Carry Bit	C ← 1	–	1	INH	39		1
SHA	Swap Shadow PC High with A	A ⇔ SPCH	–	–	INH	45		1
SLA	Swap Shadow PC Low with A	A ⇔ SPCL	–	–	INH	42		1
STA *opr8a* STA *opr5a* STA ,X [(1)] STA X	Store Accumulator in Memory	M ← (A)	↕	–	DIR SRT IX SRT	B7 E*x* / F*x* EE EF	dd	3 2 2 2

1. This is a pseudo instruction supported by the normal RS08 instruction set.
2. This instruction is different from that of the HC08 and HCS08 in that the RS08 does not auto-increment the index register.

Table 2-12. Instruction Set Summary (Sheet 5 of 5)

Source Form	Description	Operation	Effect on CCR Z	Effect on CCR C	Address Mode	Opcode	Operand	Cycles
STX *opr8a* [1]	Store Index Register in Memory	M ← (X)	↕	–	DIR	4E	0F dd	5
STOP	Put MCU into stop mode		–	–	INH	AE		2+
SUB #*opr8i* SUB *opr8a* SUB *opr4a* SUB ,X [1] SUB X	Subtract	A ← (A) – (M) A ← (A) – (X)	↕	↕	IMM DIR TNY IX DIR	A0 B0 7*x* 7E 7F	ii dd	2 3 3 3 3
TAX[1]	Transfer A to X	X ← (A)	↕	–	INH	EF		2
TST opr8a [1] TSTA [1] TST ,X [1] TSTX [1]	Test for Zero	(M) – $00 (A) – $00 (X) – $00	↕	–	DD INH IX INH	4E AA 4E 4E	dd dd 00 0E 0E 0F 0F	5 2 5 5
TXA[1]	Transfer X to A	A ← (X)	↕	–	INH	CF		3
WAIT	Put MCU into WAIT mode		–	–	INH	AF		2+

1. This is a pseudo instruction supported by the normal RS08 instruction set.
2. This instruction is different from that of the HC08 and HCS08 in that the RS08 does not auto-increment the index register.

Table 2-13. Opcode Map

HIGH \ LOW	DIR	DIR	TNY	DIR/REL	INH	TNY	TNY	TNY	SRT	SRT	IMM/INH	DIR/EXT	SRT	SRT	SRT	SRT
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	5 BRSET0 3 DIR	5 BSET0 2 DIR	4 INC 1 TNY	3 BRA 2 REL		4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2 SUB 2 IMM	3 SUB 2 DIR	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
1	5 BRCLR0 3 DIR	5 BCLR0 2 DIR	4 INC 1 TNY	5 CBEQ 3 DIR	4 CBEQA 3 IMM	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2 CMP 2 IMM	3 CMP 2 DIR	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
2	5 BRSET1 3 DIR	5 BSET1 2 DIR	4 INC 1 TNY		1 SLA 1 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2 SBC 2 IMM	3 SBC 2 DIR	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
3	5 BRCLR1 3 DIR	5 BCLR1 2 DIR	4 INC 1 TNY		1 COMA 1 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT			3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
4	5 BRSET2 3 DIR	5 BSET2 2 DIR	4 INC 1 TNY	3 BCC 2 REL	1 LSRA 1 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2 AND 2 IMM	3 AND 2 DIR	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
5	5 BRCLR2 3 DIR	5 BCLR2 2 DIR	4 INC 1 TNY	3 BCS 2 REL	1 SHA 1 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT			3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
6	5 BRSET3 3 DIR	5 BSET3 2 DIR	4 INC 1 TNY	3 BNE 2 REL	1 RORA 1 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2 LDA 2 IMM	3 LDA 2 DIR	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
7	5 BRCLR3 3 DIR	5 BCLR3 2 DIR	4 INC 1 TNY	3 BEQ 2 REL		4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT		3 STA 2 DIR	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
8	5 BRSET4 3 DIR	5 BSET4 2 DIR	4 INC 1 TNY	1 CLC 1 INH	1 LSLA 1 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2 EOR 2 IMM	3 EOR 2 DIR	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
9	5 BRCLR4 3 DIR	5 BCLR4 2 DIR	4 INC 1 TNY	1 SEC 1 INH	1 ROLA 1 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2 ADC 2 IMM	3 ADC 2 DIR	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
A	5 BRSET5 3 DIR	5 BSET5 2 DIR	4 INC 1 TNY	5 DEC 2 DIR	1 DECA 1 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2 ORA 2 IMM	3 ORA 2 DIR	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
B	5 BRCLR5 3 DIR	5 BCLR5 2 DIR	4 INC 1 TNY	6 DBNZ 3 DIR	4 DBNZA 2 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2 ADD 2 IMM	3 ADD 2 DIR	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
C	5 BRSET6 3 DIR	5 BSET6 2 DIR	4 INC 1 TNY	5 INC 2 DIR	1 INCA 1 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	1 NOP 1 INH	4 JMP 3 EXT	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
D	5 BRCLR6 3 DIR	5 BCLR6 2 DIR	4 INC 1 TNY			4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	3 BSR 2 REL	4 JSR 3 EXT	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
E	5 BRSET7 3 DIR	5 BSET7 2 DIR	4 INC 1 TNY	4 MOV 3 IMD	5 MOV 3 DD	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2+ STOP 1 INH	3 RTS 1 INH	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT
F	5 BRCLR7 3 DIR	5 BCLR7 2 DIR	4 INC 1 TNY	3 CLR 2 DIR	1 CLRA 1 INH	4 DEC 1 TNY	3 ADD 1 TNY	3 SUB 1 TNY	2 CLR 1 SRT	2 CLR 1 SRT	2+ WAIT 1 INH	5+ BGND 1 INH	3 LDA 1 SRT	3 LDA 1 SRT	2 STA 1 SRT	2 STA 1 SRT

INH Inherent **REL** Relative
IMM Immediate **SRT** Short
DIR Direct **TNY** Tiny
EXT Extended
DD Direct-Direct **IMD** Immediate-Direct

Gray box is decoded as illegal instruction

High Byte of Opcode in Hexadecimal B

Low Byte of Opcode in Hexadecimal 0

3 SUB 2 DIR	**RS08 Cycles** **Opcode Mnemonic** **Number of Bytes / Addressing Mode**

Section 3. Development Support

3.1 Introduction

Development support systems in the RS08 Family include the RS08 background debug controller (BDC).

The BDC provides a single-wire debug interface to the target MCU. This interface provides a convenient means for programming the on-chip FLASH and other nonvolatile memories. Also, the BDC is the primary debug interface for development and allows non-intrusive access to memory data and traditional debug features such as CPU register modify, breakpoint, and single-instruction trace commands.

In the RS08 Family, address and data bus signals are not available on external pins. Debug is done through commands fed into the target MCU via the single-wire background debug interface, including resetting the device without using a reset pin.

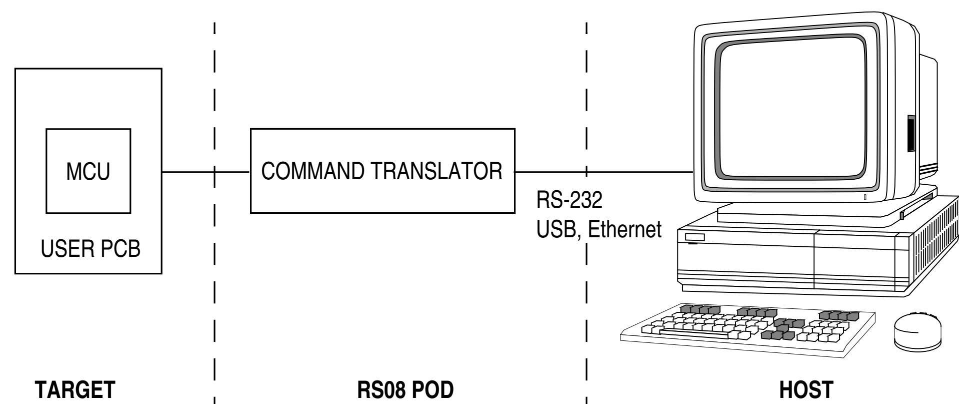


Figure 3-1. Connecting MCU to Host for Debugging

3.2 Features

Features of the RS08 background debug controller (BDC) include:

- Uses a single pin for background debug serial communications
- Non-intrusive of user memory resources; BDC registers are not located in the memory map
- SYNC command to determine target communications rate
- Non-intrusive commands allow access to memory resources while CPU is running user code without stopping applications
- Active background mode commands for CPU register access
- GO and TRACE1 commands
- BACKGROUND command can wake CPU from wait or stop modes
- BDC_RESET command allows host to reset MCU without using a reset pin
- One hardware address breakpoint built into BDC
- RS08 clock source runs in stop mode if BDM enabled to allow debugging when CPU is in stop mode
- COP watchdog suspended while in active background mode

3.3 RS08 Background Debug Controller (BDC)

All MCUs in the RS08 Family contain a single-wire background debug interface which supports in-circuit programming of on-chip non-volatile memory and sophisticated debug capabilities. Unlike debug interfaces on earlier 8-bit MCUs, this debug system provides for minimal interference with normal application resources. It does not use any user memory or locations in the memory map. It requires use of only the output-only BKGD pin. This pin will be shared with simple user output-only functions (typically port, comparator outputs, etc.), which can be easily debugged in normal user mode.

RS08 BDM commands are divided into two groups:

- Active background mode commands require that the target MCU is in active background mode (the user program is not running). The BACKGROUND command causes the target MCU to enter active background mode. Active background mode commands allow the CPU registers to be read or written and allow the user to trace one (TRACE1) user instruction at a time or GO to the user program from active background mode.
- Non-intrusive commands can be executed at any time even while the user program is running. Non-intrusive commands allow a user to read or write MCU memory locations or access status and control registers within the background debug controller (BDC).

Typically, a relatively simple interface pod is used to translate commands from a host computer into commands for the custom serial interface to the single-wire background debug system. Depending on the development tool vendor, this interface pod may use a standard RS-232 serial port, a parallel printer port, or some other type of communication such as Ethernet or a universal serial bus (USB) to communicate between the host PC and the pod.

Figure 3-2 shows the standard header for connection of a RS08 BDM pod. A pod is a small interface device that connects a host computer such as a personal computer to a target RS08 system. BKGD and GND are the minimum connections required to communicate with a target MCU. The pseudo-open-drain $\overline{\text{RESET}}$ signal is included in the connector to allow a direct hardware method for the host to force or monitor (if $\overline{\text{RESET}}$ is available as output) a target system reset.

The RS08 BDM pods supply the V_{PP} voltage to the RS08 MCU when in-circuit programming is required. The V_{PP} connection from the pod is shared with $\overline{\text{RESET}}$ as shown in Figure 3-2. For V_{PP} requirements see the FLASH specifications in the MCU electricals appendix.

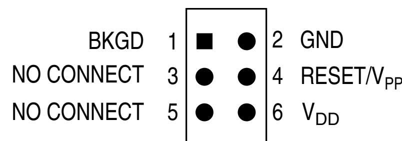


Figure 3-2. Standard RS08 BDM Tool Connector

Background debug controller (BDC) serial communications use a custom serial protocol that was first introduced on the M68HC12 Family of microcontrollers. This protocol requires that the host knows the communication clock rate, which is determined by the target BDC clock rate. If a host is attempting to communicate with a target MCU that has an unknown BDC clock rate, a SYNC command may be sent to the target MCU to request a timed sync response signal from which the host can determine the correct communication speed.

For RS08 MCUs, the BDC clock is the same frequency as the MCU bus clock. For a detailed description of the communications protocol, refer to Section 3.3.2, “Communication Details."

3.3.1 BKGD Pin Description

BKGD is the single-wire background debug interface pin. BKGD is a pseudo-open-drain pin that contains an on-chip pullup, therefore it requires no external pullup resistor. Unlike typical open-drain pins, the external resistor capacitor (RC) time constant on this pin, which is influenced by external capacitance, plays almost no role in signal rise time. The custom protocol provides for brief, actively driven speedup pulses to force rapid rise times on this pin without risking harmful drive level conflicts. Refer to Section 3.3.2, “Communication Details," for more detail.

The primary function of this pin is bidirectional serial communication of background debug commands and data. During reset, this pin selects between starting in active background mode and normal user mode running an application program. This pin is also used to request a timed sync response pulse to allow a host development tool to determine the target BDC clock frequency.

By controlling the BKGD pin and forcing an MCU reset (issuing a BDC_RESET command, or through a power-on reset (POR)), the host can force the target system to reset into active background mode rather than start the user application program. This is useful to gain control of a target MCU whose FLASH program memory has not yet been programmed with a user application program.

When no debugger pod is connected to the 6-pin BDM interface connector, the internal pullup on BKGD determines the normal operating mode.

On some RS08 devices, the BKGD pin may be shared with an alternative output-only function. To support BDM debugging, the user must disable this alternative function. Debugging of the alternative function should be done in normal user mode without using BDM.

3.3.2 Communication Details

The BDC serial interface requires the host to generate a falling edge on the BKGD pin to indicate the start of each bit time. The host provides this falling edge whether data is transmitted or received.

The BDC serial communication protocol requires the host to know the target BDC clock speed. Commands and data are sent most significant bit first (MSB-first) at 16 BDC clock cycles per bit. The interface times out if 512 BDC clock cycles occur between falling edges from the host. Any BDC command that was in progress when this timeout occurs is aborted without affecting the memory or operating mode of the target MCU system.

Figure 3-3 shows an external host transmitting a logic 1 or 0 to the BKGD pin of a target MCU. The host is asynchronous to the target so there is a 0-to-1 cycle delay from the host-generated falling edge to where the target perceives the beginning of the bit time. Ten target BDC clock cycles later, the target senses the bit level on the BKGD pin. Typically, the host actively drives the pseudo-open-drain BKGD pin during host-to-target transmissions to speed up rising edges. Because the target does not drive the BKGD pin during the host-to-target period, there is no need to treat the line as an open-drain signal during this period.

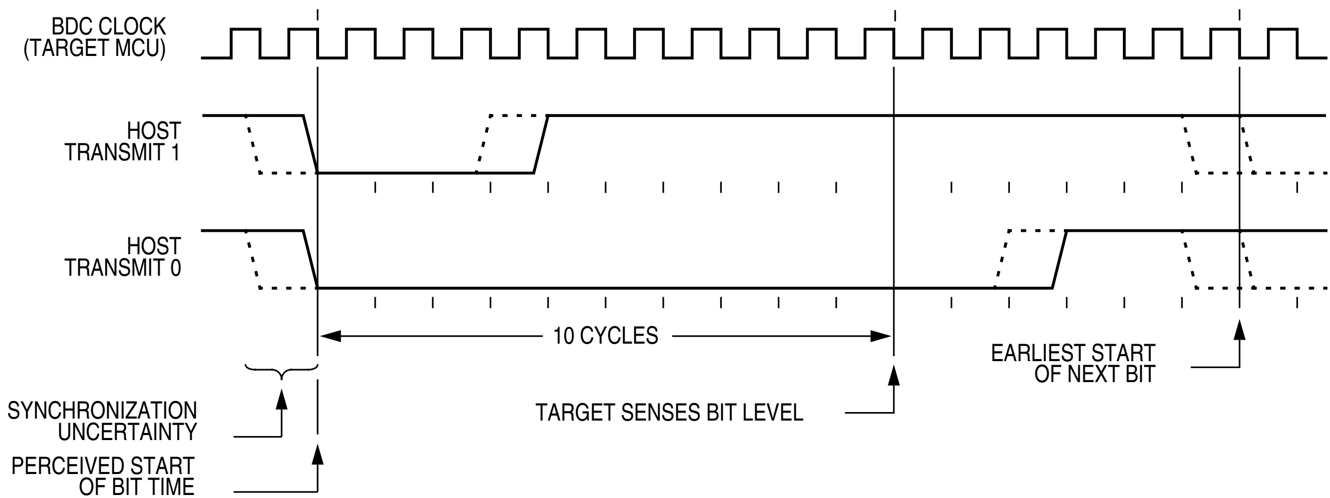


Figure 3-3. BDC Host-to-Target Serial Bit Timing

Figure 3-4 shows the host receiving a logic 1 from the target MCU. Because the host is asynchronous to the target, there is a 0-to-1 cycle delay from the host-generated falling edge on BKGD to the perceived start of the bit time in the target. The host holds the BKGD pin low long enough for the target to recognize it (at least two target BDC cycles). The host must release the low drive before the target drives a brief active-high speedup pulse seven cycles after the perceived start of the bit time. The host should sample the bit level approximately 10 cycles after it started the bit time.

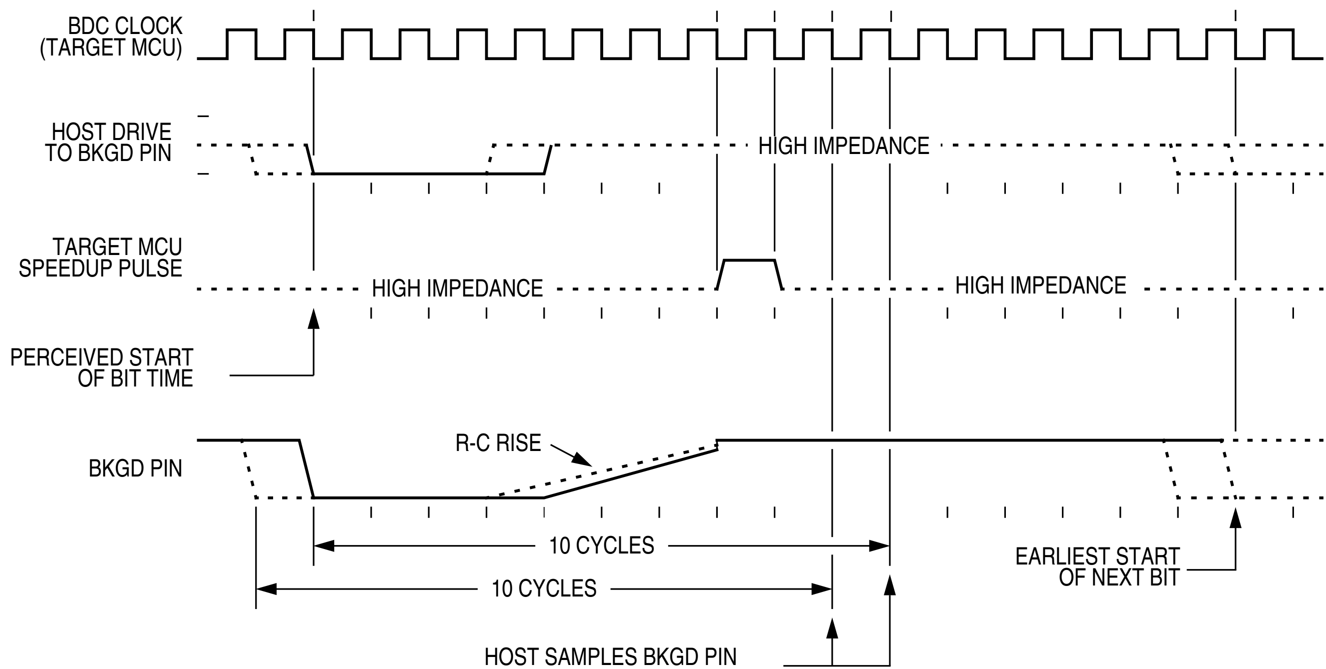


Figure 3-4. BDC Target-to-Host Serial Bit Timing (Logic 1)

Figure 3-5 shows the host receiving a logic 0 from the target MCU. Because the host is asynchronous to the target, there is a 0-to-1 cycle delay from the host-generated falling edge on BKGD to the start of the bit time as perceived by the target. The host initiates the bit time but the target finishes it. Because the target wants the host to receive a logic 0, it drives the BKGD pin low for 13 BDC clock cycles, then briefly drives it high to speed up the rising edge. The host samples the bit level approximately 10 cycles after starting the bit time.

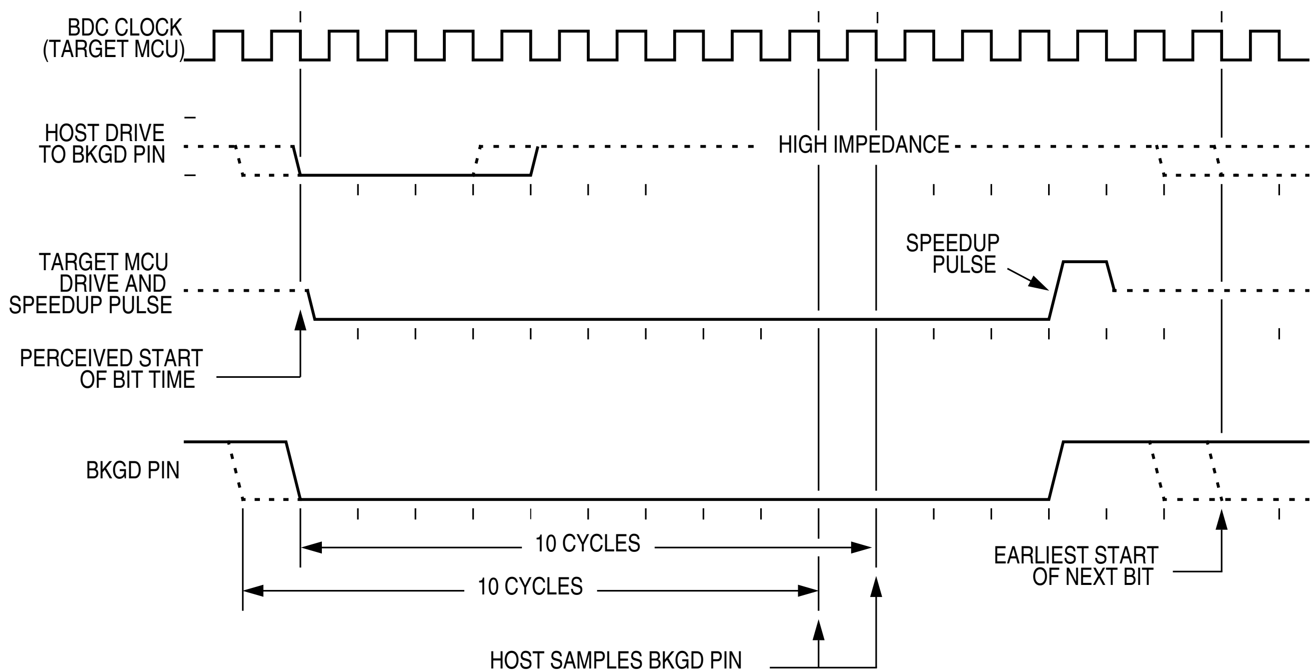


Figure 3-5. BDM Target-to-Host Serial Bit Timing (Logic 0)

3.3.3 SYNC and Serial Communication Timeout

The host initiates a host-to-target serial transmission by generating a falling edge on the BKGD pin. If BKGD is kept low for more than 128 target clock cycles, the target understands that a SYNC command was issued. In this case, the target will keep waiting for a rising edge on BKGD to answer the SYNC request pulse. If the rising edge is not detected, the target will keep waiting indefinitely, without any timeout limit. When a rising edge on BKGD occurs after a valid SYNC request, the BDC will drive the BKGD pin low for exactly 128 BDC cycles.

Consider now the case where the host returns BKGD to logic 1 before 128 cycles. This is interpreted as a valid bit transmission, and not as a SYNC request. The target will keep waiting for another falling edge marking the start of a new bit. If, however, a new falling edge is not detected by the target within 512 clock cycles since the last falling edge, a timeout occurs and the current command is discarded without affecting memory or the operating mode of the MCU. This is referred as a soft-reset to the BDC.

If a read command is issued but the data is not retrieved within 512 serial clock cycles, a soft-reset will occur causing the command to be disregarded. The data is not available for retrieving after the timeout has occurred. A soft-reset is also used to end a READ_BLOCK or WRITE_BLOCK command.

The following describes the actual bit-time requirements for a host to guarantee logic 1 or 0 bit transmission without the target timing out or interpreting the bit as a SYNC command:

- To send a logic 0, BKGD must be kept low for a minimum of 12 BDC cycles and up to 511 BDC cycles except for the first bit of a command sequence, which will be detected as a SYNC request.
- To send a logic 1, BKGD must be held low for at least four BDC cycles, be released by the eighth cycle, and be held high until at least the sixteenth BDC cycle.
- Subsequent bits must occur within 512 BDC cycles of the last bit sent.

3.4 BDC Registers and Control Bits

The BDC contains two non-CPU accessible registers:

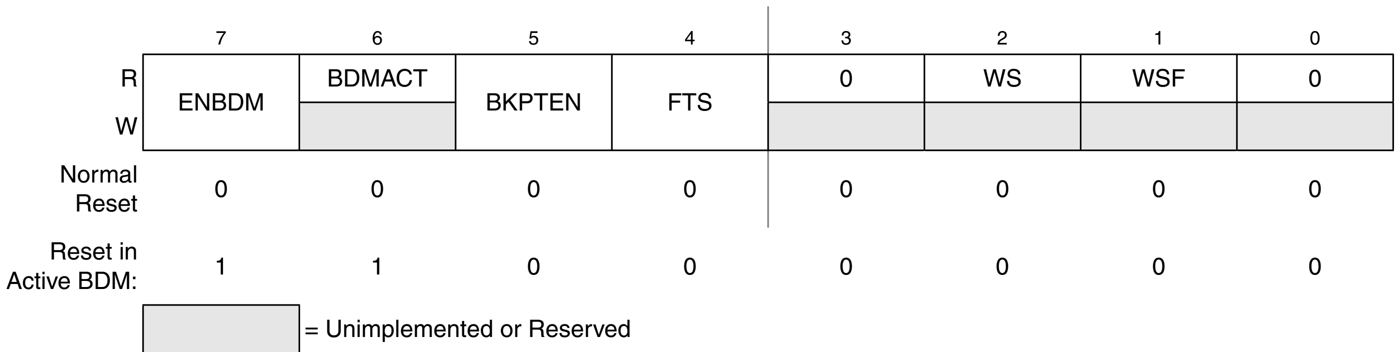
- The BDC status and control register (BDCSCR) is an 8-bit register containing control and status bits for the background debug controller.
- The BDC breakpoint register (BDCBKPT) holds a 16-bit breakpoint match address.

These registers are accessed with dedicated serial BDC commands and are not located in the memory space of the target MCU (so they do not have addresses and cannot be accessed by user programs).

Some of the bits in the BDCSCR have write limitations; otherwise, these registers may be read or written at any time. For example, the ENBDM control bit may not be written while the MCU is in active background mode. This prevents the ambiguous condition of the control bit forbidding active background mode while the MCU is already in active background mode. Also, the status bits (BDMACT, WS, and WSF) are read-only status indicators and can never be written by the WRITE_CONTROL serial BDC command.

3.4.1 BDC Status and Control Register (BDCSCR)

This register can be read or written by serial BDC commands (READ_STATUS and WRITE_CONTROL) but is not accessible to user programs because it is not located in the normal memory map of the MCU.



	7	6	5	4	3	2	1	0
R	ENBDM	BDMACT	BKPTEN	FTS	0	WS	WSF	0
W								
Normal Reset	0	0	0	0	0	0	0	0
Reset in Active BDM:	1	1	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 3-6. BDC Status and Control Register (BDCSCR)

Table 3-1. BDCSCR Register Field Descriptions

Field	Description
7 ENBDM	**Enable BDM (Permit Active Background Mode)** — Typically, this bit is written to 1 by the debug host shortly after the beginning of a debug session or whenever the debug host resets the target and remains 1 until a normal reset clears it. If the application can go into stop mode, this bit is required to be set if debugging capabilities are required. 0 BDM cannot be made active (non-intrusive commands still allowed). 1 BDM can be made active to allow active background mode commands.
6 BDMACT	**Background Mode Active Status** — This is a read-only status bit. 0 BDM not active (user application program running). 1 BDM active and waiting for serial commands.
5 BKPTEN	**BDC Breakpoint Enable** — If this bit is clear, the BDC breakpoint is disabled and the FTS (force tag select) control bit and BDCBKPT match register are ignored 0 BDC breakpoint disabled. 1 BDC breakpoint enabled.
4 FTS	**Force/Tag Select** — When FTS = 1, a breakpoint is requested whenever the CPU address bus matches the BDCBKPT match register. When FTS = 0, a match between the CPU address bus and the BDCBKPT register causes the fetched opcode to be tagged. If this tagged opcode ever reaches the end of the instruction queue, the CPU enters active background mode rather than executing the tagged opcode. 0 Tag opcode at breakpoint address and enter active background mode if CPU attempts to execute that instruction. 1 Breakpoint match forces active background mode at next instruction boundary (address need not be an opcode).
2 WS	**Wait or Stop Status** — When the target CPU is in wait or stop mode, most BDC commands cannot function. However, the BACKGROUND command can be used to force the target CPU out of wait or stop and into active background mode where all BDC commands work. Whenever the host forces the target MCU into active background mode, the host should issue a READ_STATUS command to check that BDMACT = 1 before attempting other BDC commands. 0 Target CPU is running user application code or in active background mode (was not in wait or stop mode when background became active). 1 Target CPU is in wait or stop mode, or a BACKGROUND command was used to change from wait or stop to active background mode.
1 WSF	**Wait or Stop Failure Status** — This status bit is set if a memory access command failed due to the target CPU executing a wait or stop instruction at or about the same time. The usual recovery strategy is to issue a BACKGROUND command to get out of wait or stop mode into active background mode, repeat the command that failed, then return to the user program. (Typically, the host would restore CPU registers and stack values and re-execute the wait or stop instruction.) 0 Memory access did not conflict with a wait or stop instruction. 1 Memory access command failed because the CPU entered wait or stop mode.

3.4.2 BDC Breakpoint Match Register

This 16-bit register holds the 14-bit address for the hardware breakpoint in the BDC. The BKPTEN and FTS control bits in BDCSCR are used to enable and configure the breakpoint logic. Dedicated serial BDC commands (READ_BKPT and WRITE_BKPT) are used to read and

write the BDCBKPT register. Breakpoints are normally set while the target MCU is in active background mode before running the user application program. However, because READ_BKPT and WRITE_BKPT are non-intrusive commands, they could be executed even while the user program is running. For additional information about setup and use of the hardware breakpoint logic in the BDC, refer to Section 3.6, "BDC Hardware Breakpoint."

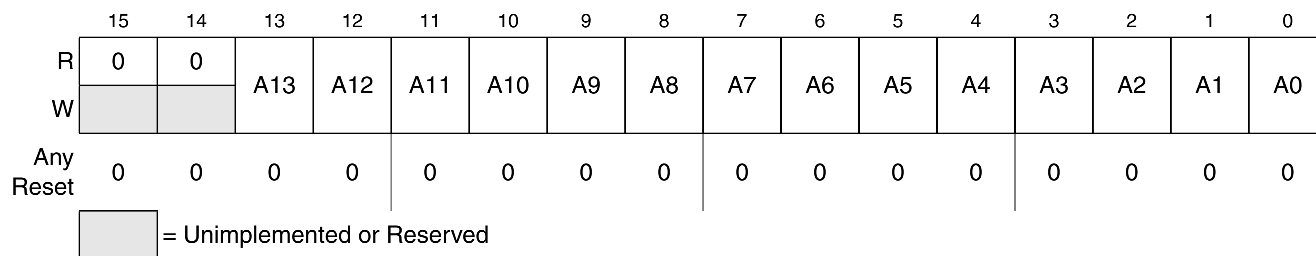


Figure 3-7. BDC Breakpoint Match Register (BDCBKPT)

3.5 RS08 BDC Commands

BDC commands are sent serially from a host computer to the BKGD pin of the target MCU. All commands and data are sent MSB-first using a custom BDC communications protocol. Active background mode commands require that the target MCU is currently in the active background mode while non-intrusive commands may be issued at any time whether the target MCU is in active background mode or running a user application program.

Table 3-2 shows all RS08 BDC commands, a shorthand description of their coding structure, and the meaning of each command.

Coding Structure Nomenclature

The following nomenclature is used in Table 3-2 to describe the coding structure of the BDC commands.

Commands begin with an 8-bit command code in the host-to-target direction (most significant bit first)

/ = Separates parts of the command

d	=	Delay 16 to 511 target BDC clock cycles
soft-reset	=	Delay of at least 512 BDC clock cycles from last host falling-edge
AAAA	=	16-bit address in the host-to-target direction[1]
RD	=	Eight bits of read data in the target-to-host direction
WD	=	Eight bits of write data in the host-to-target direction
RD16	=	16 bits of read data in the target-to-host direction
WD16	=	16 bits of write data in the host-to-target direction
SS	=	the contents of BDCSCR in the target-to-host direction (STATUS)
CC	=	Eight bits of write data for BDCSCR in the host-to-target direction (CONTROL)
RBKP	=	16 bits of read data in the target-to-host direction (from BDCBKPT breakpoint register)
WBKP	=	16 bits of write data in the host-to-target direction (for BDCBKPT breakpoint register)

Table 3-2. RS08 BDC Command Summary

Command Mnemonic	Active Background Mode/ Non-Intrusive	Coding Structure	Description
SYNC	Non-intrusive	n/a(1)	Request a timed reference pulse to determine target BDC communication speed
BDC_RESET	Any CPU mode	18(2)	Request an MCU reset
BACKGROUND	Non-intrusive	90/d	Enter active background mode if enabled (ignore if ENBDM bit equals 0)
READ_STATUS	Non-intrusive	E4/SS	Read BDC status from BDCSCR
WRITE_CONTROL	Non-intrusive	C4/CC	Write BDC controls in BDCSCR
READ_BYTE	Non-intrusive	E0/AAAA/d/RD	Read a byte from target memory
READ_BYTE_WS	Non-intrusive	E1/AAAA/d/SS/RD	Read a byte and report status
WRITE_BYTE	Non-intrusive	C0/AAAA/WD/d	Write a byte to target memory
WRITE_BYTE_WS	Non-intrusive	C1/AAAA/WD/d/SS	Write a byte and report status
READ_BKPT	Non-intrusive	E2/RBKP	Read BDCBKPT breakpoint register
WRITE_BKPT	Non-intrusive	C2/WBKP	Write BDCBKPT breakpoint register

1. The RS08 CPU uses only 14 bits of address and occupies the lower 14 bits of the 16-bit AAAA address field. The values of address bits 15 and 14 in AAAA are truncated and thus do not matter.

Table 3-2. RS08 BDC Command Summary (Continued)

Command Mnemonic	Active Background Mode/ Non-Intrusive	Coding Structure	Description
GO	Active background mode	08/d	Go to execute the user application program starting at the address currently in the PC
TRACE1	Active background mode	10/d	Trace one user instruction at the address in the PC, then return to active background mode
READ_BLOCK	Active background mode	80/AAAA/d/RD[3]	Read a block of data from target memory starting from AAAA continuing until a soft-reset is detected
WRITE_BLOCK	Active background mode	88/AAAA/WD/d[4]	Write a block of data to target memory starting at AAAA continuing until a soft-reset is detected
READ_A	Active background mode	68/d/RD	Read accumulator (A)
WRITE_A	Active background mode	48/WD/d	Write accumulator (A)
READ_CCR_PC	Active background mode	6B/d/RD16[5]	Read the CCR bits z, c concatenated with the 14-bit program counter (PC) RD16=zc:PC
WRITE_CCR_PC	Active background mode	4B/WD16/d[6]	Write the CCR bits z, c concatenated with the 14-bit program counter (PC) WD16=zc:PC
READ_SPC	Active background mode	6F/d/RD16[7]	Read the 14-bit shadow program counter (SPC) RD16=0:0:SPC
WRITE_SPC	Active background mode	4F/WD16/d[8]	Write 14-bit shadow program counter (SPC) WD16 = x:x:SPC, the two most significant bits shown by “x” are ignored by target

1. The SYNC command is a special operation which does not have a command code.
2. 18 was HCS08 BDC command for TAGGO.
3. Each RD requires a delay between host read data byte and next read, command ends when target detects a soft-reset.
4. Each WD requires a delay between host write data byte and next byte, command ends when target detects a soft-reset.
5. HCS08 BDC had separate READ_CCR and READ_PC commands, the RS08 BDC combined this commands.
6. HCS08 BDC had separate WRITE_CCR and WRITE_PC commands, the RS08 BDC combined this commands.
7. 6F is READ_SP (read stack pointer) for HCS08 BDC.
8. 4F is WRITE_SP (write stack pointer) for HCS08 BDC.

3.5.1 SYNC

Request a timed pulse (128 BDC clock cycles) from target Non-Intrusive

The SYNC command is unlike other BDC commands because the host does not necessarily know the correct communications speed to use for BDC communications until after it has analyzed the response to the SYNC command.

To issue a SYNC command, the host:

- Drives the BKGD pin low for at least 128 cycles of the slowest possible BDC clock (the slowest clock)
- Drives BKGD high for a brief speedup pulse to get a fast rise time (this speedup pulse is typically one cycle of the fastest clock in the system)
- Removes all drive to the BKGD pin so it reverts to high impedance
- Monitors the BKGD pin for the sync response pulse

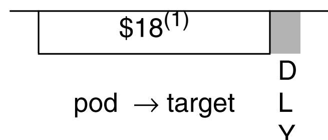
The target, upon detecting the SYNC request from the host (which is a much longer low time than would ever occur during normal BDC communications):

- Waits for BKGD to return to a logic 1
- Delays 16 cycles to allow the host to stop driving the high speedup pulse
- Drives BKGD low for 128 BDC clock cycles
- Drives a 1-cycle high speedup pulse to force a fast rise time on BKGD
- Removes all drive to the BKGD pin so it reverts to high impedance

The host measures the low time of this 128-cycle sync response pulse and determines the correct speed for subsequent BDC communications. Typically, the host can determine the correct communication speed within a few percent of the actual target speed and the communication protocol can easily tolerate speed errors of several percent.

3.5.2 BDC_RESET

Reset the MCU — Any CPU Mode

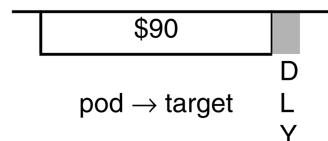


1. $18 is the HCS08 BDC command for TAGGO

Provided the BKGD pin is available, the target MCU can be reset to enter active background mode by the BDC_RESET command followed immediately by asserting the BKGD pin low until the MCU reset sequence finishes. If BKGD is left high after a BDC_RESET, the target MCU will reset into normal user mode. Systems that can place the CPU into wait or stop mode require ENBDM to be set to allow the BDC clocks to remain active while the CPU is in stop mode.

3.5.3 BACKGROUND

Enter Active Background Mode (if Enabled) — Non-intrusive



Provided ENBDM is set, the BACKGROUND command causes the target MCU to enter active background mode as soon as the current CPU instruction finishes.

If ENBDM is clear (its default value), the BACKGROUND command will be ignored by the BDC. The host should attempt to enable ENBDM using WRITE_STATUS and verify that ENBDM is set using READ_STATUS before issuing a BACKGROUND command.

If the target application uses wait or stop mode, it may not be possible to enter active background mode without causing a wakeup using an external interrupt.

A delay of 16 BDC clock cycles is required after the BACKGROUND command to allow the target MCU to finish its current CPU instruction and enter active background mode before a new BDC command can be accepted.

Normally, the development host would set ENBDM once at the beginning of a debug session or after a target system reset, and then leave the ENBDM bit set during debugging operations. During debugging, the host would use GO and TRACE1 commands to move from active background mode to normal user program execution and would use BACKGROUND commands or breakpoints to return to active background mode. This method of debugging allow the host debugger to enter active background mode even if the CPU enters wait or stop mode.

3.5.4 READ_STATUS

Read Status from BDCSCR | Non-intrusive

$E4	Read BDCSCR (8)
pod → target	target → pod

This command allows a host to read the contents of the BDC status and control register (BDCSCR). This register is not in the memory map of the target MCU and is accessible only through READ_STATUS and WRITE_CONTROL serial BDC commands.

The most common use for this command is to allow the host to determine whether the target MCU is executing normal user program instructions or if it is in active background mode. For example, during a typical debug session, the host might set breakpoints in the user program and then use a GO command to begin normal user program execution. The host would then periodically execute READ_STATUS commands to determine when a breakpoint has been encountered and the target processor has gone into active background mode. After the target has entered active background mode, the host reads the contents of target CPU registers.

READ_STATUS can also be used to check whether the target MCU has gone into wait or stop mode. During a debug session, the host or user may decide that it has taken too long to reach a breakpoint in the user program. The host could then issue a READ_STATUS command and check the WS status bit to determine whether the target MCU is still running user code or whether it has entered wait or stop mode. If WS = 0 and BDMACT = 0, meaning it is running user code and is not in wait or stop, the host might choose to issue a BACKGROUND command to stop the user program and enter active background mode where the host can check the CPU registers and find out what the target program is doing.

3.5.5 WRITE_CONTROL

Write Control Bits in BDCSCR | Non-intrusive

$C4	Write BDCSCR (8)
pod → target	pod → target

This command is used to enable active background mode and control the hardware breakpoint logic in the BDC by writing to control bits in the BDC status and control register (BDCSCR). This register is not in the memory map of the target MCU and is only accessible through READ_STATUS and WRITE_CONTROL serial BDC commands. Some bits in BDCSCR have write restrictions (such as status bits BDMACT, WS, and WSF, which are read-only status indicators, and ENBDM, which cannot be cleared while BDM is active.

The ENBDM control bit defaults to 0 (active background mode not allowed) when the target MCU is reset in normal operating mode. WRITE_CONTROL is used to enable the active background mode. This is normally done once and ENBDM is left enabled throughout the debug session. However, the debug system may want to change ENBDM to 0 and measure true stop current in the target system (because ENBDM = 1 prevents the clock generation circuitry from disabling the internal clock oscillator or crystal oscillator to allow the BDC clock to continue when the CPU executes a STOP instruction).

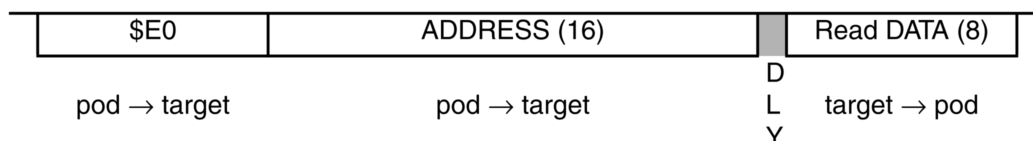
The breakpoint enable (BKPTEN) and force/tag select (FTS) control bits are used to control the hardware breakpoint logic in the BDC. This is a

single breakpoint that compares the current CPU address against the value in the BDCBKPT register. A WRITE_CONTROL command is used to change BKPTEN and FTS, and a WRITE_BKPT command is used to write the 16-bit BDCBKPT address match register.

3.5.6 READ_BYTE

Read Data from Target Memory Location

Non-intrusive

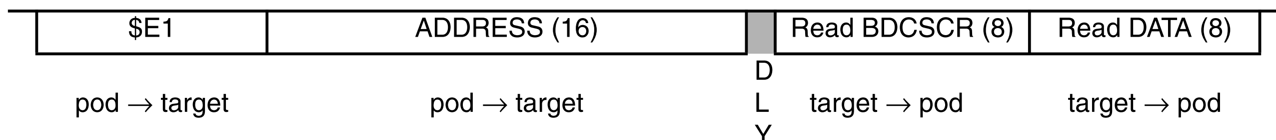


This command is used to read the contents of a memory location in the target MCU without checking the BDC status to ensure the data is valid. In systems where the target is currently in active background mode or is known to be executing a program which has no STOP or WAIT instructions, READ_BYTE is faster than the more general READ_BYTE_WS, which reports status in addition to returning the requested read data. The most significant use of the READ_BYTE command is during in-circuit FLASH programming where the host downloads data to be programmed at the same time the target CPU is executing the code that actually programs the FLASH memory. Because the host provides the FLASH programming code, it can guarantee that there are no STOP or WAIT instructions.

The READ_BYTE command should not be used in general-purpose user programs that use STOP or WAIT instructions. To avoid invalid data due to CPU operation in stop or wait mode, the READ_BYTE_WS should be used determine whether the data is valid.

3.5.7 READ_BYTE_WS

Read Data from Target and Report Status Non-intrusive

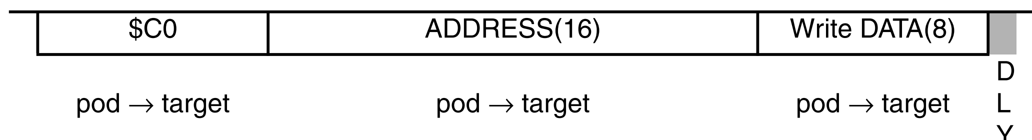


This is the command normally used by a host debug system to perform general-purpose memory read operations. In addition to returning the data from the requested target memory location, this command returns the contents of the BDC status and control register. The status information can be used to determine whether the data that was returned is valid or not. If the target MCU was just entering wait or stop mode at the time of the read, the wait/stop failure (WSF) status bit will be 1. If WSF is 0, the data that was returned is valid.

If the WSF bit indicates a WAIT or STOP instruction caused the read operation to fail, do a BACKGROUND command to force the target system out of wait or stop mode and into active background mode. From there, repeat the failed read operation, and if desired, adjust the PC to point to the WAIT or STOP instruction and issue a GO to return the target to wait or stop mode.

3.5.8 WRITE_BYTE

Write Data to Target Memory Location Non-intrusive



This command is used to write the contents of a memory location in the target MCU without checking the BDC status to ensure the write was completed successfully. In systems where the target is currently in active background mode or is known to be executing a program that has no STOP or WAIT instructions, WRITE_BYTE is faster than the more general WRITE_BYTE_WS, which reports status in addition to

performing the requested write operation. The most significant use of the WRITE_BYTE command is during in-circuit FLASH programming where the host downloads data to be programmed at the same time the target CPU is executing the code that actually programs the FLASH memory. Because the host provides the FLASH programming code, it can guarantee that there are no STOP or WAIT instructions.

The WRITE_BYTE command should not be used in general-purpose user programs which use STOP or WAIT instructions that can occur at any time. To avoid invalid data due to CPU in stop or wait mode, the WRITE_BYTE_WS should be used to determine whether the data that was returned is valid or not.

3.5.9 WRITE_BYTE_WS

Write Data to Target and Report Status — Non-intrusive

$C1	ADDRESS (16)	Write DATA (8)		Read BDCSCR (8)
pod → target	pod → target	pod → target	D L Y	target → pod

This is the command normally used by a host debug system to perform general-purpose memory write operations. In addition to performing the requested write to a target memory location, this command returns the contents of the BDC status and control register. The status information can be used to determine whether the write operation was completed successfully. If the target MCU was just entering wait or stop mode at the time of the read, the wait/stop failure (WSF) status bit will be 1 and the write command is cancelled. If WSF is 0, the write operation was completed successfully.

If the WSF bit indicates a WAIT or STOP instruction caused the write operation to fail, do a BACKGROUND command to force the target system out of wait or stop mode and into active background mode. From there, repeat the failed write operation, and if desired, adjust the PC to point to the WAIT or STOP instruction and issue a GO to return the target to wait or stop mode.

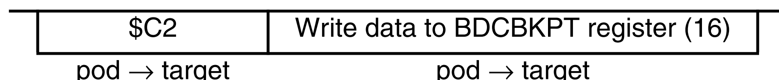
3.5.10 READ_BKPT

Read 16-Bit BDC Breakpoint Register (BDCBKPT) Non-intrusive

This command is used to read the 16-bit BDCBKPT address match register in the hardware breakpoint logic in the BDC.

3.5.11 WRITE_BKPT

Write 16-Bit BDC Breakpoint Register (BDCBKPT) Non-intrusive

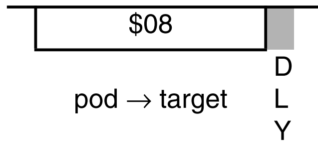


This command is used to write a 16-bit address value into the BDCBKPT register in the BDC. This establishes the address of a breakpoint. The BKPTEN bit in the BDCSCR determines whether the breakpoint is enabled. If BKPTEN = 1 and the FTS control bit in the BDCSCR is set (force), a successful match between the CPU address and the value in the BDCBKPT register will force a transition to active background mode at the next instruction boundary. If BKPTEN = 1 and FTS = 0, the opcode at the address specified in the BDCBKPT register will be tagged as it is fetched into the instruction queue. If and when a tagged opcode reaches the top of the instruction queue and is about to be executed, the MCU will enter active background mode rather than execute the tagged instruction.

In normal debugging environments, breakpoints are established while the target MCU is in active background mode before going to the user program. However, because this is a non-intrusive command, it could be executed even when the MCU is running a user application program.

3.5.12 GO

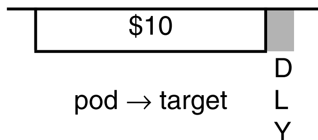
Start Execution of User Program Starting at Current PC — Active Background Mode



This command is used to exit the active background mode and begin execution of user program instructions starting at the address in the PC. Typically, the host debug monitor program modifies the PC value (using a WRITE_PC command) before issuing a GO command to go to an arbitrary point in the user program. This WRITE_PC command is not needed if the host simply wants to continue the user program where it left off when it entered active background mode.

3.5.13 TRACE1

Run One User Instruction Starting at the Current PC — Active Background Mode

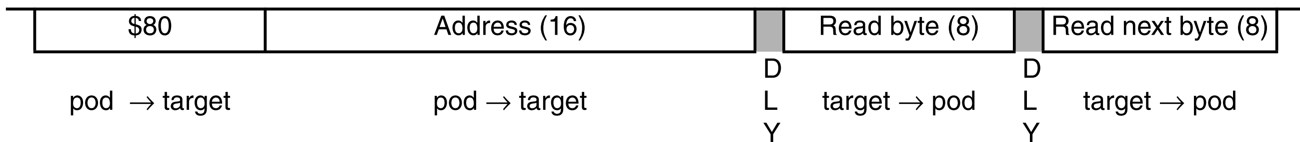


This command is used to run one user instruction and return to active background mode. The address in the PC determines what user instruction will be executed, and the PC value after TRACE1 is completed will reflect the results of the executed instruction.

3.5.14 READ_BLOCK

Read a block of data from target memory

Active Background Mode

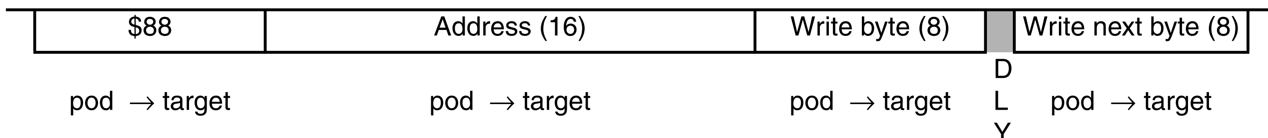


This command is used to read the contents of a block of memory starting at the location provided in the command. Because this command is only available in active background mode, the CPU cannot enter wait or stop; therefore the command does not return the contents of the BDMSCR. This command will continue to read data from the next memory location until the BDC detects a soft-reset, which is a timeout of 512 BDC cycles from the last falling edge of the host. This command is useful when dumping large blocks of data for memory displays or in programmers to verify the device data after programming.

3.5.15 WRITE_BLOCK

Write a block of data to target memory

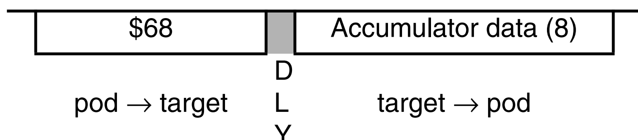
Active Background Mode



This command is used to write data to a block of memory starting at the location provided in the command. Because this command is only available in active background mode, the CPU cannot enter wait or stop, therefore the command does not return the contents of the BDMSCR. This command will continue to write data to the next memory location until the BDC detects a soft-reset, which is a timeout of 512 BDC cycles from the last falling edge of the host. This command is useful when writing large blocks of data such as full blocks of RAM.

3.5.16 READ_A

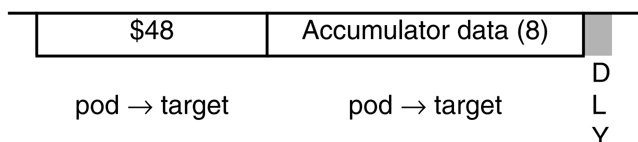
Read Accumulator A of the Target CPU | Active Background Mode



Read the contents of the accumulator (A) of the target CPU. Because the CPU in the target MCU is effectively halted while the target is in active background mode, there is no need to save the target CPU registers on entry into active background mode and no need to restore them on exit from active background mode to a user program.

3.5.17 WRITE_A

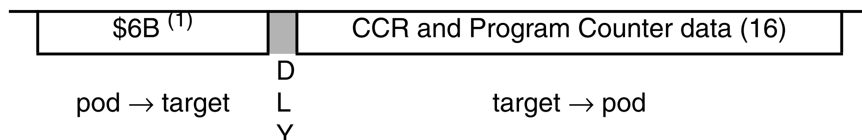
Write Accumulator A of the Target CPU | Active Background Mode



Write new data to the accumulator (A) of the target CPU. This command can be used to change the value in the accumulator before returning to the user application program via a GO or TRACE1 command.

3.5.18 READ_CCR_PC

Reads the CCR and the Program Counter of the Target CPU | Active Background Mode



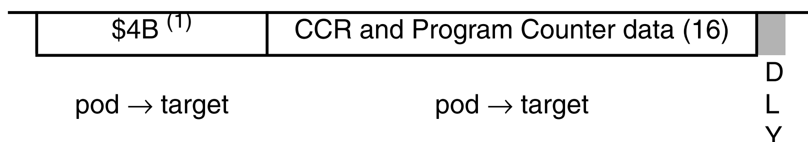
1. $6B is the HCS08 BDC command for READ_PC, the HCS08 CCR was read using READ_CCR

Read the Z and C bits of the condition code register (CCR) and contents of the 14-bit program counter (PC) of the target CPU.

The value in the PC when the target MCU enters active background mode is the address of the instruction that would have executed next if the MCU had not entered active background mode. If the target CPU was in wait or stop mode when a BACKGROUND command caused it to go to active background mode, the PC will hold the address of the instruction after the WAIT or STOP instruction that was responsible for the target CPU being in wait or stop, and the WS bit will be set. In the boundary case (where an interrupt and a BACKGROUND command arrived at approximately the same time and the interrupt was responsible for the target CPU leaving wait or stop—and then the BACKGROUND command took effect), the WS bit will be clear and the PC will be pointing at the next instruction after the WAIT or STOP. In the case of a software breakpoint (where the host placed a BGND opcode at the desired breakpoint address), the PC will be pointing at the address immediately following the inserted BGND opcode, and the host monitor will adjust the PC backward by one after removing the software breakpoint.

3.5.19 WRITE_CCR_PC

Write the CCR and Program Counter of the Target CPU — Active Background Mode



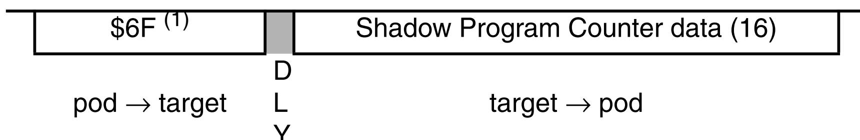
1. $4B is the HCS08 BDC command for WRITE_PC, the HCS08 CCR was written using WRITE_CCR

This command is used to change the contents of Z and C bits the condition code register (CCR) and the 14-bit program counter (PC) of the target CPU before returning to the user application program via a GO or TRACE1 command.

3.5.20 READ_SPC

Reads the Shadow Program Counter of the Target CPU

Active Background Mode



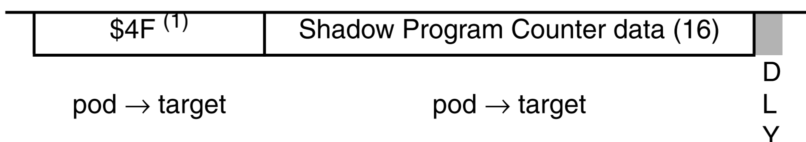
1. $6F is the HCS08 BDC command for READ_SP (stack pointer)

Read the contents of the 14-bit shadow program counter (PC) of the target CPU.

3.5.21 WRITE_SPC

Write the Shadow Program Counter of the Target CPU

Active Background Mode



1. $4F is the HCS08 BDC command for WRITE_SP (stack pointer)

Writes the contents of the 14-bit shadow program counter (PC) of the target CPU. The two most significant bits of the 16-bit WD16 are ignored by the target.

3.6 BDC Hardware Breakpoint

The BDC includes one hardware breakpoint, which compares the CPU address bus to a 14-bit match value in the BDCBKPT register. This breakpoint can generate a forced breakpoint or a tagged breakpoint.

A forced breakpoint causes the CPU to enter active background mode at the first instruction boundary following any access to the breakpoint address. The tagged breakpoint causes the instruction opcode at the breakpoint address to be tagged so that the CPU will enter active background mode rather than executing that instruction if and when it reaches the end of the instruction queue. Tagged breakpoints must be

placed only at the address of an instruction opcode while forced breakpoints can be set at any address.

The breakpoint enable (BKPTEN) control bit in the BDC status and control register (BDCSCR) is used to enable the breakpoint logic (BKPTEN = 1). When BKPTEN = 0, its default value after reset, the breakpoint logic is disabled and no BDC breakpoints are requested regardless of the values in other BDC breakpoint registers and control bits. The force/tag select (FTS) control bit in BDCSCR is used to select forced (FTS = 1) or tagged (FTS = 0) type breakpoints.

The 8-bit BDCSCR and the 16-bit BDCBKPT address match register are built directly into the BDC and are not accessible in the normal MCU memory map. This means that the user application program cannot access these registers. Dedicated BDC serial commands are the only way to access these registers. READ_STATUS and WRITE_CONTROL are used to read or write BDCSCR, respectively. READ_BKPT and WRITE_BKPT are used to read or write the 16-bit BDCBKPT address match register, respectively.

If the background mode has not been enabled, ENBDM = 0, the CPU will cause an illegal opcode reset instead of going into active background mode.

3.7 BDM in Stop and Wait Modes

The clock architecture of the RS08 permits the BDC to prevent the BDC clock from stopping during wait or stop mode if ENBDM is set. In such a system, the debug host can use READ_STATUS commands to determine whether the target is in a low-power mode (wait or stop). If the target is in wait or stop (WS =1), the BACKGROUND command may be used to wake the target and place it in active background mode. When the CPU returns to active background mode, the PC will be pointing at the address of the instruction after the WAIT or STOP. From active background mode, the debug host can read or write memory or registers. The debug host can then choose to adjust the PC such that a GO command will return the target MCU to wait or stop mode.

If the CPU is in active background mode and the user issues a GO command and the next instruction is WAIT or STOP, the CPU goes into wait or stop mode.

If the user issues a TRACE1 command and the next instruction is WAIT or STOP, the CPU executes and completes the instruction and re-enters active background mode. When the CPU returns to active background mode, the PC will be pointing at the address of the instruction after the WAIT or STOP.

3.8 BDC Command Execution

The RS08 BDC requires no system resources except for the BKGD pin. A running application that is not in wait or stop mode can have its memory or register contents read or written without stopping the application. The RS08 BDC steals CPU cycles whenever a BDC command requires reading or writing memory or registers. This has little impact on real-time operation of user code because a memory access command takes eight bits for the command, 16 bits for the address, at least eight bits for the data, and a 16-cycle delay within the command. Each bit time is at least 16 BDC clock cycles so (32 x 16) +16 = 528 cycles, thus the worst case impact is no more than 1/528 cycles, even if there are continuous back-to-back memory access commands through the BDM, which would be very ugrep timenlikely.

Because the RS08 BDC doesn't wait for free cycles, the delays between address and data in read commands and the delay after the data portion of a write command can be very short. In the RS08, the delay within a memory access command is 16 target bus cycles. For read or write accesses to registers within the BDC (STATUS and BDCBKPT), no delay is required.

Because the memory access commands in the RS08 BDC are actually performed by the CPU circuitry, it is possible for a memory access to fail when the memory access command coincides with the CPU entering wait or stop.

The WSF status bit was added to indicate an access failed because the CPU was just entering wait or stop mode. The READ_BYTE_WS

command returns byte of status information and read data byte. The WRITE_BYTE_WS command includes the byte of status information in the target-to-host direction after the write data byte (which is in the host-to-target direction).

Appendix A. Instruction Set Details

A.1 Introduction

This section contains detailed information for all RS08 Family instructions. The instructions are arranged in alphabetical order with the instruction mnemonic set in larger type for easy reference.

A.2 Nomenclature

This nomenclature is used in the instruction descriptions throughout this section.

Operators

( )	=	Contents of register or memory location shown inside parentheses
←	=	Is loaded with (read: "gets")
⇔	=	Exchange with
&	=	Boolean AND
|	=	Boolean OR
⊕	=	Boolean exclusive-OR
:	=	Concatenate
+	=	Add

CPU registers

A	=	Accumulator
CCR	=	Condition code register
PC	=	Program counter
PCH	=	Program counter, higher order (most significant) eight bits
PCL	=	Program counter, lower order (least significant) eight bits
SPC	=	Shadow program counter

SPCH = Shadow program counter, higher order (most significant) six bits

SPCL = Shadow program counter, lower order (least significant) eight bits

Memory and addressing

M = A memory location or absolute data, depending on addressing mode

rel = The relative offset, which is the two's complement number stored in the last byte of machine code corresponding to a branch instruction

X = Pseudo index register, memory location $000F

,X or D[X] = Memory location $000E pointing to the memory location defined by the pseudo index register (location $000F)

Condition code register (CCR) bits

Z = Zero indicator

C = Carry/borrow

Bit status before execution of an instruction (*n* = 7, 6, 5, ... 0)

M*n* = Bit *n* of memory location used in operation

A*n* = Bit *n* of accumulator

b*n* = Bit *n* of the source operand (M, or A)

Bit status after execution of an instruction (*n* = 7, 6, 5, … 0)

R*n* = Bit *n* of the result of an operation

CCR activity figure notation

– = Bit not affected

0 = Bit forced to 0

1 = Bit forced to 1

↕ = Bit set or cleared according to results of operation

U = Undefined after the operation

Machine coding notation

dd = Low-order eight bits of a direct address $0000–$00FF (high byte assumed to be $00)

ii = One byte of immediate data

kk	=	Low-order byte of a 16-bit immediate data value
hh	=	High-order byte of 16-bit extended address
ll	=	Low-order byte of 16-bit extended address
rr	=	Relative offset

Source forms

The instruction detail pages provide only essential information about assembler source forms. Assemblers generally support a number of assembler directives, allow definition of program labels, and have special conventions for comments. For complete information about writing source files for a particular assembler, refer to the documentation provided by the assembler vendor.

Typically, assemblers are flexible about the use of spaces and tabs. Often, any number of spaces or tabs can be used where a single space is shown on the glossary pages. Spaces and tabs are also normally allowed before and after commas. When program labels are used, there must also be at least one tab or space before all instruction mnemonics. This required space is not apparent in the source forms.

Everything in the source forms columns, *except expressions in italic characters*, is literal information which must appear in the assembly source file exactly as shown. The initial 3- to 5-letter mnemonic is always a literal expression. All commas, pound signs (#), parentheses, and plus signs (+) are literal characters.

The definition of a legal label or expression varies from assembler to assembler. Assemblers also vary in the way CPU registers are specified. Refer to assembler documentation for detailed information. The recommended register designator is ‘A’.

n	—	Any label or expression that evaluates to a single integer in the range 0–7
x	—	Any label or expression that evaluates to a single hexadecimal integer in the range $0–$F
opr8i	—	Any label or expression that evaluates to an 8-bit immediate value

opr4a — Any label or expression that evaluates to a Tiny address (4-bit value). The instruction treats this 4-bit value as the low order four bits of an address in the 16-Kbyte address space ($0000–$000F). This 4-bit is embedded in the low order 4 bits in the opcode.

opr5a — Any label or expression that evaluates to a Short address (5-bit value). The instruction treats this 5-bit value as the low order five bits of an address in the 16-Kbyte address space ($0000–$001F). This 5-bit value is embedded in the low order 5 bits in the opcode.

opr8a — Any label or expression that evaluates to an 8-bit value. The instruction treats this 8-bit value as the low order eight bits of an address in the 16-Kbyte address space ($0000–$00FF).

opr16a — Any label or expression that evaluates to a 16-bit value. The instruction treats this value as an address in the 64-Kbyte address space.

rel — Any label or expression that refers to an address that is within –128 to +127 locations from the next address after the last byte of object code for the current instruction. The assembler will calculate the 8-bit signed offset and include it in the object code for this instruction.

Cycle-by-cycle execution

This information is found in the tables at the bottom of each instruction glossary page. Entries show how many bytes of information are accessed from different areas of memory during the course of instruction execution. With this information and knowledge of the bus frequency, a user can determine the execution time for any instruction in any system.

A single letter code in the column represents a single CPU cycle. There are cycle codes for each addressing mode variation of each instruction. Simply count code letters to determine the execution time of an instruction.

This list explains the cycle-by-cycle code letters:

f — Free cycle. This indicates a cycle where the CPU does not require use of the system buses. An f cycle is always one cycle of the system bus clock.

p — Program byte access

r — 8-bit data read

w — 8-bit data write

Address modes

INH	=	Inherent (no operands)
IMD	=	Immediate to Direct (in MOV instruction)
IMM	=	Immediate
DD	=	Direct to Direct (in MOV instruction)
DIR	=	Direct
SRT	=	Short
TNY	=	Tiny
EXT	=	Extended
REL	=	8-bit relative offset

A.3 Convention Definitions

Set refers specifically to establishing logic level 1 on a bit or bits.

Cleared refers specifically to establishing logic level 0 on a bit or bits.

A specific bit is referred to by mnemonic and bit number. A7 is bit 7 of accumulator A. **A range of bits** is referred to by mnemonic and the bit numbers that define the range. A [7:4] are bits 7 to 4 of the accumulator.

Parentheses indicate the contents of a register or memory location, rather than the register or memory location itself. (A) is the contents of the accumulator. In Boolean expressions, parentheses have the traditional mathematical meaning.

A.4 Use of ‘X’, ‘,X’ and ‘D[X]’ as instruction operands

In an RS08 assembler X is defined by the assembler as being address $000F, the index notation ,X or D[X] is defined as location $000E.

The use of ,X is supported to aid assembly language compatibility with the notation used with the HC(S)08 assembly language.

- Where instructions use the ,X notation the use of D[X] is permissible
- The use of X will refer to location $000F unless it is the first operand and is prefixed by a comma where it will refer to location $000E

Valid uses of ‘X’, ‘,X’ and ‘D[X’] are shown in Table 3-3.

Table 3-3. Valid Uses of X, ',X' & D[X]

Instruction	Use
MOV #$20,X	Load index register with $20
MOV #$AA,D[X]	Store $AA into location pointed to by index register
LDA ,X LDA D[X]	Load accumulator from location pointed to by index register

A.5 Instruction Set

The following pages summarize each instruction, including operation and description, condition codes and Boolean formulae, and a table with source forms, addressing modes, machine code, and cycles.

ADC — Add with Carry — ADC

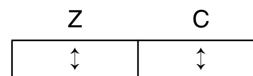
Operation

A ← (A) + (M) + (C)

Description

Adds the contents of the C bit to the sum of the contents of A and M and places the result in A. This operation is useful for addition of operands that are larger than eight bits.

Condition Codes and Boolean Formulae



Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

C: $A7\&M7 \mid M7\&\overline{R7} \mid \overline{R7}\&A7$
Set if there was a carry from the most significant bit (MSB) of the result; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code Opcode	Machine Code Operand(s)	RS08 Cycles	Access Detail
ADC *#opr8i*	IMM	A9	ii	2	pp
ADC *opr8a*	DIR	B9	dd	3	rpp

Pseudo Instruction

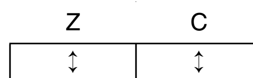
Source Form	Addr. Mode	Machine Code Opcode	Machine Code Operand(s)	RS08 Cycles	Access Detail
ADC ,X	IX	B9	0E	3	rpp
ADC X	DIR	B9	0F	3	rpp

ADD — Add without Carry — ADD

Operation A ← (A) + (M)

Description Adds the contents of M to the contents of A and places the result in A

Condition Codes and Boolean Formulae



Z: $\overline{R7}$&$\overline{R6}$&$\overline{R5}$&$\overline{R4}$&$\overline{R3}$&$\overline{R2}$&$\overline{R1}$&$\overline{R0}$
Set if result is $00; cleared otherwise

C: A7&M7 | M7&$\overline{R7}$ | $\overline{R7}$&A7
Set if there was a carry from the MSB of the result; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
ADD *#opr8i*	IMM	AB	ii	2	pp
ADD *opr8a*	DIR	BB	dd	3	rpp
ADD *opr4a*	TNY	6*x*		3	rfp

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
ADD ,X	IX	6E		3	rfp
ADD X	DIR	6F		3	rfp

AND Logical AND AND

Operation $A \leftarrow (A) \& (M)$

Description

Performs the logical AND between the contents of A and the contents of M and places the result in A. Each bit of A after the operation will be the logical AND of the corresponding bits of M and of A before the operation.

Condition Codes and Boolean Formulae

Z	C
↕	—

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

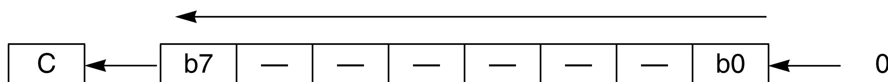
Source Form		Addr. Mode	Machine Code		RS08 Cycles	Access Detail
			Opcode	Operand(s)		
AND	*#opr8i*	IMM	A4	ii	2	pp
AND	*opr8a*	DIR	B4	dd	3	rpp

Pseudo Instruction

Source Form		Addr. Mode	Machine Code		RS08 Cycles	Access Detail
			Opcode	Operand(s)		
AND	,X	IX	B4	0E	3	rpp
AND	X	DIR	B4	0F	3	rpp

ASLA — Arithmetic Shift Left (Pseudo Instruction) — ASLA

Operation



Description

Shifts all bits of the A one place to the left. Bit 0 is loaded with a 0. The C bit in the CCR is loaded from the most significant bit of A.

Condition Codes and Boolean Formulae

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

C: b7
Set if, before the shift, the MSB of A was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
ASLA	INH	48		1	p

BCC Branch if Carry Bit Clear (Same as BHS) BCC

Operation

If (C) = 0, PC ← (PC) + $0002 + *rel*

Description

Tests state of C bit in CCR and causes a branch if C is clear. BCC can be used after shift or rotate instructions or to check for overflow after operations on unsigned numbers. See the BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BCC *rel*	REL	34	rr	3	ppp

See the BRA instruction for a summary of all branches and their complements.

BCLR *n* — Clear Bit *n* in Memory — BCLR *n*

Operation

M*n* ← 0

Description

Clear bit *n* (*n* = 7, 6, 5, … 0) in location M. All other bits in M are unaffected. In other words, M can be any random-access memory (RAM) or input/output (I/O) register address from $0000 to $00FF. This instruction reads the specified 8-bit location, modifies the specified bit, and then writes the modified 8-bit value back to the memory location.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BCLR 0,*opr8a*	DIR (b0)	11	dd	5	rfwpp
BCLR 1,*opr8a*	DIR (b1)	13	dd	5	rfwpp
BCLR 2,*opr8a*	DIR (b2)	15	dd	5	rfwpp
BCLR 3,*opr8a*	DIR (b3)	17	dd	5	rfwpp
BCLR 4,*opr8a*	DIR (b4)	19	dd	5	rfwpp
BCLR 5,*opr8a*	DIR (b5)	1B	dd	5	rfwpp
BCLR 6,*opr8a*	DIR (b6)	1D	dd	5	rfwpp
BCLR 7,*opr8a*	DIR (b7)	1F	dd	5	rfwpp

The BCLR n description continues next page.

BCLR *n* Clear Bit *n* in Memory (Continued) BCLR *n*

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BCLR 0,D[X]	IX (b0)	11	0E	5	rfwpp
BCLR 1,D[X]	IX (b1)	13	0E	5	rfwpp
BCLR 2,D[X]	IX (b2)	15	0E	5	rfwpp
BCLR 3,D[X]	IX (b3)	17	0E	5	rfwpp
BCLR 4,D[X]	IX (b4)	19	0E	5	rfwpp
BCLR 5,D[X]	IX (b5)	1B	0E	5	rfwpp
BCLR 6,D[X]	IX (b6)	1D	0E	5	rfwpp
BCLR 7,D[X]	IX (b7)	1F	0E	5	rfwpp
BCLR 0,X	DIR (b0)	11	0F	5	rfwpp
BCLR 1,X	DIR (b1)	13	0F	5	rfwpp
BCLR 2,X	DIR (b2)	15	0F	5	rfwpp
BCLR 3,X	DIR (b3)	17	0F	5	rfwpp
BCLR 4,X	DIR (b4)	19	0F	5	rfwpp
BCLR 5,X	DIR (b5)	1B	0F	5	rfwpp
BCLR 6,X	DIR (b6)	1D	0F	5	rfwpp
BCLR 7,X	DIR (b7)	1F	0F	5	rfwpp

BCS — Branch if Carry Bit Set (Same as BLO) — BCS

Operation

If (C) = 1, PC ← (PC) + $0002 + *rel*

Description

Tests the state of the C bit in the CCR and causes a branch if C is set. BCS can be used after shift or rotate instructions or to check for overflow after operations on unsigned numbers. See the BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BCS *rel*	REL	35	rr	3	ppp

See the BRA instruction for a summary of all branches and their complements.

BEQ — Branch if Equal — BEQ

Operation

If (Z) = 1, PC ← (PC) + $0002 + *rel*

Description

Tests the state of the Z bit in the CCR and causes a branch if Z is set. Compare instructions perform a subtraction with two operands and produce an internal result without changing the original operands. If the two operands were equal, the internal result of the subtraction for the compare will be zero so the Z bit will be equal to one and the BEQ will cause a branch.

This instruction can also be used after a load or store without having to do a separate test or compare on the loaded value. See the BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BEQ *rel*	REL	37	rr	3	ppp

See the BRA instruction for a summary of all branches and their complements.

BGND Background BGND

Operation Enter active background debug mode (if allowed by ENBDM = 1)

Description This instruction is used to establish software breakpoints during debug by replacing a user opcode with this opcode. BGND causes the user program to stop and the CPU enters active background mode (provided it has been enabled previously by a serial WRITE_CONTROL command from a host debug pod). The CPU remains in active background mode until the debug host sends a serial GO or TRACE1 command to return to the user program. This instruction is never used in normal user application programs. If the ENBDM control bit in the BDC status/control register is clear, BGND is treated as an illegal opcode.

Condition Codes and Boolean Formulae None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BGND	INH	BF		5+	fff...ppp

BHS Branch if Higher or Same (Pseudo Instruction, same as BCC) BHS

Operation

If (C) = 0, PC ← (PC) + \$0002 + *rel*

Description

If the BHS instruction is executed immediately after execution of a CMP, SBC, or SUB instruction, the branch will occur if the unsigned binary number in the A register was greater than or equal to the unsigned binary number in memory. Generally not useful after CLR, COM, DEC, INC, LDA, or STA because these instructions do not affect the carry bit in the CCR. See the BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BHS *rel*	REL	34	rr	3	ppp

See the BRA instruction for a summary of all branches and their complements.

BLO Branch if Lower BLO

(Pseudo Instruction, same as BCS)

Operation

If (C) = 1, PC ← (PC) + \$0002 + *rel*

For unsigned values, if (Accumulator) < (Memory), then branch

Description

If the BLO instruction is executed immediately after execution of a CMP, SBC, or SUB instruction, the branch will occur if the unsigned binary number in the A register was less than the unsigned binary number in memory. Generally not useful after CLR, COM, DEC, INC, LDA, or STA because these instructions do not affect the carry bit in the CCR. See the BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BLO *rel*	REL	35	rr	3	ppp

See the BRA instruction for a summary of all branches and their complements.

BNE Branch if Not Equal BNE

Operation

If (Z) = 0, PC ← (PC) + $0002 + *rel*

Description

Tests the state of the Z bit in the CCR and causes a branch if Z is clear.

Following a compare or subtract instruction, the branch will occur if the arguments were not equal. This instruction can also be used after a load or store without having to do a separate test or compare on the loaded value. See the BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BNE *rel*	REL	36	rr	3	ppp

See the BRA instruction for a summary of all branches and their complements.

BRA — Branch Always — BRA

Operation

PC ← (PC) + \$0002 + *rel*

Description

Performs an unconditional branch to the address given in the foregoing formula. In this formula, *rel* is the two's-complement relative offset in the last byte of machine code for the instruction and (PC) is the address of the opcode for the branch instruction.

A source program specifies the destination of a branch instruction by its absolute address, either as a numerical value or as a symbol or expression which can be numerically evaluated by the assembler. The assembler calculates the 8-bit relative offset *rel* from this absolute address and the current value of the location counter.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BRA *rel*	REL	30	rr	3	ppp

Table A-1 is a summary of all branch instructions.

The BRA description continues next page.

BRA Branch Always (Continued) BRA

Branch Instruction Summary

Table A-1 is a summary of all branch instructions.

Table A-1. Branch Instruction Summary

Branch				Complementary Branch			Type
Test	Boolean	Mnemonic	Opcode	Test	Mnemonic	Opcode	
r≥m	(C)=0	BHS/BCC	34	r<m	BLO/BCS	35	Unsigned
r=m	(Z)=1	BEQ	37	r≠m	BNE	36	Unsigned
r<m	(C)=1	BLO/BCS	35	r≥m	BHS/BCC	34	Unsigned
Carry	(C)=1	BCS	35	No carry	BCC	34	Simple
result=0	(Z)=1	BEQ	37	result≠0	BNE	36	Simple
Always	—	BRA	30	—	BRN	30 00	Uncond

r = register: A m = memory operand

During program execution, if the tested condition is true, the two's complement offset is sign-extended to a 14-bit value, which is added to the current program counter. This causes program execution to continue at the address specified as the branch destination. If the tested condition is not true, the program simply continues to the next instruction after the branch.

BRN Branch Never BRN

Operation PC ← (PC) + $0002

Description Never branches. In effect, this instruction can be considered a 2-byte no operation (NOP) requiring three cycles for execution. Its inclusion in the instruction set provides a complement for the BRA instruction. The BRN instruction is useful during program debugging to negate the effect of another branch instruction without disturbing the offset byte.

This instruction can be useful in instruction-based timing delays. Instruction-based timing delays are usually discouraged because such code is not portable to systems with different clock speeds.

See the BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BRN *rel*	REL	30	00	3	ppp

See the BRA instruction for a summary of all branches and their complements.

BRCLR *n* — Branch if Bit *n* in Memory Clear — BRCLR *n*

Operation

If bit *n* of M = 0, PC ← (PC) + $0003 + *rel*

Description

Tests bit *n* (*n* = 7, 6, 5, … 0) of location M and branches if the bit is clear. M can be any RAM or I/O register address in range from $0000 to $00FF because direct addressing mode is used to specify the address of the operand.

The C bit is set to the state of the tested bit. When used with an appropriate rotate instruction, BRCLR *n* provides an easy method for performing serial-to-parallel conversions.

Condition Codes and Boolean Formulae

Z	C
—	↕

C: Set if M*n* = 1; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BRCLR 0,*opr8a*,*rel*	DIR (b0)	01	dd rr	5	rpppp
BRCLR 1,*opr8a*,*rel*	DIR (b1)	03	dd rr	5	rpppp
BRCLR 2,*opr8a*,*rel*	DIR (b2)	05	dd rr	5	rpppp
BRCLR 3,*opr8a*,*rel*	DIR (b3)	07	dd rr	5	rpppp
BRCLR 4,*opr8a*,*rel*	DIR (b4)	09	dd rr	5	rpppp
BRCLR 5,*opr8a*,*rel*	DIR (b5)	0B	dd rr	5	rpppp
BRCLR 6,*opr8a*,*rel*	DIR (b6)	0D	dd rr	5	rpppp
BRCLR 7,*opr8a*,*rel*	DIR (b7)	0F	dd rr	5	rpppp

The BRCLR description continues next page.

BRCLR *n* Branch if Bit *n* in Memory Clear (Continued) BRCLR *n*

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BRCLR 0,D[X],*rel*	IX (b0)	01	0E rr	5	rpppp
BRCLR 1,D[X],*rel*	IX (b1)	03	0E rr	5	rpppp
BRCLR 2,D[X],*rel*	IX (b2)	05	0E rr	5	rpppp
BRCLR 3,D[X],*rel*	IX (b3)	07	0E rr	5	rpppp
BRCLR 4,D[X],*rel*	IX (b4)	09	0E rr	5	rpppp
BRCLR 5,D[X],*rel*	IX (b5)	0B	0E rr	5	rpppp
BRCLR 6,D[X],*rel*	IX (b6)	0D	0E rr	5	rpppp
BRCLR 7,D[X],*rel*	IX (b7)	0F	0E rr	5	rpppp
BRCLR 0,X,*rel*	DIR (b0)	01	0F rr	5	rpppp
BRCLR 1,X,*rel*	DIR (b1)	03	0F rr	5	rpppp
BRCLR 2,X,*rel*	DIR (b2)	05	0F rr	5	rpppp
BRCLR 3,X,*rel*	DIR (b3)	07	0F rr	5	rpppp
BRCLR 4,X,*rel*	DIR (b4)	09	0F rr	5	rpppp
BRCLR 5,X,*rel*	DIR (b5)	0B	0F rr	5	rpppp
BRCLR 6,X,*rel*	DIR (b6)	0D	0F rr	5	rpppp
BRCLR 7,X,*rel*	DIR (b7)	0F	0F rr	5	rpppp

BRSET *n* — Branch if Bit *n* in Memory Set — BRSET *n*

Operation

If bit *n* of M = 1, PC ← (PC) + $0003 + *rel*

Description

Tests bit *n* (*n* = 7, 6, 5, … 0) of location M and branches if the bit is set. M can be any RAM or I/O register address in the $0000 to $00FF area of memory because direct addressing mode is used to specify the address of the operand.

The C bit is set to the state of the tested bit. When used with an appropriate rotate instruction, BRSET *n* provides an easy method for performing serial-to-parallel conversions.

Condition Codes and Boolean Formulae

Z	C
—	↕

C: Set if M*n* = 1; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BRSET 0,*opr8a*,*rel*	DIR (b0)	00	dd rr	5	rpppp
BRSET 1,*opr8a*,*rel*	DIR (b1)	02	dd rr	5	rpppp
BRSET 2,*opr8a*,*rel*	DIR (b2)	04	dd rr	5	rpppp
BRSET 3,*opr8a*,*rel*	DIR (b3)	06	dd rr	5	rpppp
BRSET 4,*opr8a*,*rel*	DIR (b4)	08	dd rr	5	rpppp
BRSET 5,*opr8a*,*rel*	DIR (b5)	0A	dd rr	5	rpppp
BRSET 6,*opr8a*,*rel*	DIR (b6)	0C	dd rr	5	rpppp
BRSET 7,*opr8a*,*rel*	DIR (b7)	0E	dd rr	5	rpppp

The BRSET description continues next page.

BRSET *n* Branch if Bit *n* in Memory Set (Continued) BRSET *n*

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BRSET 0,D[X],*rel*	IX (b0)	00	0E rr	5	rpppp
BRSET 1,D[X],*rel*	IX (b1)	02	0E rr	5	rpppp
BRSET 2,D[X],*rel*	IX (b2)	04	0E rr	5	rpppp
BRSET 3,D[X],*rel*	IX (b3)	06	0E rr	5	rpppp
BRSET 4,D[X],*rel*	IX (b4)	08	0E rr	5	rpppp
BRSET 5,D[X],*rel*	IX (b5)	0A	0E rr	5	rpppp
BRSET 6,D[X],*rel*	IX (b6)	0C	0E rr	5	rpppp
BRSET 7,D[X],*rel*	IX (b7)	0E	0E rr	5	rpppp
BRSET 0,X,*rel*	DIR (b0)	00	0F rr	5	rpppp
BRSET 1,X,*rel*	DIR (b1)	02	0F rr	5	rpppp
BRSET 2,X,*rel*	DIR (b2)	04	0F rr	5	rpppp
BRSET 3,X,*rel*	DIR (b3)	06	0F rr	5	rpppp
BRSET 4,X,*rel*	DIR (b4)	08	0F rr	5	rpppp
BRSET 5,X,*rel*	DIR (b5)	0A	0F rr	5	rpppp
BRSET 6,X,*rel*	DIR (b6)	0C	0F rr	5	rpppp
BRSET 7,X,*rel*	DIR (b7)	0F	0E rr	5	rpppp

BSET *n* — Set Bit *n* in Memory — BSET *n*

Operation

M*n* ← 1

Description

Set bit *n* (*n* = 7, 6, 5, … 0) in location M. All other bits in M are unaffected. In other words, M can be any random-access memory (RAM) or input/output (I/O) register address from $0000 to $00FF. This instruction reads the specified 8-bit location, modifies the specified bit, and then writes the modified 8-bit value back to the memory location.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BSET 0,*opr8a*	DIR (b0)	10	dd	5	rfwpp
BSET 1,*opr8a*	DIR (b1)	12	dd	5	rfwpp
BSET 2,*opr8a*	DIR (b2)	14	dd	5	rfwpp
BSET 3,*opr8a*	DIR (b3)	16	dd	5	rfwpp
BSET 4,*opr8a*	DIR (b4)	18	dd	5	rfwpp
BSET 5,*opr8a*	DIR (b5)	1A	dd	5	rfwpp
BSET 6,*opr8a*	DIR (b6)	1C	dd	5	rfwpp
BSET 7,*opr8a*	DIR (b7)	1E	dd	5	rfwpp

The BSET description continues next page.

BSET *n* Set Bit *n* in Memory (Continued) BSET *n*

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BSET 0,D[X]	IX (b0)	10	0E	5	rfwpp
BSET 1,D[X]	IX (b1)	12	0E	5	rfwpp
BSET 2,D[X]	IX (b2)	14	0E	5	rfwpp
BSET 3,D[X]	IX (b3)	16	0E	5	rfwpp
BSET 4,D[X]	IX (b4)	18	0E	5	rfwpp
BSET 5,D[X]	IX (b5)	1A	0E	5	rfwpp
BSET 6,D[X]	IX (b6)	1C	0E	5	rfwpp
BSET 7,D[X]	IX (b7)	1E	0E	5	rfwpp
BSET 0,X	DIR (b0)	10	0F	5	rfwpp
BSET 1,X	DIR (b1)	12	0F	5	rfwpp
BSET 2,X	DIR (b2)	14	0F	5	rfwpp
BSET 3,X	DIR (b3)	16	0F	5	rfwpp
BSET 4,X	DIR (b4)	18	0F	5	rfwpp
BSET 5,X	DIR (b5)	1A	0F	5	rfwpp
BSET 6,X	DIR (b6)	1C	0F	5	rfwpp
BSET 7,X	DIR (b7)	1E	0F	5	rfwpp

BSR Branch to Subroutine BSR

Operation

PC ← (PC) + $0002 Advance PC to return address
SPC ← (PC) Save current PC to SPC
PC ← (PC) + *rel* Load PC with start address of requested subroutine

Description

The program counter is incremented by 2 from the opcode address (so it points to the opcode of the next instruction, which will be the return address). The program counter is saved into the shadow PC. A branch then occurs to the location specified by the branch offset. See the BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
BSR *rel*	REL	AD	rr	3	ppp

CBEQ — Compare and Branch if Equal — CBEQ

Operation

For DIR or IMM modes: if (A) = (M), PC ← (PC) + $0003 + *rel*

Description

CBEQ compares the operand with the accumulator against the contents of a memory location and causes a branch if the accumulator is equal to the memory contents. The CBEQ instruction combines CMP and BEQ for faster table lookup routines and condition codes are not changed.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form		Addr. Mode	Machine Code			RS08 Cycles	Access Detail
			Opcode	Operand(s)			
CBEQ	*opr8a,rel*	DIR	31	dd	rr	5	rpppp
CBEQA	#*opr8i,rel*	IMM	41	ii	rr	4	pppp

Pseudo Instruction

Source Form		Addr. Mode	Machine Code			RS08 Cycles	Access Detail
			Opcode	Operand(s)			
CBEQ	*,X,rel* (1)	IX	31	0E	rr	5	rpppp
CBEQX	X,*rel*	DIR	31	0F	rr	5	rpppp

1. Auto-increment on X register is not available in this pseudo instruction. But in the HCS08 CBEQ *,X+,rel* instruction, H:X is auto-incremented after the compare operation.

CLC Clear Carry Bit CLC

Operation

C bit ← 0

Description

Clears the C bit in the CCR. CLC may be used to set up the C bit prior to a shift or rotate instruction that involves the C bit. The C bit can also be used to pass status information between a subroutine and the calling program.

Condition Codes and Boolean Formulae

Z	C
—	0

C: 0
Cleared

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
CLC	INH	38		1	p

CLR Clear CLR

Operation

A ← $00
Or M ← $00

Description

The contents of memory (M), or A are replaced with zeros.

Condition Codes and Boolean Formulae

Z	C
1	—

Z: 1
Set

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
CLR *opr8a*	DIR	3F	dd	3	wpp
CLR *opr5a*	SRT	8*x* / 9*x*		2	wp
CLRA	INH	4F		1	p

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
CLR *,X*	IX	8E		2	wp
CLRX	INH	8F		2	wp

CMP Compare Accumulator with Memory CMP

Operation

(A) – (M)

Description

Compares the contents of A to the contents of M and sets the condition codes, which may then be used for arithmetic (signed or unsigned) and logical conditional branching. The contents of both A and M are unchanged.

Condition Codes and Boolean Formulae

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

C: $\overline{A7}\&M7 \mid M7\&R7 \mid R7\&\overline{A7}$
Set if the unsigned value of the contents of memory is larger than the unsigned value of the accumulator; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
CMP *#opr8i*	IMM	A1	ii	2	pp
CMP *opr8a*	DIR	B1	dd	3	rpp

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
CMP *,X*	IX	B1	0E	3	rpp
CMP X	INH	B1	0F	3	rpp

COMA Complement (One's Complement) COMA

Operation

$A \leftarrow \overline{A} = \$FF - (A)$

Description

Replaces the contents of A with the one's complement. Each bit of A is replaced with the complement of that bit.

Condition Codes and Boolean Formulae

Z	C
↕	1

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: 1
Set

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
COMA	INH	43		1	p

DBNZ — Decrement and Branch if Not Zero — DBNZ

Operation

A ← (A) – $01
Or M ← (M) – $01
For DIR mode: PC ← (PC) + $0003 + *rel* if (result) ≠ 0
Or for INH mode: PC ← (PC) + $0002 + *rel* if (result) ≠ 0

Description

Subtract 1 from the contents of A, or M; then branch using the relative offset if the result of the subtraction is not $00.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code Opcode	Machine Code Operand(s)	RS08 Cycles	Access Detail
DBNZ *opr8a*,*rel*	DIR	3B	dd rr	7	rfwpppp
DBNZA *rel*	INH	4B	rr	4	fppp

Pseudo Instruction

Source Form	Addr. Mode	Machine Code Opcode	Machine Code Operand(s)	RS08 Cycles	Access Detail
DBNZ *,X*,*rel*	IX	3B	0E rr	7	rfwpppp
DBNZX *rel*	DIR	3B	0F rr	7	rfwpppp

DEC Decrement DEC

Operation

A ← (A) – $01
Or M ← (M) – $01

Description

Subtract 1 from the contents of A, or M. The Z bit in the CCR is set or cleared according to the results of this operation. The C bit in the CCR is not affected; therefore, the BHS and BLO branch instructions are not useful following a DEC instruction.

Condition Codes and Boolean Formulae

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
DEC *opr8a*	DIR	3A	dd	5	rfwpp
DEC *opr4a*	TNY	5*x*		4	rfwp
DECA	INH	4A		1	p

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
DEC *,X*	IX	5E		4	rfwp
DECX	INH	5F		4	rfwp

EOR Exclusive-OR Memory with Accumulator EOR

Operation

$A \leftarrow (A \oplus M)$

Description

Performs the logical exclusive-OR between the contents of A and the contents of M and places the result in A. Each bit of A after the operation will be the logical exclusive-OR of the corresponding bits of M and A before the operation.

Condition Codes and Boolean Formulae

Z	C
↕	—

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form		Addr. Mode	Machine Code		RS08 Cycles	Access Detail
			Opcode	Operand(s)		
EOR	*#opr8i*	IMM	A8	ii	2	pp
EOR	*opr8a*	DIR	B8	dd	3	rpp

Pseudo Instruction

Source Form		Addr. Mode	Machine Code		RS08 Cycles	Access Detail
			Opcode	Operand(s)		
EOR	*,X*	IX	B8	0E	3	rpp
EOR	X	DIR	B8	0F	3	rpp

INC Increment INC

Operation

A ← (A) + $01
Or M ← (M) + $01

Description

Add 1 to the contents of A or M. The Z bit in the CCR is set or cleared according to the results of this operation. The C bit in the CCR is not affected; therefore, the BHS and BLO branch instructions are not useful following an INC instruction.

Condition Codes and Boolean Formulae

Z	C
↕	—

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
INC *opr8a*	DIR	3C	dd	5	rfwpp
INC *opr4a*	TNY	2*x*		4	rfwp
INCA	INH	4C		1	p

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
INC *,X*	IX	2E		4	rfwp
INCX	INH	2F		4	rfwp

JMP Jump JMP

Operation PC ← effective address

Description A jump to the instruction stored at the effective address occurs. The effective address is obtained according to the rules for extended addressing.

Condition Codes and Boolean Formulae None affected

Z	C
—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
JMP *opr16a*	EXT	BC	hh ll	4	fppp

JSR Jump to Subroutine JSR

Operation

PC ← (PC) + *3*;
SPC ← (PC) Save current PC to SPC
PC ← effective address Load PC with start address of requested subroutine

Description

The program counter is incremented by *3* so that it points to the opcode of the next instruction that follows the JSR instruction. The PC is then saved to the SPC. A jump to the instruction stored at the effective address occurs. The effective address is obtained according to the rules for extended addressing.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
JSR *opr16a*	EXT	BD	hh ll	4	fppp

LDA — Load Accumulator from Memory — LDA

Operation

A ← (M)

Description

Loads the contents of the specified memory location into A. The Z condition code is set or cleared according to the loaded data. This allows conditional branching after the load without having to perform a separate test or compare.

Condition Codes and Boolean Formulae

Z	C
↕	—

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form		Addr. Mode	Machine Code		RS08 Cycles	Access Detail
			Opcode	Operand(s)		
LDA	*#opr8i*	IMM	A6	ii	2	pp
LDA	*opr8a*	DIR	B6	dd	3	rpp
LDA	*opr5a*	SRT	C*x* / D*x*		3	rfp

Pseudo Instruction

Source Form		Addr. Mode	Machine Code		RS08 Cycles	Access Detail
			Opcode	Operand(s)		
LDA	*,X*	IX	CE		3	rfp

LDX Load X Index Register from Memory (Pseudo Instruction) LDX

Operation

X ← (M)

Description

This is a pseudo instruction implemented by using a MOV instruction that loads the contents of the specified memory location into the memory mapped X index register. The Z condition code is set or cleared according to the loaded data. This allows conditional branching after the load without having to perform a separate test or compare.

Condition Codes and Boolean Formulae

Z	C
↕	—

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

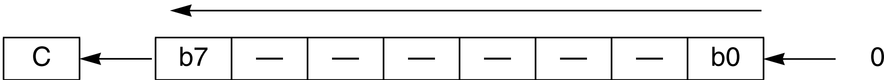
Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
LDX *#opr8i*	IMM	3E	ii 0F	4	pwpp
LDX *opr8a*	DIR	4E	dd 0F	5	rpwpp
LDX *,X*	IX	4E	0E 0F	5	rpwpp

LSLA Logical Shift Left LSLA

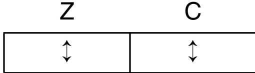
Operation



Description

Shifts all bits of the A one place to the left. Bit 0 is loaded with a 0. The C bit in the CCR is loaded from the most significant bit of A.

Condition Codes and Boolean Formulae



Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

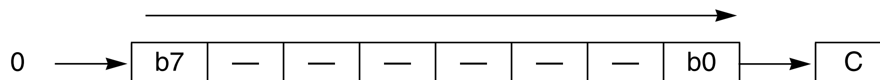
C: b7
Set if, before the shift, the MSB of A was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
LSLA	INH	48		1	p

LSRA Logical Shift Right LSRA

Operation



Description

Shifts all bits of A one place to the right. Bit 7 is loaded with a 0. Bit 0 is shifted into the C bit.

Condition Codes and Boolean Formulae

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

C: b0
Set if, before the shift, the LSB of A, was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
LSRA	INH	44		1	p

MOV **Move** MOV

Operation $(M)_{Destination} \leftarrow (M)_{Source}$

Description Moves a byte of data from a source address to a destination address. Data is examined as it is moved, and condition codes are set. Source data is not changed. The accumulator is not affected.

The two addressing modes for the MOV instruction are:

1. IMM/DIR moves an immediate byte to a direct memory location.
2. DIR/DIR moves a direct location byte to another direct location.

Condition Codes and Boolean Formulae

Z	C
↕	—

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form		Addr. Mode	Machine Code			RS08 Cycles	Access Detail
			Opcode	Operand(s)			
MOV	*opr8a*,*opr8a*	DD	4E	dd	dd	5	rpwpp
MOV	#*opr8i*,*opr8a*	IMD	3E	ii	dd	4	pwpp
MOV	D[X],*opr8a*	IX/DIR	4E	0E	dd	5	rpwpp
MOV	*opr8a*,D[X]	DIR/IX	4E	dd	0E	5	rpwpp
MOV	#*opr8i*,D[X]	IMM/IX	3E	ii	0E	4	pwpp

NOP — No Operation — NOP

Operation

Uses one bus cycle

Description

This is a single-byte instruction that does nothing except to consume one CPU bus cycle while the program counter is advanced to the next instruction. No register or memory contents are affected by this instruction.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
NOP	INH	AC		1	p

ORA — Inclusive-OR Accumulator and Memory — ORA

Operation

A ← (A) | (M)

Description

Performs the logical inclusive-OR between the contents of A and the contents of M and places the result in A. Each bit of A after the operation will be the logical inclusive-OR of the corresponding bits of M and A before the operation.

Condition Codes and Boolean Formulae

Z	C
↕	—

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

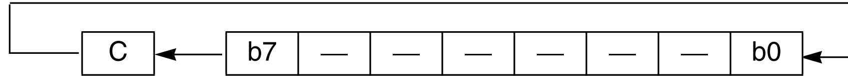
Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
ORA *#opr8i*	IMM	AA	ii	2	pp
ORA *opr8a*	DIR	BA	dd	3	rpp

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
ORA *,X*	IX	BA	0E	3	rpp
ORA X	DIR	BA	0F	3	rpp

ROLA Rotate Left through Carry ROLA

Operation



Description

Shifts all bits of A one place to the left. Bit 0 is loaded from the C bit. The C bit is loaded from the most significant bit of A. The rotate instructions include the carry bit to allow extension of the shift and rotate instructions to multiple bytes.

Condition Codes and Boolean Formulae

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

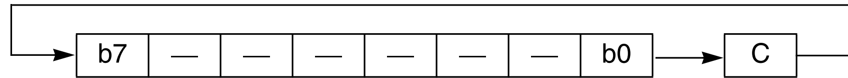
C: b7
Set if, before the rotate, the MSB of A was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
ROLA	INH	49		1	p

RORA Rotate Right through Carry RORA

Operation



Description

Shifts all bits of A one place to the right. Bit 7 is loaded from the C bit. Bit 0 is shifted into the C bit. The rotate instructions include the carry bit to allow extension of the shift and rotate instructions to multiple bytes.

Condition Codes and Boolean Formulae

Z	C
↕	↕

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: b0
Set if, before the shift, the LSB of A was set; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
RORA	INH	46		1	p

RTS — Return from Subroutine — RTS

Operation PC ← SPC Restore PC from SPC

Description The contents of the shadow program counter is restored to the program counter. Program execution resumes at the address that was just restored from the shadow program counter.

Condition Codes and Boolean Formulae None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
RTS	INH	BE		3	ppp

SBC Subtract with Carry SBC

Operation $A \leftarrow (A) - (M) - (C)$

Description Subtracts the contents of M and the contents of the C bit of the CCR from the contents of A and places the result in A. This is useful for multi-precision subtract algorithms involving operands with more than eight bits.

Condition Codes and Boolean Formulae

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is \$00; cleared otherwise

C: $\overline{A7}\&M7 \mid M7\&R7 \mid R7\&\overline{A7}$
Set if the unsigned value of the contents of memory plus the previous carry are larger than the unsigned value of the accumulator; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form		Addr. Mode	Machine Code		RS08 Cycles	Access Detail
			Opcode	Operand(s)		
SBC	*#opr8i*	IMM	A2	ii	2	pp
SBC	*opr8a*	DIR	B2	dd	3	rpp

Pseudo Instruction

Source Form		Addr. Mode	Machine Code		RS08 Cycles	Access Detail
			Opcode	Operand(s)		
SBC	*,X*	IX	B2	0E	3	rpp
SBC	X	DIR	B2	0F	3	rpp

SEC Set Carry Bit SEC

Operation

C bit ← 1

Description

Sets the C bit in the condition code register (CCR). SEC may be used to set up the C bit prior to a shift or rotate instruction that involves the C bit.

Condition Codes and Boolean Formulae

Z	C
—	1

C: 1
Set

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
SEC	INH	39		1	p

SHA — Swap Shadow PC High with A — SHA

Operation

A ⇔ SPCH

Description

Exchange the contents of SPCH and accumulator. The least significant six bits of SPCH prefixed with two bits of 0 is exchanged with the content with A. The most significant two bits transferring from A to SPCH are ignored.

Condition Codes and Boolean Formulae

None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
SHA	INH	45		1	p

SLA Swap Shadow PC Low with A SLA

Operation A ⇔ SPCL

Description Exchange the content of SPCL and accumulator.

Condition Codes and Boolean Formulae None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
SLA	INH	42		1	p

STA — Store Accumulator in Memory — STA

Operation

M ← (A)

Description

Stores the contents of A in memory. The contents of A remain unchanged. The Z bit is set if A was $00. This allows conditional branching after the store without having to do a separate test or compare.

Condition Codes and Boolean Formulae

Z	C
↕	—

Z: $\overline{A7}\&\overline{A6}\&\overline{A5}\&\overline{A4}\&\overline{A3}\&\overline{A2}\&\overline{A1}\&\overline{A0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
STA *opr8a*	DIR	B7	dd	3	wpp
STA *opr5a*	SRT	E*x*/F*x*		2	wp

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
STA *,X*	IX	EE		2	wp
STA X	SRT	EF		2	wp

STOP Stop Processing STOP

Operation Stop processing

Description Reduces power consumption by eliminating all dynamic power dissipation. This instruction is used to enter stop mode.

Refer to specific device specification for behavior of each individual on-chip module during stop operation.

Condition Codes and Boolean Formulae None affected

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
STOP	INH	AE		2+stop	f...p

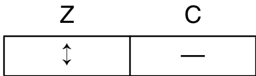
STX Store X (Index Register Low) in Memory (Pseudo Instruction) STX

Operation M ← (X)

Description

This is a pseudo instruction implemented by using a MOV instruction that stores the content of the memory-mapped X index register into the specified memory location. The contents of X remain unchanged. The Z bit is set if X was \$00. This allows conditional branching after the store without having to do a separate test or compare.

Condition Codes and Boolean Formulae



Z	C
↕	—

Z: $\overline{X7}\&\overline{X6}\&\overline{X5}\&\overline{X4}\&\overline{X3}\&\overline{X2}\&\overline{X1}\&\overline{X0}$
Set if X is \$00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
STX *opr8a*	DIR	4E	0F dd	5	rpwpp

SUB Subtract SUB

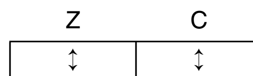
Operation

A ← (A) – (M)

Description

Subtracts the contents of M from A and places the result in A.

Condition Codes and Boolean Formulae



Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

C: $\overline{A7}\&M7 \mid M7\&R7 \mid R7\&\overline{A7}$
Set if the unsigned value of the contents of memory is larger than the unsigned value of the accumulator; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form		Addr. Mode	Machine Code		RS08 Cycles	Access Detail
			Opcode	Operand(s)		
SUB	*#opr8i*	IMM	A0	ii	2	pp
SUB	*opr8a*	DIR	B0	dd	3	rpp
SUB	*opr4a*	TNY	7*x*		3	rfp

Pseudo Instruction

Source Form		Addr. Mode	Machine Code		RS08 Cycles	Access Detail
			Opcode	Operand(s)		
SUB	,X	IX	7E		3	rfp
SUB	X	DIR	7F		3	rfp

TAX — Transfer A to X (Pseudo Instruction) — TAX

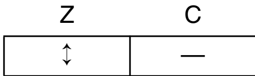
Operation

(X) ← A

Description

This is a pseudo instruction implemented using a STA instruction that transfers a copy of the contents of A to the Index register at location \$000F. The contents of \$000F remain unchanged. The Z bit is set if A contains \$00, this allows conditional branching after the store without having to do a separate test or compare.

Condition Codes and Boolean Formulae



Z: $\overline{A7}\&\overline{A6}\&\overline{A5}\&\overline{A4}\&\overline{A3}\&\overline{A2}\&\overline{A1}\&\overline{A0}$
Set if result is \$00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Pseudo Instruction (This is a pseudo code only instruction.)

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
TAX	INH	EF		2	wp

TST Test for Zero TST

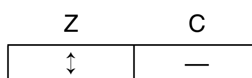
Operation

(A) – $00
Or (X) – $00
Or (M) – $00

Description

Sets the Z condition codes according to the contents of A, X, or M. The contents of A, X, and M are not altered.

Condition Codes and Boolean Formulae



Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Pseudo Instruction

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
TST *opr8a*	DD	4E	dd dd	5	rpwpp
TSTA	INH	AA	00	2	pp
TST ,*X*	IX	4E	0E 0E	5	rpwpp
TSTX	INH	4E	0F 0F	5	rpwpp

TXA Transfer X to A (Pseudo Instruction) TXA

Operation

A ← (X)

Description

This is a pseudo instruction implemented using a LDA instruction that transfers a copy of the contents of the Index register at location $000F to the accumulator. The contents of X remain unchanged. The Z bit is set if A is $00, this allows conditional branching after the store without having to do a separate test or compare.

Condition Codes and Boolean Formulae

Z	C
↕	—

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Pseudo Instruction (This is a pseudo code only instruction.)

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Detail
		Opcode	Operand(s)		
TXA	INH	CF		3	rfp

WAIT — Stop CPU Clock — WAIT

Operation

Inhibit CPU clocking until interrupted

Description

Reduces power consumption by eliminating dynamic power dissipation in some portions of the MCU. The timer, the timer prescaler, and the on-chip peripherals continue to operate (if enabled) because they are potential sources of an interrupt.

Interrupts from on-chip peripherals may be enabled or disabled by local control bits prior to execution of the WAIT instruction.

When either the $\overline{\text{RESET}}$ goes low or when any on-chip system requests interrupt service, the processor clocks are enabled and the reset, or other interrupt service request, is processed.

Condition Codes and Boolean Formulae

Z	C
—	—

Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Addr. Mode	Machine Code		RS08 Cycles	Access Details
		Opcode	Operand(s)		
WAIT	INH	AF		2+wait	f...p

Appendix B. Code Examples

B.1 Illegal Table

```
RS08 assembler version 1.13 (c)2004,5 Freescale Semiconductor
Author: S. Pickering

RS08 assembler pass #1...
RS08 assembler pass #2...
            ;
            ; This example demonstrates a table correctly mapped in a non-overlapping
            ; 64 byte page and another table which spans more than one page boundary

            ; The following table is incorrectly mapped into an overlapping page in flash
363a                         org      $363A
363a        table            dc.b     1,2,3,4,5,6,7,8
363a 01 02 03 04
363e 05 06 07 08
            ; The following assembler directives are used to determine if table is located
            ; within a single 64 byte page
3642 0008   size             set      *-table
3642 0001   error            set      ((table%64)+size)>64
                 if error
###PASS2 - ERROR### Undefined label: [table_Overflows_page_boundary]:29
###PASS2 - ERROR### Unknown opcode: [error] >>>ERRORtable_Overflows_page_boundary
                 endif

            ; The following table is correctly mapped into a non overlapping page in flash
373a                         org      $373A
373a        table1           dc.b     1,2,3,4
373a 01 02 03 04
            ; The following assembler directives are used to determine if table is located
            ; within a single 64 byte page
373e 0004   size             set      *-table1
373e 0000   error            set      ((table1%64)+size)>64
                 if error
                 ;;;ERROR"table1 Overflows page boundary"
                 endif

======================================================
```

B.2 lda

```
========================================================
RS08 assembler version 1.13 (c)2004,5 Freescale Semiconductor
Author: S. Pickering

RS08 assembler pass #1...
RS08 assembler pass #2...
0000 001f   pagesel             equ      $001f
0000 3ffc   nvopt               equ      $3ffc
3800                            org      $3800

            ;A load from extended memory

3800 3e ff 1F   restart      mov    #(NVOPT/64)%256,PAGESEL   ;Set PAGESEL register
3803 b6 fc                   lda    $C0+(NVOPT%64)            ;access at offset into window

========================================================
```

B.3 probe

```
========================================================
RS08 assembler version 1.13 (c)2004,5 Freescale Semiconductor
Author: S. Pickering

RS08 assembler pass #1...
RS08 assembler pass #2...
          ;
          ;The purpose of this routine is to read the contents of all 16K of address space
          ;
0000 001f   pagesel             equ      $001f
0000 00c0   window              equ      $C0
            ;
0000                            org      $0
0000 0001   i                   ds.b     1                ;outer loop counter
0001 0001   j                   ds.b     1                ;inner loop counter
            ;
            ;
0020                            org      $20
0020 3e 00 00                   mov      #0,i             ;Need to step round 256 pages!
0023 3e 00 1f                   mov      #0,PAGESEL       ;Start at beginning of memory map
0026 3e 40 01   outer           mov      #64,j
0029 3e c0 0f                ldx      #$C0           ;Point index register at start of window
002c ce         inner           lda      ,X               ;load via window
002d 2f                         incx
002e 51                         dec      j
002f 36 fb                      bne      inner
0031 3c 1f                      inc      PAGESEL          ;bump to next page of flash
0033 50                         dec      i
0034 36 f0                      bne      outer
            ;
0036 0000                       end
```

```
========================================================
```

B.4 walk

```
========================================================
RS08 assembler version 1.13 (c)2004,5 Freescale Semiconductor
Author: S. Pickering

RS08 assembler pass #1...
RS08 assembler pass #2...
            ; Author: Stephen Pickering
            ; Purpose:
            ; This program demonstrates how to access a sequence of values held
            ; in flash memory, outputting each value to PORTA. Access to flash is
            ; via the the page window, indexed addressing, interrupts.
            ;
            ; Note: This example does not cater for the data spanning multiple pages
            ;

            ;Assumes device has been initialized...
0000 001f   pagesel            equ      $001f           ;PAGESEL register
0000 0010   ptad               equ      $10             ;Port A data register
0000 0007   cnt                equ      $7              ;loop counter

            ;Declare a sequence of values in flash
3650                           org      $3650
3650        table              dc.b     1,2,3,4,5,6,7,8
3650 01 02 03 04
3654 05 06 07 08
3658 0008   size               equ      *-table         ;calculate the size of the table

3800                           org      $3800

            ;Set PAGESEL register such that the table is mapped within the memory window
3800 a6 d9      restart        lda      #table/64
3802 ff                        sta      PAGESEL

            ; Table is now in high page window $C0 .. $FF
            ; calculate offset within memory window
3803 3e 10 0f        ldx      #table%64    ;set index register to table address modulus 64
3806 a6 c0           lda      #$C0         ;load base address of memory window
3808 6f              addx                  ;add the
3809 4e 0f 1f        stx      PAGESEL

            ;Loop through table and output to porta after each interrupt
380c 3e 08 07                  mov      #size,cnt           ;set counter to table size
380f ce         next           lda      ,x                  ;get a byte from the table
3810 f0                        sta      PTAD                ;output to PORTA
3811 af                        wait                         ;wait for an interrupt
3812 2f                      incx                         ;Increment to next byte in table
            ;decrement count and if not complete (ie !=0) fetch next byte from flash
```

```
3813 3b 07 f9              dbnz    cnt,next


            ;all done
3816 ae                    stop                      ;wait for interrupt
3817 bc 38 00              jmp     restart           ;restart
```

Appendix C. Assembler and Disassembler Style Guide

C.1 Support Notes for RS08 Tools

To promote consistency, the following sections provide guidance for RS08 tools. This section covers only basic compatibility issues primarily targeted at the assembler and disassembler, which use the RS08 instruction set.

Refer to the tool vendor's documentation for specific implementation details.

C.1.1 Pseudo Instructions

All assemblers should accept the full RS08 instruction set including the pseudo instructions as input. Use of HC(S)08 operands as well as RS08 should be supported for all instructions where appropriate. For example:

```
        LDX      $0FF      ;Load direct
        LDA      ,X        ;Load accumulator indexed
        LDA      D[X]      ;Load accumulator indexed (Indirect via X)
```

Accepting all notations eases the use of existing HC(S)08 code and allows new users to use the D[X] style of operand, which is slightly more intuitive for users from a high-level language background.

C.1.2 Tiny and Short Addressing

The assembler should use the shortest addressing mode possible. Wherever possible, tiny and short versions of the instruction should be used. For example:

```
        ORG      $0
FRED:   RMB      1         ;Define fred within tiny space !
        LDA      $1F       ;Short addressing should be used (<32)
        LDA      FRED      ;short addressing should be used (<32)
        DEC      ,X        ;Tiny addressing should be used (<16)
```

```
         INC      $10      ;Direct addressing should be used (>15)
```

In the case of forward references, direct addressing should be used. For example:

```
         ORG      $0
         LDA      FRED     ;Direct addressing should be used as
                           ;FRED is undefined at this point
FRED:    RMB      1        ;even though fred will be within tiny space !
```

The exception to this strategy is where code forces a specific addressing mode. For example:

```
         LDA      <$0FF    ;Force short addressing (will use $1F)
         DEC      <$13     ;Force tiny addressing (will use $03)
         INC      >$01     ;Force direct addressing
```

C.1.3 Forcing Tiny/Short and Direct Addressing

The operators ‘<’ and ‘>’ are used to force tiny/short and direct addressing, respectively, where appropriate. For instructions that do not support tiny or short addressing modes, the ‘<’ and ‘>’ will be ignored. For example, SBC does not support tiny or short, so including ‘<’ and ‘>’ will have no effect and direct addressing mode will be used.

```
         SBC      <$01     ;Force tiny/short addressing
```

will generate a direct address instruction because SBC does not support tiny or short operands.

```
         ADD      >$01     ;Force direct addressing
```

will generate an instruction that uses direct addressing even though the operand is in tiny address space.

C.1.4 Unsupported Instructions

The assembler should cause an error for all instructions and addressing mode combinations not supported by the RS08 architecture. This will assist in porting HC(S)08 code to the RS08 architecture.

C.1.5 Tiny, Short, and Direct Address Usage Statistics

The assembler should be able to show the frequency of usage of the various addressing modes. This will assist in developing compact code by allowing a user to locate the most frequently used variables in the tiny address space. A major objective of the RS08's use of tiny and short addressing modes instructions is in producing compact code.

C.2 Debugger and Disassembly

By default, any RS08 code disassembled by a tool should produce HC(S)08 style code for all pseudo instructions. Further options can be provided to disassemble pseudo instructions to RS08 style or native RS08 instructions.

The method used to select the disassembly style will be tool vendor specific, but typically mode selection would be accomplished by an option or switch setting for the IDE/disassembler or a #pragma statement in the original source code. Consult tool vendor documentation for specific method.

Style	User Base
HC(S)08 style	The preferred mode for existing HC(S)08 customers in order to maintain code compatibility with other HC(S)08 projects.
RS08 style	The preferred mode for users new to the RS08 and users. It is more natural and descriptive in nature, especially for users with a high-level language background, such as C.
RS08 native	Rarely used — absolute disassembly.

C.2.1 HC(S)08 Style

The purpose of this mode is to maintain compatibility with HC(S)08 assembly language.

An example of RS08 code:

```
MOV     #$45,$0F
```

```
LDA     $0E
```

Should be disassembled as:

```
LDX     #$45
LDA     ,X
```

C.2.2 RS08 Style

The purpose of this mode is to maintain RS08 pseudo code usage.

An example of RS08 code:

```
MOV      #$45,$0F
LDA      $0E
```

Should be disassembled as:

```
LDX      #$45
LDA      D[X]
```

C.2.3 Native RS08 Style

The purpose of this mode is to show the exact RS08 assembly language used.

An example of RS08 code:

```
MOV      #$45,$0F
LDA      $0E
```

Should be disassembled as:

```
MOV      #$45,$0F
LDA      $0E
```

C.3 Compilers

High level language compilers, such as those used for C code, should provide a mechanism (switch or #pragma statement) to set the disassembler style. The mechanism to select the disassembly style is determined and documented by the tool vendor. Refer to the tool vendor documentation for details.

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

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2006. All rights reserved.

RS08RM
Rev 1.0, 04/2006

freescale™
semiconductor