

# Section 3 CPU

## 3.1 Overview

The H8/532 chip has the H8/500 Family CPU: a high-speed central processing unit designed for realtime control of a wide range of medium-scale office and industrial equipment. Its Hitachi-original architecture features eight 16-bit general registers, internal 16-bit data paths, and an optimized instruction set.

Section 3 summarizes the CPU architecture and instruction set.

### 3.1.1 Features

The main features of the H8/500 CPU are listed below.

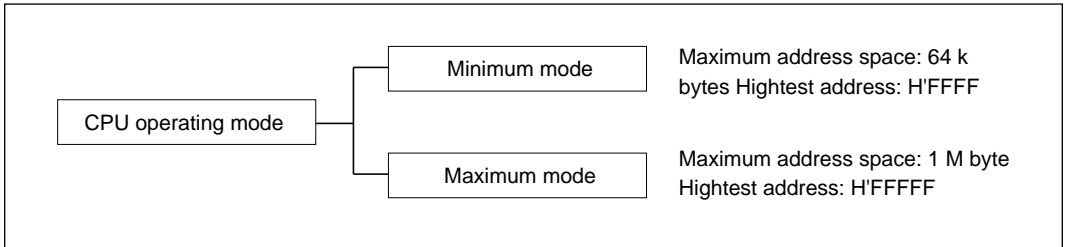
- General-register machine
  - Eight 16-bit general registers
  - Seven control registers (two 16-bit registers, five 8-bit registers)
- High speed: maximum 10MHz
  - At 10MHz a register-register add operation takes only 200ns.
- Address space managed in 64k-byte pages, expandable to 1M byte\*
  - Page registers make four pages available simultaneously: a code page, stack page, data page, and extended page.
- Two CPU operating modes:
  - Minimum mode: Maximum 64k-byte address space
  - Maximum mode: Maximum 1M-byte address space\*
- Highly orthogonal instruction set
  - Addressing modes and data sizes can be specified independently within each instruction.
- 1.5 Addressing modes
  - Register-register and register-memory operations are supported.
- Optimized for efficient programming in C language
  - In addition to the general registers and orthogonal instruction set, the CPU has special short formats for frequently-used instructions and addressing modes.

\* The CPU architecture supports up to 16M bytes of external memory, but the H8/532 chip has only enough address pins to address 1M byte.

### 3.1.2 Address Space

The address space size depends on the operating mode.

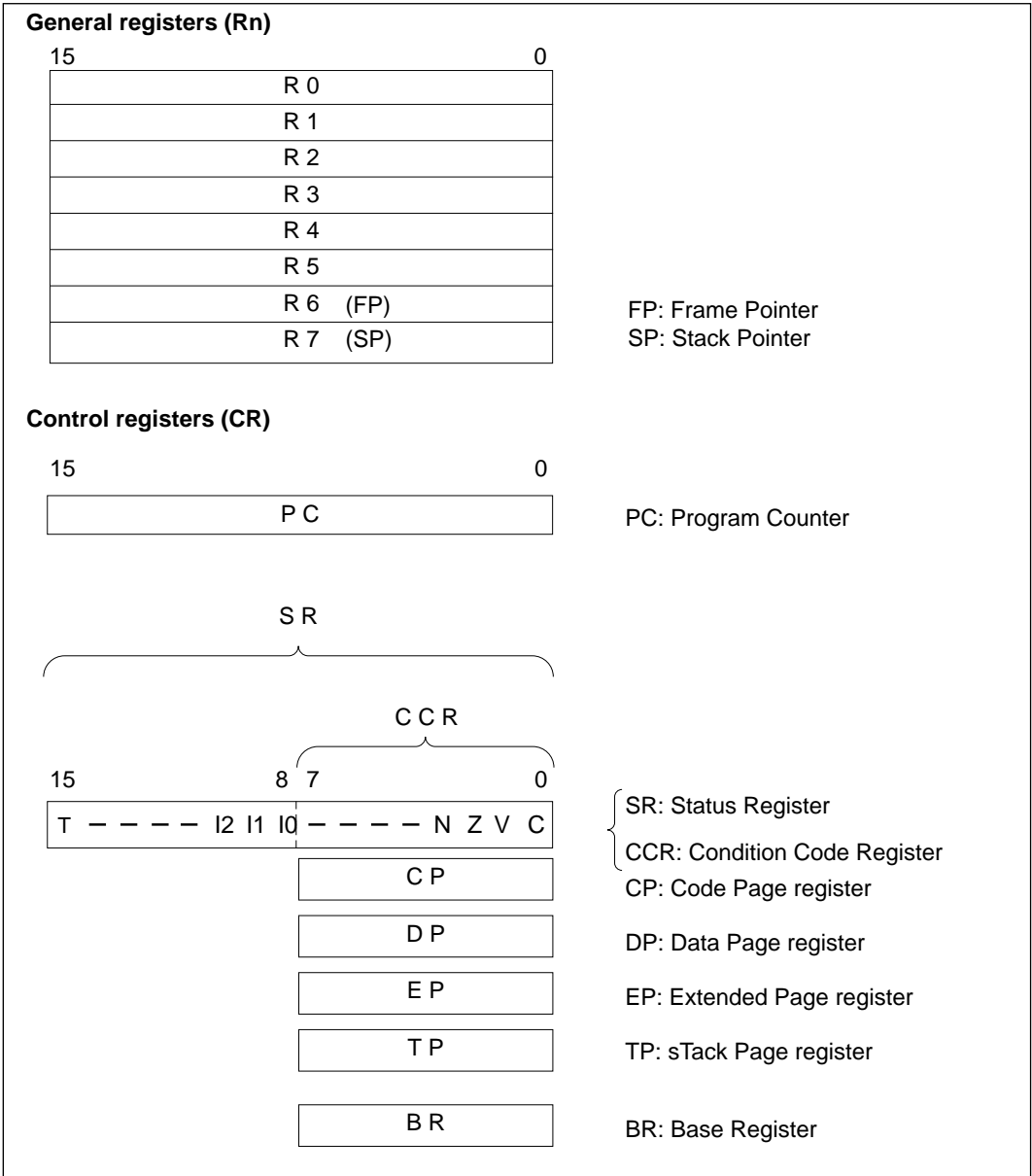
The H8/532 MCU has five operating modes, which are selected by the input to the mode pins (MD2 to MD0) when the chip comes out of a reset. The CPU, however, has only two operating modes. The MCU operating mode determines the CPU operating mode, which in turn determines the maximum address space size as indicated in figure 3-1.



**Figure 3-1 CPU Operating Modes**

### 3.1.3 Register Configuration

Figure 3-2 shows the register structure of the CPU. There are two groups of registers: the general registers (Rn) and control registers (CR).



**Figure 3-2 Registers in the CPU**

## 3.2 CPU Register Descriptions

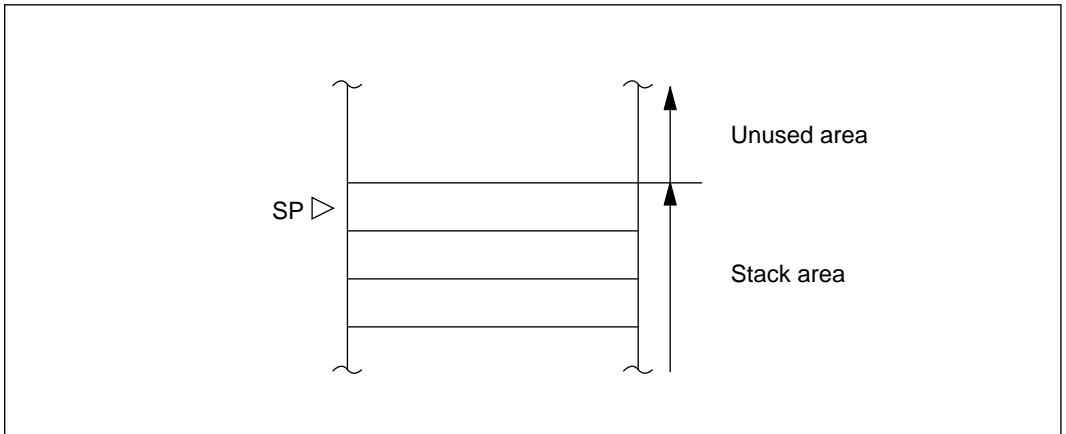
### 3.2.1 General Registers

All eight of the 16-bit general registers are functionally alike; there is no distinction between data registers and address registers. When these registers are accessed as data registers, either byte or word size can be selected.

R6 and R7, in addition to functioning as general registers, have special assignments.

R7 is the stack pointer, used implicitly in exception handling and subroutine calls. It can be designated by the name SP, which is synonymous with R7. As indicated in figure 3-3, it points to the top of the stack. It is also used implicitly by the LDM and STM instructions, which load and store multiple registers from and to the stack and pre-decrement or post-increment R7 accordingly.

R6 functions as a frame pointer (FP). The LINK and UNLK use R6 implicitly to reserve or release a stack frame.



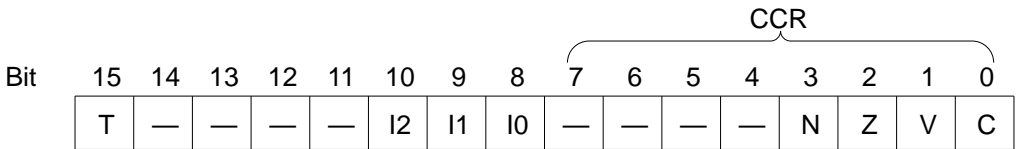
**Figure 3-3 Stack Pointer**

### 3.2.2 Control Registers

The CPU control registers (CR) include a 16-bit program counter (PC), a 16-bit status register (SR), four 8-bit page registers, and one 8-bit base register (BR).

**Program Counter (PC):** This 16-bit register indicates the address of the next instruction the CPU will execute.

**Status Register (SR):** This 16-bit register contains internal status information. The lower half of the status register is referred to as the condition code register (CCR): it can be accessed as a separate condition code byte.



**Bit 15—Trace (T):** When this bit is set to “1,” the CPU operates in trace mode and generates a trace exception after every instruction. See section 4.4, “Trace” for a description of the trace exception-handling sequence.

When the value of this bit is “0,” instructions are executed in normal continuous sequence. This bit is cleared to “0” at a reset.

**Bits 14 to 11—Reserved:** These bits cannot be modified and are always read as “0.”

**Bits 10 to 8—Interrupt Mask (I2, I1, I0):** These bits indicate the interrupt request mask level (0 to 7). As shown in table 3-1, an interrupt request is not accepted unless it has a higher level than the value of the mask. A nonmaskable interrupt (NMI), which has level 8, is accepted at any mask level. After an interrupt is accepted, I2, I1, and I0 are changed to the level of the interrupt. Table 3-2 indicates the values of the I bits after an interrupt is accepted.

A reset sets all three of bits (I2, I1, and I0) to “1,” masking all interrupts except NMI.

**Table 3-1 Interrupt Mask Levels**

Priority	Mask Level	Mask Bits			Interrupts Accepted
		I2	I1	I0	
High	7	1	1	1	NMI
↑	6	1	1	0	Level 7 and NMI
	5	1	0	1	Levels 6 to 7 and NMI
	4	1	0	0	Levels 5 to 7 and NMI
	3	0	1	1	Levels 4 to 7 and NMI
	2	0	1	0	Levels 3 to 7 and NMI
	1	0	0	1	Levels 2 to 7 and NMI
	Low	0	0	0	Levels 1 to 7 and NMI

**Table 3-2 Interrupt Mask Bits after an Interrupt is Accepted**

Level of Interrupt Accepted	I2	I1	I0
NMI (8)	1	1	1
7	1	1	1
6	1	1	0
5	1	0	1
4	1	0	0
3	0	1	1
2	0	1	0
1	0	0	1

**Bits 7 to 4—Reserved:** These bits cannot be modified and are always read as “0.”

**Bit 3—Negative (N):** This bit indicates the most significant bit (sign bit) of the result of an instruction.

**Bit 2—Zero (Z):** This bit is set to “1” to indicate a zero result and cleared to “0” to indicate a nonzero result.

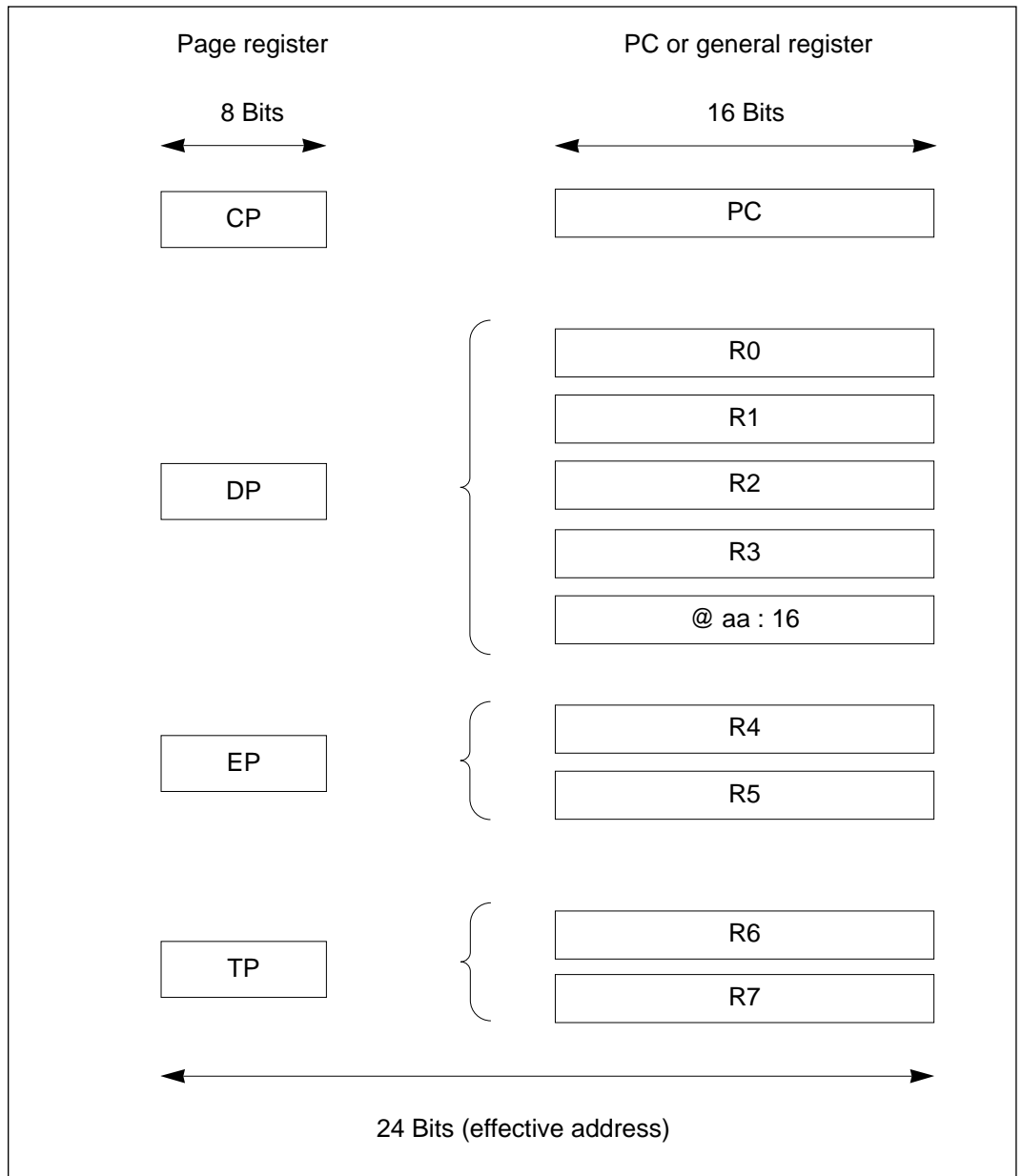
**Bit 1—Overflow (V):** This bit is set to “1” when an arithmetic overflow occurs, and cleared to “0” at other times.

**Bit 0—Carry (C):** This bit is set to “1” when a carry or borrow occurs at the most significant bit, and is cleared to “0” (or left unchanged) at other times.

The specific changes that occur in the condition code bits when each instruction is executed are listed in appendix A.1 “Instruction Tables.” See the *H8/500 Series Programming Manual* for further details.

**Page Registers:** The code page register (CP), data page register (DP), extended page register (EP), and stack page register (TP) are 8-bit registers that are used only in the maximum mode. No use of their contents is made in the minimum mode.

In the maximum mode, the page registers combine with the program counter and general registers to generate 24-bit effective addresses as shown in figure 3-4, thereby expanding the program area, data area, and stack area.



**Figure 3-4 Combinations of Page Registers with Other Registers**

**Code Page Register (CP):** The code page register and the program counter combine to generate a 24-bit program code address. In the maximum mode, the code page register is initialized at a reset to a value loaded from the vector table, and both the code page register and program counter



are saved and restored in exception handling.

**Data Page Register (DP):** The data page register combines with general registers R0 to R3 to generate a 24-bit effective address. The data page register contains the upper 8 bits of the address. It is used to calculate effective addresses in the register indirect addressing mode using R0 to R3, and in the 16-bit absolute addressing mode (@aa:16).

The data page register is rewritten by the LDC instruction.

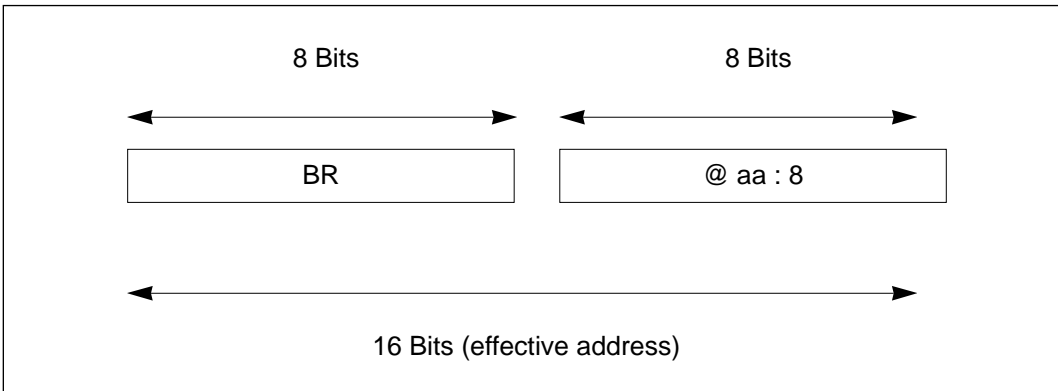
**Extended Page Register (EP):** The extended page register combines with general register R4 or R5 to generate a 24-bit operand address. The extended page register contains the upper 8 bits of the address. It is used to calculate effective addresses in the register indirect addressing mode using R4 or R5.

The extended page can be used as an additional data page.

**Stack Page Register (TP):** The stack page register combines with R6 (FP) or R7 (SP) to generate a 24-bit stack address. The stack page register contains the upper 8 bits of the address. It is used to calculate effective addresses in the register indirect addressing mode using R6 or R7, in exception handling, and subroutine calls.

**Base Register (BR):** This 8-bit register stores the base address used in the short absolute addressing mode (@aa:8). In this addressing mode a 16-bit effective address in page 0 is generated by using the contents of the base register as the upper 8 bits and an address given in the instruction code as the lower 8 bits. See figure 3-5.

In the short absolute addressing mode the address is always located in page 0.



**Figure 3-5 Short Absolute Addressing Mode and Base Register**

### 3.2.3 Initial Register Values

When the CPU is reset, its internal registers are initialized as shown in table 3-3. Note that the stack pointer (R7) and base register (BR) are not initialized to fixed values. Also, of the page registers used in maximum mode, only the code page register (CP) is initialized; the other three page registers come out of the reset state with undetermined values.

Accordingly, in the minimum mode the first instruction executed after a reset should initialize the stack pointer. The base register must also be initialized before the short absolute addressing mode (@aa:8) is used.

In the maximum mode, the first instruction executed after a reset should initialize the stack page register (TP) and the next instruction should initialize the stack pointer. Later instructions should initialize the base register and the other page registers as necessary.

**Table 3-3 Initial Values of Registers**

Register	Initial Value	
	Minimum Mode	Maximum Mode
General registers		
15 0 R7 – R0	Undetermined	Undetermined
Control registers		
15 0 PC	Loaded from vector table	Loaded from vector table
SR		
15 8 7 0 T-----I2I110-----NZVC	H'070x (x: undetermined)	H'070x (x: undetermined)
7 0 CP	Undetermined	Loaded from vector table
7 0 DP	Undetermined	Undetermined
7 0 EP	Undetermined	Undetermined
7 0 TP	Undetermined	Undetermined
7 0 BR	Undetermined	Undetermined

### 3.3 Data Formats

The H8/500 can process 1-bit data, 4-bit BCD data, 8-bit (byte) data, 16-bit (word) data, and 32-bit (longword) data.

- Bit manipulation instructions operate on 1-bit data.
- Decimal arithmetic instructions operate on 4-bit BCD data.
- Almost all instructions operate on byte and word data.
- Multiply and divide instructions operate on longword data.

#### 3.3.1 Data Formats in General Registers

Data of all the sizes above can be stored in general registers as shown in table 3-4.

Bit data locations are specified by bit number. Bit 15 is the most significant bit. Bit 0 is the least significant bit. BCD and byte data are stored in the lower 8 bits of a general register. Word data use all 16 bits of a general register. Longword data use two general registers: the upper 16 bits are stored in Rn (n must be an even number); the lower 16 bits are stored in Rn+1.

Operations performed on BCD data or byte data do not affect the upper 8 bits of the register.

**Table 3-4 General Register Data Formats**

Data Type	Register No.	Data Structure																
1-Bit	Rn	<div style="display: flex; justify-content: space-between; align-items: center;"> <span>15</span> <span>0</span> </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td> </tr> </table>	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
BCD	Rn	<div style="display: flex; justify-content: space-between; align-items: center;"> <span>15</span> <span>8 7</span> <span>4 3</span> <span>0</span> </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 50%;">Don't-care</td> <td style="width: 25%;">Upper digit</td> <td style="width: 25%;">Lower digit</td> </tr> </table>	Don't-care	Upper digit	Lower digit													
Don't-care	Upper digit	Lower digit																
Byte	Rn	<div style="display: flex; justify-content: space-between; align-items: center;"> <span>15</span> <span>8 7</span> <span>0</span> </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 50%;">Don't-care</td> <td style="width: 25%;">MSB</td> <td style="width: 25%;">LSB</td> </tr> </table>	Don't-care	MSB	LSB													
Don't-care	MSB	LSB																
Word	Rn	<div style="display: flex; justify-content: space-between; align-items: center;"> <span>15</span> <span>0</span> </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 50%;">MSB</td> <td style="width: 50%;">LSB</td> </tr> </table>	MSB	LSB														
MSB	LSB																	
Longword	Rn* Rn+1*	<div style="display: flex; justify-content: space-between; align-items: center;"> <span>31</span> <span>16</span> </div> <table border="1" style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td style="width: 50%;">MSB</td> <td style="width: 50%;">Upper 16 bits</td> </tr> <tr> <td style="width: 50%;"></td> <td style="width: 50%;">Lower 16 bits</td> </tr> <tr> <td style="width: 50%;"></td> <td style="width: 50%;">LSB</td> </tr> </table> <div style="display: flex; justify-content: space-between; align-items: center;"> <span>15</span> <span>0</span> </div>	MSB	Upper 16 bits		Lower 16 bits		LSB										
MSB	Upper 16 bits																	
	Lower 16 bits																	
	LSB																	

\* For longword data n must be even (0, 2, 4, or 6).

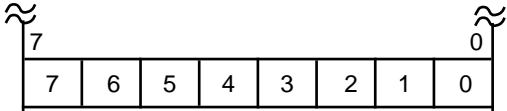
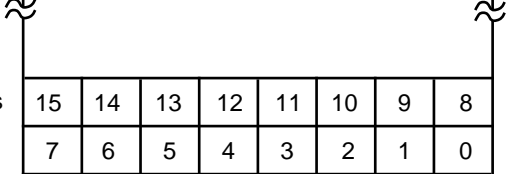
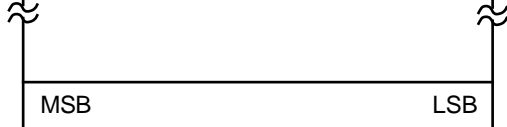
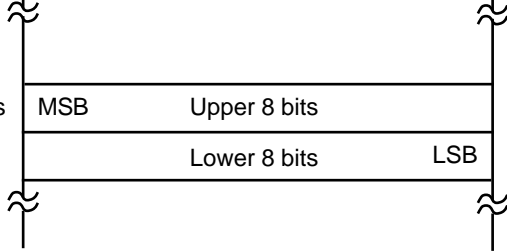
### 3.3.2 Data Formats in Memory

Table 3-5 indicates the data formats in memory.

Instructions that access bit data in memory have byte or word operands. The instruction specifies a bit number to indicate a specific bit in the operand.

Access to word data in memory must always begin at an even address. Access to word data starting at an odd address causes an address error. The upper 8 bits of word data are stored in address n (where n is an even number); the lower 8 bits are stored in address n+1.

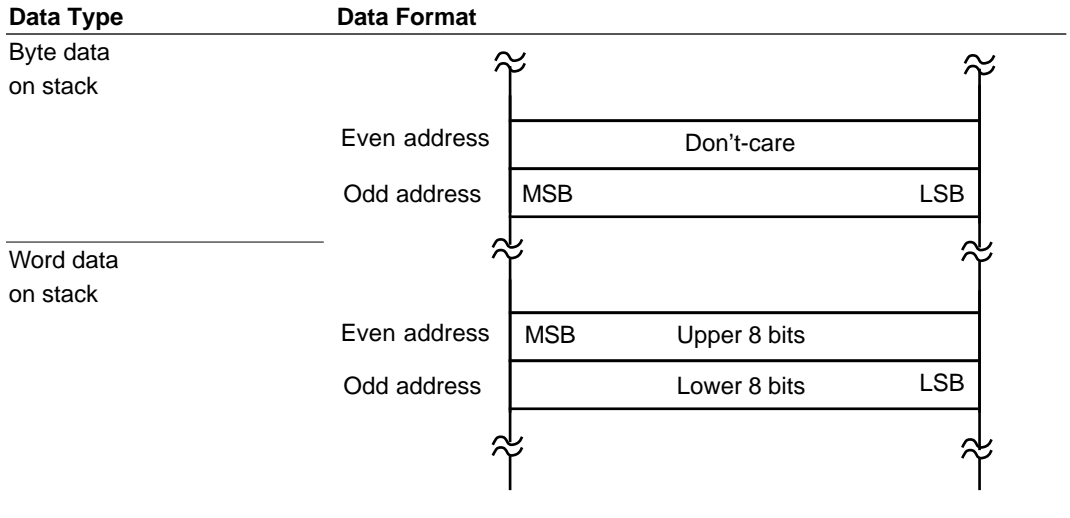
**Table 3-5 Data Formats in Memory**

Data Type	Data Format
1-Bit (in byte operand data)	 <p>Address n</p>
1-Bit (in word operand data)	 <p>Even address</p> <p>Odd address</p>
Byte	 <p>Address n</p> <p>MSB</p> <p>LSB</p>
Word	 <p>Even address</p> <p>MSB</p> <p>Upper 8 bits</p> <p>Odd address</p> <p>Lower 8 bits</p> <p>LSB</p>

When the stack is accessed in exception processing (to save or restore the program counter, code page register, or status register), word access is always performed, regardless of the actual data size. Similarly, when the stack is accessed by an instruction using the pre-decrement or post-increment register indirect addressing mode specifying R7 (@-R7 or @R7+), which is the stack pointer, word access is performed regardless of the operand size specified in the instruction. An address error will therefore occur if the stack pointer indicates an odd address. Programs should be coded so that the stack pointer always indicates an even address.

Table 3-6 shows the data formats on the stack.

**Table 3-6 Data Formats on the Stack**

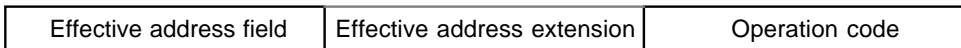


## 3.4 Instructions

### 3.4.1 Basic Instruction Formats

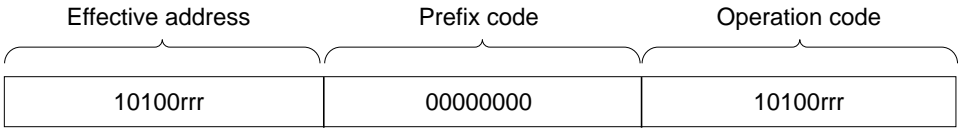
There are two basic CPU instruction formats: the general format and the special format.

**General format:** This format consists of an effective address (EA) field, an effective address extension field, and an operation code (OP) field. The effective address is placed before the operation code because this results in faster execution of the instruction.

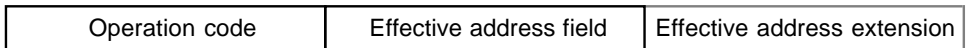


- **Effective address field:** One byte containing information used to calculate the effective address of an operand.
- **Effective address extension:** Zero to two bytes containing a displacement value, immediate data, or an absolute address. The size of the effective address extension is specified in the effective address field.
- **Operation code:** Defines the operation to be carried out on the operand located at the address calculated from the effective address information. Some instructions (DADD, DSUB, MOVFPE, MOVTPE) have an extended format in which the operand code is preceded by a one-byte prefix code.

- (Example of prefix code in DADD instruction)



**Special Format:** In this format the operation code comes first, followed by the effective address field and effective address extension. This format is used in branching instructions, system control instructions, and other instructions that can be executed faster if the operation is specified before the operand.



- Operation code: One or two bytes defining the operation to be performed by the instruction.
- Effective address field and effective address extension: Zero to three bytes containing information used to calculate an effective address.

### 3.4.2 Addressing Modes

The CPU supports 7 addressing modes: (1) register direct; (2) register indirect; (3) register indirect with displacement; (4) register indirect with pre-decrement or post-increment; (5) immediate; (6) absolute; and (7) PC-relative.

Due to the highly orthogonal nature of the instruction set, most instructions having operands can use any applicable addressing mode from (1) through (6). The PC-relative mode (7) is used by branching instructions.

In most instructions, the addressing mode is specified in the effective address field. The effective-address extension, if present, contains a displacement, immediate data, or an absolute address.

Table 3-7 indicates how the addressing mode is specified in the effective address field.

**Table 3-7 Addressing Modes**

No.	Addressing Mode	Mnemonic	EA Field	EA Extension
1	Register direct	Rn	<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     1 0 1 0 Sz r r r                 </div> <small>*1 *2</small>	None
2	Register indirect	@Rn	<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     1 1 0 1 Sz r r r                 </div>	None
3	Register indirect with displacement	@(d:8,Rn)	<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     1 1 1 0 Sz r r r                 </div>	Displacement (1 byte)
		@(d:16,Rn)	<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     1 1 1 1 Sz r r r                 </div>	Displacement (2 bytes)
4	Register indirect with pre-decrement	@-Rn	<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     1 0 1 1 Sz r r r                 </div>	None
	Register indirect with post-increment	@Rn+	<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     1 1 0 0 Sz r r r                 </div>	
5	Immediate	#xx:8	<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     0 0 0 0 0 1 0 0                 </div>	Immediate data (1 byte)
		#xx:16	<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     0 0 0 0 1 1 0 0                 </div>	Immediate data (2 bytes)
6	Absolute *3	@aa:8	<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     0 0 0 0 Sz 1 0 1                 </div>	1-Byte absolute address (offset from BR)
		@aa:16	<div style="border: 1px solid black; padding: 2px; display: inline-block;">                     0 0 0 1 Sz 1 0 1                 </div>	2-Byte absolute address
7	PC-relative	disp	No EA field. Addressing mode is specified in the operation code.	1- or 2-byte displacement

**Notes:** \* 1 Sz: Specifies the operand size.

When Sz = 0: byte operand

When Sz = 1: word operand

\* 2 rrr: Register number field, specifying a general register number.

0 0 0 — R0    0 0 1 — R1    0 1 0 — R2    0 1 1 — R3

1 0 0 — R4    1 0 1 — R5    1 1 0 — R6    1 1 1 — R7

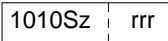
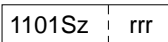
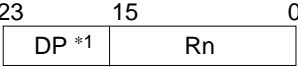
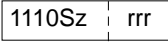
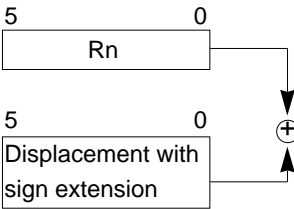
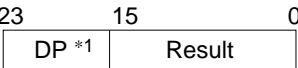
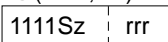
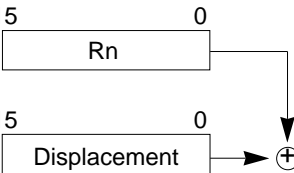
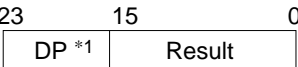
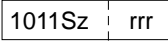
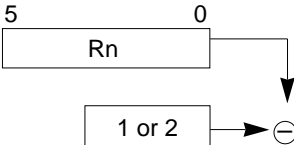
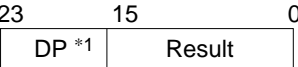

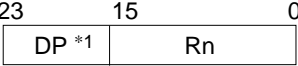
\* 3 The @aa:8 addressing mode is also referred to as the short absolute addressing mode.



### 3.4.3 Effective Address Calculation

Table 3-8 explains how the effective address is calculated in each addressing mode.

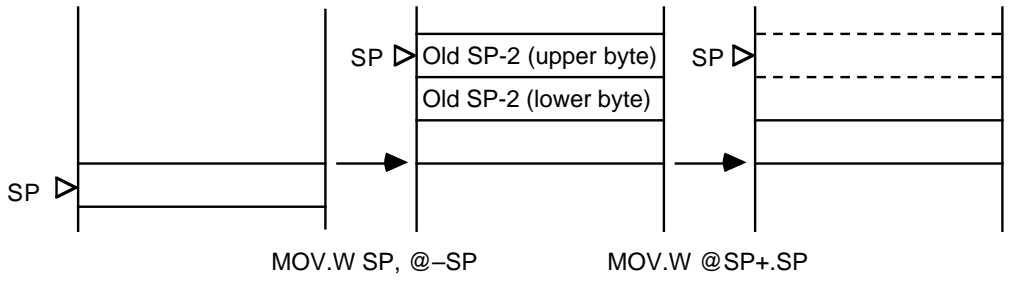
**Table 3-8 Effective Address Calculation**

No.	Addressing Mode	Effective Address Calculation	Effective Address
1	Register direct Rn 	—	Operand is contents of Rn
2	Register indirect @Rn 	—	 Or TP or EP *2
3	Register indirect with displacement @(d:8,Rn) 	8 Bits 	 Or TP or EP *2
	@(d:16,Rn) 	16 Bits 	 Or TP or EP *2
4	Register indirect with pre-decrement @-Rn 	 Rn is decremented by -1 or -2 before instruction execution.*3*4*5	 Or TP or EP *2
	Register indirect with post-increment @Rn+ 	— Rn is incremented by +1 or +2 after instruction execution.*3*4*5	 Or TP or EP *2

**Table 3-8 Effective Address Calculation (cont)**

No.	Addressing Mode	Effective Address Calculation	Effective Address
5	Absolute address	—	23 15 0
	@aa:8	<div style="border: 1px solid black; padding: 2px; width: fit-content;">0000Sz101</div>	<div style="border: 1px solid black; padding: 2px; width: 100%; display: flex; justify-content: space-between;"> <span>H'00</span> <span>BR</span> <span></span> </div> <div style="text-align: right; margin-top: 5px;">EA extension data <span style="font-size: 0.8em;">▲</span></div>
5	@aa:16	—	23 15 0
		<div style="border: 1px solid black; padding: 2px; width: fit-content;">0001Sz101</div>	<div style="border: 1px solid black; padding: 2px; width: 100%; display: flex; justify-content: space-between;"> <span>DP</span> <span>EA extension data</span> </div>
6	Immediate	—	Operand is 1-byte EA extension data.
	#xx:8	<div style="border: 1px solid black; padding: 2px; width: fit-content;">00000100</div>	
6	#xx:16	—	Operand is 2-byte EA extension data.
		<div style="border: 1px solid black; padding: 2px; width: fit-content;">00001100</div>	
7	PC-relative	8 Bits	23 15 0
	disp:8	15 0	23 15 0
No EA code		<div style="border: 1px solid black; padding: 2px; width: 100%;">PC</div>	<div style="border: 1px solid black; padding: 2px; width: 100%; display: flex; justify-content: space-between;"> <span>CP *1</span> <span>Result</span> </div>
Specified in OP code		<div style="border: 1px solid black; padding: 2px; width: 100%;">Displacement with sign extension</div>	$\oplus$
			<span style="font-size: 0.8em;">▲</span>
7	PC-relative	16 Bits	23 15 0
	disp:16	15 0	23 15 0
No EA code		<div style="border: 1px solid black; padding: 2px; width: 100%;">PC</div>	<div style="border: 1px solid black; padding: 2px; width: 100%; display: flex; justify-content: space-between;"> <span>CP *1</span> <span>Result</span> </div>
Specified in OP code		<div style="border: 1px solid black; padding: 2px; width: 100%;">Displacement</div>	$\oplus$
			<span style="font-size: 0.8em;">▲</span>

- Notes:**
- \* 1 The page register is ignored in minimum mode.
  - \* 2 The page register used in addressing modes 2, 3, and 4 depends on the general register : DP for R0, R1, R2, or R3; EP for R4 or R5; TP for R6 or R7.
  - \* 3 Decrement by  $-1$  for a byte operand, and by  $-2$  for a word operand.
  - \* 4 The pre-decrement or post-increment is always  $\pm 2$  when R7 is specified, even if the operand is byte size.
  - \* 5 The drawing below shows what happens when the @-SP and @ SP+ addressing modes are used to save and restore the stack pointer.



## 3.5 Instruction Set

### 3.5.1 Overview

The main features of the CPU instruction set are:

- A general-register architecture.
- Orthogonality. Addressing modes and data sizes can be specified independently in each instruction.
- 1.5 addressing modes (supporting register-register and register-memory operations)
- Affinity for high-level languages, particularly C, with short formats for frequently-used instructions and addressing modes.
- Standard mnemonics, common throughout the H Series.

The CPU instruction set includes 63 types of instructions, listed by function in table 3-9.

**Table 3-9 Instruction Classification**

Function	Instructions	Types
Data transfer	MOV, LDM, STM, XCH, SWAP, MOVTPPE, MOVFPPE	7
Arithmetic operations	ADD, SUB, ADDS, SUBS, ADDX, SUBX, DADD, DSUB, MULXU, DIVXU, CMP, EXTS, EXTU, TST, NEG, CLR, TAS	17
Logic operations	AND, OR, XOR, NOT	4
Shift	SHAL, SHAR, SHLL, SHLR, ROTL, ROTR, ROTXL, ROTXR	8
Bit manipulation	BSET, BCLR, BTST, BNOT	4
Branch	Bcc*, JMP, PJMP, BSR, JSR, PJSR, RTS, PRTD, PRTS, RTD, SCB (/F, /NE, /EQ)	11
System control	TRAPA, TRAP/VS, RTE, SLEEP, LDC, STC, ANDC, ORC, XORC, NOP, LINK, UNLK	12
	Total	63

\* Bcc is a conditional branch instruction in which cc represents a condition code.

Tables 3-10 to 3-16 give a concise summary of the instructions in each functional category. The MOV, ADD, and CMP instructions have special short formats, which are listed in table 3-17. For detailed descriptions of the instructions, refer to the *H8/500 Series Programming Manual*.

The notation used in tables 3-10 to 3-17 is defined below.

## Operation Notation

Rd	General register (destination)
Rs	General register (source)
Rn	General register
(EAd)	Destination operand
(EAs)	Source operand
CCR	Condition code register
N	N (negative) bit of CCR
Z	Z (zero) bit of CCR
V	V (overflow) bit of CCR
C	C (carry) bit of CCR
CR	Control register
PC	Program counter
CP	Code page register
SP	Stack pointer
FP	Frame pointer
#IMM	Immediate data
disp	Displacement
+	Addition
-	Subtraction
×	Multiplication
÷	Division
^	AND logical
∨	OR logical
⊕	Exclusive OR logical
→	Move
↔	Exchange
¬	Not

### 3.5.2 Data Transfer Instructions

Table 3-10 describes the seven data transfer instructions.

**Table 3-10 Data Transfer Instructions**

Instruction	Size*	Function
Data transfer	MOV	(EAs) → (EAd), #IMM → (EAd)
	MOV:G	B/W
	MOV:E	B
	MOV:I	W
	MOV:F	B/W
	MOV:L	B/W
	MOV:S	B/W
	LDM	W Stack → Rn (register list) Pops data from the stack to one or more registers.
	STM	W Rn (register list) → stack Pushes data from one or more registers onto the stack.
	XCH	W Rs ↔ Rd Exchanges data between two general registers.
	SWAP	B Rd (upper byte) ↔ Rd (lower byte) Exchanges the upper and lower bytes in a general register.
	MOVTPE	B Rn → (EAd) Transfers data from a general register to memory in synchronization with the E clock.
	MOVFPPE	B (EAs) → Rd Transfers data from memory to a general register in synchronization with the E clock.

**Note:** B—byte; W—word

### 3.5.3 Arithmetic Instructions

Table 3-11 describes the 17 arithmetic instructions.

**Table 3-11 Arithmetic Instructions**

Instruction	Size	Function
Arithmetic operations	ADD	$Rd \pm (EAs) \rightarrow Rd, (EAd) \pm \#IMM \rightarrow (EAd)$
	ADD:G	Performs addition or subtraction on data in a general register and data in another general register or memory, or on immediate data and data in a general register or memory.
	ADD:Q	
	SUB	
	ADDS	
	SUBS	
	ADDX	$Rd \pm (EAs) \pm C \rightarrow Rd$
	SUBX	Performs addition or subtraction with carry or borrow on data in a general register and data in another general register or memory, or on immediate data and data in a general register or memory.
	DADD	$(Rd)_{10} \pm (Rs)_{10} \pm C \rightarrow (Rd)_{10}$
	DSUB	Performs decimal addition or subtraction on data in two general registers.
	MULXU	$Rd \times (EAs) \rightarrow Rd$ Performs 8-bit $\times$ 8-bit or 16-bit $\times$ 16-bit unsigned multiplication on data in a general register and data in another general register or memory, or on data in a general register and immediate data.
	DIVXU	$Rd \div (EAs) \rightarrow Rd$ Performs 16-bit $\div$ 8-bit or 32-bit $\div$ 16-bit unsigned division on data in a general register and data in another general register or memory, or on data in a general register and immediate data.
	CMP	$Rn - (EAs), (EAd) - \#IMM$
	CMP:G	Compares data in a general register with data in another general register or memory, or with immediate data, or compares immediate data with data in memory.
	CMP:E	
	CMP:I	

**Note:** B—byte; W—word

**Table 3-11 Arithmetic Instructions (cont)**

<b>Instruction</b>		<b>Size</b>	<b>Function</b>
Arithmetic operations	EXTS	B	$(\langle \text{bit } 7 \rangle \text{ of } \langle \text{Rd} \rangle) \rightarrow (\langle \text{bits } 15 \text{ to } 8 \rangle \text{ of } \langle \text{Rd} \rangle)$ Converts byte data in a general register to word data by extending the sign bit.
	EXTU	B	$0 \rightarrow (\langle \text{bits } 15 \text{ to } 8 \rangle \text{ of } \langle \text{Rd} \rangle)$ Converts byte data in a general register to word data by padding with zero bits.
	TST	B/W	$(\text{EAd}) - 0$ Compares general register or memory contents with 0.
	NEG	B/W	$0 - (\text{EAd}) \rightarrow (\text{EAd})$ Obtains the two's complement of general register or memory contents.
	CLR	B/W	$0 \rightarrow (\text{EAd})$ Clears general register or memory contents to 0.
	TAS	B	$(\text{EAd}) - 0, (1)_2 \rightarrow (\langle \text{bit } 7 \rangle \text{ of } \langle \text{EAd} \rangle)$ Tests general register or memory contents, then sets the most significant bit (bit 7) to "1."

**Note:** B—byte; W—word

### 3.5.4 Logic Operations

Table 3-12 lists the four instructions that perform logic operations.

**Table 3-12 Logic Operation Instructions**

<b>Instruction</b>		<b>Size</b>	<b>Function</b>
Logical operations	AND	B/W	$\text{Rd} \wedge (\text{EAs}) \rightarrow \text{Rd}$ Performs a logical AND operation on a general register and another general register, memory, or immediate data.
	OR	B/W	$\text{Rd} \vee (\text{EAs}) \rightarrow \text{Rd}$ Performs a logical OR operation on a general register and another general register, memory, or immediate data.
	XOR	B/W	$\text{Rd} \oplus (\text{EAs}) \rightarrow \text{Rd}$ Performs a logical exclusive OR operation on a general register and another general register, memory, or immediate data.
	NOT	B/W	$\neg (\text{EAd}) \rightarrow (\text{EAd})$ Obtains the one's complement of general register or memory contents.

**Note:** B—byte; W—word



### 3.5.5 Shift Operations

Table 3-13 lists the eight shift instructions.

**Table 3-13 Shift Instructions**

<b>Instruction</b>		<b>Size</b>	<b>Function</b>
Shift operations	SHAL	B/W	(EAd) shift → (EAd)
	SHAR	B/W	Performs an arithmetic shift operation on general register or memory contents.
	SHLL	B/W	(EAd) shift → (EAd)
	SHLR	B/W	Performs a logical shift operation on general register or memory contents.
	ROTL	B/W	(EAd) shift → (EAd)
	ROTR	B/W	Rotates general register or memory contents.
	ROTXL	B/W	(EAd) rotate through carry → (EAd)
	ROTXR	B/W	Rotates general register or memory contents through the C (carry) bit.

**Note:** B—byte; W—word

### 3.5.6 Bit Manipulations

Table 3-14 describes the four bit-manipulation instructions.

**Table 3-14 Bit-Manipulation Instructions**

<b>Instruction</b>		<b>Size</b>	<b>Function</b>
Bit manipulations	BSET	B/W	$\neg$ (<bit-No.> of <EAd>) $\rightarrow$ Z, 1 $\rightarrow$ (<bit-No.> of <EAd>) Tests a specified bit in a general register or memory, then sets the bit to “1.” The bit is specified by a bit number given in immediate data or a general register.
	BCLR	B/W	$\neg$ (<bit-No.> of <EAd>) $\rightarrow$ Z, 0 $\rightarrow$ (<bit-No.> of <EAd>) Tests a specified bit in a general register or memory, then clears the bit to “0.” The bit is specified by a bit number given in immediate data or a general register.
	BNOT	B/W	$\neg$ (<bit-No.> of <EAd>) $\rightarrow$ Z, $\rightarrow$ (<bit-No.> of <EAd>) Tests a specified bit in a general register or memory, then inverts the bit. The bit is specified by a bit number given in immediate data or a general register.
	BTST	B/W	$\neg$ (<bit-No.> of <EAd>) $\rightarrow$ Z Tests a specified bit in a general register or memory. The bit is specified by a bit number given in immediate data or a general register.

**Note:** B—byte; W—word

### 3.5.7 Branching Instructions

Table 3-15 describes the 11 branching instructions.

**Table 3-15 Branching Instructions**

Instruction	Size	Function
Branch	Bcc	—
		Branches if condition cc is true.
Mnemonic	Description	Condition
BRA (BT)	Always (true)	True
BRN (BF)	Never (false)	False
BHI	High	$C \vee Z = 0$
BLS	Low or Same	$C \vee Z = 1$
BCC (BHS)	Carry Clear (High or Same)	$C = 0$
BCS (BLO)	Carry Set (Low)	$C = 1$
BNE	Not Equal	$Z = 0$
BEQ	Equal	$Z = 1$
EVC	Overflow Clear	$V = 0$
EVS	Overflow Set	$V = 1$
BPL	Plus	$N = 0$
BMI	Minus	$N = 1$
BGE	Greater or Equal	$N \oplus V = 0$
BLT	Less Than	$N \oplus V = 1$
BGT	Greater Than	$Z \vee (N \oplus V) = 0$
BLE	Less or Equal	$Z \vee (N \oplus V) = 1$
JMP	—	Branches unconditionally to a specified address in the same page.
PJMP	—	Branches unconditionally to a specified address in a specified page.
BSR	—	Branches to a subroutine at a specified address in the same page.
JSR	—	Branches to a subroutine at a specified address in the same page.
PJSR	—	Branches to a subroutine at a specified address in a specified page.
RTS	—	Returns from a subroutine in the same page.

**Table 3-15 Branching Instructions (cont)**

<b>Instruction</b>	<b>Size</b>	<b>Function</b>	
Branch	PRTS	—	Returns from a subroutine in a different page.
	RTD	—	Returns from a subroutine in the same page and adjusts the stack pointer.
	PRTD	—	Returns from a subroutine in a different page and adjusts the stack pointer.
	SCB/F	—	Controls a loop using a loop counter and/or a specified termination condition.
	SCB/NE	—	
SCB/EQ	—		

### 3.5.8 System Control Instructions

Table 3-16 describes the 12 system control instructions.

**Table 3-16 System Control Instructions**

Instruction	Size	Function	
System control	TRAPA	—	Generates a trap exception with a specified vector number.
	TRAP/VS	—	Generates a trap exception if the V bit is set to “1” when the instruction is executed.
	RTE	—	Returns from an exception-handling routine.
	LINK	—	FP → @-SP; SP → FP; SP + #IMM → SP Creates a stack frame.
	UNLK	—	FP → SP; @SP+ → FP Deallocates a stack frame created by the LINK instruction.
	SLEEP	—	Causes a transition to the power-down state.
	LDC	B/W*	(EAs) → CR Moves immediate data or general register or memory contents to a specified control register.
	STC	B/W*	CR → (EAd) Moves control register data to a specified general register or memory location.
	ANDC	B/W*	CR ∧ #IMM → CR Logically ANDs a control register with immediate data.
	ORC	B/W*	CR ∨ #IMM → CR Logically ORs a control register with immediate data.
	XORC	B/W*	CR ⊕ #IMM → CR Logically exclusive-ORs a control register with immediate data.
	NOP	—	PC + 1 → PC No operation. Only increments the program counter.

\* The size depends on the control register.

When using the LDC and STC instructions to stack and unstack the BR, CCR, TP, DP, and EP control registers in the H8/500 family, note the following point.

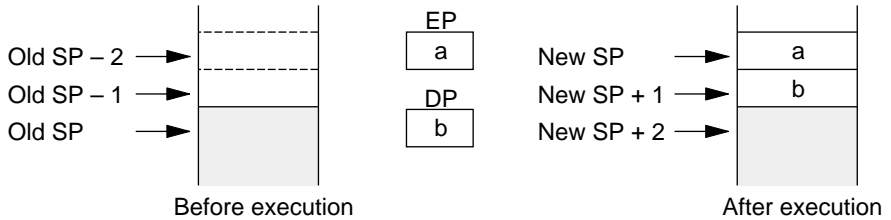
H8/500 hardware does not permit byte access to the stack. If the LDC.B or STC.B assembler mnemonic is coded with the @R7 + (@SP+) or @-R7 (@-SP) addressing mode, the stack-pointer addressing mode takes precedence and hardware automatically performs word access.

Specifically, the LDC.B and STC.B instructions are executed as follows.

The following applies only to the stack-pointer addressing modes. In addressing modes that do not use the stack pointer, byte data access is performed as specified by the assembler mnemonic.

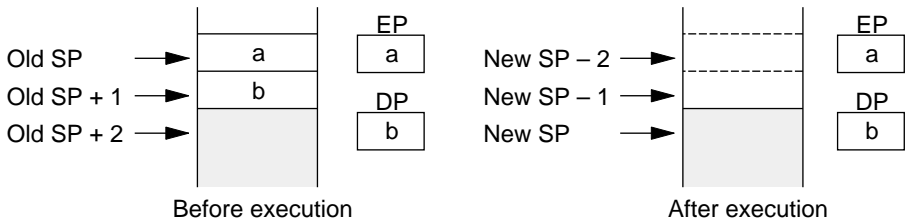
(1) STC.B EP, @-SP

When word data access is applied to EP, both EP and DP are accessed. This instruction stores EP at address SP (old) - 2, and DP at address SP (old) - 1.



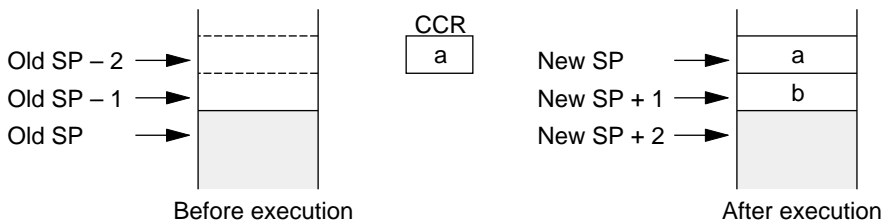
(2) LDC.B @SP+, EP

When word data access is applied to EP, both EP and DP are accessed. This instruction loads EP from address SP (old), and DP from address SP (old) + 1, updating the DP value as well as the EP value.



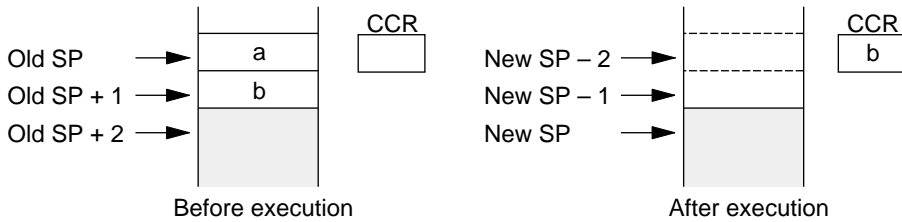
(3) STC.B CCR, @-SP

When word data access is applied to CCR, only CCR is accessed. This instruction stores identical CCR contents at both address SP (old) - 2 and address SP (old) - 1.



(4) LDC.B @SP+, CCR

When word data access is applied to CCR, only CCR is accessed. This instruction loads CCR from address SP (old) +1. Note that the value in address SP (old) is not loaded.



BR, DP, and TP are accessed in the same way as CCR. When DP is specified, both EP and DP are accessed, but when CCR, BR, DP, or TP is specified, only the specified register is accessed.

### 3.5.9 Short-Format Instructions

The ADD, CMP, and MOV instructions have special short formats. Table 3-17 lists these short formats together with the equivalent general formats.

The short formats are a byte shorter than the corresponding general formats, and most of them execute one state faster.

**Table 3-17 Short-Format Instructions and Equivalent General Formats**

Short-Format Instruction	Length	Execution States <sup>*2</sup>	Equivalent General-Format Instruction	Length	Execution States <sup>*2</sup>
ADD:Q #xx,Rd <sup>*1</sup>	2	2	ADD:G #xx:8,Rd	3	3
CMP:E #xx:8,Rd	2	2	CMP:G.B #xx:8,Rd	3	3
CMP:I #xx:16,Rd	3	3	CMP:G.W #xx:16,Rd	4	4
MOV:E #xx:8,Rd	2	2	MOV:G.B #xx:8,Rd	3	3
MOV:I #xx:16,Rd	3	3	MOV:G.W #xx:16,Rd	4	4
MOV:L @aa:8,Rd	2	5	MOV:G @aa:8,Rd	3	5
MOV:S Rs,@aa:8	2	5	MOV:G Rs,@aa:8	3	5
MOV:F @(d:8,R6),Rd	2	5	MOV:G @(d:8,R6),Rd	3	5
MOV:F Rs,@(d:8,R6)	2	5	MOV:G Rs,@(d:8,R6)	3	5

**Notes:** \* 1 The ADD:Q instruction accepts other destination operands in addition to a general register, but the immediate data value (#xx) is limited to  $\pm 1$  or  $\pm 2$ .

\* 2 Number of execution states for access to on-chip memory.

## 3.6 Operating Modes

The CPU operates in one of two modes: the minimum mode or the maximum mode. These modes are selected by the mode pins (MD2 to MD0).

### 3.6.1 Minimum Mode

The minimum mode supports a maximum address space of 64k bytes. The page registers are ignored. Instructions that branch across page boundaries (PJMP, PJSR, PRTS, PRTD) are invalid.



### 3.6.2 Maximum Mode

In the maximum mode the page registers are valid, expanding the maximum address space to 1M byte.

The address space is divided into 64k-byte pages. The pages are separate; it is not possible to move continuously across a page boundary.

It is possible to move from one page to another with branching instructions (PJMP, PJSR, PRTS, PRTD). The TRAPA instruction and branches to interrupt-handling routines can also jump across page boundaries. It is not necessary for a program to be contained in a single 64k-byte page.

When data access crosses a page boundary, the program must rewrite the page register before it can access the data in the next page.

For further information on the operating modes, see section 2, “MCU Operating Modes and Address Space.”

## 3.7 Basic Operational Timing

### 3.7.1 Overview

The CPU operates on a system clock ( $\phi$ ) which is created by dividing an oscillator frequency ( $f_{osc}$ ) by two. One period of the system clock is referred to as a “state.” The CPU accesses memory in a cycle consisting of 2 or 3 states. The CPU uses different methods to access on-chip memory, the on-chip register field, and external devices.

**Access to On-Chip Memory (RAM, ROM):** For maximum speed, access to on-chip memory (RAM, ROM) is performed in two states, using a 16-bit-wide data bus.

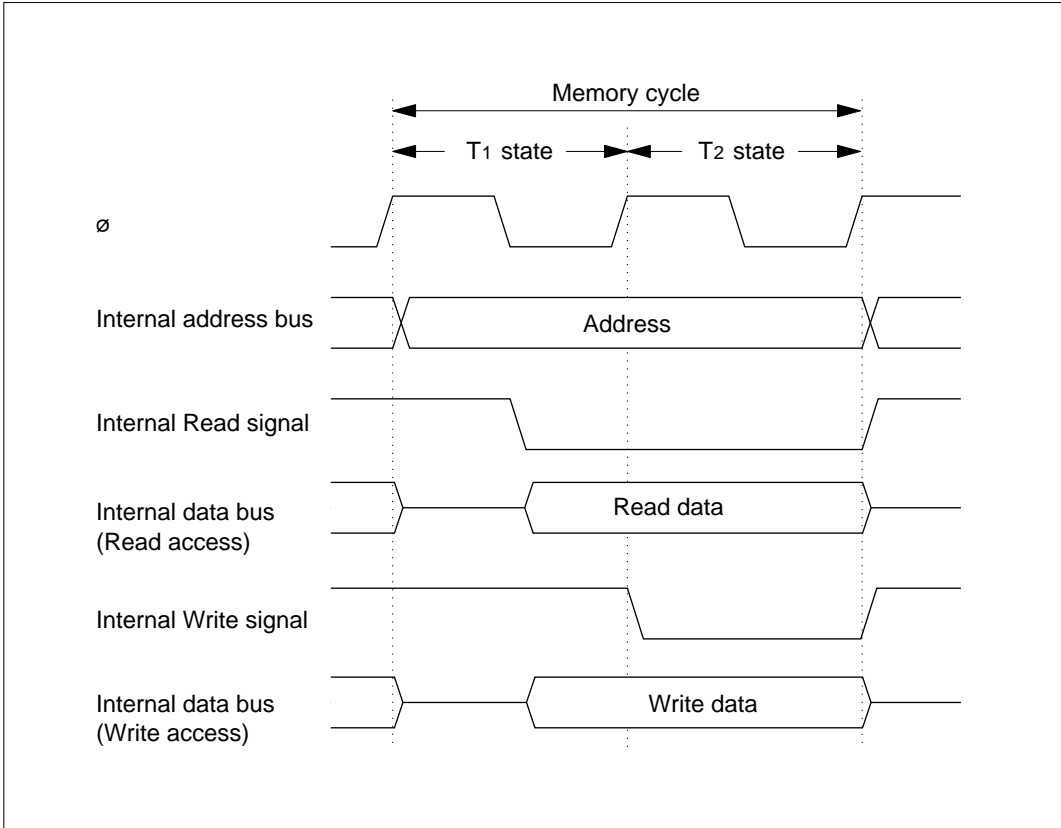
Figure 3-6 shows the on-chip memory access cycle. Figure 3-7 indicates the pin states. The bus control signals output from the H8/532 chip go to the nonactive state during the access.

**Access to On-Chip Register Field (Addresses H'FF80 to H'FFFF):** The access cycle consists of three states. The data bus is 8 bits wide.

Figure 3-8 shows the on-chip supporting module access cycle. Figure 3-9 indicates the pin states.

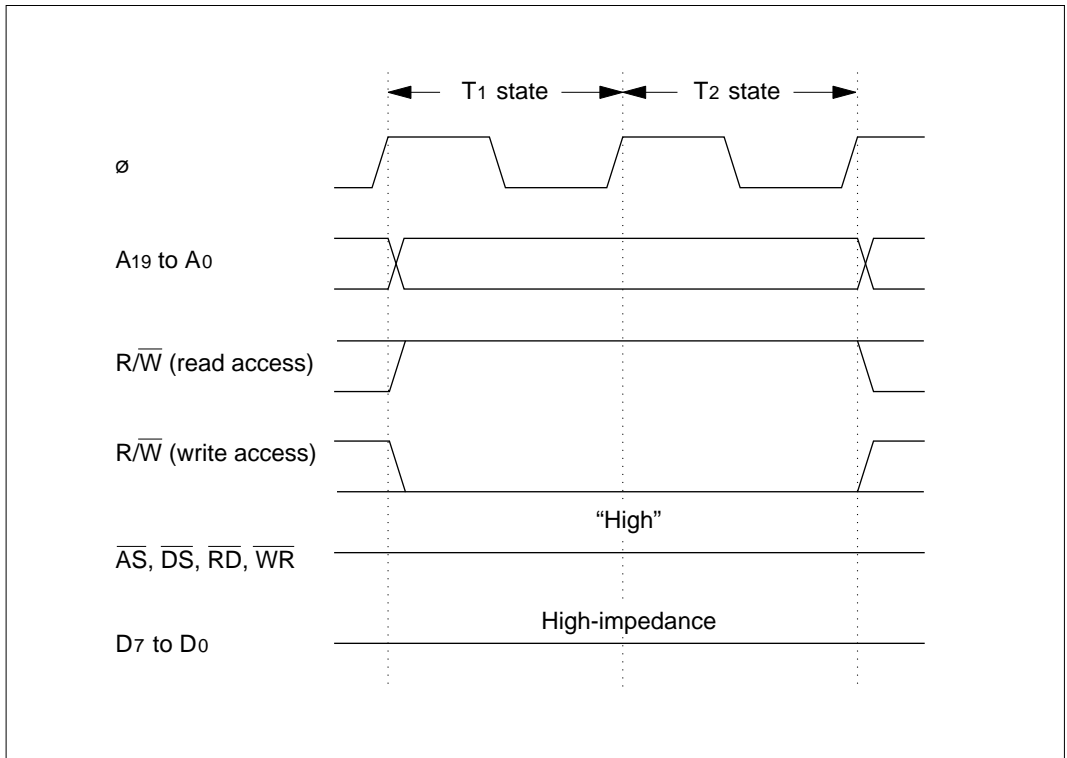
**Access to External Devices:** The access cycle consists of three states. The data bus is 8 bits wide. Figure 3-10 (a) and (b) shows the external access cycle. Additional wait states ( $T_w$ ) can be inserted by the wait-state controller (WSC).

### 3.7.2 On-Chip Memory Access Cycle



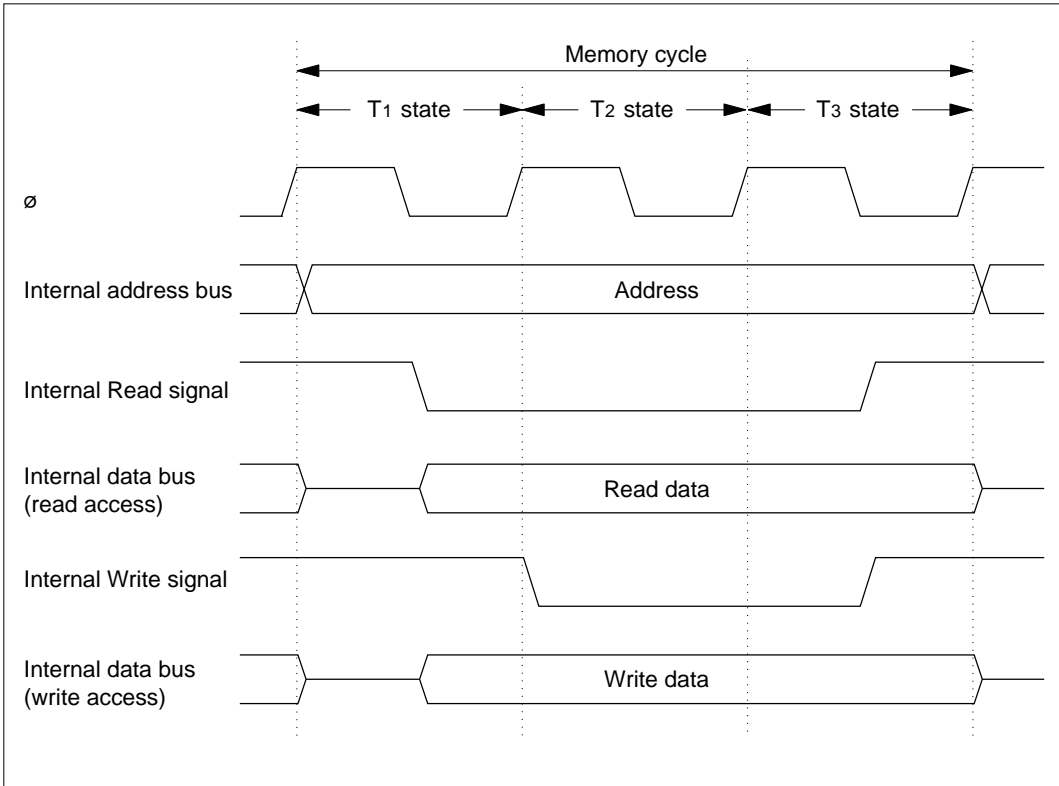
**Figure 3-6 On-Chip Memory Access Timing**

### 3.7.3 Pin States during On-Chip Memory Access



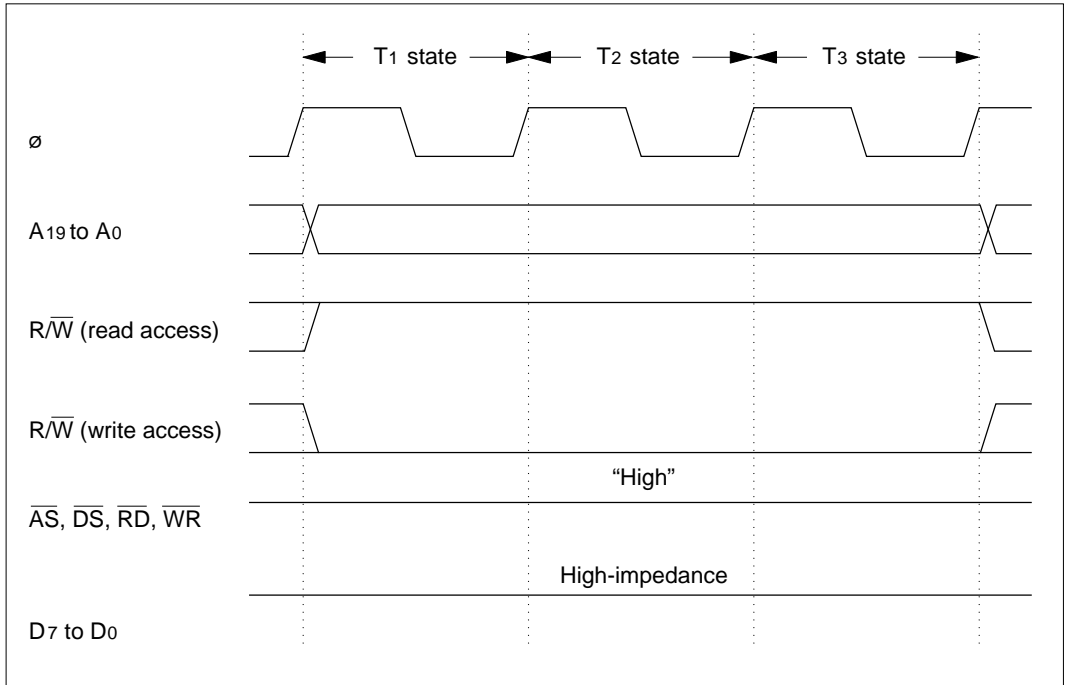
**Figure 3-7 Pin States during Access to On-Chip Memory**

### 3.7.4 Register Field Access Cycle (Addresses H'FF80 to H'FFFF)



**Figure 3-8 Register Field Access Timing**

### 3.7.5 Pin States during Register Field Access (Addresses H'FF80 to H'FFFF)



**Figure 3-9 Pin States during Register Field Access**

### 3.7.6 External Access Cycle

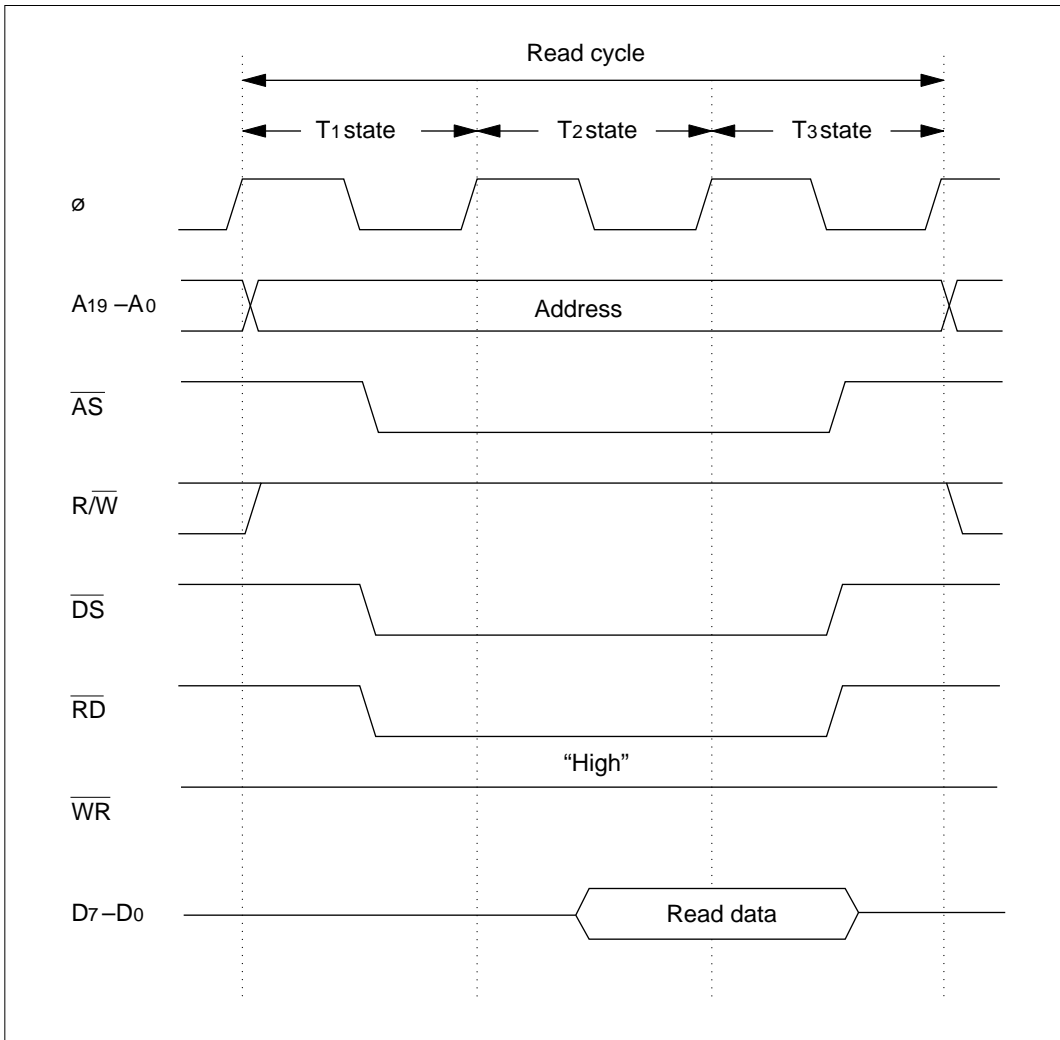
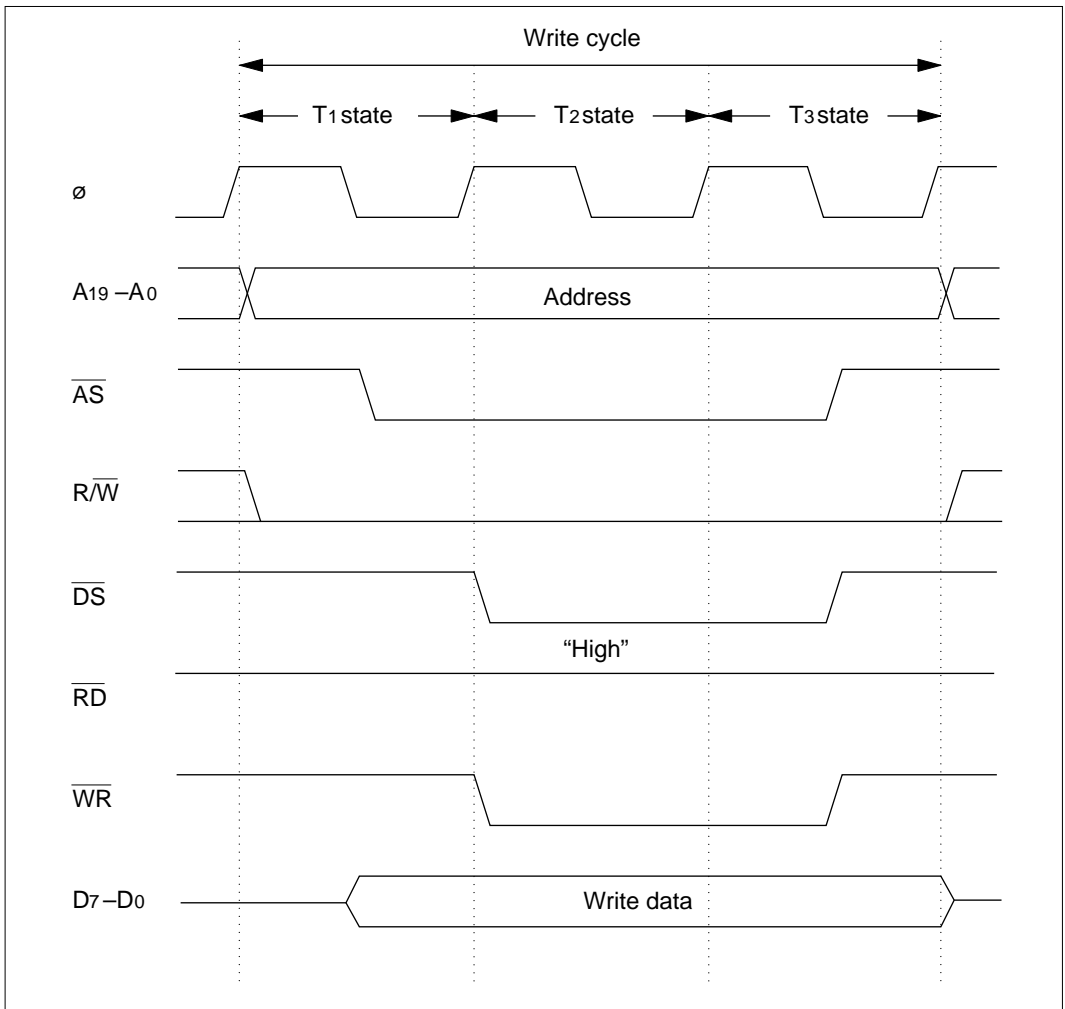


Figure 3-10 (a) External Access Cycle (Read Access)

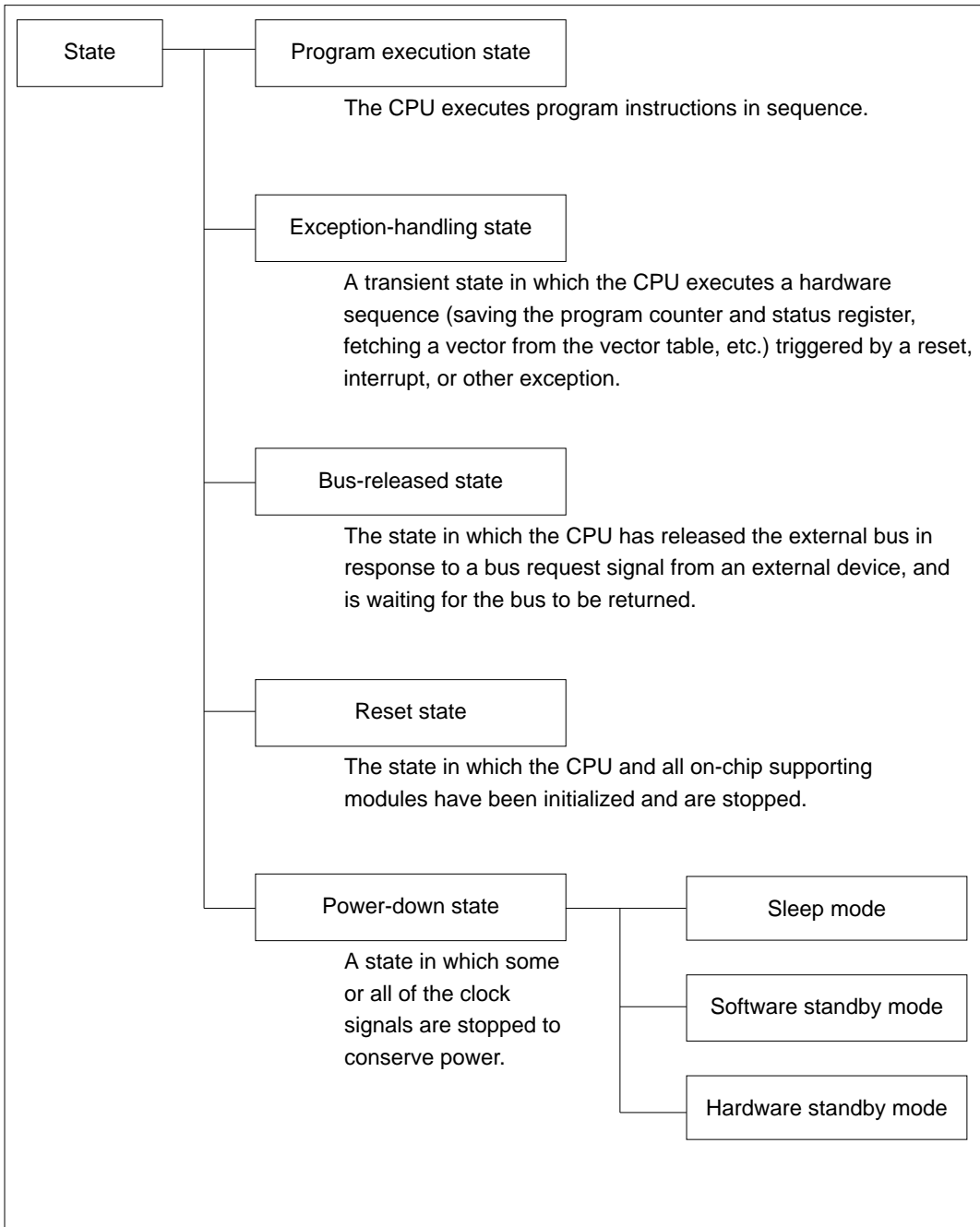


**Figure 3-10 (b) External Access Cycle (Write Access)**

## 3.8 CPU States

### 3.8.1 Overview

The CPU has five states: the program execution state, exception-handling state, bus-released state, reset state, and power-down state. The power-down state is further divided into the sleep mode, software standby mode, and hardware standby mode. Figure 3-11 summarizes these states, and figure 3-12 shows a map of the state transitions.



**Figure 3-11 Operating States**





In the hardware exception-handling sequence the CPU does the following:

1. Saves the program counter and status register (in minimum mode) or program counter, code page register, and status register (in maximum mode) to the stack.
2. Clears the T bit in the status register to “0.”
3. Fetches the start address of the exception-handling routine from the exception vector table.
4. Branches to that address, returning to the program execution state.

See section 4, “Exception Handling,” for further information on the exception-handling state.

### 3.8.4 Bus-Released State

When so requested, the CPU can grant control of the external bus to an external device. While an external device has the bus right, the CPU is said to be in the bus-released state. The bus right is controlled by two pins:

- $\overline{\text{BREQ}}$ : Input pin for the Bus Request signal from an external device
- $\overline{\text{BACK}}$ : Output pin for the Bus Request Acknowledge signal from the CPU, indicating that the CPU has released the bus

The procedure by which the CPU enters and leaves the bus-released state is:

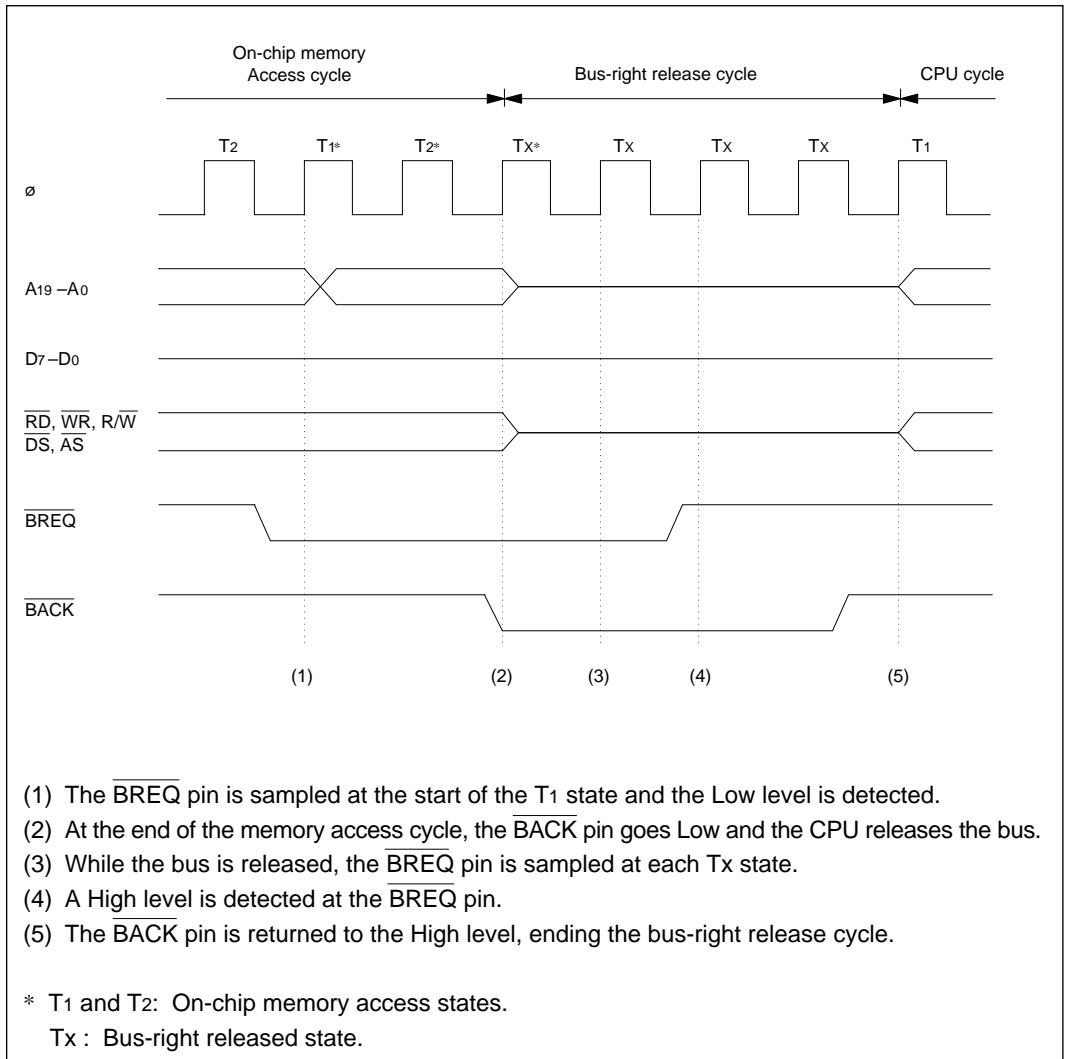
1. The CPU receives a Low  $\overline{\text{BREQ}}$  signal from an external device.
2. The CPU places the address bus pins ( $A_{19} - A_0$ ), data bus pins ( $D_7 - D_0$ ) and bus control pins ( $\overline{\text{RD}}$ ,  $\overline{\text{WR}}$ ,  $\text{R}/\overline{\text{W}}$ ,  $\overline{\text{DS}}$ , and  $\overline{\text{AS}}$ ) in the high-impedance state, sets the  $\overline{\text{BACK}}$  pin to the Low level to indicate that it has released the bus, then halts.
3. The external device that requested the bus (with the  $\overline{\text{BREQ}}$  signal) becomes the bus master. It can use the data bus and address bus. The external device is responsible for manipulating the bus control signals ( $\overline{\text{RD}}$ ,  $\overline{\text{WR}}$ ,  $\text{R}/\overline{\text{W}}$ ,  $\overline{\text{DS}}$ , and  $\overline{\text{AS}}$ ).
4. When the external device finishes using the bus, it clears the  $\overline{\text{BREQ}}$  signal to the High level. The CPU then reassumes control of the bus and returns to the program execution state.

**Bus Release Timing:** The CPU can release the bus right at the following times:

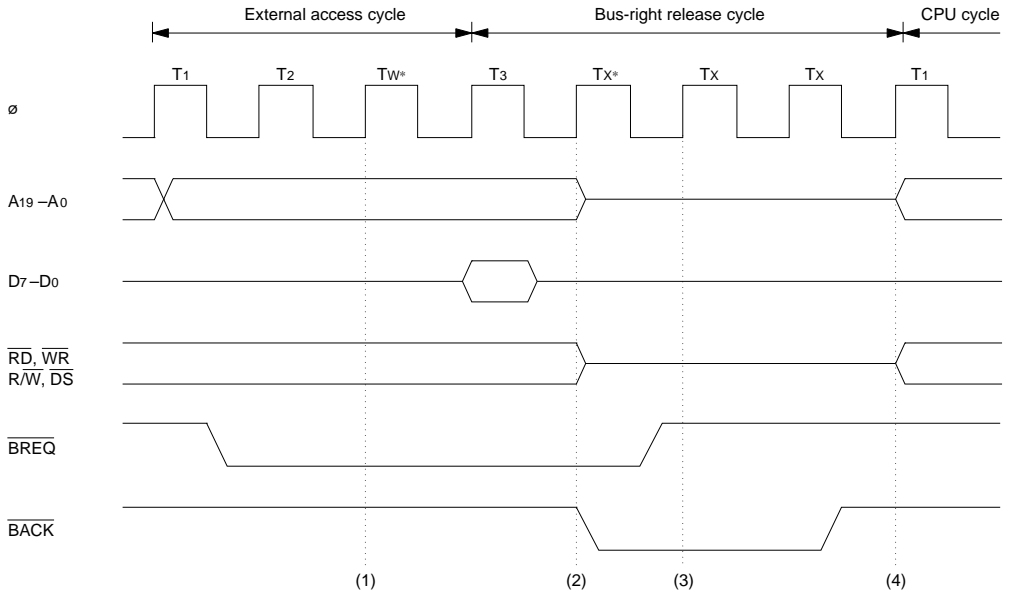
1. The  $\overline{\text{BREQ}}$  signal is sampled during every memory access cycle (instruction prefetch or data read/write). If  $\overline{\text{BREQ}}$  is Low, the CPU releases the bus right at the end of the cycle. (In word data access to external memory or an address from H'FF80 to H'FFFF, the CPU does not release the bus right until it has accessed both the upper and lower data bytes.)
2. During execution of the MULXU and DIVXU instructions, since considerable time may pass without an instruction prefetch or data read/write,  $\overline{\text{BREQ}}$  is also sampled at internal machine cycles, and the bus right is released if  $\overline{\text{BREQ}}$  is Low.
3. The bus right can also be released in the sleep mode.

The CPU does not recognize interrupts while the bus is released.

**Timing Charts:** Timing charts of the operation by which the bus is released are shown in figure 3-13 for the case of bus release during an on-chip memory read cycle, in figure 3-14 for bus release during an external memory read cycle, and in figure 3-15 for bus release while the CPU is performing an internal operation.



**Figure 3-13 Bus-Right Release Cycle (During On-Chip Memory Access Cycle)**

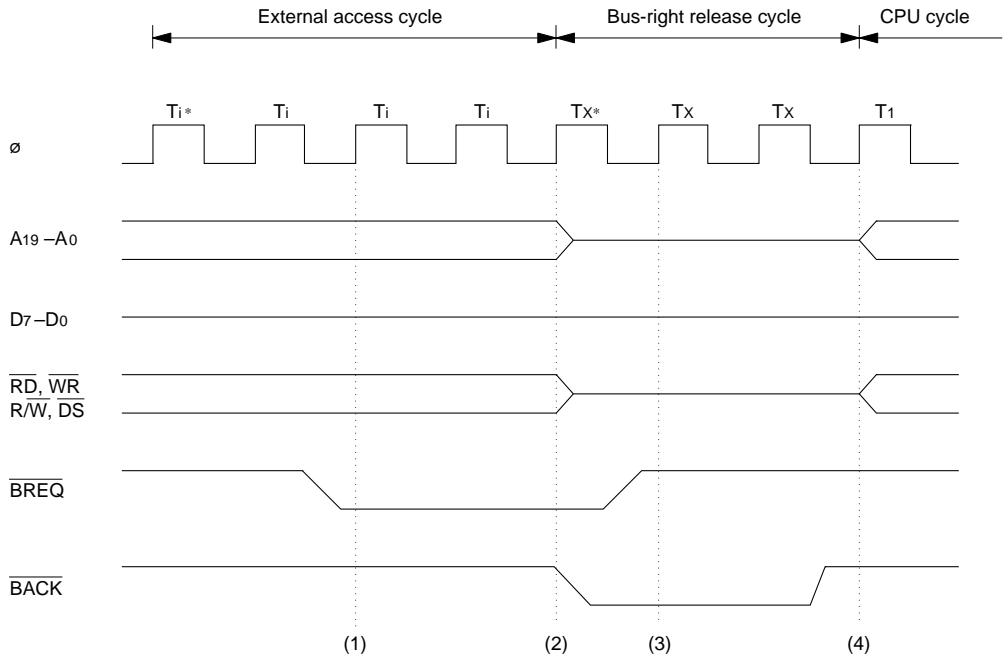


- (1) The  $\overline{\text{BREQ}}$  pin is sampled at the start of the Tw state and the Low level is detected.
- (2) At the end of the external access cycle, the  $\overline{\text{BACK}}$  pin goes Low and the CPU releases the bus.
- (3) The  $\overline{\text{BREQ}}$  pin is sampled at the Tx state and a High level is detected.
- (4) The  $\overline{\text{BACK}}$  pin is returned to the High level, ending the bus-right release cycle.

\* Tw : Wait state.

Tx : Bus-right released state.

**Figure 3-14 Bus-Right Release Cycle (During External Access Cycle)**



- (1) The  $\overline{\text{BREQ}}$  pin is sampled at the start of a  $T_i$  state and the Low level is detected.
- (2) At the end of the internal operation cycle, the  $\overline{\text{BACK}}$  pin goes Low and the CPU releases the bus.
- (3) The  $\overline{\text{BREQ}}$  pin is sampled at the  $T_x$  state and a High level is detected.
- (4) The  $\overline{\text{BACK}}$  pin is returned to the High level, ending the bus-right release cycle.

\*  $T_i$  : Internal CPU operation state.  
 $T_x$  : Bus-right released state.

**Figure 3-15 Bus-Right Release Cycle (During Internal CPU Operation)**

**Notes:** The  $\overline{\text{BREQ}}$  signal must be held Low until  $\overline{\text{BACK}}$  goes Low. If  $\overline{\text{BREQ}}$  returns to the High level before  $\overline{\text{BACK}}$  goes Low, the bus release operation may be executed incorrectly.

To leave the bus-released state, the High level at the  $\overline{\text{BREQ}}$  pin must be sampled two times. If the  $\overline{\text{BREQ}}$  returns to Low before it is sampled two times, the bus released cycle will not end.

The bus release operation is enabled only when the BRLE bit in the port 1 control register (P1CR) is set to “1.” When this bit is cleared to “0” (its initial value), the  $\overline{\text{BREQ}}$  and  $\overline{\text{BACK}}$  pins are used for general-purpose input and output, as P13 and P12.

An instruction that sets the BRLE bit is: `BSET.B #3, @H'FFFC`

Note the following point when using the H8/532's release function.

If the  $\overline{\text{BREQ}}$  signal is asserted and an interrupt is requested simultaneously during execution of the SLEEP instruction, the  $\overline{\text{BACK}}$  signal may fail to be output even though the CPU has released the bus. This may cause the system to stop for the interval during which  $\overline{\text{BREQ}}$  is asserted, with no device in control of the bus. The interrupts that can cause this state include NMI, IRQ, and all the interrupts from on-chip supporting modules. When the  $\overline{\text{BREQ}}$  signal is deasserted, ending this state, the CPU takes control of the bus again and resumes normal instruction execution.

The following methods can be used to avoid entering this state.

**Method 1:** If the  $\overline{\text{BREQ}}$  signal is used, do not use the SLEEP instruction.

**Method 2:** Disable the  $\overline{\text{BREQ}}$  signal during execution of the SLEEP instruction. This can be done by clearing the bus release enable bit (BRLE) in the port 1 control register (P1CR) to 0 immediately before executing the SLEEP instruction. (When the BRLE bit is cleared, low inputs on the  $\overline{\text{BREQ}}$  line are not latched on-chip.) Place instructions to set the BRLE bit to 1 at the beginning of interrupt-handling routines. If the data transfer controller (DTC) is used, place an instruction to set the BRLE bit immediately after the SLEEP instruction.

If method 2 is used,  $\overline{\text{BREQ}}$  inputs will be ignored while the chip is in sleep mode.

(Coding example)

#### Main Program

```
-----  
BCLR.B #3, @P1CR  
SLEEP  
BSET.B #3, @P1CR  
-----
```

#### Interrupt-Handling Routine

```
BSET.B #3, @P1CR  
-----  
RTE
```

### **3.8.5 Reset State**

In the reset state, the CPU and all on-chip supporting modules are initialized and placed in the stopped state. The CPU enters the reset state whenever the  $\overline{\text{RES}}$  pin goes Low, unless the CPU is currently in the hardware standby mode. It remains in the reset state until the  $\overline{\text{RES}}$  pin goes High.

See section 4.2, “Reset,” for further information on the reset state.

### **3.8.6 Power-Down State**

The power-down state comprises three modes: the sleep mode, the software standby mode, and the hardware standby mode.

See section 18, “Power-Down State,” for further information.

## 3.9 Programming Notes

### 3.9.1 Restriction on Address Location

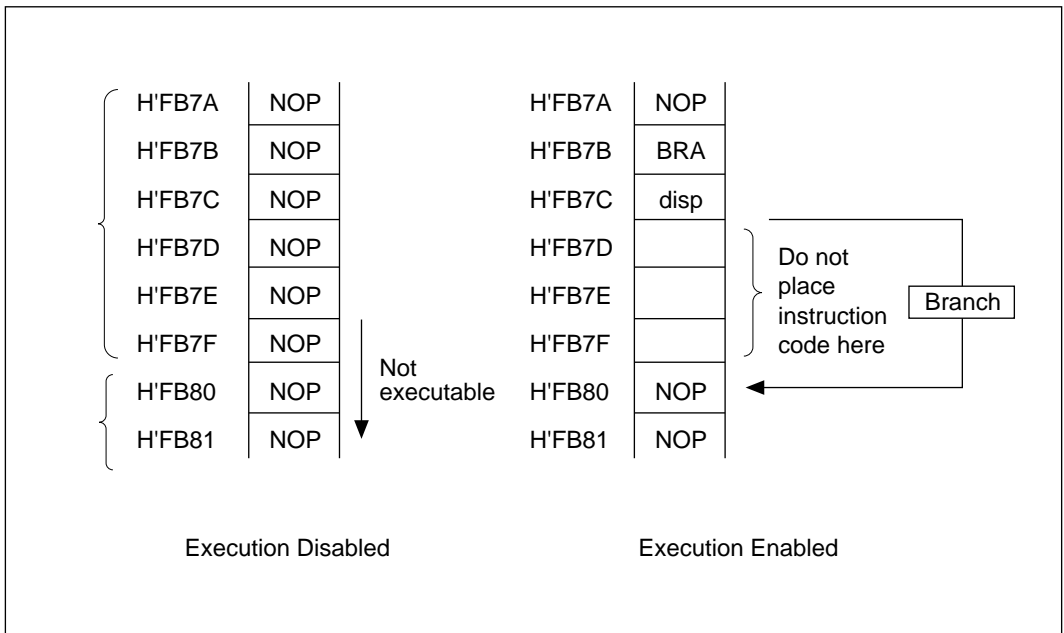
The following restriction applies when instructions are located in on-chip RAM.

- Restriction

Instruction execution cannot proceed continuously from an external address to on-chip RAM in the ZTAT versions. This restriction does not apply to versions with masked ROM.

- Solution

To execute instructions located in on-chip RAM, use a branch instruction (examples: Bcc, JMP, etc.) to branch to the first instruction located in on-chip RAM. Do not place instruction code in the last three bytes of external memory (H'FB7D to H'FB7F).





### 3.9.2 Note on MULXU Instruction

Note that in the case described below, the H8/532 multiply instruction does not give correct results.

#### (1) Problem

The result of a squaring operation such as `MULXU.B Rn, Rn` is indeterminate. This problem occurs when the same register is specified for the source and destination of a byte multiplication operation.

This problem occurs only in ZTAT versions of the H8/532. It does not occur in versions with masked ROM.

#### (2) Solution

The problem can be avoided by the following methods.

- ① Place the source and destination operands in different registers.

Example: `MULXU.B R4, R4` → `MOV.W R4, R5`  
`MULXU.B R5, R4`

- ② Use a word multiplication instruction.

Example: `MULXU.B R4, R4` → `MULXU.W R4, R4`  
`MOV.W R5, R4`

- ③ Place one of the operands in memory.

Example: `MULXU.B R4, R4` → `MOV.W R4, @-SP`  
`MULXU.B @(1,SP), R4`  
`ADDS #2, SP`

This problem occurs only in the H8/532. It does not occur in other chips in the H8/500 Series (such as the H8/520).

#### (3) Note on usage of C compiler

Programmers using the C compiler should bear the following programming note in mind.

- Conditions under which the compiler generates a `MULXU.B Rn, Rn` instruction

The C compiler generates a `MULXU.B Rn, Rn` instruction when the following two conditions are satisfied in the source program:

① A one-byte variable (char or unsigned char) is declared as a register variable.

② The variable declared as in ① is squared by compound substitution

Example: register char a;  
a \*= a;

- Solution

The problem can be avoided as follows:

① In the example above, do not declare the variable (a) as a register variable.

Example: register char a;                   →     char a;  
a \*= a;   a \*= a;

② When squaring one-byte data, do not use compound substitution. Code as follows:

Example: a \*= a;                             →     a = a \* a;