OKI SEMICONDUCTOR CO., LTD.

FEZ0317A0-U8-INST-02

nX-U8/100 Core
Instruction Manual

CMOS 8-bit microcontroller

Issue Date: Oct. 17th, 2008

NOTICE

1. The information contained herein can change without notice owing to product and/or technical improvements. Before using the product, please make sure that the information being referred to is up-to-date.

2. The outline of action and examples for application circuits described herein have been chosen as an explanation for the standard action and performance of the product. When planning to use the product, please ensure that the external conditions are reflected in the actual circuit, assembly, and program designs.

3. When designing your product, please use our product below the specified maximum ratings and within the specified operating ranges including, but not limited to, operating voltage, power dissipation, and operating temperature.

4. OKI SEMICONDUCTOR assumes no responsibility or liability whatsoever for any failure or unusual or unexpected operation resulting from misuse, neglect, improper installation, repair, alteration or accident, improper handling, or unusual physical or electrical stress including, but not limited to, exposure to parameters beyond the specified maximum ratings or operation outside the specified operating range.

5. Neither indemnity against nor license of a third party's industrial and intellectual property right, etc. is granted by us in connection with the use of the product and/or the information and drawings contained herein. No responsibility is assumed by us for any infringement of a third party's right which may result from the use thereof.

6. The products listed in this document are intended for use in general electronics equipment for commercial applications (e.g., office automation, communication equipment, measurement equipment, consumer electronics, etc.). These products are not, unless specifically authorized by OKI SEMICONDUCTOR, authorized for use in any system or application that requires special or enhanced quality and reliability characteristics nor in any system or application where the failure of such system or application may result in the loss or damage of property, or death or injury to humans.
   Such applications include, but are not limited to, traffic and automotive equipment, safety devices, aerospace equipment, nuclear power control, medical equipment, and life-support systems.

7. Certain products in this document may need government approval before they can be exported to particular countries. The purchaser assumes the responsibility of determining the legality of export of these products and will take appropriate and necessary steps at their own expense for these.

8. No part of the contents contained herein may be reprinted or reproduced without our prior permission.

Copyright 2008 OKI SEMICONDUCTOR CO.,LTD.

Contents

1. Architecture 1-1

2. Addressing Types 2-1

3. Instruction Descriptions 3-1

4. Appendix 4-1

1. Architecture

1.1 Overview

1.1.1 Features

The U8 architecture has the following features.

- **Powerful Instruction Set**

  Instructions for data transfers, arithmetic, comparison, logic operations, bit manipulation, bitwise logic operations, branches, conditional branches, call/return stack manipulation, and arithmetic shifts

- **Variety of Addressing Modes**

  Register addressing
  Register indirect addressing
  Stack pointer addressing
  Control register addressing
  EA register indirect addressing
  General-purpose register indirect addressing
  Direct addressing
  Register indirect bit addressing
  Direct bit addressing

- **Memory Spaces**

  Program/code memory (ROM)

  Up to 16 segments of 32 kilowords (0000H-FFFFH) each

  Data memory (RAM)

  Up to 256 segments of 64 kilobytes (0000H-FFFFH) each

- **Interrupts**

  Dedicated emulator interrupts
  Nonmaskable interrupts
  Maskable interrupts
  Software interrupts

1.2 CPU Resources and Programming Model

The U8 architecture features two address spaces: 1 megabyte for code and 16 megabytes for data. Both address spaces are divided into physical segments of 64 kilobytes each. The memory configuration for physical segment #0 (0:0000H to 0:FFFFH) differs, however, from the others.

Physical segment #0 provides two sets of addresses and separate registers for accessing them: a 32-kiloword program/code memory segment accessed with the program counter (PC) and a 64-kilobyte data memory segment accessed with the address register (AR). If the address in AR is within the ROM window, however, the register accesses program/code memory, not the data memory.

Physical segments #1 and higher, with addresses above the first 64 kilobytes, form a single address space mixing program/code and data memory. Accessing a physical segment assigned to program/code memory requires a 20-bit address (CSR:PC) combining four bits from the code segment register (CSR) and 16 bits from program counter (PC); a physical segment assigned to data memory, a 24-bit address (DSR:AR) combining eight bits from the data segment register (DSR) and 16 bits from the address register (AR).

Figure 1.1 summarizes the layout of these U8 memory spaces.

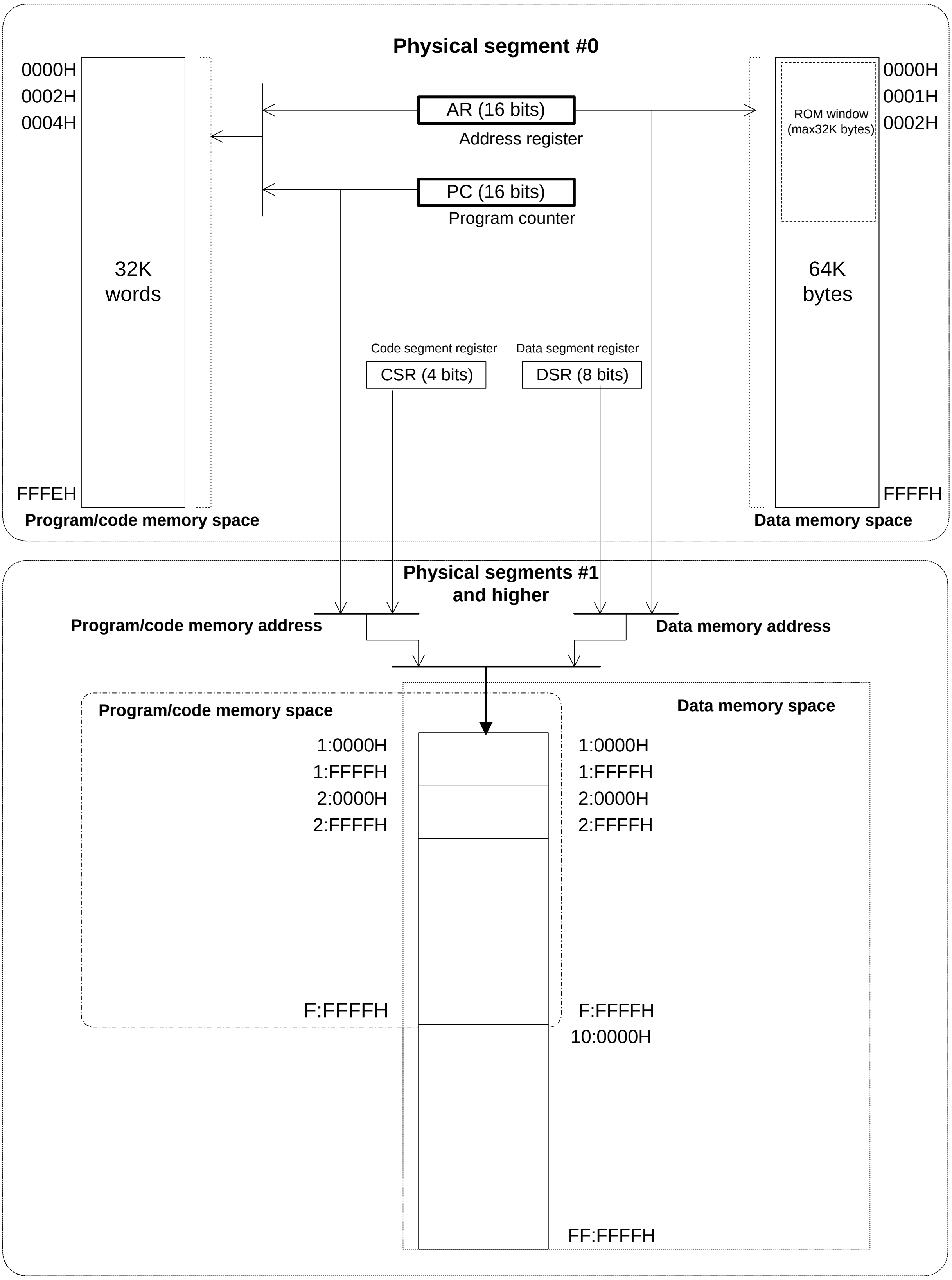


Figure 1.1. U8 Memory Spaces

1.2.1 Registers

General registers lie at the center of U8 hardware operation. Also shown in Figure 1.2 are the control registers.

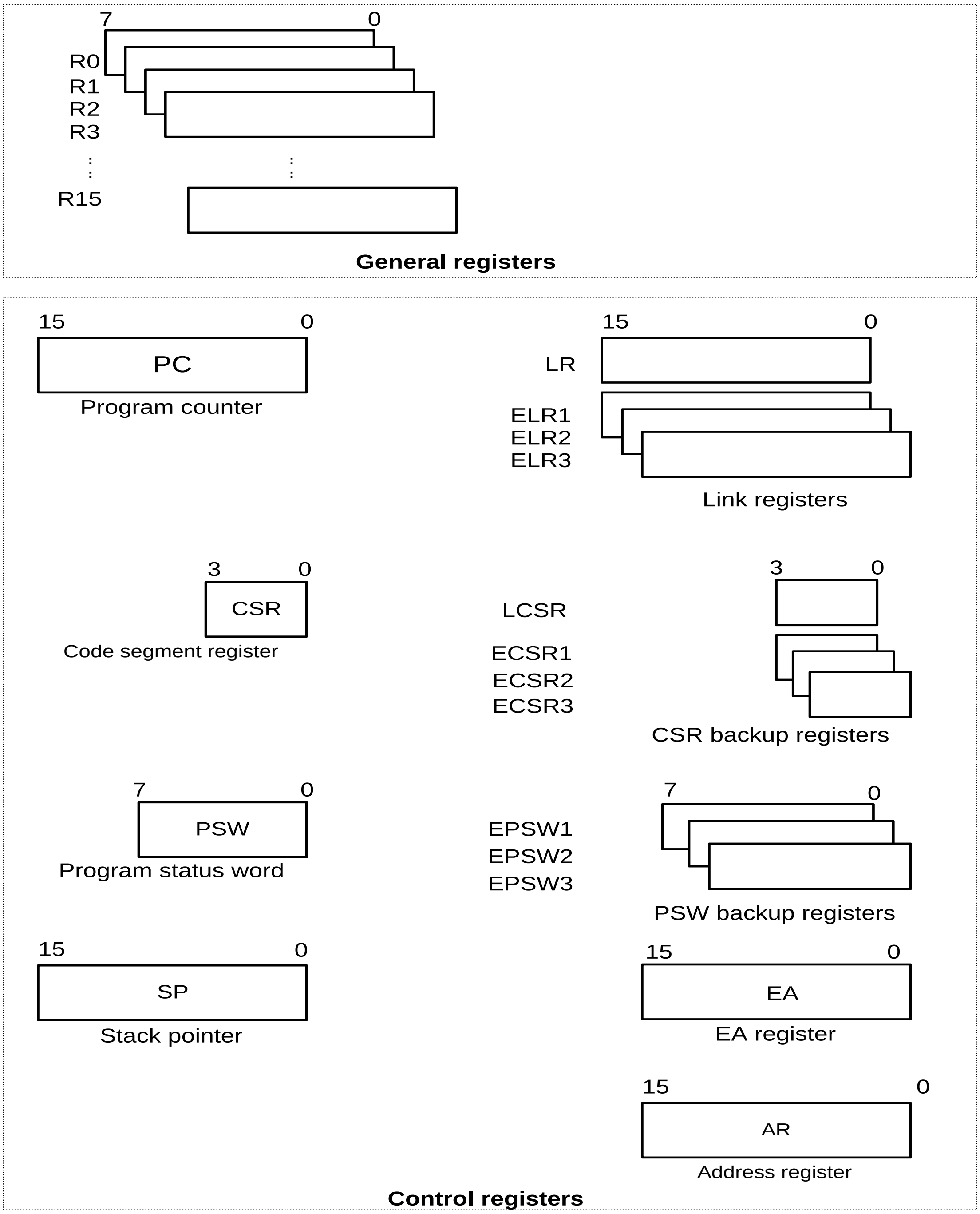


Figure 1.2.　Register Set

1.2.1.1 General Registers

These 16 registers at the center of calculations are one byte wide. Special addressing modes, however, also group adjacent registers together to permit access as eight word-sized registers (ERn), four double word-sized registers (XRn), and two quad word-sized registers (QRn).

If an interrupt handler modifies the contents of these registers, it must explicitly save them with PUSH instructions at its entry point and restore them with POP instructions before returning.

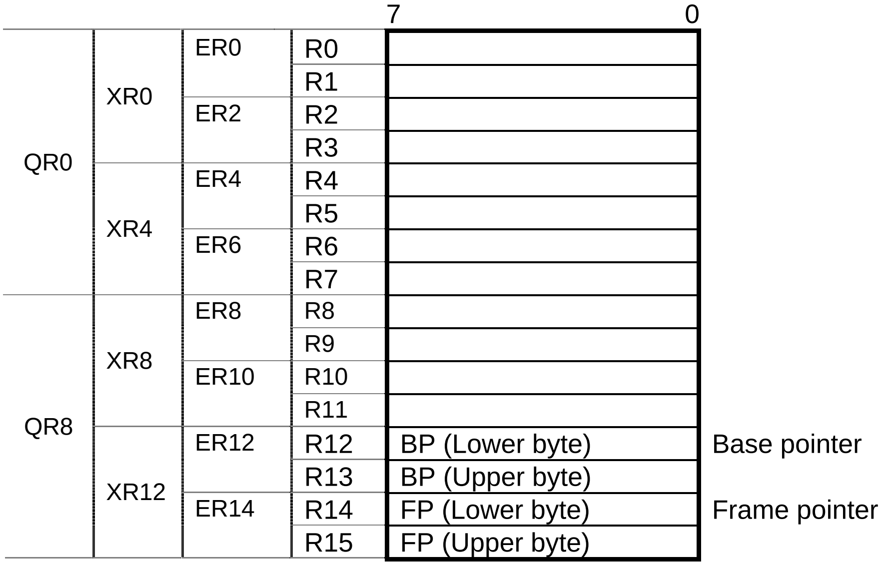


Figure 1.3. General Registers

Examples: Using general registers

```
MOV   R0   ,  #7      ; byte-sized register
L     ER0  , [EA+]    ; word-sized register
L     XR0  , [EA]     ; double word-sized register
ST    QR0  , [EA]     ; quad word-sized register
SB    R3.2            ; individual bit in register
```

1.2.1.2 Base and Frame Pointers

The C compiler uses two global pointers. It uses ER12 as the base pointer (BP) and ER14 as the frame pointer (FP). These two registers therefore offer special addressing modes in addition to their roles as general registers. For further details, see Chapter 2.

1.2.2 Control Registers

These registers control program flow and hold operational status information. There are 18 such registers, each with its own special function. The contents of the entire group is sometimes referred to as the program context.

1.2.2.1 Program Status Word (PSW)

	7	6	5	4	3	2	1	0
PSW	C	Z	S	OV	MIE	HC	ELEVEL	

This 8-bit register contains five flags tracking the results of instruction execution, one control bit, and one field.

The hardware automatically saves these contents to an exception program status word (EPSW) register when it accepts an interrupt request. The RTI instruction at the end of the interrupt handler restores them.

This register contains five flags tracking the results of arithmetic instruction execution, one bit controlling interrupt acceptance, and a 2-bit field indicating the exception level (ELEVEL). The program can change these contents at any time. After a reset, they are all zero.

These flags, bit, and field have the following functions.

- Bit 7: Carry flag (C)

  This bit goes to “1” if an arithmetic, shift, or comparison instruction produces a carry out of bit 7 or bit 0 or a borrow into bit 7. Otherwise, it goes to “0.”

  The contents can also be directly set, reset, and inverted with the SC, RC, or CPLC instructions and tested with the conditional branch instructions.

- Bit 6: Zero flag (Z)

  This bit goes to “1” if an arithmetic or data transfer instruction produces a zero result. Otherwise, it goes to “0.”

  The contents can be tested with the conditional branch instructions.

- Bit 5: Sign flag (S)

  This bit tracks the sign bit in the result from an arithmetic, comparison, or bitwise logical instruction: “1” for negative, “0” for positive.

- Bit 4: Overflow flag (OV)

  This bit goes to “1” if a signed arithmetic instruction produces a carry out of or a borrow into bit 7—that is, a result that does not fit into the twos complement range available. Otherwise, it goes to “0.”

- Bit 3: Master interrupt enable bit (MIE)

  This bit is a mask controlling the acceptance of maskable interrupt requests. Setting it to “1” enables such interrupt requests; “0” disables them.

  The hardware automatically sets this bit to “0” when it accepts a maskable interrupt request. The contents can also be directly set or reset with the EI and DI instructions.

- Bit 2: Half carry flag (HC)

  This bit, used in BCD arithmetic, goes to “1” if an arithmetic or comparison instruction produces a carry out of or a borrow into bit 3 or bit 11. Otherwise, it goes to “0.”

- Bits 1 and 0: Exception level (ELEVEL)

  This field gives the current exception level, an integer between 0 and 3 indicating the interrupt priority. The higher this number, the greater the priority. For a list of interrupts and their exception/priority levels, see Section 1.3.7 “Interrupt Operation.”

  The U8 hardware accepts an interrupt request only if its interrupt priority is the same or greater than the current exception level (ELEVEL) setting.

1.2.2.1.1 Instructions Modifying PSW Flags

For further details on instructions modifying PSW flags and the exact nature of those modifications, see Section 3.2 “Instructions by Functional Group” and Chapter 4 “Appendix.”

1.2.2.2 Program Counter (PC)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PC																

This 16-bit register holds the offset portion of the address of the next instruction to be executed. The hardware automatically increments it immediately after fetching an instruction from program/code memory, creating the cycle necessary for sequential execution. Branch and other instructions, however, break this cycle by overwriting this default with a different address.

Instructions always start on word boundaries, so the hardware increments the program counter (PC) by two each time and forces the lowest bit in any address loaded to “0” to enforce this alignment.

After a reset, the program counter (PC) starts with the contents of the vector corresponding to the reset factor.

When the hardware accepts an interrupt request, it automatically saves the contents of this register for use as part of the return address in the exception link register (ELR1 to ELR3) for the current exception level (ELEVEL) setting. The RTI instruction at the end of the interrupt handler restores them.

1.2.2.3 Code Segment Register (CSR)

	3	2	1	0
CSR				

This 4-bit register holds the physical segment number (0 to 15) portion of the address for the current instruction. The remaining 16 bits (0 to FFFFH), representing an offset within that physical segment, come from the program counter (PC). Together, these two registers specify a 20-bit address (CSR:PC) accessing the entire program/code memory space.

Address calculations apply only to the 16-bit offset, ignoring any over- or underflow, so never modify the CSR contents. The same applies to PC overflow. Program execution thus continuously cycles through the addresses in the same physical segment until the program explicitly overwrites the CSR contents.

The following actions modify the CSR contents.

- interrupt acceptance: CSR goes to zero.
- reset : CSR goes to zero.
- B *Cadr* instruction: CSR goes to the value specified in the instruction.
- BL *Cadr* instruction: CSR goes to the value specified in the instruction.
- RTI instruction: CSR goes to the value from the ECSR register corresponding to the current

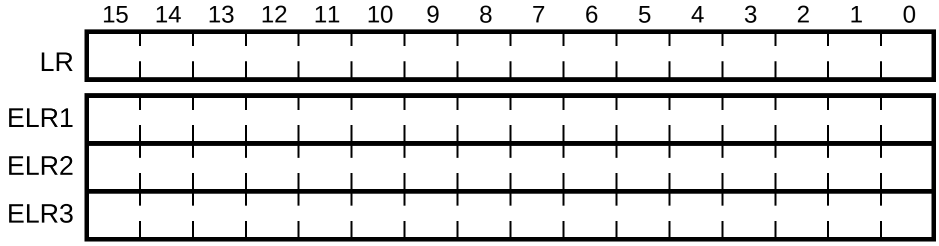
exception level (ELEVEL) setting from program status word (PSW).

- RT instruction: CSR goes to the value from the LCSR register.
- POP PC instruction: CSR goes to the value from the stack.

When the hardware accepts an interrupt request, it automatically saves the contents of this register for use as part of the return address in the ECSR register (ECSR1 to ECSR3) for the current exception level (ELEVEL) setting. The RTI instruction at the end of the interrupt handler restores them.

After a reset, this register contains zero.

1.2.2.4 Link Registers (LR, ELR1, ELR2, and ELR3)



These four 16-bit registers are for saving the contents of the program counter (PC) during subroutines (LR) and interrupt handlers (ELR1 to ELR3). The lowest bit is always “0.”

The LR register holds the offset portion of the return address for a subroutine called with a BL instruction. The RT instruction at the end of the subroutine loads the LR contents back into the program counter (PC).

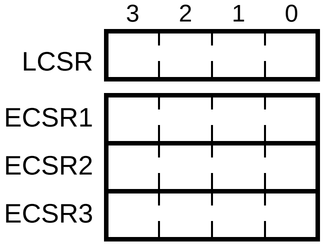
Note that the program has a choice of two instructions for returning from a subroutine to its caller: RT or POP. For further details, see Section 1.4 “Exception Levels and Backup Registers.”

The registers ELR1 to ELR3 hold the offset portions of the return addresses for interrupt handlers at the corresponding exception levels. The hardware saves the return address using the index number assigned to the interrupt being accepted. For a list of interrupts and their exception/priority levels, see Section 1.3.7 “Interrupt Operation.”

Note that modifying the ELEVEL portion of the program status word (PSW) in software requires particular care because it changes the index pointing to the most recently used ELR-ECSR register pair.

Note also that the ELR3-ECSR3 register pair is only physically present in models including the on-chip debugger. Accessing these registers on other models leads to unpredictable operation. Always check the User’s Manual for the target device first.

1.2.2.5 CSR Backup Registers (LCSR, ECSR1, ECSR2, and ECSR3)



These four 4-bit registers are for saving the contents of the code segment register (CSR) during subroutines (LCSR) and interrupt handlers (ECSR1 to ECSR3).

The LCSR register holds the physical segment portion of the return address for a subroutine called with a BL instruction. The RT instruction at the end of the subroutine loads the LCSR contents back into the code segment register (CSR).

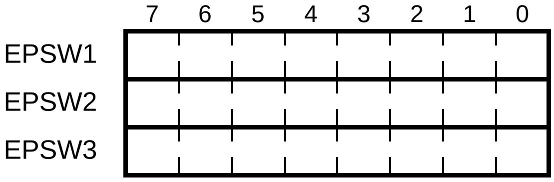
Note that the program has a choice of two instructions for returning from a subroutine to its caller: RT or POP. For further details, see Section 1.4 “Exception Levels and Backup Registers.”

The registers ECSR1 to ECSR3 hold the physical segment portions of the return addresses for interrupt handlers at the corresponding exception levels. The hardware saves the return address using the index number assigned to the interrupt being accepted. For a list of interrupts and their exception/priority levels, see Section 1.3.7 “Interrupt Operation.”

Note that modifying the ELEVEL portion of the program status word (PSW) in software requires particular care because it changes the index pointing to the most recently used ELR-ECSR register pair.

Note also that the ELR3-ECSR3 register pair is only physically present in models including the on-chip debugger. Accessing these registers on other models leads to unpredictable operation. Always check the User’s Manual for the target device first.

1.2.2.6 PSW Backup Registers (EPSW1, EPSW2, and EPSW3)



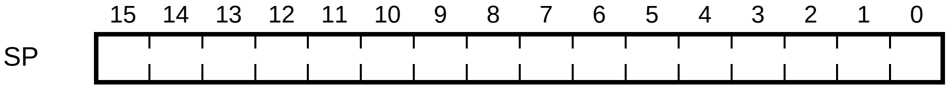
These three 8-bit registers are for saving the contents of the program status word (PSW) during interrupt handlers.

The hardware saves the program status word (PSW) using the index number assigned to the interrupt being accepted. For a list of interrupts and their exception/priority levels, see Section 1.3.7 “Interrupt Operation.”

Note that modifying the ELEVEL portion of the program status word (PSW) in software requires particular care because it changes the index pointing to the most recently used EPSW register.

Note also that the EPSW3 register is only physically present in models including the on-chip debugger. Accessing this register on other models leads to unpredictable operation. Always check the User’s Manual for the target device first.

1.2.2.7 Stack Pointer (SP)



This 16-bit register holds a pointer to the start of the stack for saving and restoring the contents of registers—with the PUSH and POP instructions, for example.

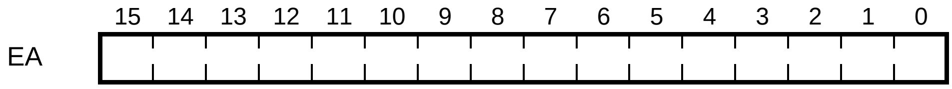
Stack operations are always word sized. One saving word-sized data to the stack subtracts 2 from this register and then copies the data to that new address. Restoring data copies a word from the stack to the specified destination and then adds 2 to this register.

There is no automatic word alignment. If the contents of this register are odd, that address is used as is.

This register is an independent one, fully accessible from programs with the appropriate instructions—PUSH and POP, for example.

After a reset, this register contains the contents of addresses 0000H and 0001H in the program/code memory in its lower and upper bytes, respectively.

1.2.2.8 EA Register (EA)

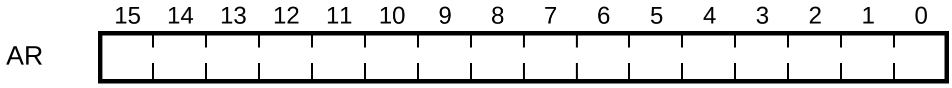


This 16-bit register holds an address for use by instructions that access data memory indirectly via this register.

These 16 bits are sufficient for accessing data memory addresses in physical segment #0. Accessing physical segments #1 and higher, however, requires prefixing this offset with the contents of the data segment register (DSR), described below, to form a 24-bit address (DSR:EA).

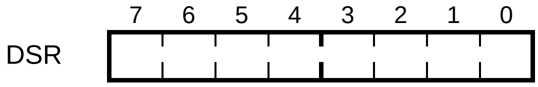
This register is accessible from programs with the LEA instruction for loading it and with the stack manipulation instructions PUSH and POP.

1.2.2.9 Address Register (AR)



This 16-bit register temporarily holds an address for use by instructions accessing data memory. It is for the exclusive use by the U8 core, so is not accessible from programs.

1.2.2.10 Data Segment Register (DSR)



This 8-bit register holds a physical segment number for accessing data memory in physical segments #1 and higher. This number can be anywhere between 0 and 255.

Accessing addresses within the specified physical segment uses the 16-bit offset (0 to FFFFH) in the EA address—that is, a 24-bit address with the contents of this register in the top eight bits and the contents of the EA register in the lower 16 bits.

Memory access instructions specifying a numeric value for the physical segment first update DSR to this new value. Those with the notation DSR in that position use the current contents of this register. In the absence of either notation, the instruction always ignores the DSR contents and uses physical segment #0.

The following code fragment gives some examples of data memory access.

```
   L      R0       ,5:1234H   ;    Set DSR to 5 and load R0 from 5:1234H, an address in
                                   physical segments #1 and higher.
;
   LEA             55AAH
   ST     R0       ,3:[EA+]   ;    Set DSR to 3 and store the contents of R0 in 3:55AAH, an
                                   address in physical segments #1 and higher.
                              ;    Increment EA.
   ST     R1       ,3:[EA+]   ;    Set DSR to 3 and store the contents of R1 in 3:55ABH, an
                                   address in physical segments #1 and higher.
                              ;    Increment EA.
   ST     R2       ,3:[EA+]   ;    Set DSR to 3 and store the contents of R2 in 3:55ACH, an
                                   address in physical segments #1 and higher.
                              ;    Increment EA.
;
   L      R0       ,5:1234H   ;   Set DSR to 5 and load R0 from 5:1234H, an address in
                                  physical segments #1 and higher.
   L      R1       ,1234H     ;   Load R1 from offset 1234H in data memory physical segment
                                  #0.
   L      R2       ,01235H    ;   Set DSR to 0 and load R2 from offset 1235H in data memory
                                  physical segment #0.
;
   LEA    AA55H               ;
   L      R5,DSR: [EA+]       ;   Load R5 from the physical segment currently in DSR using the
                                  offset in EA (AA55H).
                              ;   Increment EA.
   L      R6,DSR:[EA+]        ;   Load R6 from the physical segment currently in DSR using the
                                  offset in EA (AA56H).
                              ;   Increment EA.
```

After a reset, this register contains zero.

1.3 Memory Spaces

The U8 memory space consists of 256 physical segments of 64K bytes each. It is shared by a 1-megabyte program/code memory space (0:0000H to F:FFFFH) and a 16-megabyte data memory space (0:0000H to FF:FFFFH). Physical segment #0, however, has a different structure from the others, #1 and higher.

1.3.1 Program/Code Memory Space

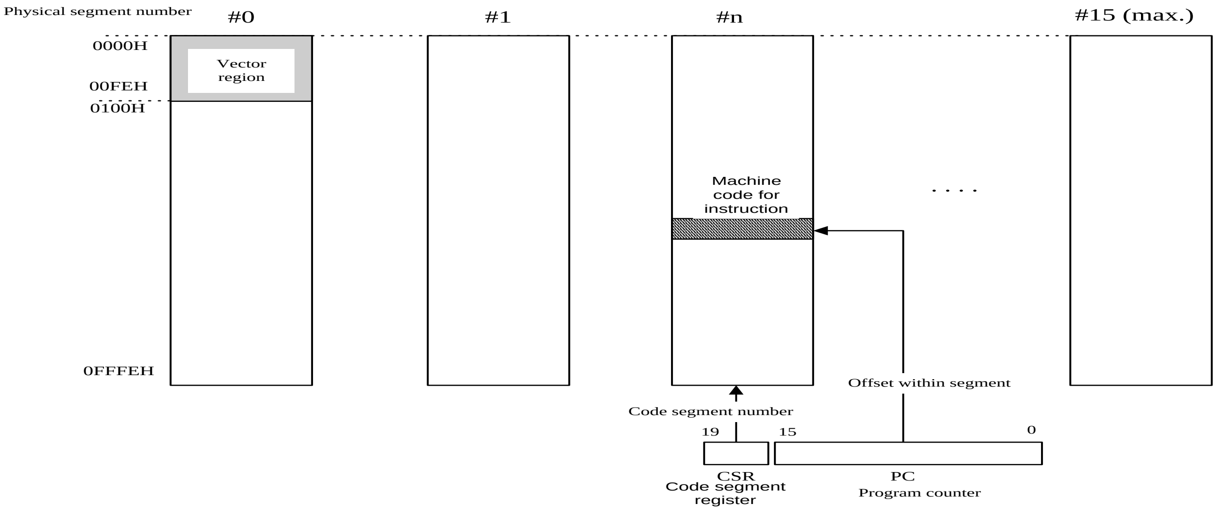
The U8's 1-megabyte program/code memory space features 16 physical segments with 32 kilowords each. Its primary uses are holding the machine code necessary for program execution and read-only data tables. Programs access this space with 20-bit addresses (CSR:PC) combining the contents of the code segment register (CSR) in the top four bits and those of the program counter (PC) in the remaining 16 bits. The contents of the code segment register (CSR) are called the code segment.

Such address calculations as incrementing the program counter (PC) and adding or subtracting a displacement to calculate a relative branch target ignore any over- or underflow, so never modify the CSR contents.

The ROM window, a special region in physical segment #0, is accessible using RAM addressing.

Access with a physical segment uses a 16-bit offset between 0 and 0FFFEH. Address calculations affect only this 16-bit offset and ignore any over- or underflow.

The following illustrates the layout of this memory space.



1.3.2 Vector Table

Addresses 0:0H to 0:0FEH in the program/code memory space are reserved for a vector table containing 16-bit offsets to the routines processing resets and interrupts. Each vector in the table starts at an even address. The hardware automatically resets the code segment register (CSR) to zero, so these routines must always start in physical segment #0.

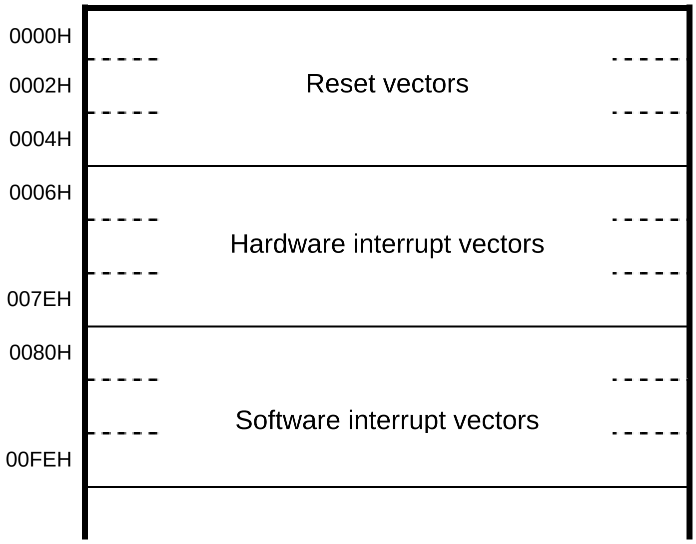


Figure 1.4. Vector Table

1.3.2.1 Reset Vectors

This portion of the vector table holds the entry points for processing resets—that is, the initial value for the stack pointer at address 0 and the reset routine entry points at addresses 2 and 4.

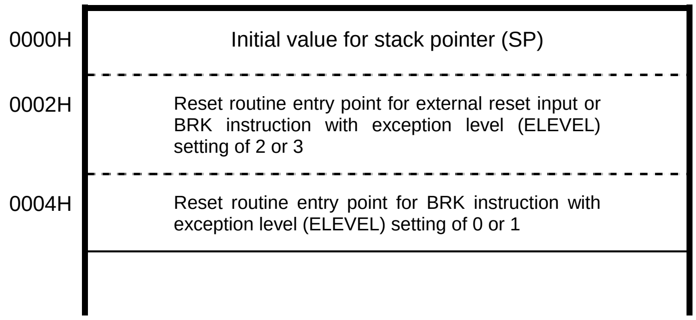


Figure 1.5. Reset Vectors

1.3.2.2 Interrupt Vectors

1.3.2.2.1 Hardware Interrupt Vectors

This portion of the vector table holds the entry points for processing hardware interrupts. There are two nonmaskable interrupt requests, NMICE and NMI, plus room for up to 59 maskable ones.

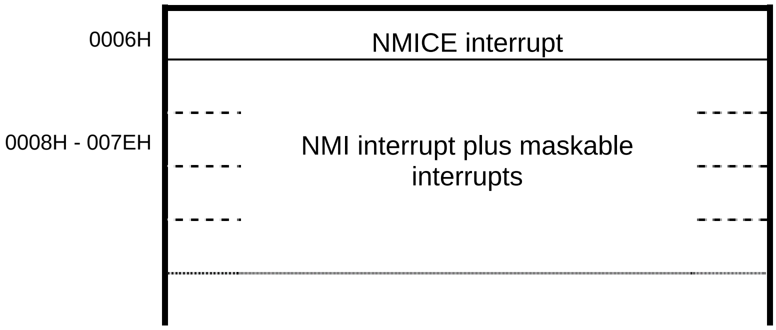


Figure 1.6. Hardware Interrupt Vectors

1.3.2.2.2 Software Interrupt Vectors

This portion of the vector table holds the entry points for interrupt requests from SWI instructions in the program.

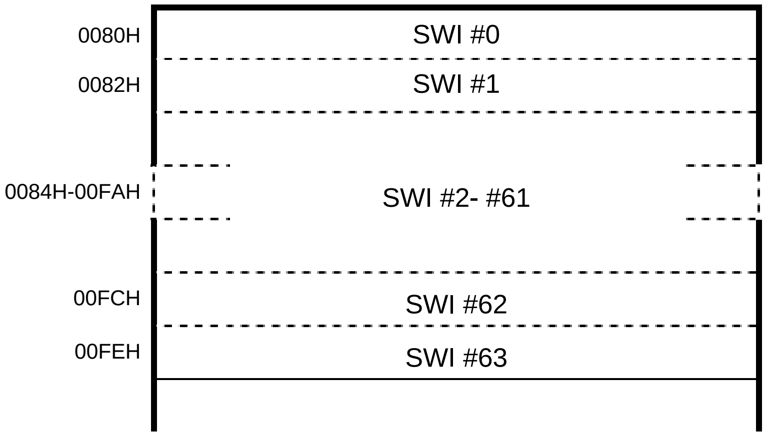


Figure 1.7. Software Interrupt Vectors

1.3.2.3 Writing Vector Table

In assembly language, use DW directives with labels representing the entry points as their operands as shown in the following code fragment.

Note that only the reset vectors must always be present. If the program does not use these interrupts, this region is available for normal program code.

```
;
;reset vector table
;
        cseg    at              0000h

        dw      spinit                  ; Initial value for stack pointer
        dw      start                   ; Initial value for program counter
        dw      brk                     ; Reset routine entry point for BRK instruction

        org     0008h
        dw      nmi_entry               ; Nonmaskable interrupt
        dw      Int1_entry              ; Maskable interrupt #0
        dw      Int2_entry              ; Maskable interrupt #1
                :
                :
                :
;
;software interrupts
;
        cseg    at              0080h
swi_0:  dw      sw0_entry               ; Software interrupt #0
swi_1:  dw      sw1_entry               ; Software interrupt #1
                :
                :
;
;start of main procedure
;
start:                                  ; Program entry point
                :
                :
                :
```

1.3.3 Program/Code Memory Space

From the programming standpoint, there is no logical difference between physical segment #0 and the others. The linker and other tools automatically assign the program code to onboard memory available on the chip and then to external memory.

1.3.4 DSR Prefix Instructions

The U8 architecture divides memory spaces into physical segments of 64K bytes each, so accessing data in a physical segment other than physical segment #0 requires manipulating the data segment register (DSR) with one of the following three DSR prefix instructions.

DSR Prefix Instruction	**Function**
`1110_0011_iiii_iiii`	Load DSR with the 8-bit immediate value iiii_iiii.
`1001_0000_dddd_1111`	Load DSR with the contents of the general register Rd.
`1111_1110_1001_1111`	Use the current DSR value.

DSR prefix instructions have this prefixing effect only when they immediately precede a memory access instruction. Memory access instructions without an immediately preceding DSR prefix instruction access physical segment #0.

The hardware automatically disables all interrupts between a DSR prefix instruction and the immediately following instruction. For further details, see Section 1.5 “Interrupt Blocking” below.

Note: To prevent unintended operation and provide the strongest checking possible of memory access, the U8 assembly language specifications deliberately forbid the use of the DSR prefix instructions in program source code. Instead, use the corresponding DSR prefix inside the memory access instruction itself. For further details, see Section 2.3 “Memory Addressing” below.

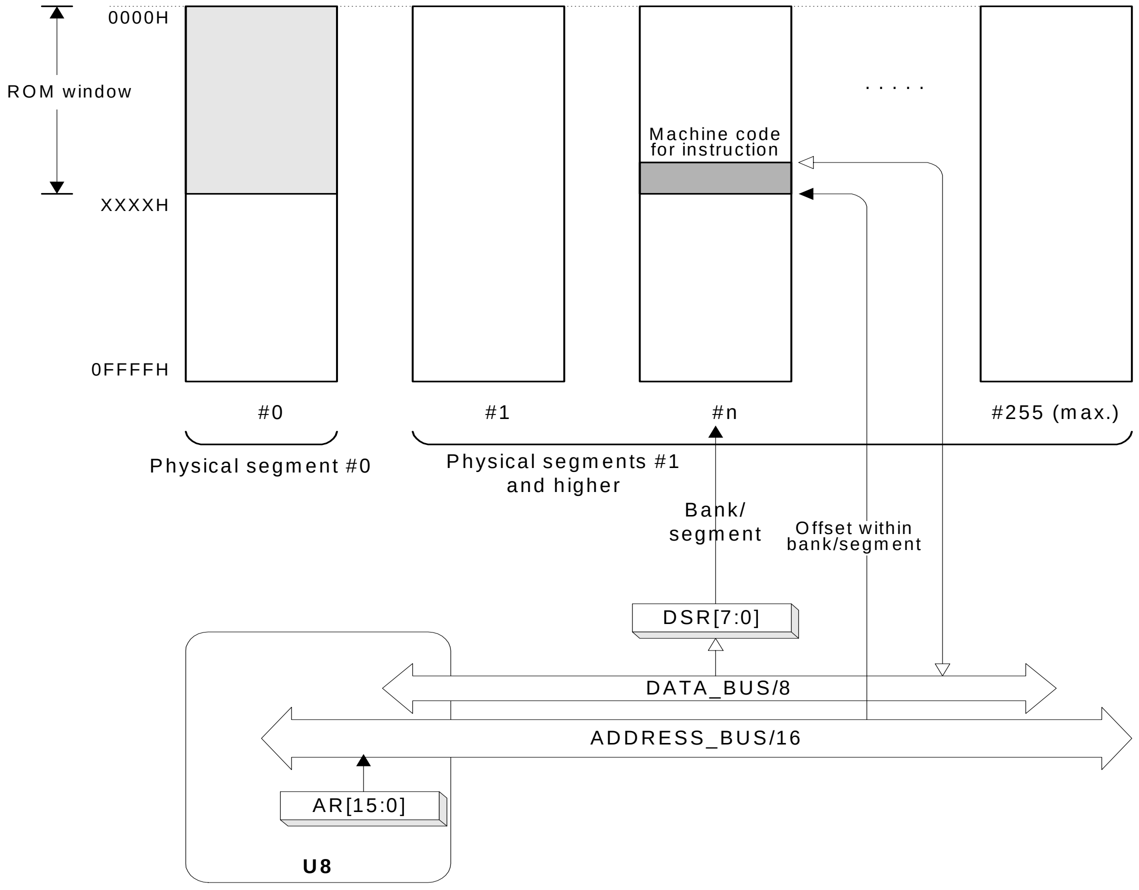
1.3.5 Data Memory Space

The U8's 16-megabyte data memory space features 256 physical segments with 64 kilobytes each. Its primary use is holding data that is written as well as read.

Programs access this space with 24-bit addresses (DSR:AR) combining the contents of the data segment register (DSR) in the top eight bits and those of the address register (AR) in the remaining 16 bits. The contents of the data segment register (DSR) are called the data segment.

Physical segment #0 consists of the ROM window plus one or two data regions. The ROM window is a special region accessing program/code memory addresses using RAM addressing. The corresponding data memory addresses are not physically present. The primary use for this window is accessing table data in ROM.

The following illustrates the layout of this memory space.



1.3.5.1 Data Types

This Section describes the data types supported by U8 instructions.

Unsigned Byte

This data type is used by instructions operating on bytes. Values range from 0 to 255. Arithmetic operations that underflow or overflow this range set the carry flag (C) to “1” and discard all but the lowest eight bits to produce a result modulo 256.

Bit operations manipulate individual bits. The bits are numbered 0 to 7 from the least significant bit (LSB) to the most significant bit (MSB).

Signed Byte

This data type is used by instructions operating on bytes. The top bit is considered the sign bit, producing twos complement values ranging from -128 to +127. Arithmetic operations that underflow or overflow this range set the overflow flag (OV) to “1.”

Unsigned Word

This data type is used by instructions operating on words. Values range from 0 to 65,535. Arithmetic operations that underflow or overflow this range set the carry flag (C) to “1” and discard all but the lowest 16 bits to produce a result modulo 65,536.

Memory storage is little endian, with the lower byte (bits 7 to 0) preceding the upper (bits 15 to 8). Data memory requires word boundary alignment with the lower byte at an even address and the upper at the next, odd address. Program/code memory does not impose this restriction. The address of word data is always the address of its lower byte.

Bitwise operations manipulate individual bits. The bits are numbered 0 to 15 from the least significant bit (LSB) to the most significant bit (MSB).

Signed Word

This data type is used by instructions operating on words. The top bit is considered the sign bit, producing twos complement values ranging from -32768 to + 32767. Arithmetic operations that underflow or overflow this range set the overflow flag (OV) to “1.”

Memory storage is little endian, with the lower byte (bits 7 to 0) preceding the upper (bits 15 to 8). Data memory requires word boundary alignment with the lower byte at an even address and the upper at the next, odd address. Program/code memory does not impose this restriction. The address of word data is always the address of its lower byte.

Bit

This data type is used by instructions operating on bits. The only values are “0” and “1.” This type applies to individual bits in most registers and all bits in memory. Bit addressing uses the name of a byte-sized register or a memory address plus, the dot operator, and the number (0 to 7). The operations available for these bits include transfers, logical operations, and bit test and jump.

1.3.5.2 Address Assignment

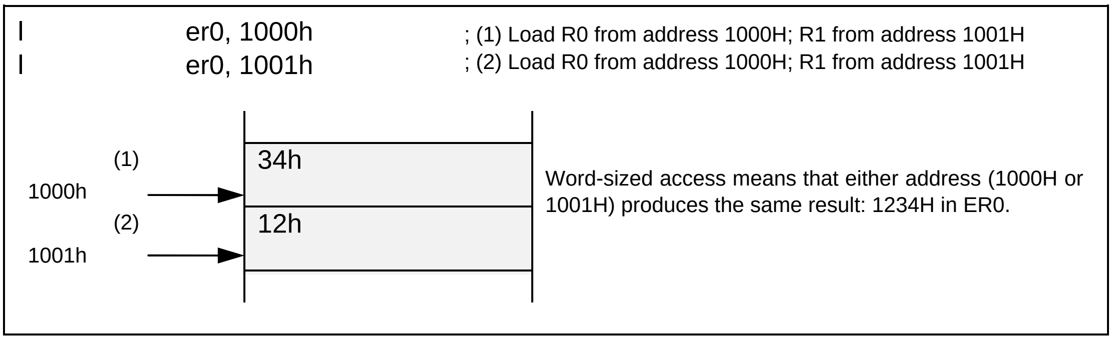
Memory addresses are in bytes. Byte addressing assigns a unique address to every byte in memory. These addresses run from 0 to FFFFH (65,535) in each 64-kilobyte physical segment.

The U8 architecture separates memory into program/code memory and data memory, each with their own set of byte addresses.

1.3.5.3 Word Boundaries

The U8 data memory has word boundaries. The hardware automatically enforces word alignment by ignoring the lowest bit in the address, forcing it to “0.” Instructions accessing word-, double word-, or quad word-sized data using odd-numbered therefore access the preceding even-numbered address without triggering an addressing error. The programmer must, therefore, assign these larger data types to word boundaries. Note that there are no additional boundaries for multiword types. The only requirement is that they be on word boundaries.

Program/code memory also has word boundaries. So too does program/code memory accessed by the ROM window.



```
l         er0, 1000h      ; (1) Load R0 from address 1000H; R1 from address 1001H
l         er0, 1001h      ; (2) Load R0 from address 1000H; R1 from address 1001H
```

Word-sized access means that either address (1000H or 1001H) produces the same result: 1234H in ER0.

Figure 1.8. Word Boundaries in Memory

1.3.5.4 ROM Window

This window, assigned to an unused portion of physical segment #0 in data memory, is for accessing the corresponding program/code memory addresses with RAM addressing. The U8 architecture thus does not need special instructions for accessing data in program/code memory (ROM). Using a RAM access instruction with an address in the ROM window produces the same result.

Accessing an address in the ROM window takes, however, more instruction cycles than the same instruction accessing data memory. For further details, see Section 3.3 “Instruction Execution Times.”

The ROM window supports only read access. Writing to a ROM window address does not produce meaningful results.

1.3.6 Hardware Memory Models

The U8 architecture provides hardware control over the number of physical segments accessible as program/code memory: 64 kilobytes (32 kilowords) or 1 megabyte (512 kilowords). The procedure for specifying this hardware memory model is in the User’s Manual for the target device.

The following Table summarizes the models available.

Model Name	Program/Code Addresses	CSR CSR Backup Registers
SMALL	Program memory: 0H - FFFFH Data memory: 0H - FF:FFFFH	Not used
LARGE	Program memory: 0H - F:FFFFH Data memory: 0H - FF:FFFFH	Used

The choice of hardware memory model affects the following aspects of operation.

- Amount of program/code memory available
- Operation of subroutine calls and corresponding RT instructions
- Operation of interrupts and corresponding RTI instructions
- Operation of PUSH and POP instructions

The following Table outlines these differences in more detail.

	SMALL	LARGE
Amount of program/code memory available	64K bytes (0H - 0FFFFH)	1M bytes (0H - F:FFFFH)
Registers saved during subroutine call	PC	PC CSR
Registers restored by RT instruction	PC	PC CSR
Registers saved during interrupt acceptance	PC PSW	PC PSW CSR
Registers restored by RTI instruction	PC PSW	PC PSW CSR
Registers saved by PUSH LR/ELR instruction	LR	LR LCSR
	ELR	ELR ECSR
Registers restored by POP LR/PC instruction	LR	LR LCSR
	PC	PC CSR

1.3.7 Interrupt Operation

1.3.7.1 Interrupt Acceptance

The U8 hardware accepts an interrupt request (NMICE, NMI, or MI) only if its interrupt level is the same or greater than the current exception level (ELEVEL) setting.

The following Table lists the interrupt level, an integer between 0 and 3 indicating the interrupt priority, for each type of interrupt.

Interrupt Type	Interrupt Level
Emulator interrupt (NMICE)*[1]	3
Nonmaskable interrupt (NMI)	2
Software interrupt (SWI)	1
Maskable interrupt (MI)	1

*1 This interrupt request requires an in-circuit emulator. It is not available to user application programs.

An exception level (ELEVEL) setting of zero indicates that there are no interrupt requests pending.

The higher the interrupt level, the greater the priority.

When the hardware accepts an interrupt request, it saves the interrupt level in the ELEVEL field of the program status word (PSW).

When the U8 hardware receives an interrupt request, it first compares the interrupt level with the current exception level (ELEVEL) setting. If the interrupt level is the same or greater than ELEVEL, the hardware loads the program counter (PC) from the appropriate entry in the vector table.

Address in vector table	Description
0000H	Initial value for stack pointer
0002H	Reset routine entry point for external reset input or BRK instruction with exception level (ELEVEL) setting of 2 or 3
0004H	Reset routine entry point for BRK instruction with exception level (ELEVEL) setting of 0 or 1
0006H	Interrupt handler entry point for emulator interrupt (NMICE)
0008H - 007EH	Interrupt handler entry points for nonmaskable (NMI) and maskable (MI) interrupts
0080H - 00FEH	Interrupt handler entry points for software interrupts (SWI)

The following gives the detailed acceptance procedure for each interrupt type.

1.3.7.2 Nonmaskable Interrupts (NMI)

User application programs have no means of masking nonmaskable interrupts. When the hardware detects one, control immediately transfers to the appropriate NMI interrupt handler. If the CPU is already executing the NMI interrupt handler, control still returns to the beginning.

The hardware masks them, however, in the following situations.

- Between a reset (either hardware reset input or a BRK instruction with an ELEVEL setting of 3) and the end of the first instruction in the reset handler
- Between the start of the interrupt acceptance cycle and the end of the first instruction in the interrupt handler
- Between a DSR prefix instruction and the immediately following instruction

A nonmaskable interrupt request automatically causes the hardware to perform the following actions.

1. Save PC in ELR2.
2. Save CSR in ECSR2.
3. Save PSW in EPSW2.
4. Set ELEVEL field in PSW to 2.
5. Reset CSR to zero.
6. Load program counter (PC) from vector table.
7. Disable interrupt requests for the duration of the first instruction in the interrupt handler.

The processing time required for the above actions is 3 cycles, however, when the interruption occurs immediately after the instruction using [EA+] addressing, the interruption sequence is started after one machine cycle of wait cycles is performed. For further details, see Section 3.3 “Instruction Execution Times.”

The NMI interrupt handler can exit in different ways. For further details, see Section 1.4 “Exception Levels and Backup Registers.”

1.3.7.3 Maskable Interrupts (MI)

Maskable interrupts have many sources among the onboard peripherals and external input pins. The hardware only accepts them, however, if the MIE bit in the program status word (PSW) is “1.”

The hardware masks them, however, in the following situations.

- Between a reset (either hardware reset input or a BRK instruction with an ELEVEL setting of 3) and the end of the first instruction in the reset handler
- Between the start of the interrupt acceptance cycle and the end of the first instruction in the interrupt handler
- Between a DSR prefix instruction and the immediately following instruction
- While the ELEVEL setting is 2 or 3

Acceptance of a maskable interrupt request automatically causes the hardware to perform the following actions.

1. Save PC in ELR1.
2. Save CSR in ECSR1.
3. Save PSW in EPSW1.
4. Set ELEVEL field in PSW to 1.
5. Set MIE bit in PSW to “0” to disable further interrupt requests.
6. Reset CSR to zero.
7. Load program counter (PC) from vector table.
8. Disable all interrupt requests for the duration of the first instruction in the interrupt handler.

The processing time required for the above actions is 3 cycles, however, when the interruption occurs immediately after the instruction using [EA+] addressing, the interruption sequence is started after one machine cycle of wait cycles is performed. For further details, see Section 3.3 “Instruction Execution Times.”

The MI interrupt handler can exit in different ways. For further details, see Section 1.4 “Exception Levels and Backup Registers.”

1.3.7.4 Software Interrupts (SWI)

Software interrupts come from inside the user application program, so are immediately accepted. The operand to the SWI instruction specifies the interrupt number.

A software interrupt request automatically causes the hardware to perform the following actions.

1. Save PC in ELR1.
2. Save CSR in ECSR1.
3. Save PSW in EPSW1.
4. Set ELEVEL field in PSW to 1.
5. Set MIE bit in PSW to “0” to disable further interrupt requests.
6. Reset CSR to zero.
7. Load program counter (PC) from vector table.
8. Disable all interrupt requests for the duration of the first instruction in the interrupt handler.

The processing time required for the above actions is 3 cycles, however, when the SWI instruction is executed immediately after the instruction using [EA+] addressing, the interruption sequence is started after one machine cycle of wait cycles is performed. For further details, see Section 3.3 “Instruction Execution Times.”

The software interrupt handler can exit in different ways. For further details, see Section 1.4 “Exception Levels and Backup Registers.”

1.4 Exception Levels and Backup Registers

The U8 architecture provides three sets of backup registers for saving the contents of the program counter (PC), code segment register (CSR), and program status word (PSW) during subroutine calls and interrupt handlers. The PC and CSR backup registers apply to both situations; the PSW ones, only to interrupt handlers.

The following Table summarizes the use of these backup registers.

PC Backup Registers

Name	Description
LR	This holds the offset portion of the return address for a subroutine call with the BL instruction.
ELR1	This holds the offset portion of the return address for a maskable interrupt or a SWI instruction.
ELR2	This holds the offset portion of the return address for a nonmaskable interrupt.
ELR3	This holds the offset portion of the return address for an emulator interrupt.

CSR Backup Registers

Name	Description
LCSR	This holds the physical segment portion of the return address for a subroutine call with the BL instruction.
ECSR1	This holds the physical segment portion of the return address for a maskable interrupt or a SWI instruction.
ECSR2	This holds the physical segment portion of the return address for a nonmaskable interrupt.
ECSR3	This holds the physical segment portion of the return address for an emulator interrupt.

PSW Backup Registers

Name	Description
EPSW1	This holds the PSW from just before a maskable interrupt or a SWI instruction.
EPSW2	This holds the PSW from just before a nonmaskable interrupt.
EPSW3	This holds the PSW from just before an emulator interrupt.

LR and LCSR are saved by the BL instruction calling a subroutine and restored by the RT instruction ending the subroutine.

The ELR, ECSR, and EPSW registers used depend on the index value from the ELEVEL field in the program status word (PSW). The hardware saves to them during the interrupt acceptance cycle; the RTI instruction at the end of the interrupt handler restores from them.

Note that the sets provide only one register for each level. If subroutines or interrupt handlers are nested, therefore, it is not possible to return with the RT and RTI instructions normally used. The subroutine or interrupt handler must use PUSH instructions to save register contents to the stack before nesting and return with POP instructions instead.

Choosing the appropriate method for saving these registers depends on the CPU state, so requires particular attention during the design phase. The following pages show how to tailor the user application program to the U8 execution state.

The following describes the programming considerations for each possible ELEVEL setting and for whether the user application program nests subroutine and interrupts.

A: ELEVEL is 0

An exception level (ELEVEL) setting of zero indicates that there are no interrupt requests pending. How procedures begin and end depends solely on whether subroutines are nested.

A-1: Subroutines are not nested.

Beginning of procedure: No special steps necessary.

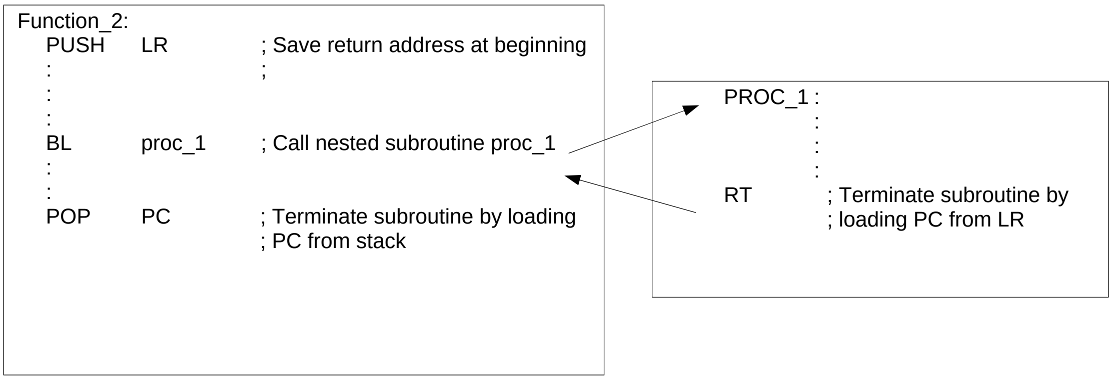
End of procedure: Restore PC from LR with an RT instruction.

A-2: Subroutines are nested.

Beginning of procedure: Use PUSH LR instruction to save return address to the stack.

End of procedure: Use POP LR instruction instead of RT to return from subroutine.

Example for A-2



```
Function_2:
    PUSH    LR          ; Save return address at beginning
    :                   ;
    :
    :
    BL      proc_1      ; Call nested subroutine proc_1
    :
    :
    POP     PC          ; Terminate subroutine by loading
                        ; PC from stack
```

```
PROC_1 :
        :
        :
        :
RT            ; Terminate subroutine by
              ; loading PC from LR
```

B: ELEVEL is 1

An exception level (ELEVEL) setting of 1 indicates a maskable interrupt. In addition to the considerations under A above, how procedures begin and end also depends on whether interrupts are nested.

B-1: Subroutines are not nested.

B-1-1: Interrupts are not nested.

Beginning of procedure: No special steps necessary.

End of procedure: Restore PC from ELR1 and PSW from EPSW1 with an RTI instruction.

B-1-2: Interrupts are nested.

Beginning of procedure: Use PUSH ELR, EPSW instruction to save return address and current PSW to the stack.

End of procedure: Restore PC and PSW from stack with POP LR, PSW instruction instead of RTI.

Example for B-1-1 and B-1-2

```
Intrrpput_B-1-1:     ; B-1-1 example

     :
     :
     :
     RTI        ; Terminate interrupt handler
                ; by restoring PC from ELR1
                ; and PSW from EPSW1 with
                ; RTI instruction
```

```
Intrrpput_B-1-2:                      ; B-1-2 example
     PUSH     ELR,EPSW                ; Save ELR and EPSW contents to
                                      ; stack at beginning of interrupt
                                      ; handler
     EI                               ; Enable interrupts
     :
     :
     :
     :
     :
     POP      PC ,PSW                 ; Terminate interrupt handler by
                                      ; restoring PC and PSW from stack
```

B-2: Subroutines are nested.

B-2-1: Interrupts are not nested.

Beginning of procedure: Use PUSH LR instruction to save return address to the stack.

End of procedure: Use POP LR instruction to restore return address and then RTI to return from subroutine.

B-2-2: Interrupts are nested.

Beginning of procedure: Use PUSH LR, ELR, EPSW instruction to save return addresses and current PSW to the stack.

End of procedure: Restore PC and PSW from stack with POP LR, PSW, LR instruction instead of RTI.

Example for B-2-2

```
Procedure_B-2-2:                ; B-2-2 example
PUSH       ELR,EPSW,LR          ; Save ELR, EPSW, and LR
                                ; contents to stack at
                                ; beginning of subroutine
     EI                         ; Enable interrupts
     :                          ;
     :                          ;


BL  proc_1                       ; Call nested subroutine proc_1

     :
     POP    LR,PSW ,PC          ; Terminate subroutine by
                                ; restoring LR, PSW, and PC
                                ; from stack
```

```
    PROC_1:
                ;
                ;
                ;
                ;
    RT        ; Terminate subroutine by
              ; loading PC from LR
```

C: ELEVEL is 2

An exception level (ELEVEL) setting of 2 indicates a nonmaskable interrupt. The backup registers used are ELR2, EPSW2, and ECSR2.

C-1: Subroutines are not nested.

Beginning of procedure: No special steps necessary.

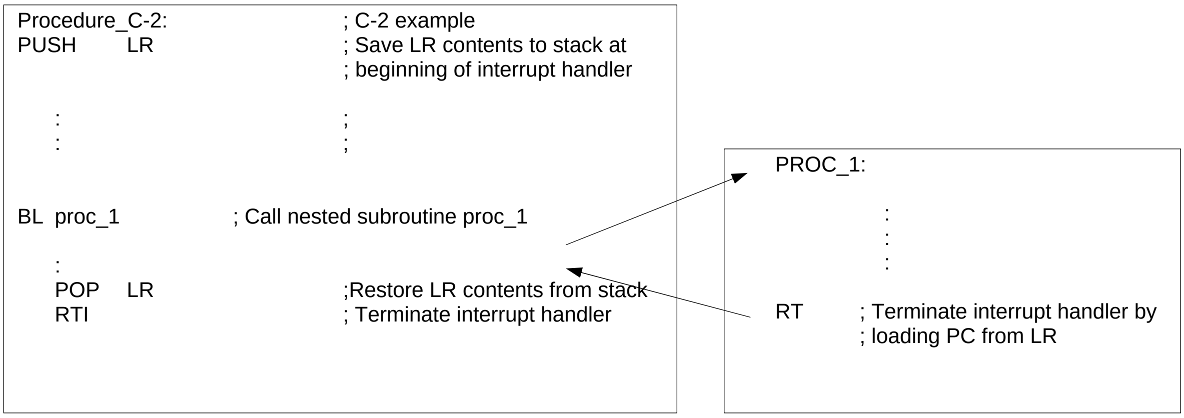
End of procedure: Restore PC from ELR2 and PSW from EPSW2 with an RTI instruction.

C-2: Subroutines are nested.

Beginning of procedure: Use PUSH LR instruction to save return address to the stack.

End of procedure: Use POP LR instruction to restore return address and then RTI to return from interrupt handler.

Example for C-2



```
Procedure_C-2:                        ; C-2 example
PUSH       LR                         ; Save LR contents to stack at
                                      ; beginning of interrupt handler

     :                                ;
     :                                ;


BL  proc_1                ; Call nested subroutine proc_1

     :
     POP    LR                        ;Restore LR contents from stack
     RTI                              ; Terminate interrupt handler
```

```
PROC_1:

                 :
                 :
                 :

RT          ; Terminate interrupt handler by
            ; loading PC from LR
```

Choosing the appropriate method for saving these registers depends on the CPU state, so requires particular attention during the design phase.

1.5 Interrupt Blocking

As has already been mentioned above, the hardware masks all pending interrupt requests for the duration of the following situations.

(1) Between the start of the interrupt acceptance cycle and the end of the first instruction in the interrupt handler

The hardware delays acceptance of the new interrupt request until it has completed execution of the first instruction in the interrupt handler for the old.

(2) Between a DSR prefix instruction and the immediately following instruction

The hardware delays acceptance of the new interrupt request until it has completed execution of both instructions in the pair.

For further details on DSR prefix instructions, see Section 1.3.4 “DSR Prefix Instructions.” Although a sequence of DSR prefix instructions properly re-enables interrupt requests, such sequences must not appear in program code as they can lead to unintended behavior.

1.6 Stack Modifications

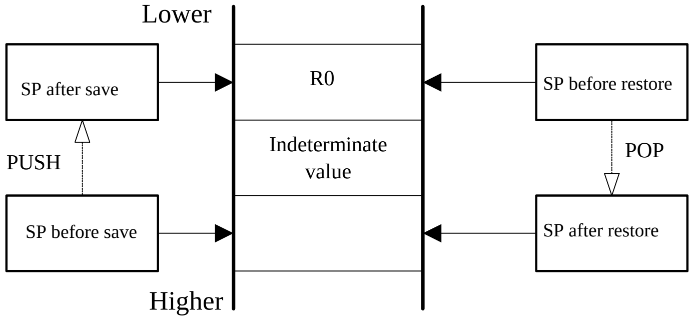
This Section summarizes the effects that PUSH and POP instructions have on the stack. For further details, see Chapter 3 “Instruction Descriptions.”

The stack pointer (SP) always moves by an even number of bytes. If the PUSH instruction operand represents an odd number of bytes, the hardware first introduces a dummy cycle that decrements SP without saving any other registers. This dummy cycle writes an indeterminate byte of data to the stack. Similarly, if the POP instruction operand represents an odd number of bytes, the hardware restores the register and then introduces a dummy cycle that increments SP without restoring any registers.

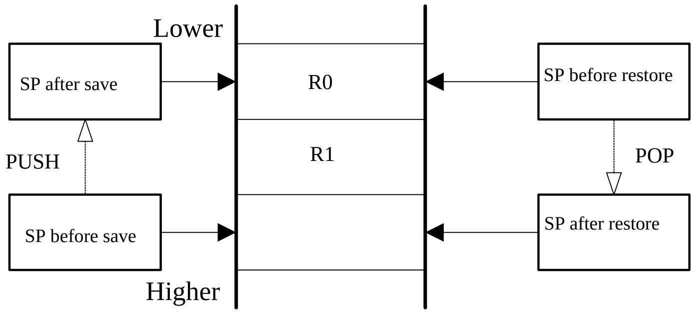
Note that the operation of instructions specifying LR or ELR as the operand depends on the hardware memory model.

The following Figures illustrate the operation of these two instructions.

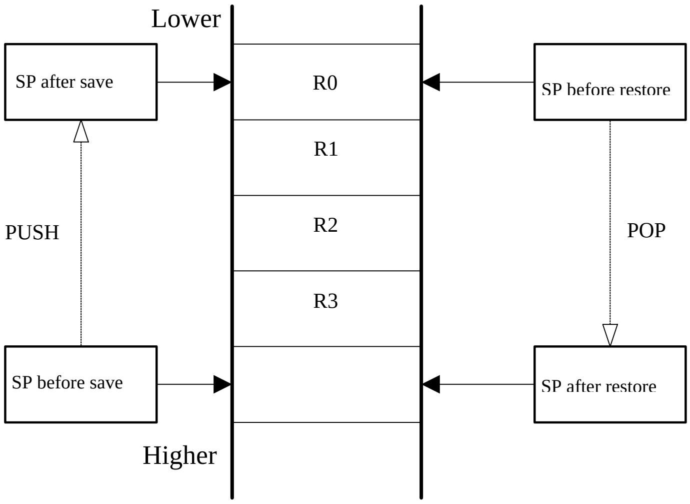
PUSH R0 / POP R0



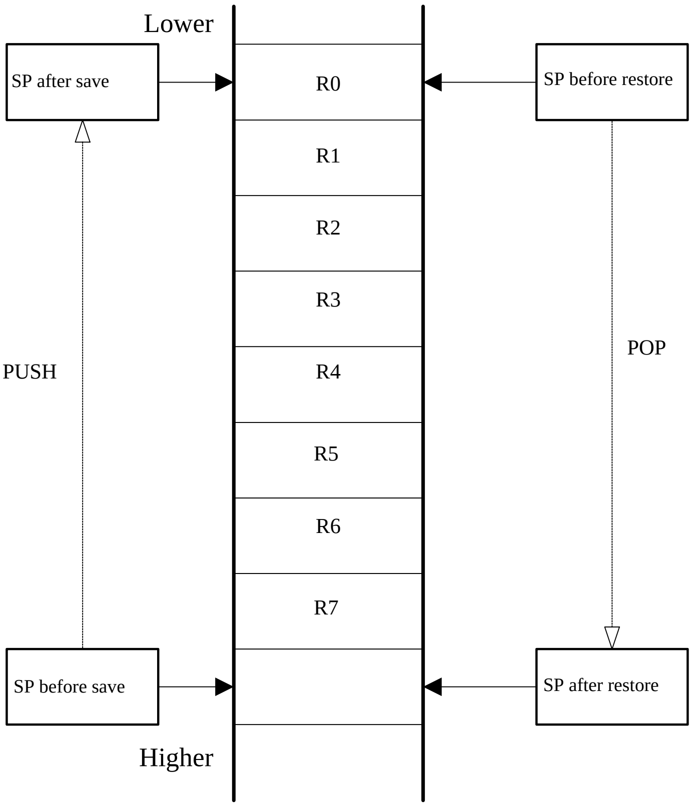
PUSH ER0 / POP ER0



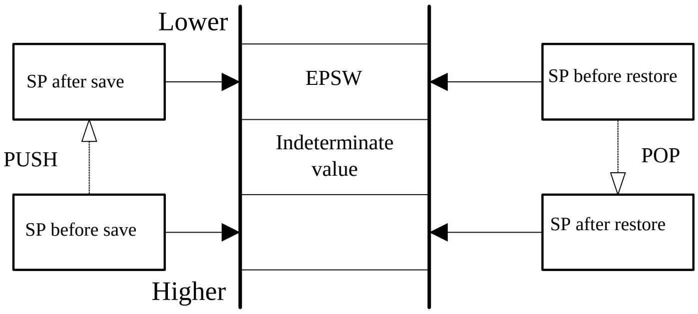
PUSH XR0 / POP XR0



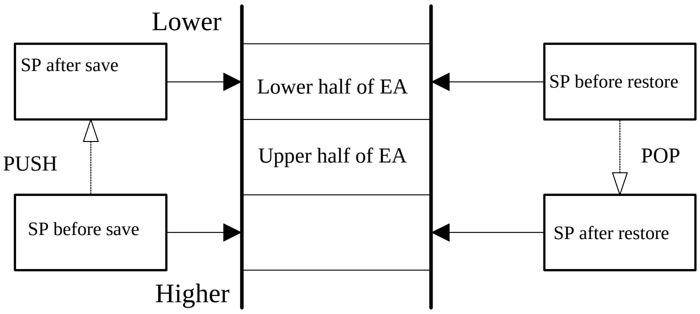
PUSH QR0 / POP QR0



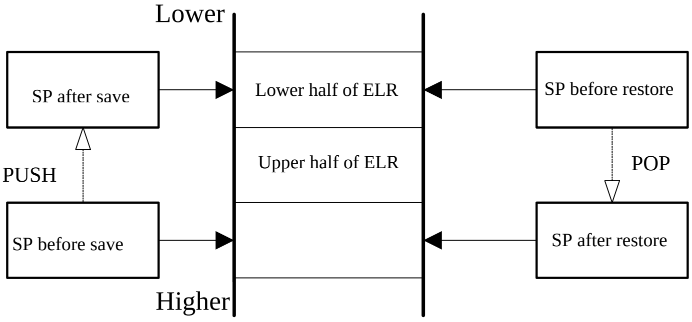
PUSH EPSW / POP PSW



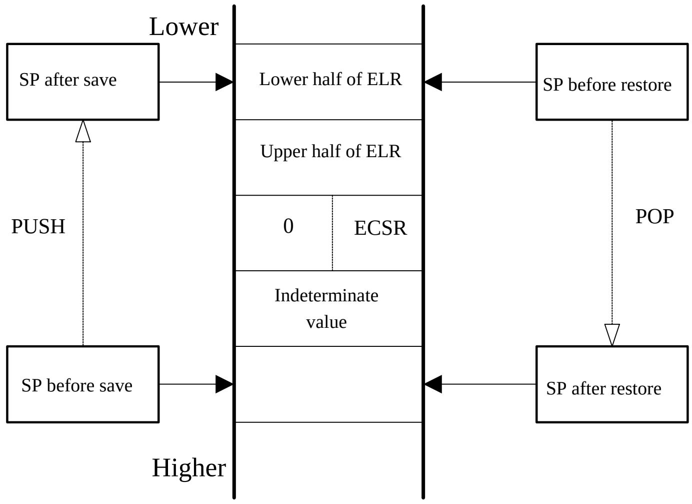
PUSH EA / POP EA



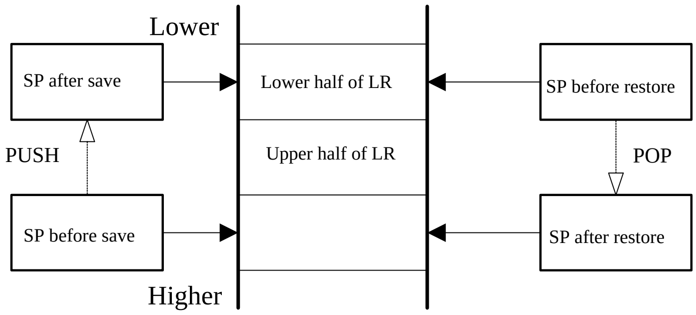
PUSH ELR / POP PC (SMALL model)



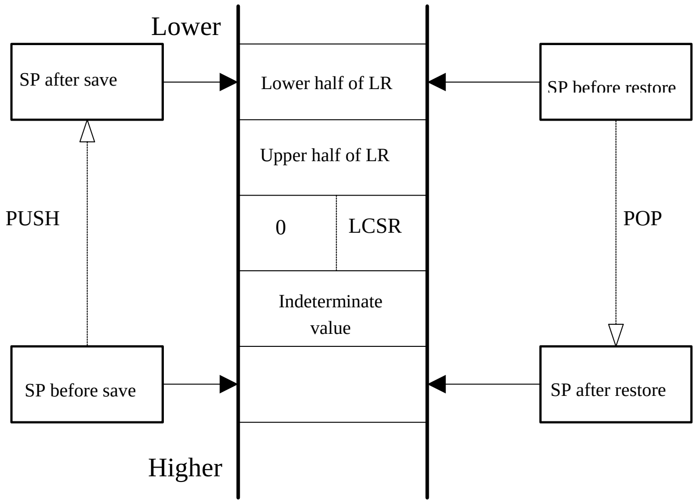
PUSH ELR / POP PC (LARGE model)



PUSH LR / POP LR (SMALL model)



PUSH LR / POP LR (LARGE model)



2. Addressing Types

2.1 Addressing Types

The nX-U8/100 architecture has four addressing types:

- register addressing for accessing internal and coprocessor registers
- memory addressing for accessing data memory and program/code memory inside the ROM window
- immediate addressing for specifying numeric values
- program/code memory addressing for accessing program/code memory

2.2 Register Addressing

The following register addressing types access the contents of the specified register.

Addressing Notation	**Function**
R*n*	This addressing type accesses the contents of the specified byte-sized general register (R*n*).
ER*n*	This addressing type accesses the contents of the specified word-sized general register (ER*n*). When the instruction table lists ER*n* in an operand, BP may be substituted for ER12 and FP for ER14.
XR*n*	This addressing type accesses the contents of the specified double word-sized general register (XR*n*).
QR*n*	This addressing type accesses the contents of the specified quad word-sized general register (QR*n*).
CR*n*	This addressing type accesses the contents of the specified byte-sized coprocessor register (CR*n*).
CER*n*	This addressing type accesses the contents of the specified word-sized coprocessor register (CER*n*).
CXR*n*	This addressing type accesses the contents of the specified double word-sized coprocessor register (CXR*n*).
CQR*n*	This addressing type accesses the contents of the specified quad word-sized coprocessor register (CQR*n*).
PC	This addressing type accesses the contents of the program counter.
LR	This addressing type accesses the contents of the link register.
EA	This addressing type accesses the contents of the EA register.
SP	This addressing type accesses the contents of the stack pointer.
PSW	This addressing type accesses the contents of the program status word.
ELR	This addressing type accesses the contents of an exception link register.
ECSR	This addressing type accesses the contents of a CSR backup register.
EPSW	This addressing type accesses the contents of a PSW backup register.
R*n.bit_offset*	This addressing type accesses the contents of bit specified by *bit_offset* in general register R*n*.

2.3 Memory Addressing

This addressing type accesses the contents of an address in the data memory space.

Accessing data in a physical segment other than physical segment #0 requires manipulating the data segment register (DSR) with a DSR prefix instruction.

To prevent unintended operation and provide the strongest checking possible of memory access, the U8 assembly language specifications deliberately forbid the use of the DSR prefix instructions in program source code. Instead, use the corresponding DSR prefix inside the memory access instruction itself.

DSR Prefix Instruction	**Function**	**Corresponding Prefix**
`1110_0011_iiii_iiii`	Load DSR with the 8-bit immediate value iiii_iiii.	*pseg_addr* : or FAR
`1001_0000_dddd_1111`	Load DSR with the contents of the general register Rd.	R*d* :
`1111_1110_1001_1111`	Use the current DSR value.	DSR :

The following Table shows examples of these prefixes on the left and their results on the right.

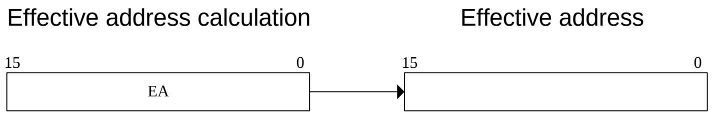
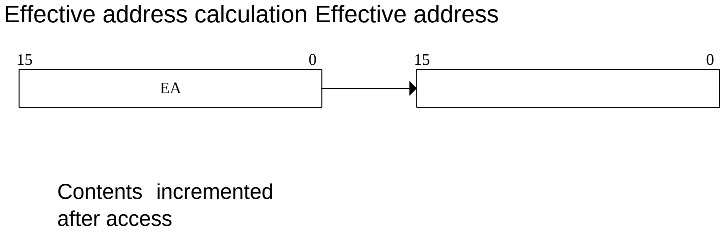
Assembly Language Source Code	**Actual Instruction Sequence**
`L R0, 1:2345H`	`DSR ← 1` `R0 ← [2345H]`
`L R0, R1:[ER2]`	`DSR ← R1` `R0 ← [ER2]`
`ST R1, DSR:[EA]`	`[(DSR<<16)|EA] ← R1`

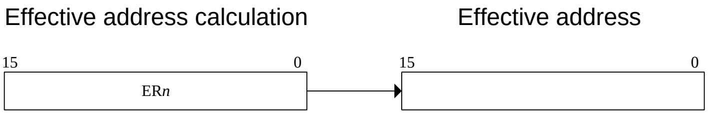
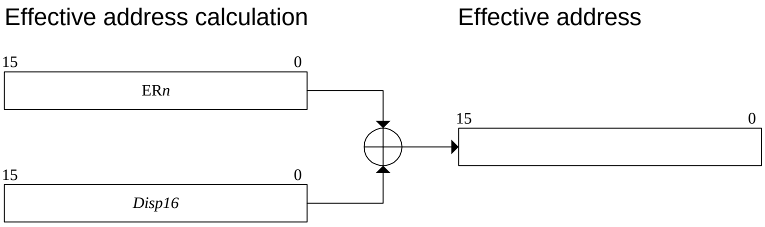
The underlined portions in the above Table indicate the DSR prefixes producing the desired DSR manipulations.

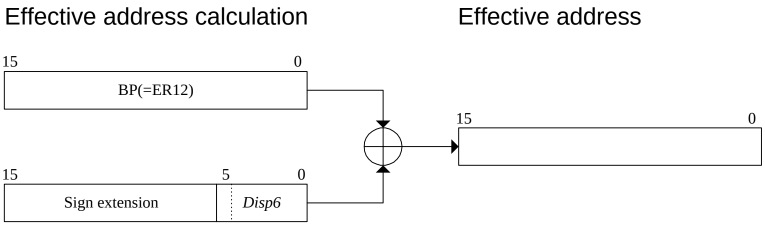
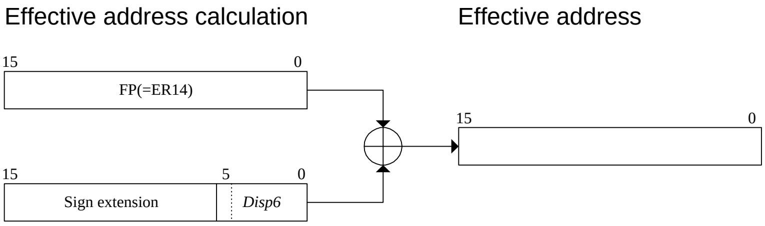
If there is no DSR prefix, the instruction accesses physical segment #0 in the data memory space.

2.3.1 Register Indirect Addressing

The following register indirect addressing types access the contents of the data memory address in the specified register.

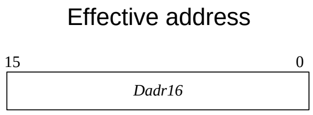
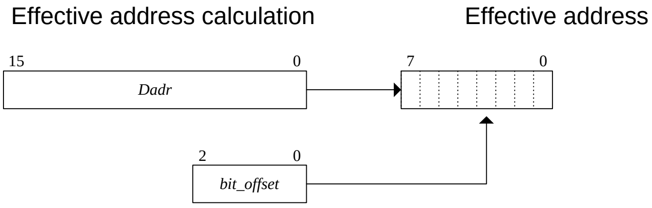
<table>
<tr><th>Addressing Notation</th><th>Function</th></tr>
<tr><td>[EA]</td><td>This addressing type accesses the contents of the data memory space at the offset in the EA register.
 </td></tr>
<tr><td><i>pseg_addr</i>:[EA]</td><td>This variant uses the physical segment number specified by #<i>pseg_addr</i>.</td></tr>
<tr><td>DSR:[EA]</td><td>This variant uses the physical segment number in DSR.</td></tr>
<tr><td>R<i>d</i>:[EA]</td><td>This variant uses the physical segment number in general register R<i>d</i>.</td></tr>
<tr><td>[EA+]</td><td>This addressing type accesses the contents of the data memory space at the offset in the EA register.
After the access, the contents of the EA register are incremented by the operand size in bytes and, for all sizes except byte, rounded down to an even address.
<table>
<tr><th>Operand size</th><th>EA Contents</th><th>Increment</th></tr>
<tr><td>Byte</td><td>Even</td><td>1</td></tr>
<tr><td></td><td>Odd</td><td>1</td></tr>
<tr><td>Word</td><td>Even</td><td>2</td></tr>
<tr><td></td><td>Odd</td><td>1</td></tr>
<tr><td>Double word</td><td>Even</td><td>4</td></tr>
<tr><td></td><td>Odd</td><td>3</td></tr>
<tr><td>Quad word</td><td>Even</td><td>8</td></tr>
<tr><td></td><td>Odd</td><td>7</td></tr>
</table>
 </td></tr>
</table>

Addressing Notation	**Function**
pseg_addr:[EA+]	This variant uses the physical segment number specified by #*pseg_addr*.
DSR:[EA+]	This variant uses the physical segment number in DSR.
R*d*:[EA+]	This variant uses the physical segment number in general register R*d*.
[ER*n*]	This addressing type accesses the contents of the data memory space at the offset in the word-sized general register ER*n*. [BP] instead of [ER12] and [FP] instead of [ER14] are also acceptable. 
pseg_addr:[ER*n*]	This variant uses the physical segment number specified by #*pseg_addr*.
DSR:[ER*n*]	This variant uses the physical segment number in DSR.
R*d*:[ER*n*]	This variant uses the physical segment number in general register R*d*.
Disp16[ER*n*]	This addressing type accesses the contents of the data memory space at the byte address formed by adding the displacement *Disp16* to the contents of the word-sized general register ER*n*. 
pseg_addr:*Disp16*[ER*n*]	This variant uses the physical segment number specified by #*pseg_addr*.
DSR:*Disp16*[ER*n*]	This variant uses the physical segment number in DSR.
R*d*:*Disp16*[ER*n*]	This variant uses the physical segment number in general register R*d*.

Addressing Notation	**Function**
Disp6[BP]	This addressing type accesses the contents of the data memory space at the byte address formed by adding the sign-extended displacement *Disp6* to the contents of the base pointer (BP). If there is no DSR prefix, this addressing type accesses physical segment #0 in the data memory space. 
pseg_addr:*Disp6*[BP]	This variant uses the physical segment number specified by #*pseg_addr*.
DSR:*Disp6*[BP]	This variant uses the physical segment number in DSR.
R*d*:*Disp6*[BP]	This variant uses the physical segment number in general register R*d*.
Disp6[FP]	This addressing type accesses the contents of the data memory space at the byte address formed by adding the sign-extended displacement *Disp6* to the contents of the frame pointer (FP). If there is no DSR prefix, this addressing type accesses physical segment #0 in the data memory space. 
pseg_addr:*Disp6*[FP]	This variant uses the physical segment number specified by #*pseg_addr*.
DSR:*Disp6*[FP]	This variant uses the physical segment number in DSR.
R*d*:*Disp6*[FP]	This variant uses the physical segment number in general register R*d*.

2.3.2 Direct Addressing

The following direct addressing types access the contents of the specified data memory address.

Addressing Notation	**Function**
Dadr	This addressing type accesses the contents of the data memory space at the byte address in the instruction. 
pseg_addr:*Dadr*	This variant uses the physical segment number *pseg_addr*.
DSR:*Dadr*	This variant uses the physical segment number in DSR.
R*d*:*Dadr*	This variant uses the physical segment number in general register R*d*.
Dbitadr	This addressing type accesses the contents of the data memory space at the bit address (*Dadr*.*bit_offset*) in the instruction. 
pseg_addr:*Dbitadr*	This variant uses the physical segment number *pseg_addr*.
DSR:*Dbitadr*	This variant uses the physical segment number in DSR.
R*d*:*Dbitadr*	This variant uses the physical segment number in general register R*d*.

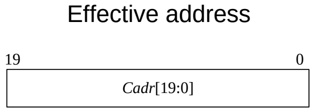
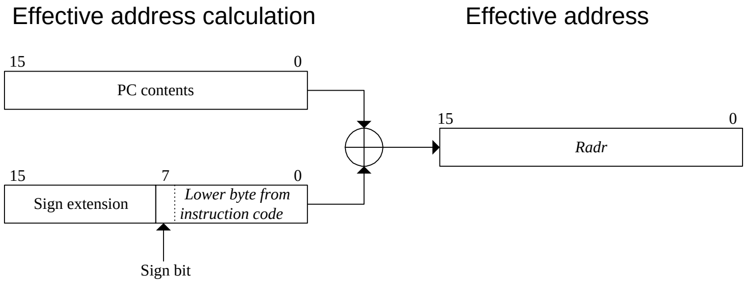
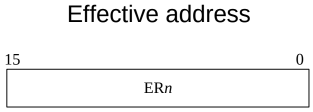
2.4 Immediate Addressing

The following immediate value addressing types use an immediate value contained in the instruction.

Addressing Notation	**Function**
#imm8	The specified value is treated as an 8-bit immediate value.
#signed8	The specified value is treated as a signed 8-bit immediate value. The instruction ADD SP, *#imm8* treats *imm8* as *signed8*. The valid range for *signed8* is between -128 and +127.
#unsigned8	The specified value is treated as an unsigned 8-bit immediate value. The instruction MOV PSW, *#imm8* treats *imm8* as *unsigned8*. The valid range for *unsigned8* is between 0 and 0FFH.
#width	The specified value is treated as a shift size. The valid range for *width* is between 0 and 7.
#snum	The specified value is treated as a SWI instruction vector number. The valid range for *snum7* is between 0 and 63.
#imm7	The specified value is treated as a signed 7-bit immediate value. The valid range for *imm7* is between -64 and +63.

2.5 Program/Code Memory Addressing

The following addressing types access the contents of program/code memory addresses.

Addressing Notation	**Function**
Cadr	This addressing type specifies the 20-bit branch target address for the B and BL instructions. Note that it includes a physical segment number, so the instruction can produce a branch to a different physical segment. 
Radr	This addressing type specifies a relative branch target address for the conditional branch instructions and optimized branch directives. The target must be in within the same physical segment. 
ER*n*	This addressing type specifies the contents of a word-sized general register ER*n* as the branch target offset for the B and BL instructions. The target must be in within the same physical segment. 

3. Instruction Descriptions

This Chapter describes the detailed operation of each instruction.

3.1 Overview

nX-U8/100 core instructions have between zero and two operands. When there are two, the first is the destination; the second, the source.

These operands use the addressing types described in Chapter 2.

For ease of explication, this document uses the following symbols to describe instruction operation.

Symbol	Meaning
←	Assignment
+ or ⊕	Addition
–	Subtraction
*	Multiplication
/	Division
>>	Shift right
<<	Shift left
=	Equality
!=	Inequality
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise inversion

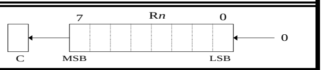
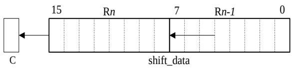
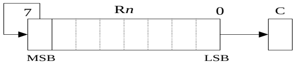
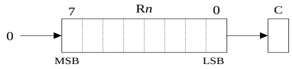
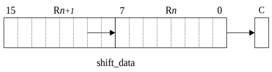
3.2 Instructions by Functional Group

Please refer to Section 3.4 “Instruction Descriptions” about detailed operation of each instruction.

Arithmetic Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
ADD	R*n*	R*m*	*	*	*	*		*	Addition (8-bit)	R*n*← R*n*+obj
		#imm8	*	*	*	*		*		
MOV	R*n*	R*m*		*	*				Data transfer (8-bit)	R*n* ← obj
		#imm8		*	*					
ADDC	R*n*	R*m*	*	*	*	*		*	Addition with carry	R*n*←R*n*+obj+c
		#imm8	*	*	*	*		*		
CMP	R*n*	R*m*	*	*	*	*		*	Comparison (8-bit)	R*n*–obj
		#imm8	*	*	*	*		*		
CMPC	R*n*	R*m*	*	*	*	*		*	Comparison with carry	R*n*–obj–c
		#imm8	*	*	*	*		*		
AND	R*n*	R*m*		*	*				Bitwise AND	R*n*←R*n*&obj
		#imm8		*	*					
OR	R*n*	R*m*		*	*				Bitwise OR	R*n*←R*n* | obj
		#imm8		*	*					
XOR	R*n*	R*m*		*	*				Bitwise exclusive OR	R*n*←R*n*^obj
		#imm8		*	*					
SUB	R*n*	R*m*	*	*	*	*		*	Subtraction	R*n*←R*n*–R*m*
SUBC	R*n*	R*m*	*	*	*	*		*	Subtraction with carry	R*n*←R*n*–R*m* –c
MOV	ER*n*	ER*m*		*	*				Data transfer (16-bit)	ER*n* ← obj
		#imm7		*	*					
ADD	ER*n*	ER*m*	*	*	*	*		*	Addition (16-bit)	ER*n* ← ER*n*+obj
		#imm7	*	*	*	*		*		
CMP	ER*n*	ER*m*	*	*	*	*		*	Comparison (16-bit)	ER*n*–ER*m*

Shift Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
SLL	R*n*	R*m* *#width*	* *						Byte-sized shift left logical	
SLLC	R*n*	R*m* *#width*	* *						Shift left logical continued	
SRA	R*n*	R*m* *#width*	* *						Shift right arithmetic	
SRL	R*n*	R*m* *#width*	* *						Shift right logical	
SRLC	R*n*	R*m* *#width*	* *						Shift right logical continued	

Load/Store Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
L	R*n*	[EA]		*	*				Byte-sized data transfer	R*n*← [EA]
		pseg_addr:[EA]		*	*					
		DSR:[EA]		*	*					
		R*d*:[EA]		*	*					
		[EA+]		*	*				Byte-sized data transfer	R*n*← [EA] EA←EA+1
		pseg_addr:[EA+]		*	*					
		DSR:[EA+]		*	*					
		R*d*:[EA+]		*	*					
		[ER*m*]		*	*				Byte-sized data transfer	R*n*← [ER*m*]
		pseg_addr:[ER*m*]		*	*					
		DSR:[ER*m*]		*	*					
		R*d*:[ER*m*]		*	*					
		Disp16[ER*m*]		*	*				Byte-sized data transfer	R*n*← *Disp16*[ER*m*]
		pseg_addr: *Disp16*[ER*m*]		*	*					
		DSR:*Disp16*[ER*m*]		*	*					
		R*d*:*Disp16*[ER*m*]		*	*					
		Disp6[BP]		*	*				Byte-sized data transfer	R*n*← *Disp6*[BP]
		pseg_addr:*Disp6*[BP]		*	*					
		DSR: *Disp6*[BP]		*	*					
		R*d*: *Disp6*[BP]		*	*					
		Disp6[FP]		*	*				Byte-sized data transfer	R*n*← *Disp6*[FP]
		pseg_addr:*Disp6*[FP]		*	*					
		DSR: *Disp6*[FP]		*	*					
		R*d*: *Disp6*[FP]		*	*					
		Dadr		*	*				Byte-sized data transfer	R*n*← *Dadr*
		pseg_addr: *Dadr*		*	*					
		DSR:*Dadr*		*	*					
		R*d*:*Dadr*		*	*					
	ERn	[EA]		*	*				Word-sized data transfer	ER*n*← [EA]
		pseg_addr:[EA]		*	*					
		DSR:[EA]		*	*					
		R*d*:[EA]		*	*					
		[EA+]		*	*				Word-sized data transfer	ER*n*← [EA] EA←EA+1
		pseg_addr:[EA+]		*	*					
		DSR:[EA+]		*	*					
		R*d*:[EA+]		*	*					
		[ER*m*]		*	*				Word-sized data transfer	ER*n*← [ER*m*]
		pseg_addr:[ER*m*]		*	*					
		DSR:[ER*m*]		*	*					
		R*d*:[ER*m*]		*	*					
		Disp16[ER*m*]		*	*				Word-sized data transfer	ER*n*← *Disp16*[ER*m*]
		pseg_addr: *Disp16*[ER*m*]		*	*					
		DSR:*Disp16*[ER*m*]		*	*					
		R*d*:*Disp16*[ER*m*]		*	*					
		Disp6[BP]		*	*				Word-sized data transfer	ER*n*← *Disp6*[BP]
		pseg_addr:*Disp6*[BP]		*	*					
		DSR: *Disp6*[BP]		*	*					
		R*d*: *Disp6*[BP]		*	*					
		Disp6[FP]		*	*				Word-sized data transfer	ER*n*← *Disp6*[FP]
		pseg_addr:*Disp6*[FP]		*	*					
		DSR: *Disp6*[FP]		*	*					
		R*d*: *Disp6*[FP]		*	*					
		Dadr		*	*				Word-sized data transfer	ER*n*← *Dadr*
		pseg_addr: *Dadr*		*	*					
		DSR:*Dadr*		*	*					
		R*d*:*Dadr*		*	*					

(continued on next page)

Load/Store Instructions (cont.)

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
L	XR*n*	[EA]		*	*				Double word-sized data transfer	XR*n*← [EA]
		pseg_addr:[EA]		*	*					
		DSR:[EA]		*	*					
		R*d*:[EA]		*	*					
		[EA+]		*	*				Double word-sized data transfer	XR*n*← [EA] EA←EA+1
		pseg_addr:[EA+]		*	*					
		DSR:[EA+]		*	*					
		R*d*:[EA+]		*	*					
	QR*n*	[EA]		*	*				Quad word-sized data transfer	QR*n*← [EA]
		pseg_addr:[EA]		*	*					
		DSR:[EA]		*	*					
		R*d*:[EA]		*	*					
		[EA+]		*	*				Quad word-sized data transfer	QR*n*← [EA] EA←EA+1
		pseg_addr:[EA+]		*	*					
		DSR:[EA+]		*	*					
		R*d*:[EA+]		*	*					

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
ST	R*n*	[EA]							Byte-sized data transfer	[EA] ← R*n*
		pseg_addr:[EA]								
		DSR:[EA]								
		R*d*:[EA]								
		[EA+]							Byte-sized data transfer	[EA] ←R*n* EA←EA+1
		pseg_addr:[EA+]								
		DSR:[EA+]								
		R*d*:[EA+]								
		[ER*m*]							Byte-sized data transfer	[ER*m*] ← R*n*
		pseg_addr:[ER*m*]								
		DSR:[ER*m*]								
		R*d*:[ER*m*]								
		Disp16[ER*m*]							Byte-sized data transfer	*Disp16*[ER*m*] ← R*n*
		pseg_addr: *Disp16*[ER*m*]								
		DSR:*Disp16*[ER*m*]								
		R*d*:*Disp16*[ER*m*]								
		Disp6[BP]							Byte-sized data transfer	*Disp6*[BP] ← R*n*
		pseg_addr:*Disp6*[BP]								
		DSR: *Disp6*[BP]								
		R*d*: *Disp6*[BP]								
		Disp6[FP]							Byte-sized data transfer	*Disp6*[FP] ← R*n*
		pseg_addr:*Disp6*[FP]								
		DSR: *Disp6*[FP]								
		Rd: *Disp6*[FP]								
		Dadr							Byte-sized data transfer	[*Dadr*] ← R*n*
		pseg_addr: *Dadr*								
		DSR:*Dadr*								
		R*d*:*Dadr*								
	ERn	[EA]							Word-sized data transfer	[EA] ← ER*n*
		pseg_addr:[EA]								
		DSR:[EA]								
		R*d*:[EA]								
		[EA+]							Word-sized data transfer	[EA] ←ER*n* EA←EA+1
		pseg_addr:[EA+]								
		DSR:[EA+]								
		R*d*:[EA+]								

Load/Store Instructions (cont.)

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
ST	ERn	[ER*m*] *pseg_addr*:[ER*m*] DSR:[ER*m*] R*d*:[ER*m*]							Word-sized data transfer	[ER*m*] ← ER*n*
		Disp16[ER*m*] *pseg_addr*: *Disp16*[ER*m*] DSR:*Disp16*[ER*m*] R*d*:*Disp16*[ER*m*]							Word-sized data transfer	*Disp16*[ER*m*] ←ER*n*
		Disp6[BP] *pseg_addr*:*Disp6*[BP] DSR: *Disp6*[BP] R*d*: *Disp6*[BP]							Word-sized data transfer	*Disp6*[BP] ← ER*n*
		Disp6[FP] *pseg_addr*:*Disp6*[FP] DSR: *Disp6*[FP] R*d*: *Disp6*[FP]							Word-sized data transfer	*Disp6*[FP] ← ER*n*
		Dadr *pseg_addr*: *Dadr* DSR:*Dadr* R*d*:*Dadr*							Word-sized data transfer	[*Dadr*] ← ER*n*
	XR*n*	[EA] *pseg_addr*:[EA] DSR:[EA] R*d*:[EA]							Double word-sized data transfer	[EA] ← XR*n*
		[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*d*:[EA+]							Double word-sized data transfer	[EA] ← XR*n* EA←EA+1
	QR*n*	[EA] *pseg_addr*:[EA] DSR:[EA] R*d*:[EA]							Quad word-sized data transfer	[EA] ← QR*n*
		[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*d*:[EA+]							Quad word-sized data transfer	[EA] ← QR*n* EA←EA+1

Control Register Access Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC		Function
ADD	SP	*#signed8*							Addition	SP←SP+*signed8*
MOV	ECSR	R*m*							Data transfer	if ELEVEL is zero LCSR ←R*m* if ELEVEL is nonzero ECSR[ELEVEL] ←R*m*
	ELR	ER*m*							Data transfer	if ELEVEL is zero LR ← ER*m* if ELEVEL is nonzero ELR[ELEVEL] ← ER*m*
	EPSW	R*m*							Data transfer	if ELEVEL is nonzero EPSW[ELEVEL] ←R*m*
	ER*n*	ELR							Data transfer	if ELEVEL is zero ER*n* ←LR if ELEVEL is nonzero ER*n* ←ELR[ELEVEL]
		SP							Data transfer	ER*n*←SP
	PSW	R*m*	*	*	*	*	*	*	Data transfer	PSW← R*m*
		#unsigned8	*	*	*	*	*	*	Data transfer	PSW←*unsigned8*
	R*n*	ECSR							Data transfer	if ELEVEL is zero R*n* ← LCSR if ELEVEL is nonzero R*n* ← ECSR[ELEVEL]
		EPSW							Data transfer	if ELEVEL is nonzero R*n* ← EPSW[ELEVEL]
		PSW							Data transfer	R*n*←PSW
	SP	ER*m*							Data transfer	SP←ER*m*

PUSH/POP Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC		Function
PUSH	ER*n* R*n* QR*n* XR*n*								General register save	SP←SP-n Stack ← General register
	register_list								Control register save	SP←SP-n Stack ← Register set
POP	ER*n* R*n* QR*n* XR*n*								General register restore	General register ← Stack SP←SP+n
	register_list		*	*	*	*	*	*	Control register restore	Register set ← Stack[*1] SP←SP+n

*1: The program status word (PSW) only changes when it is included in *register_list*.

Coprocessor Data Transfer Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
MOV	CR*n*	R*m*							Byte-sized data transfer	CR*n* ← R*m*
	CR*n*	[EA] *pseg_addr*:[EA] DSR:[EA] R*d*:[EA]							Byte-sized data transfer	CR*n* ← [EA]
		[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*d*:[EA+]							Byte-sized data transfer	CR*n* ← [EA+] EA ← EA+1
	CER*n*	[EA] *pseg_addr*:[EA] DSR:[EA] R*d*:[EA]							Word-sized data transfer	CER*n* ← [EA]
		[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*d*:[EA+]							Word-sized data transfer	CER*n* ← [EA+] EA ← EA+1
	CXR*n*	[EA] *pseg_addr*:[EA] DSR:[EA] R*d*:[EA]							Double word-sized data transfer	CXR*n* ← [EA]
		[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*d*:[EA+]							Double word-sized data transfer	CXR*n* ← [EA+] EA ← EA+1
	CQR*n*	[EA] *pseg_addr*:[EA] DSR:[EA] R*d*:[EA]							Quad word-sized continuous data transfer	CQR*n* ← [EA]
		[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*d*:[EA+]							Quad word-sized continuous data transfer	CQR*n* ← [EA+] EA ← EA+1

(continued on next page)

Coprocessor Data Transfer Instructions (continued from previous page)

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
MOV	R*n*	CR*m*							Byte-sized data transfer	R*n* ←CR*m*
	[EA] *pseg_addr*:[EA] DSR:[EA] R*d*:[EA]	CR*m*							Byte-sized data transfer	[EA] ←CR*m*
	[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*d*:[EA+]	CR*m*							Byte-sized data transfer	[EA] ←CR*m* EA ←EA+1
	[EA] *pseg_addr*:[EA] DSR:[EA] R*d*:[EA]	CER*m*							Word-sized data transfer	[EA] ←CER*m*
	[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*d*:[EA+]	CER*m*							Word-sized data transfer	[EA]←CER*m* EA ←EA+1
	[EA] *pseg_addr*:[EA] DSR:[EA] R*d*:[EA]	CXR*m*							Double word-sized data transfer	[EA]←CXR*m*
	[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*d*:[EA+]	CXR*m*							Double word-sized data transfer	[EA] ←CXR*m* EA←EA+1
	[EA] *pseg_addr*:[EA] DSR:[EA] R*d*:[EA]	CQR*m*							Quad word-sized continuous data transfer	[EA] ←CQR*m*
	[EA+] *pseg_addr*:[EA+] DSR:[EA+] R*d*:[EA+]	CQR*m*							Quad word-sized continuous data transfer	[EA] ←CQR*m* EA←EA+1

EA Register Data Transfer Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
LEA	[ER*n*]								Data transfer to EA	EA← ER*n*
	Disp16[ER*m*]									EA← *Disp16*+ER*m*
	Dadr									EA← *Dadr*

ALU Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
DAA	R*n*		*	*	*			*	Byte-sized decimal adjustment for addition	
DAS	R*n*		*	*	*			*	Byte-sized decimal adjustment for subtraction	
NEG	R*n*		*	*	*	*		*	Negate	R*n* ← 0 – R*n*

Bit Access Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
SB	R*n.bit_offset*			*					Set bit	z← ~R*n. bit_offset* R*n. bit_offset* ←1
	Dbitadr			*					Set bit	z← ~[*Dbitadr*] [*Dbitadr*]← 1
	pseg_addr: *Dbitadr*			*						
	DSR: *Dbitadr*			*						
	R*d*: *Dbitadr*			*						
RB	R*n. bit_offset*			*					Reset bit	z← ~R*n.bit_offset* R*n.bit_offset* ←0
	Dbitadr			*					Reset bit	z← ~[*Dbitadr*] [*Dbitadr*]← 0
	pseg_addr: *Dbitadr*			*						
	DSR: *Dbitadr*			*						
	R*d*: *Dbitadr*			*						
TB	R*n. bit_offset*			*					Test bit	z← ~R*n.bit_offset*
	Dbitadr			*					Test bit	z← ~[*Dbitadr*]
	pseg_addr: *Dbitadr*			*						
	DSR: *Dbitadr*			*						
	R*d*: *Dbitadr*			*						

PSW Access Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
EI							*		Enable interrupts	MIE←1
DI							*		Disable interrupts	MIE←0
SC			*						Set carry flag	C←1
RC			*						Reset carry flag	C←0
CPLC			*						Complement carry flag	C←–C

Conditional Relative Branch Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
B*cond*	*Radr*								Conditional branch	if *cond* ? *Radr* : PC+2
BC	*cond*									

Sign Extension Instruction

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
EXTBW	ER*n*			*	*				Extend sign	ER*n*←(sign-extends)R*n*

Software Interrupt Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
SWI	*#snum*						*		Software interrupt instruction	address← (*snum*<<1), PC←Vector–table(address)
BRK									Break instruction	If ELEVEL greater than 1 System reset If ELEVEL less than 2 PC← (Vector–table 0004H)

Branch Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
B	*Cadr*								Branch instruction	CSR ← *Cadr*[19:16] PC ← *Cadr*[15:0]
	ER*n*									PC ← ER*n*
BL	*Cadr*								Branch instruction	LR ← Address of next instruction LCSR ← CSR CSR ← *Cadr*[19:16] PC ← *Cadr*[15:0]
	ER*n*									LR ← Address of next instruction LCSR ← CSR PC ← ER*n*

Multiplication and Division Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
MUL	ER*n*	R*m*		*					Multiplication	ER*n* ← R*n* * R*m*
DIV	ER*n*	R*m*	*	*					Division	ER*n* ← ER*n*/ R*m* , R*m*←ER*n* mod R*m*

Miscellaneous

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function	
INC	[EA]			*	*	*		*	Memory increment	[EA] ← [EA] + 1
	pseg_addr:[EA]			*	*	*		*		
	DSR:[EA]			*	*	*		*		
	R*d*:[EA]			*	*	*		*		
DEC	[EA]			*	*	*		*	Memory decrement	[EA] ← [EA] – 1
	pseg_addr:[EA]			*	*	*		*		
	DSR:[EA]			*	*	*		*		
	R*d*:[EA]			*	*	*		*		
RT									Return from subroutine	CSR ← LCSR PC ← LR
RTI			*	*	*	*	*	*	Return from interrupt	CSR ← ECSR[ELEVEL] PC ← ELR [ELEVEL] PSW ← EPSW[ELEVEL]
NOP										

3.3 Instruction Execution Times

This Section discusses nX-U8/100 core instruction execution times. To eliminate dependencies on clock frequency, it gives these times in clock cycles.

This Section also assumes that memory read and write cycles are all exactly one clock cycle long. In actual practice, however, execution times for instruction accessing slower memory will have to include memory wait cycles.

Each instruction takes at least three machine cycles to execute—one each for instruction fetch, instruction decode, and instruction execution plus result write. The nX-U8/100 architecture, however, pipelines instructions so that these three stages run in parallel, producing, under optimal conditions, faster execution than suggested by the machine cycles counts for the individual instructions. These execution times under optimal conditions are called minimum execution times.

Competition for CPU resources, however, mean that certain instruction sequences cannot run in parallel. The nX-U8/100 architecture resolves such conflicts by inserting a wait cycle at least one machine cycle long into the pipeline, delaying the execution of the later instruction.

There are following three conditions in which a wait cycle is inserted.

(1) Accessing ROM window addresses introduces a wait cycle of n × m machine cycles, where n is the number of bytes accessed and m the memory wait cycles for accessing a single byte. The handling of the Rom window region and the numbers of wait cycles inserted when the Rom window region is accessed differ for every product. Please refer to the manual of each product about the detailed number of cycles at the time of accessing the Rom window region.

(2) When the data region of the physical segment 0 is accessed using [EA+] addressing, the bus inside CPU competes and it becomes the factor which a wait cycle generates.

(3) The NMI interrupt and MI interrupt are influenced of [EA+] addressing. These interrupts require 3 cycles for hardware processing time,however, when the interruption occurs immediately after the physical segment 0 is accessed using [EA+] addressing, the interruption sequence is started after one machine cycle of wait cycles is performed .

The total execution time for an instruction, therefore, is the minimum execution time plus any wait cycles for resolving bus conflicts and any memory wait cycles.
The Table beginning on the next page lists these three quantities for all nX-U8/100 instructions. A blank indicates that the corresponding instruction either does not compete for CPU resources or does not access memory.

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
ADD	ER*n*	ER*m*	2		
		#imm7	2		
ADD	R*n*	R*m*	1		
		#imm8	1		
	SP	*#signed8*	2		
ADDC	R*n*	R*m*	1		
		#imm8	1		
AND	R*n*	R*m*	1		
		#imm8	1		
B	*Cadr*		2		1
	ER*n*		2		1
B*cond*	*Radr*		1 / 3(*1)		1
BL	*Cadr*		2		1
	ER*n*		2		1
BRK			7		1
CMP	ER*n*	ER*m*	2		
	R*n*	R*m*	1		
		#imm8	1		
CMPC	R*n*	R*m*	1		
		#imm8	1		
CPLC			1		
DAA	R*n*		1		
DAS	R*n*		1		
DEC	[EA]		2		1
	pseg_addr:[EA] DSR:[EA] R*d*:[EA]		3		
DI			3		
DIV	ER*n*	R*m*	16		
EI			1		
EXTBW	ER*n*		1		
INC	[EA]		2		1
	pseg_addr:[EA] DSR:[EA] R*d*:[EA]		3		

*1: The higher count is for when the branching condition is met; the lower one, for when the branching condition is not met.

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
L	ER*n*	[EA]	2	2	
		pseg_addr:[EA] DSR:[EA] R*d*:[EA]	3	2	
		[EA+]	2	2	
		pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	3	2	
		[ER*m*]	2	2	1
		pseg_addr:[ER*m*] DSR:[ER*m*] R*d*:[ER*m*]	3	2	
		Disp16[ER*m*]	3	2	1
		pseg_addr: *Disp16*[ER*m*] DSR:*Disp16*[ER*m*] R*d*: *Disp16*[ER*m*]	4	2	
		Disp6[BP]	3	2	1
		pseg_addr: *Disp6*[BP] DSR: *Disp6*[BP] R*d*: *Disp6*[BP]	4	2	
		Disp6[FP]	3	2	1
		pseg_addr: *Disp6*[FP] DSR: *Disp6*[FP] R*d*: *Disp6*[FP]	4	2	
		Dadr	2	2	1
		pseg_addr: *Dadr* DSR: *Dadr* R*d*: *Dadr*	3	2	
	QR*n*	[EA]	8	8	
		pseg_addr:[EA] DSR:[EA] R*d*:[EA]	9	8	
		[EA+]	8	8	
		pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	9	8	

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
L	R*n*	[EA]	1	1	
		pseg_addr:[EA] DSR:[EA] R*d*:[EA]	2	1	
		[EA+]	1	1	
		pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	2	1	
		[ER*m*]	1	1	1
		pseg_addr:[ER*m*] DSR:[ER*m*] R*d*:[ER*m*]	2	1	
		Disp16[ER*m*]	2	1	1
		pseg_addr: *Disp16*[ER*m*] DSR:*Disp16*[ER*m*] R*d*: *Disp16*[ER*m*]	3	1	
		Disp6[BP]	2	1	1
		pseg_addr: *Disp6*[BP] DSR: *Disp6*[BP] R*d*: *Disp6*[BP]	3	1	
		Disp6[FP]	2	1	1
		pseg_addr: *Disp6*[FP] DSR: *Disp6*[FP] R*d*: *Disp6*[FP]	3	1	
		Dadr	2	1	1
		pseg_addr: *Dadr* DSR: *Dadr* R*d*: *Dadr*	3	1	
	XR*n*	[EA]	4	4	
		pseg_addr:[EA] DSR:[EA] R*d*:[EA]	5	4	
		[EA+]	4	4	
		pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	5	4	

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
LEA	[ER*m*]		1		
	Disp16[ER*m*]		2		
	Dadr		2		
MOV	CER*n*	[EA]	2	2	1
		pseg_addr:[EA]	3	2	
		DSR:[EA]			
		R*d*:[EA]			
		[EA+]	2	2	1
		pseg_addr:[EA+]	3	2	
		DSR:[EA+]			
		R*d*:[EA+]			
	CQR*n*	[EA]	8	8	1
		pseg_addr:[EA]	9	8	
		DSR:[EA]			
		R*d*:[EA]			
		[EA+]	8	8	1
		pseg_addr:[EA+]	9	8	
		DSR:[EA+]			
		R*d*:[EA+]			
	CR*n*	[EA]	1	1	1
		pseg_addr:[EA]	2	1	
		DSR:[EA]			
		R*d*:[EA]			
		[EA+]	1	1	1
		pseg_addr:[EA+]	2	1	
		DSR:[EA+]			
		R*d*:[EA+]			
	CR*n*	R*m*	1		
	CXR*n*	[EA]	4	4	1
		pseg_addr:[EA]	5	4	
		DSR:[EA]			
		R*d*:[EA]			
		[EA+]	4	4	1
		pseg_addr:[EA+]	5	4	
		DSR:[EA+]			
		R*d*:[EA+]			
	ECSR	R*m*	2		
	ELR	ER*m*	3		
	EPSW	R*m*	2		
	ER*n*	ELR	3		
		ER*m*	2		
		#imm7	2		
		SP	2		

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
MOV	[EA]	CER*m*	2	2	1
	pseg_addr:[EA] DSR:[EA] R*d*:[EA]	CER*m*	3	2	
	[EA+]	CER*m*	2	2	1
	pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	CER*m*	3	2	
	[EA]	CQR*m*	8	8	1
	pseg_addr:[EA] DSR:[EA] R*d*:[EA]	CQR*m*	9	8	
	[EA+]	CQR*m*	8	8	1
	pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	CQR*m*	9	8	
	[EA]	CR*m*	1	1	1
	pseg_addr:[EA] DSR:[EA] R*d*:[EA]	CR*m*	2	1	
	[EA+]	CR*m*	1	1	1
	pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	CR*m*	2	1	
	[EA]	CXR*m*	4	4	1
	pseg_addr:[EA] DSR:[EA] R*d*:[EA]	CXR*m*	5	4	
	[EA+]	CXR*m*	4	4	1
	pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	CXR*m*	5	4	
	PSW	R*m*	1		
		#unsigned8	1		
	R*n*	CR*m*	1		
		ECSR	2		
		EPSW	2		
		PSW	1		
		R*m*	1		
		#imm8	1		
	SP	ER*m*	1		

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
MUL	ER*n*	R*m*	8		
NEG	R*n*		1		
NOP			1		
OR	R*n*	R*m*	1		
		#imm8	1		
POP	EA		4		1
	EA,LR		6 / 8 (*1)		1
	EA,PC		8 / 9 (*1)		1
	EA,PC,LR		10 / 13 (*1)		1
	EA,PC,PSW		10 / 11 (*1)		1
	EA,PC,PSW,LR		12 / 15 (*1)		1
	EA,PSW		6		1
	EA,PSW,LR		8 / 10 (*1)		1
	LR		2 / 4 (*1)		1
	LR,PSW		4 / 6 (*1)		1
	PC		4 / 5 (*1)		1
	PC,LR		6 / 9 (*1)		1
	PC,PSW		6 / 7 (*1)		1
	PC,PSW,LR		8 / 11 (*1)		1
	PSW		2		1
	ER*n*		2		1
	QR*n*		8		1
	R*n*		2		1
	XR*n*		4		1

*1: The lower count is for the SMALL memory model; the higher, for the LARGE model.

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
PUSH	EA		2		1
	ELR		2 / 4 (*1)		1
	EA,ELR		4 / 6 (*1)		1
	EPSW		2		1
	EPSW,EA		4		1
	EPSW,ELR		4 / 6 (*1)		1
	EPSW,ELR,EA		6 / 8 (*1)		1
	LR		2 / 4 (*1)		1
	LR,EA		4 / 6 (*1)		1
	LR,ELR		4 / 8 (*1)		1
	LR,EA,ELR		6 / 10 (*1)		1
	LR,EPSW		4 / 6 (*1)		1
	LR,EPSW,EA		6 / 8 (*1)		1
	LR,EPSW,ELR		6 / 10 (*1)		1
	LR,ELR,EPSW,EA		8 / 12 (*1)		1
	ER*n*		2		1
	QR*n*		8		1
	R*n*		2		1
	XR*n*		4		1
RB	*Dbitadr*		2		1
	pseg_addr: *Dbitadr* DSR: *Dbitadr* R*d*: *Dbitadr*		3		
	R*n*. *bit_offset*		1		
RC			1		
RT			2		1
RTI			2		1
SB	*Dbitadr*		2		1
	pseg_addr: *Dbitadr* DSR: *Dbitadr* R*d*: *Dbitadr*		3		
	R*n*.*bit_offset*		1		
SC			1		

*1: The lower count is for the SMALL memory model; the higher, for the LARGE model.

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
SLL	R*n*	R*m*	1		1
		#width	1		1
SLLC	R*n*	R*m*	1		1
		#width	1		1
SRA	R*n*	R*m*	1		1
		#width	1		1
SRL	R*n*	R*m*	1		1
		#width	1		1
SRLC	R*n*	R*m*	1		1
		#width	1		1
ST	ER*n*	[EA]	2		
		pseg_addr:[EA] DSR:[EA] R*d*:[EA]	3		
		[EA+]	2		
		pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	3		
		[ER*m*]	2		1
		pseg_addr:[ER*m*] DSR:[ER*m*] R*d*:[ER*m*]	3		
		Disp16[ER*m*]	3		1
		pseg_addr: *Disp16*[ER*m*] DSR:*Disp16*[ER*m*] R*d*: *Disp16*[ER*m*]	4		
		Disp6[BP]	3		1
		pseg_addr: *Disp6*[BP] DSR: *Disp6*[BP] R*d*: *Disp6*[BP]	4		
		Disp6[FP]	3		1
		pseg_addr: *Disp6*[FP] DSR: *Disp6*[FP] R*d*: *Disp6*[FP]	4		
		Dadr	2		1
		pseg_addr: *Dadr* DSR: *Dadr* R*d*: *Dadr*	3		

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
ST	QR*n*	[EA]	8		
		pseg_addr:[EA] DSR:[EA] R*d*:[EA]	9		
		[EA+]	8		
		pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	9		
	R*n*	[EA]	1		
		pseg_addr:[EA] DSR:[EA] R*d*:[EA]	2		
		[EA+]	1		
		pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	2		
		[ER*m*]	1		1
		pseg_addr:[ER*m*] DSR:[ER*m*] R*d*:[ER*m*]	2		
		Disp16[ER*m*]	2		1
		pseg_addr: *Disp16*[ER*m*] DSR:*Disp16*[ER*m*] R*d*: *Disp16*[ER*m*]	3		
		Disp6[BP]	2		1
		pseg_addr: *Disp6*[BP] DSR: *Disp6*[BP] R*d*: *Disp6*[BP]	3		
		Disp6[FP]	2		1
		pseg_addr: *Disp6*[FP] DSR: *Disp6*[FP] R*d*: *Disp6*[FP]	3		
		Dadr	2		1
		pseg_addr: *Dadr* DSR: *Dadr* R*d*: *Dadr*	3		
	XR*n*	[EA]	4		
		pseg_addr:[EA] DSR:[EA] R*d*:[EA]	5		
		[EA+]	4		
		pseg_addr:[EA+] DSR:[EA+] R*d*:[EA+]	5		

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
SUB	R*n*	R*m*	1		
SUBC	R*n*	R*m*	1		
SWI	*#snum*		3		1
TB	*Dbitadr*		2	1	1
	pseg_addr: *Dbitadr* DSR: *Dbitadr* R*d*: *Dbitadr*		3	1	
	R*n*. *bit_offset*		1		
XOR	R*n*	R*m*	1		
		#imm8	1		

3.4 Instruction Descriptions

The following Figure describes the layout of the instruction descriptions beginning on the next page. Using bit patterns other than those listed can produce unreliable execution. The instruction descriptions are one or two pages long with the instructions in alphabetical order. The following Figure indicates the major portions of these instruction descriptions.

General instruction syntax
This consists of the mnemonic plus general symbols indicating the operand types.

SB Dbitadr

Meta description of instruction operation
These symbols schematically outline instruction operation.

Function

Z ← ~[*Dbitadr*]
[*Dbitadr*] ←1

Detailed instruction description
This portion details instruction operation, operand notation, and limitations.

Description

- This instruction tests the specified bit by reading it from memory, inverting it, and storing the result in the Z flag. It then sets the original bit to “1.”
- The bit address *Dbitadr* has the format *Dadr16.bit*, where *bit* is an integer between 0 and 7 specifying the bit position within the memory byte.

Flags

C	Z	S	OV	MIE	HC
–	*	–	–	–	–

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
–: No change

Instruction code
This Table lists the addressing types available for each operand and the resulting bit patterns in the machine code for the instruction.

Instruction Format

Mnemonic	First operand	DSR prefix code	Instruction Format					
			First word					Second word
SB	*Dbitadr*		A	0	1	*bit*	0	*Dadr*
	*: *Dbitadr*	<word>	A	0	1	*bit*	0	*Dadr*

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

DSR prefix instruction codes
This Table lists the bit patterns for use in the portion of the first operand indicated with an asterisk in the immediately preceding Table.

ADD ER*n* , ER*m*

Add

Function

ER*n* ← ER*n* + ER*m*

Description

- This instruction adds the contents of the second word-sized register to those of the first and stores the result in the first.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	–	*

C:	This bit goes to “1” if the operation produces a carry out of bit 15 and to “0” otherwise.
Z:	This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
S:	This bit tracks the top bit of the result.
OV:	This bit goes to “1” if the operation produces overflow and to “0” otherwise.
HC:	This bit goes to “1” if the operation produces a carry out of or borrow into bit 11 and to “0” otherwise.
–:	No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
ADD	ER*n*	ER*m*	F	*n*	*m*	6	

ADD ER*n* , #*imm7*

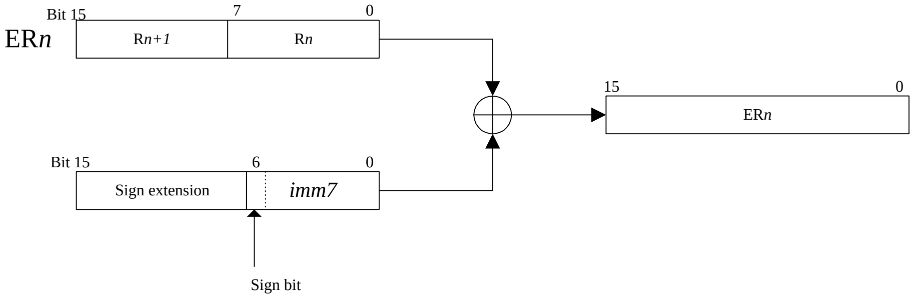
Add

Function

ER*n* ← ER*n* + (signed)*imm7*

Description

- This instruction adds the sign-extended immediate value to the contents of the specified word-sized register and stores the result in the register. The following Figure represents instruction operation schematically.



Flags

C	Z	S	OV	MIE	HC
*	*	*	*	–	*

C: This bit goes to “1” if the operation produces a carry out of bit 15 and to “0” otherwise.

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.

S: This bit tracks the top bit of the result.

OV: This bit goes to “1” if the operation produces overflow and to “0” otherwise.

HC: This bit goes to “1” if the operation produces a carry out of or borrow into bit 11 and to “0” otherwise.

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format	
			First word	Second word
ADD	ER*n*	#*imm7*	E *n* 1 *imm7*	

ADD R*n* , *obj*

Add

Function

R*n* ← R*n* + *obj*

Description

- This instruction adds the contents of the specified byte-sized object to those of the specified byte-sized register and stores the result in that register.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	–	*

C:	This bit goes to “1” if the operation produces a carry out of bit 7 and to “0” otherwise.
Z:	This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
S:	This bit tracks the top bit of the result.
OV:	This bit goes to “1” if the operation produces overflow and to “0” otherwise.
HC:	This bit goes to “1” if the operation produces a carry out of or borrow into bit 3 and to “0” otherwise.
–:	No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
ADD	R*n*	R*m*	8	*n*	*m*	1	
		#imm8	1	*n*	*imm8*		

ADD SP , #*signed8*

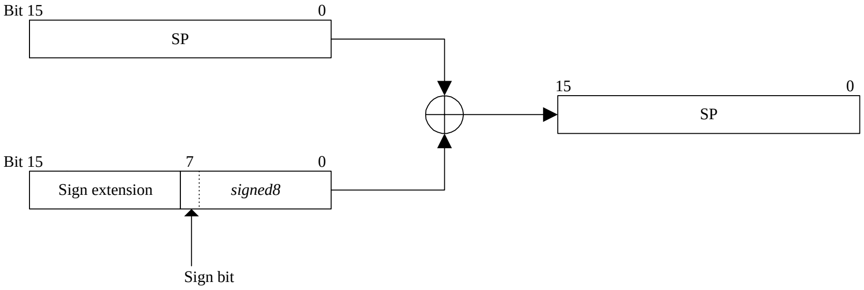
Add

Function

SP ← SP + signed *8*

Description

- This instruction adds the sign-extended *signed8* to the contents of the stack pointer and stores the result in the stack pointer.
- Bit 7 in *signed8* is interpreted as the sign bit, so signed8 is an integer quantity between -128 and +127. The following Figure represents instruction operation schematically.



Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change.

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format			
			First word			Second word
ADD	SP	*#signed8*	E	1	*signed8*	

ADDC R*n* , *obj*

Add with carry

Function

R*n* ← Rn + *obj* + C

Description

- This instruction adds the contents of the specified byte-sized register, the specified byte-sized object, and the carry flag C and stores the result in the register.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	–	*

C: This bit goes to "1" if the operation produces a carry out of bit 7 and to "0" otherwise.

Z: This flag remains "1" only if it was "1" before execution and the result is zero. Otherwise, it remains or goes to "0."

S: This bit tracks the top bit of the result.

OV: This bit goes to "1" if the operation produces overflow and to "0" otherwise.

HC: This bit goes to "1" if the operation produces a carry out of or borrow into bit 3 and to "0" otherwise.

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
ADDC	R*n*	R*m*	8	*n*	*m*	6	
		#imm8	6	*n*	*imm8*		

AND R*n* , *obj*

Bitwise AND

Function

R*n* ← R*n* & *obj*

Description

- This instruction ANDs the contents of the specified byte-sized register and object and stores the result in the register.

Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
S: This bit tracks the top bit of the result.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
AND	R*n*	R*m*	8	*n*	*m*	2	
		#imm8	2	*n*	*imm8*		

B Cadr

Direct branch

Function

CSR ← Cadr[19:16]
PC ← Cadr[15:0]

Description

- This instruction jumps to the specified address anywhere in the program/code memory space.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Instruction Format				
		First word				Second word
B	*Cadr*	F	*Cadr*[19:16]	0	0	*Cadr*[15:0]

B ER*n*

Indirect branch

Function

PC ← ER*n*

Description

- This instruction jumps within the same physical segment to the offset in the specified word-sized register.
- The program must load the target offset into the register before executing this instruction.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Instruction Format			
		First word			
B	ER*n*	F	0	*n*	2

Bcond Radr
BC *cond* , *Radr*

Conditional branch

Function

If (*cond* = true) then PC ← *Radr*
Note that the distance from the address of the next instruction (NextPC) to Radr must be between -128 and +127.

Description

- This instruction jumps to the specified address if the program status word (PSW) contents satisfy the specified condition.
- It assumes a preceding comparison or other instruction setting PSW flags for testing with this instruction.
- It is possible to specify the condition by two ways, one is to specify it as a part of mnemonic, and the other is to specify it as an operand.

Example

```
            CMP     R0,#21H
            BEQ     LABEL    ;The condition specifies as a part of mnemonic.
            CMP     R0,#56H
            BC      NC,LABEL ; The condition specifies as first operand.
                    :
                    :
LBAEL:
```

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

<table>
<tr><th rowspan="2">Mnemonic</th><th rowspan="2">First operand</th><th rowspan="2">Second operand</th><th colspan="3">Instruction Format</th></tr>
<tr><th colspan="3">First word</th></tr>
<tr><td>B<i>cond</i></td><td><i>Radr</i></td><td></td><td rowspan="2">C</td><td rowspan="2"><i>condition</i></td><td rowspan="2">(<i>Radr –NextPC</i>)>>1</td></tr>
<tr><td>BC</td><td><i>cond</i></td><td><i>Radr</i></td></tr>
</table>

Condition

<table>
<tr><th colspan="3">Instruction Syntax</th><th colspan="2"></th></tr>
<tr><th>B<i>cond</i></th><th>BC <i>cond</i></th><th><i>Condition</i></th><th>Meaning</th><th>Flag condition</th></tr>
<tr><td>BGE
BNC</td><td>BC GE
BC NC</td><td>0000</td><td>Unsigned ≥</td><td>C=0</td></tr>
<tr><td>BLT
BCY</td><td>BC LT
BC CY</td><td>0001</td><td>Unsigned <</td><td>C=1</td></tr>
<tr><td>BGT</td><td>BC GT</td><td>0010</td><td>Unsigned ></td><td>(C=0)&&(Z=0)</td></tr>
<tr><td>BLE</td><td>BC LE</td><td>0011</td><td>Unsigned ≤</td><td>(Z=1)||(C=1)</td></tr>
<tr><td>BGES</td><td>BC GES</td><td>0100</td><td>Signed ≥</td><td>(OV^S)=0</td></tr>
<tr><td>BLTS</td><td>BC LTS</td><td>0101</td><td>Signed <</td><td>(OV^S)=1</td></tr>
<tr><td>BGTS</td><td>BC GTS</td><td>0110</td><td>Signed ></td><td>((OV^S)| Z) = 0</td></tr>
<tr><td>BLES</td><td>BC LES</td><td>0111</td><td>Signed ≤</td><td>((OV^S)| Z) = 1</td></tr>
<tr><td>BNE
BNZ</td><td>BC NE
BC NZ</td><td>1000</td><td>!=</td><td>Z=0</td></tr>
<tr><td>BEQ
BZ</td><td>BC EQ
BC ZF</td><td>1001</td><td>=</td><td>Z=1</td></tr>
<tr><td>BNV</td><td>BC NV</td><td>1010</td><td>No overflow</td><td>OV=0</td></tr>
<tr><td>BOV</td><td>BC OV</td><td>1011</td><td>Overflow</td><td>OV=1</td></tr>
<tr><td>BPS</td><td>BC PS</td><td>1100</td><td>Positive</td><td>S=0</td></tr>
<tr><td>BNS</td><td>BC NS</td><td>1101</td><td>Negative</td><td>S=1</td></tr>
<tr><td>BAL</td><td>BC AL</td><td>1110</td><td>Unconditional</td><td></td></tr>
</table>

BL *Cadr*

Branch and link

Function

LR ← Address of next instruction
LCSR← CSR
CSR ← *Cadr*[19:16]
PC ← *Cadr*[15:0]

Description

- This instruction saves the address of the next instruction in the link register (LR) and the current CSR contents in the local code segment register (LCSR) and then jumps to the specified address anywhere in the program/code memory space.
- This instruction is for calling a subroutine. To return from the subroutine, use the RT instruction.
- If the subroutine calls another subroutine, it must use PUSH instructions to save the contents of the link (LR) and local code segment (LCSR) registers to the stack before the first such call and POP instructions to restore the link (LR) and local code segment (LCSR) registers after the last one.
- If a program uses this instruction in an interrupt handler, the interrupt handler must first use PUSH instructions to save the contents of the link (LR) and local code segment (LCSR) registers to the stack before calling the subroutine, and the subroutine must return with the corresponding POP instructions.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Instruction Format				
		First word				Second word
BL	*Cadr*	F	*Cadr*[19:16]	0	1	*Cadr*[15:0]

BL ER*n*

Branch and link

Function

PC ← ER*n*
LR ← Address of next instruction
LCSR ← CSR

Description

- This instruction saves the address of the next instruction in the link register (LR) and the current CSR contents in the local code segment register (LCSR) and then jumps within the same physical segment to the offset in the specified word-sized register.
- This instruction is for calling a subroutine. To return from the subroutine, use the RT instruction.
- If the subroutine calls another subroutine, it must use PUSH instructions to save the contents of the link (LR) and local code segment (LCSR) registers to the stack before the first such call and POP instructions to restore the link (LR) and local code segment (LCSR) registers after the last one.
- If a program uses this instruction in an interrupt handler, the interrupt handler must first use PUSH instructions to save the contents of the link (LR) and local code segment (LCSR) registers to the stack before calling the subroutine, and the subroutine must return with the corresponding POP instructions.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Instruction Format			
		First word			
BL	ER*n*	F	0	*n*	3

BRK

Break instruction
(software reset)

Function

- ELEVEL greater than 1: System reset
- ELEVEL less than 2:

  ELR2 ← Address of next instruction
  ECSR2 ← CSR
  EPSW2 ← PSW
  ELEVEL ← 2
  PC ← (Vector table 0004H)

Description

- This instruction is for resetting the user application system in software.
- An ELEVEL greater than 1 produces a CPU system reset, which
  1. initializes all internal CPU registers
  2. loads the stack pointer (SP) with the word data from address 0 in the code/program memory space
  3. loads the program counter (PC) with the word data from address 2 in the code/program memory space
- An ELEVEL less than 2 produces the equivalent of a nonmaskable interrupt. The CPU then loads the program counter (PC) with the word data from vector table address 4 at the beginning of the code/program memory space.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Instruction Format			
		First word			
BRK		F	F	F	F

CMP ER*n* , ER*m*

Compare

Function

ER*n* – ER*m*

Description

- This instruction compares the contents of the two specified word-sized registers by subtracting the latter from the former and setting the PSW flags for testing with a conditional branch or similar instruction.
- The register contents do not change.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	–	*

C: This bit goes to “1” if the operation produces a carry out of bit 15 and to “0” otherwise.
Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
S: This bit tracks the top bit of the result.
OV: This bit goes to “1” if the operation produces overflow and to “0” otherwise.
HC: This bit goes to “1” if the operation produces a carry out of or borrow into bit 11 and to “0” otherwise.
–: No change

Instruction Format

<table>
<tr><th rowspan="2">Mnemonic</th><th rowspan="2">First operand</th><th rowspan="2">Second operand</th><th colspan="5">Instruction Format</th></tr>
<tr><th colspan="4">First word</th><th>Second word</th></tr>
<tr><td>CMP</td><td>ER<i>n</i></td><td>ER<i>m</i></td><td>F</td><td><i>n</i></td><td><i>m</i></td><td>7</td><td></td></tr>
</table>

CMP R*n* , *obj*

Compare

Function

R*n* – *obj*

Description

- This instruction compares the contents of the specified byte-sized register and object by subtracting the latter from the former and setting the PSW flags for testing with a conditional branch or similar instruction.
- The register contents do not change.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	–	*

C: This bit goes to “1” if the operation produces a carry out of bit 7 and to “0” otherwise.
Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
S: This bit tracks the top bit of the result.
OV: This bit goes to “1” if the operation produces overflow and to “0” otherwise.
HC: This bit goes to “1” if the operation produces a carry out of or borrow into bit 3 and to “0” otherwise.
–: No change

Instruction Format

<table>
<tr><th rowspan="2">Mnemonic</th><th rowspan="2">First operand</th><th rowspan="2">Second operand</th><th colspan="5">Instruction Format</th></tr>
<tr><th colspan="4">First word</th><th>Second word</th></tr>
<tr><td rowspan="2">CMP</td><td rowspan="2">R<i>n</i></td><td>R<i>m</i></td><td>8</td><td><i>n</i></td><td><i>m</i></td><td>7</td><td></td></tr>
<tr><td><i>#imm8</i></td><td>7</td><td><i>n</i></td><td colspan="2"><i>imm8</i></td><td></td></tr>
</table>

CMPC R*n* , *obj*

Compare with carry

Function

R*n* – *obj*– C

Description

- This instruction compares the contents of the R*n* and obj by subtracting the latter and the carry flag from the former and setting the PSW flags for testing with a conditional branch or similar instruction.
- The register contents do not change.
- This instruction can be used after a CMP instruction to compare multibyte sequences.

Example: CMP R0, R4

CMPC R1, R5

Together, these two instructions compare the word-sized registers ER0 and ER4.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	–	*

C: This bit goes to “1” if the operation produces a carry out of bit 15 and to “0” otherwise.

Z: This flag remains “1” only if it was “1” before execution and the result is zero. Otherwise, it remains or goes to “0.”

S: This bit tracks the top bit of the result.

OV: This bit goes to “1” if the operation produces overflow and to “0” otherwise.

HC: This bit goes to “1” if the operation produces a carry out of or borrow into bit 3 and to “0” otherwise.

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
CMPC	R*n*	R*m*	8	*n*	*m*	5	
		#imm8	5	*n*	*imm8*		

CPLC

Complement carry flag

Function

C ← ~C

Description

- This instruction inverts the contents of the carry flag.

Flags

C	Z	S	OV	MIE	HC
*	–	–	–	–	–

C: Inversion of the original setting

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
CPLC			F	E	C	F	

DAA R*n*

Byte-sized decimal
adjustment for addition

Function

R*n* ← (decimal adjustment) R*n*

Description

- This instruction converts the contents of the specified byte-sized register into a binary coded decimal (BCD) value by adding the appropriate value, based on the contents of the register as well as the C and HC flags, from the following Table. An “X” indicates that the CPU does not care about the contents of that portion.

C	**R*n*[7:4]**	**HC**	**R*n*[3:0]**	**Adjustment**	**C flag after adjustment**
0	0–9	0	0–9	00	0
0	0–8	0	A–F	06	0
0	0–9	1	x	06	0
0	A–F	0	0–9	60	1
0	9–F	0	A–F	66	1
0	A–F	1	x	66	1
1	x	0	0–9	60	1
1	x	0	A–F	66	1
1	x	1	x	66	1

- A binary addition instruction (ADD Rn, obj) must precede this instruction, and any intervening instruction must not alter the contents of the register or the program status word (PSW).

Flags

C	Z	S	OV	MIE	HC
*	*	*	–	–	*

C: This flag goes to “1” if execution produces a carry into the 100s position. Otherwise, it remains unchanged.

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.

S: This bit tracks the top bit of the result.

HC: This bit goes to “1” if the operation produces a carry out of or borrow into bit 3 and to “0” otherwise.

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
DAA	R*n*		8	*n*	1	F	

DAS R*n*

Byte-sized decimal adjustment for subtraction

Function

R*n* ← (decimal adjustment) R*n*

Description

- This instruction converts the contents of the specified byte-sized register into a binary coded decimal (BCD) value by subtracting the appropriate value, based on the contents of the register as well as the C and HC flags, from the following Table.

An "X" indicates that the CPU does not care about the contents of that portion.

C	R*n*[7:4]	HC	R*n*[3:0]	Adjustment
0	0–9	0	0–9	00
0	0–9	0	A–F	06
0	0–9	1	x	06
0	A–F	0	0–9	60
0	A–F	1	x	66
0	A–F	0	A–F	66
1	x	0	0–9	60
1	x	1	x	66
1	x	0	A–F	66

- A binary subtraction instruction (SUB Rn, obj) must precede this instruction, and any intervening instruction must not alter the contents of the register or the program status word (PSW).

Flags

C	Z	S	OV	MIE	HC
*	*	*	–	–	*

C: This flag goes to "1" if execution produces a borrow from the 100s position. Otherwise, it remains unchanged.

Z: This bit goes to "1" if the operation produces a zero result and to "0" otherwise.

S: This bit tracks the top bit of the result.

HC: This bit goes to "1" if the operation produces a carry out of or borrow into bit 3 and to "0" otherwise.

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
DAS	R*n*		8	*n*	3	F	

DEC [EA]

Memory decrement
(using EA indirect addressing)

Function

[EA] ← [EA] –1

Description

- This instruction subtracts one from the byte at the address in the EA register.

Flags

C	Z	S	OV	MIE	HC
–	*	*	*	–	*

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
S: This bit tracks the top bit of the result.
OV: This bit goes to “1” if the operation produces overflow and to “0” otherwise.
HC: This bit goes to “1” if the operation produces a carry out of or borrow into bit 3 and to “0” otherwise.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
DEC	[EA]			F	E	3	F	
	*:[EA]		<word>	F	E	3	F	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

DI

Disable interrupts

Function

MIE ← 0

Description

- This instruction sets the master interrupt enable (MIE) bit to “0” to disable maskable interrupts.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	*	–

MIE: This goes to “0.”
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
DI			E	B	F	7	

DIV ER*n* , R*m*

Division

Function

ERn ← ER*n* / R*m*
Rm ← ER*n* mod R*m*

Description

- This instruction divides the contents of the specified word-sized register by those of the specified byte-sized register, stores the 16-bit dividend in the former, and stores the 8-bit remainder in the latter.
- A zero divisor sets the carry flag to “1” and leaves indeterminate values in both registers.

Flags

C	Z	S	OV	MIE	HC
*	*	–	–	–	–

C: This flag goes to “1” if the divisor is zero. Otherwise, it goes to “0.”
Z: This flag goes to “1” if the dividend is zero. Otherwise, it goes to “0.”
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
DIV	ER*n*	R*m*	F	*n*	*m*	9	

EI

Enable interrupts

Function

MIE ← 1

Description

- This instruction sets the master interrupt enable (MIE) bit to “1” to enable maskable interrupts.
- Note that the MIE bit does not go to “1” for three cycles from the start of this instruction, so the user application program must support maskable interrupts for the two cycles following this instruction.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	*	–

MIE: This goes to “1.”
–: No change

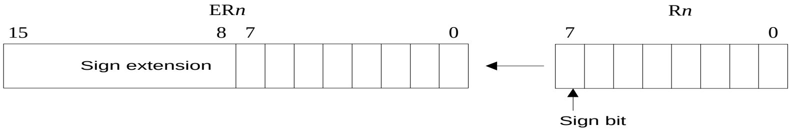
Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
EI			E	D	0	8	

EXTBW ER*n*

Extend sign

Function



Description

- This instruction extends the contents of the R*n* register to signed 16-bit format and stores it in the ER*n* register.
- The contents of the R*n*+1 are filled with bit 7 of the R*n* register, as the result.

Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This bit goes to “1” if the R*n* register value is zero and to “0” otherwise.

S: This bit tracks the bit 7 of the R*n* register.

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
EXTBW	ER*n*		8	*n*+1	*n*	F	

INC [EA]

Memory increment
(using EA indirect addressing)

Function

[EA] ← [EA] +1

Description

- This instruction adds one to the byte at the address in the EA register.

Flags

C	Z	S	OV	MIE	HC
–	*	*	*	–	*

Z:	This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
S:	This bit tracks the top bit of the result.
OV:	This bit goes to “1” if the operation produces overflow and to “0” otherwise.
HC:	This bit goes to “1” if the operation produces a carry out of or borrow into bit 3 and to “0” otherwise.
–:	No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
INC	[EA]			F	E	2	F	
	*:[EA]		<word>	F	E	2	F	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

L ERn, obj

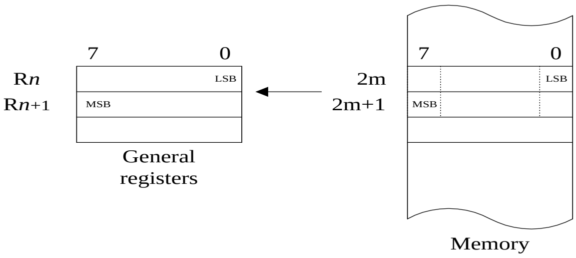
Word-sized data transfer

Function

ER*n* ← *obj*

Description

- This instruction loads the specified 16-bit register with the data at the specified word address.



Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
S: This bit tracks the top bit of the result.
–: No change

Instruction Format

(See next page)

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
L	ER*n*	[EA]		9	n	3	2	
		*: [EA]	<word>	9	n	3	2	
		[EA+]		9	*n*	5	2	
		*:[EA+]	<word>	9	*n*	5	2	
		[ER*m*]		9	*n*	*m*	2	
		*:[ER*m*]	<word>	9	*n*	*m*	2	
		Disp16[ER*m*]		A	*n*	*m*	8	*Disp16*
		*:*Disp16*[ER*m*]	<word>	A	*n*	*m*	8	*Disp16*
		Disp6[BP]		B	*n*	0 0	*Disp6*	
		*:*Disp6*[BP]	<word>	B	*n*	0 0	*Disp6*	
		Disp6[FP]		B	*n*	0 1	*Disp6*	
		*:*Disp6*[FP]	<word>	B	*n*	0 1	*Disp6*	
		Dadr		9	*n*	1	2	*Dadr*
		*: *Dadr*	<word>	9	*n*	1	2	*Dadr*

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

L QR*n*,*obj*

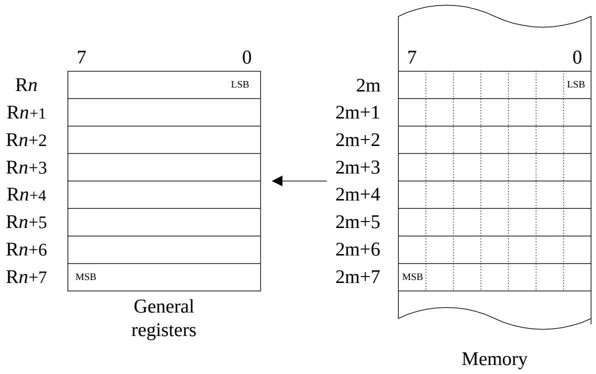
Quad word-sized
data transfer

Function

QR*n* ← *obj*

Description

- This instruction loads the specified 64-bit register with the data at the specified word address.



Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This flag goes to “1” if the new register contents are zero. Otherwise, it goes to “0.”
S: This bit tracks the top bit of the result.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
L	QR*n*	[EA]		9	*n*	3	6	
		*: [EA]	<word>	9	*n*	3	6	
		[EA+]		9	*n*	5	6	
		*:[EA+]	<word>	9	*n*	5	6	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

L R*n*, *obj*

Byte-sized data transfer

Function

R*n* ←*obj*

Description

- This instruction loads the specified 8-bit register with the data at the specified byte address.

Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This flag goes to “1” if the new register contents are zero. Otherwise, it goes to “0.”
S: This bit tracks the top bit of the result.
–: No change

Instruction Format

(See next page)

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
L	R*n*	[EA]		9	n	3	0	
		*: [EA]	<word>	9	n	3	0	
		[EA+]		9	*n*	5	0	
		*:[EA+]	<word>	9	*n*	5	0	
		[ERm]		9	*n*	*m*	0	
		*:[ERm]	<word>	9	*n*	*m*	0	
		Disp16[ER*m*]		9	*n*	*m*	8	*Disp16*
		*:*Disp16*[ER*m*]	<word>	9	*n*	*m*	8	*Disp16*
		Disp6[BP]		D	*n*	0 0	*Disp6*	
		*:*Disp6*[BP]	<word>	D	*n*	0 0	*Disp6*	
		Disp6[FP]		D	*n*	0 1	*Disp6*	
		*:*Disp6*[FP]	<word>	D	*n*	0 1	*Disp6*	
		Dadr		9	*n*	1	0	*Dadr*
		*: *Dadr*	<word>	9	*n*	1	0	*Dadr*

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

L XR*n*,*obj*

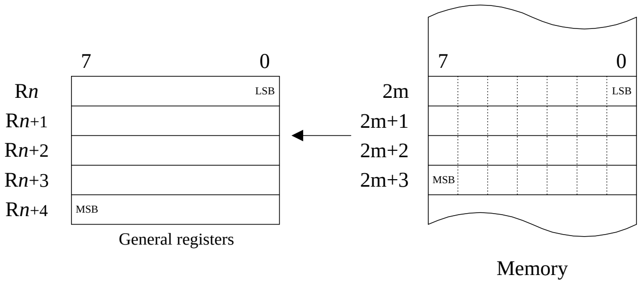
Double word-sized data transfer

Function

XR*n* ← *obj*

Description

- This instruction loads the specified 32-bit register with the data at the specified word address.



Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This flag goes to “1” if the new register contents are zero. Otherwise, it goes to “0.”
S: This bit tracks the top bit of the result.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
L	XR*n*	[EA]		9	*n*	3	4	
		*: [EA]	<word>	9	*n*	3	4	
		[EA+]		9	*n*	5	4	
		*:[EA+]	<word>	9	*n*	5	4	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

LEA *obj*

Load EA

Function

EA ← *obj*

Description

- This instruction loads the EA register with the specified word value.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
LEA	[ER*m*]		F	0	*m*	A	
	Dadr		F	0	0	C	*Dadr*
	Disp16[ER*m*]		F	0	*m*	B	*Disp16*

MOV CER*n* , *obj*

Coprocessor data transfer

Function

CER*n* ←*obj*

Description

- This instruction loads the specified coprocessor word-sized register from the specified word address.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
MOV	CER*n*	[EA]		F	*n*	2	D	
		*: [EA]	<word>	F	*n*	2	D	
		[EA+]		F	*n*	3	D	
		*:[EA+]	<word>	F	*n*	3	D	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

MOV CQR*n* , *obj*

Coprocessor data transfer

Function

CQR*n* ← *obj*

Description

- This instruction loads the specified coprocessor quad word-sized register from the specified word address.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
MOV	CQR*n*	[EA]		F	*n*	6	D	
		*: [EA]	<word>	F	*n*	6	D	
		[EA+]		F	*n*	7	D	
		*:[EA+]	<word>	F	*n*	7	D	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

MOV CR*n* , *obj*

Coprocessor data transfer

Function

CR*n* ←*obj*

Description

- This instruction loads the specified coprocessor byte-sized register from the specified byte address.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
MOV	CR*n*	[EA]		F	*n*	0	D	
		*: [EA]	<word>	F	*n*	0	D	
		[EA+]		F	*n*	1	D	
		*:[EA+]	<word>	F	*n*	1	D	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

MOV CR*n* , R*m*

Coprocessor data transfer

Function

CR*n* ←R*m*

Description

- This instruction loads the specified coprocessor byte-sized register from the specified byte-sized internal register.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	CR*n*	R*m*	A	*n*	*m*	E	

MOV CXR*n* , *obj*

Coprocessor data transfer

Function

CXR*n* ← *obj*

Description

- This instruction loads the specified coprocessor double word-sized register from the specified double word-sized internal register.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
MOV	CXR*n*	[EA]		F	*n*	4	D	
		*: [EA]	<word>	F	*n*	4	D	
		[EA+]		F	*n*	5	D	
		*:[EA+]	<word>	F	*n*	5	D	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

MOV ECSR , R*m*

Data transfer

Function

- If ELEVEL is zero

  LCSR ←R*m*
- If ELEVEL is nonzero

  ECSR[ELEVEL] ←R*m*

Description

- This instruction loads the contents of the specified register into the local code segment register (LCSR) if ELEVEL is zero and into the ECSR register (ECSR1 to ECSR3) for the current exception level (ELEVEL) setting otherwise.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	ECSR	R*m*	A	0	*m*	F	

MOV ELR , ER*m*

Data transfer

Function

- If ELEVEL is zero

  LR ← ER*m*
- If ELEVEL is nonzero

  ELR[ELEVEL] ← ER*m*

Description

- This instruction loads the contents of the specified word-sized register into the link register (LR) if ELEVEL is zero and into the exception link register (ELR1 to ELR3) for the current exception level (ELEVEL) setting otherwise.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	ELR	ER*m*	A	*m*	0	D	

MOV EPSW , R*m*

Data transfer

Function

- If ELEVEL is nonzero
  EPSW[ELEVEL] ←R*m*

Description

- This instruction loads the contents of the specified register into the exception program status word (EPSW1 to EPSW3) register for the current exception level (ELEVEL) setting if ELEVEL is nonzero.
- If ELEVEL is zero, this instruction does nothing. The program counter (PC) simply advances to the next instruction.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

<table>
<tr><th rowspan="2">Mnemonic</th><th rowspan="2">First operand</th><th rowspan="2">Second operand</th><th colspan="5">Instruction Format</th></tr>
<tr><th colspan="4">First word</th><th>Second word</th></tr>
<tr><td>MOV</td><td>EPSW</td><td>R<i>m</i></td><td>A</td><td>0</td><td><i>m</i></td><td>C</td><td></td></tr>
</table>

MOV ER*n* , ELR

Data transfer

Function

- If ELEVEL is zero

  ER*n* ←LR
- If ELEVEL is nonzero

  ER*n* ←ELR[ELEVEL]

Description

- This instruction loads the specified word-sized register from the link register (LR) if ELEVEL is zero and from the exception link register (ELR1 to ELR3) for the current exception level (ELEVEL) setting otherwise.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	ER*n*	ELR	A	*n*	0	5	

MOV ER*n* , ER*m*

Data transfer

Function

ER*n* ←ER*m*

Description

- This instruction loads the first word-sized register from the second.

Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This flag goes to “1” if the new register contents are zero. Otherwise, it goes to “0.”
S: This bit tracks the top bit of the result.
–: No change

Instruction Format

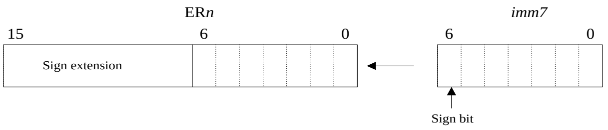
Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	ER*n*	ER*m*	F	*n*	*m*	5	

MOV ER*n* , #*imm7*

Data transfer

Function

ER*n* ← (sign–extends)*imm7*



Description

- This instruction loads the sign-extended *imm7* into the specified word-sized register. More precisely, it loads the immediate value into R*n*, the lower half of the register, and copies bit 6 from the immediate value into R*n* bit 7 and all bits of R*n*+1.

Example:

```
MOV     R0,#07Fh
MOV     R1,#0h
MOV     ER0,#-64    ; Execution replicates the top bit (“1”), setting R0 to 0C0H
                      and R1 to 0FFH

MOV     R0,#03Fh
MOV     R1,#0FFh
MOV     ER0,#3Fh    ; Execution replicates the top bit (“0”), setting R0 to 03FH
                      and R1 to 0H
```

Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This flag goes to “1” if the new register contents are zero. Otherwise, it goes to “0.”

S: This bit tracks the top bit of the result.

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format	
			First word	Second word
MOV	ER*n*	#*imm7*	E *n* 0 *imm7*	

MOV ER*n* , SP

Data transfer

Function

ER*n* ←SP

Description

- This instruction saves the contents of the stack pointer (SP) in the specified word-sized register.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	ER*n*	SP	A	*n*	1	A	

MOV *obj* , CER*m*

Coprocessor data transfer

Function

(WORD) *obj* ←CER*m*

Description

- This instruction saves the contents of the specified coprocessor word-sized register at the specified word address in the EA register.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
MOV	[EA]	CER*m*		F	*m*	A	D	
	* : [EA]	CER*m*	<word>	F	*m*	A	D	
	[EA+]	CER*m*		F	*m*	B	D	
	* : [EA+]	CER*m*	<word>	F	*m*	B	D	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

MOV *obj* , CQR*m*

Coprocessor data transfer

Function

(QWORD)*obj* ←CQR*m*

Description

- This instruction saves the contents of the specified coprocessor quad word-sized register at the specified word address in the EA register.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
MOV	[EA]	CQR*m*		F	*m*	E	D	
	*: [EA]	CQR*m*	<word>	F	*m*	E	D	
	[EA+]	CQR*m*		F	*m*	F	D	
	*: [EA+]	CQR*m*	<word>	F	*m*	F	D	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

MOV *obj* , CR*m*

Coprocessor data transfer

Function

(BYTE) *obj* ←CR*m*

Description

- This instruction saves the contents of the specified coprocessor byte-sized register at the specified byte address in the EA register.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
MOV	[EA]	CR*m*		F	*m*	8	D	
	*: [EA]	CR*m*	<word>	F	*m*	8	D	
	[EA+]	CR*m*		F	*m*	9	D	
	*: [EA+]	CR*m*	<word>	F	*m*	9	D	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

MOV *obj* , CXR*m*

Coprocessor data transfer

Function

(DOUBLE WORD)*obj* ← CXR*m*

Description

- This instruction saves the contents of the specified coprocessor double word-sized register at the specified word address in the EA register.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR Prefix code	Instruction Format				
				First word				Second word
MOV	[EA]	CXR*m*		F	*m*	C	D	
	*: [EA]	CXR*m*	<word>	F	*m*	C	D	
	[EA+]	CXR*m*		F	*m*	D	D	
	*: [EA+]	CXR*m*	<word>	F	*m*	D	D	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

MOV PSW , *obj*

Data transfer

Function

PSW ← *obj*

Description

- This instruction loads the program status word (PSW) from the specified byte-sized object.
- When the current exception level (ELEVEL) is changed, it is necessary to arrange an NOP instruction immediately after. Otherwise, the following command operates before changing ELEVEL, and as a result, the program might malfunction.

  Example:

  ```
  MOV PSW, #05h
  NOP
  RTI
  ```

- When the value of the master interrupt enable (MIE) bit is reset in 0, the DI instruction is used, and this instruction is not used. Otherwise, the MIE bit does not go to "0" for three cycles from the start of this instruction, as a result, maskable interrupt that the programmer doesn't intend is permitted, and there is a possibility that the application program malfunctions.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	*	*

*: Contents reflect the corresponding source bit.

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	PSW	*#unsigned8*	E	9	*unsigned8*		
		R*m*	A	0	*m*	B	

MOV R*n* , CR*m*

Coprocessor data transfer

Function

R*n* ← CR*m*

Description

- This instruction loads the specified byte-sized register from the specified coprocessor byte-sized register.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	R*n*	CR*m*	A	*n*	*m*	6	

MOV R*n* , ECSR

Data transfer

Function

- If ELEVEL is zero

  R*n* ← LCSR
- If ELEVEL is nonzero

  R*n* ← ECSR[ELEVEL]

Description

- This instruction loads the specified byte-sized register from the local code segment register (LCSR) if ELEVEL is zero and from the ECSR register (ECSR1 to ECSR3) for the current exception level (ELEVEL) setting otherwise.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	R*n*	ECSR	A	*n*	0	7	

MOV R*n* , EPSW

Data transfer

Function

- If ELEVEL is nonzero

  R*n* ← EPSW[ELEVEL]

Description

- This instruction loads the specified byte-sized register from the exception program status word (EPSW1 to EPSW3) register for the current exception level (ELEVEL) setting if ELEVEL is nonzero.
- If ELEVEL is zero, this instruction does nothing. The program counter (PC) simply advances to the next instruction.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

<table>
<tr><th rowspan="2">Mnemonic</th><th rowspan="2">First operand</th><th rowspan="2">Second operand</th><th colspan="5">Instruction Format</th></tr>
<tr><th colspan="4">First word</th><th>Second word</th></tr>
<tr><td>MOV</td><td>R<i>n</i></td><td>EPSW</td><td>A</td><td><i>n</i></td><td>0</td><td>4</td><td></td></tr>
</table>

MOV R*n* , PSW

Data transfer

Function

R*n* ← PSW

Description

- This instruction loads the specified byte-sized register from the program status word (PSW).

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	R*n*	PSW	A	*n*	0	3	

MOV R*n* , *obj*

Data transfer

Function

Rn ←*obj*

Description

- This instruction loads the specified byte-sized register from the specified byte-sized object.

Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This flag goes to “1” if the new register contents are zero. Otherwise, it goes to “0.”

S: This bit tracks the top bit of the result.

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	R*n*	R*m*	8	*n*	*m*	0	
		#imm8	0	*n*	*imm8*		

MOV SP , ER*m*

Data transfer

Function

SP ← ER*m*

Description

- This instruction loads the stack pointer (SP) from the specified word-sized register.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MOV	SP	ER*m*	A	1	*m*	A	

MUL ER*n*,R*m*

Multiplication

Function

ER*n* ← R*n* * R*m* (*n* must be even)

Description

- This instruction multiplies the contents of the two specified byte-size registers and stores the 16-bit product in the word-sized register corresponding to the first register.

Flags

C	Z	S	OV	MIE	HC
–	*	–	–	–	–

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
MUL	ER*n*	R*m*	F	*n*	*m*	4	

NEG R*n*

Negate

Function

R*n* ←0–R*n*

Description

- This instruction calculates the twos complement of the contents of the specified byte-size register and stores the result in that register.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	–	*

C: This bit goes to “1” if the operation produces a carry out of bit 7 and to “0” otherwise.
Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
S: This bit tracks the top bit of the result.
OV: This bit goes to “1” if the operation produces overflow and to “0” otherwise.
HC: This bit goes to “1” if the operation produces a carry out of or borrow into bit 3 and to “0” otherwise.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
NEG	R*n*		8	*n*	5	F	

NOP

No operation

Function

No operation

Description

- This instruction advances the program counter (PC) to the next instruction.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
NOP			F	E	8	F	

OR Rn , *obj*

Bitwise OR

Function

R*n* ←R*n* | *obj*

Description

- This instruction ORs the contents of the specified byte-sized register and object and stores the result in the register.

Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This flag goes to “1” if the new register contents are zero. Otherwise, it goes to “0.”
S: This bit tracks the top bit of the result.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
OR	R*n*	R*m*	8	*n*	*m*	3	
		#imm8	3	*n*	*imm8*		

POP register list

Restore control registers

Function

Control registers ← (SP)

SP ←SP + n

Description

- This instruction loads the specified control registers from the system stack pointed to by the stack pointer (SP) and then increments SP by the corresponding number of bytes. For further details, see Section 1.6 "Stack Modifications."
- The following control registers can appear in this list.
  - (1) EA register
  - (2) link register (LR) for saving the program counter (PC) when calling a subroutine
  - (3) program status word (PSW)
  - (4) program counter (PC)
- This list need not contain all, but it must contain at least one.
- This list can appear in any order, but the hardware always uses the order given below:

EA → LR → PSW → PC

- There is no automatic word alignment about the stack operations. Therefore, if the contents of the stack pointer are odd, that address is used as is.
- The normal procedure for returning from a subroutine or interrupt handler is with an RT or RTI instruction, respectively, but it is sometimes necessary to save the contents of backup registers to the stack with PUSH instructions when subroutines or interrupt handlers are nested and restore them with POP instructions afterward. For further details, see Section 1.4 "Exception Levels and Backup Registers."

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	*	*

*: Contents change only if PSW is on the list.

Instruction Format

Mnemonic	First operand	Instruction Format			
		First word			
POP	EA	F	1	8	E
	PC	F	2	8	E
	EA, PC	F	3	8	E
	PSW	F	4	8	E
	EA, PSW	F	5	8	E
	PC, PSW	F	6	8	E
	EA, PC, PSW	F	7	8	E
	LR	F	8	8	E
	EA, LR	F	9	8	E
	PC, LR	F	A	8	E
	EA, PC, LR	F	B	8	E
	LR, PSW	F	C	8	E
	EA, PSW, LR	F	D	8	E
	PC, PSW, LR	F	E	8	E
	EA, PC, PSW, LR	F	F	8	E

POP *obj*

Restore general registers

Function

General registers ← (SP)

SP ←SP + n

Description

- This instruction loads the specified general registers from the system stack pointed to by the stack pointer (SP) as it increments SP by the corresponding number of bytes.
- Because the stack operations are always word sized, this instruction with a byte-sized operand (R*n*) loads the specified register and then automatically introduces a dummy cycle that increments SP without modifying any other registers.
  For further details, see Section 1.6 “Stack Modifications.”
- There is no automatic word alignment about the stack operations. Therefore, if the contents of the stack pointer are odd, that address is used as is.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Instruction Format			
		First word			
POP	R*n*	F	*n*	0 0 0 0	E
	ER*n*	F	*n*	0 0 0 1	E
	XR*n*	F	*n*	0 0 1 0	E
	QR*n*	F	*n*	0 0 1 1	E

PUSH register list

Save control registers

Function

SP ←SP – n

(SP) ← Control registers

Description

- This instruction saves the specified control registers to the system stack pointed to by the stack pointer (SP) as it decrements SP by the corresponding number of bytes. For further details, see Section 1.6 “Stack Modifications.”
- The following control registers can appear in this list.
  (1) exception link register (ELR)
  (2) exception program status word (EPSW)
  (3) link register (LR) for saving the program counter (PC) when calling a subroutine
  (4) EA register
- This list can appear in any order, but the hardware always uses the order given below:

  ELR → EPSW → LR → EA
- This instruction assumes that preceding PUSH instructions have saved the specified control registers on the stack in the appropriate order.
- There is no automatic word alignment about the stack operations. Therefore, if the contents of the stack pointer are odd, that address is used as is.
- The normal procedure for returning from a subroutine or interrupt handler is with an RT or RTI instruction, respectively, but it is sometimes necessary to save the contents of backup registers to the stack with PUSH instructions when subroutines or interrupt handlers are nested and restore them with POP instructions afterward. For further details, see Section 1.4 “Exception Levels and Backup Registers.”

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Instruction Format			
		First word			
PUSH	EA	F	1	C	E
	ELR	F	2	C	E
	EA, ELR	F	3	C	E
	EPSW	F	4	C	E
	EPSW, EA	F	5	C	E
	EPSW, ELR	F	6	C	E
	EPSW, ELR, EA	F	7	C	E
	LR	F	8	C	E
	LR, EA	F	9	C	E
	LR, ELR	F	A	C	E
	LR, EA, ELR	F	B	C	E
	LR, EPSW	F	C	C	E
	LR, EPSW, EA	F	D	C	E
	LR, EPSW, ELR	F	E	C	E
	LR, EPSW, ELR, EA	F	F	C	E

PUSH *obj*

Save general registers

Function

SP ←SP – n

(SP) ← General registers

Description

- This instruction loads the specified general registers from the system stack pointed to by the stack pointer (SP) as it decrements SP by the corresponding number of bytes.
- Because Stack operations are always word sized, this instruction with a byte-sized operand (R*n*) loads the specified register and then automatically introduces a dummy cycle that decrements SP without modifying any other registers.
  For further details, see Section 1.6 "Stack Modifications."
- There is no automatic word alignment about the stack operations. Therefore, if the contents of the stack pointer are odd, that address is used as is.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Instruction Format						
		First word						
PUSH	R*n*	F	*n*	0	1	0	0	E
	ER*n*	F	*n*	0	1	0	1	E
	XR*n*	F	*n*	0	1	1	0	E
	QR*n*	F	*n*	0	1	1	1	E

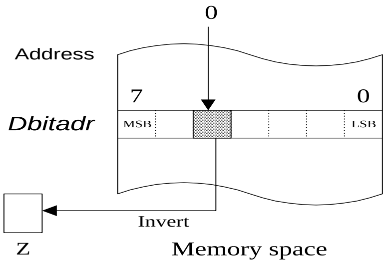
RB Dbitadr

Reset bit

Function

Z ← ~[*Dbitadr*]

[*Dbitadr*] ← 0



Description

- This instruction tests the specified bit by reading it from memory, inverting it, and storing the result in the Z flag. It then resets the original bit to “0.”
- The bit address *Dbitadr* has the format *Dadr.bit,* where *bit* is an integer between 0 and 7 specifying the bit position within the memory byte.

Flags

C	Z	S	OV	MIE	HC
–	*	–	–	–	–

Z: Inverse of the original bit

–: No change

Instruction Format

Mnemonic	First operand	DSR prefix code	Instruction Format					
			First word					Second word
RB	*Dbitadr*		A	0	1	*bit*	2	*Dadr*
	*: *Dbitadr*	<word>	A	0	1	*bit*	2	*Dadr*

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

RB R*n* . *bit_offset*

Reset bit

Function

Z ← ~Rn[*bit_offset*]

Rn[*bit_offset*] ← 0

Description

- This instruction reads the specified bit from the specified byte-sized register, inverts it, and stores it in the Z flag. It then resets the original bit to “0.”
- *bit_offset* is an integer between 0 and 7 specifying the bit position within the register.

Flags

C	Z	S	OV	MIE	HC
–	*	–	–	–	–

Z: Inverse of the original bit

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format	
			First word	Second word
RB	R*n*.*bit_offset*		A *n* 0 *bit* 2	

RC

Reset carry flag

Function

C ← 0

Description

- This instruction resets the carry flag to “0.”

Flags

C	Z	S	OV	MIE	HC
*	–	–	–	–	–

C: This goes to “0.”
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
RC			E	B	7	F	

RT

Return from subroutine

Function

CSR ← LCSR
PC ← LR

Description

- This instruction is for returning from a subroutine called with a BL instruction. It restores the address of the instruction following the BL instruction by loading the code segment register from the local code segment register (LCSR) and the program counter (PC) from the link register (LR).

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
RT			F	E	1	F	

RTI

Return from interrupt

Function

CSR ← ECSR[ELEVEL]
PC ← ELR [ELEVEL]
PSW ← EPSW[ELEVEL]

Description

- This instruction is for returning from an interrupt handler. It restores the program status word (PSW) and program counter (PC) from the exception program status word (EPSW1 to EPSW3) register and exception link register (ELR1 to ELR3), respectively, for the current exception level (ELEVEL) setting—1 for maskable interrupts and 2 for nonmaskable ones.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	*	*

*: Contents reflect the corresponding EPSW bit.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
RTI			F	E	0	F	

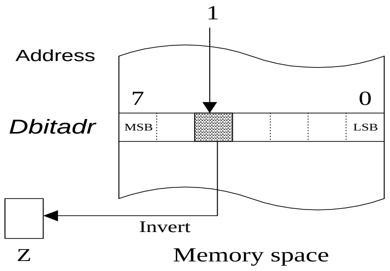
SB Dbitadr

Set bit

Function

Z ← ~[*Dbitadr*]

[*Dbitadr*] ←1



Description

- This instruction tests the specified bit by reading it from memory, inverting it, and storing the result in the Z flag. It then sets the original bit to “1.”
- The bit address *Dbitadr* has the format *Dadr16.bit*, where *bit* is an integer between 0 and 7 specifying the bit position within the memory byte.

Flags

C	Z	S	OV	MIE	HC
–	*	–	–	–	–

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.

–: No change

Instruction Format

Mnemonic	First operand	DSR prefix code	Instruction Format					
			First word					Second word
SB	*Dbitadr*		A	0	1	*bit*	0	*Dadr*
	*: *Dbitadr*	<word>	A	0	1	*bit*	0	*Dadr*

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

SB R*n* . *bit_offset*

Set bit

Function

Z ← ~R*n*[*bit_offset*]
Rn[*bit_offset*] ← 1

Description

- This instruction reads the specified bit from the specified byte-sized register, inverts it, and stores it in the Z flag. It then sets the original bit to “1.”
- *bit_offset* is an integer between 0 and 7 specifying the bit position within the register.

Flags

C	Z	S	OV	MIE	HC
–	*	–	–	–	–

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format	
			First word	Second word
SB	R*n*.*bit_offset*		A *n* 0 bit 0	

SC

Set carry flag

Function

C ← 1

Description

- This instruction sets the carry flag to “1.”

Flags

C	Z	S	OV	MIE	HC
*	–	–	–	–	–

C: This goes to “1.”

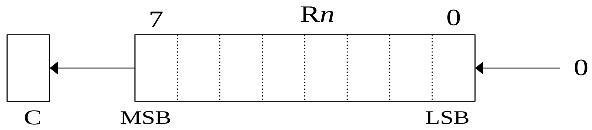
Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
SC			E	D	8	0	

SLL R*n* , *obj*

Shift left logical

Function



Description

- This instruction shifts the bits in the specified byte-sized register left the number of places specified by the second operand and shifts in zeros from the right. The carry flag retains the last bit shifted out.
- The meaningful range for shift sizes is 0 to 7. If the second operand is a byte-sized register, the hardware ignores bits 7 to 3 in that register and uses only the lowest three bits, thus restricting the shift size to the range 0 to 7. A shift size of 0 produces the equivalent of a NOP instruction. Preceding this instruction with a sequence of SLLC instructions permits a shift operation on longer bit sequences in multiple registers. (See SLLC example.)

Flags

C	Z	S	OV	MIE	HC
*	–	–	–	–	–

C: This bit retains the last bit shifted out.

–: No change

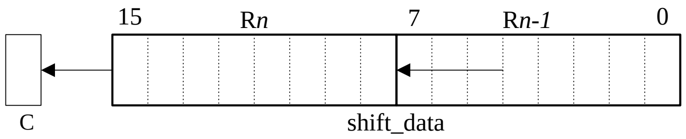
Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
SLL	R*n*	R*m*	8	*n*	*m*	A	
		#*width*	9	*n*	0 *width*	A	

SLLC R*n* , *obj*

Shift left logical continued

Function



Description

- This instruction shifts the 16 bits in the specified byte-sized register and the register below it (or R15 if R0 is specified) left the number of places specified by the second operand (up to a maximum of 7 places) and stores the upper eight bits in the specified register. The carry flag retains the last bit shifted out.
- The meaningful range for shift sizes is 0 to 7. If the second operand is a byte-sized register, the hardware ignores bits 7 to 3 in that register and uses only the lowest three bits, thus restricting the shift size to the range 0 to 7. A shift size of 0 produces the equivalent of a NOP instruction.
- A sequence of these instructions followed by an SLL instruction permits a shift operation on longer bit sequences in multiple registers. (See example.)

Example: Shift left for double word data

```
SLLC  R3, R5
SLLC  R2, R5
SLLC  R1, R5
SLL   R0, R5          This completes shift of XR0 contents
```

Flags

C	Z	S	OV	MIE	HC
*	–	–	–	–	–

C: This bit retains the last bit shifted out.

–: No change

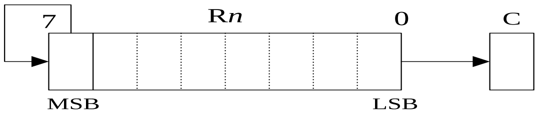
Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
SLLC	R*n*	R*m*	8	*n*	*m*	B	
		#*width*	9	*n*	0 *width*	B	

SRA R*n* , *obj*

Shift right arithmetic

Function



Description

- This instruction shifts the bits in the specified byte-sized register right the number of places specified by the second operand and shifts in duplicates of the original sign bit (bit 7) from the left. The carry flag retains the last bit shifted out.
- The meaningful range for shift sizes is 0 to 7. If the second operand is a byte-sized register, the hardware ignores bits 7 to 3 in that register and uses only the lowest three bits, thus restricting the shift size to the range 0 to 7. A shift size of 0 produces the equivalent of a NOP instruction.

Flags

C	Z	S	OV	MIE	HC
*	–	–	–	–	–

C: This bit retains the last bit shifted out.

–: No change

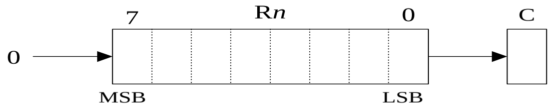
Instruction Format

Mnemonic	First operand	Second operand	Instruction Format					
			First word					Second word
SRA	R*n*	R*m*	8	*n*	*m*		E	
		#*width*	9	*n*	0	*width*	E	

SRL R*n* , *obj*

Shift right logical

Function



Description

- This instruction shifts the bits in the specified byte-sized register right the number of places specified by the second operand and shifts in zeros from the left. The carry flag retains the last bit shifted out.
- The meaningful range for shift sizes is 0 to 7. If the second operand is a byte-sized register, the hardware ignores bits 7 to 3 in that register and uses only the lowest three bits, thus restricting the shift size to the range 0 to 7. A shift size of 0 produces the equivalent of a NOP instruction. Preceding this instruction with a sequence of SRLC instructions permits a shift operation on longer bit sequences in multiple registers. (See SRLC example.)

Flags

C	Z	S	OV	MIE	HC
*	–	–	–	–	–

C: This bit retains the last bit shifted out.

–: No change

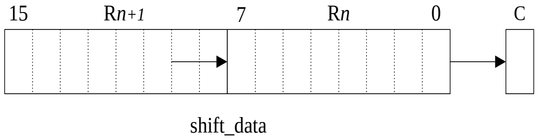
Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
SRL	R*n*	R*m*	8	*n*	*m*	C	
		#*width*	9	*n*	0 *width*	C	

SRLC R*n* , *obj*

Shift right logical continued

Function



Description

- This instruction shifts the 16 bits in the specified byte-sized register and the register above it (or R0 if R15 specified) right the number of places specified by the second operand (up to a maximum of 7 places) and stores the lower eight bits in the specified register. The carry flag retains the last bit shifted out.
- The meaningful range for shift sizes is 0 to 7. If the second operand is a byte-sized register, the hardware ignores bits 7 to 3 in that register and uses only the lowest three bits, thus restricting the shift size to the range 0 to 7. A shift size of 0 produces the equivalent of a NOP instruction.

Example: Shift right for double word data

```
SRLC  R0, R5
SRLC  R1, R5
SRLC  R2, R5
SRL   R3, R5         This completes shift of XR0 contents
```

Flags

C	Z	S	OV	MIE	HC
*	–	–	–	–	–

C: This bit retains the last bit shifted out.

–: No change

Instruction Format

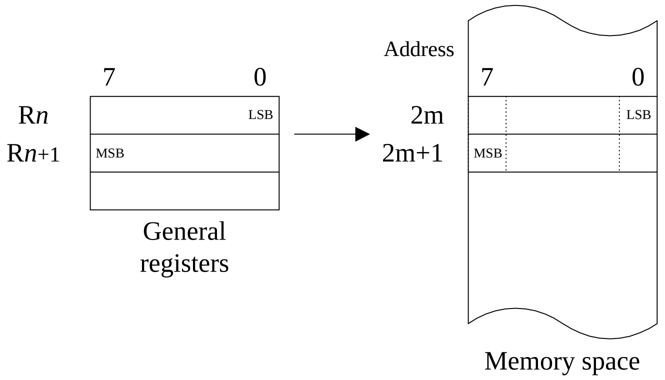
Mnemonic	First operand	Second operand	Instruction Format	
			First word	Second word
SRLC	R*n*	R*m*	8 *n* *m* D	
		#*width*	9 *n* 0 *width* D	

ST ER*n* , *obj*

Word-sized data transfer

Function

obj ← ER*n*



Description

- This instruction stores the contents of the specified 16-bit register at the specified word address.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

(See next page)

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
ST	ER*n*	[EA]		9	*n*	3	3	
		*: [EA]	<word>	9	*n*	3	3	
		[EA+]		9	*n*	5	3	
		*:[EA+]	<word>	9	*n*	5	3	
		[ER*m*]		9	*n*	*m*	3	
		*:[ER*m*]	<word>	9	*n*	*m*	3	
		Disp16[ER*m*]		A	*n*	*m*	9	*Disp16*
		*:*Disp16*[ER*m*]	<word>	A	*n*	*m*	9	*Disp16*
		Disp6[BP]		B	*n*	1 0	*Disp6*	
		*:*Disp6*[BP]	<word>	B	*n*	1 0	*Disp6*	
		Disp6[FP]		B	*n*	1 1	*Disp6*	
		*:*Disp6*[FP]	<word>	B	*n*	1 1	*Disp6*	
		Dadr		9	*n*	1	3	*Dadr*
		*: *Dadr*	<word>	9	*n*	1	3	*Dadr*

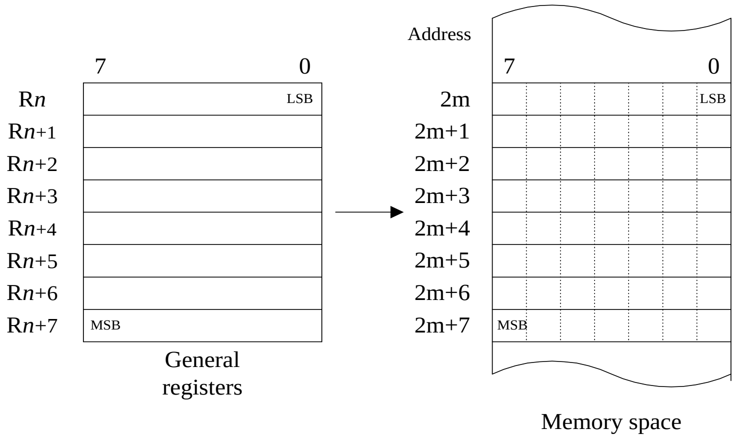
*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

ST QR*n* , *obj*

Quad word-sized
data transfer

Function

obj ← QR*n*



Description

- This instruction stores the contents of the specified 64-bit register at the specified word address.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
ST	QR*n*	[EA]		9	*n*	3	7	
		*: [EA]	<word>	9	*n*	3	7	
		[EA+]		9	*n*	5	7	
		*:[EA+]	<word>	9	*n*	5	7	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

ST R*n* , *obj*

Byte-sized data transfer

Function

obj ← R*n*

Description

- This instruction stores the contents of the specified 8-bit register at the specified address.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–:   No change

Instruction Format

(See next page)

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
ST	R*n*	[EA]		9	n	3	1	
		*: [EA]	<word>	9	n	3	1	
		[EA+]		9	*n*	5	1	
		*:[EA+]	<word>	9	*n*	5	1	
		[ER*m*]		9	*n*	*m*	1	
		*:[ER*m*]	<word>	9	*n*	*m*	1	
		Disp16[ER*m*]		9	*n*	*m*	9	*Disp16*
		*:*Disp16*[ER*m*]	<word>	9	*n*	*m*	9	*Disp16*
		Disp6[BP]		D	*n*	1 0	*Disp6*	
		*:*Disp6*[BP]	<word>	D	*n*	1 0	*Disp6*	
		Disp6[FP]		D	*n*	1 1	*Disp6*	
		*:*Disp6*[FP]	<word>	D	*n*	1 1	*Disp6*	
		Dadr		9	*n*	1	1	*Dadr*
		*: *Dadr*	<word>	9	*n*	1	1	*Dadr*

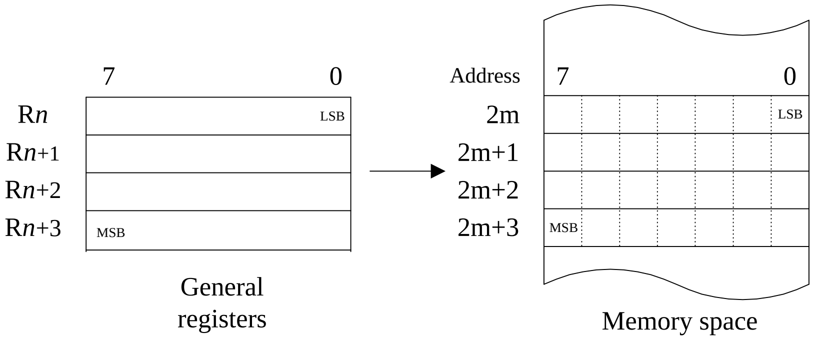
*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

ST XR*n* , *obj*

Double word-sized
data transfer

Function

obj ← XR*n*



Description

- This instruction stores the contents of the specified 32-bit register at the specified word address.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Second operand	DSR prefix code	Instruction Format				
				First word				Second word
ST	XR*n*	[EA]		9	*n*	3	5	
		*: [EA]	<word>	9	*n*	3	5	
		[EA+]		9	*n*	5	5	
		*:[EA+]	<word>	9	*n*	5	5	

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

SUB R*n* , R*m*

Subtract

Function

R*n* ← R*n* – R*m*

Description

- This instruction subtracts the contents of the second byte-sized register from those of the first and stores the result in the first.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	–	*

C: This bit goes to "1" if the operation produces a borrow into bit 7 and to "0" otherwise.
Z: This bit goes to "1" if the operation produces a zero result and to "0" otherwise.
S: This bit tracks the top bit of the result.
OV: This bit goes to "1" if the operation produces overflow and to "0" otherwise.
HC: This bit goes to "1" if the operation produces a carry out of or borrow into bit 3 and to "0" otherwise.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
SUB	R*n*	R*m*	8	*n*	*m*	8	

SUBC R*n* , R*m*

Subtract with carry

Function

R*n* ← R*n* – R*m* – C

Description

- This instruction subtracts the contents of the second byte-sized register and the carry flag from the contents of the first register and stores the result in the first register.

Flags

C	Z	S	OV	MIE	HC
*	*	*	*	–	*

C: This bit goes to "1" if the operation produces a borrow into bit 7 and to "0" otherwise.
Z: This flag remains "1" only if it was "1" before execution and the result is zero. Otherwise, it remains or goes to "0."
S: This bit tracks the top bit of the result.
OV: This bit goes to "1" if the operation produces overflow and to "0" otherwise.
HC: This bit goes to "1" if the operation produces a carry out of or borrow into bit 3 and to "0" otherwise.
–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
SUBC	R*n*	R*m*	8	*n*	*m*	9	

SWI #*snum*

Software interrupt

Function

EPSW1 ←PSW
ELEVEL ← 1
ELR1 ← PC+2
ECSR1 ←CSR
MIE ←0
PC ←TABLE[*snum*<<1]

Description

- This instruction loads the specified vector table entry into the program counter (PC). The operand is an integer between 0 and 63. During the interrupt cycle, this instruction also saves the address of the next instruction in the ELR1 register.

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	*	–

MIE: This goes to "0."
–: No change

Instruction Format

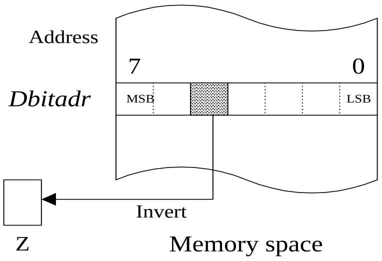
Mnemonic	First operand	Instruction Format					
		First word					Second word
SWI	#*snum*	E	5	0	0	*snum*	

TB Dbitadr

Test bit

Function

Z ← ~[*Dbitadr*]



Description

- This instruction tests the specified bit by reading it from memory, inverting it, and storing the result in the Z flag.
- The bit address *Dbitadr* has the format *Dadr16.bit*, where *bit* is an integer between 0 and 7 specifying the bit position within the memory byte.

Flags

C	Z	S	OV	MIE	HC
–	*	–	–	–	–

Z: This bit goes to "1" if the operation produces a zero result and to "0" otherwise.
–: No change

Instruction Format

Mnemonic	First operand	DSR prefix code	Instruction Format					
			First word					Second word
TB	*Dbitadr*		A	0	1	*bit*	1	*Dadr*
	*: *Dbitadr*	<word>	A	0	1	*bit*	1	*Dadr*

*	<word>			
pseg_addr	E	3	*pseg_addr*	
DSR	F	E	9	F
R*d*	9	0	*d*	F

TB Rn . bit_offset

Test bit

Function

Z ← ~R*n*[bit_offset]

Description

- This instruction tests the specified bit by reading it from memory, inverting it, and storing the result in the Z flag.
- *bit_offset* is an integer between 0 and 7 specifying the bit position within the register.

Flags

C	Z	S	OV	MIE	HC
–	*	–	–	–	–

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format					
			First word					Second word
TB	R*n*.bit		A	*n*	0	bit	1	

XOR R*n* , *obj*

Bitwise exclusive OR

Function

R*n* ←R*n*^ *obj*

Description

- This instruction XORs the contents of the specified byte-sized register and object and stores the result in the register.

Flags

C	Z	S	OV	MIE	HC
–	*	*	–	–	–

Z: This bit goes to “1” if the operation produces a zero result and to “0” otherwise.

S: This bit tracks the top bit of the result.

–: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format				
			First word				Second word
XOR	R*n*	R*m*	8	*n*	*m*	4	
		#imm8	4	*n*	*imm8*		

4. Appendix

This appendix lists the nX-U8/100 core instructions in functional groups, giving the operand syntax and instruction code for each instruction.

The descriptions of the DSR prefix instructions are omitted in this chapter. Threfore, please refer to Chapter 3 about the details of each instruction.

Arithmetic Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
ADD	R*n*	R*m*	*	*	*	*		*	1000_*nnnn*_*mmmm*_0001		1
		#imm8	*	*	*	*		*	0001_*nnnn*_iiii_iiii		1
ADD	ER*n*	ER*m*	*	*	*	*		*	1111_*nnn*0_*mmm*0_0110		2
		#imm7	*	*	*	*		*	1110_*nnn*0_1iii_iiii		2
ADDC	R*n*	R*m*	*	*	*	*		*	1000_*nnnn*_*mmmm*_0110		1
		#imm8	*	*	*	*		*	0110_*nnnn*_iiii_iiii		1
AND	R*n*	R*m*		*	*				1000_*nnnn*_*mmmm*_0010		1
		#imm8		*	*				0010_*nnnn*_iiii_iiii		1
CMP	R*n*	R*m*	*	*	*	*		*	1000_*nnnn*_*mmmm*_0111		1
		#imm8	*	*	*	*		*	0111_*nnnn*_iiii_iiii		1
CMPC	R*n*	R*m*	*	*	*	*		*	1000_*nnnn*_*mmmm*_0101		1
		#imm8	*	*	*	*		*	0101_*nnnn*_iiii_iiii		1
MOV	ER*n*	ER*m*		*	*				1111_*nnn*0_*mmm*0_0101		2
		#imm7		*	*				1110_*nnn*0_0iii_iiii		2
MOV	R*n*	R*m*		*	*				1000_*nnnn*_*mmmm*_0000		1
		#imm8		*	*				0000_*nnnn*_iiii_iiii		1
OR	R*n*	R*m*		*	*				1000_*nnnn*_*mmmm*_0011		1
		#imm8		*	*				0011_*nnnn*_iiii_iiii		1
XOR	R*n*	R*m*		*	*				1000_*nnnn*_*mmmm*_0100		1
		#imm8		*	*				0100_*nnnn*_iiii_iiii		1
CMP	ER*n*	ER*m*	*	*	*	*		*	1111_*nnn*0_*mmm*0_0111		2
SUB	R*n*	R*m*	*	*	*	*		*	1000_*nnnn*_*mmmm*_1000		1
SUBC	R*n*	R*m*	*	*	*	*		*	1000_*nnnn*_*mmmm*_1001		1

Shift Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
SLL	R*n*	R*m*	*						1000_*nnnn*_*mmmm*_1010		1
		#width	*						1001_*nnnn*_0www_1010		1
SLLC	R*n*	R*m*	*						1000_*nnnn*_*mmmm*_1011		1
		#width	*						1001_*nnnn*_0www_1011		1
SRA	R*n*	R*m*	*						1000_*nnnn*_*mmmm*_1110		1
		#width	*						1001_*nnnn*_0www_1110		1
SRL	R*n*	R*m*	*						1000_*nnnn*_*mmmm*_1100		1
		#width	*						1001_*nnnn*_0www_1100		1
SRLC	R*n*	R*m*	*						1000_*nnnn*_*mmmm*_1101		1
		#width	*						1001_*nnnn*_0www_1101		1

Load/Store Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
L	ER*n*	[EA]		*	*				1001_*nnn*0_0011_0010		2
		[EA+]		*	*				1001_*nnn*0_0101_0010		2
		[ER*m*]		*	*				1001_*nnn*0_*mmm*0_0010		2
		Disp16[ER*m*]		*	*				1010_*nnn*0_*mmm*0_1000	DDDD_DDDD_DDDD_DDDD	3
		Disp6[BP]		*	*				1011_*nnn*0_00DD_DDDD		3
		Disp6[FP]		*	*				1011_*nnn*0_01DD_DDDD		3
		Dadr		*	*				1001_*nnn*0_0001_0010	DDDD_DDDD_DDDD_DDDD	2
	R*n*	[EA]		*	*				1001_*nnnn*_0011_0000		1
		[EA+]		*	*				1001_*nnnn*_0101_0000		1
		[ER*m*]		*	*				1001_*nnnn*_*mmm*0_0000		1
		Disp16[ER*m*]		*	*				1001_*nnnn*_*mmm*0_1000	DDDD_DDDD_DDDD_DDDD	2
		Disp6[BP]		*	*				1101_*nnnn*_00DD_DDDD		2
		Disp6[FP]		*	*				1101_*nnnn*_01DD_DDDD		2
		Dadr		*	*				1001_*nnnn*_0001_0000	DDDD_DDDD_DDDD_DDDD	2
	XR*n*	[EA]		*	*				1001_*nn*00_0011_0100		4
		[EA+]		*	*				1001_*nn*00_0101_0100		4
	QR*n*	[EA]		*	*				1001_*n*000_0011_0110		8
		[EA+]		*	*				1001_*n*000_0101_0110		8

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
ST	ER*n*	[EA]							1001_*nnn*0_0011_0011		2
		[EA+]							1001_*nnn*0_0101_0011		2
		[ER*m*]							1001_*nnn*0_*mmm*0_0011		2
		Disp16[ER*m*]							1010_*nnn*0_*mmm*0_1001	DDDD_DDDD_DDDD_DDDD	3
		Disp6[BP]							1011_*nnn*0_10DD_DDDD		3
		Disp6[FP]							1011_*nnn*0_11DD_DDDD		3
		Dadr							1001_*nnn*0_0001_0011	DDDD_DDDD_DDDD_DDDD	2
	R*n*	[EA]							1001_*nnnn*_0011_0001		1
		[EA+]							1001_*nnnn*_0101_0001		1
		[ER*m*]							1001_*nnnn*_*mmm*0_0001		1
		Disp16[ER*m*]							1001_*nnnn*_*mmm*0_1001	DDDD_DDDD_DDDD_DDDD	2
		Disp6[BP]							1101_*nnnn*_10DD_DDDD		2
		Disp6[FP]							1101_*nnnn*_11DD_DDDD		2
		Dadr							1001_*nnnn*_0001_0001	DDDD_DDDD_DDDD_DDDD	2
	XR*n*	[EA]							1001_*nn*00_0011_0101		4
		[EA+]							1001_*nn*00_0101_0101		4
	QR*n*	[EA]							1001_*n*000_0011_0111		8
		[EA+]							1001_*n*000_0101_0111		8

Control Register Access Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
ADD	SP	*#signed8*							1110_0001_iiii_iiii		2
MOV	ECSR	R*m*							1010_0000_*mmmm*_1111		2
	ELR	ER*m*							1010_*mmm*0_0000_1101		3
	EPSW	R*m*							1010_0000_*mmmm*_1100		2
	ER*n*	ELR							1010_*nnn*0_0000_0101		3
		SP							1010_*nnn*0_0001_1010		2
	PSW	R*m*	*	*	*	*	*	*	1010_0000_*mmmm*_1011		1
		#unsigned8	*	*	*	*	*	*	1110_1001_iiii_iiii		1
	R*n*	ECSR							1010_*nnnn*_0000_0111		2
		EPSW							1010_*nnnn*_0000_0100		2
		PSW							1010_*nnnn*_0000_0011		1
	SP	ER*m*							1010_0001_*mmm*0_1010		1

PUSH/POP Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
PUSH	ER*n*								1111_*nnn*0_0101_1110		2
	QR*n*								1111_*n*000_0111_1110		8
	R*n*								1111_*nnnn*_0100_1110		2
	XR*n*								1111_*nn*00_0110_1110		4
	register_list								1111_lepa_1100_1110		2-12
POP	ER*n*								1111_*nnn*0_0001_1110		2
	QR*n*								1111_*n*000_0011_1110		8
	R*n*								1111_*nnnn*_0000_1110		2
	XR*n*								1111_*nn*00_0010_1110		4
	register_list		*	*	*	*	*	*	1111_lepa_1000_1110		2-15

Coprocessor Data Transfer Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
MOV	CR*n*	R*m*							1010_*nnnn*_*mmmm*_1110		1
	CER*n*	[EA]							1111_*nnn*0_0010_1101		2
		[EA+]							1111_*nnn*0_0011_1101		2
	CR*n*	[EA]							1111_*nnnn*_0000_1101		1
		[EA+]							1111_*nnnn*_0001_1101		1
	CXR*n*	[EA]							1111_*nn*00_0100_1101		4
		[EA+]							1111_*nn*00_0101_1101		4
	CQR*n*	[EA]							1111_*n*000_0110_1101		8
		[EA+]							1111_*n*000_0111_1101		8
	R*n*	CR*m*							1010_*nnnn*_*mmmm*_0110		1
	[EA]	CER*m*							1111_*mmm*0_1010_1101		2
	[EA+]	CER*m*							1111_*mmm*0_1011_1101		2
	[EA]	CR*m*							1111_*mmmm*_1000_1101		1
	[EA+]	CR*m*							1111_*mmmm*_1001_1101		1
	[EA]	CXR*m*							1111_*mm*00_1100_1101		4
	[EA+]	CXR*m*							1111_*mm*00_1101_1101		4
	[EA]	CQR*m*							1111_*m*000_1110_1101		8
	[EA+]	CQR*m*							1111_*m*000_1111_1101		8

EA Register Data Transfer Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
LEA	[ER*m*]								1111_0000_*mmm*0_1010		1
	Disp16[ER*m*]								1111_0000_*mmm*0_1011	DDDD_DDDD_DDDD_DDDD	2
	Dadr								1111_0000_0000_1100	DDDD_DDDD_DDDD_DDDD	2

ALU Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
DAA	R*n*		*	*	*			*	1000_*nnnn*_0001_1111		1
DAS	R*n*		*	*	*			*	1000_*nnnn*_0011_1111		1
NEG	R*n*		*	*	*	*		*	1000_*nnnn*_0101_1111		1

Bit Access Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
SB	R*n.bit_offset*			*					1010_*nnnn*_0bbb_0000		1
	Dbitadr			*					1010_0000_1bbb_0000	DDDD_DDDD_DDDD_DDDD	2
RB	R*n. bit_offset*			*					1010_*nnnn*_0bbb_0010		1
	Dbitadr			*					1010_0000_1bbb_0010	DDDD_DDDD_DDDD_DDDD	2
TB	R*n. bit_offset*			*					1010_*nnnn*_0bbb_0001		1
	Dbitadr			*					1010_0000_1bbb_0001	DDDD_DDDD_DDDD_DDDD	2

PSW Access Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
EI							*		1110_1101_0000_1000		1
DI							*		1110_1011_1111_0111		3
SC			*						1110_1101_1000_0000		1
RC			*						1110_1011_0111_1111		1
CPLC			*						1111_1110_1100_1111		1

Conditional Relative Branch Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
BGE	*Radr*								1100_0000_rrrr_rrrr		1/3
BLT									1100_0001_rrrr_rrrr		1/3
BGT									1100_0010_rrrr_rrrr		1/3
BLE									1100_0011_rrrr_rrrr		1/3
BGES									1100_0100_rrrr_rrrr		1/3
BLTS									1100_0101_rrrr_rrrr		1/3
BGTS									1100_0110_rrrr_rrrr		1/3
BLES									1100_0111_rrrr_rrrr		1/3
BNE									1100_1000_rrrr_rrrr		1/3
BEQ									1100_1001_rrrr_rrrr		1/3
BNV									1100_1010_rrrr_rrrr		1/3
BOV									1100_1011_rrrr_rrrr		1/3
BPS									1100_1100_rrrr_rrrr		1/3
BNS									1100_1101_rrrr_rrrr		1/3
BAL									1100_1110_rrrr_rrrr		3

Sign Extension Instruction

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
EXTBW	ER*n*			*	*				1000_*nnn*1_*nnn*0_1111		1

Software Interrupt Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
SWI	*#snum*						*		1110_0101_00ii_iiii		3
BRK									1111_1111_1111_1111		7

Branch Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
B	*Cadr*								1111_gggg_0000_0000	cccc_cccc_cccc_cccc	2
	ER*n*								1111_0000_*nnn*0_0010		2
BL	*Cadr*								1111_gggg_0000_0001	cccc_cccc_cccc_cccc	2
	ER*n*								1111_0000_*nnn*0_0011		2

Multiplication and Division Instructions

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
MUL	ER*n*	R*m*		*					1111_*nnn*0_*mmmm*_0100		8
DIV	ER*n*	R*m*	*	*					1111_*nnn*0_*mmmm*_1001		16

Miscellaneous

Mnemonic	First operand	Second operand	Flag changes						Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word	
INC	[EA]			*	*	*		*	1111_1110_0010_1111		2
DEC	[EA]			*	*	*		*	1111_1110_0011_1111		2
RT									1111_1110_0001_1111		2
RTI			*	*	*	*	*	*	1111_1110_0000_1111		2
NOP									1111_1110_1000_1111		1

Revision History

REVISION HISTORY

Document No.	Date	Page		Description
		Previous Edition	Current Edition	
FEUZ0317A0-U8-INST-01	Jul 25, 2006	—	—	1st edition.
FEUZ0317A0-U8-INST-02	Oct 17,2008	Cover	Cover	Updated for OKI SEMICONDUCTOR logo, version, and date.
		Notice	Notice	Updated for OKI SEMICONDUCTOR.
		Revision History	Revision History	Update for OKI SEMICONDUCTOR, version, and date.

NOTICE

1. The information contained herein can change without notice owing to product and/or technical improvements. Before using the product, please make sure that the information being referred to is up-to-date.

2. The outline of action and examples for application circuits described herein have been chosen as an explanation for the standard action and performance of the product. When planning to use the product, please ensure that the external conditions are reflected in the actual circuit, assembly, and program designs.

3. When designing your product, please use our product below the specified maximum ratings and within the specified operating ranges including, but not limited to, operating voltage, power dissipation, and operating temperature.

4. OKI SEMICONDUCTOR assumes no responsibility or liability whatsoever for any failure or unusual or unexpected operation resulting from misuse, neglect, improper installation, repair, alteration or accident, improper handling, or unusual physical or electrical stress including, but not limited to, exposure to parameters beyond the specified maximum ratings or operation outside the specified operating range.

5. Neither indemnity against nor license of a third party's industrial and intellectual property right, etc. is granted by us in connection with the use of the product and/or the information and drawings contained herein. No responsibility is assumed by us for any infringement of a third party's right which may result from the use thereof.

6. The products listed in this document are intended for use in general electronics equipment for commercial applications (e.g., office automation, communication equipment, measurement equipment, consumer electronics, etc.). These products are not, unless specifically authorized by OKI SEMICONDUCTOR, authorized for use in any system or application that requires special or enhanced quality and reliability characteristics nor in any system or application where the failure of such system or application may result in the loss or damage of property, or death or injury to humans.
   Such applications include, but are not limited to, traffic and automotive equipment, safety devices, aerospace equipment, nuclear power control, medical equipment, and life-support systems.

7. Certain products in this document may need government approval before they can be exported to particular countries. The purchaser assumes the responsibility of determining the legality of export of these products and will take appropriate and necessary steps at their own expense for these.

8. No part of the contents contained herein may be reprinted or reproduced without our prior permission.

Copyright 2008 OKI SEMICONDUCTOR CO.,LTD.