

# nX-U16/100 Core Instruction Manual

---

CMOS 16-bit microcontroller

Issue Date: Jan. 2015

## NOTES

No copying or reproduction of this document, in part or in whole, is permitted without the consent of LAPIS Semiconductor Co., Ltd. The content specified herein is subject to change for improvement without notice.

Examples of application circuits, circuit constants and any other information contained herein illustrate the standard usage and operations of the Products. The peripheral conditions must be taken into account when designing circuits for mass production.

Great care was taken in ensuring the accuracy of the information specified in this document. However, should you incur any damage arising from any inaccuracy or misprint of such information, LAPIS Semiconductor shall bear no responsibility for such damage.

The technical information specified herein is intended only to show the typical functions of and examples of application circuits for the Products. LAPIS Semiconductor does not grant you, explicitly or implicitly, any license to use or exercise intellectual property or other rights held by LAPIS Semiconductor and other parties. LAPIS Semiconductor shall bear no responsibility whatsoever for any dispute arising from the use of such technical information.

The Products specified in this document are intended to be used with general-use electronic equipment or devices (such as audio visual equipment, office-automation equipment, communication devices, electronic appliances and amusement devices).

The Products specified in this document are not designed to be radiation tolerant.

While LAPIS Semiconductor always makes efforts to enhance the quality and reliability of its Products, a Product may fail or malfunction for a variety of reasons.

Please be sure to implement in your equipment using the Products safety measures to guard against the possibility of physical injury, fire or any other damage caused in the event of the failure of any Product, such as derating, redundancy, fire control and fail-safe designs. LAPIS Semiconductor shall bear no responsibility whatsoever for your use of any Product outside of the prescribed scope or not in accordance with the instruction manual.

The Products are not designed or manufactured to be used with any equipment, device or system which requires an extremely high level of reliability the failure or malfunction of which may result in a direct threat to human life or create a risk of human injury (such as a medical instrument, transportation equipment, aerospace machinery, nuclear-reactor controller, fuel-controller or other safety device). LAPIS Semiconductor shall bear no responsibility in any way for use of any of the Products for the above special purposes. If a Product is intended to be used for any such special purpose, please contact a ROHM sales representative before purchasing.

If you intend to export or ship overseas any Product or technology specified herein that may be controlled under the Foreign Exchange and the Foreign Trade Law, you will be required to obtain a license or permit under the Law.

Copyright 2011 - 2014 LAPIS Semiconductor Co., Ltd.

# Contents

<b>1. Architecture</b>	<b>1-1</b>
<b>1.1 Overview</b>	<b>1-1</b>
1.1.1 Features	1-1
<b>1.2 CPU Resources and Programming Model</b>	<b>1-2</b>
1.2.1 Registers	1-4
1.2.1.1 General Registers	1-5
1.2.1.2 Base and Frame Pointers	1-5
1.2.2 Control Registers	1-6
1.2.2.1 Program Status Word (PSW)	1-6
1.2.2.2 Program Counter (PC)	1-8
1.2.2.3 Code Segment Register (CSR)	1-8
1.2.2.4 Link Registers (LR, ELR1, ELR2, and ELR3)	1-9
1.2.2.5 CSR Backup Registers (LCSR, ECSR1, ECSR2, and ECSR3)	1-10
1.2.2.6 PSW Backup Registers (EPSW1, EPSW2, and EPSW3)	1-11
1.2.2.7 Stack Pointer (SP)	1-11
1.2.2.8 EA Register (EA)	1-12
1.2.2.9 Address Register (AR)	1-12
1.2.2.10 Data Segment Register (DSR)	1-13
<b>1.3 Memory Spaces</b>	<b>1-14</b>
1.3.1 Program/Code Memory Space	1-14
1.3.2 Vector Table	1-15
1.3.2.1 Reset Vectors	1-15
1.3.2.2 Interrupt Vectors	1-16
1.3.2.3 Writing Vector Table	1-17
1.3.3 Program/Code Memory Space	1-18
1.3.4 DSR Prefix Instructions	1-18
1.3.5 Data Memory Space	1-19
1.3.5.1 Data Types	1-20
1.3.5.2 Address Assignment	1-21
1.3.5.3 Word Boundaries	1-21
1.3.5.4 ROM Window	1-22
1.3.6 Hardware Memory Models	1-22
1.3.7 Interrupt Operation	1-24
1.3.7.1 Interrupt Acceptance	1-24
1.3.7.2 Non-maskable Interrupts (NMI)	1-25
1.3.7.3 Maskable Interrupts (MI)	1-26
1.3.7.4 Software Interrupts (SWI)	1-27

---

<b>1.4 Exception Levels and Backup Registers .....</b>	<b>1-28</b>
<b>1.5 Notes about Non-maskable interrupts .....</b>	<b>1-34</b>
<b>1.6 Interrupt Blocking .....</b>	<b>1-35</b>
<b>1.7 Stack Modifications.....</b>	<b>1-36</b>
 <b>2. Addressing Types .....</b>	 <b>2-1</b>

---

<b>2.1 Addressing Types .....</b>	<b>2-1</b>
<b>2.2 Register Addressing .....</b>	<b>2-2</b>
<b>2.3 Memory Addressing.....</b>	<b>2-3</b>
2.3.1 Register Indirect Addressing .....	2-4
2.3.2 Direct Addressing .....	2-7
<b>2.4 Immediate Addressing.....</b>	<b>2-8</b>
<b>2.5 Program/Code Memory Addressing .....</b>	<b>2-9</b>
 <b>3. Instruction Descriptions .....</b>	 <b>3-1</b>

---

<b>3.1 Overview .....</b>	<b>3-1</b>
<b>3.2 Instructions by Functional Group.....</b>	<b>3-2</b>
Arithmetic Instructions .....	3-2
Shift Instructions .....	3-2
Load/Store Instructions .....	3-3
Load/Store Instructions (cont.) .....	3-4
Load/Store Instructions (cont.) .....	3-5
Control Register Access Instructions .....	3-6
PUSH/POP Instructions.....	3-6
Coprocessor Data Transfer Instructions.....	3-7
Coprocessor Data Transfer Instructions (continued from previous page) .....	3-8
Bit Access Instructions .....	3-9
PSW Access Instructions .....	3-9
Conditional Relative Branch Instructions.....	3-9
Sign Extension Instruction.....	3-9
Software Interrupt Instructions .....	3-10
Branch Instructions.....	3-10
Multiplication and Division Instructions.....	3-10
Miscellaneous.....	3-10
 <b>3.3 Instruction Execution Times .....</b>	 <b>3-11</b>

---

<b>3.4 Instruction Descriptions .....</b>	<b>3-22</b>
ADD ERn , ERm.....	3-23
ADD ERn , #imm7.....	3-24
ADD Rn , obj.....	3-25
ADD SP , #signed8.....	3-26
ADDC Rn , obj.....	3-27
AND Rn , obj.....	3-28
B Cadr.....	3-29
B ERn.....	3-30
Bcond Radr.....	3-31
BL Cadr.....	3-33
BL ERn.....	3-34
BRK.....	3-35
CMP ERn , ERm.....	3-36
CMP Rn , obj.....	3-37
CMPC Rn , obj.....	3-38
CPLC.....	3-39
DAA Rn.....	3-40
DAS Rn.....	3-41
DEC [EA].....	3-42
DI.....	3-43
DIV ERn , Rm.....	3-44
EI.....	3-45
EXTBW ERn.....	3-46
INC [EA].....	3-47
L ERn, obj.....	3-48
L QRn,obj.....	3-50
L Rn, obj.....	3-51
L XRn,obj.....	3-53
LEA obj.....	3-54
MOV CERN , obj.....	3-55
MOV CQRn , obj.....	3-56
MOV CRn , obj.....	3-57
MOV CRn , Rm.....	3-58
MOV CXRn , obj.....	3-59
MOV ECSR , Rm.....	3-60
MOV ELR , ERm.....	3-61
MOV EPSW , Rm.....	3-62
MOV ERn , ELR.....	3-63
MOV ERn , ERm.....	3-64
MOV ERn , #imm7.....	3-65
MOV ERn , SP.....	3-66

---

MOV <i>obj</i> , CERM.....	3-67
MOV <i>obj</i> , CQRm.....	3-68
MOV <i>obj</i> , CRM.....	3-69
MOV <i>obj</i> , CXRM.....	3-70
MOV PSW, <i>obj</i> .....	3-71
MOV <i>Rn</i> , CRM.....	3-72
MOV <i>Rn</i> , ECSR.....	3-73
MOV <i>Rn</i> , EPSW.....	3-74
MOV <i>Rn</i> , PSW.....	3-75
MOV <i>Rn</i> , <i>obj</i> .....	3-76
MOV SP, ERm.....	3-77
MUL ERn,Rm.....	3-78
NEG <i>Rn</i> .....	3-79
NOP.....	3-80
OR <i>Rn</i> , <i>obj</i> .....	3-81
POP register list.....	3-82
POP <i>obj</i> .....	3-84
PUSH register list.....	3-85
PUSH <i>obj</i> .....	3-87
RB Dbitadr.....	3-88
RB <i>Rn</i> . <i>bit_offset</i> .....	3-89
RC.....	3-90
RT.....	3-91
RTI.....	3-92
SB Dbitadr.....	3-93
SB <i>Rn</i> . <i>bit_offset</i> .....	3-94
SC.....	3-95
SLL <i>Rn</i> , <i>obj</i> .....	3-96
SLLC <i>Rn</i> , <i>obj</i> .....	3-97
SRA <i>Rn</i> , <i>obj</i> .....	3-98
SRL <i>Rn</i> , <i>obj</i> .....	3-99
SRLC <i>Rn</i> , <i>obj</i> .....	3-100
ST ERn, <i>obj</i> .....	3-101
ST QRn, <i>obj</i> .....	3-103
ST <i>Rn</i> , <i>obj</i> .....	3-104
ST XRn, <i>obj</i> .....	3-106
SUB <i>Rn</i> , Rm.....	3-107
SUBC <i>Rn</i> , Rm.....	3-108
SWI # <i>snum</i> .....	3-109
TB Dbitadr.....	3-110
TB <i>Rn</i> . <i>bit_offset</i> .....	3-111
XOR <i>Rn</i> , <i>obj</i> .....	3-112

---

---

**4. Appendix****4-1**

---

Arithmetic Instructions .....	4-1
Shift Instructions .....	4-1
Load/Store Instructions .....	4-2
Control Register Access Instructions .....	4-3
PUSH/POP Instructions.....	4-3
Coprocessor Data Transfer Instructions.....	4-4
EA Register Data Transfer Instructions.....	4-4
ALU Instructions .....	4-4
Bit Access Instructions .....	4-5
PSW Access Instructions .....	4-5
Conditional Relative Branch Instructions.....	4-5
Sign Extension Instruction.....	4-6
Software Interrupt Instructions .....	4-6
Branch Instructions.....	4-6
Multiplication and Division Instructions.....	4-6
Miscellaneous.....	4-6

# **1. Architecture**

---





## 1.1 Overview

The nX-U16/100 CPU is capable of executing instructions efficiently on a one-instruction-per-clock-pulse basis through parallel processing by the 3-stage pipelined architecture. The nX-U16/100 is categorized as follows from the difference in the number of execution cycles when conditional branch is concluded.

CPU core type	Execution cycles of the conditional branch instruction.
A33 , A34	1/3    ••The branch condition is (not met / met).
A35	1/2    ••The branch condition is (not met / met).

The CPU core type is different for each product. For details, please refer to a user's manual. When the CPU type is not described in the user's manual, the A34 core is equipped.

### 1.1.1 Features

The U16 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  
 Non-maskable interrupts  
 Maskable interrupts  
 Software interrupts

## 1.2 CPU Resources and Programming Model

The U16 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 U16 memory spaces.

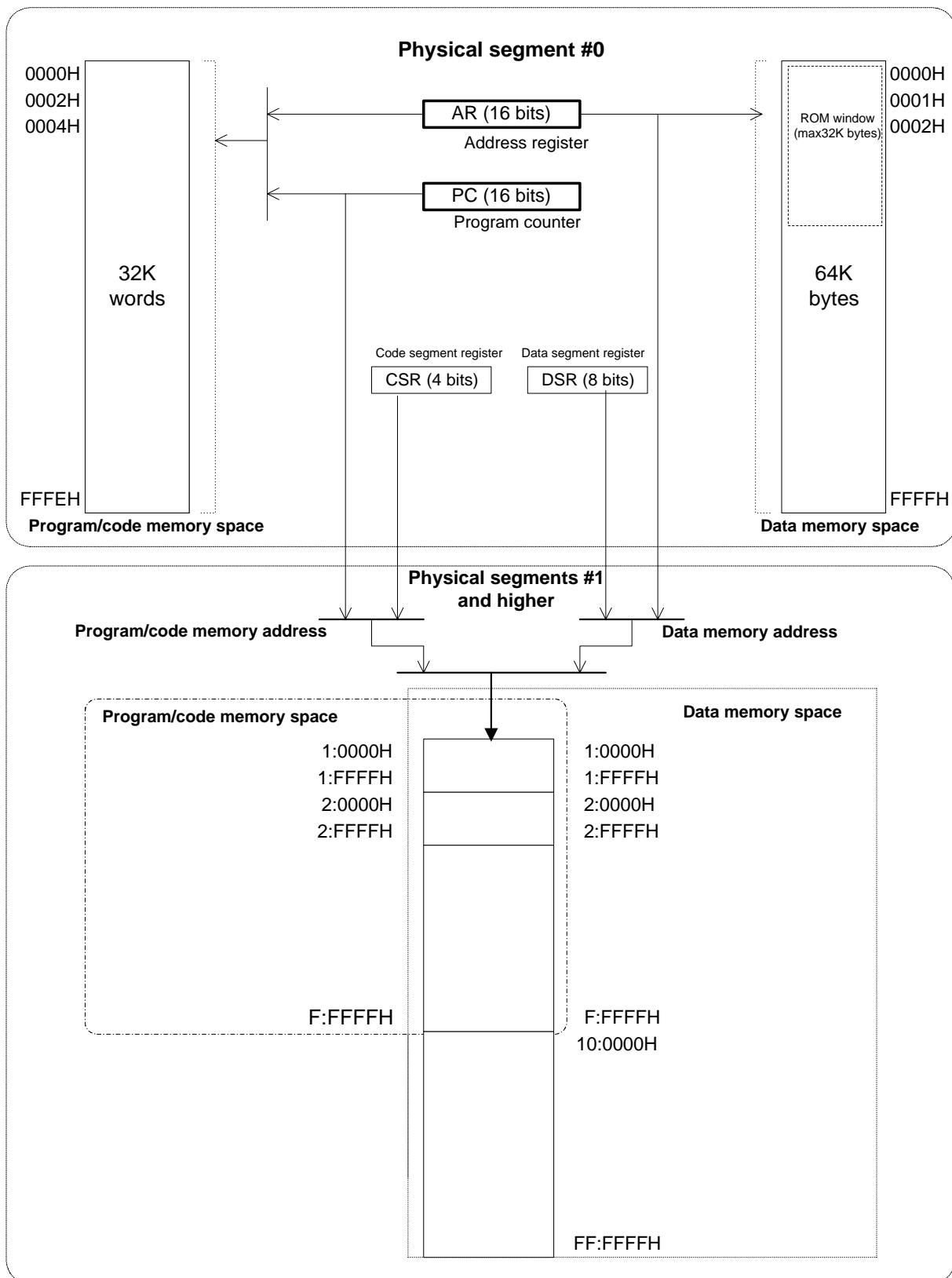


Figure 1.1. U16 Memory Spaces

## 1.2.1 Registers

General registers lie at the center of U16 hardware operation. Also shown in Figure 1.2 are the control registers.

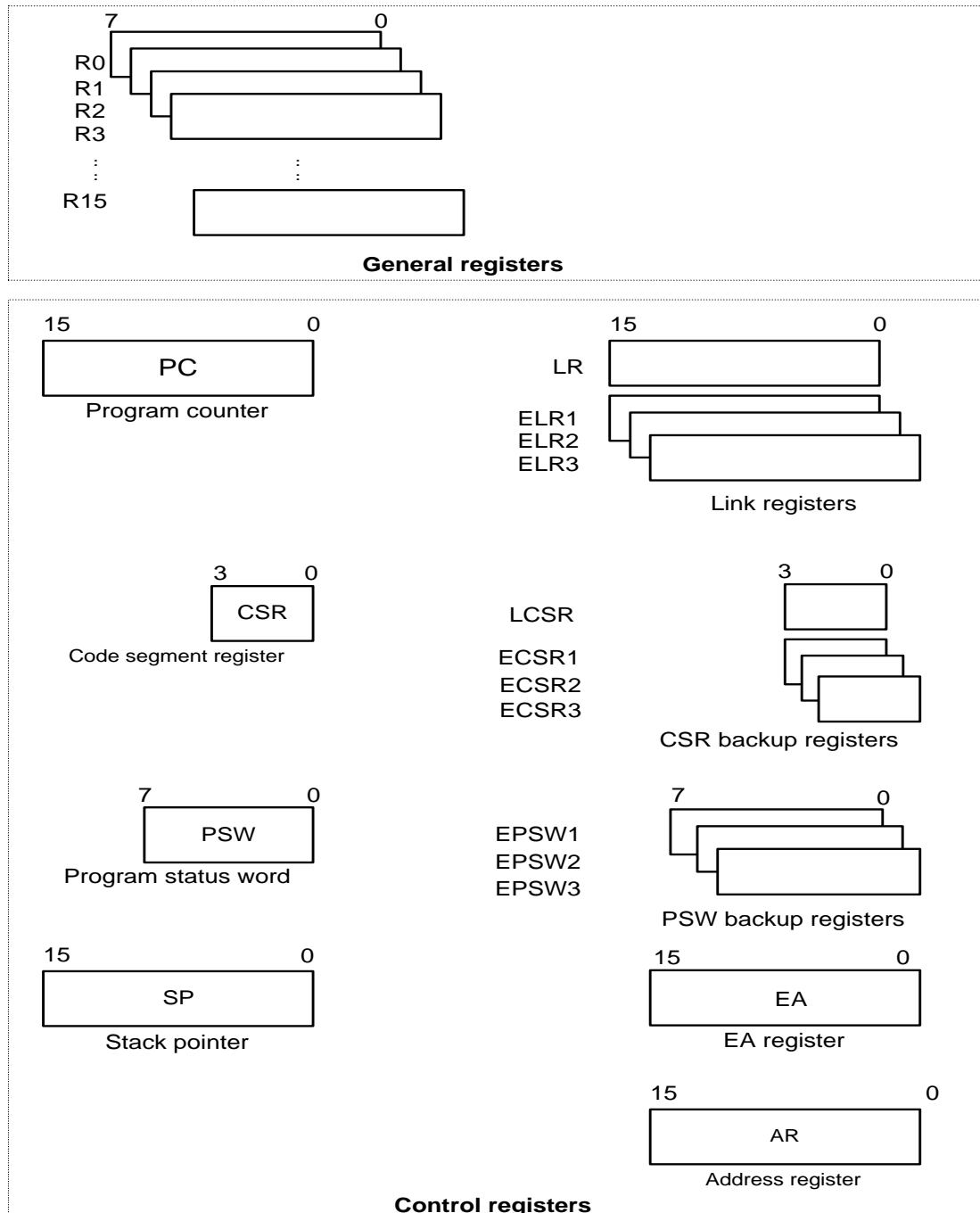


Figure 1.2. Register Set

### 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.

				7	0
QR0	XR0	ER0	R0		
			R1		
		ER2	R2		
			R3		
	XR4	ER4	R4		
			R5		
		ER6	R6		
			R7		
QR8	XR8	ER8	R8		
			R9		
		ER10	R10		
			R11		
	XR12	ER12	R12	BP (Lower byte)	Base pointer
			R13	BP (Upper byte)	
		ER14	R14	FP (Lower byte)	Frame pointer
			R15	FP (Upper byte)	

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.”

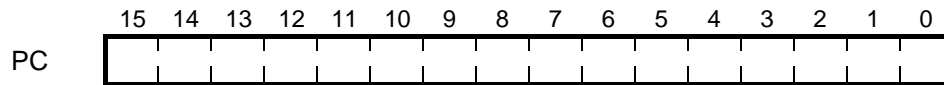
The U16 hardware accepts an interrupt request only if its interrupt priority is the same or greater than the current exception level (ELEVEL) setting.

#### **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)



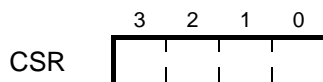
This 16-bit register holds the offset portion of the address of the next instruction to be executed. The hardware automatically increments it immediately after fetching an instruction from program/code memory, creating the cycle necessary for sequential execution. Branch and other instructions, however,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



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

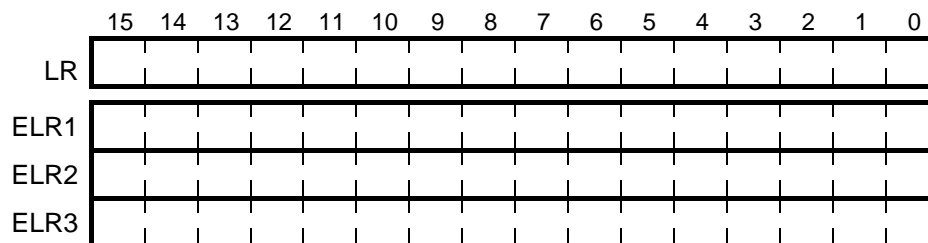
exception level (ELEVEL) setting from program status word (PSW).

- 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.”

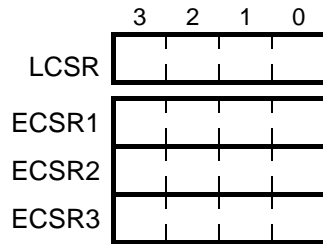
The LR register holds the offset portion of the return address for a subroutine called with a BL instruction. The RT instruction at the end of the subroutine loads the LR contents back into the program counter (PC).

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).

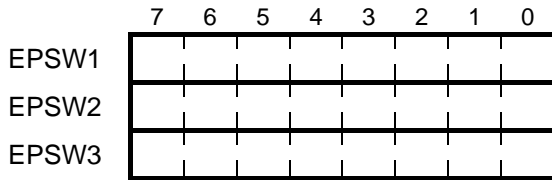
Note that the program has a choice of two instructions for returning from a subroutine to its caller: RT or POP. For further details, see Section 1.4 “Exception Levels and Backup Registers.”

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)



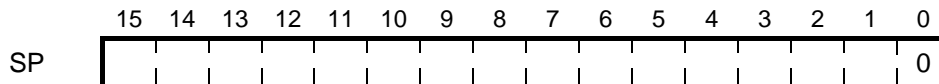
These three 8-bit registers are for saving the contents of the program status word (PSW) during interrupt handlers.

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.

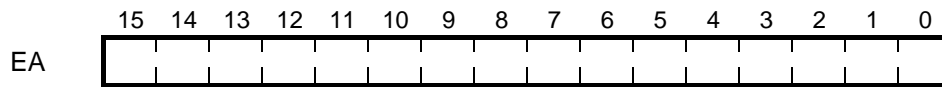
Stack operations are always word sized. One saving word-sized data to the stack subtracts 2 from this register and then copies the data to that new address. Restoring data copies a word from the stack to the specified destination and then adds 2 to this register.

Bit 0 of this register is hardwired to 0.

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)

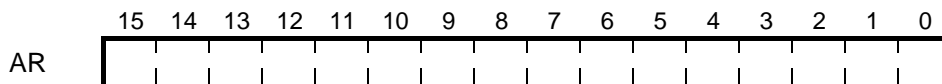


This 16-bit register holds an address for use by instructions that access data memory indirectly via this register.

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).

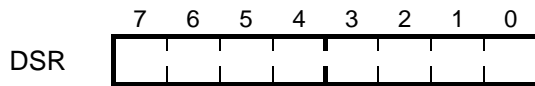
This register is accessible from programs with the LEA instruction for loading it and with the stack manipulation instructions PUSH and POP.

### 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 U16 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.



### 1.3.2 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.

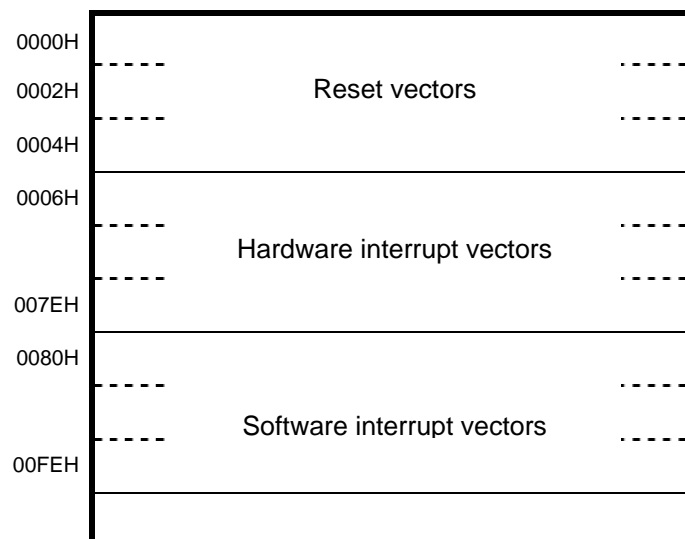


Figure 1.4. Vector Table

#### 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.

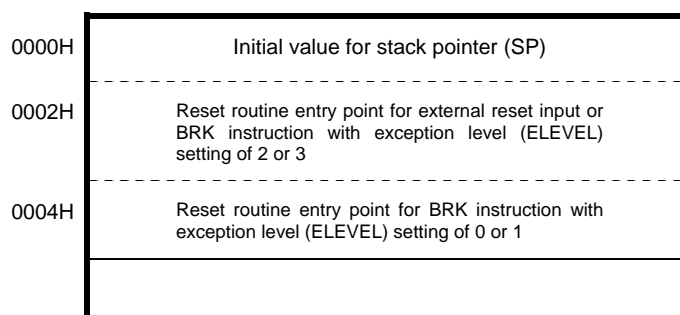


Figure 1.5. Reset Vectors



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 non-maskable interrupt requests, NMICE and NMI, plus room for up to 59 maskable ones.

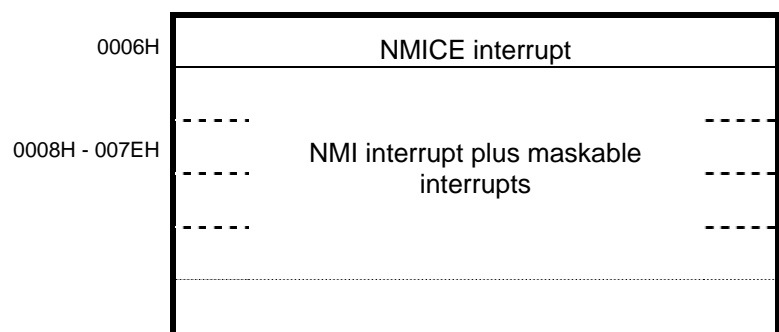


Figure 1.6. Hardware Interrupt Vectors

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.

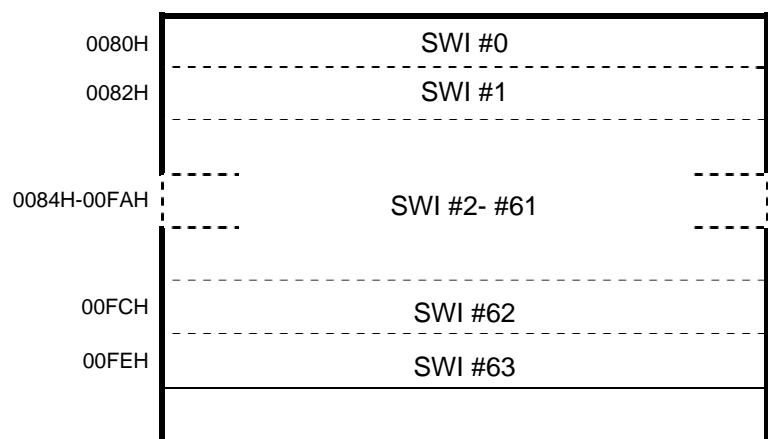


Figure 1.7. Software Interrupt Vectors

### 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       ; Non-maskable 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 U16 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 <code>iiii_iiii</code> .
1001_0000_ddd_1111	Load DSR with the contents of the general register <code>Rd</code> .
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 U16 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.

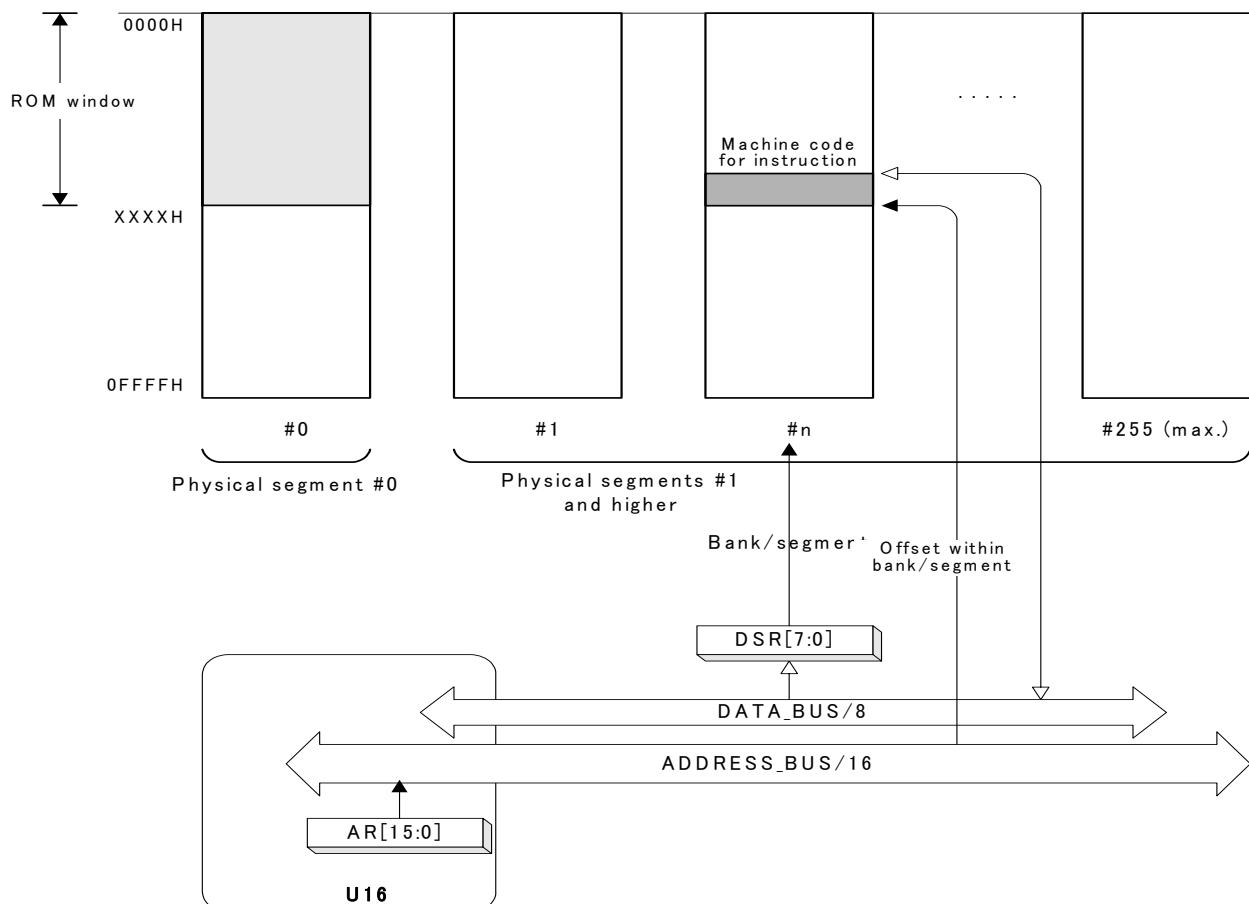
### 1.3.5 Data Memory Space

The U16'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 U16 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 U16 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 U16 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.

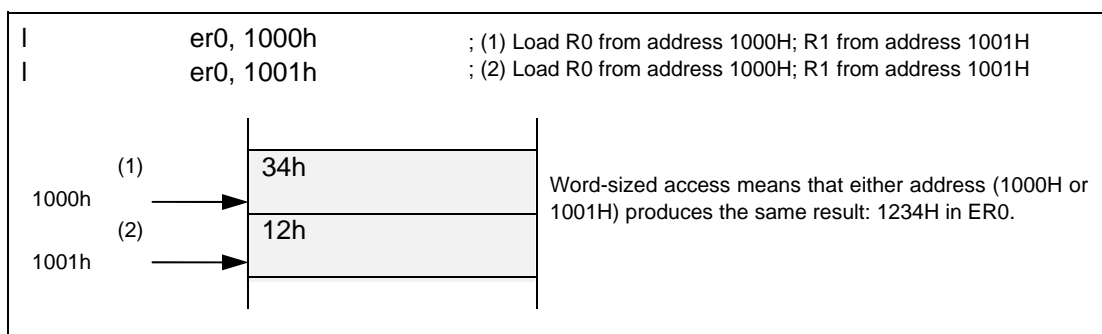


Figure 1.8. Word Boundaries in Memory

### 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 U16 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 U16 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 U16 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)* <sup>1</sup>	3
Non-maskable interrupt (NMI)	2
Software interrupt (SWI)	1
Maskable interrupt (MI)	1

\*<sup>1</sup> 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 U16 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 non-maskable (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 Non-maskable Interrupts (NMI)

User application programs have no means of masking non-maskable 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 non-maskable 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 U16 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 non-maskable 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 non-maskable 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 non-maskable 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 U16 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: Any interrupts are not being processed

The running state of exception level (ELEVEL) is 0, and the backup registers used are LR and LCSR. How procedures begin and end depends solely on whether subroutines are nested.

#### A-1: When a subroutine is not called by the program in a subroutine.

- Processing immediately after the start of subroutine

No specific notes.

- Processing at the end of subroutine

Restore PC from LR with an RT instruction.

Example of description: State A-1

Sub_A-1:		; beginning of subroutine.
:		
:		
RT		; Restore PC from LR.
		; Terminate subroutine

#### A-2: When a subroutine is called by the program in a subroutine.

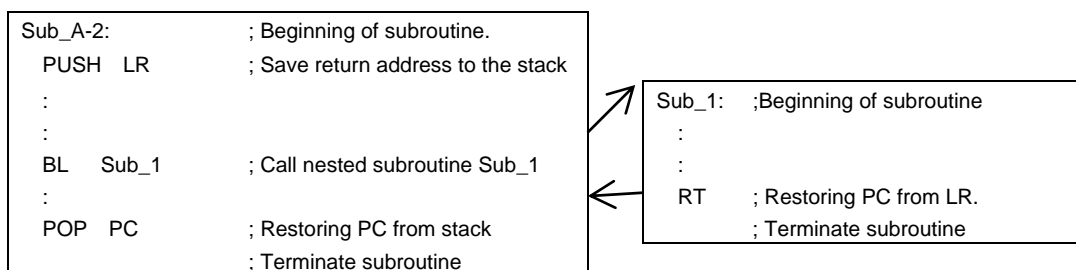
- Processing immediately after the start of subroutine

Use PUSH LR instruction to save return address to the stack.

- Processing at the end of subroutine

Use POP PC instruction instead of RT to return from subroutine.

Example of description: State A-2



**B: Maskable interrupt is being processed**

The running state of exception level (ELEVEL) is 1, and the backup registers used are ELR1, ECSR1, and EPSW1. How procedures begin and end depends on whether multiple interrupts are enabled or disabled.

**B-1: When a subroutine is not called by the program in executing an interrupt routine.****B-1-1: When multiple interrupts are disabled.**

- Processing immediately after the start of interrupt routine execution

No specific notes.

- Processing at the end of interrupt routine execution

Restore PC from ELR1 and PSW from EPSW1 with an RTI instruction.

Example of description: State B-1-1

Intrpt_B-1-1:	; Beginning of an interrupt routine.
:	
:	
RTI	; Return PC from ELR
	; Return PSW form EPSW
	; End

**B-1-2: When multiple interrupts are enabled.**

- Processing immediately after the start of interrupt routine execution

Specify “PUSH ELR, EPSW” to save the interrupt return address and the PSW status in the stack.

- Processing at the end of interrupt routine execution

Specify “POP PSW, PC” instead of the RTI instruction to return the contents of the stack to PC and PSW.

Example of description: State B-1-2

Intrpt_B-1-2:	; Start
PUSH ELR, EPSW	; Save ELR and EPSW at the beginning
:	
EI	; Enable interrupt
:	
:	
POP PSW,PC	; Return PC from the stack
	; Return PSW from the stack
	; End



**B-2: When a subroutine is called by the program in executing an interrupt routine.****B-2-1: When multiple interrupts are disabled.**

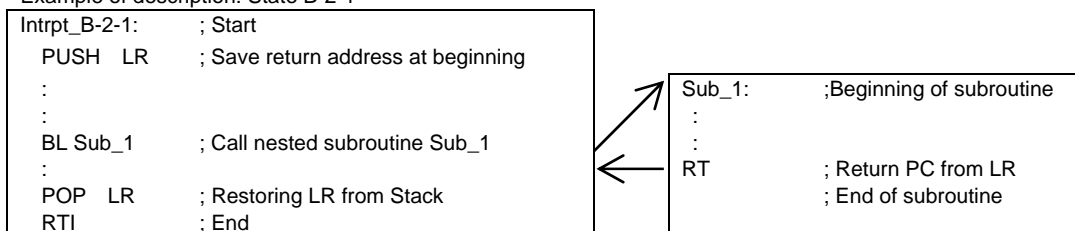
- Processing immediately after the start of interrupt routine execution

Specify the “PUSH LR” instruction to save the subroutine return address in the stack.

- Processing at the end of interrupt routine execution

Specify “POP LR” immediately before the RTI instruction to return from the interrupt processing after returning the subroutine return address to LR.

Example of description: State B-2-1

**B-2-2: When multiple interrupts are enabled.**

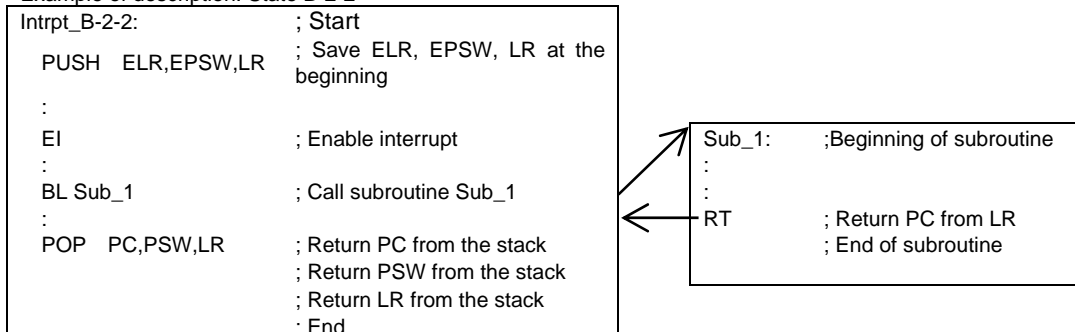
- Processing immediately after the start of interrupt routine execution

Specify “PUSH ELR, EPSW, LR” to save the interrupt return address, the subroutine return address, and the EPSW status in the stack.

- Processing at the end of interrupt routine execution

Specify “POP PC, PSW, LR” instead of the RTI instruction to return the saved data of the interrupt return address to PC, the saved data of EPSW to PSW, and the saved data of LR to LR.

Example of description: State B-2-2



**C: Non-maskable interrupt is being processed**

The running state of exception level (ELEVEL) is 2, and the backup registers used are ELR2, ECSR2, and EPSW2. How procedures begin and end depends on whether a subroutine is called or not.

**C-1: When a subroutine is not called by the program in executing an interrupt routine.**

- Processing immediately after the start of interrupt routine execution

Specify “PUSH ELR, EPSW” to save the interrupt return address, the subroutine return address, and the EPSW status in the stack.

- Processing at the end of interrupt routine execution

Specify “POP PSW, PC” instead of the RTI instruction to return the saved data of the interrupt return address to PC, the saved data of EPSW to PSW.

Example of description: State C-1

```
Intrpt_C-1:           ; Start
  PUSH  ELR,EPSW       ; Save ELR and EPSW at the beginning
  :
  :
  POP   PSW,PC         ; Return PC from the stack
                      ; Return PSW from the stack
                      ; End
```

**C-2: When a subroutine is called by the program in executing an interrupt routine.**

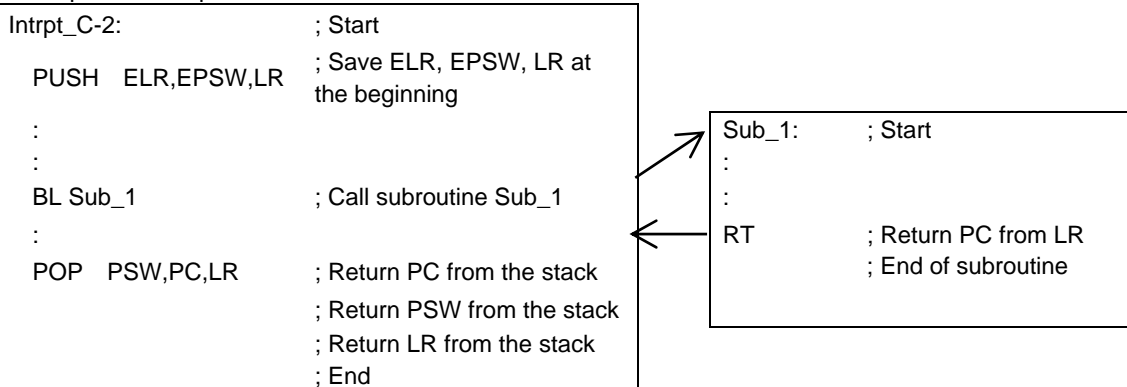
- Processing immediately after the start of interrupt routine execution

Specify “PUSH ELR, EPSW, LR” to save the interrupt return address, the subroutine return address, and the EPSW status in the stack.

- Processing at the end of interrupt routine execution

Specify “POP PSW, PC, LR” instead of the RTI instruction to return the saved data of the interrupt return address to PC, the saved data of EPSW to PSW, and the saved data of LR to LR.

Example of description: State C-2



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

## 1.5 Notes about Non-maskable interrupts

This clause describes the notes about the non-maskable interrupt at the time of the program development by C.

Since a non-maskable interrupt request cannot be disabled, it needs multiple interrupt processing of a non-maskable interrupt in an application program. If the measure against a multiple interrupt is not implemented, it may become impossible for a program to return from an interrupt routine.

Therefore, when target chips use non-maskable interrupts, please set category as two by an INTERRUPT pragma.

```
static void int_WDTINT(void);  
#pragma interrupt int_WDTINT 0x08 2  
static void int_WDTINT(void)  
{  
    :  
    :  
}
```

The assembly code which a compiler outputs to the above-mentioned C program is the following.

```
_int_WDTINT :  
    push elr, epsw  
    :  
    :  
    pop psw, pc
```

\* When an interrupt routine contains function call, in addition to the backup register group, the PUSH/POP code of the above-mentioned backup register group and the register group (LR, EA, R0-R3) which may be changed within the function called is generated.

## 1.6 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.

## 1.7 Stack Modifications

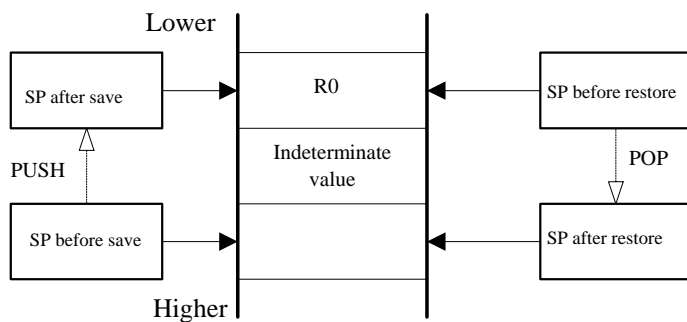
This Section summarizes the effects that PUSH and POP instructions have on the stack. For further details, see Chapter 3 “Instruction Descriptions.”

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 stores contexts of those operand with a dummy byte. Similarly, if the POP instruction operand represents an odd number of bytes, the hardware restores the register and reads a dummy byte without restoring any registers.

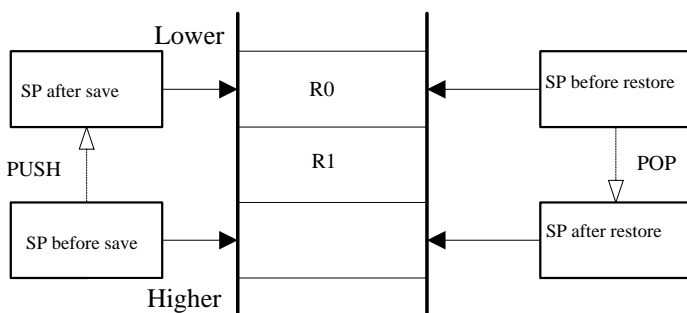
Note that the operation of instructions specifying LR or ELR as the operand depends on the hardware memory model.

The following Figures illustrate the operation of these two instructions.

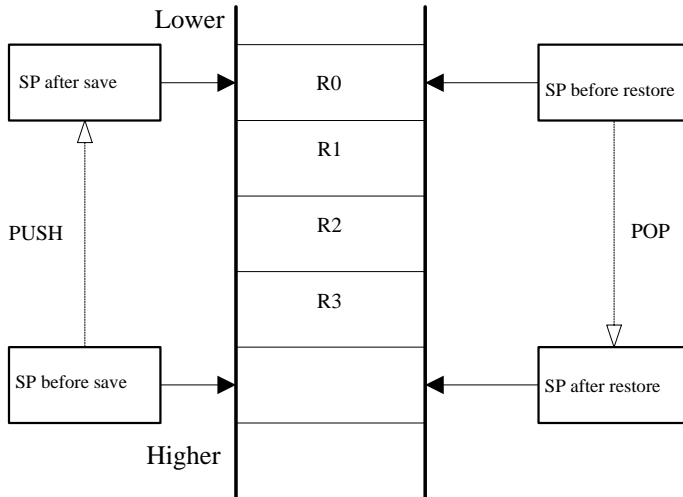
### PUSH R0 / POP R0



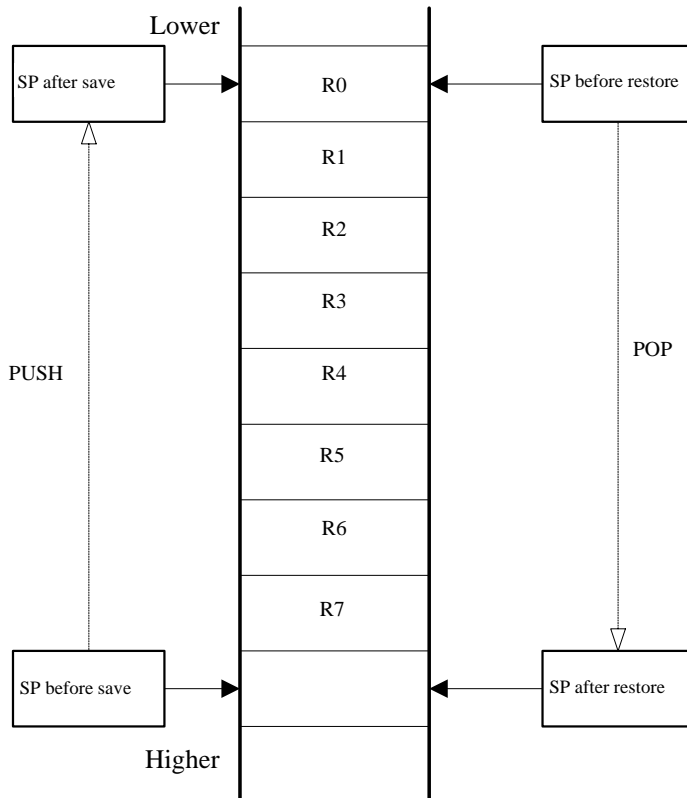
### PUSH ER0 / POP ER0



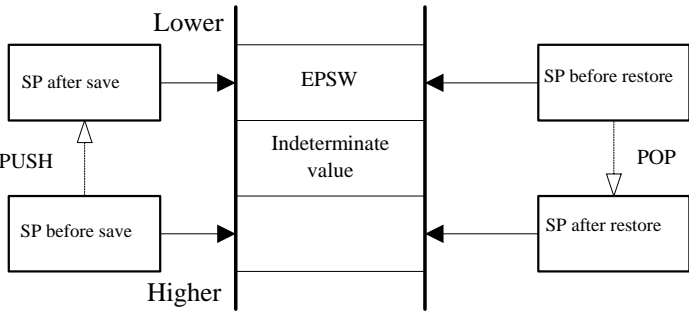
### PUSH XR0 / POP XR0



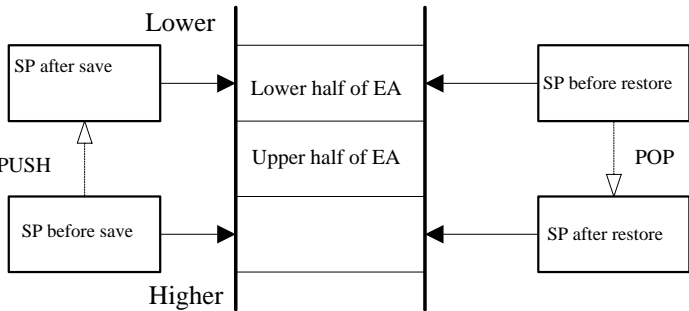
### PUSH QR0 / POP QR0



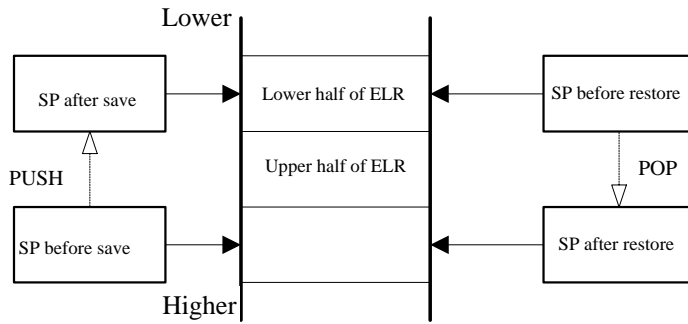
**PUSH EPSW / POP PSW**



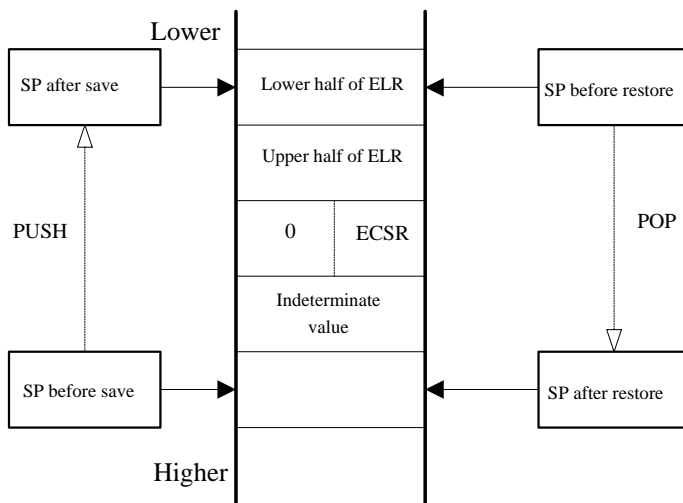
**PUSH EA / POP EA**



### PUSH ELR / POP PC (SMALL model)

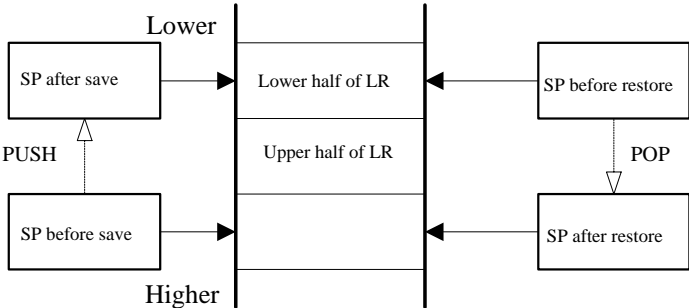


### PUSH ELR / POP PC (LARGE model)

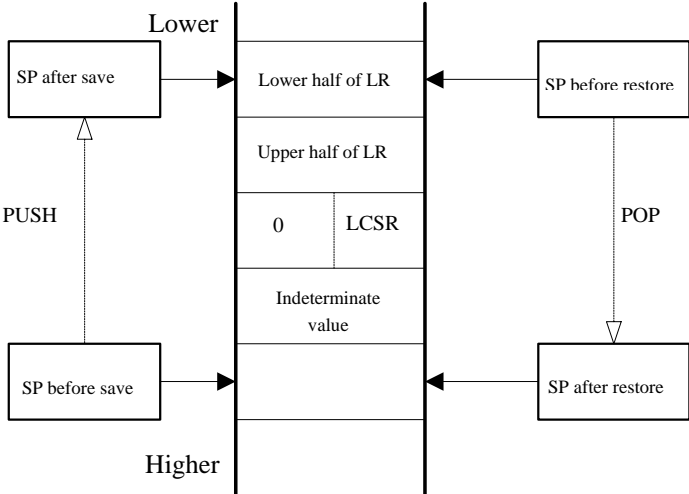




**PUSH LR / POP LR (SMALL model)**



**PUSH LR / POP LR (LARGE model)**



## **2. Addressing Types**



## 2.1 Addressing Types

The nX-U16/100 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
$Rn$	This addressing type accesses the contents of the specified byte-sized general register ( $Rn$ ).
$ERn$	This addressing type accesses the contents of the specified word-sized general register ( $ERn$ ). When the instruction table lists $ERn$ in an operand, BP may be substituted for ER12 and FP for ER14.
$XRn$	This addressing type accesses the contents of the specified double word-sized general register ( $XRn$ ).
$QRn$	This addressing type accesses the contents of the specified quad word-sized general register ( $QRn$ ).
$CRn$	This addressing type accesses the contents of the specified byte-sized coprocessor register ( $CRn$ ).
$CERn$	This addressing type accesses the contents of the specified word-sized coprocessor register ( $CERn$ ).
$CXRn$	This addressing type accesses the contents of the specified double word-sized coprocessor register ( $CXRn$ ).
$CQRn$	This addressing type accesses the contents of the specified quad word-sized coprocessor register ( $CQRn$ ).
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.
$Rn.bit\_offset$	This addressing type accesses the contents of bit specified by <i>bit_offset</i> in general register $Rn$ .

## 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 U16 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 <i>pseg_addr</i> : or FAR immediate value <i>iiii_iiii</i> .	
1001_0000_ddd_1111	Load DSR with the contents <i>Rd</i> : of the general register <i>Rd</i> .	
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, <u>1</u> :2345H	DSR ← 1 R0 ← [2345H]
L R0, <u>R1</u> : [ER2]	DSR ← R1 R0 ← [ER2]
ST R1, <u>DSR</u> : [EA]	[ (DSR << 16)   EA ] ← R1



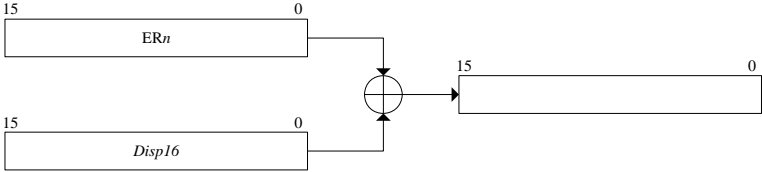
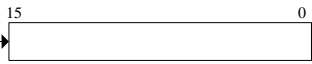
The underlined portions in the above Table indicate the DSR prefixes producing the desired DSR manipulations.

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

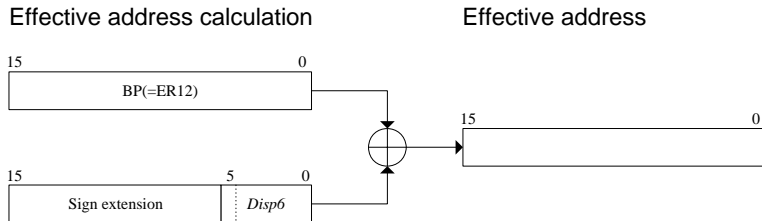
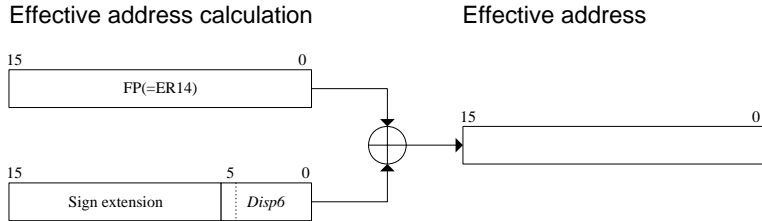
## 2.3.1 Register Indirect Addressing

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

Addressing Notation	Function																							
[EA]	<p>This addressing type accesses the contents of the data memory space at the offset in the EA register.</p> <div><div>Effective address calculation</div><div><div>150EA</div><div>→</div><div>150</div></div><div>Effective address</div></div>																							
<i>pseg_addr</i> : [EA]	<p>This variant uses the physical segment number specified by <i>#pseg_addr</i>.</p>																							
DSR: [EA]	<p>This variant uses the physical segment number in DSR.</p>																							
Rd: [EA]	<p>This variant uses the physical segment number in general register <i>Rd</i>.</p>																							
[EA+]	<p>This addressing type accesses the contents of the data memory space at the offset in the EA register.</p> <p>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.</p> <table><thead><tr><th>Operand size</th><th>EA Contents</th><th>Increment</th></tr></thead><tbody><tr><td rowspan="2">Byte</td><td>Even</td><td>1</td></tr><tr><td>Odd</td><td>1</td></tr><tr><td rowspan="2">Word</td><td>Even</td><td>2</td></tr><tr><td>Odd</td><td>1</td></tr><tr><td rowspan="2">Double word</td><td>Even</td><td>4</td></tr><tr><td>Odd</td><td>3</td></tr><tr><td rowspan="2">Quad word</td><td>Even</td><td>8</td></tr><tr><td>Odd</td><td>7</td></tr></tbody></table> <div><div>Effective address calculation</div><div><div>150EA</div><div>→</div><div>150</div></div><div>Effective address</div></div> <div><div>Contents incremented</div><div>after access</div></div>	Operand size	EA Contents	Increment	Byte	Even	1	Odd	1	Word	Even	2	Odd	1	Double word	Even	4	Odd	3	Quad word	Even	8	Odd	7
Operand size	EA Contents	Increment																						
Byte	Even	1																						
	Odd	1																						
Word	Even	2																						
	Odd	1																						
Double word	Even	4																						
	Odd	3																						
Quad word	Even	8																						
	Odd	7																						

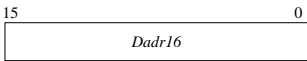
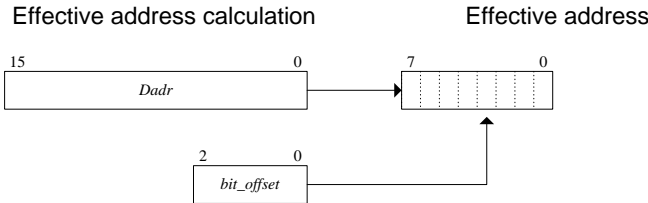
Addressing Notation	Function
<i>pseg_addr</i> : <i>[EA+]</i>	This variant uses the physical segment number specified by <i>#pseg_addr</i> .
DSR: <i>[EA+]</i>	This variant uses the physical segment number in DSR.
<i>Rd</i> : <i>[EA+]</i>	This variant uses the physical segment number in general register <i>Rd</i> .
<i>[ERn]</i>	This addressing type accesses the contents of the data memory space at the offset in the word-sized general register <i>ERn</i> . <i>[BP]</i> instead of <i>[ER12]</i> and <i>[FP]</i> instead of <i>[ER14]</i> are also acceptable.
<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <p>Effective address calculation</p>  </div> <div style="text-align: center;"> <p>Effective address</p>  </div> </div>	
<i>pseg_addr</i> : <i>[ERn]</i>	This variant uses the physical segment number specified by <i>#pseg_addr</i> .
DSR: <i>[ERn]</i>	This variant uses the physical segment number in DSR.
<i>Rd</i> : <i>[ERn]</i>	This variant uses the physical segment number in general register <i>Rd</i> .
<i>Disp16</i> [ <i>ERn</i> ]	This addressing type accesses the contents of the data memory space at the byte address formed by adding the displacement <i>Disp16</i> to the contents of the word-sized general register <i>ERn</i> .
<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <p>Effective address calculation</p>  </div> <div style="text-align: center;"> <p>Effective address</p>  </div> </div>	
<i>pseg_addr</i> : <i>Disp16</i> [ <i>ERn</i> ]	This variant uses the physical segment number specified by <i>#pseg_addr</i> .
DSR: <i>Disp16</i> [ <i>ERn</i> ]	This variant uses the physical segment number in DSR.
<i>Rd</i> : <i>Disp16</i> [ <i>ERn</i> ]	This variant uses the physical segment number in general register <i>Rd</i> .



Addressing Notation	Function
<i>Disp6</i> [BP]	<p>This addressing type accesses the contents of the data memory space at the byte address formed by adding the sign-extended displacement <i>Disp6</i> to the contents of the base pointer (BP).</p> <p>If there is no DSR prefix, this addressing type accesses physical segment #0 in the data memory space.</p> <div style="text-align: center;"> <p>Effective address calculation</p>  <p>Effective address</p> </div>
<i>pseg_addr:Disp6</i> [BP]	This variant uses the physical segment number specified by # <i>pseg_addr</i> .
DSR: <i>Disp6</i> [BP]	This variant uses the physical segment number in DSR.
Rd: <i>Disp6</i> [BP]	This variant uses the physical segment number in general register Rd.
<i>Disp6</i> [FP]	<p>This addressing type accesses the contents of the data memory space at the byte address formed by adding the sign-extended displacement <i>Disp6</i> to the contents of the frame pointer (FP).</p> <p>If there is no DSR prefix, this addressing type accesses physical segment #0 in the data memory space.</p> <div style="text-align: center;"> <p>Effective address calculation</p>  <p>Effective address</p> </div>
<i>pseg_addr:Disp6</i> [FP]	This variant uses the physical segment number specified by # <i>pseg_addr</i> .
DSR: <i>Disp6</i> [FP]	This variant uses the physical segment number in DSR.
Rd: <i>Disp6</i> [FP]	This variant uses the physical segment number in general register Rd.

## 2.3.2 Direct Addressing

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

Addressing Notation	Function
<i>Dadr</i>	This addressing type accesses the contents of the data memory space at the byte address in the instruction.  <div style="text-align: center;">Effective address  </div>
<i>pseg_addr:Dadr</i>	This variant uses the physical segment number <i>pseg_addr</i> .
<i>DSR:Dadr</i>	This variant uses the physical segment number in DSR.
<i>Rd:Dadr</i>	This variant uses the physical segment number in general register <i>Rd</i> .
<i>Dbitadr</i>	This addressing type accesses the contents of the data memory space at the bit address ( <i>Dadr.bit_offset</i> ) in the instruction.  <div style="text-align: center;">Effective address calculation      Effective address  </div>
<i>pseg_addr:Dbitadr</i>	This variant uses the physical segment number <i>pseg_addr</i> .
<i>DSR:Dbitadr</i>	This variant uses the physical segment number in DSR.
<i>Rd:Dbitadr</i>	This variant uses the physical segment number in general register <i>Rd</i> .

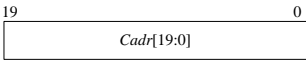
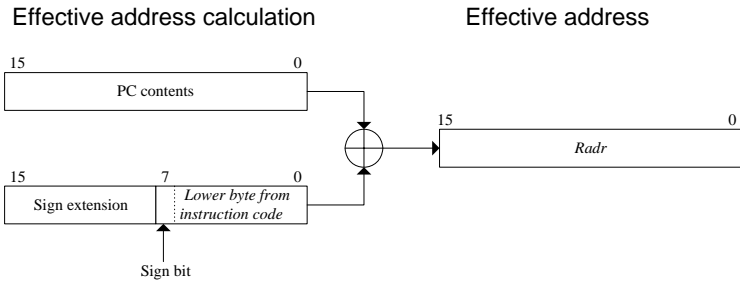
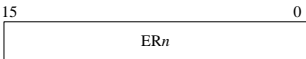
## 2.4 Immediate Addressing

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

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

## 2.5 Program/Code Memory Addressing

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

Addressing Notation	Function
<i>Cadr</i>	<p>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.</p> <p>Effective address</p> 
<i>Radr</i>	<p>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.</p> <p>Effective address calculation</p> 
<i>ERn</i>	<p>This addressing type specifies the contents of a word-sized general register <i>ERn</i> as the branch target offset for the B and BL instructions. The target must be in within the same physical segment.</p> <p>Effective address</p> 

# **3. Instruction Descriptions**

---

This Chapter describes the detailed operation of each instruction.



## 3.1 Overview

nX-U16/100 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
$\leftarrow$	Assignment
+ or $\oplus$	Addition
–	Subtraction
*	Multiplication
/	Division
$\gg$	Shift right
$\ll$	Shift left
=	Equality
!=	Inequality
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise inversion

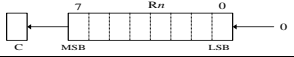
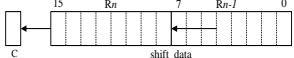
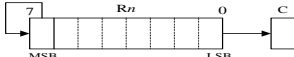
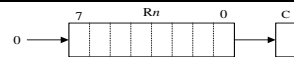
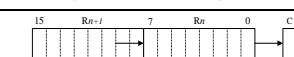
## 3.2 Instructions by Functional Group

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	$R_n$	$R_m$ $\#imm8$	*	*	*	*		*	Addition (8-bit) $R_n \leftarrow R_n + obj$
MOV	$R_n$	$R_m$ $\#imm8$		*	*				Data transfer (8-bit) $R_n \leftarrow obj$
ADDC	$R_n$	$R_m$ $\#imm8$	*	*	*	*		*	Addition with carry $R_n \leftarrow R_n + obj + c$
CMP	$R_n$	$R_m$ $\#imm8$	*	*	*	*		*	Comparison (8-bit) $R_n - obj$
CMPC	$R_n$	$R_m$ $\#imm8$	*	*	*	*		*	Comparison with carry $R_n - obj - c$
AND	$R_n$	$R_m$ $\#imm8$		*	*				Bitwise AND $R_n \leftarrow R_n \& obj$
OR	$R_n$	$R_m$ $\#imm8$		*	*				Bitwise OR $R_n \leftarrow R_n   obj$
XOR	$R_n$	$R_m$ $\#imm8$		*	*				Bitwise exclusive OR $R_n \leftarrow R_n \wedge obj$
SUB	$R_n$	$R_m$	*	*	*	*		*	Subtraction $R_n \leftarrow R_n - R_m$
SUBC	$R_n$	$R_m$	*	*	*	*		*	Subtraction with carry $R_n \leftarrow R_n - R_m - c$
MOV	$ER_n$	$ER_m$ $\#imm7$		*	*				Data transfer (16-bit) $ER_n \leftarrow obj$
ADD	$ER_n$	$ER_m$ $\#imm7$	*	*	*	*		*	Addition (16-bit) $ER_n \leftarrow ER_n + obj$
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	Rn	[EA]	*	*					Byte-sized data transfer Rn ← [EA]
		pseg_addr:[EA]	*	*					
		DSR:[EA]	*	*					
		Rd:[EA]	*	*					
		[EA+]	*	*					Byte-sized data transfer Rn ← [EA]
		pseg_addr:[EA+]	*	*					EA ← EA+1
		DSR:[EA+]	*	*					
		Rd:[EA+]	*	*					
		[ERm]	*	*					Byte-sized data transfer Rn ← [ERm]
		pseg_addr:[ERm]	*	*					
		DSR:[ERm]	*	*					
		Rd:[ERm]	*	*					
		Disp16[ERm]	*	*					Byte-sized data transfer Rn ← Disp16[ERm]
		pseg_addr: Disp16[ERm]	*	*					
		DSR: Disp16[ERm]	*	*					
		Rd: Disp16[ERm]	*	*					
		Disp6[BP]	*	*					Byte-sized data transfer Rn ← Disp6[BP]
		pseg_addr: Disp6[BP]	*	*					
		DSR: Disp6[BP]	*	*					
		Rd: Disp6[BP]	*	*					
		Disp6[FP]	*	*					Byte-sized data transfer Rn ← Disp6[FP]
		pseg_addr: Disp6[FP]	*	*					
		DSR: Disp6[FP]	*	*					
		Rd: Disp6[FP]	*	*					
		Dadr	*	*					Byte-sized data transfer Rn ← Dadr
		pseg_addr: Dadr	*	*					
		DSR: Dadr	*	*					
		Rd: Dadr	*	*					
	ERn	[EA]	*	*					Word-sized data transfer ERn ← [EA]
		pseg_addr:[EA]	*	*					
		DSR:[EA]	*	*					
		Rd:[EA]	*	*					
		[EA+]	*	*					Word-sized data transfer ERn ← [EA]
		pseg_addr:[EA+]	*	*					EA ← EA+1
		DSR:[EA+]	*	*					
		Rd:[EA+]	*	*					
		[ERm]	*	*					Word-sized data transfer ERn ← [ERm]
		pseg_addr:[ERm]	*	*					
		DSR:[ERm]	*	*					
		Rd:[ERm]	*	*					
		Disp16[ERm]	*	*					Word-sized data transfer ERn ← Disp16[ERm]
		pseg_addr: Disp16[ERm]	*	*					
		DSR: Disp16[ERm]	*	*					
		Rd: Disp16[ERm]	*	*					
		Disp6[BP]	*	*					Word-sized data transfer ERn ← Disp6[BP]
		pseg_addr: Disp6[BP]	*	*					
		DSR: Disp6[BP]	*	*					
		Rd: Disp6[BP]	*	*					
		Disp6[FP]	*	*					Word-sized data transfer ERn ← Disp6[FP]
		pseg_addr: Disp6[FP]	*	*					
		DSR: Disp6[FP]	*	*					
		Rd: Disp6[FP]	*	*					
		Dadr	*	*					Word-sized data transfer ERn ← Dadr
		pseg_addr: Dadr	*	*					
		DSR: Dadr	*	*					
		Rd: Dadr	*	*					

(continued on next page)

### Load/Store Instructions (cont.)

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

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

### Load/Store Instructions (cont.)

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function
ST	ER <sub>n</sub>	[ER <sub>m</sub> ] <i>pseg_addr</i> : [ER <sub>m</sub> ] DSR: [ER <sub>m</sub> ] Rd: [ER <sub>m</sub> ]							Word-sized data transfer [ER <sub>m</sub> ] ← ER <sub>n</sub>
		<i>Disp16</i> [ER <sub>m</sub> ] <i>pseg_addr</i> : <i>Disp16</i> [ER <sub>m</sub> ] DSR: <i>Disp16</i> [ER <sub>m</sub> ] Rd: <i>Disp16</i> [ER <sub>m</sub> ]							Word-sized data transfer <i>Disp16</i> [ER <sub>m</sub> ] ← ER <sub>n</sub>
		<i>Disp6</i> [BP] <i>pseg_addr</i> : <i>Disp6</i> [BP] DSR: <i>Disp6</i> [BP] Rd: <i>Disp6</i> [BP]							Word-sized data transfer <i>Disp6</i> [BP] ← ER <sub>n</sub>
		<i>Disp6</i> [FP] <i>pseg_addr</i> : <i>Disp6</i> [FP] DSR: <i>Disp6</i> [FP] Rd: <i>Disp6</i> [FP]							Word-sized data transfer <i>Disp6</i> [FP] ← ER <sub>n</sub>
		<i>Dadr</i> <i>pseg_addr</i> : <i>Dadr</i> DSR: <i>Dadr</i> Rd: <i>Dadr</i>							Word-sized data transfer [ <i>Dadr</i> ] ← ER <sub>n</sub>
	XR <sub>n</sub>	[EA] <i>pseg_addr</i> : [EA] DSR: [EA] Rd: [EA]							Double word-sized data transfer [EA] ← XR <sub>n</sub>
		[EA+] <i>pseg_addr</i> : [EA+] DSR: [EA+] Rd: [EA+]							Double word-sized data transfer [EA] ← XR <sub>n</sub> EA ← EA+1
	QR <sub>n</sub>	[EA] <i>pseg_addr</i> : [EA] DSR: [EA] Rd: [EA]							Quad word-sized data transfer [EA] ← QR <sub>n</sub>
		[EA+] <i>pseg_addr</i> : [EA+] DSR: [EA+] Rd: [EA+]							Quad word-sized data transfer [EA] ← QR <sub>n</sub> EA ← EA+1

### Control Register Access Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function
ADD	SP	<i>#signed8</i>							Addition $SP \leftarrow SP + \text{signed8}$
MOV	ECSR	<i>Rm</i>							Data transfer if ELEVEL is zero $LCSR \leftarrow Rm$ if ELEVEL is nonzero $ECSR[ELEVEL] \leftarrow Rm$
	ELR	<i>ERm</i>							Data transfer if ELEVEL is zero $LR \leftarrow ERm$ if ELEVEL is nonzero $ELR[ELEVEL] \leftarrow ERm$
	EPSW	<i>Rm</i>							Data transfer if ELEVEL is nonzero $EPSW[ELEVEL] \leftarrow Rm$
	<i>ERn</i>	ELR							Data transfer if ELEVEL is zero $ERn \leftarrow LR$ if ELEVEL is nonzero $ERn \leftarrow ELR[ELEVEL]$
		SP							Data transfer $ERn \leftarrow SP$
	PSW	<i>Rm</i>	*	*	*	*	*	*	Data transfer $PSW \leftarrow Rm$
		<i>#unsigned8</i>	*	*	*	*	*	*	Data transfer $PSW \leftarrow \text{unsigned8}$
	<i>Rn</i>	ECSR							Data transfer if ELEVEL is zero $Rn \leftarrow LCSR$ if ELEVEL is nonzero $Rn \leftarrow ECSR[ELEVEL]$
		EPSW							Data transfer if ELEVEL is nonzero $Rn \leftarrow EPSW[ELEVEL]$
		PSW							Data transfer $Rn \leftarrow PSW$
		SP							Data transfer $SP \leftarrow ERm$

### PUSH/POP Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function
PUSH	<i>ERn</i>								General register save $SP \leftarrow SP - n$ Stack $\leftarrow$ General register
	<i>Rn</i> <i>QRn</i> <i>XRn</i>								
	<i>register_list</i>								Control register save $SP \leftarrow SP - n$ Stack $\leftarrow$ Register set
POP	<i>ERn</i>								General register restore General register $\leftarrow$ Stack $SP \leftarrow SP + n$
	<i>Rn</i> <i>QRn</i> <i>XRn</i>								
	<i>register_list</i>		*	*	*	*	*	*	Control register restore Register set $\leftarrow$ Stack* <sup>1</sup> $SP \leftarrow SP + n$

\*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 <sub>n</sub>	R <sub>m</sub>		Byte-sized data transfer CR <sub>n</sub> ← R <sub>m</sub>
	CR <sub>n</sub>	[EA] pseg_addr:[EA] DSR:[EA] Rd:[EA]		Byte-sized data transfer CR <sub>n</sub> ← [EA]
		[EA+] pseg_addr:[EA+] DSR:[EA+] Rd:[EA+]		Byte-sized data transfer CR <sub>n</sub> ← [EA+] EA ← EA+1
	CER <sub>n</sub>	[EA] pseg_addr:[EA] DSR:[EA] Rd:[EA]		Word-sized data transfer CER <sub>n</sub> ← [EA]
		[EA+] pseg_addr:[EA+] DSR:[EA+] Rd:[EA+]		Word-sized data transfer CER <sub>n</sub> ← [EA+] EA ← EA+1
	CXR <sub>n</sub>	[EA] pseg_addr:[EA] DSR:[EA] Rd:[EA]		Double word-sized data transfer CXR <sub>n</sub> ← [EA]
		[EA+] pseg_addr:[EA+] DSR:[EA+] Rd:[EA+]		Double word-sized data transfer CXR <sub>n</sub> ← [EA+] EA ← EA+1
	CQR <sub>n</sub>	[EA] pseg_addr:[EA] DSR:[EA] Rd:[EA]		Quad word-sized continuous data transfer CQR <sub>n</sub> ← [EA]
		[EA+] pseg_addr:[EA+] DSR:[EA+] Rd:[EA+]		Quad word-sized continuous data transfer CQR <sub>n</sub> ← [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 $R_n \leftarrow CR_m$
	[EA] <i>pseg_addr</i> : [EA] DSR: [EA] <i>Rd</i> : [EA]	$CR_m$							Byte-sized data transfer $[EA] \leftarrow CR_m$
	[EA+] <i>pseg_addr</i> : [EA+] DSR: [EA+] <i>Rd</i> : [EA+]	$CR_m$							Byte-sized data transfer $[EA] \leftarrow CR_m$ $EA \leftarrow EA+1$
	[EA] <i>pseg_addr</i> : [EA] DSR: [EA] <i>Rd</i> : [EA]	$CER_m$							Word-sized data transfer $[EA] \leftarrow CER_m$
	[EA+] <i>pseg_addr</i> : [EA+] DSR: [EA+] <i>Rd</i> : [EA+]	$CER_m$							Word-sized data transfer $[EA] \leftarrow CER_m$ $EA \leftarrow EA+1$
	[EA] <i>pseg_addr</i> : [EA] DSR: [EA] <i>Rd</i> : [EA]	$CXR_m$							Double word-sized data transfer $[EA] \leftarrow CXR_m$
	[EA+] <i>pseg_addr</i> : [EA+] DSR: [EA+] <i>Rd</i> : [EA+]	$CXR_m$							Double word-sized data transfer $[EA] \leftarrow CXR_m$ $EA \leftarrow EA+1$
	[EA] <i>pseg_addr</i> : [EA] DSR: [EA] <i>Rd</i> : [EA]	$CQR_m$							Quad word-sized continuous data transfer $[EA] \leftarrow CQR_m$
	[EA+] <i>pseg_addr</i> : [EA+] DSR: [EA+] <i>Rd</i> : [EA+]	$CQR_m$							Quad word-sized continuous data transfer $[EA] \leftarrow CQR_m$ $EA \leftarrow EA+1$

### EA Register Data Transfer Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function
LEA	$[ER_n]$								Data transfer to EA $EA \leftarrow ER_n$
	<i>Disp16</i> [ $ER_m$ ]								$EA \leftarrow Disp16 + ER_m$
	<i>Dadr</i>								$EA \leftarrow Dadr$

### 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 $R_n \leftarrow 0 - R_n$

**Bit Access Instructions**

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function
SB	<i>Rn.bit_offset</i>			*					Set bit $z \leftarrow \sim Rn.bit\_offset$ $Rn.bit\_offset \leftarrow 1$
	<i>Dbitadr</i>			*					
	<i>pseg_addr: Dbitadr</i>			*					Set bit $z \leftarrow \sim [Dbitadr]$ $[Dbitadr] \leftarrow 1$
	<i>DSR: Dbitadr</i>			*					
	<i>Rd: Dbitadr</i>			*					
RB	<i>Rn.bit_offset</i>			*					Reset bit $z \leftarrow \sim Rn.bit\_offset$ $Rn.bit\_offset \leftarrow 0$
	<i>Dbitadr</i>			*					
	<i>pseg_addr: Dbitadr</i>			*					Reset bit $z \leftarrow \sim [Dbitadr]$ $[Dbitadr] \leftarrow 0$
	<i>DSR: Dbitadr</i>			*					
	<i>Rd: Dbitadr</i>			*					
TB	<i>Rn.bit_offset</i>			*					Test bit $z \leftarrow \sim Rn.bit\_offset$
	<i>Dbitadr</i>			*					
	<i>pseg_addr: Dbitadr</i>			*					Test bit $z \leftarrow \sim [Dbitadr]$
	<i>DSR: Dbitadr</i>			*					
	<i>Rd: Dbitadr</i>			*					

**PSW Access Instructions**

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

**Conditional Relative Branch Instructions**

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function
<i>Bcond</i>	<i>Radr</i>								Conditional branch if <i>cond</i> ? <i>Radr</i> : PC+2
BC	<i>cond</i>								

**Sign Extension Instruction**

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function
EXTBW	<i>ERn</i>			*	*				Extend sign $ERn \leftarrow (\text{sign-extends})Rn$

### Software Interrupt Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function
SWI	# <i>snum</i>						*		Software interrupt instruction address ← ( <i>snum</i> << 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	<i>Cadr</i>								Branch instruction CSR ← <i>Cadr</i> [19:16] PC ← <i>Cadr</i> [15:0]
	<i>ERn</i>								PC ← <i>ERn</i>
BL	<i>Cadr</i>								Branch instruction LR ← Address of next instruction LCSR ← CSR CSR ← <i>Cadr</i> [19:16] PC ← <i>Cadr</i> [15:0]
	<i>ERn</i>								LR ← Address of next instruction LCSR ← CSR PC ← <i>ERn</i>

### Multiplication and Division Instructions

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function
MUL	<i>ERn</i>	<i>Rm</i>			*				Multiplication $ERn \leftarrow Rn * Rm$
DIV	<i>ERn</i>	<i>Rm</i>		*	*				Division $ERn \leftarrow ERn / Rm, Rm \leftarrow ERn \bmod Rm$

### Miscellaneous

Mnemonic	First operand	Second operand	C	Z	S	OV	MIE	HC	Function
INC	[EA] <i>pseg_addr</i> : [EA] DSR: [EA] Rd: [EA]			*	*	*		*	Memory increment [EA] ← [EA] + 1
DEC	[EA] <i>pseg_addr</i> : [EA] DSR: [EA] Rd: [EA]			*	*	*		*	Memory decrement [EA] ← [EA] - 1
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-U16/100 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-U16/100 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-U16/100 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  $n \times m$  machine cycles, where  $n$  is the number of accesses and  $m$  the memory wait cycles for single access. 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-U16/100 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	ERn	ERm	1		
		#imm7	1		
ADD	Rn	Rm	1		
		#imm8	1		
	SP	#signed8	1		
ADDC	Rn	Rm	1		
		#imm8	1		
AND	Rn	Rm	1		
		#imm8	1		
B	Cadr		2		1
	ERn		2		1
Bcond	Radr		A34 core : 1 / 3 <sup>(*)</sup> A35 core: 1 / 2 <sup>(*)</sup>		1
BL	Cadr		2		1
	ERn		2		1
BRK			7		1
CMP	ERn	ERm	1		
	Rn	Rm	1		
		#imm8	1		
CMPC	Rn	Rm	1		
		#imm8	1		
CPLC			1		
DAA	Rn		1		
DAS	Rn		1		
DEC	[EA]		2		1
	pseg_addr:[EA]				
	DSR:[EA]		3		
	Rd:[EA]				
DI			3		
DIV	ERn	Rm	17		
EI			1		
EXTBW	ERn		1		
INC	[EA]		2		1
	pseg_addr:[EA]				
	DSR:[EA]		3		
	Rd:[EA]				

\*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 <sub>n</sub>	[EA]	1	1	
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	2	1	
		Rd: [EA]			
		[EA+]	1	1	
		<i>pseg_addr</i> : [EA+]			
		DSR: [EA+]	2	1	
		Rd: [EA+]			
		[ER <sub>m</sub> ]	1	1	1
		<i>pseg_addr</i> : [ER <sub>m</sub> ]			
		DSR: [ER <sub>m</sub> ]	2	1	
		Rd: [ER <sub>m</sub> ]			
		<i>Disp16</i> [ER <sub>m</sub> ]	2	2	1
		<i>pseg_addr</i> : <i>Disp16</i> [ER <sub>m</sub> ]			
		DSR: <i>Disp16</i> [ER <sub>m</sub> ]	4	2	
		Rd: <i>Disp16</i> [ER <sub>m</sub> ]			
		<i>Disp6</i> [BP]	2	2	1
		<i>pseg_addr</i> : <i>Disp6</i> [BP]			
		DSR: <i>Disp6</i> [BP]	4	2	
		Rd: <i>Disp6</i> [BP]			
		<i>Disp6</i> [FP]	2	2	1
		<i>pseg_addr</i> : <i>Disp6</i> [FP]			
		DSR: <i>Disp6</i> [FP]	4	2	
		Rd: <i>Disp6</i> [FP]			
		<i>Dadr</i>	2	2	1
		<i>pseg_addr</i> : <i>Dadr</i>			
		DSR: <i>Dadr</i>	3	2	
		Rd: <i>Dadr</i>			
	QR <sub>n</sub>	[EA]	4	4	
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	5	4	
		Rd: [EA]			
		[EA+]	4	4	
		<i>pseg_addr</i> : [EA+]			
		DSR: [EA+]	5	4	
		Rd: [EA+]			

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
L	Rn	[EA]	1	1	
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	2	1	
		Rd: [EA]			
		[EA+]	1	1	
		<i>pseg_addr</i> : [EA+]			
		DSR: [EA+]	2	1	
		Rd: [EA+]			
		[ERm]	1	1	1
		<i>pseg_addr</i> : [ERm]			
		DSR: [ERm]	2	1	
		Rd: [ERm]			
		<i>Disp16</i> [ERm]	2	1	1
		<i>pseg_addr</i> : <i>Disp16</i> [ERm]			
		DSR: <i>Disp16</i> [ERm]	3	1	
		Rd: <i>Disp16</i> [ERm]			
		<i>Disp6</i> [BP]	2	1	1
		<i>pseg_addr</i> : <i>Disp6</i> [BP]			
		DSR: <i>Disp6</i> [BP]	3	1	
		Rd: <i>Disp6</i> [BP]			
		<i>Disp6</i> [FP]	2	1	1
		<i>pseg_addr</i> : <i>Disp6</i> [FP]			
		DSR: <i>Disp6</i> [FP]	3	1	
		Rd: <i>Disp6</i> [FP]			
		<i>Dadr</i>	2	1	1
		<i>pseg_addr</i> : <i>Dadr</i>			
		DSR: <i>Dadr</i>	3	1	
		Rd: <i>Dadr</i>			
	XRn	[EA]	2	2	
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	3	2	
		Rd: [EA]			
		[EA+]	2	2	
		<i>pseg_addr</i> : [EA+]			
		DSR: [EA+]	3	2	
		Rd: [EA+]			

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
LEA	[ERm]		1		
	<i>Disp16</i> [ERm]		2		
	<i>Dadr</i>		2		
MOV	CERn	[EA]	1	1	1
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	2	1	
		Rd: [EA]			
		[EA+]	1	1	1
		<i>pseg_addr</i> : [EA+]			
	CQRn	DSR: [EA+]	2	1	
		Rd: [EA+]			
		[EA]	4	4	1
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	5	4	
		Rd: [EA]			
	CRn	[EA+]	4	4	1
		<i>pseg_addr</i> : [EA+]			
		DSR: [EA+]	5	4	
		Rd: [EA+]			
		[EA]	1	1	1
		<i>pseg_addr</i> : [EA]			
	CRn	DSR: [EA]	2	1	
		Rd: [EA]			
		[EA+]	1	1	1
		<i>pseg_addr</i> : [EA+]			
		DSR: [EA+]	2	1	
		Rd: [EA+]			
	CRn	Rm	1		
	CXRn	[EA]	2	2	1
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	3	2	
		Rd: [EA]			
		[EA+]	2	2	1
		<i>pseg_addr</i> : [EA+]			
	ECSR	DSR: [EA+]	3	2	
		Rd: [EA+]			
		Rm	1		
	ELR	ERm	1		
	EPSW	Rm	1		
	ERn	ELR	1		
		ERm	1		
		#imm7	1		
		SP	1		

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

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
MUL	$ER_n$	$Rm$	9		
NEG	$Rn$		1		
NOP			1		
OR	$Rn$	$Rm$	1		
		$\#imm8$	1		
POP	EA		2		1
	EA,LR		3 / 4 <sup>(*)</sup>		1
	EA,PC		5 / 6 <sup>(*)</sup>		1
	EA,PC,LR		6 / 8 <sup>(*)</sup>		1
	EA,PC,PSW		6 / 7 <sup>(*)</sup>		1
	EA,PC,PSW,LR		7 / 9 <sup>(*)</sup>		1
	EA,PSW		3		1
	EA,PSW,LR		4 / 5 <sup>(*)</sup>		1
	LR		1 / 2 <sup>(*)</sup>		1
	LR,PSW		2 / 3 <sup>(*)</sup>		1
	PC		3 / 4 <sup>(*)</sup>		1
	PC,LR		4 / 6 <sup>(*)</sup>		1
	PC,PSW		4 / 5 <sup>(*)</sup>		1
	PC,PSW,LR		5 / 7 <sup>(*)</sup>		1
	PSW		1		1
	$ER_n$		1		1
	$QR_n$		4		1
	$Rn$		1		1
	$XR_n$		2		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		1		1
	ELR		1 / 2 <sup>(*)</sup>		1
	EA,ELR		2 / 3 <sup>(*)</sup>		1
	EPSW		1		1
	EPSW,EA		2		1
	EPSW,ELR		2 / 3 <sup>(*)</sup>		1
	EPSW,ELR,EA		3 / 4 <sup>(*)</sup>		1
	LR		1 / 2 <sup>(*)</sup>		1
	LR,EA		2 / 3 <sup>(*)</sup>		1
	LR,ELR		2 / 4 <sup>(*)</sup>		1
	LR,EA,ELR		3 / 5 <sup>(*)</sup>		1
	LR,EPSW		2 / 3 <sup>(*)</sup>		1
	LR,EPSW,EA		3 / 4 <sup>(*)</sup>		1
	LR,EPSW,ELR		3 / 5 <sup>(*)</sup>		1
	LR,ELR,EPSW,EA		4 / 6 <sup>(*)</sup>		1
	<i>ERn</i>		1		1
	<i>QRn</i>		4		1
	<i>Rn</i>		1		1
	<i>XRn</i>		2		1
RB	<i>Dbitadr</i>		2		1
	<i>pseg_addr: Dbitadr</i>				
	<i>DSR: Dbitadr</i>		3		
	<i>Rd: Dbitadr</i>				
	<i>Rn.bit_offset</i>		1		
RC			1		
RT			2		1
RTI			2		1
SB	<i>Dbitadr</i>		2		1
	<i>pseg_addr: Dbitadr</i>				
	<i>DSR: Dbitadr</i>		3		
	<i>Rd: Dbitadr</i>				
	<i>Rn.bit_offset</i>		1		
SC			1		

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



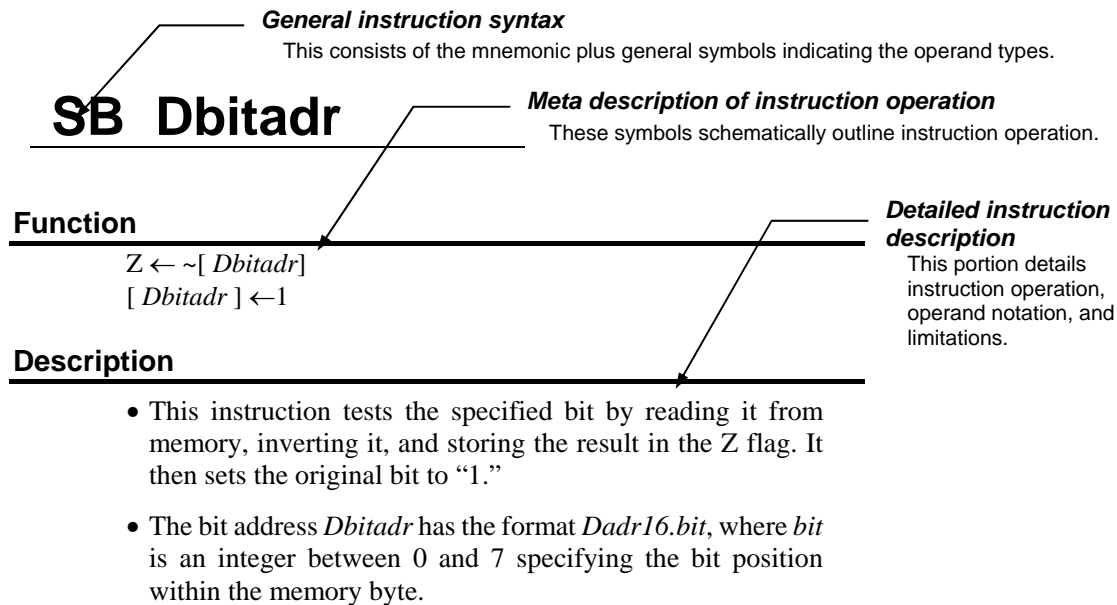
Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
SLL	$R_n$	$R_m$	1		1
		$\#width$	1		1
SLLC	$R_n$	$R_m$	1		1
		$\#width$	1		1
SRA	$R_n$	$R_m$	1		1
		$\#width$	1		1
SRL	$R_n$	$R_m$	1		1
		$\#width$	1		1
SRLC	$R_n$	$R_m$	1		1
		$\#width$	1		1
ST	$ER_n$	[EA]	1		
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	2		
		Rd: [EA]			
		[EA+]	1		
		<i>pseg_addr</i> : [EA+]			
		DSR: [EA+]	2		
		Rd: [EA+]			
		[ER $m$ ]	1		1
		<i>pseg_addr</i> : [ER $m$ ]			
		DSR: [ER $m$ ]	2		
		Rd: [ER $m$ ]			
		Disp16[ER $m$ ]	2		1
		<i>pseg_addr</i> : Disp16[ER $m$ ]			
		DSR: Disp16[ER $m$ ]	3		
		Rd: Disp16[ER $m$ ]			
		Disp6[BP]	2		1
		<i>pseg_addr</i> : Disp6[BP]			
		DSR: Disp6[BP]	3		
		Rd: Disp6[BP]			
		Disp6[FP]	2		1
		<i>pseg_addr</i> : Disp6[FP]			
		DSR: Disp6[FP]	3		
		Rd: Disp6[FP]			
		Dadr	2		1
		<i>pseg_addr</i> : Dadr			
		DSR: Dadr	3		
		Rd: Dadr			

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
ST	QRn	[EA]	4		
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	5		
		Rd: [EA]			
		[EA+]	4		
		<i>pseg_addr</i> : [EA+]			
		DSR: [EA+]	5		
		Rd: [EA+]			
	Rn	[EA]	1		
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	2		
		Rd: [EA]			
		[EA+]	1		
		<i>pseg_addr</i> : [EA+]			
		DSR: [EA+]	2		
		Rd: [EA+]			
		[ERm]	1		1
		<i>pseg_addr</i> : [ERm]			
		DSR: [ERm]	2		
		Rd: [ERm]			
		<i>Disp16</i> [ERm]	2		1
		<i>pseg_addr</i> : <i>Disp16</i> [ERm]			
		DSR: <i>Disp16</i> [ERm]	3		
		Rd: <i>Disp16</i> [ERm]			
		<i>Disp6</i> [BP]	2		1
		<i>pseg_addr</i> : <i>Disp6</i> [BP]			
		DSR: <i>Disp6</i> [BP]	3		
		Rd: <i>Disp6</i> [BP]			
		<i>Disp6</i> [FP]	2		1
		<i>pseg_addr</i> : <i>Disp6</i> [FP]			
		DSR: <i>Disp6</i> [FP]	3		
		Rd: <i>Disp6</i> [FP]			
		<i>Dadr</i>	2		1
		<i>pseg_addr</i> : <i>Dadr</i>			
		DSR: <i>Dadr</i>	3		
		Rd: <i>Dadr</i>			
	XRn	[EA]	2		
		<i>pseg_addr</i> : [EA]			
		DSR: [EA]	3		
		Rd: [EA]			
		[EA+]	2		
		<i>pseg_addr</i> : [EA+]			
		DSR: [EA+]	3		
		Rd: [EA+]			

Mnemonic	First operand	Second operand	Minimum execution time (cycles)	ROM window access	[EA+] addressing delay
SUB	$Rn$	$Rm$	1		
SUBC	$Rn$	$Rm$	1		
SWI	$\#snum$		3		1
TB	$Dbitadr$		2	1	1
	$pseg\_addr: Dbitadr$				
	DSR: $Dbitadr$		3	1	
	$Rd: Dbitadr$				
	$Rn.bit\_offset$		1		
XOR	$Rn$	$Rm$	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.



### 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	<i>Dbitadr</i>		A	0	<i>bit</i>	0
	*: <i>Dbitadr</i>	<word>	A	0	<i>bit</i>	0

#### Instruction code

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

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	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

$$ER_n \leftarrow ER_n + ER_m$$

### 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 $\vdots$ $n$ $\vdots$ $m$ $\vdots$ 6	

ADD ERn , #imm7

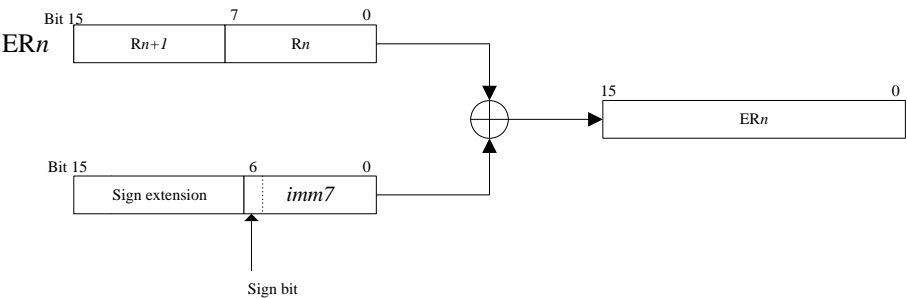
Add

Function

$$ERn \leftarrow ERn + (\text{signed})imm7$$

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	ERn	#imm7	E   n   1   imm7	

## ADD $R_n$ , $obj$

Add

### Function

$R_n \leftarrow R_n + obj$

### 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

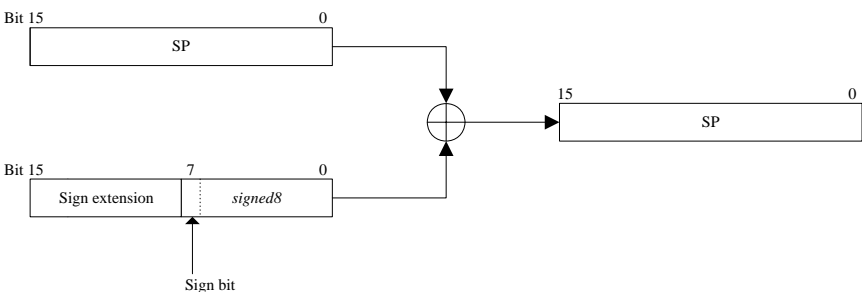
Add

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 $Rn, obj$

Add with carry

### Function

$$Rn \leftarrow Rn + obj + C$$

### 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	$Rn$	$Rm$	8	$n$	$m$	6
		$\#imm8$	6	$n$	$imm8$	

## AND $Rn, obj$

Bitwise AND

### Function

$Rn \leftarrow Rn \& obj$

### 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	$Rn$	$Rm$	8 $n$ $m$ 2	
		$\#imm8$	2 $n$ $imm8$	

## B Cadr

Direct branch

### Function

CSR  $\leftarrow$  Cadr[19:16]  
PC  $\leftarrow$  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	<i>Cadr</i>	F	<i>Cadr</i> [19:16]	0	<i>Cadr</i> [15:0]

B ERn

Indirect branch

Function

PC ← ERn

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	ERn	F	0	n	2

# Bcond Radr

## BC *cond* , *Radr*

Conditional branch

### Function

If (*cond* = true) then  $PC \leftarrow Radr$

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

Mnemonic	First operand	Second operand	Instruction Format		
			First word		
<i>Bcond</i>	<i>Radr</i>		<i>C</i>	<i>condition</i>	<i>(Radr - NextPC)&gt;&gt;1</i>
BC	<i>cond</i>	<i>Radr</i>			

## Condition

Instruction Syntax				
<i>Bcond</i>	BC <i>cond</i>	<i>Condition</i>	Meaning	Flag condition
BGE	BC GE	0000	Unsigned $\geq$	$C=0$
BNC	BC NC			
BLT	BC LT	0001	Unsigned $<$	$C=1$
BCY	BC CY			
BGT	BC GT	0010	Unsigned $>$	$(C=0) \& \& (Z=0)$
BLE	BC LE	0011	Unsigned $\leq$	$(Z=1) \vee (C=1)$
BGES	BC GES	0100	Signed $\geq$	$(OV \wedge S)=0$
BLTS	BC LTS	0101	Signed $<$	$(OV \wedge S)=1$
BGTS	BC GTS	0110	Signed $>$	$((OV \wedge S) \vee Z) = 0$
BLES	BC LES	0111	Signed $\leq$	$((OV \wedge S) \vee Z) = 1$
BNE	BC NE	1000	$\neq$	$Z=0$
BNZ	BC NZ			
BEQ	BC EQ	1001	$=$	$Z=1$
BZ	BC ZF			
BNV	BC NV	1010	No overflow	$OV=0$
BOV	BC OV	1011	Overflow	$OV=1$
BPS	BC PS	1100	Positive	$S=0$
BNS	BC NS	1101	Negative	$S=1$
BAL	BC AL	1110	Unconditional	

## 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	<i>Cadr</i>	F	<i>Cadr</i> [19:16]	0	1	<i>Cadr</i> [15:0]

## BL ER<sub>n</sub>

Branch and link

### Function

PC ← ER<sub>n</sub>  
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 <sub>n</sub>	F	0	n	3



# BRK

Break instruction  
(software reset)

## Function

- ELEVEL greater than 1: System reset
- ELEVEL less than 2:
  - ELR2  $\leftarrow$  Address of next instruction
  - ECSR2  $\leftarrow$  CSR
  - EPSW2  $\leftarrow$  PSW
  - ELEVEL  $\leftarrow$  2
  - PC  $\leftarrow$  (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<sub>n</sub> , ER<sub>m</sub>

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

Mnemonic	First operand	Second operand	Instruction Format			
			First word		Second word	
CMP	ER <sub>n</sub>	ER <sub>m</sub>	F	n	m	7

## CMP $Rn, obj$

Compare

### Function

$Rn - obj$

### 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

Mnemonic	First operand	Second operand	Instruction Format	
			First word	Second word
CMP	$Rn$	$Rm$	8 $n$ $m$ 7	
		$\#imm8$	7 $n$ $imm8$	

## CMPC $Rn$ , $obj$

Compare with carry

### Function

$Rn - obj - C$

### Description

- This instruction compares the contents of the  $Rn$  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	$Rn$	$Rm$	8 $n$ $m$ 5	
		$\#imm8$	5 $n$ $imm8$	

# CPLC

Complement carry flag

## Function

$C \leftarrow \sim C$

## 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 $Rn$

Byte-sized decimal  
adjustment for addition

## Function

$Rn \leftarrow (\text{decimal adjustment}) Rn$

## 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	$Rn[7:4]$	HC	$Rn[3:0]$	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	$Rn$		8	$n$	1	F

# DAS $Rn$

Byte-sized decimal adjustment  
for subtraction

## Function

$Rn \leftarrow (\text{decimal adjustment}) Rn$

## 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	$Rn[7:4]$	HC	$Rn[3:0]$	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	$Rn$		8 $n$ 3    F	

# DEC [EA]

Memory decrement  
(using EA indirect addressing)

## Function

$[EA] \leftarrow [EA] - 1$

## 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>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F



# DI

Disable interrupts

## Function

$MIE \leftarrow 0$

## 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

$$ER_n \leftarrow ER_n / R_m$$

$$R_m \leftarrow ER_n \bmod R_m$$

### 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

$MIE \leftarrow 1$

## 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

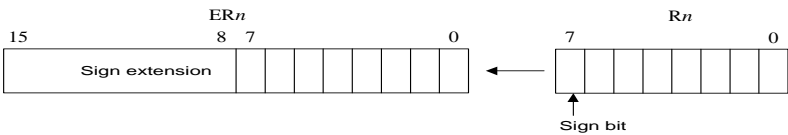
## Instruction Format

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

EXTBW ERn

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

$[EA] \leftarrow [EA] + 1$

## 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>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

L ERn, obj

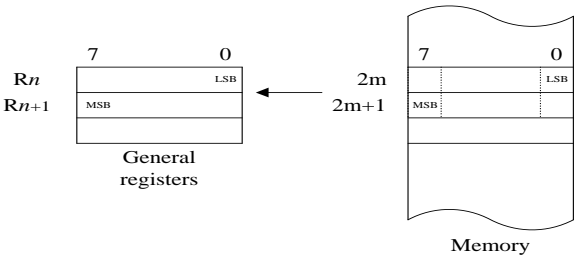
Word-sized data transfer

Function

$$ERn \leftarrow obj$$

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 <sub>n</sub>	[EA]		9	n	3	2
		*:[EA]	<word>	9	n	3	2
		[EA+]		9	n	5	2
		*:[EA+]	<word>	9	n	5	2
		[ER <sub>m</sub> ]		9	n	m	2
		*:[ER <sub>m</sub> ]	<word>	9	n	m	2
		Disp16[ER <sub>m</sub> ]		A	n	m	8
		*:Disp16[ER <sub>m</sub> ]	<word>	A	n	m	8
		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	<word>	9	n	1	2

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

# L QRn,obj

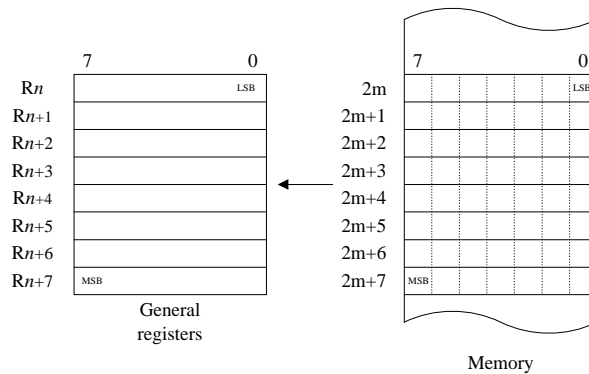
Quad word-sized  
data transfer

## Function

$QRn \leftarrow obj$

## 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	QRn	[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
Rd	9	0	d	F



## L *Rn, obj*

Byte-sized data transfer

---

### Function

$Rn \leftarrow obj$

---

### 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 <sub>n</sub>	[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[ERm]		9	n	m	8	Disp16	
		*:Disp16[ERm]	<word>	9	n	m	8	Disp16	
		Disp6[BP]		D	n	0:0:0:0	Disp6		
		*:Disp6[BP]	<word>	D	n	0:0:0:0	Disp6		
		Disp6[FP]		D	n	0:1:0:0	Disp6		
		*:Disp6[FP]	<word>	D	n	0:1:0:0	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
Rd	9	0	d	F

# L $XRn, obj$

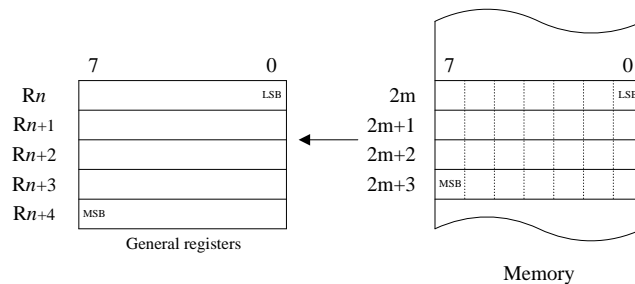
Double word-sized  
data transfer

## Function

$XRn \leftarrow obj$

## 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	$XRn$	[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
$Rd$	9	0	$d$	F

# LEA *obj*

Load EA

## Function

$EA \leftarrow obj$

## 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	[ERm ]		F	0	m	A		
	<i>Dadr</i>		F	0	0	C	<i>Dadr</i>	
	<i>Disp16</i> [ERm]		F	0	m	B	<i>Disp16</i>	

# MOV CER<sub>n</sub>, obj

Coprocessor data transfer

## Function

$CER_n \leftarrow obj$

## 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 <sub>n</sub>	[EA]		F <i>n</i> 2    D	
		*:[EA]	<word>	F <i>n</i> 2    D	
		[EA+]		F <i>n</i> 3    D	
		*:[EA+]	<word>	F <i>n</i> 3    D	

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

# MOV CQR*n* , *obj*

Coprocessor data transfer

## Function

$CQRn \leftarrow obj$

## 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 <i>n</i>	[EA]		F	<i>n</i>	6	D
		*:[EA]	<word>	F	<i>n</i>	6	D
		[EA+]		F	<i>n</i>	7	D
		*:[EA+]	<word>	F	<i>n</i>	7	D

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

# MOV CR<sub>n</sub> , obj

Coprocessor data transfer

## Function

$CR_n \leftarrow obj$

## 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 <sub>n</sub>	[EA]		F    n    0    D	
		*: [EA]	<word>	F    n    0    D	
		[EA+]		F    n    1    D	
		*:[EA+]	<word>	F    n    1    D	

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

MOV CRn , Rm

Coprocessor data transfer

Function

CRn ←Rm

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	CRn	Rm	A    n    m    E	



# MOV CXR*n* , *obj*

Coprocessor data transfer

## Function

$CXR_n \leftarrow obj$

## 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 <sub><i>n</i></sub>	[EA]		F	<i>n</i>	4	D
		*:[EA]	<word>	F	<i>n</i>	4	D
		[EA+]		F	<i>n</i>	5	D
		*:[EA+]	<word>	F	<i>n</i>	5	D

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

MOV ECSR , Rm

Data transfer

Function

- If ELEVEL is zero  
LCSR ←Rm
- If ELEVEL is nonzero  
ECSR[ELEVEL] ←Rm

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	Rm	A	0	m	F

# MOV ELR , ER<sub>m</sub>

Data transfer

## Function

- If ELEVEL is zero  
 $LR \leftarrow ER_m$
- If ELEVEL is nonzero  
 $ELR[ELEVEL] \leftarrow ER_m$

## 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 <sub>m</sub>	A	m	0	D

MOV EPSW , Rm

Data transfer

Function

- If ELEVEL is nonzero  
EPSW[ELEVEL] ←Rm

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

Mnemonic	First operand	Second operand	Instruction Format			
			First word			Second word
MOV	EPSW	Rm	A	0	m	C

# MOV ER<sub>n</sub>, ELR

Data transfer

## Function

- If ELEVEL is zero  
 $ER_n \leftarrow LR$
- If ELEVEL is nonzero  
 $ER_n \leftarrow ELR[ELEVEL]$

## 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 <sub>n</sub>	ELR	A	n	0	5	

# MOV ER<sub>n</sub>, ER<sub>m</sub>

Data transfer

## Function

$ER_n \leftarrow ER_m$

## 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

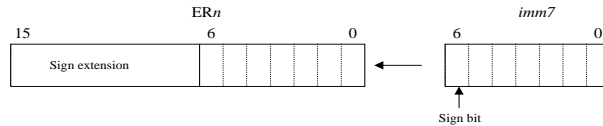
Mnemonic	First operand	Second operand	Instruction Format	
			First word	Second word
MOV	ER <sub>n</sub>	ER <sub>m</sub>	F    n    m    5	

# MOV ERn, #imm7

Data transfer

## Function

$ERn \leftarrow (\text{sign-extends})imm7$



## Description

- This instruction loads the sign-extended *imm7* into the specified word-sized register. More precisely, it loads the immediate value into *Rn*, the lower half of the register, and copies bit 6 from the immediate value into *Rn* bit 7 and all bits of *Rn+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	ERn	#imm7	E n 0 imm7	

# MOV ER<sub>n</sub>, SP

Data transfer

## Function

ER<sub>n</sub> ← 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 <sub>n</sub>	SP	A	n	1	A



# MOV *obj*, *CERm*

Coprocessor data transfer

## Function

(WORD) *obj* ← *CERm*

## 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]	<i>CERm</i>		F	<i>m</i>	A	D
	* : [EA]	<i>CERm</i>	<word>	F	<i>m</i>	A	D
	[EA+]	<i>CERm</i>		F	<i>m</i>	B	D
	* : [EA+]	<i>CERm</i>	<word>	F	<i>m</i>	B	D

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	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 <i>m</i>		F	<i>m</i>	E	D
	*: [EA]	CQR <i>m</i>	<word>	F	<i>m</i>	E	D
	[EA+]	CQR <i>m</i>		F	<i>m</i>	F	D
	*: [EA+]	CQR <i>m</i>	<word>	F	<i>m</i>	F	D

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

# MOV *obj*, *CRm*

Coprocessor data transfer

## Function

(BYTE) *obj* ← *CRm*

## 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]	<i>CRm</i>		F <i>m</i> 8   D	
	*: [EA]	<i>CRm</i>	<word>	F <i>m</i> 8   D	
	[EA+]	<i>CRm</i>		F <i>m</i> 9   D	
	*: [EA+]	<i>CRm</i>	<word>	F <i>m</i> 9   D	

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	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 <i>m</i>		F	<i>m</i>	C	D
	*: [EA]	CXR <i>m</i>	<word>	F	<i>m</i>	C	D
	[EA+]	CXR <i>m</i>		F	<i>m</i>	D	D
	*: [EA+]	CXR <i>m</i>	<word>	F	<i>m</i>	D	D

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

# MOV PSW , *obj*

Data transfer

## Function

$PSW \leftarrow obj$

## 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	# <i>unsigned8</i>	E 9 <i>unsigned8</i>	
		<i>Rm</i>	A 0 <i>m</i> B	

MOV *Rn* , *CRm*

Coprocessor data transfer

Function

$Rn \leftarrow CRm$

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	<i>Rn</i>	<i>CRm</i>	A	<i>n</i>	<i>m</i>	6

# MOV $R_n$ , ECSR

Data transfer

## Function

- If ELEVEL is zero  
 $R_n \leftarrow \text{LCSR}$
- If ELEVEL is nonzero  
 $R_n \leftarrow \text{ECSR}[\text{ELEVEL}]$

## 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 Rn , EPSW

Data transfer

Function

- If ELEVEL is nonzero  
 $Rn \leftarrow \text{EPSW}[\text{ELEVEL}]$

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 loads 0xFF.

Flags

C	Z	S	OV	MIE	HC
—	—	—	—	—	—

—: No change

Instruction Format

Mnemonic	First operand	Second operand	Instruction Format	
			First word	Second word
MOV	Rn	EPSW	A    n    0    4	



## MOV $R_n$ , PSW

Data transfer

### Function

$R_n \leftarrow \text{PSW}$

### 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

$R_n \leftarrow obj$

## 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<sub>m</sub>

Data transfer

## Function

SP ← ER<sub>m</sub>

## 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 <sub>m</sub>	A	1	<i>m</i>	A

# MUL ER<sub>n</sub>,R<sub>m</sub>

Multiplication

## Function

$ER_n \leftarrow R_n * R_m$  ( $n$  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 <sub>n</sub>	R <sub>m</sub>	F	n	m	4

# NEG $R_n$

Negate

## Function

$R_n \leftarrow 0 - R_n$

## Description

- This instruction calculates the two's 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 $R_n$ , $obj$

Bitwise OR

## Function

$R_n \leftarrow R_n \mid obj$

## 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  $\leftarrow$  (SP)

SP  $\leftarrow$  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 1.7 “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  $\rightarrow$  LR  $\rightarrow$  PSW  $\rightarrow$  PC

- 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 odd number of bytes operand (*Rn*) loads the specified register and reads a dummy byte without modifying any other registers.  
For further details, see Section 1.7 “Stack Modifications.”

Flags

C	Z	S	OV	MIE	HC
–	–	–	–	–	–

–: No change

Instruction Format

Mnemonic	First operand	Instruction Format			
		First word			
POP	<i>Rn</i>	F	<i>n</i>	00000000	E
	<i>ERn</i>	F	<i>n</i>	00001000	E
	<i>XRn</i>	F	<i>n</i>	00011000	E
	<i>QRn</i>	F	<i>n</i>	00011111	E

## PUSH register list

Save control registers

### Function

$SP \leftarrow SP - n$

$(SP) \leftarrow \text{Control registers}$

### 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 1.7 “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:

$ELR \rightarrow EPSW \rightarrow LR \rightarrow EA$

- This instruction assumes that preceding PUSH instructions have saved the specified control registers on the stack in the appropriate order.
- 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

$SP \leftarrow SP - n$

$(SP) \leftarrow \text{General registers}$

## 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 odd number of bytes operand ( $Rn$ ) stores the specified register with a dummy byte.  
For further details, see Section 1.7 “Stack Modifications.”

## Flags

C	Z	S	OV	MIE	HC
—	—	—	—	—	—

—: No change

## Instruction Format

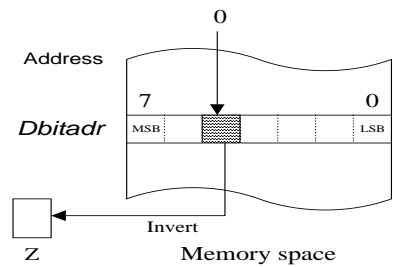
Mnemonic	First operand	Instruction Format			
		First word			
PUSH	$Rn$	F	$n$	0100	E
	$ERn$	F	$n$	0101	E
	$XRn$	F	$n$	0110	E
	$QRn$	F	$n$	0111	E

RB Dbitadr

Reset bit

Function

$Z \leftarrow \sim[Dbitadr]$   
 $[Dbitadr] \leftarrow 0$



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	<i>Dbitadr</i>		A	0	<i>bit</i>	2
	*: <i>Dbitadr</i>	<word>	A	0	<i>bit</i>	2

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

## RB Rn . *bit\_offset*

Reset bit

### Function

$$Z \leftarrow \sim Rn[ \textit{bit\_offset} ]$$

$$Rn[ \textit{bit\_offset} ] \leftarrow 0$$

### 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	<i>Rn.bit_offset</i>		A <i>n</i> 0 <i>bit</i> 2	

RC

Reset carry flag

Function

$C \leftarrow 0$

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

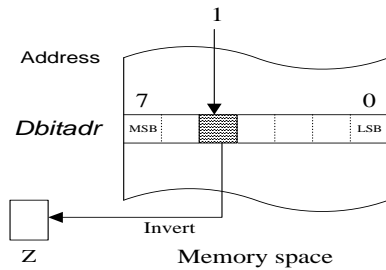
Mnemonic	First operand	Second operand	Instruction Format			
			First word			
RTI			F	E	0	F

# SB Dbitadr

Set bit

## Function

$$Z \leftarrow \sim [Dbitadr]$$

$$[Dbitadr] \leftarrow 1$$


## 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	<i>Dbitadr</i>		A	0	1	<i>Dadr</i>
	*: <i>Dbitadr</i>	<word>	A	0	1	<i>Dadr</i>

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

# SB *Rn . bit\_offset*

Set bit

## Function

$Z \leftarrow \sim Rn[ \textit{bit\_offset} ]$   
 $Rn[ \textit{bit\_offset} ] \leftarrow 1$

## 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	<i>Rn.bit_offset</i>		A <i>n</i> 0    bit    0	

# SC

Set carry flag

## Function

$C \leftarrow 1$

## Description

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

## Flags

C	Z	S	OV	MIE	HC
*	—	—	—	—	—

C: This goes to “1.”

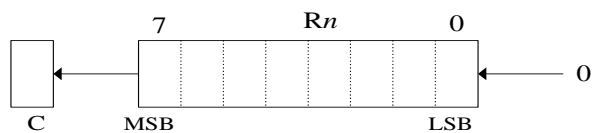
## Instruction Format

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

SLL *Rn* , *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

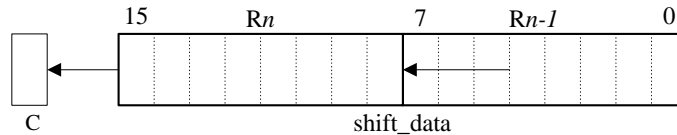
Instruction Format

Mnemonic	First operand	Second operand	Instruction Format			
			First word			Second word
SLL	<i>Rn</i>	<i>Rm</i>	8	<i>n</i>	<i>m</i>	A
		<i>#width</i>	9	<i>n</i>	0 <i>width</i>	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

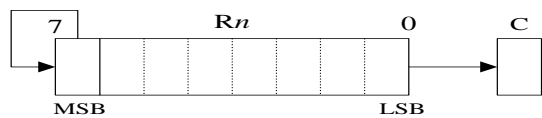
### Instruction Format

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 *Rn* , *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

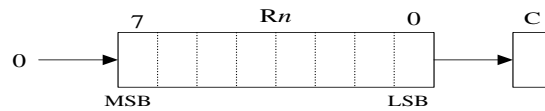
Mnemonic	First operand	Second operand	Instruction Format			
			First word			Second word
SRA	<i>Rn</i>	<i>Rm</i>	8	<i>n</i>	<i>m</i>	E
		<i>#width</i>	9	<i>n</i>	0 <i>width</i>	E



## SRL $Rn$ , $obj$

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

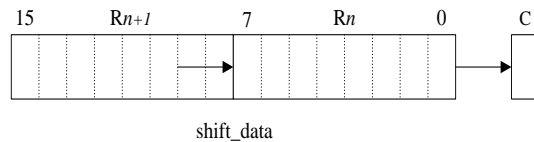
### Instruction Format

Mnemonic	First operand	Second operand	Instruction Format			
			First word			Second word
SRL	$Rn$	$Rm$	8	$n$	$m$	C
		$\#width$	9	$n$	0	$width$

## 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

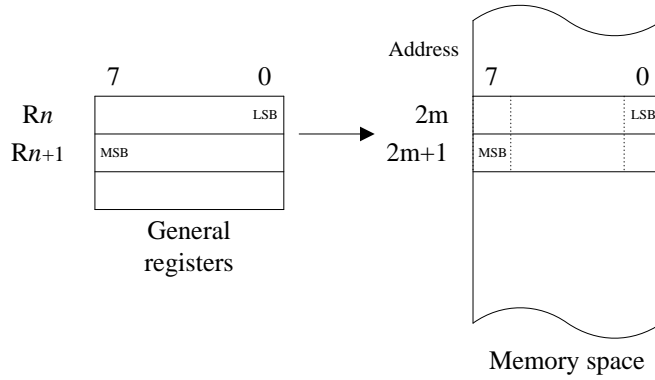
Mnemonic	First operand	Second operand	Instruction Format			
			First word		Second word	
SRLC	$R_n$	$R_m$	8	$n$	$m$	D
		$\#width$	9	$n$	0 $width$	D

# ST ERn , obj

Word-sized data transfer

## Function

$obj \leftarrow ERn$



## 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[ER $m$ ]	<word>	A	$n$	$m$	9
		Disp6[BP]		B	$n$	Disp6	
		*:Disp6[BP]	<word>	B	$n$	Disp6	
		Disp6[FP]		B	$n$	Disp6	
		*:Disp6[FP]	<word>	B	$n$	Disp6	
		Dadr		9	$n$	1	3
		*: Dadr	<word>	9	$n$	1	3

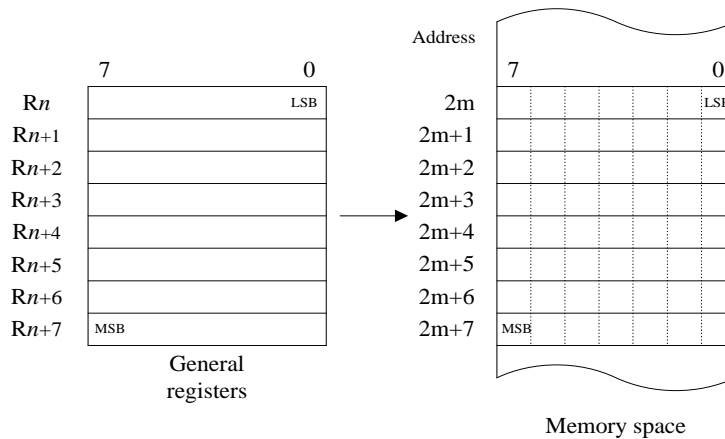
*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

# ST QR<sub>n</sub>, obj

Quad word-sized  
data transfer

## Function

$obj \leftarrow QR_n$



## 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 <sub>n</sub>	[EA]		9 <i>n</i> 3    7	
		*: [EA]	<word>	9 <i>n</i> 3    7	
		[EA+]		9 <i>n</i> 5    7	
		*:[EA+]	<word>	9 <i>n</i> 5    7	

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
Rd	9	0	<i>d</i>	F

## ST $Rn$ , $obj$

Byte-sized data transfer

---

### Function

$obj \leftarrow Rn$

---

### 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	
		[ERm]		9	n	m	1	
		*:[ERm]	<word>	9	n	m	1	
		Disp16[ERm]		9	n	m	9	Disp16
		*:Disp16[ERm]	<word>	9	n	m	9	Disp16
		Disp6[BP]		D	n	0	Disp6	
		*:Disp6[BP]	<word>	D	n	0	Disp6	
		Disp6[FP]		D	n	1	Disp6	
		*:Disp6[FP]	<word>	D	n	1	Disp6	
		Dadr		9	n	1	1	Dadr
		*: Dadr	<word>	9	n	1	1	Dadr

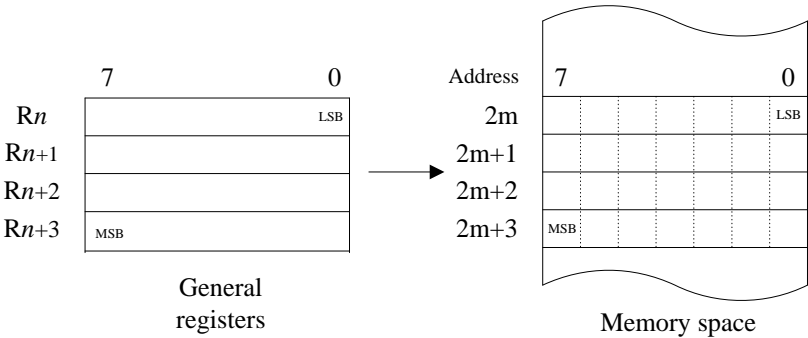
*	<word>			
pseg_addr	E	3	pseg_addr	
DSR	F	E	9	F
Rd	9	0	d	F

# ST XRn , obj

Double word-sized  
data transfer

## Function

$obj \leftarrow XRn$



## 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	XRn	[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
Rd	9	0	d	F



# SUB $Rn, Rm$

Subtract

## Function

$$Rn \leftarrow Rn - Rm$$

## 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	$Rn$	$Rm$	8 $n$ $m$ 8	

# SUBC $Rn, Rm$

Subtract with carry

## Function

$$Rn \leftarrow Rn - Rm - C$$

## 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	$Rn$	$Rm$	8	$n$	$m$	9

# SWI #*snum*

Software interrupt

## Function

EPSW1  $\leftarrow$  PSW  
 ELEVEL  $\leftarrow$  1  
 ELR1  $\leftarrow$  PC+2  
 ECSR1  $\leftarrow$  CSR  
 MIE  $\leftarrow$  0  
 PC  $\leftarrow$  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

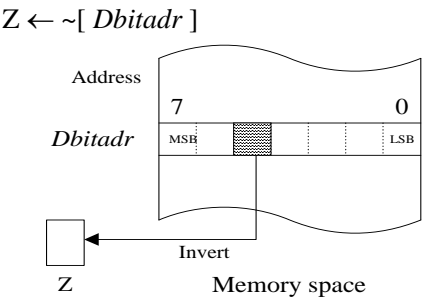
## Instruction Format

Mnemonic	First operand	Instruction Format				
		First word				Second word
SWI	# <i>snum</i>	E	5	0	0	<i>snum</i>

TB Dbitadr

Test bit

Function



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	<i>Dbitadr</i>		A	0	<i>bit</i>	<i>Dadr</i>
	*: <i>Dbitadr</i>	<word>	A	0	<i>bit</i>	<i>Dadr</i>

*	<word>			
<i>pseg_addr</i>	E	3	<i>pseg_addr</i>	
DSR	F	E	9	F
<i>Rd</i>	9	0	<i>d</i>	F

## TB Rn . bit\_offset

Test bit

### Function

$$Z \leftarrow \sim Rn[\text{bit\_offset}]$$

### 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	Rn.bit		A    n    0 bit    1	

# XOR $Rn, obj$

Bitwise exclusive OR

## Function

$$Rn \leftarrow Rn \wedge obj$$

## 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	$Rn$	$Rm$	8 $n$ $m$ 4	
		$\#imm8$	4 $n$ $imm8$	

# 4. Appendix

---

This appendix lists the nX-U16/100 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. Therefore, 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	Rn	Rm	*	*	*	*		*	1000_nnnn_mmmmm_0001		1	
		#imm8	*	*	*	*		*	0001_nnnn_iiii_iiii		1	
ADD	ERn	ERm	*	*	*	*		*	1111_nnn0_mmm0_0110		1	
		#imm7	*	*	*	*		*	1110_nnn0_iiii_iiii		1	
ADDC	Rn	Rm	*	*	*	*		*	1000_nnnn_mmmmm_0110		1	
		#imm8	*	*	*	*		*	0110_nnnn_iiii_iiii		1	
AND	Rn	Rm		*	*				1000_nnnn_mmmmm_0010		1	
		#imm8		*	*				0010_nnnn_iiii_iiii		1	
CMP	Rn	Rm	*	*	*	*		*	1000_nnnn_mmmmm_0111		1	
		#imm8	*	*	*	*		*	0111_nnnn_iiii_iiii		1	
CMPC	Rn	Rm	*	*	*	*		*	1000_nnnn_mmmmm_0101		1	
		#imm8	*	*	*	*		*	0101_nnnn_iiii_iiii		1	
MOV	ERn	ERm		*	*				1111_nnn0_mmm0_0101		1	
		#imm7		*	*				1110_nnn0_0iii_iiii		1	
MOV	Rn	Rm		*	*				1000_nnnn_mmmmm_0000		1	
		#imm8		*	*				0000_nnnn_iiii_iiii		1	
OR	Rn	Rm		*	*				1000_nnnn_mmmmm_0011		1	
		#imm8		*	*				0011_nnnn_iiii_iiii		1	
XOR	Rn	Rm		*	*				1000_nnnn_mmmmm_0100		1	
		#imm8		*	*				0100_nnnn_iiii_iiii		1	
CMP	ERn	ERm	*	*	*	*		*	1111_nnn0_mmm0_0111		1	
SUB	Rn	Rm	*	*	*	*		*	1000_nnnn_mmmmm_1000		1	
SUBC	Rn	Rm	*	*	*	*		*	1000_nnnn_mmmmm_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_ <i>nnnn</i> _ <i>mmmm</i> _1010		1	
		$\#width$	*						1001_ <i>nnnn</i> _0www_1010		1	
SLLC	$R_n$	$R_m$	*						1000_ <i>nnnn</i> _ <i>mmmm</i> _1011		1	
		$\#width$	*						1001_ <i>nnnn</i> _0www_1011		1	
SRA	$R_n$	$R_m$	*						1000_ <i>nnnn</i> _ <i>mmmm</i> _1110		1	
		$\#width$	*						1001_ <i>nnnn</i> _0www_1110		1	
SRL	$R_n$	$R_m$	*						1000_ <i>nnnn</i> _ <i>mmmm</i> _1100		1	
		$\#width$	*						1001_ <i>nnnn</i> _0www_1100		1	
SRLC	$R_n$	$R_m$	*						1000_ <i>nnnn</i> _ <i>mmmm</i> _1101		1	
		$\#width$	*						1001_ <i>nnnn</i> _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	ERn	[EA]		*	*				1001_nnn0_0011_0010		1	
		[EA+]		*	*				1001_nnn0_0101_0010		1	
		[ERm]		*	*				1001_nnn0_nmm0_0010		1	
		Disp16[ERm]		*	*				1010_nnn0_nmm0_1000	DDDD_DDDD_DDDD_DDDD	2	
		Disp6[BP]		*	*				1011_nnn0_00DD_DDDD		2	
		Disp6[FP]		*	*				1011_nnn0_01DD_DDDD		2	
		Dadr		*	*				1001_nnn0_0001_0010	DDDD_DDDD_DDDD_DDDD	2	
	Rn	[EA]		*	*				1001_nnnn_0011_0000		1	
		[EA+]		*	*				1001_nnnn_0101_0000		1	
		[ERm]		*	*				1001_nnnn_nmm0_0000		1	
		Disp16[ERm]		*	*				1001_nnnn_nmm0_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	
	XRn	[EA]		*	*				1001_nn00_0011_0100		2	
		[EA+]		*	*				1001_nn00_0101_0100		2	
	QRn	[EA]		*	*				1001_n000_0011_0110		4	
		[EA+]		*	*				1001_n000_0101_0110		4	

Mnemonic	First operand	Second operand	Flag changes							Instruction code		Minimum execution time (cycles)
			C	Z	S	OV	MIE	HC	First word	Second word		
ST	ERn	[EA]							1001_nnn0_0011_0011		1	
		[EA+]							1001_nnn0_0101_0011		1	
		[ERm]							1001_nnn0_nmm0_0011		1	
		Disp16[ERm]							1010_nnn0_nmm0_1001	DDDD_DDDD_DDDD_DDDD	2	
		Disp6[BP]							1011_nnn0_10DD_DDDD		2	
		Disp6[FP]							1011_nnn0_11DD_DDDD		2	
		Dadr							1001_nnn0_0001_0011	DDDD_DDDD_DDDD_DDDD	2	
	Rn	[EA]							1001_nnnn_0011_0001		1	
		[EA+]							1001_nnnn_0101_0001		1	
		[ERm]							1001_nnnn_nmm0_0001		1	
		Disp16[ERm]							1001_nnnn_nmm0_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	
	XRn	[EA]							1001_nn00_0011_0101		2	
		[EA+]							1001_nn00_0101_0101		2	
	QRn	[EA]							1001_n000_0011_0111		4	
		[EA+]							1001_n000_0101_0111		4	

**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	<i>#signed8</i>							1110_0001_iiii_iiii		1
MOV	ECSR	<i>Rm</i>							1010_0000_iiii_iiii		1
	ELR	<i>ERm</i>							1010_iiii0_0000_1101		1
	EPSW	<i>Rm</i>							1010_0000_iiii_iiii		1
	<i>ERn</i>	ELR							1010_iiii0_0000_0101		1
		SP							1010_iiii0_0001_1010		1
	PSW	<i>Rm</i>	*	*	*	*	*	*	1010_0000_iiii_iiii		1
		<i>#unsigned8</i>	*	*	*	*	*	*	1110_1001_iiii_iiii		1
	<i>Rn</i>	ECSR							1010_iiii0_0000_0111		1
		EPSW							1010_iiii0_0000_0100		1
		PSW							1010_iiii0_0000_0011		1
	SP	<i>ERm</i>							1010_0001_iiii_iiii		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	<i>ERn</i>								1111_iiii0_0101_1110		1
	<i>QRn</i>								1111_iiii0_0111_1110		4
	<i>Rn</i>								1111_iiii0_0100_1110		1
	<i>XRn</i>								1111_iiii0_0110_1110		2
	<i>register_list</i>								1111_1111_1100_1110		1-6
POP	<i>ERn</i>								1111_iiii0_0001_1110		1
	<i>QRn</i>								1111_iiii0_0011_1110		4
	<i>Rn</i>								1111_iiii0_0000_1110		1
	<i>XRn</i>								1111_iiii0_0010_1110		2
	<i>register_list</i>		*	*	*	*	*	*	1111_1111_1000_1110		1-9

**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 <i>n</i>	R <i>m</i>							1010_ <i>nnnn</i> _ <i>mmmm</i> _1110		1
	CER <i>n</i>	[EA]							1111_ <i>nnn</i> 0 _0010 _1101		1
		[EA+]							1111_ <i>nnn</i> 0 _0011 _1101		1
	CR <i>n</i>	[EA]							1111_ <i>nnnn</i> _0000 _1101		1
		[EA+]							1111_ <i>nnnn</i> _0001 _1101		1
	CXR <i>n</i>	[EA]							1111_ <i>nn</i> 00 _0100 _1101		2
		[EA+]							1111_ <i>nn</i> 00 _0101 _1101		2
	CQR <i>n</i>	[EA]							1111_ <i>n</i> 000 _0110 _1101		4
		[EA+]							1111_ <i>n</i> 000 _0111 _1101		4
	R <i>n</i>	CR <i>m</i>							1010_ <i>nnnn</i> _ <i>mmmm</i> _0110		1
	[EA]	CER <i>m</i>							1111_ <i>mmmm</i> _0101 _1101		1
	[EA+]	CER <i>m</i>							1111_ <i>mmmm</i> _0101 _1101		1
	[EA]	CR <i>m</i>							1111_ <i>mmmm</i> _1000 _1101		1
	[EA+]	CR <i>m</i>							1111_ <i>mmmm</i> _1001 _1101		1
	[EA]	CXR <i>m</i>							1111_ <i>mm</i> 00 _1100 _1101		2
	[EA+]	CXR <i>m</i>							1111_ <i>mm</i> 00 _1101 _1101		2
[EA]	CQR <i>m</i>							1111_ <i>m</i> 000 _1110 _1101		4	
[EA+]	CQR <i>m</i>							1111_ <i>m</i> 000 _1111 _1101		4	

**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	[ERm]								1111_0000_mmm0_1010		1	
	Disp16[ERm]								1111_0000_mmm0_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 <sub>n</sub>		*	*	*			*	1000_ <i>nnnn</i> _0001_1111		1	
DAS	R <sub>n</sub>		*	*	*			*	1000_ <i>nnnn</i> _0011_1111		1	
NEG	R <sub>n</sub>		*	*	*	*		*	1000_ <i>nnnn</i> _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	<i>Rn.bit_offset</i>			*					1010_0nnnn_0bbb_0000		1	
	<i>Dbitadr</i>			*					1010_0000_1bbb_0000	DDDD_DDDD_DDDD_DDDD	2	
RB	<i>Rn.bit_offset</i>			*					1010_0nnnn_0bbb_0010		1	
	<i>Dbitadr</i>			*					1010_0000_1bbb_0010	DDDD_DDDD_DDDD_DDDD	2	
TB	<i>Rn.bit_offset</i>			*					1010_0nnnn_0bbb_0001		1	
	<i>Dbitadr</i>			*					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		A34 core: 1/3 (*1)  A35 core 1/2 (*1)  3	
BLT									1100_0001_rrrr_rrrr			
BGT									1100_0010_rrrr_rrrr			
BLE									1100_0011_rrrr_rrrr			
BGES									1100_0100_rrrr_rrrr			
BLTS									1100_0101_rrrr_rrrr			
BGTS									1100_0110_rrrr_rrrr			
BLES									1100_0111_rrrr_rrrr			
BNE									1100_1000_rrrr_rrrr			
BEQ									1100_1001_rrrr_rrrr			
BNV									1100_1010_rrrr_rrrr			
BOV									1100_1011_rrrr_rrrr			
BPS									1100_1100_rrrr_rrrr			
BNS									1100_1101_rrrr_rrrr			
BAL									1100_1110_rrrr_rrrr			

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

**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 <sub>n</sub>			*	*				1000_nnn1_nnn0_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 <sub>n</sub>								1111_0000_nnn0_0010		2
BL	Cadr								1111_gggg_0000_0001	cccc_cccc_cccc_cccc	2
	ER <sub>n</sub>								1111_0000_nnn0_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 <sub>n</sub>	R <sub>m</sub>		*					1111_nnn0_nnnnnn_0100		9
DIV	ER <sub>n</sub>	R <sub>m</sub>		*	*				1111_nnn0_nnnnnn_1001		17

**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.	Issue Date	Page		Description
		Previous Edition	Current Edition	
PEUL-U16-100-INST-02	July 2013	—	—	2nd edition.
PEUL-U16-100-INST-03	Jan.2015	P1-1	P1-1	Added explanation of the kind of the CPU core.
		P3-12 P4-5	P3-12 P4-5	The statement of the number of execution cycles of a conditional branch instruction was divided for every kind of CPU.