PanaXSeries

The One to Watch for Constant Innovation-Making the Future Come Alive

MICROCOMPUTER MN1030/MN103S

MN1030/MN103S Series

Instruction Manual

Pub.No.13250-040E

Panasonic

Request for your special attention and precautions in using the technical information and semiconductors described in this book

(1) An export permit needs to be obtained from the competent authorities of the Japanese Government if any of the products or technologies described in this book and controlled under the "Foreign Exchange and Foreign Trade Law" is to be exported or taken out of Japan.

(2) The technical information described in this book is limited to showing representative characteristics and applied circuits examples of the products. It neither warrants non-infringement of intellectual property right or any other rights owned by our company or a third party, nor grants any license.

(3) We are not liable for the infringement of rights owned by a third party arising out of the use of the product or technologies as described in this book.

(4) The products described in this book are intended to be used for standard applications or general electronic equipment (such as office equipment, communications equipment, measuring instruments and household appliances).
Consult our sales staff in advance for information on the following applications:
- Special applications (such as for airplanes, aerospace, automobiles, traffic control equipment, combustion equipment, life support systems and safety devices) in which exceptional quality and reliability are required, or if the failure or malfunction of the products may directly jeopardize life or harm the human body.
- Any applications other than the standard applications intended.

(5) The products and product specifications described in this book are subject to change without notice for modification and/or improvement. At the final stage of your design, purchasing, or use of the products, therefore, ask for the most up-to-date Product Standards in advance to make sure that the latest specifications satisfy your requirements.

(6) When designing your equipment, comply with the guaranteed values, in particular those of maximum rating, the range of operating power supply voltage, and heat radiation characteristics. Otherwise, we will not be liable for any defect which may arise later in your equipment.
Even when the products are used within the guaranteed values, take into the consideration of incidence of break down and failure mode, possible to occur to semiconductor products. Measures on the systems such as redundant design, arresting the spread of fire or preventing glitch are recommended in order to prevent physical injury, fire, social damages, for example, by using the products.

(7) When using products for which damp-proof packing is required, observe the conditions (including shelf life and amount of time let standing of unsealed items) agreed upon when specification sheets are individually exchanged.

(8) This book may be not reprinted or reproduced whether wholly or partially, without the prior written permission of Matsushita Electric Industrial Co., Ltd.

If you have any inquiries or questions about this book or our semiconductors, please contact one of our sales offices listed at the back of this book.

About This Manual

This document contains a detailed description of the instruction set for the MN1030/MN103S Series.
This document concentrates on the AM32 microcontroller core. When the specifications differ between cores, separate descriptions appear next to icons indicating the appropriate core or cores
Chapter 1 provides an overview of the instruction set--instruction functions, formats, and the like.
Chapter 2 contains detailed descriptions of the individual instructions--operation, effect on PSW flags, and the like.
Chapter 3 contains usage notes--a description of the pipeline architecture, programming notes, usage recommendations, and the like.
The Appendix contains charts for the entire instruction set and instruction mappings.

■ Finding Information

This document incorporates the following aids for locating necessary information as quickly as possible.

(1) Index tabs in the inside margins of left-hand pages indicate Chapters.

(2) The table of contents near the beginning of this document lists section headings.

(3) As you flip through the document, the page header gives the chapter; the footer, the section heading.

(4) The index near the end of this document lists page references for all instructions and instruction variants. In Chapter 2, the instruction mnemonic appears in the page footer for right-hand pages.

■ Related Manuals

The following related manuals are available. Please contact our sales representative for more details.

< For MN1030 Series Users >

MN1030 Series Cross Assembler User's Manual
<Describes the assembler syntax and notation>

MN1030 Series C Compiler User's Manual: Usage Guide
<Describes the installation, the commands, and options of the C Compiler>

MN1030/MN103S/MN103E Series C Compiler User's Manual: Language Description
<Describes the syntax of the C Compiler>

MN1030/MN103S/MN103E Series C Compiler User's Manual: Library Reference
<Describes the the standard library of the C Compiler>

MN1030/MN103S Series C Source Code Debugger for Windows® User's Manual
<Describes the use of the C source code debugger for Windows®>

MN1030/MN103S Series Installation Manual
<Describes the installation of the C compiler, cross-assembler and C source code debugger and the procedure for bringing up the in-circuit emulator>

< For MN103S Series Users >

MN103S Series Cross Assembler User's Manual

<Describes the assembler syntax and notation>

MN103S Series C Compiler User's Manual: Usage Guide

<Describes the installation, the commands, and options of the C Compiler>

MN1030/MN103S/MN103E Series C Compiler User's Manual: Language Description

<Describes the syntax of the C Compiler>

MN1030/MN103S/MN103E Series C Compiler User's Manual: Library Reference

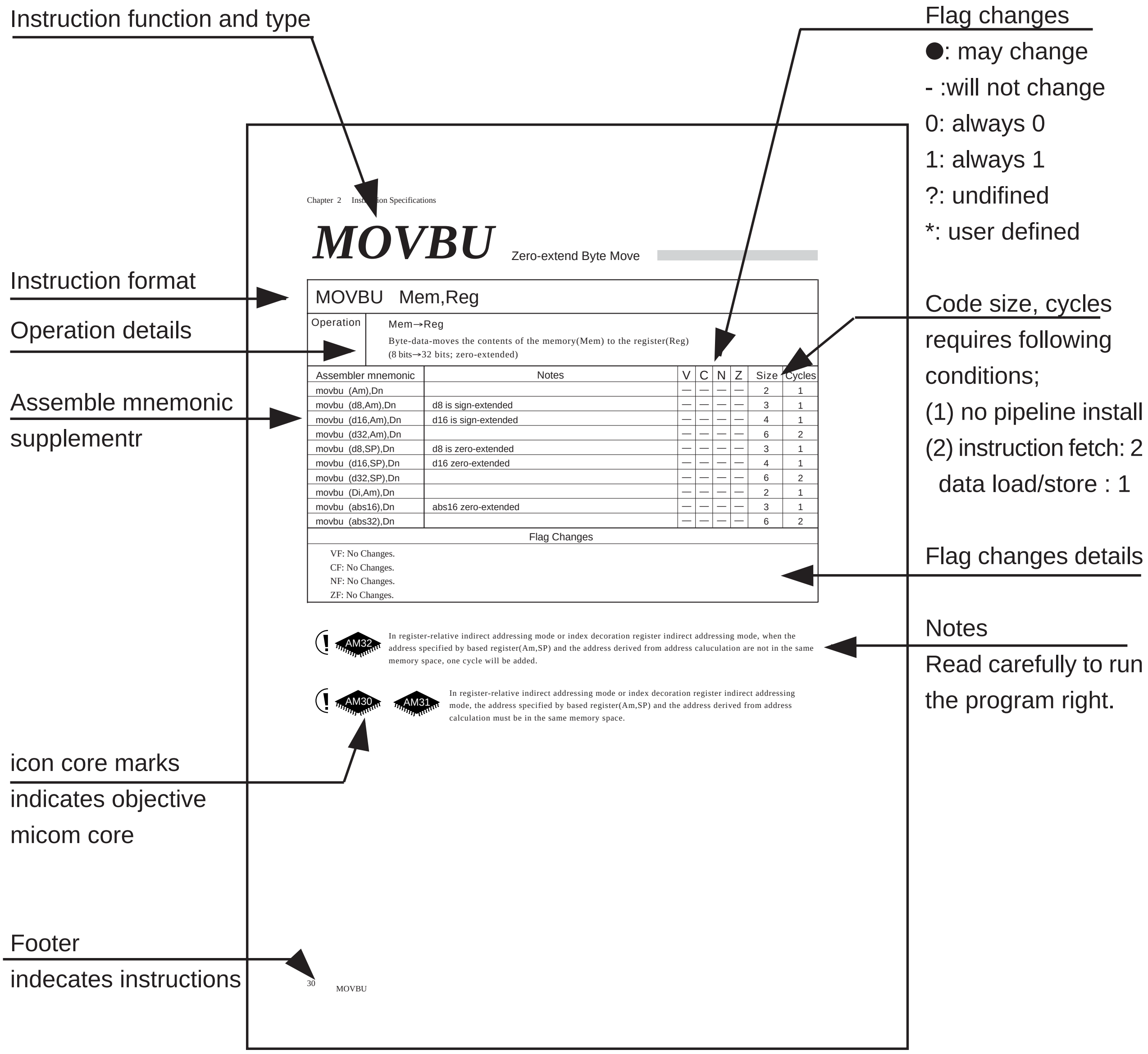
<Describes the the standard library of the C Compiler>

MN1030/MN103S Series C Source Code Debugger for Windows® User's Manual

<Describes the use of the C source code debugger for Windows®>

MN1030/MN103S Series Installation Manual

<Describes the installation of the C compiler, cross-assembler and C source code debugger and the procedure for bringing up the in-circuit emulator>


Instruction function and type
Flag changes
●: may change
- :will not change
0: always 0
1: always 1
?: undifined
*: user defined
Chapter 2 Instruction Specifications
MOVBU
Zero-extend Byte Move
Instruction format
MOVBU Mem,Reg
Operation details
Operation
Mem→Reg
Byte-data-moves the contents of the memory(Mem) to the register(Reg)
(8 bits→32 bits; zero-extended)
Code size, cycles
requires following
conditions;
(1) no pipeline install
(2) instruction fetch: 2
data load/store : 1
Assemble mnemonic
supplementr
Assembler mnemonic
Notes
V
C
N
Z
Size
Cycles
movbu (Am),Dn
movbu (d8,Am),Dn
d8 is sign-extended
movbu (d16,Am),Dn
d16 is sign-extended
movbu (d32,Am),Dn
movbu (d8,SP),Dn
d8 is zero-extended
movbu (d16,SP),Dn
d16 zero-extended
movbu (d32,SP),Dn
movbu (Di,Am),Dn
movbu (abs16),Dn
abs16 zero-extended
movbu (abs32),Dn
Flag Changes
VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.
Flag changes details
AM32
In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(Am,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.
Notes
Read carefully to run
the program right.
AM30
AM31
In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(Am,SP) and the address derived from address calculation must be in the same memory space.
icon core marks
indicates objective
micom core
Footer
indecates instructions
30
MOVBU

■ Page Layouts

The three layouts below are the standards for the three Chapters.
Chapter 1 pages give the section title, an overview, the main text, and notes.
Chapter 2 pages give the instruction syntax, operational description, and notes.
Chapter 3 pages feature such items as pipeline operation diagrams, code samples, and notes.

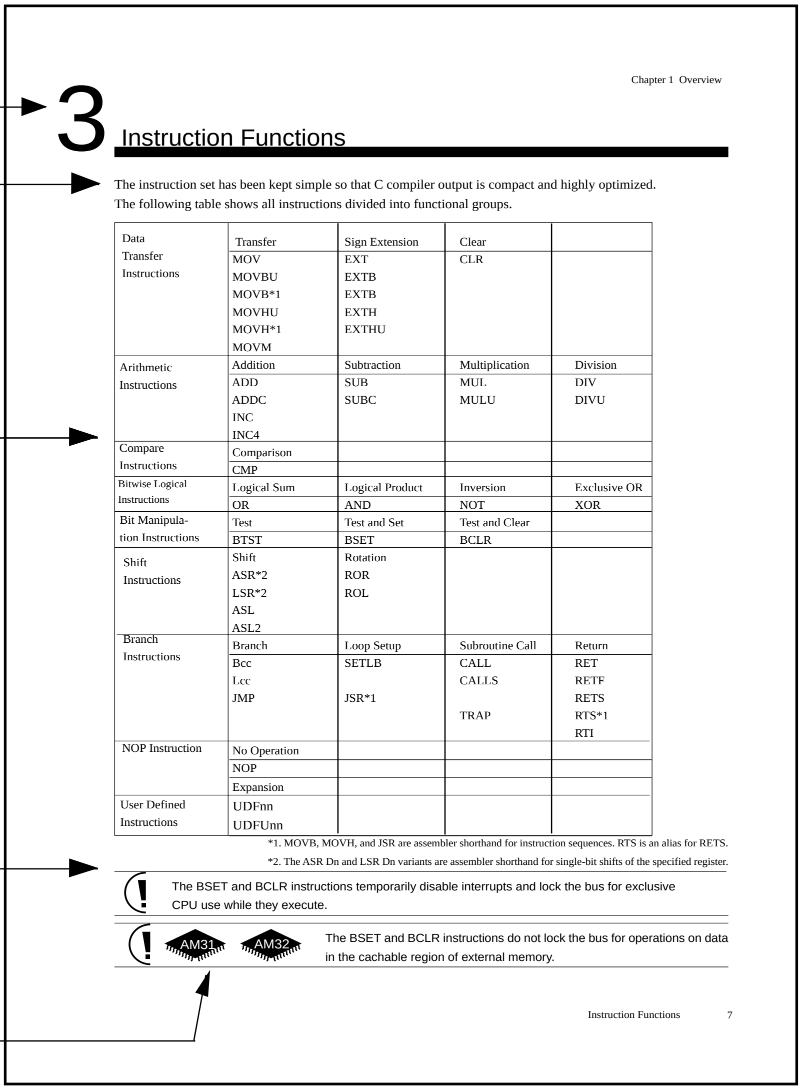
Section title

Overview
(First page of section only)

Table Summary of
section contents

Notes Observe
these precautions to ensure
reliable program operation.

Core icons
These indicate the applicable
microcontroller cores.



Chapter 1 Overview

3 Instruction Functions

The instruction set has been kept simple so that C compiler output is compact and highly optimized.
The following table shows all instructions divided into functional groups.

Data Transfer Instructions	Transfer	Sign Extension	Clear	
	MOV MOVBU MOVB*1 MOVHU MOVH*1 MOVM	EXT EXTB EXTB EXTH EXTHU	CLR	
Arithmetic Instructions	Addition	Subtraction	Multiplication	Division
	ADD ADDC INC INC4	SUB SUBC	MUL MULU	DIV DIVU
Compare Instructions	Comparison			
	CMP			
Bitwise Logical Instructions	Logical Sum	Logical Product	Inversion	Exclusive OR
	OR	AND	NOT	XOR
Bit Manipula-tion Instructions	Test	Test and Set	Test and Clear	
	BTST	BSET	BCLR	
Shift Instructions	Shift	Rotation		
	ASR*2 LSR*2 ASL ASL2	ROR ROL		
Branch Instructions	Branch	Loop Setup	Subroutine Call	Return
	Bcc Lcc JMP	SETLB JSR*1	CALL CALLS TRAP	RET RETF RETS RTS*1 RTI
NOP Instruction	No Operation			
	NOP			
	Expansion			
User Defined Instructions	UDFnn UDFUnn			

*1. MOVB, MOVH, and JSR are assembler shorthand for instruction sequences. RTS is an alias for RETS.
*2. The ASR Dn and LSR Dn variants are assembler shorthand for single-bit shifts of the specified register.

The BSET and BCLR instructions temporarily disable interrupts and lock the bus for exclusive CPU use while they execute.

AM31 AM32 The BSET and BCLR instructions do not lock the bus for operations on data in the cachable region of external memory.

Instruction Functions 7

MN1030 Series

AM30: First generation

AM31: Second generation

MN103S Series

AM32: Third generation

Expalanation for Code Sequences to avoid

Description

Pipeline architecture

Example of problematic program

Example of revised program

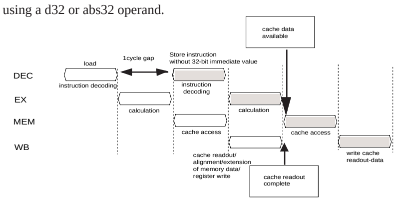
Applicable instructions

Notes
Read carefully to run the program right

(3) Cachable Memory, DCBYPS = 0, Follower Storing without d32/abs32

[Description]

To prevent pipeline stalls, we recommend inserting at least one cycle when the DCBYPS bit is “0,” the leading instruction accesses cachable external memory, and the following instruction stores the data without using a d32 or abs32 operand.



[Example]

[Problematic Version]

```
inc     a1
mov     (a0),d0     Load instruction
mov     d0,(a1)     Instruction storing loaded data without d32/abs32 operand
```

Retrieving and aligning the data from the cache takes one cycle, so the pipeline stalls until the data loaded with the first MOV instruction is available to the second MOV instruction.

[Revised Version]

```
mov     (a0),d0     Load instruction
inc     a1
mov     d0,(a1)     Instruction storing loaded data without d32/abs32 operand
```

The data loaded with the first MOV instruction is available to the second MOV instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction>	All load variants of MOV, MOVBU, and MOVHU
<Following instruction>	MOV, MOVBU, MOVB, MOVHU, and MOVH store without d32/abs32 operands

Here cachable external memory refers to the cachable portions of AM31 or AM32 external memory.

Table of Contents

Chapter 1 Overview

Chapter 2 Instruction Specifications

Bit Manipulation Instructions

Shift Instructions

Branch Instructions

NOP Instruction

User Defined Instructions

Chapter 3 Usage Notes

Appendix

INDEX

1 Overview

1 Instruction Set

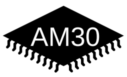
The MN1030/MN103S Series of 32-bit microcontrollers has a simple instruction set designed to make C compiler output compact and highly optimized. It minimizes code size by adopting a variable length instruction format with a basic instruction length of only one byte. It is thus able to minimize increases in assembler program code size even though the only data transfers supported by the simple instruction set are load and store.

CPU Cores

AM30, AM31, and AM32 are 32-bit embedded application microcontroller cores from the Matsushita AM Series of C-oriented 8-, 16-, and 32-bit microcontrollers. Their specifications differ for certain instructions. The following are brief overviews of these three cores.

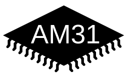
MN1030 Series

AM30: First-generation microcontroller core



supporting connection to ROM, RAM, and Flash memory for instructions and RAM for data.

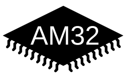
AM31: Second-generation microcontroller core



supporting connection to cache memory for both instructions and data.
General-purpose microcomputer based on this core:
MN103002A

MN103S Series

AM32: Third-generation microcontroller core



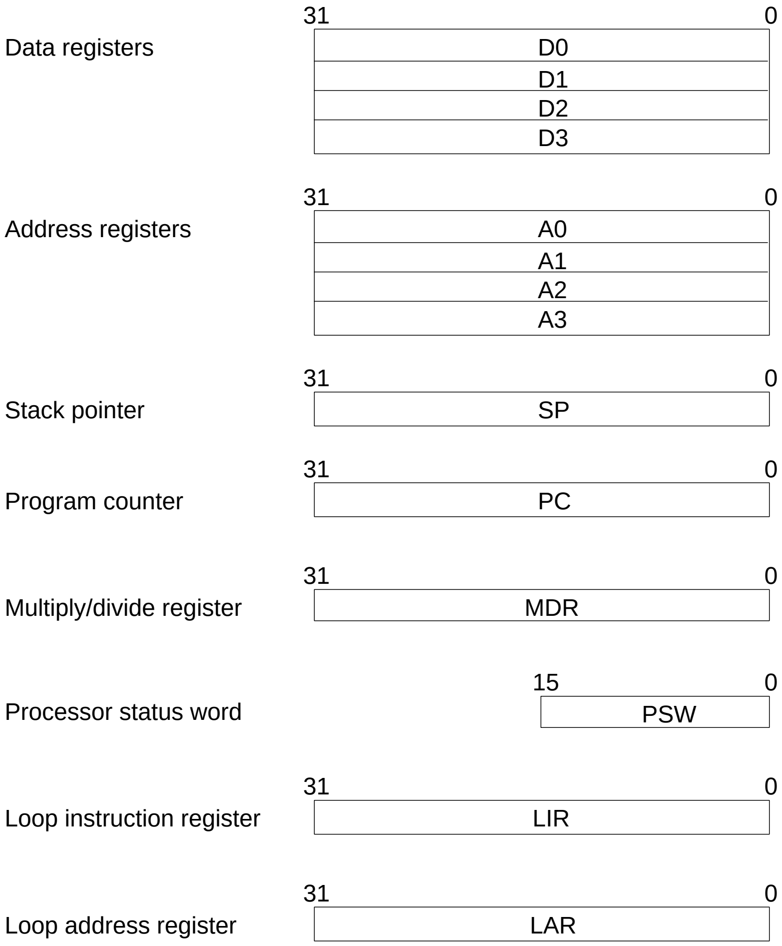
supporting connection to ROM, RAM, Flash memory, and cache memory for instructions and RAM and cache memory for data.

This document concentrates on the AM32 microcontroller core. When the specifications differ between cores, separate descriptions appear next to icons indicating the appropriate core or cores.

2 Register Set

The register set includes data registers for arithmetic and general use, address registers for use as pointers, and the stack pointer. This set greatly contributes to increasing the internal architecture's performance by reducing code size and boosting parallel use of pipeline stages.

This register set incorporates features enabling the use of C and other high-level languages.



The Loop Instruction Register (LIR) and Loop Address Register (LAR) are for speeding up the branch to and execution of the first instruction in a loop. The SETLB (Set Loop Beginning) instruction loads them with the next four instruction bytes and the address of the fifth, respectively. The Lcc (Loop) instruction then uses these stored values to jump-start execution of the first instruction in the loop while fetching additional instruction bytes.

2.1 Data Registers

D0 to D3: Data Registers (32 bits x 4)

These four 32-bit registers are for arithmetic and general use. Data values are automatically zero-extended to 32 bits when they are loaded from memory. The EXTB and EXTH instructions are also available for sign-extending them once loaded.
For 8-bit data, a load operation copies the data from memory into the lowest eight bits of the register and zeros the other bits. A store copies the lowest eight bits of the register to memory. Following the load operation with an EXTB instruction sign-extends it from 8 bits to 32.
For 16-bit data, a load operation copies the data from memory into the lowest 16 bits of the register and zeros the other bits. A store copies the lowest 16 bits of the register to memory. Following the load operation with an EXTH instruction sign-extends it from 16 bits to 32.

2.2 Address Registers

A0 to A3: Address Registers (32 bits x 4)

These four 32-bit registers are for use as address pointers, so support only the operations relevant to address calculations: addition, subtraction, and comparison.
Because the contents are pointers, transfers to and from memory are always 32 bits wide.

2.3 Stack Pointer

SP: Stack Pointer (32 bits x 1)

This 32-bit pointer indicates the address at the top of the stack.
Because addressing is by the word, the lowest two bits of any value loaded into this register must be '00'--that is, a multiple of four.

2.4 Program Counter

PC: Program Counter (32 bit x1)

This 32-bit register holds the address of the instruction currently executing.

2.5 Multiply/Divide Register

MDR: Multiply/ Divide Register (32 bits x 1)

This 32-bit register is for use by multiply and divide instructions. After a multiply operation, it holds the top 32 bits of the 64-bit result. After a divide operation, it holds the 32-bit remainder; before, the top 32 bits of the 64-bit dividend.

2.6 Processor Status Word

PSW: Processor Status Word (16 bits x 1)

This 16-bit register displays CPU status and controls certain operations. Examples of the former function include the flag bits indicating calculation results; of the latter, the interrupt mask level bits.

15															0
0	0	S1	S0	IE	IM2	IM1	IM0	0	0	0	0	V	C	N	Z

Z: Zero flag
This bit goes to "1" if the calculation leaves "0" in all bits of the result and to "0" otherwise. After a reset, it is "0."

N: Negative flag
This bit goes to "1" if the calculation leaves "1" in the most significant bit (MSB) of the result and to "0" otherwise. After a reset, it is "0."

C: Carry flag
This bit goes to "1" if the calculation produces a carry out of or borrow into the most significant bit (MSB) of the result and to "0" otherwise. After a reset, it is "0."

V: Overflow flag
This bit goes to "1" if the result exceeds the bounds for signed integers and to "0" otherwise. After a reset, it is "0."

IM2 to IM0: Interrupt mask level
These three bits offer a choice of eight interrupt mask levels from 0 (000B) to 7 (111B). The hardware accepts only interrupt requests with levels higher than the specified value, and, when it accepts one, sets these bits to the interrupt request level to block subsequent interrupt requests at that and lower levels until interrupt processing is complete. After a reset, all bits are "0" for an interrupt mask level of 0.

IE: Interrupt Enable
This control bit is normally "1" to enable interrupts. When the hardware accepts an interrupt request, however, this bit goes to "0" to disable further interrupts. To support nested interrupts, the user application program must, therefore, reset this bit to "1." After a reset, it is "0."

S1 to S0: Software Bits
These two bits are for operating system use in controlling software. They are not for use by user application programs. After a reset, they are both "0."

2.7 Loop Instruction Register

LIR: Loop Instruction Register (32 bits x 1)

This 32-bit register, used only by the SETLB (Set Loop Beginning) and Lcc (Loop) instructions, holds the first four instruction bytes of the loop for use in speeding up iterations. The SETLB instruction loads it prior to the loop, and the Lcc instruction at the end of the loop then executes the copy while the pipeline fetches more instruction bytes starting from the fifth.
For further details, see the SETLB description in Chapter 2.

2.8 Loop Address Register

LAR: Loop Address Register (32 bit x 1)

This 32-bit register, used only by the SETLB (Set Loop Beginning) and Lcc (Loop) instructions, holds the address of the fifth instruction byte of the loop.

3 Instruction Functions

The instruction set has been kept simple so that C compiler output is compact and highly optimized. The following table shows all instructions divided into functional groups.

Data Transfer Instructions	Transfer	Sign Extension	Clear	
	MOV	EXT	CLR	
	MOVBU	EXTB		
	MOVB*1	EXTB		
	MOVHU	EXTH		
	MOVH*1	EXTHU		
	MOVM			
Arithmetic Instructions	Addition	Subtraction	Multiplication	Division
	ADD	SUB	MUL	DIV
	ADDC	SUBC	MULU	DIVU
	INC			
	INC4			
Compare Instructions	Comparison			
	CMP			
Bitwise Logical Instructions	Logical Sum	Logical Product	Inversion	Exclusive OR
	OR	AND	NOT	XOR
Bit Manipulation Instructions	Test	Test and Set	Test and Clear	
	BTST	BSET	BCLR	
Shift Instructions	Shift	Rotation		
	ASR*2	ROR		
	LSR*2	ROL		
	ASL			
	ASL2			
Branch Instructions	Branch	Loop Setup	Subroutine Call	Return
	Bcc	SETLB	CALL	RET
	Lcc		CALLS	RETF
	JMP	JSR*1		RETS
			TRAP	RTS*1
				RTI
NOP Instruction	No Operation			
	NOP			
	Expansion			
User Defined Instructions	UDFnn			
	UDFUnn			

*1. MOVB, MOVH, and JSR are assembler shorthand for instruction sequences. RTS is an alias for RETS.

*2. The ASR Dn and LSR Dn variants are assembler shorthand for single-bit shifts of the specified register.

The BSET and BCLR instructions temporarily disable interrupts and lock the bus for exclusive CPU use while they execute.

AM31 AM32 The BSET and BCLR instructions do not lock the bus for operations on data in the cachable region of external memory.

3.1 Data Transfer Instructions

Data transfer instructions copy data between registers or between a register and memory. They fall into three groups: MOV, EXT, and CLR.
The MOV group offers a variety of modes for addressing data and provides sign- and zero-extension as necessary for displacements, immediate values, etc.
The EXT group provides sign- and zero-extension within the specified register or to the Multiply/Divide Register (MDR).
The CLR instruction sets the specified register to zero.

Instruction	Description
MOV	Word (32-bit) transfer between registers, word transfer between a register and memory, or loading of an immediate value into a register
MOVBU	Byte transfer between registers with zero-extension for loads
MOVB*1	Byte transfer between registers with sign-extension for loads
MOVHU	Half-word (16-bit) transfer between registers with zero-extension for loads
MOVH*1	Half-word (16-bit) transfer between registers with sign-extension for loads
MOVM	Multiregister transfer to and from stack in memory
EXT	Sign-extension of 32-bit word register into Multiply/Divide Register (MDR)
EXTB	Sign-extension of byte to 32 bits
EXTBU	Zero-extension of byte to 32 bits
EXTH	Sign-extension of half-word to 32 bits
EXTHU	Zero-extension of half-word to 32 bits
CLR	Register clear

*1. MOVB and MOVH are assembler shorthand for instruction sequences.

3.2 Arithmetic Instructions

Arithmetic instructions perform an arithmetic operation on the two source operands (or one), store the result in a register, and--except for INC and INC4 with address registers, ADD with the Stack Pointer (SP), etc.--update the PSW flags according to the result. Because of their frequent use in address calculations, there are separate instructions for incrementing by 1 and 4.

Instruction	Description
Addition	Addition with carry
Subtraction	Subtraction with carry
Multiplication (signed)	Multiplication (unsigned)
Division (signed)	Division (unsigned)

3.3 Compare Instructions

The compare instructions subtract an immediate value or the contents of a register from the contents of another register, setting PSW flags for use in conditional branch instructions.

Instruction	Description
CMP	Comparison

3.4 Bitwise Logical Instructions

Bitwise logical instructions perform a logical operation on the two source operands (or one), store the result in a register, and update the PSW flags according to the result.

Instruction	Description
AND	Logical Product
OR	Logical Sum
XOR	Exclusive OR
NOT	Inversion (ones complement)

3.5 Bit Manipulation Instructions

Bit manipulation instructions perform logical operations on the two source operands--an immediate value and a register, an immediate value and a memory location, a register and a memory location--and update the PSW flags according to the result.

Instruction	Description
BTST	Bit test
BSET	Bit test and set (byte)
BCLR	Bit test and clear (byte)

3.6 Shift Instructions

Shift instructions shift or rotate the specified register by the specified (or implied) amount and update the PSW flags according to the result.

Instruction	Description
ASR*2	Arithmetic shift right
LSR*2	Logical shift right
ASL	Arithmetic shift left
ASL2	Arithmetic 2-bit shift left
ROR	Single-bit rotation right
ROL	Single-bit rotation left

*2. The ASR Dn and LSR Dn variants are assembler shorthand for single-bit shifts of the specified register.

3.7 Branch Instructions

Branch instructions change the flow of execution. In addition to the usual conditional branch (Bcc) instruction, there is a separate variant (Lcc) for use in loops. The latter relies on special registers to reduce the penalty normally associated with taking the branch and thus speed up loop execution. The subroutine call and return instructions feature high-performance specifications that automatically take care of manipulating the Program Counter (PC), saving the appropriate registers to and restoring them from the stack, and securing and releasing the necessary stack space.

Instruction	Description
Bcc	Conditional branch (relative to PC)
Lcc	Loop conditional branch (relative to PC)
SETLB	Loop setup
JMP	Unconditional branch (relative to PC or register indirect)
CALL	Subroutine call (high-performance variant)
CALLS	Subroutine call
RET	Return from subroutine (high-performance variant)
RETF	Return from subroutine (high-performance, high-speed variant)
RETS	Return from subroutine
JSR*3	Subroutine call
RTS*3	Return from subroutine
RTI	Return from interrupt handler
TRAP	Subroutine call to predetermined address

*3. JSR is assembler shorthand for an instruction sequence.

*4. RTS is an alias for RETS.

3.8 NOP Instruction

The NOP instruction does nothing but consume one cycle. It does not affect any resources.

Instruction	Description
NOP	No Operation

3.9 User Defined Instructions

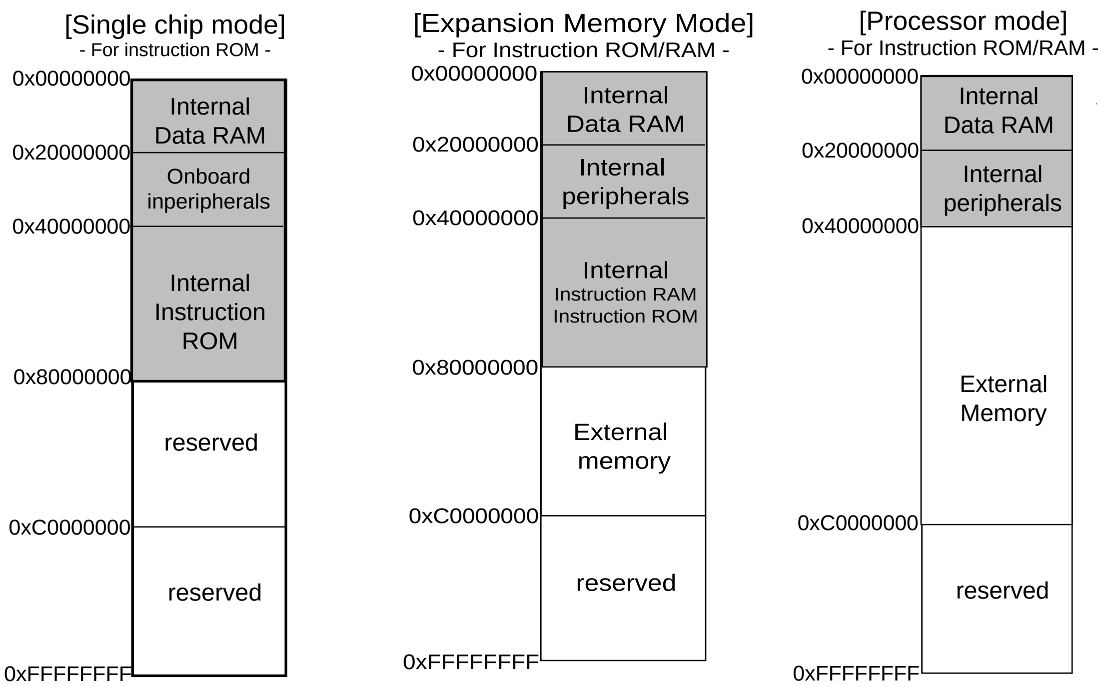
User defined instructions access add-on expansion units. They have a fixed format and reserved positions in the instruction mapping. For further details, refer to the documentation for the particular device.

Instruction	Description
UDFnn	User defined instruction (with sign extension)
UDFUnn	User defined instruction (with zero extension)

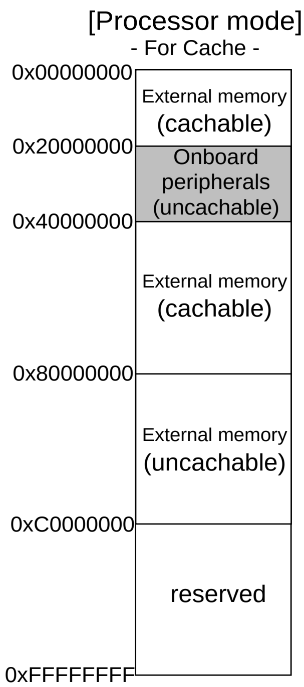
4 Memory Layout

The MN1030/MN103S Series of 32-bit microcontrollers has a 4-gigabyte linear address space. Memory assignments within this address space follow the patterns below. Note how the memory map varies with such factors as internal memory configuration and memory mode. One assignment that is common throughout, however, is the location of the reset vector. It is always at 0x40000000.

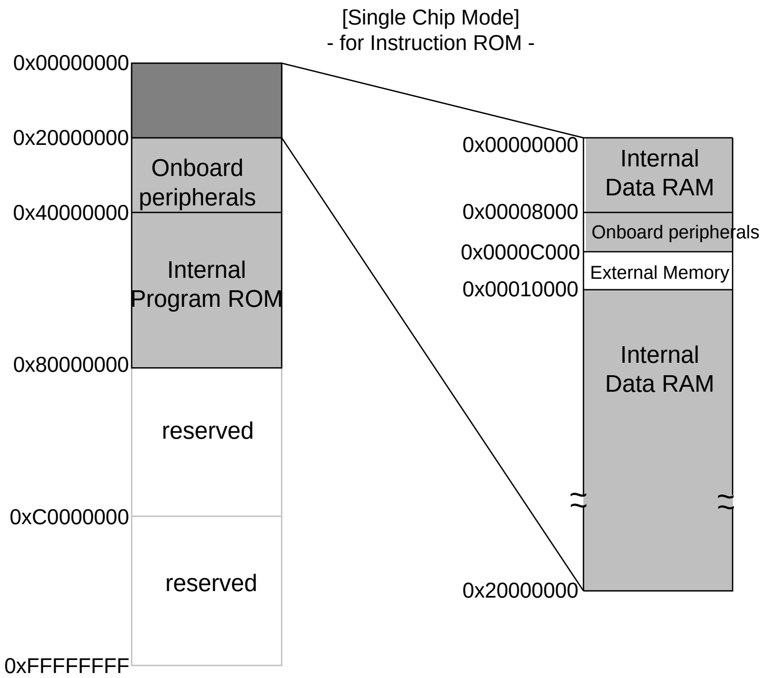
■ AM30

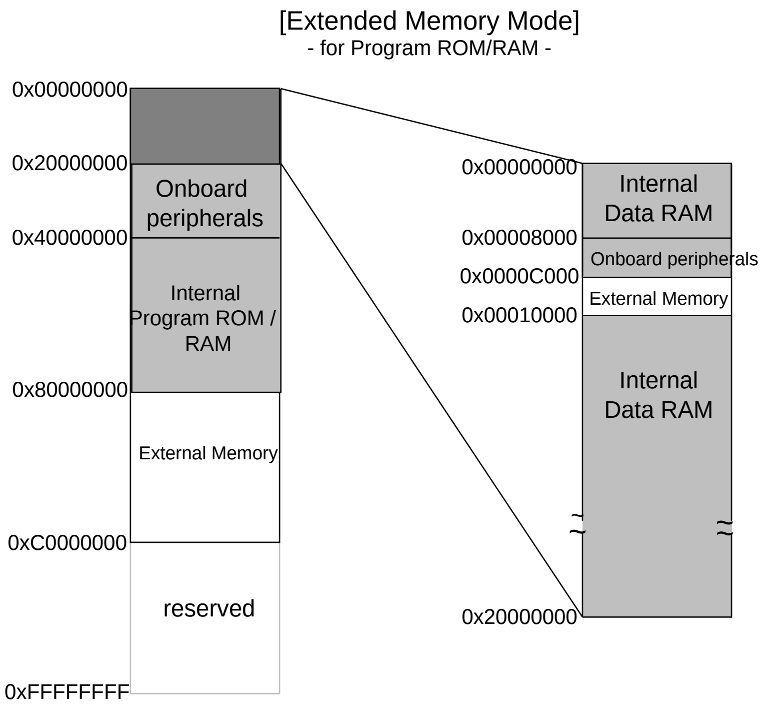


■ AM31

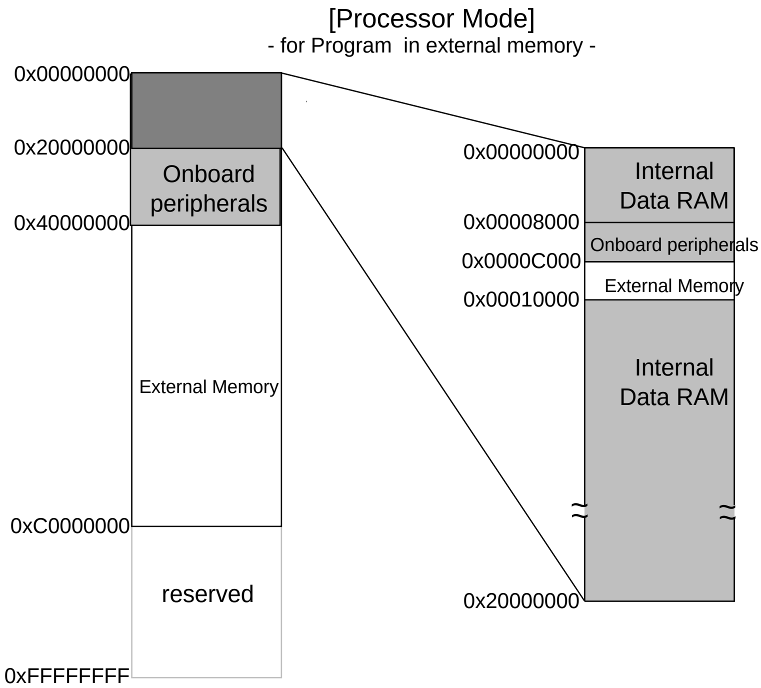


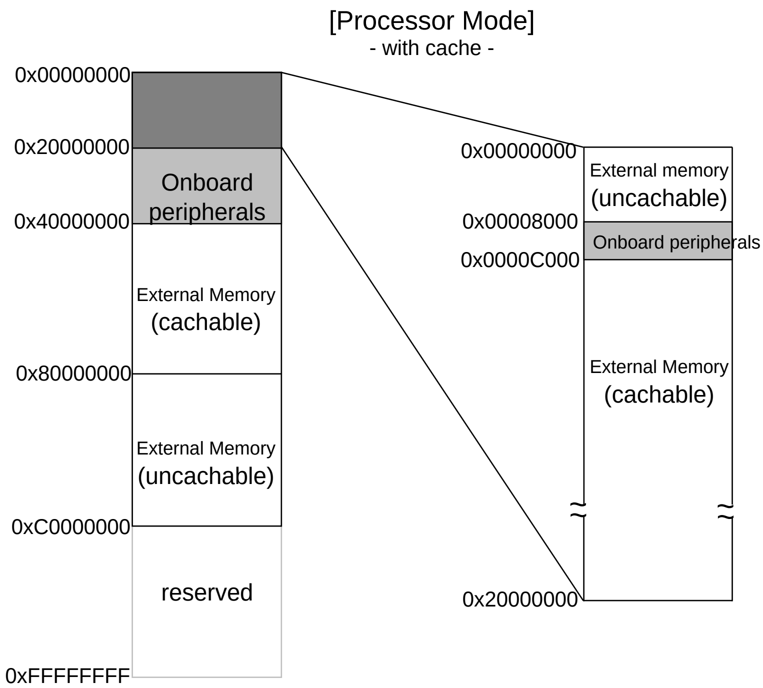
■ AM32

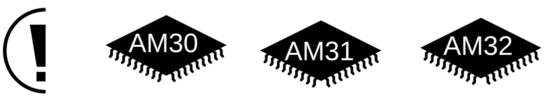

[Single Chip Mode]
- for Instruction ROM -
0x00000000
0x20000000
Onboard peripherals
0x40000000
Internal Program ROM
0x80000000
reserved
0xC0000000
reserved
0xFFFFFFFF
0x00000000
Internal Data RAM
0x00008000
Onboard peripherals
0x0000C000
External Memory
0x00010000
Internal Data RAM
0x20000000


[Extended Memory Mode]
- for Program ROM/RAM -
0x00000000
0x20000000
Onboard peripherals
0x40000000
Internal Program ROM / RAM
0x80000000
External Memory
0xC0000000
reserved
0xFFFFFFFF
0x00000000
Internal Data RAM
0x00008000
Onboard peripherals
0x0000C000
External Memory
0x00010000
Internal Data RAM
0x20000000

■ AM32







Memory layout varies with such factors as model and pin specifications. For further details, refer to the documentation for the particular device.

5 Addressing Modes

The addressing modes available consist of the following six most heavily used by C compilers.

1. Register direct
2. Immediate value
3. Register indirect
4. Register relative indirect
5. Absolute
6. Register indirect with indexing

Data transfer instructions offer all six addressing modes: register direct, immediate, register indirect, register relative indirect, absolute, and register indirect with indexing.

Register arithmetic instructions offer only two addressing modes: register direct and absolute.

Register indirect addressing with indexing is for more efficient access to arrays and the like.

■ Addressing Modes

Addressing Mode		Address Calculation	Final Address
Register direct	Dm/Dn Am/An SP/PSW/MDR	―――	―――
Immediate	imm8/regs imm16 imm24 imm32 imm40 imm48	―――	―――
Register indirect	(am)/(An)	31 0 [Am/An]	31 0 [(32-bit address)]
Register relative indirect	(d8,Am)/(d8,An) :d8 sigm-extended (d16,Am)/(d16,An) :d16 sign-extended (d32,Am)/(d32,An)	31 0 [Am/An] + 31 15 7 0 [d32/d6/d8]	31 0 [(32-bit address)]
	(Branch instructions only) (d8,PC) :d8 sign-extended (d16,PC) :d16 sign-extended (d32,PC)	31 0 [PC] + 31 15 7 0 [d32/d16/d8]	31 0 [(32-bit address)]
	(d8,PC) :d8 zero-extended (d16, SP) :d16 zero-extended (d32, SP)	31 0 [SP] + 31 15 7 0 [d32/d16/d8]	31 0 [(32-bit address)]
Absolute	(abs16) :abs16 zero-extended (abs32)	31 0 [abs16/abs32]	31 0 [(32-bit address)]
Register indirect with indexing	(di,Am)/(Di,An)	31 0 [Am/An] + 31 0 [Di]	31 0 [(32-bit address)]

(!) The suffixes m, n, and i indicate the source, destination, and index registers, respectively. All three have the range 0 to 3.

5.1 Register Direct Addressing

Register direct addressing specifies an operand as the name of a register from the following list.

Dn/Dm (32-bit) Data register
An/Am (32-bit) Address register
SP (32-bit) Stack Pointer
PSW (16-bit) Processor Status Word
MDR (32-bit) Multiply/Divide Register

5.2 Immediate Addressing

Immediate addressing specifies an operand as a value incorporated as is into the instruction code. Examples include numbers for loading into registers, masks, and multiregister specifications (regs) for transfers to and from the stack.
These operands are abbreviated to imm8, imm16, imm24, imm32, imm40, and imm48, where the numeric suffix indicates the size in bits.

The abbreviation regs denotes an 8-bit immediate value containing five bits specifying the registers D2, D3, A2, and A3 individually and seven other registers as a group.

7							0
D2	D3	As	A3	other	-	-	-

other : D0, D1, A0, A1, MDR, LIR, and LAR
- : Reserved (always set to 0)

5.3 Register Indirect Addressing

Register indirect addressing, (An) or (Am), specifies an address operand as the contents of a 32-bit address register.

Operand format: (An) or (Am)

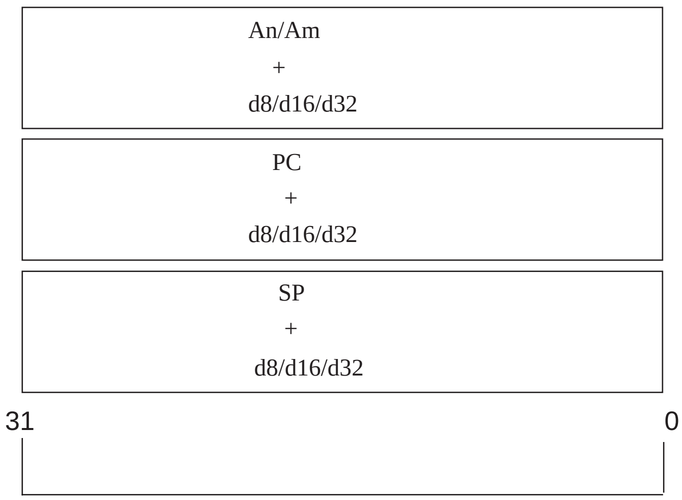
An/Am	
31	0

32-bit value used as memory address

5.4 Register Relative Indirect Addressing

Register relative indirect addressing specifies an address operand as the sum of a displacement and a base address in an address register (An or Am), the Program Counter (PC), or Stack Pointer (SP). Displacements can be 8, 16, or 32 bits wide.
Short (8- or 16-bit) displacements are zero-extended for the base register Stack Pointer (SP) and sign-extended for the others (An, Am, and PC).

Operand formats:		
	(d8, An) or (d8, Am)	:d8 sign-extended
	(d16, An) or (d16, Am)	:d16 sign-extended
	(d32, An) or (d32, Am)	:
	(d8, PC)	:d8 sign-extended
	(d16, PC)	:d16 sign-extended
	(d32, PC)	:
	(d8, SP)	:d8 zero-extended
	(d16, SP)	:d16 zero-extended
	(d32, SP)	:



32-bit value used as memory address

AM30 AM31: The result of adding the displacement to the base register An, Am, or PC must be in the same memory address space as the address in the base register.

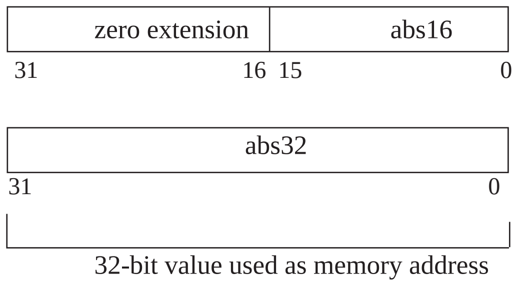
Any overflow arising during addition to the Program Counter (PC) is ignored. The effective address is the lowest 32 bits of the result.

5.5 Absolute Addressing

Absolute addressing specifies an address operand as a 16- or 32-bit value incorporated as is into the instruction code.

A 16-bit operand is zero-extended to 32 bits.

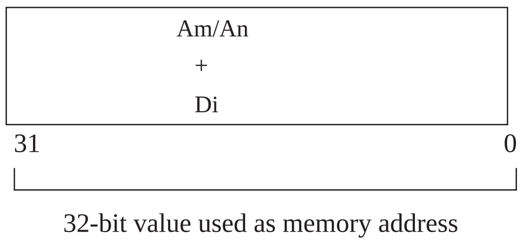
Operand formats: (abs16) :16-bit absolute address
(abs32) :32-bit absolute address



5.6 Register Indirect Addressing with Indexing

Register indirect addressing with indexing specifies an address operand as the sum of a base address in an address register (An or Am) and an index in a data register (Di).

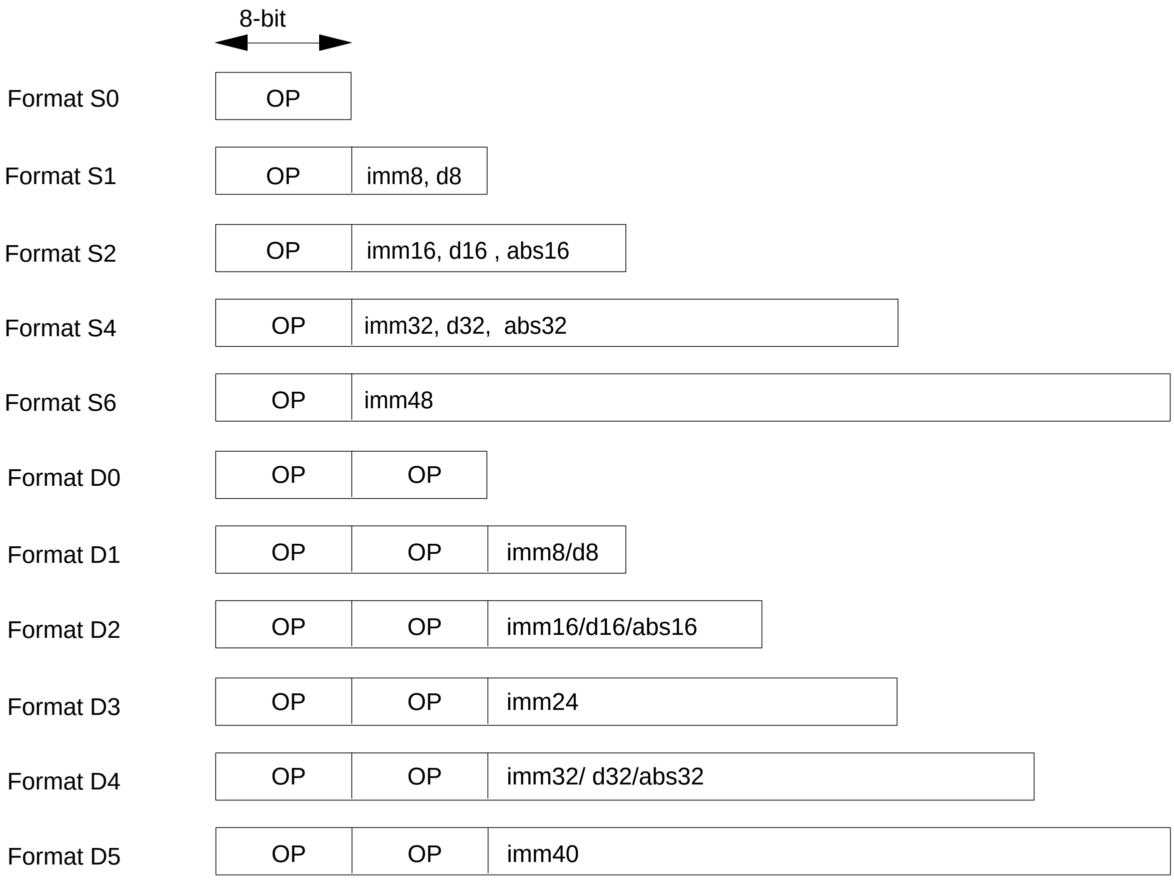
Operand format: (Di, An) or (Di, Am)



AM30 AM31 The result of adding the index to the base register (An or Am) must be in the same memory address space as the address in the base register.

6 Instruction Formats

There are 11 instruction formats.



The normal pattern consists of one or two opcode bytes following by an immediate value, displacement, or absolute value that is 8, 16, or 32 bits long. Formats S2, S4, S6, D2, D3, and D5, however, can have two or more such operands. For simplicity, the above diagram combines them under the immediate value labels imm16, imm24, imm32, imm40, and imm48, where the numeric suffix indicates the size in bits.
The following are the instructions affected.

imm16:	RET	regs, imm8	imm32:	CALL	(D16, PC), regs, imm8
	RETF	regs, imm8	imm40:	BTST	imm8, (abs32)
	BTST	imm8, (d8, An)		BSET	imm8, (abs32)
	BSET	imm8, (d8, An)		BCLR	imm8, (abs32)
	BCLR	imm8, (d8, An)	imm48:	CALL	(d32, PC), regs, imm8
imm24	BTST	imm8, (abs16)			
	BSET	imm8, (abs16)			
	BCLR	imm8, (abs16)			

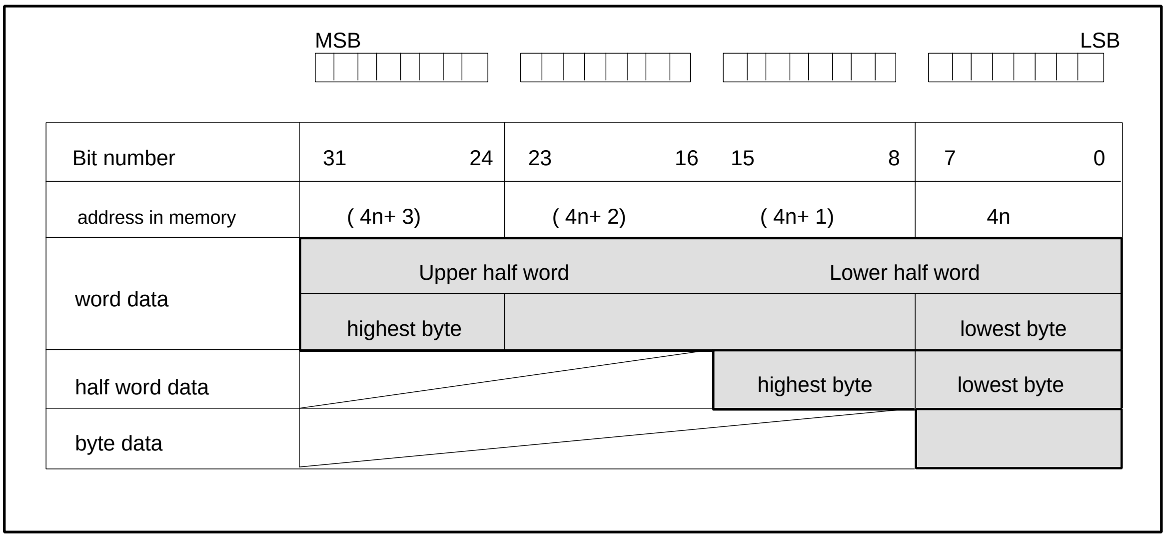
The assembler does not normally specify the two operands regs and imm8 for the RET, RETF, and CALL instructions directly. It uses an indirect approach, specifying them, at the subroutine entry point, in a directive subsequently resolved by the linker. For further details, refer to the Cross Assembler User's Manual.

6.1 Data Formats

Processing uses four data types: bit, byte, half-word, and word. The last three can be either signed or unsigned. The sign bit is the most significant one (MSB) for the data size.

Alignment restrictions apply. The storage address for a word data item must have '00' in its lowest two bits--that is, must be a multiple of four. Similarly, that for a half-word data item must have '0' in its lowest bit--that is, must be a multiple of two.

(1) Bit data	
(2) Byte data	
unsigned 8-bit data	
signed 8-bit data (sign in bit 7)	(signed:MSB)
(3) Half-word data	
unsigned 16-bit data	
signed 16-bit data (sign in bit 15)	(signed: MSB)
(4) Word data	
unsigned 32-bit data	
signed 32-bit data (sign in bit 31)	(signed: MSB)



6.2 Byte Order

Byte order is little endian: the bytes making up a 16- or 32-bit immediate value, displacement, or absolute value (imm16, d16, abs16, imm32, d32, or abs32) are stored from least significant byte to most as addresses increase.

[Example]
Little endian order stores the four bytes in the 32-bit immediate value 0x01234567 in the following order.

Address n	0x67
Address n+1	0x45
Address n+2	0x23
Address n+3	0x01

The formats with two or more operands (S2, S4, S6, D2, D3, and D5) maintain little endian order for their 16- or 32-bit immediate values and displacements (d16, abs16, d32, and abs32), but the order of the operands making up the fields abbreviated to imm16, imm24, imm32, imm40, and imm48 varies with the instruction.

RET/RETF regs, imm8

Address n	RET/RETF	.....RET or RETF Opecode
Address n+1	regs	
Address n+2	imm8	

BTST/BSET/BCLR imm8, (d8, An)

Address n Address n+1	BTST/BSET/BCLR	.....BTST, BSET or BCLR Opecode
Address n+2	d8	
Address n+3	imm8	

BTST/BSET/BCLR imm8, (abs16)

Address n Address n+1	BTST/BSET/BCLR	.....BTST, BSET or BCLR Opecode
Address n+2 Address n+3	abs16	
Address n+4	imm8	

CALL (d16, PC), regs, imm8

Address n	CALL	.....CALL Opecode
Address n+1	d16	
Address n+2		
Address n+3	regs	
Address n+4	imm8	

BTST/BSET/BCLR imm8, (abs32)

Address n	BTST/BSET/BCLR	.....BTST, BSET or BCLR Opecode
Address n+1		
Address n+2	abs32	
Address n+3		
Address n+4		
Address n+5		
Address n+6	imm8	

CALL (d32, PC), regs, imm8

Address n	CALL	.....CALL Opecode
Address n+1	d32	
Address n+2		
Address n+3		
Address n+4		
Address n+5	regs	
Address n+6	imm8	

Instruction Specifications

2

Symbol Definitions

■ Following is the list of symbols used in the instruction specifications.

Reg :register (used for general meaning)

Am, An :address register (m, n=3 to 0)

Dm, Dn,Di :data register (m, n, i=3 to 0)

MDR :multiply/divide register

PSW :processor status word

PC :program counter

SP :stack pointer

LIR :loop instruction register

LAR :loop address register

{MDR,Dn} :64-bit data defined whose upper 32-bit in MDR and lower 32-bit in register Dn within a "{ }".

Mem :memory (used for general meaning)

imm :immediate value (used for general meaning)

imm8 :8-bit immediate value

imm16 :16-bit immediate value

imm32 :32-bit immediate value

d8 :8-bit displacement

d16 :16-bit displacement

d32 :32-bit displacement

abs16 :16-bit absolute

abs32 :32-bit absolute

( ) :indirect addressing

Refer to "Chapter 1 section 5, Addressing Mode" for details.

regs :multiple registers specification

0x . . . . :hexadecimal notation(the numbers following 0x are expressed in hexadecimal notation.)

.bpn :bit location ("n" means location of bit; 0 to 31)

.lsb :bit location (bit 0)

.msb :bit location (bit 31)

& :logical AND

| :logical OR

^ :exclusive OR

~ :bit inverted

<<n :n-bit shift left

>>n :n-bit shift right

→ :move

: :reflection of operation result

(sign_ext) :sign-extend

(zero_ext) :zero-extend

label :address

VF :overflow flag

CF :carry flag

NF :nagative flag

ZF :zero flag

temp :temporary register

mem8(xxx) :8-bit data in memory specified with xxx
mem16(xxx) :16-bit data in memory specified with xxx
mem32(xxx) :32-bit data in memory specified with xxx
CodeSize :code size of assembler mnemonic

■ Following is the list of symbols used in flag changes.
("flag" is a general term of lower 4-bit(V, C, N, Z) of PSW.

● :flag changes
_ :no flag change
0 :flag is always "0"
1 :flag is always "0"
? :flag change undefined
* :change by user defined

■ "Cycles" will be changed by status of pipeline or memory space to access.
"Cycles" written in this chapter are calculated on the following conditions;

(1) No pipleline installation
(2) Instruction fetch: 2 cycles, data load/store: 1 cycle
(ROM/RAM/flash build-in products:
Instructions: accessing internal instruction ROM space or internal instruction RAM space
Data: accessomg internal data RAM space
Cache build-in products:
Instructions/data: when accessing cachable area, cache is always hit.

Refer to "Chapter 3, Using Instructions" for influence by pipleline installation, LSI Manual of each product for cycle changes in memory space.

■ Symbols for Notation
Each microcomputer core has different notations. Therefore, each notation is written with each microcomputer core mark in this manual. The microcomputer core marks are as followings;

AM30 Notice for AM30 core

AM31 Notice for AM31 core

AM32 Notice for AM32 core

MOV

Move

MOV Reg1,Reg2

Operation

Reg1→Reg2

Moves the contents of the register(Reg1) to the register(Reg2). Not moves to the same register.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
mov Dm,Dn	Dm=Dn cannot be specified	—	—	—	—	1	1
mov Dm,An		—	—	—	—	2	1
mov Am,Dn		—	—	—	—	2	1
mov Am,An	Am=An cannot be specified	—	—	—	—	1	1
mov SP,An		—	—	—	—	1	1
mov Am,SP		—	—	—	—	2	1
mov PSW,Dn	Zero-extends the upper 16 bits	—	—	—	—	2	1
mov Dm,PSW	Omits the upper 16 bits	●	●	●	●	2	1
mov MDR,Dn		—	—	—	—	2	1
mov Dm,MDR		—	—	—	—	2	1

Flag Changes

Other than mov Dm,PSW

- VF: No Changes.
- CF: No Changes.
- NF: No Changes.
- ZF: No Changes.

mov Dm,PSW

- VF: Reflects the 3rd bit of Dm.
- CF: Reflects the 2nd bit of Dm.
- NF: Reflects the 1st bit of Dm.
- ZF: Reflects the zero bit of Dm.

PSW-update by mov Dm,PSW instruction can be delayed for two instructions at most.
Especially at interrupting affected by IE bit or IM field, note that the instruction during updating will be executed in the status before/after updaiing.

MOV Mem,Reg

Operation	Mem→Reg Word-data-moves the contents of the memory(Mem) to the register(Reg).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
mov (Am),Dn		—	—	—	—	1	1
mov (d8,Am),Dn	d8 is sign-extended	—	—	—	—	3	1
mov (d16,Am),Dn	d16 is sign-extended	—	—	—	—	4	1
mov (d32,Am),Dn		—	—	—	—	6	2
mov (d8,SP),Dn	d8 is zero-extended	—	—	—	—	2	1
mov (d16,SP),Dn	d16 is zero-extended	—	—	—	—	4	1
mov (d32,SP),Dn		—	—	—	—	6	2
mov (Di,Am),Dn		—	—	—	—	2	1
mov (abs16),Dn	abs16 is zero-extended	—	—	—	—	3	1
mov (abs32),Dn		—	—	—	—	6	2
mov (Am),An		—	—	—	—	2	1
mov (d8,Am),An	d8 is sign-exended	—	—	—	—	3	1
mov (d16,Am),An	d16 is sign-extended	—	—	—	—	4	1
mov (d32,Am),An		—	—	—	—	6	2
mov (d8,SP),An	d8 is zero-extended	—	—	—	—	2	1
mov (d16,SP),An	d16 is zero-extended	—	—	—	—	4	1
mov (d32,SP),An		—	—	—	—	6	2
mov (Di,Am),An		—	—	—	—	2	1
mov (abs16),An	abs16 is zero-extended	—	—	—	—	4	1
mov (abs32),An		—	—	—	—	6	2
mov (d8,Am),SP	d8 is sign-extended	—	—	—	—	3	1

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! The operation of the memory(Mem) address other than multiple of four is not guaranteed.

! AM32 In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(Am,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.

! AM30 AM31 In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(Am,SP) and the address derived from address calculation must be in the same memory space.

MOV Reg,Mem

Operation	Reg1→Mem Word-data-moves the contents of the memory(Mem) to the register(Reg).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
mov Dm,(An)		—	—	—	—	1	1
mov Dm,(d8,An)	d8 is sign-extended	—	—	—	—	3	1
mov Dm,(d16,An)	d16 is sign-extended	—	—	—	—	4	1
mov Dm,(d32,An)		—	—	—	—	6	2
mov Dm,(d8,SP)	d8 is zero-extended	—	—	—	—	2	1
mov Dm,(d16,SP)	d16 is zero-extended	—	—	—	—	4	1
mov Dm,(d32,SP)		—	—	—	—	6	2
mov Dm,(Di,An)		—	—	—	—	2	2
mov Dm,(abs16)	abs16 is zero-extended	—	—	—	—	3	1
mov Dm,(abs32)		—	—	—	—	6	2
mov Am,(An)		—	—	—	—	2	1
mov Am,(d8,An)	d8 is sign-extended	—	—	—	—	3	1
mov Am,(d16,An)	d16 is sign-extended	—	—	—	—	4	1
mov Am,(d32,An)		—	—	—	—	6	2
mov Am,(d8,SP)	d8 is zero-extended	—	—	—	—	2	1
mov Am,(d16,SP)	d16 is zero-extended	—	—	—	—	4	1
mov Am,(d32,SP)		—	—	—	—	6	2
mov Am,(Di,An)		—	—	—	—	2	2
mov Am,(abs16)	abs16 is zero-extended	—	—	—	—	4	1
mov Am,(abs32)		—	—	—	—	6	2
mov SP,(d8,An)	d8 is sign-extended	—	—	—	—	3	1

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! The operation of the memory(Mem) address other than multiple of four is not guaranteed.

! AM32 In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(An,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.

! AM30 AM31 In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(An,SP) and the address derived from address calculation must be in the same memory space.

MOV imm,Reg

Operation	imm→Reg Moves the contents of the immediate value(imm) to the register(Reg).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
mov imm8,Dn	imm8 issign-extended	—	—	—	—	2	1
mov imm16,Dn	imm16 is sign-extended	—	—	—	—	3	1
mov imm32,Dn		—	—	—	—	6	2
mov imm8,An	imm8 is zero-extended	—	—	—	—	2	1
mov imm16,An	imm16 is zero-extended	—	—	—	—	3	1
mov imm32,An		—	—	—	—	6	2

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

MOVBU Zero-extend Byte Move

MOVBU Mem,Reg

Operation	Mem→Reg Byte-data-moves the contents of the memory(Mem) to the register(Reg) (8 bits→32 bits; zero-extended)

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
movbu (Am),Dn		—	—	—	—	2	1
movbu (d8,Am),Dn	d8 is sign-extended	—	—	—	—	3	1
movbu (d16,Am),Dn	d16 is sign-extended	—	—	—	—	4	1
movbu (d32,Am),Dn		—	—	—	—	6	2
movbu (d8,SP),Dn	d8 is zero-extended	—	—	—	—	3	1
movbu (d16,SP),Dn	d16 zero-extended	—	—	—	—	4	1
movbu (d32,SP),Dn		—	—	—	—	6	2
movbu (Di,Am),Dn		—	—	—	—	2	1
movbu (abs16),Dn	abs16 zero-extended	—	—	—	—	3	1
movbu (abs32),Dn		—	—	—	—	6	2

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! AM32 In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(Am,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.

! AM30 AM31 In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(Am,SP) and the address derived from address calculation must be in the same memory space.

MOVBU Reg,Mem

Operation

Reg→Mem

Byte-moves the contents of the register(Reg) to the memory(Mem).
(32 bits→8bits: Omit the upper)

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
movbu Dm,(An)		—	—	—	—	2	1
movbu Dm,(d8,An)	d8 is sign-extended	—	—	—	—	3	1
movbu Dm,(d16,An)	d16 is sign-extended	—	—	—	—	4	1
movbu Dm,(d32,An)		—	—	—	—	6	2
movbu Dm,(d8,SP)	d8 is zero-extended	—	—	—	—	3	1
movbu Dm,(d16,SP)	d16 zero-extended	—	—	—	—	4	1
movbu Dm,(d32,SP)		—	—	—	—	6	2
movbu Dm,(Di,An)		—	—	—	—	2	2
movbu Dm,(abs16)	abs16 zero-extended	—	—	—	—	3	1
movbu Dm,(abs32)		—	—	—	—	6	2

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! AM32

In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(Am,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.

! AM30 AM31

In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(An,SP) and the address derived from address calculation must be in the same memory space.

MOVB

Sign-extend Byte Move
[combination of multiple instructions]

MOVB Mem,Reg

Operation

Mem→Reg

Byte-data-moves the contents of the memory(Mem) to the register(Reg).
(8 bits→32bits: sign-extended)

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
movb (Am),Dn		—	—	—	—	3	2
movb (d8,Am),Dn	d8 is sign-extended	—	—	—	—	4	2
movb (d16,Am),Dn	d16 is sign-extended	—	—	—	—	5	2
movb (d32,Am),Dn		—	—	—	—	7	3
movb (d8,SP),Dn	d8 is zero-extended	—	—	—	—	4	2
movb (d16,SP),Dn	d16 is zero-extended	—	—	—	—	5	2
movb (d32,SP),Dn		—	—	—	—	7	3
movb (Di,Am),Dn		—	—	—	—	3	2
movb (abs16),Dn	abs16 is zero-extended	—	—	—	—	4	2
movb (abs32),Dn		—	—	—	—	7	3

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! This instruction is executed with the combination of multiple instructions and the assembler generates the following instructions.

```
MOVBU    Mem, Reg
EXTB     Reg
```

! The numbers of Size and Cycles contain those of the multiple instructions mentioned above. For the optimaization of assembler, the location within the multiple instructions may change the number of Cycles. Refer to "Chapter 3 Using the Instructions" for details.

! AM32 In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(Am,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.

! AM30 AM31 In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(Am,SP) and the address derived from address calculation must be in the same memory space.

MOVB Reg,Mem

Operation	Reg→Mem Byte-data-moves the contents of the register(Reg) to thememory(Mem) . (32 bits→8bits: Omit the upper)

Assembler mnemonic	Notes	V	C	N	Z	Sign	Cycles
movb Dm,(An)		—	—	—	—	2	1
movb Dm,(d8,An)	d8 is sign-extended	—	—	—	—	3	1
movb Dm,(d16,An)	d16 is sign-extended	—	—	—	—	4	1
movb Dm,(d32,An)		—	—	—	—	6	2
movb Dm,(d8,SP)	d8 is zero-extended	—	—	—	—	3	1
movb Dm,(d16,SP)	d16 is zero-extended	—	—	—	—	4	1
movb Dm,(d32,SP)		—	—	—	—	6	2
movb Dm,(Di,An)		—	—	—	—	2	2
movb Dm,(abs16)	abs16 is zero-extended	—	—	—	—	3	1
movb Dm,(abs32)		—	—	—	—	6	2

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! This instruction is executed by overwriting the instructions and the assembler generates the following instructions.

MOVBU Reg, Mem

! The numbers of Size and Cycles are those of the instructions mentioned above.

! AM32 In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(An,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.

! AM30 AM31 In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(An,SP) and the address derived from address calculation must be in the same memory space.

MOVHU Zero-extend Half Word Move

MOVHU Mem,Reg

Operation	
	Mem→Reg Half-word-moves the contents of the memory(Mem) to the register(Reg). (16 bits→35bits: zero-extended)

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
movhu (Am),Dn		—	—	—	—	2	1
movhu (d8,Am),Dn	d8 is sign-extended	—	—	—	—	3	1
movhu (d16,Am),Dn	d16 is sign-extended	—	—	—	—	4	1
movhu (d32,Am),Dn		—	—	—	—	6	2
movhu (d8,SP),Dn	d8 is zero-extended	—	—	—	—	3	1
movhu (d16,SP),Dn	d16 is zero-extended	—	—	—	—	4	1
movhu (d32,SP),Dn		—	—	—	—	6	2
movhu (Di,Am),Dn		—	—	—	—	2	1
movhu (abs16),Dn	abs16 is zero-extended	—	—	—	—	3	1
movhu (abs32),Dn		—	—	—	—	6	2

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! The operation of the memory(Mem) address other than multiple of two is not guaranteed.

! AM32 In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(Am,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.

! AM30 AM31 In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(Am,SP) and the address derived from address calculation must be in the same memory space.

MOVHU Reg,Mem

Operation	
	Reg→Mem Half-word-moves the contents of the register(Reg) to the memory(Mem). (32 bits→16bits: Omit the upper)

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
movhu Dm,(An)		—	—	—	—	2	1
movhu Dm,(d8,An)	d8 is sign-extended	—	—	—	—	3	1
movhu Dm,(d16,An)	d16 is sign-extended	—	—	—	—	4	1
movhu Dm,(d32,An)		—	—	—	—	6	2
movhu Dm,(d8,SP)	d8 is zero-extended	—	—	—	—	3	1
movhu Dm,(d16,SP)	d16 is zero-extended	—	—	—	—	4	1
movhu Dm,(d32,SP)		—	—	—	—	6	2
movhu Dm,(Di,An)		—	—	—	—	2	2
movhu Dm,(abs16)	abs16 is zero-extended	—	—	—	—	3	1
movhu Dm,(abs32)		—	—	—	—	6	2

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! The operation of the memory(Mem) address other than multiple of two is not guaranteed.

! AM32 In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(An,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.

! AM30 AM31 In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(An,SP) and the address derived from address calculation must be in the same memory space.

MOVH

Sign-extend Half Word Move
[Combination of Multiple Instructions]

MOVH Mem,Reg

Operation

Mem→Reg

Half-word-moves the contents of the memory(Mem) to the register(Reg).
(16 bits→32bits: sign-extended)

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
movh (Am),Dn		—	—	—	—	3	2
movh (d8,Am),Dn	d8 is sign-extended	—	—	—	—	4	2
movh (d16,Am),Dn	d16 is sign-extended	—	—	—	—	5	2
movh (d32,Am),Dn		—	—	—	—	7	3
movh (d8,SP),Dn	d8 is zero-extended	—	—	—	—	4	2
movh (d16,SP),Dn	d16 is zero-extended	—	—	—	—	5	2
movh (d32,SP),Dn		—	—	—	—	7	3
movh (Di,Am),Dn		—	—	—	—	3	2
movh (abs16),Dn	abs16 is zero-extended	—	—	—	—	4	2
movh (abs32),Dn		—	—	—	—	7	3

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! The operation of the memory(Mem) address other than multiple of two is not guaranteed.

! This instruction is executed with the combination of multiple instructions and the assembler generates the following instructions.

```
MOVHU    Mem, Reg
EXTH     Reg
```

! The numbers of Size and Cycles contain those of the multiple instructions mentioned above. For the optimaization of assembler, the location within the multiple instructions may change the number of Cycles. Refer to "Chapter 3 Using the Instructions" for details.

! AM32 In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(Am,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.

! AM30 AM31 In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(Am,SP) and the address derived from address calculation must be in the same memory space.

MOVH Reg,Mem

Operation	Reg→Mem Half-word-moves the contents of the register(Reg) to the memory(Mem). (32 bits→16 bits: Omit the upper)

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
movh Dm,(An)		—	—	—	—	2	1
movh Dm,(d8,An)	d8 is sign-extended	—	—	—	—	3	1
movh Dm,(d16,An)	d16 is sign-extended	—	—	—	—	4	1
movh Dm,(d32,An)		—	—	—	—	6	2
movh Dm,(d8,SP)	d8 is zero-extended	—	—	—	—	3	1
movh Dm,(d16,SP)	d16 is zero-extended	—	—	—	—	4	1
movh Dm,(d32,SP)		—	—	—	—	6	2
movh Dm,(Di,An)		—	—	—	—	2	2
movh Dm,(abs16)	abs16 is zero-extended	—	—	—	—	3	1
movh Dm,(abs32)		—	—	—	—	6	2

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! The operation of the memory(Mem) address other than multiple of two is not guaranteed.

! This instruction is executed by overwriting the instructions and the assembler generates the following instructions.
MOVHU Reg, mem

! The numbers of Size and Cycles are those of the instructions mentioned above.

! AM32 In register-relative indirect addressing mode or index decoration register indirect addressing mode, when the address specified by based register(An,SP) and the address derived from address caluculation are not in the same memory space, one cycle will be added.

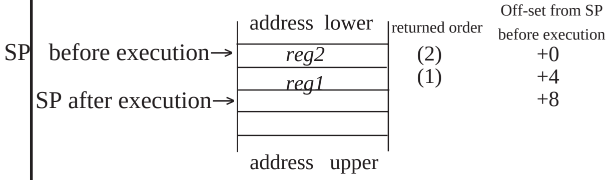
! AM30 AM31 In register-relative indirect addressing mode or index decoration register indirect addressing mode, the address specified by based register(An,SP) and the address derived from address calculation must be in the same memory space.

MOVM Move Between Multiple Memory and Register

MOVM (SP),regs

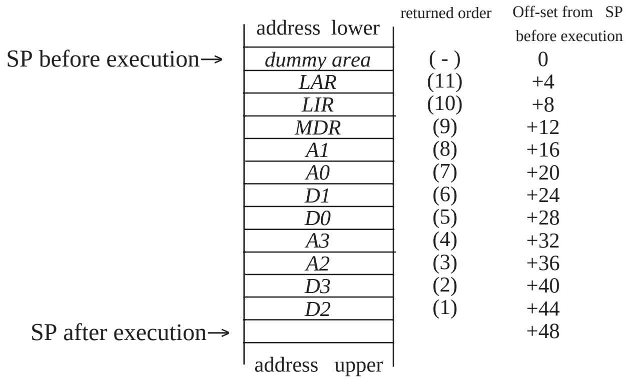
Operation

No "other" in the specified register;
If regs = [Reg1,Reg2],
(regs-specified registers=2)
mem32(SP+4)→reg1,
mem32(SP)→reg2,SP+8→SP



Block-moves from the memory specified with SP to the multiple registers.
The "regs" specifies the multiple registers to move data and it can specify each D2,D3, A2, A3 and the other registers(D0, D1, A0, A1, MDR, LIR, LAR).

"Other" in the specified register;
If regs = [Reg1,Reg2,Reg3,Reg4,Reg5]
(regs-specified registers=11)
mem32(SP+44)→D2,mem32(SP+40)→D3,
mem32(SP+36)→A2,mem32(SP+32)→A3,
mem32(SP+28)→D0,mem32(SP+24)→D1,
mem32(SP+20)→A0,mem32(SP+16)→A1,
mem32(SP+12)→MDR,mem32(SP+8)→LIR,
mem32(SP+4)→LAR,SP+48→SP



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
movm (SP),[Reg1,..,Regn]	Block-moves from memory to multiple registers (regs-specified registers= 0)	—	—	—	—	2	1
	(regs-specified registers = 1)	—	—	—	—	2	2
	(regs-specified registers= 2)	—	—	—	—	2	3
	(regs-specified registers = 3)	—	—	—	—	2	4
	(regs-specified registers = 4)	—	—	—	—	2	5
	(regs-specified registers = 7)	—	—	—	—	2	8
	(regs-specified registers = 8)	—	—	—	—	2	9
	(regs-specified registers = 9)	—	—	—	—	2	10
	(regs-specified registers= 10)	—	—	—	—	2	11
	(regs-specified registers= 11)	—	—	—	—	2	12

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

! The register(any of D2,D3,A2, A3 or other) specified by the assembler is separatedwith comma(,) for each and parenthsize with ([ ]). However, you can not specifiy the same register twice or more.

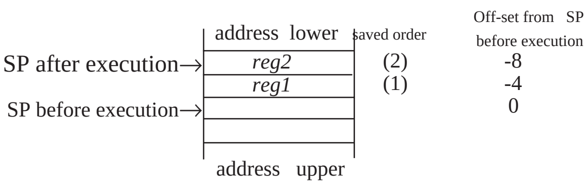
! No order to speify the registers by the assembler, however, the order of the registers to be returned is fixed as D2, D3, A2, A3, other(D0, D1, A0, A1, MDR, LIR, LAR). (Non-specified registers will be skipped.) If specifying "other", 4 byte of dummy area will be stored at last to simplify of move area calculation(4 byte x 8). (No moving operation.) If not specifying "other", no dummy area will be stored.

! Refer to "Appendix : Instruction set" for operating expressions of each register specified with "regs".

MOVM egs,(SP)

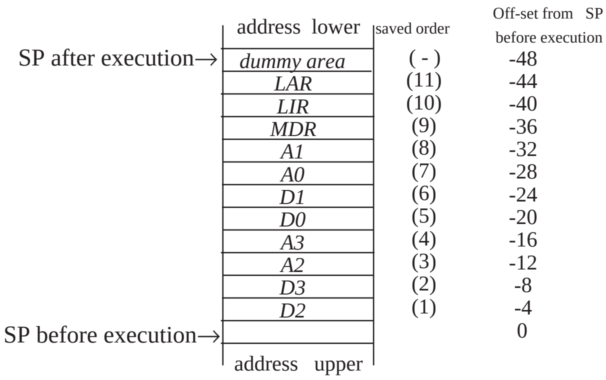
Operation

No "other" in the specified register;
If regs = [Reg1,Reg2],
(regs-specified registers=2)
reg1→mem32(SP-4),
reg2→mem32(SP-8),SP-8→SP



Block-moves from the multiple registers to the memory specified with SP. The "regs" specifies the multiple registers to move data and it can specify each D2,D3, A2, A3 and the other registers(D0, D1, A0, A1, MDR, LIR, LAR).

"Other" in the specified register;
If regs = [Reg1,Reg2,Reg3,Reg4,Reg5]
(regs-specified registers=11)
D2→mem32(SP-4),D3→mem32(SP-8),
A2→mem32(SP-12),A3→mem32(SP-16),
D0→mem32(SP-20),D1→mem32(SP-24),
A0→mem32(SP-28),A1→mem32(SP-32),
MDR→mem32(SP-36),LIR→mem32(SP-40),
LAR→mem32(SP-44),SP-48→SP



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
movm [Reg1,..,Regn],(SP)	Block-move from multiple registers to memory (regs-specified register = 0)	—	—	—	—	2	1
	(regs-specified register = 1)	—	—	—	—	2	1
	(regs-specified registers = 2)	—	—	—	—	2	2
	(regs-specified registers = 3)	—	—	—	—	2	3
	(regs-specified registers= 4)	—	—	—	—	2	4
	(regs-specified registers= 7)	—	—	—	—	2	8
	(regs-specified registers= 8)	—	—	—	—	2	9
	(regs-specified registers = 9)	—	—	—	—	2	10
	(regs-specified registers = 10)	—	—	—	—	2	11
	(regs-specified registers = 11)	—	—	—	—	2	12

Flag Changes

VF: No Changes.
CF: No Changes.
NF: No Changes.
ZF: No Changes.

❗ The register(any of D2,D3,A2, A3 or other) specified by the assembler is separatedwith comma(,) for each and parenthsize with ([ ]). However, you can not specifiy the same register twice or more.

❗ No order to speify the registers by the assembler, however, the order of the registers to be returned is fixed as D2, D3, A2, A3, other(D0, D1, A0, A1, MDR, LIR, LAR). (Non-specified registers will be skipped.) If specifying "other", 4 byte of dummy area will be stored at last to simplify of move area calculation(4 byte x 8). (No moving operation.) If not specifying "other", no dummy area will be stored.

❗ Refer to "Appendix : Instruction set" for operating expressions of each specified register.

EXT Sign-extend Word Data to 64 Bits

EXT Dn

Operation	If Dn.bp31= 0, 0x00000000→MDR If Dn.bp31= 1, 0xFFFFFFFF→MDR Sign-extends the value of register Dn to 64 bits and moves the extended upper 32 bits to MDR. No changes for the contents of Dn register.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
ext Dn		—	—	—	—	2	1

Flag Changes

VF: Nochanges
CF: No changes
NF: No changes
ZF: No changes

EXTB Sign-extend Byte Data to 32 Bits

EXTB Dn

Operation

If Dn.bp7 = 0,Dn & 0x000000FF→Dn
If Dn.bp7 = 1,Dn | 0xFFFFFF00→Dn

Sign-extends the lower 8 bits of register Dn to 32 bits and stores in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
extb Dn		—	—	—	—	1	1

Flag Changes

VF: Nochanges
CF: No changes
NF: No changes
ZF: No changes

EXTBU Zero-extend Byte Data to 32 Bits

EXTBU Dn

Operation	Dn & 0x000000FF→Dn Zero-extends the lower 8 bits of regiser Dn to 32 bits and stores in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
extbu Dn		—	—	—	—	1	1

Flag Changes

VF: Nochanges
CF: No changes
NF: No changes
ZF: No changes

EXTH Sign-extend Half Word Data to 32 Bits

EXTH Dn

Operation

If Dn.bp15 = 0, Dn & 0x0000FFFF→Dn
If Dn.bp15 = 1, Dn | 0xFFFF0000→Dn

Sign-extends the lower 16 bits of register Dn to 32 bits and stores in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
exth Dn		—	—	—	—	1	1

Flag Changes

VF: Nochanges
CF: No changes
NF: No changes
ZF: No changes

EXTHU

Zero-extend Half Word Data to 32 Bits

EXTHU Dn

Operation	
	Dn&0x0000FFFF→Dn Zero-extends the lower 16 bits of register Dn to 32 bits and stores in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
exthu Dn		—	—	—	—	1	1

Flag Changes

VF: Nochanges
CF: No changes
NF: No changes
ZF: No changes

CLR Data Clear

CLR Dn

Operation	
	0→Dn Clears the contents of register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
clr Dn		●	●	●	1	1	1

Flag Changes

VF: Always 0
CF: Always 0
NF: Always 0
ZF: Always 1

! One instruction is delayed when updating PSW by flag-change.
However, Bcc and Lcc instructions can evaluate the flag before affecting the flag.

ADD Addition

ADD Reg1,Reg2

Operation	
	Reg1+Reg2→Reg2 Adds the contents of the register(Reg1) and the register(Reg2) and stores the results in the register(Reg2).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
add Dm,Dn		●	●	●	●	1	1
add Dm,An		●	●	●	●	2	1
add Am,Dn		●	●	●	●	2	1
add Am,An		●	●	●	●	2	1

Flag Changes

VF: 1 if an overflow is generated as 32 bits signed-numeric value; 0 otherwise.
CF: 1 if a carry is generated from bit 31; 0 otherwise.
NF: 1 if the bit 31 of the result is '1'; 0 otherwise.
ZF: 1 if the result is '0'; 0 othewise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

ADD imm,Reg

Operation

imm+Reg→Reg

Adds the immediate value(imm) and the contents of the register(Reg) and stores the result in the register(Reg).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
add imm8,Dn	imm8 is sign-extended.	●	●	●	●	2	1
add imm16,Dn	imm16 is sign-extended.	●	●	●	●	4	1
add imm32,Dn		●	●	●	●	6	2
add imm8,An	imm8 is sign-extended.	●	●	●	●	2	1
add imm16,An	imm16 is sign-extended.	●	●	●	●	4	1
add imm32,An		●	●	●	●	6	2
add imm8,SP	imm8 is sign-extended.	—	—	—	—	3	1
add imm16,SP	imm16 is sign-extended.	—	—	—	—	4	1
add imm32,SP		—	—	—	—	6	2

Flag Change

Other than add imm,SP

- VF: 1 if an overflow is generated as 32 bits signed numeric value; 0 otherwise.
- CF: 1 if a carry is generated from bit 31; 0 otherwise.
- NF: 1 if the bit 31 of the result is '1'; 0 otherwise.
- ZF: 1 if the result is '0'; 0 othewise.

add imm,SP

- VF: No Changes.
- CF: No Changes.
- NF: No Changes.
- ZF: No Changes.

Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

ADDC Addition With Carry

ADDC Dm,Dn

Operation	Dm+Dn+CF→Dn Adds the contents of register Dm including C flag and register Dn and stores the result in the register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
addc Dm,Dn		●	●	●	●	2	1

Flag Changes

VF: 1 if an overflow is generated as 32 bits signed-numeric value; 0 otherwise.
CF: 1 if a carry is generated from bit 31; 0 otherwise.
NF: 1 if the bit 31 of the result is '1'; 0 otherwise.
ZF: 1 if the result is '0'; 0 othewise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

SUB Subtraction

SUB Reg1,Reg2

Operation

Reg2-Reg1→Reg2

Subtracts the contents of the register(Reg1) from the register(Reg2) and stores the result in the register(Reg2).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
sub Dm,Dn		●	●	●	●	2	1
sub Dm,An		●	●	●	●	2	1
sub Am,Dn		●	●	●	●	2	1
sub Am,An		●	●	●	●	2	1

Flag Changes

VF: 1 if an overflow is generated as 32 bits signed-numeric value; 0 otherwise.
CF: 1 if a carry is generated from bit 31; 0 otherwise.
NF: 1 if the bit 31 of the result is '1'; 0 otherwise.
ZF: 1 if the result is '0'; 0 othewise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

SUB imm,Reg

Operation

Reg- imm→Reg

Subtracts the immediate value from the register(Reg) and stores the result in the register(Reg).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
sub imm32,Dn		●	●	●	●	6	2
sub imm32,An		●	●	●	●	6	2

Flag Changes

VF: 1 if an overflow is generated as 32 bits signed-numeric value; 0 otherwise.
CF: 1 if a carry is generated from bit 31; 0 otherwise.
NF: 1 if the bit 31 of the result is '1'; 0 otherwise.
ZF: 1 if the result is '0'; 0 othewise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! Using add imm,Reg instruction may shrink the instruction code size.

SUBC Subtraction With Borrow

SUBC Dm,Dn

Operation	Dn-Dm-CF→Dn Subtracts the contents of register Dm including C flag from register Dn and stores the result in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
subc Dm,Dn		●	●	●	●	2	1

Flag Changes

VF: 1 if an overflow is generated as 32 bits signed-numeric value; 0 otherwise.
CF: 1 if a carry is generated from bit 31; 0 otherwise.
NF: 1 if the bit 31 of the result is '1'; 0 otherwise.
ZF: 1 if the result is '0'; 0 othewise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

MUL Multiplication With Signed

MUL Dm,Dn

Operation

(Dn*Dm)→{ MDR, Dn}

Multiplicants the contents of register Dm(signed 32 bits integer: non-multiplier) and register Dn(signed 32 bits integer:multiplier) and stores the upper 32 bits of the result(64 bits) in MDR and the lower 32 bits in register Dn.
The siginificant number of bytes from the LSB of the multiplier loaded to Dn before the operation is judged, and the operation is only performed for the range(byte unit) containing these significant values. In other words, the smaller the contents loaded to Dn, the faster operation results can be obtained.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
mul Dm,Dn	Dn = 0	?	?	●	●	2	3
	Value Dn can specify by 1-byte.	?	?	●	●	2	13
	Value Dn can specify by 2-byte.	?	?	●	●	2	21
	Value Dn can specify by 3-byte.	?	?	●	●	2	29
	Value Dn can specify by 4-byte.	?	?	●	●	2	34

Flag Changes

VF: Undefined.
CF: Undefined.
NF: 1 if the bit 31 of the result of the lower 32-bit is '1'; 0 otherwise.
ZF: 1 if the lower 32-bit of the result is '0'; 0 otherwise.

Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

AM30

Locating writing-instrucion to address register A0 at one or two instructions preceding this instrucion is prohibited. Refer to "Chapter 3, 2. 5 Notes for location of MUL/MULU instruction following A0 writing-instruction" for details. Some examples of prohibition are follows.

(Example1) A writing-instruction to register A0 is located in one instruction preceding this instruction

```
mov    0x80040900, a0
mul    d1, d0
```

(Example2) A writing-instruction to register A0 is located in two instructions preceding this instruction.

```
mov    0x80040900, a0
mov    0x0c, d0
mul    d1, d0
```

(Example2) An instruction to write in register A0 is located in one instruction preceding this instruction.

```
inc    a0
mul    d1, d0
```

MULU Multiplication Without Signed

MULU Dm,Dn

Operation

(Dn*Dm)→{ MDR, Dn}

Multiplicants the contents of register Dm(unsigned 32 bits integer: non-multiplier) and register Dn(unsigned 32 bits integer:multiplier) and stores the upper 32 bits of the result(64 bits) in MDR and the lower 32 bits in register Dn.

The siginificant number of bytes from the LSB of the multiplier loaded to Dn before the operation is judged, and the operation is only performed for the range(byte unit) containing these significant values. In other words, the smaller the contents loaded to Dn, the faster operation results can be obtained.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
mulu Dm,Dn	Dn = 0	?	?	●	●	2	3
	Value Dn can specify by 1-byte.	?	?	●	●	2	13
	Value Dn can specify by 2-byte.	?	?	●	●	2	21
	Value Dn can specify by 3-byte.	?	?	●	●	2	29
	Value Dn can specify by 4-byte.	?	?	●	●	2	34

Flag Changes

VF: Undefined.
CF: Undefined.
NF: 1 if the bit 31 of the result of the lower 32-bit is '1'; 0 otherwise.
ZF: 1 if the lower 32-bit of the result is '0'; 0 otherwise.

Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

AM30

Locating writing-instrucion to address register A0 at one or two instructions preceding this instrucion is prohibited. Refer to "Chapter 3, 2. 5 Notes for location of MUL/MULU instruction following A0 writing-instruction" for details. Some examples of prohibition are follows.

(Example1) A writing-instruction to register A0 is located in one instruction preceding this instruction

```
mov     0x80040900, a0
mulu    d1, d0
```

(Example2) A writing-instruction to register A0 is located in two instructions preceding this instruction.

```
mov     0x80040900, a0
mov     0x0c, d0
mulu    d1, d0
```

(Example2) An instruction to write in register A0 is located in one instruction preceding this instruction.

```
inc     a0
mulu    d1, d0
```

DIV

Division with signed

DIV Dm,Dn

Operation

((MDR << 32)&0xFFFFFFFF00000000+Dn)/Dm→Dn
((MDR << 32)&0xFFFFFFFF00000000+Dn)%Dm→MDR

Divides signed 64-bit interger combined with MDR(undivided upper 32-bit) and Dn register(undevided lower 32-bit) by the contents of register Dm(division of signed 32-bit interger) stores the remainder(32-bit) in MDR and the quotient(32-bit) in register Dn. If the quotient can not be specified as signed 32-bit value, V flag will be '1' and MDR and register Dn will be undefined. When zero-division is performed(divisor=0), V flag will be '1'.
The significant number of bytes from the LSB of the 64-bit dividend obtained by linking MDR and Dn before the operation is judged (none that MDR is judged in word units), and the operation is only performed for the range containing these significant values. In other words, the smaller the dividend obtained by linking MDR and Dn, the faster operation results can be obtained.

Assembler mnemonic	Notes		V	C	N	Z	Size	Cycles
div Dm,Dn	Nomal performance (The operation performed normally)	{MDR,Dn}=0	0	?	●	●	2	4
		Value {MDR,Dn} can specify by 1-byte.	0	?	●	●	2	14
		Value {MDR,Dn} can specify by 2-byte.	0	?	●	●	2	22
		Value {MDR,Dn} can specify by 3-byte.	0	?	●	●	2	30
		Value {MDR,Dn} can specify by 4-byte or more.	0	?	●	●	2	38
	Divisor not specified as signed value or zero-division	{MDR,Dn}=0	1	?	?	?	2	4
		Value {MDR,Dn} can specify by 1-byte.	1	?	?	?	2	14
		Value {MDR,Dn} can specify by 2-byte.	1	?	?	?	2	22
		Value {MDR,Dn} can specify by 3-byte.	1	?	?	?	2	30
		Value {MDR,Dn} can specify by 4-byte or more.	1	?	?	?	2	38

Flag Changes

Nomal performance(the operation performed normally)

VF: Always 0.
CF: Undefined.
NF: 1 if MSB of the divisor(32-bit) is '1'. 0 otherwise.
ZF: 1 if the divisor(32-bit) is '0'. 0 otherwise.

Divisor not specified as signed value or zero-division performed.

VF: Always 1.
CF: Undefined.
NF: Undefined.
ZF: Undefined.

After the operation, if V flag is '1', the other flog will be undefined. Also the divisor and the remainder will be undefined.

Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

DIVU

Division without signed

DIVU Dm,Dn

Operation

((MDR<<32)&0xFFFFFFFF00000000+Dn)/Dm→Dn
((MDR<<32)&0xFFFFFFFF00000000+Dn)%Dm→MDR

Divides signed 64-bit interger combined with MDR(undivided upper 32-bit) and Dn register(undevided lower 32-bit) by the contents of register Dm(division of signed 32-bit interger) stores the remainder(32-bit) in MDR and the quotient(32-bit) in register Dn. If the quotient can not be specified as signed 32-bit value, V flag will be '1' and MDR and register Dn will be undefined. When zero-division is performed(divisor=0), V flag will be '1'.
The significant number of bytes from the LSB of the 64-bit dividend obtained by linking MDR and Dn before the operation is judged (none that MDR is judged in word units), and the operation is only performed for the range containing these significant values. In other words, the smaller the dividend obtained by linking MDR and Dn, the faster operation results can be obtained.

Assembler mnemonic	Notes		V	C	N	Z	Size	Cycles
divu Dm,Dn	Nomal performance (The operation performed normally)	{MDR,Dn}=0	0	?	●	●	2	4
		Value {MDR,Dn} can specify by 1-byte.	0	?	●	●	2	14
		Value {MDR,Dn} can specify by 2-byte.	0	?	●	●	2	22
		Value {MDR,Dn} can specify by 3-byte.	0	?	●	●	2	30
		Value {MDR,Dn} can specify by 4-byte or more.	0	?	●	●	2	38
	Divisor not specified as signed value or zero-division	{MDR,Dn}=0	1	?	?	?	2	4
		Value {MDR,Dn} can specify by 1-byte.	1	?	?	?	2	14
		Value {MDR,Dn} can specify by 2-byte.	1	?	?	?	2	22
		Value {MDR,Dn} can specify by 3-byte.	1	?	?	?	2	30
		Value {MDR,Dn} can specify by 4-byte or more.	1	?	?	?	2	38

Flag Changes

Nomal performance(the operation performed normally)

VF: Always 0.
CF: Undefined.
NF: 1 if MSB of the divisor(32-bit) is '1'. 0 otherwise.
ZF: 1 if the divisor(32-bit) is '0'. 0 otherwise.

Divisor not specified as signed value or zero-division performed.

VF: Always 1.
CF: Undefined.
NF: Undefined.
ZF: Undefined.

! After the operation, if V flag is '1', the other flog will be undefined. Also the divisor and the remainder will be undefined.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

INC 1 Addition

INC Reg

Operation	Reg+1→Reg Adds '1' to the register(Reg) and stores the result in the register(Reg).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
inc Dn		●	●	●	●	1	1
inc An		—	—	—	—	1	1

Flag Changes

inc Dn

VF: 1 if a divisor overflows as 32-bit signed numerical value. 0 otherwise.
CF: 1 if a carry is generated from bit 31. 0 otherwise.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise

Other than inc Dn

VF: No changes.
CF: No changes.
NF: No changes.
ZF: No changes.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

INC4 4 Addition

INC4 An

Operation	An+4→An Adds '4' to register A4 and stores the result in register An.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
inc4 An		—	—	—	—	1	1

Flag Changes

VF: No changes.
CF: No changes.
NF: No changes.
ZF: No changes.

CMP

Comparison

CMP Reg1,Reg2

Operation	Reg2-Reg:PSW Subtracts the contents of the register(Reg1) from the register(Reg2) and reflects the result to the flag. Tha same register can not be specified.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
cmp Dm,Dn	Dm=Dn cannot be specified	●	●	●	●	1	1
cmp Dm,An		●	●	●	●	2	1
cmp Am,Dn		●	●	●	●	2	1
cmp Am,An	Am=An cannot be specified	●	●	●	●	1	1

Flag Changes

VF: 1 if an overflows is generated as 32-bit signed numerical value. 0 otherwise.
CF: 1 if a borrow is generated from bit 31. 0 otherwise.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

CMP imm,Reg

Operation	Reg-imm:PSW Subtracts the immediate value from the register(Reg) and reflets the resutls to the flag.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
cmp imm8,Dn	imm8 is signed-extended	●	●	●	●	2	1
cmp imm16,Dn	imm16 is signed-extended	●	●	●	●	4	1
cmp imm32,Dn		●	●	●	●	6	2
cmp imm8,An	imm8 is zero-extended	●	●	●	●	2	1
cmp imm16,An	imm16 is zero-extended	●	●	●	●	4	1
cmp imm32,An		●	●	●	●	6	2

Flag Changes

VF: 1 if an overflows is generated as 32-bit signed numerical value. 0 otherwise.
CF: 1 if a borrow is generated from bit 31. 0 otherwise.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

AND Logical AND

AND Dm,Dn

Operation

Dm&Dn→Dn

Performs a logical AND and stores the result in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
and Dm,Dn		0	0	●	●	2	1

Flag Changes

VF: Always 0.
CF: Always 0.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

AND imm,Dn

Operation

imm&Dn→Dn

Performs a logical AND of the immediate value(imm) and register Dn and stores the result in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
and imm8,Dn	imm8 is zero-extended	0	0	●	●	3	1
and imm16,Dn	imm16 is zero-extended	0	0	●	●	4	1
and imm32,Dn		0	0	●	●	6	2

Flag Changes

VF: Always 0.
CF: Always 0.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

AND imm,PSW

Operation	imm&PSW→PSW Performs a logical AND of the immediate value(imm) and stores the result.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
and imm16,PSW	imm16 is zero-extended	●	●	●	●	4	1

Flag Changes

VF: Will be set to bit 3 of the result.
CF: Will be set to bit 2 of the result.
NF: Will be set to bit 1 of the result.
ZF: Will be set to bit 0 of the result.

(!) Updating of PSW by and imm16, PSW is delayed for two instuructions at most.
Especially for interruption affected by IE bit or IM field, note that the instruction during updating will be executed in the status before/after updating.

OR Logical OR

OR Dm,Dn

Operation

Dm|Dn→Dn

Performs a logical OR of register Dm and register Dn and stores the result in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
or Dm,Dn		0	0	●	●	2	1

Flag Changes

VF: Always 0.
CF: Always 0.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

OR imm,Dn

Operation

imm|Dn→Dn

Performs a logical OR of the immediate value(imm) and register Dn and stores the result in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
or imm8,Dn	imm8 is zero-extended	0	0	●	●	3	1
or imm16,Dn	imm16 is zero-extended	0	0	●	●	4	1
or imm32,Dn		0	0	●	●	6	2

Flag Changes

VF: Always 0.
CF: Always 0.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

OR imm,PSW

Operation

imm|PSW→PSW

Performs a logical OR of the immediate value and PSW and stores the result in PSW.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
or imm16,PSW	imm16 is zero-extended	●	●	●	●	4	1

Flag Changes

VF: Will be set to bit 3 of the result.
CF: Will be set to bit 2 of the result.
NF: Will be set to bit 1 of the result.
ZF: Will be set to bit 0 of the result.

Updating of PSW by and imm16, PSW is delayed for two instuructions at most.
Especially for interruption affected by IE bit or IM field, note that the instruction during updating will be executed in the status before/after updating.

XOR Exclusive Logical OR

XOR Dm,Dn

Operation

Dm^Dn→Dn

Performs an exclusive logical OR of register Dm and register Dn and stores the result in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
xor Dm,Dn		0	0	●	●	2	1

Flag Changes

VF: Always 0.
CF: Always 0.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

XOR imm,Dn

Operation

imm^Dn→Dn

Performs an exclusive logical OR of the immediate value and register Dn and stores the result in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
xor imm16,Dn	imm16 is zero-extended	0	0	●	●	4	1
xor imm32,Dn		0	0	●	●	6	2

Flag Changes

VF: Always 0.
CF: Always 0.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

NOT All Bits Inverted

NOT Dn

Operation

Dn^0xFFFFFFFF→Dn

Inverts all bits in register Dn and stores the result in register Dn.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
not Dn		0	0	●	●	2	1

Flag Changes

VF: Always 0.
CF: Always 0.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

BTST

Multiple Bits Test

BTST imm,Dn

Operation	
	imm&Dn→PSW Performs a logical AND of the immediate value and the contents of register Dn and reflects the result to the flag.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
btst imm8,Dn	imm8 is zero-extended	0	0	●	●	3	1
btst imm16,Dn	imm16 is zero-extended	0	0	●	●	4	1
btst imm32,Dn		0	0	●	●	6	2

Flag Changes

VF: Always 0.
CF: Always 0.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the operation result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

BTST imm,Mem

Operation	
	imm & Mem:PSW Performs a logical AND of the immediate value(imm) and the contents(byte data) of the memory(Mem) zero-extended to 32-bit and reflects the result to the flag.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
btst imm8,(d8,An)	imm8 is zero-extended, d8 is sign-extended	0	0	0	●	4	4
btst imm8,(abs16)	imm8 is zero-extended, abs16 is zero-extended	0	0	0	●	5	4
btst imm8,(abs32)	imm8 is zero-extended	0	0	0	●	7	5

Flag Changes

VF: Always 0.
CF: Always 0.
NF: Always 0.
ZF: 1 if the operation result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! AM30 AM31 btst imm8,(abs16) is only for Am32. Not usable for AM30/AM31.

BSET

Multiple Bits Test & Set

BSET Dm,(An)

Operation

mem8(An)(zero_ext)→ temp
temp&Dm:PSW
temp|Dm→mem8(An)

1. Zero-extends the contents(byte data) of (An) to 32-bit and load to the internal temporary register(temp).
2. Performs a logical AND of the contents of the temporary register(temp) and the contents of register Dm and reflects the result to PSW.
3. Performs a logical OR of the contents of the temporary register(temp) and the contents of register Dm and stores the lower 8-bit of the result in (An).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
bset Dm,(An)		0	0	0	●	2	5

Flag Changes

VF: Always 0.
CF: Always 0.
NF: Always 0.
ZF: 1 if the operation result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! All the operation by this instruction will be done during bus-locked and interruption-disabled.

! AM31 AM32 The operation corresponding to the data of cachable area in the external memory is not bus-locked.

BSET imm,Mem

Operation	
	Mem(zero_ext)→temp temp&imm:PSW temp|imm→Mem 1. Zero-extends the contents(byte data) of the memory(Mem) to 32-bit and loads to the internal temporary register(temp). 2. Performs a logical AND of the contents of the temporary register(temp) and the immediate value(imm) and reflects the result to PSW. 3. Performs a logical OR of the contents of the temporary register(temp) and immediate value(imm) and stores the lower 8-bit of the result in the memory(Mem).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
bset imm8,(d8,An)	imm8 is zero-extended, d8 is sign-extended	0	0	0	●	4	5
bset imm8,(abs16)	imm8 is zero-extended, abs16 is zero-extended	0	0	0	●	5	5
bset imm8,(abs32)	imm8 is zero-extended	0	0	0	●	7	6

Flag Changes

VF: Always 0.
CF: Always 0.
NF: Always 0.
ZF: 1 if the operation result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! All the operation by this instruction will be done during bus-locked and interruption-disabled.

! AM31 AM32 The operation corresponding to the data of cachable area in the external memory is not bus-locked.

! AM30 AM31 btst imm8,(abs16) is only for Am32. Not usable for AM30/AM31.

BCLR

Multiple Bits Test & Clear

BCLR Dm,(An)

Operation

mem8(An)(zero_ext)→temp
temp&Dm:PSW
temp&(Dm^0xFFFFFFFF)→mem8(An)

1. Zero-extends the contents(byte data) of (An) to 32-bit and load to the internal temporary register(temp).
2. Performs a logical AND of the contents of the temporary register(temp) and the contents of register Dm and reflects the result to PSW.
3. Performs a logical AND of the contents of the temporary register(temp) and the logical-inverted data of the contents of register Dm and stores the lower 8-bit of the result in (An).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
bclr Dm,(An)		0	0	0	●	2	5

Flag Changes

VF: Always 0.
CF: Always 0.
NF: Always 0.
ZF: 1 if the operation result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! All the operation by this instruction will be done during bus-locked and interruption-disabled.

! AM31 AM32 The operation corresponding to the data of cachable area in the external memory is not bus-locked.

BCLR imm,Mem

Operation

Mem(zero_ext)→temp
temp&imm:PSW
temp&(imm^0xFFFFFFFF)→Mem

1. Zero-extends the contents(byte data) of the memory(Mem) to 32-bit and load to the internal temporary register(temp).
2. Performs a logical AND of the contents of the temporary register(temp) and the immediate value(imm) and reflects the result to PSW.
3. Performs a logical AND of the contents of the temporary register(temp) and the logical-inverted data of the immediate value(imm) and stores the lower 8-bit of the result in the memory(Mem).

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
bclr imm8,(d8,An)	imm8 is zero-extended, d8 is sign-extended	0	0	0	●	4	5
bclr imm8,(abs16)	imm8 is zero-extended, abs16 is zero extended	0	0	0	●	5	5
bclr imm8,(abs32)	imm8 is zero-extended	0	0	0	●	7	6

Flag Changes

VF: Always 0.
CF: Always 0.
NF: Always 0.
ZF: 1 if the operation result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! All the operation by this instruction will be done during bus-locked and interruption-disabled.

! AM31 AM32 The operation corresponding to the data of cachable area in the external memory is not bus-locked.

! AM30 AM31 bclr imm8,(abs16) is only for Am32. Not usable for AM30/AM31.

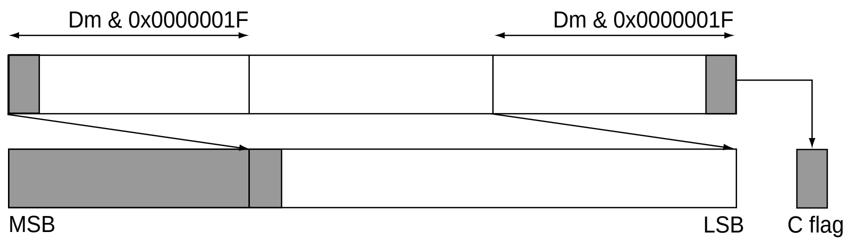
ASR Arithmetic Shift Right for Optional Bit

ASR Dm,Dn

Operation

If not (Dm&0x0000001F)=0
    Dn.lsb→CF
    (Dn >> (Dm&0x0000001F))(sign_ext)→Dn
If (Dm & 0x0000001F) =0
    PC+2→PC

Performs an arithmetic shift right on the contents of register Dn for bits specified with the lower 5 bits of register Dm and stores the result in register Dn.
No shift-operation if the contents of lower 5 bits of register Dm is '0'.
The upper 27 bits of register Dm will be ignored.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
asr Dm,Dn	Contents of lower 5 bits of Dm are other than '0'	?	●	●	●	2	1
	Contents of lower 5 bits of Dm are '0'	?	?	●	●		

Flag Changes

Contents of lower 5 bits of Dm are other than '0'
- VF: Not specified.
- CF: Reflects the bit value firstly shifted out.
- NF: 1 if bit 31 of the result is '1'. 0 otherwise.
- ZF: 1 if the result is '0'. 0 otherwise.

Contents of lower 5 bits of Dm are '0'
- VF: Not specified.
- CF: Not specified.
- NF: 1 if bit 31 of Dn is '1'. 0 otherwise.
- ZF: 1 if Dn is '0'. 0 otherwise.

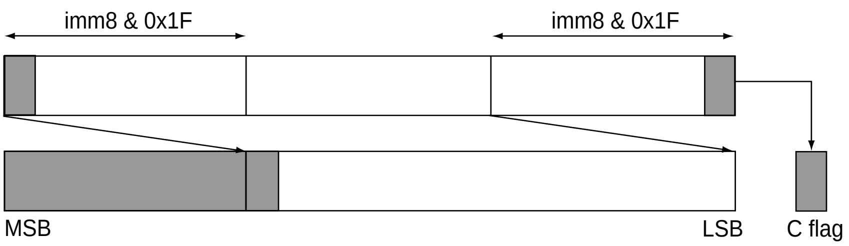
Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

ASR imm8,Dn

Operation

If not (imm8 & 0x1F)=0
    Dn.lsb→CF
    (Dn>>(imm8&0x1F))(sign_ext)→Dn
If (imm8&0x1F)=0
    PC+3→PC

Performs an arithmetic shift right on the contents of register Dn for bits specified with the lower 5 bits of the immediate value(imm8) and stores the result in register Dn.
No shift-operation if the contents of lower 5 bits of the immediate value(imm8) is '0'.
The upper 3 bits of the immediate value(imm8) will be ignored.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
asr imm8,Dn	Contents of lower 5 bits of imm8 are other than '0'	?	●	●	●	3	1
	Contents of lower 5 bits of imm8 are '0'	?	?	●	●		

Flag Changes

Contents of lower 5 bits of imm8 are other than '0'
- VF: Not specified.
- CF: Reflects the bit value firstly shifted out.
- NF: 1 if bit 31 of the result is '1'. 0 otherwise.
- ZF: 1 if the result is '0'. 0 otherwise.

Contents of lower 5 bits of imm8 are '0'
- VF: Not specified.
- CF: Not specified.
- NF: 1 if bit 31 of Dn is '1'. 0 otherwise.
- ZF: 1 if Dn is '0'. 0 otherwise.

Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

ASR

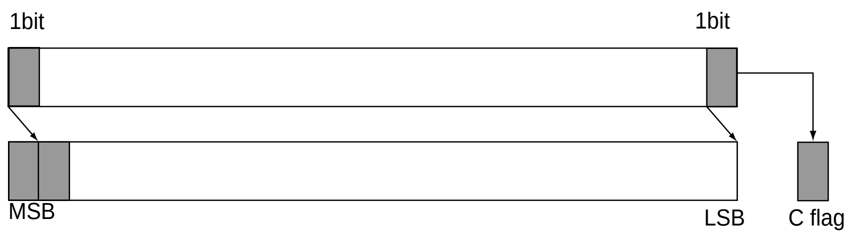
1 Bit Arithmetic Shift Right [Overwriting Instructions]

ASR Dn

Operation

Dn.lsb→CF
(Dn>>1)(sign_ext)→Dn

Performs a 1-bit arithmetic shift right on the contents of register Dn and stores in register Dn.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
asr Dn		?	●	●	●	3	1

Flag Changes

VF: Not specified.
CF: Reflects the bit value firstly shifted out.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! .This instruction is executed by overwriting the instructions and the assembler generates the following instructions.
asr 1,Dn

! The numbers of Size and Cycles are those of the instruction mentioned above.

LSR

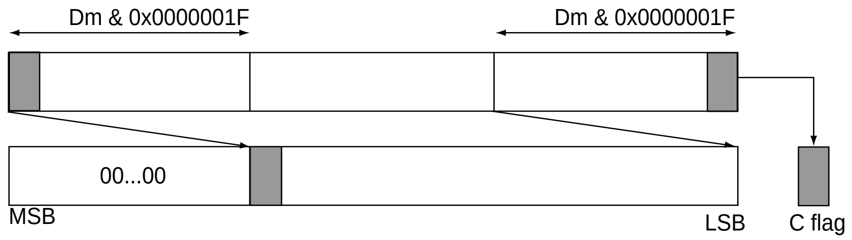
Logical Shift Right for Optional Bit

LSR Dm,Dn

Operation

If not (Dm&0x0000001F)=0
 Dn.lsb→CF
 (Dn>>(Dm&0x0000001F))(zero_ext)→Dn
If (Dm&0x0000001F)=0
 PC + 2→PC

Performs a logical shift right on the contents of register Dn for bits specified with the lower 5 bits of register Dm and stores the result in register Dn.
No shift-operation if the contents of lower 5 bits of register Dm is '0'.
The upper 27 bits of register Dm will be ignored.
'0' is input in MSB.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
lsr Dm,Dn	Contents of lower 5 bits of Dm are other than '0'	?	●	●	●	2	1
	Contents of lower 5 bits of Dm are '0'	?	?	●	●		

Flag Changes

Contents of lower 5 bits of Dm are other than '0'
- VF: Not specified.
- CF: Reflects the bit value firstly shifted out.
- NF: 1 if bit 31 of the result is '1'. 0 otherwise.
- ZF: 1 if the result is '0'. 0 otherwise.

Contents of lower 5 bits of Dm are '0'
- VF: Not specified.
- CF: Not specified.
- NF: 1 if bit 31 of Dn is '1'. 0 otherwise.
- ZF: 1 if Dn is '0'. 0 otherwise.

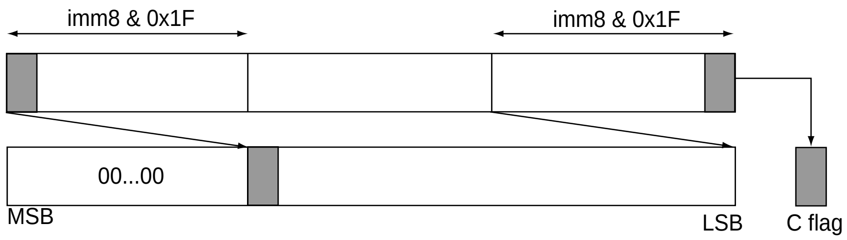
Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

LSR imm8,Dn

Operation

If not (imm8&0x1F)=0
  Dn.lsb→CF
  (Dn>>(imm8&0x1F))(zero_ext)→Dn
If (imm8&0x1F)=0
  PC+3→PC

Performs a logical shift right on the contents of register Dn for bits specified with the lower 5 bits of the immediate value(imm8) and stores the result in register Dn.
No shift-operation if the contents of lower 5 bits of the immediate value(imm8) is '0'.
The upper 3 bits of the immediate value(imm8) will be ignored.
'0' is input in MSB.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
lsr imm8,Dn	Contents of lower 5 bits of imm8 are other than '0'	?	●	●	●	3	1
	Contents of lower 5 bits of imm8 are '0'	?	?	●	●		

Flag Changes

Contents of lower 5 bits of imm8 are other than '0'
- VF: Not specified.
- CF: Reflects the bit value firstly shifted out.
- NF: 1 if bit 31 of the result is '1'. 0 otherwise.
- ZF: 1 if the result is '0'. 0 otherwise.

Contents of lower 5 bits of imm8 are '0'
- VF: Not specified.
- CF: Not specified.
- NF: 1 if bit 31 of Dn is '1'. 0 otherwise.
- ZF: 1 if Dn is '0'. 0 otherwise.

Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

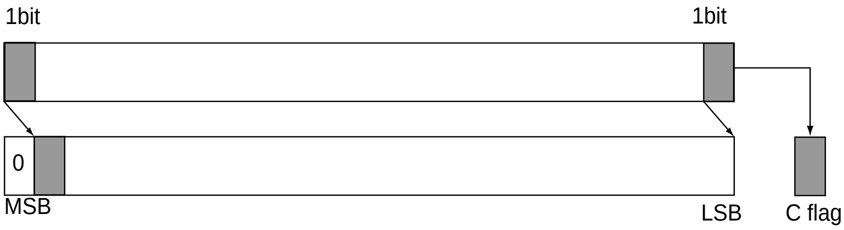
LSR 1 Bit Logical Shift Right [Overwriting Instructions]

LSR Dn

Operation

Dn.lsb→CF
(Dn>>1)(zero_ext)→Dn

Performs a 1-bit logical shift right on the contents of register Dn and stores the result in register Dn.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
lsr Dn		?	●	●	●	3	1

Flag Changes

VF: Not specified.
CF: Reflects the bit value firstly shifted out.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! .This instruction is executed by overwriting the instructions and the assembler generates the following instructions.

```
lsr 1,Dn
```

! The numbers of Size and Cycles are those of the instruction mentioned above.

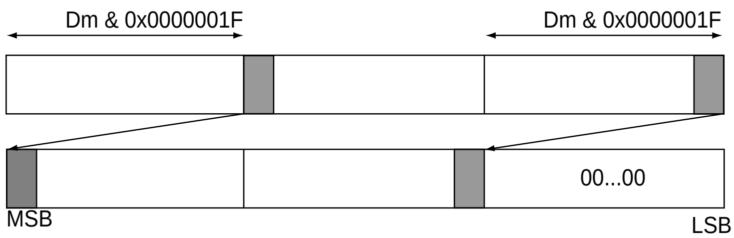
ASL Arithmetic Shift Left for Optional Bit

ASL Dm,Dn

Operation

If not (Dm&0x0000001F)=0
  Dn<<(Dm&0x0000001F)→Dn
If (Dm&0x0000001F)=0
  PC+2→PC

Performs an arithmetic shift left on the contents of register Dn for bits specified with the lower 5 bits of register Dm and stores the result in register Dn.
No shift-operation if the contents of lower 5 bits of register Dm is '0'.
The upper 27 bits of register Dm will be ignored.
'0' is input in LSB.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
asl Dm,Dn	Contents of lower 5 bits of Dm are other than '0'	?	?	●	●	2	1
	Contents of lower 5 bits of Dm are '0'						

Flag Changes

Contents of lower 5 bits of Dm are other than '0'
- VF: Not specified.
- CF: Not specified.
- NF: 1 if bit 31 of the result is '1'. 0 otherwise.
- ZF: 1 if the result is '0'. 0 otherwise.

Contents of lower 5 bits of Dm are '0'
- VF: Not specified.
- CF: Not specified.
- NF: 1 if bit 31 of Dn is '1'. 0 otherwise.
- ZF: 1 if Dn is '0'. 0 otherwise.

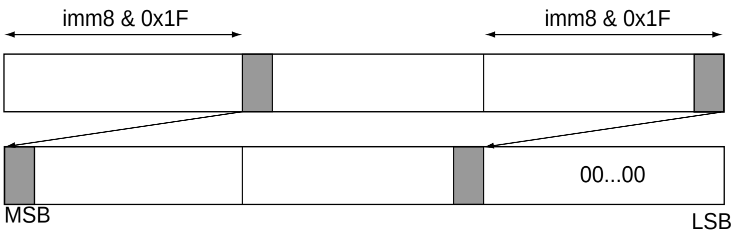
Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

ASL imm8,Dn

Operation

If not (imm8&0x1F)=0
    Dn<<(imm8&0x1F)→Dn
If (imm8&0x1F)=0
    PC+3→PC

Performs an arithmetic shift left on the contents of register Dn for bits specified with the lower 5 bits of the immediate value(imm8) and stores the result in register Dn.
No shift-operation if the contents of lower 5 bits of the immediate value(imm8) is '0'.
The upper 3 bits of the immediate value(imm8) will be ignored.
'0' is input in LSB.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
asl imm8,Dn	Contents of lower 5 bits of imm8 are other than '0'	?	?	●	●	3	1
	Contents of lower 5 bits of imm8 are '0'						

Flag Changes

Contents of lower 5 bits of imm8 are other than '0'
- VF: Not specified.
- CF: Not specified.
- NF: 1 if bit 31 of the result is '1'. 0 otherwise.
- ZF: 1 if the result is '0'. 0 otherwise.

Contents of lower 5 bits of imm8 are '0'
- VF: Not specified.
- CF: Not specified.
- NF: 1 if bit 31 of Dn is '1'. 0 otherwise.
- ZF: 1 if Dn is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

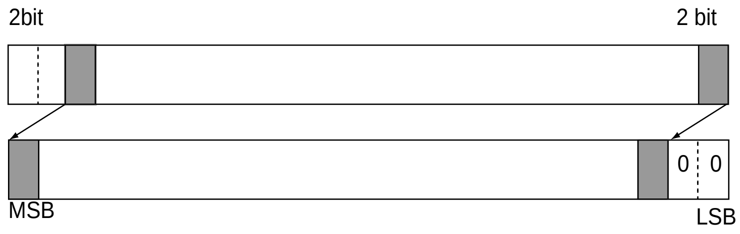
ASL2 2-Bit Arithmetic Shift Left

ASL2 Dn

Operation

(Dn<< 2)&0xFFFFFFFC→Dn

Performs a 2-bitArithmetic shift left on the contents of register Dn and stores the result in register Dn. '0' is input in LSB.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
asl2 Dn		?	?	●	●	1	1

Flag Changes

VF: Not specified.
CF: Not specified.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

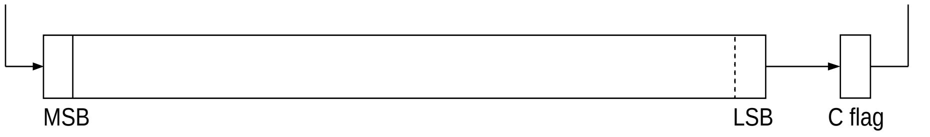
ROR 1-bit Rotate Right

ROR Dn

Operation

CF<<31→temp
Dn.lsb→CF
(Dn >>1)(zero_ext)| temp→Dn

Couples the register Dn and the carry flag(CF), perform a 1-bit rotate right on it and stores the result in register Dn.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
ror Dn		0	●	●	●	2	1

Flag Changes

VF: Not specified.
CF: Not specified.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

ROL

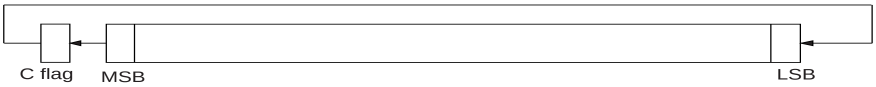
1-bit Rotate Left

ROL Dn

Operation

CF→temp
Dn.msb→CF
(Dn<<1) | temp→Dn

Couples the register Dn and the carry flag(CF), perform a 1-bit rotate left on it and stores the result in register Dn.



Assembler mnemonic	Notes	V	C	N	Z	Size	Cycle
rol Dn		0	●	●	●	2	1

Flag Changes

VF: Not specified.
CF: Not specified.
NF: 1 if bit 31 of the result is '1'. 0 otherwise.
ZF: 1 if the result is '0'. 0 otherwise.

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

Bcc

Branch on Condition Codes

Bcc label

Operation

When branch is taken,

PC d8(sign_ext)→PC

Sign-extends 8-bit displacement(d8), adds program counter(PC) and stores the result in the program counter(PC).
Stores the result in the program counter(PC). Ignore it even if the result overflows.

When branch is not taken,

PC+CodeSize→PC

Executes following instructions.

Assebler mnemonic		Notes		V	C	N	Z	Size	Cycles
beq	label	Z	Branches at Z flag set	—	—	—	—	2	3/1
bne	label	~Z	Branches at Z flag clear	—	—	—	—	2	3/1
bgt	label	~(Z | (N^V))	Branches at < (with signed)	—	—	—	—	2	3/1
bge	label	~(N^V)	Branches at ≦ (with signed)	—	—	—	—	2	3/1
ble	label	Z | (N^V)	Branches at ≧ (with signed)	—	—	—	—	2	3/1
blt	label	N^V	Branches at > (with signed)	—	—	—	—	2	3/1
bhi	label	~(C | Z)	Branches at < (no signed)	—	—	—	—	2	3/1
bcc	label	~C	Branches at ≦, C flag clear (no signed)	—	—	—	—	2	3/1
bls	label	C | Z	Branches at ≧ (no signed)	—	—	—	—	2	3/1
bcs	label	C	Branches at >, C flag clear (no signed)	—	—	—	—	2	3/1
bvc	label	~V	Branches at V flag clear	—	—	—	—	3	4/2
bvs	label	V	Branches at V flag set	—	—	—	—	3	4/2
bnc	label	~N	Branches at N flag clear	—	—	—	—	3	4/2
bns	label	N	Branches at N flag set	—	—	—	—	3	4/2
bra	label	None	Branches unconditionally	—	—	—	—	2	3

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

! "Cycles" describes the cycles of "branch"/"not branch".

! The cycles of "not branch" depend on status of an instruction queue.

Lcc Loop on Condition Codes

Lcc

Operation

When branch is taken,

LAR-4→PC

The instruction loaded to the loop instruction register (LIR) is executed and instruction fetch starts for the address loaded to the loop address register (LAR).
At the same time, 4 is subtracted from the loop address register (LAR) and the results are written into the PC. Stores the result in the program counter(PC). Ignore it even if the result overflows.
Lcc is used together with SETLB in order to increase the loop execution speed, and performs conditional branch to the top of the loop set by SETLB.

When branch is not taken,

LAR+1→PC

Executes the following instructions.

Assebler mnemonic	Notes		V	C	N	Z	Size	Cycles
leq	Z	Branches at Z flag set	—	—	—	—	1	1/2
lne	~Z	Branches at Z flag clear	—	—	—	—	1	1/2
lgt	~(Z | (N^V))	Branches at <(with signed)	—	—	—	—	1	1/2
lge	~(N^V)	Branches at ≦(with signed)	—	—	—	—	1	1/2
lle	Z | (N^V)	Branches at ≧(with signed)	—	—	—	—	1	1/2
llt	N^V	Branches at >(with signed)	—	—	—	—	1	1/2
lhi	~(C | Z)	Branches at <(no signed)	—	—	—	—	1	1/2
lcc	~C	Branches at ≦, C flag clear(no signed)	—	—	—	—	1	1/2
lls	C | Z	Branches at ≧(no signed)	—	—	—	—	1	1/2
lcs	C	Branches at <, C flag set(no signed)	—	—	—	—	1	1/2
lra	None	Branches unconditionally	—	—	—	—	1	1

Flag Changes

VF: No changes
CF: No changes
NF: No changes
ZF: No changes

The execution without corresponding to SETLB insturuction is not guaranteed.

"Cycles" describes the cycles of "branch"/"not branch".

The cycles of "not branch" depend on status of an instruction queue.

SETLB Set Loop Buffer

SETLB

Operation	mem32(PC+1)→LIR , PC+5→LAR The 4-byte instruction string and 5th byte address following to SETLB are loaded to the loop instruction register (LIR) and loop address register (LAR) respectively. SETLB is used together with Lcc in order to increase the loop (the innermost loop) execution speed. The top of the loop is set by SETLB just before the loop entrance.

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
setlb		—	—	—	—	1	1

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

! AM30 AM31 AM32 A method of storing in LIR depends on microcomputer core type(AM30/AM31/AM32).

When the instrucion strings following to SETLB are the following,

SETLB
A
B
C
D

storing in LIR is as shown below.

AM30

D	C	B	A

AM31/AM32

D	C	B	A	If LIR = 4n
C	B	A	D	If LIR = 4n+1
B	A	D	C	If LIR = 4n+2
A	D	C	B	If LIR = 4n+3

JMP Unconditional branch

JMP (An)

Operation	
	An→PC Stores the contents of register An in program counter(PC).

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
jmp (An)		—	—	—	—	2	3

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

JMP label

Operation	
	If displacement from program counter(PC) ro label is performed in 16-bit, PC+d16(sign_ext)→PC Sign-extends d16, added to PC and stores the results in PC. Ignore it even if the results overflows and stores the result in PC. If displacement from program counter(PC) ro label is performed in 32-bit, PC+d32→PC Adds the 32-bit displacement and PC and stores the results in PC. Ignore it even if the results overflows and stores the result in PC.

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
jmp label	If displacement from program counter(PC) ro label is performed in 16-bit;	—	—	—	—	3	2
	If displacement from program counter(PC) ro label is performed in 32-bit	—	—	—	—	5	3*

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

! AM30 *: "Cycles" is four.

! The assembler chooses the most suitable displacement; d16 or d32.

CALL

Subroutine Call

CALL label

Operation

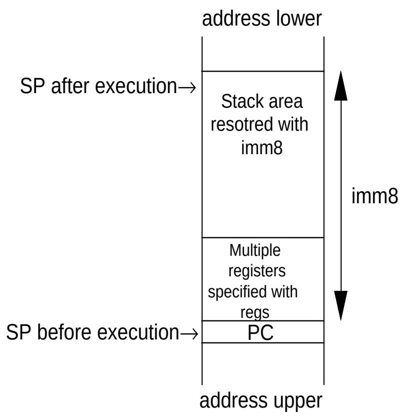
When displacement from program counter(PC) to label is performed within 16-bit,

If registers specified with "regs"= 2
PC+5→mem32(SP),
reg1→mem32(SP-4) , reg2→mem32(SP-8) , SP-imm8(zero_ext)→SP,
PC+5→MDR , PC+d16(sign_ext)→PC
If registers specified with "regs"= 11
PC+5= 2mem32(SP)
D2→mem32(SP-4), D3→mem32(SP-8), A2→mem32(SP-12),
A3→mem32(SP-16), D0→mem32(SP-20), D1→mem32(SP-24) ,
A0→mem32(SP-28), A1→mem32(SP-32),
MDR→mem32(SP-36), LIR→mem32(SP-40), LAR→mem32(SP-44) ,
SP-imm8(zero_ext)→SP, PC+5→MDR, PC+d16(sign_ext)→PC

When displacement from program counter(PC) to label is performed within 32-bit,

If registers specified with "regs"= 2
PC+7→mem32(SP),
reg1→mem32(SP-4), reg2→mem32(SP-8),
SP-imm8(zero_ext)→SP, PC+7→MDR, PC+d32→PC
If registers specified with "regs"= 11
PC+7→mem32(SP)
D2→mem32(SP-4), D3→mem32(SP-8), A2→mem32(SP-12) ,
A3→mem32(SP-16), D0→mem32(SP-20), D1→mem32(SP-24) ,
A0→mem32(SP-28), A1→mem32(SP-32) ,
MDR→mem32(SP-36), LIR→mem32(SP-40), LAR→mem32(SP-44) ,
SP-imm8(zero_ext)→SP, PC+7→MDR, PC+d32→PC

This instruction branches to the specified address after saving the PC and multiple registers for the next instruction and restoring the stack area. The immediate value(regs) specifies the multiple registers to be saved and the immediate value(imm8:zero-extended) specifies the area to be restored (bytes). (Refer to MOVM instruction for details of "regs".) CALL is used together with RET or RETF to save/restore registers and allocate/deallocate the stack area quickly during returning from subroutine. The status of the stack frame after CALL is shown at the right.



Assebler mnemonic	Notes		V	C	N	Z	Size	Cycles
call label	When displacement from program counter (PC) to label is performed within 16-bit,	registers specified with"regs"=0	—	—	—	—	5	2
		registers specified with"regs"=1	—	—	—	—	5	3
		registers specified with"regs"=2	—	—	—	—	5	4
		registers specified with"regs"=3	—	—	—	—	5	5
		registers specified with"regs"=4	—	—	—	—	5	6
		registers specified with"regs"=7	—	—	—	—	5	9
		registers specified with"regs"=8	—	—	—	—	5	10
		registers specified with"regs"=9	—	—	—	—	5	11
		registers specified with"regs"=10	—	—	—	—	5	12
		registers specified with"regs"=11	—	—	—	—	5	13
	When displacement from program counter (PC) to label is performed within 32-bit,	registers specified with"regs"=0	—	—	—	—	7	4*
		registers specified with"regs"=1	—	—	—	—	7	4*
		registers specified with"regs"=2	—	—	—	—	7	5*
		registers specified with"regs"=3	—	—	—	—	7	6*
		registers specified with"regs"=4	—	—	—	—	7	7*
		registers specified with"regs"=7	—	—	—	—	7	10*
		registers specified with"regs"=8	—	—	—	—	7	11*
		registers specified with"regs"=9	—	—	—	—	7	12*
		registers specified with"regs"=10	—	—	—	—	7	13*
		registers specified with"regs"=11	—	—	—	—	7	14*

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

! Three operands of d16, regs, imm8 are used for the bit assignment.
The assember does not specify the multiple registers to be saved and the area to be restored(regs, imm8). Pseud instruction at subroutine CALL specifies them indirectly then finally the linker executes them.
Refer to "Cross Assembler User's Manual" for details.

! Assembler selects d16 or d32 for the best.

! Refer to "Appendix Instruction Set" for operation expressions by each register specified with "regs".

! AM30 *:"Cycles" is the figures mentioned above plus 1.

CALLS

Subroutine Call

CALLS (An)

Operation	PC+2→mem32(SP) . PC+2→MDR , An→PC This instruction branches to the specified address after saving the PC for the next instruction to the stack. CALLS is used together with RETS in the case of registers to be saved and the stack area to be allocated are unclear, and to maintain compatibility(use with JSR).

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
calls (An)		—	—	—	—	2	3

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

CALLS label

Operation

When displacement from program counter(PC) to label is performed within 16-bit,

PC+4→mem32(SP), PC+4→MDR,
PC+d16(sign_ext)→PC

When displacement from program counter(PC) to label is performed within 32-bit,
PC+6→mem32(SP), PC+6→MDR ,
PC+d32→PC

This instruction branches to the specified address after saving the PC for the next instruction to the stack. Ignore them even if the result of addition is overflowed and store them into the PC. This instruction is used together with RETS in the case of registers to be saved and the stack area to be allocated are unclear, and to main compatibility(use with JSR).

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
calls label	When displacement from program counter(PC) to label is performed within 16-bit	—	—	—	—	4	3
	When displacement from program counter(PC) to label is performed within 32-bit	—	—	—	—	6	3*

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

AM30 *: "Cycles" is four.

RET

Return from subroutine

RET

Operation

If registers specified with "regs"= 2
SP+imm8(zero_ext)→SP,
mem32(SP-4)→reg1, mem32(SP-8)→reg2,
mem32(SP)→PC
If registers specified with "regs"= 11
SP+imm8(zero_ext)→SP,
mem32(SP-4)→D2, mem32(SP-8)→D3, mem32(SP-12)→A2 ,
mem32(SP-16)→A3, mem32(SP-20)→D0, mem32(SP-24)→D1 ,
mem32(SP-28)→A0, mem32(SP-32)→A1, mem32(SP-36)→MDR ,
mem32(SP-40)→LIR, mem32(SP-44)→LAR ,
mem32(SP)→PC

This instruction branches to the return address after saving the PC and multiple registers for the next instruction and restoring the stack area. The immediate value(regs) specifies the multiple registers to be saved and the immediate value(imm8:zero-extended) specifies the area to be restored (bytes). (Refer to MOVM instruction for details of "regs".) CALL is used together with RET to save/restore registers and allocate/deallocate the stack area quickly during returning from subroutine. If the subroutine does not overwrite MDR, RETF deallocate quickly.

Assebler mnemonic	Notes	V	C	N	Z	Size	Ctcles
ret	Registers specified with "regs"=0	—	—	—	—	3	5*
	Registers specified with "regs"=1	—	—	—	—	3	5*
	Registers specified with "regs"=2	—	—	—	—	3	5*
	Registers specified with "regs"=3	—	—	—	—	3	5*
	Registers specified with "regs"=4	—	—	—	—	3	5
	Registers specified with "regs"=7	—	—	—	—	3	8
	Registers specified with "regs"=8	—	—	—	—	3	9
	Registers specified with "regs"=9	—	—	—	—	3	10
	Registers specified with "regs"=10	—	—	—	—	3	11
	Registers specified with "regs"=11	—	—	—	—	3	12

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

Two operands of regs, imm8 are used for the bit assignment.
The assember does not specify the multiple registers to be saved and the area to be restored(regs, imm8). Pseud instruction at subroutine CALL specifies them indirectly then finally the linker executes them.
Refer to "Cross Assembler User's Manual" for details.

Refer to "Appendix Instruction Set" for operation expressions by each register specified with "regs".

AM30 *: "Cycles" is four.

RETF

Return from Subroutine

RETF

Operation

If registers specified with "regs"= 2
SP+imm8(zero_ext)→SP, MDR→PC ,
mem32(SP-4)→reg1, mem32(SP-8)→reg2
If registers specified with "regs"= 11
SP + imm8(zero_ext)→SP, MDR→PC ,
mem32(SP-4)→D2, mem32(SP-8)→D3, mem32(SP-12)→A2 ,
mem32(SP-16)→A3, mem32(SP-20)→D0, mem32(SP-24)→D1 ,
mem32(SP-28)→A0, mem32(SP-32)→A1, mem32(SP-36)→MDR ,
mem32(SP-40)→LIR, mem32(SP-44)→LAR

This instruction branches to the return address in MDR after saving the multiple registers and restoring the stack area. The immediate value(regs) specifies the multiple registers to be returned and the immediate value(imm8:zero-extended) specifies the area to be restored (bytes). (Refer to MOVM for details of "regs".) CALL is used together with RETF to save/restore registers and allocate/deallocate the stack area quickly during returning from subroutine.
When overwriting MDR within suburoutine, the operation for returning is not guaranteed.(Use RET.)

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
retf	Registers specified with "regs"=0	—	—	—	—	3	2
	Registers specified with "regs"=1	—	—	—	—	3	2
	Registers specified with "regs"=2	—	—	—	—	3	3
	Registers specified with "regs"=3	—	—	—	—	3	4
	Registers specified with "regs"=4	—	—	—	—	3	5
	Registers specified with "regs"=7	—	—	—	—	3	8
	Registers specified with "regs"=8	—	—	—	—	3	9
	Registers specified with "regs"=9	—	—	—	—	3	10
	Registers specified with "regs"=10	—	—	—	—	3	11
	Registers specified with "regs"=11	—	—	—	—	3	12

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

! Two operands of regs, imm8 are used for the bit assignment.
The assember does not specify the multiple registers to be saved and the area to be restored(regs, imm8). Pseud instruction at subroutine CALL specifies them indirectly then finally the linker executes them.
Refer to "Cross Assembler User's Manual" for details.

! When overwriting MDR within suburoutine, the operation for returning is not guaranteed.(Use RET.)

! Refer to "Appendix Instruction Set" for operation expressions by each register specified with "regs".

RETS Return from Subroutine

RETS

Operation

mem32(SP)→PC

Branches to the returning address stored in the stack.
RETS is used together with CALLS.
Also it is used to maintain compatibility(use with RTS).

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
rets		—	—	—	—	2	5*

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

*: "Cycles" is four.

JSR

Subroutine Call [Combination of Multiple Instructions]

JSR (An)

Operation	
	SP-4→SP , PC+2→mem32(SP), PC+2→MDR An→PC (subroutine execution) SP+4→SP Branches to the specified address after saving the PC for the next instruction to the stack.

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
jsr (An)		●	●	●	●	8	5

Flag Changes

VF: Depends on the subroutine processing
CF: Depends on the subroutine processing
NF: Depends on the subroutine processing
ZF: Depends on the subroutine processing

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! This instruction is executed by overwriting the instructions and the assembler generates the following instructions.

```
ADD     -4,SP
CALLS   (An)
ADD     4,SP
```

! The numbers of Size and Cycles are those of the instruction mentioned above. The optimization of assembler may change the cycles. Refer to "Chapter 3, Using Instructions".

JSR label

Operation

When displacement from program counter(PC) to label is performed within 16-bit,

SP-4→SP ,
PC+4→mem32(SP), PC+4→MDR ,
PC+d16(sign_ext)→PC ,
(subroutine execution)
SP+4→SP

When displacement from program counter(PC) to label is performed within 32-bit,

SP-4→SP ,
PC+6→mem32(SP), PC+6→MDR
PC+d32→PC ,
(subroutine execution)
SP+4→SP

Branches to the specified address after saving the PC for the next instruction to the stack.
Ignores even if the results overflows and stores the result in PC.

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
jsr label	When displacement from program counter(PC) to label is performed within 16-bit,	●	●	●	●	10	5
	When displacement from program counter(PC) to label is performed within 32-bit,	●	●	●	●	12	5*

Flag Changes

VF: Depends on the subroutine processing
CF: Depends on the subroutine processing
NF: Depends on the subroutine processing
ZF: Depends on the subroutine processing

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! This instruction is executed by overwriting the instructions and the assembler generates the following instructions.

```
ADD     -4,SP
CALLS   label
ADD     4,SP
```

! The numbers of Size and Cycles are those of the instruction mentioned above. The optimization of assembler may change the cycles. Refer to "Chapter 3, Using Instructions".

! AM30 *: "Cycles" is six.

RTS Return from Subroutine [Instruction replacement]

RTS

Operation	mem32(SP)→PC Branches to the returning address stored in the stack. RTS is used together with JSR to maintain compatibility.

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
rts		—	—	—	—	2	4

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

! This instruction is executed by overwriting the instructions and the assembler generates the following instructions.
rets

! The numbers of Size and Cycles are those of the instruction mentioned above.

RTI Return from Program

RTI

Operation	
Operation	mem16(SP)→PSW , mem32(SP+4)→PC , SP+8→SP Returns from the interrupt by branching to the return address stored in the stack after restoring the PSW contained in the stack.

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
rti		●	●	●	●	2	4

Flag Changes

VF: The V flag of the saved PSW.
CF: The V flag of the saved PSW.
NF: The V flag of the saved PSW.
ZF: The V flag of the saved PSW.

TRAP

Subroutine Call to a specified Address

TRAP

Operation	
	PC+2→mem32(SP), 0x40000010→PC Branches to the specified address (0x40000010) after saving the PC of the next instruction to the stack. It is used for system call (calling the OS and library).

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
trap		—	—	—	—	2	4

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

NOP No Operation

NOP

Operation	
	PC+1→PC No operation.

Assebler mnemonic	Notes	V	C	N	Z	Size	Cycles
nop		—	—	—	—	1	1

Flag Changes

VF; No changes
CF: No changes
NF: No changes
ZF: No changes

UDFnn User Extension Instruction

UDFnn Dm,Dn (nn = 00 to 15, 20 to 35)

Operation

When nn=00 to 15,

Dm op Dn→Dn

Performs an operation on the contents of register Dm and register Dn and stores the result in register Dn. The operation and flag changes are user defined.

When nn=20 to 35,

Dm op Dn

Performs an operation on the contents of register Dm and register Dn but the result is not written into register Dn and the flags are not changed.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
udfnn Dm,Dn	When nn = 00 to 15	*	*	*	*	2	User defined
	When nn = 20 to 35	—	—	—	—	2	User defined

Flag Changes

When nn = 00 to15

VF: User defined
CF: User defined
NF: User defined
ZF: User defined

When nn = 20 to 35

VF: No changes
CF: No changes
NF: No changes
ZF: No changes

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! AM31 MOVM [regs],(SP) can not be located in one instruction before this instruction.

UDFnn imm,Dn (nn=00 to 15, 20 to 35)

Operation

When nn= 00 to 15,

imm op Reg→Reg

Performs an operation on the zero-extended immediate value(imm8 or imm16) or 32-bit immediate value(imm32) and register Dn, then stores the result in register Dn.
The contents of the operation and flag changes are user defined.

When nn = 20 to 35,

imm op Reg

Performs an operation on the zero-extended immediate value(imm8 or imm16) or 32-bit immediate value(imm32) and register Dn, then stores the result in register Dn.
The result is not written into register Dn and the flags are not changed.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
udfnn imm8,Dn	When nn=00 to 15, imm8 is sign-extended	*	*	*	*	3	User defined
udfnn imm16,Dn	When nn=00 to 15, imm16 is sign-extended	*	*	*	*	4	User defined
udfnn imm32,Dn	When nn=00 to 15,	*	*	*	*	6	User defined
udfnn imm8,Dn	When nn=20 to 35, imm8 is sign-extended	—	—	—	—	3	User defined
udfnn imm16,Dn	When nn=20 to 35, imm16 is sign-extended	—	—	—	—	4	User defined
udfnn imm32,Dn	When nn=20 to 35,	—	—	—	—	6	User defined

Flag Changes

When nn = 00 to 15

VF: User defined
CF: User defined
NF: User defined
ZF: User defined

When nn = 20 to 35

VF: No changes
CF: No changes
NF: No changes
ZF: No changes

Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

AM30 "nn=20 to 35" is only for AM31/AM32. It can not be used for AM30.

AM31 MOVM [regs],(SP) can not be located in one instruction before this instruction.

UDFUnn User Extension Instruction

UDFUnn imm,Dn (nn=00 to 15, 20 to 35)

Operation

When nn=00 to 15,

imm op Reg→Reg

Performs an operation on the zero-extended immediate value(imm8 or imm16) or 32-bit immediate value(imm32) and register Dn, then stores the result in register Dn.
The contents of the operation and flag changes are user defined.

When nn=20 to 35,

imm op Reg

Performs an operation on the zero-extended immediate value(imm8 or imm16) or 32-bit immediate value(imm32) and register Dn, then stores the result in register Dn.
The result is not written into register Dn and the flags are not changed.

Assembler mnemonic	Notes	V	C	N	Z	Size	Cycles
udfnn imm8,Dn	When nn=00 to 15, imm8 is zero-extended.	*	*	*	*	3	User defined
udfnn imm16,Dn	When nn=00 to 15, imm16 is zero-extended.	*	*	*	*	4	User defined
udfnn imm32,Dn	When nn=00 to 15,	*	*	*	*	6	User defined
udfnn imm8,Dn	When nn=20 to 35, imm8 is zero-extended.	—	—	—	—	3	User defined
udfnn imm16,Dn	When nn=20 to 35, imm16 is zero-extended.	—	—	—	—	4	User defined
udfnn imm32,Dn	When nn=20 to 35,	—	—	—	—	6	User defined

Flag Changes

When nn = 00 to 15
- VF: User defined
- CF: User defined
- NF: User defined
- ZF: User defined

When nn = 20 to 35
- VF: No changes
- CF: No changes
- NF: No changes
- ZF: No changes

! Updating of PSW due to flag changes is delayed for one insturuction.
However, Bcc and Lcc instructions can evaluate flag before reflecting to PSW.

! AM30 "nn=20 to 35" is only for AM31/AM32. It can not be used for AM30.

! AM31 MOVM [regs],(SP) can not be located in one instruction before this instruction.

3 Usage Notes

Notes to Programmers

The MN1030/MN103S Series of 32-bit microcontrollers incorporates the following enhancements for boosting throughput.

• Lower cycle counts

Additional hardware bypasses and augments the five-stage pipeline to increase the execution speeds of such instructions as Lcc, SETLB, RET, and RETF.

• Higher operating frequencies

Reorganizing the pipeline stages to eliminate the bottlenecks associated with such operations as aligning and expanding load data has permitted the use of shorter clock cycles.

To help your programs to take maximum advantage of these throughput enhancements, this Chapter describes the pipeline architecture, dangerous code sequences, code sequences to avoid, and boiler plate code sequences.

1. Pipeline Architecture

   This Section covers the structure and operation of the five-stage pipeline incorporated into the MN1030/MN103S Series of 32-bit microcontrollers.

2. Dangerous Code Sequences

   This Section describes instruction variants and instruction sequences that must be strictly avoided because they lead to faulty operation.

3. Code Sequences to Avoid

   This Section describes instruction variants and instruction sequences that should be avoided not because they lead to faulty operation, but because they consume excess cycles.

4. Boiler Plate Code Sequences

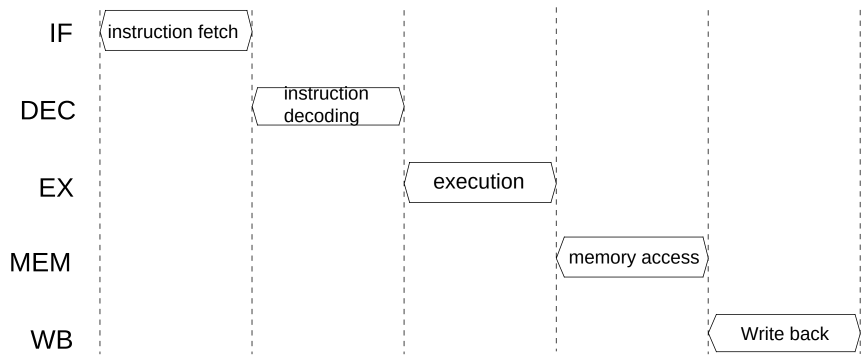
   This Section contains sample code for common programming tasks.

1 Pipeline Architecture

The 32-bit microcontrollers in the MN1030/MN103S Series boost throughput with a five-stage pipeline that, by overlapping instruction processing steps with stages operating in parallel, appears to execute a new instruction each machine cycle.

1.1 Pipeline Operation

The MN1030/MN103S pipeline has five stages.



Instruction fetch (IF):	This stage reads in the instruction from memory
Instruction decoding (DEC):	This stage decodes the instruction. For some branch instructions, it also calculates the target address.
Execution (EX):	This stage performs the calculation or calculates the target address for the decoded instruction.
Memory access (MEM):	This stage accesses memory and updates PSW flags, if required by the instruction.
Write-back (WB):	This stage stores the calculation result in a register. If the instruction reads in data from memory, this stage aligns it, extends it, and stores the result in a register.

The instruction fetch and instruction decoding stages access an instruction queue preloaded with instructions from memory. The instruction decoding stage does not start until this queue contains enough data to decode and execute the instruction. If the queue is empty (immediately after a branch, for example) or does not have all the bytes of an absolute address (abs) or immediate value (imm), the instruction decoding stage must wait at least one cycle.

Instruction queue operation can normally be safely ignored by the programmer because the hardware automatically controls it. Calculating code execution times, however, requires careful consideration of its operation.

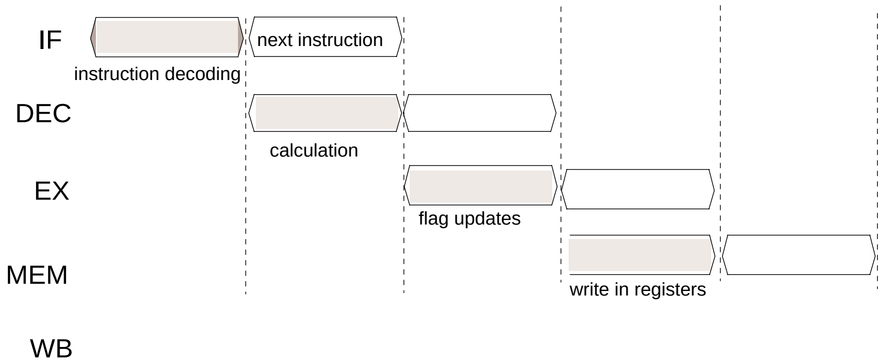
For the AM31 and AM32 cores, accessing cachable memory takes both the memory access and write-back stages. The write-back stage then aligns the data, extends it, and stores the result in a register.

1.2 Register-to-Register (RR) Operations

For register-to-register operations, the pipeline stages perform the following operations.

- DEC : This stage decodes the instruction.
- EX : This stage performs the calculation for the decoded instruction.
- MEM : If the instruction updates PSW flags, this stage does so based on the result of the preceding stage.
- WB : This stage stores the result in a register.

The instructions in this group include addition, subtraction, logical operations, and shifts.

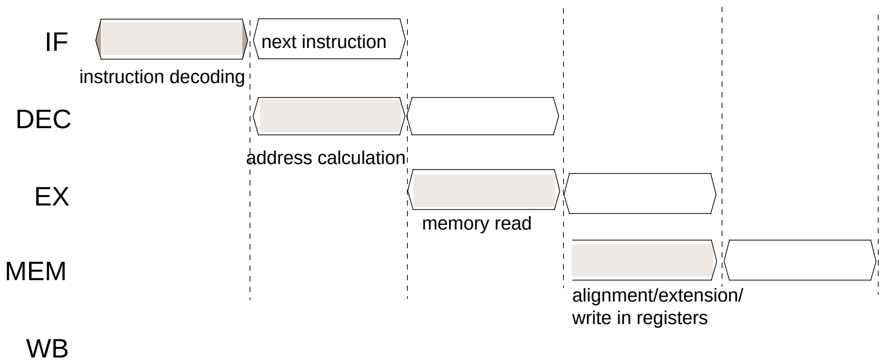


1.3 Data Load Operations

For data load operations, the pipeline stages perform the following operations.

- DEC : This stage decodes the instruction.
- EX : This stage calculates the load address and determines the corresponding address space.
- MEM : This stage loads the data from memory or, if the memory is cachable, initiates cache access.
- WB : This stage aligns the data, extends it, and stores the result in a register. If the memory is cachable, this stage reads the data from the cache before performing this operation.

The instructions in this group include data transfers from memory into a register.



1.4 Data Store Operations

For data store operations, the pipeline stages perform the following operations.

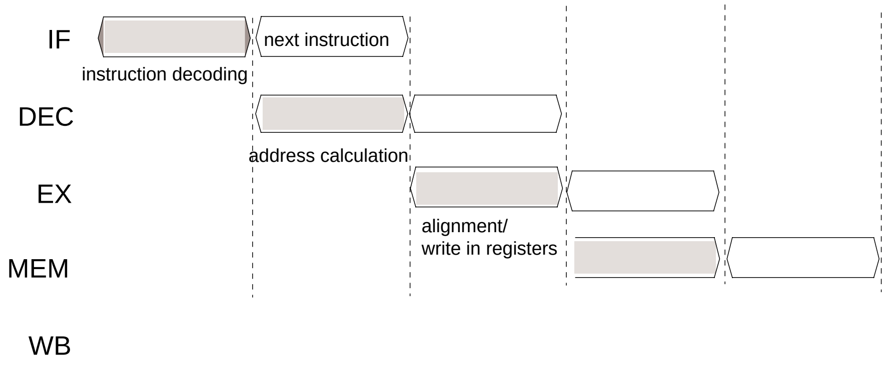
DEC : This stage decodes the instruction.

EX : This stage calculates the store address and determines the corresponding address space.

MEM : This stage aligns the data and writes it to memory or, if the memory is cachable, initiates cache access.

WB : This stage does nothing or, if the memory is cachable, writes the data to the cache.

The instructions in this group include data transfers from a register to memory.



1.5 Branching Operations

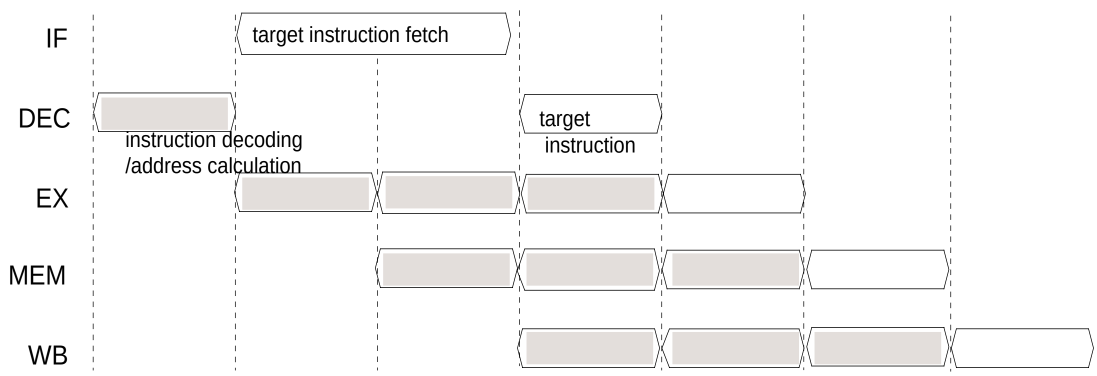
For high-speed branching operations, the pipeline stages perform the following operations.
DEC: This stage decodes the instruction and calculates the jump target address for use in fetching the instruction there during the next machine cycle.

EX : This stage does nothing.

MEM : This stage does nothing.

WB : This stage does nothing.

The instructions in this group include conditional branches.



For normal branching operations, the pipeline stages perform the following operations.

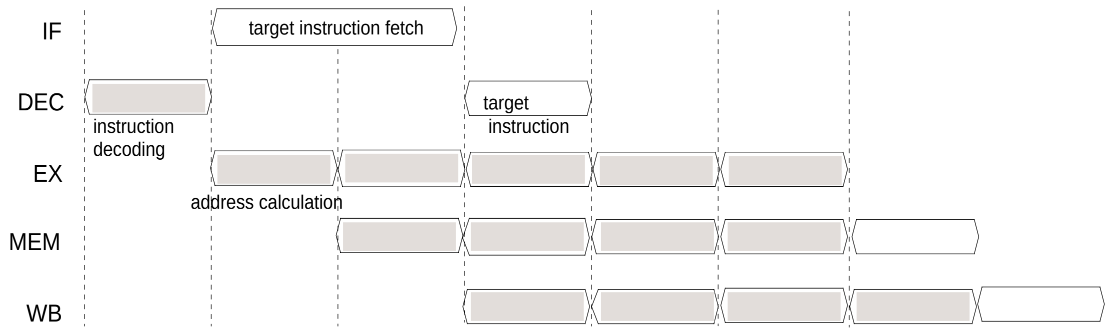
DEC : This stage decodes the instruction.

EX : This stage calculates the jump target address for use in fetching the instruction there during the next machine cycle.

MEM : This stage does nothing.

WB : This stage does nothing.

The instructions in this group include unconditional branches using register indirect addressing.



1.6 Complex Instructions

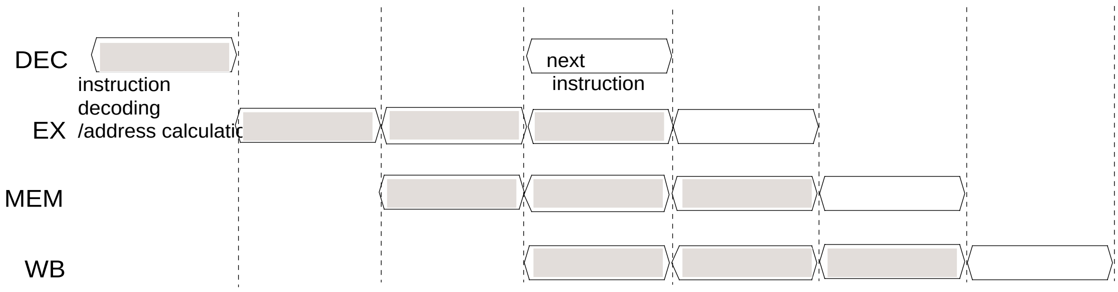
Complex instructions always require multiple cycles for the execution, memory access, and write-back stages to complete. The pipeline stages perform the following operations.

DEC : This stage decodes the instruction.

EX : This stage performs the calculation for the decoded instruction and calculates the memory access address or jump target for use in fetching the instruction there during the next machine cycle.

WB : This stage stores the calculation result in a register or aligns the data read in from memory, extends it, and stores the result in a register. For some cycles, it can be idle.

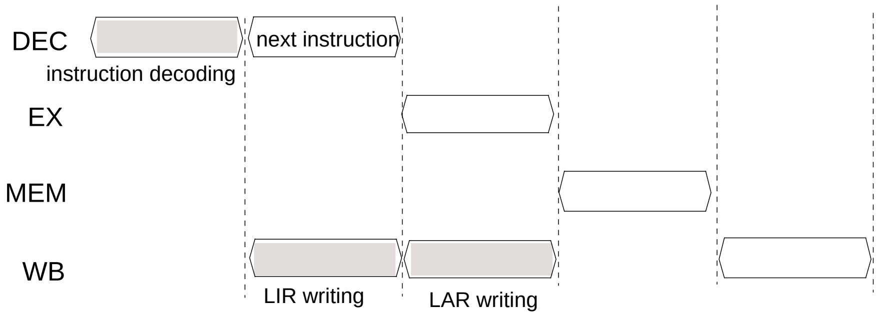
The instructions in this group include bit manipulations.



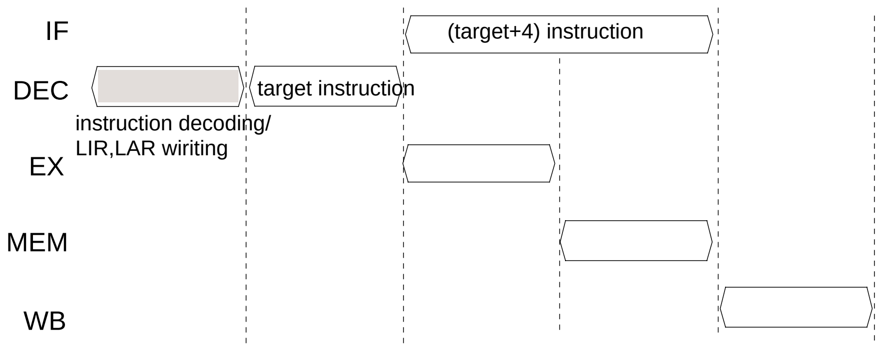
1.7 Special Instructions

Certain instructions—Lcc, SETLB, RET, and RETF, for example—reorder the later pipeline stages or bypass them with additional hardware to increase the execution speed.

Pipeline operation for SETLB



Pipeline operation for Lcc



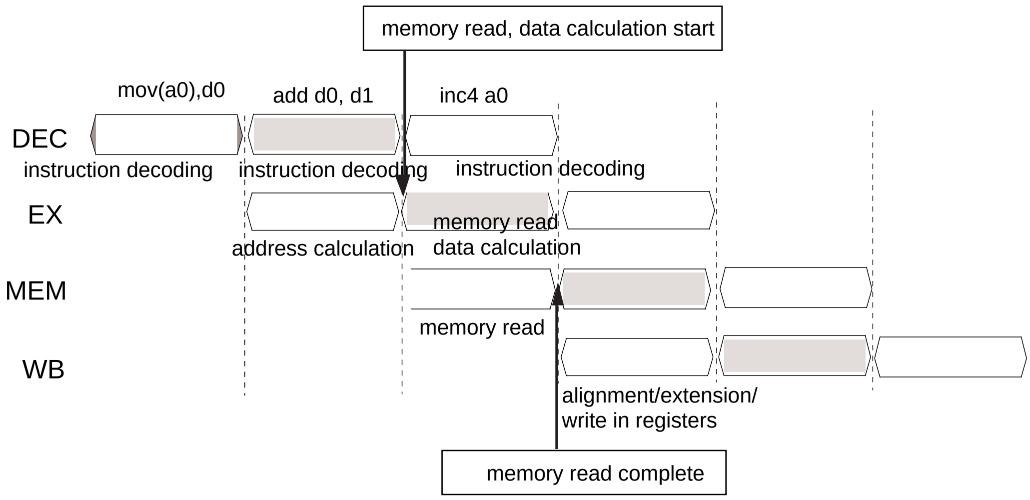
1.8 Pipeline Stall

A pipeline stall is any situation interfering with the lockstep execution of the five pipeline stages as described above—an instruction whose execution is delayed for hardware reasons or one which cannot begin execution until a preceding instruction has completed execution.
The following examples illustrate such situations arising with load instructions.
Data loaded required by next instruction Source code.

[Example]

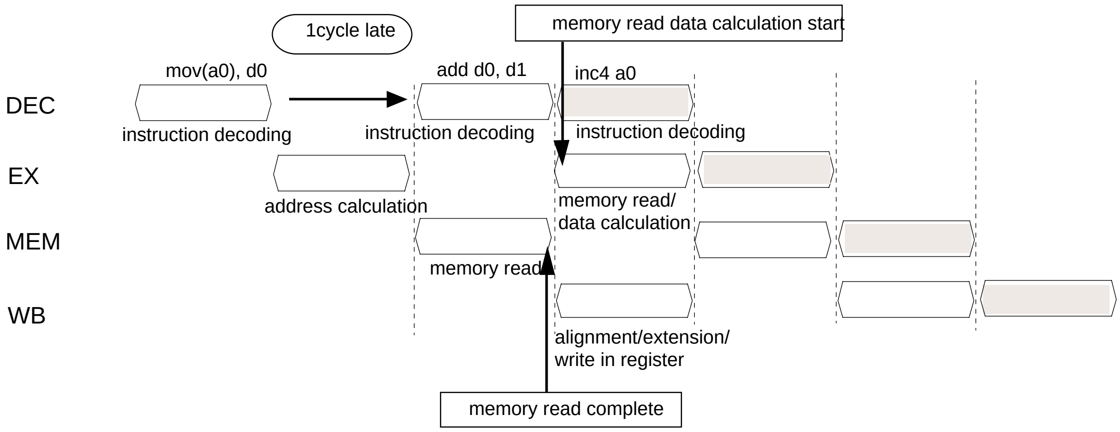
[Theoretical pipeline operation]	
mov	(a0), d0
add	d0, d1
inc4	a0

[IMPOSSIBLE pipeline operation]



In this example, the ADD instruction immediately following the MOV instruction requires the data from data register D0. This arrangement does not execute as expected because the ADD instruction's execution stage is simultaneous with the MOV instruction's memory access stage, and the data from the latter is not available until one cycle later. As a result, the ADD instruction would use the old contents of D0—not what was intended.

[Actual pipeline operation]



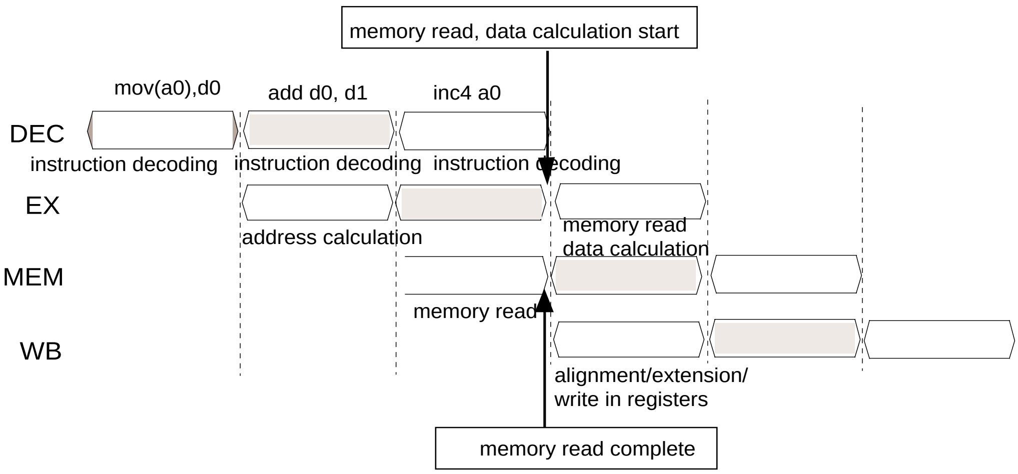
To ensure proper operation, therefore, the hardware inserts a 1-cycle delay so that the ADD instruction does not access D0 until the preceding MOV instruction has finished loading it from memory.
This type of gap in pipeline operation is called a pipeline stall.

Pipeline stalls reduce throughput and thus execution speed. One way to eliminate such pipeline stalls is to separate the two instructions with another—by moving in INC4, in our example. The resulting delay allows the MOV instruction to load D0 from memory by the time that the ADD instruction accesses that register, so the pipeline does not stall. It goes without saying, of course, that the inserted instruction (INC4 here) must not trigger a pipeline stall with the MOV instruction.

[Example]

```
[ Rearranged to eliminate pipeline stall ]
        mov     (a0), d0
        inc4    a0
        add     d0, d1
```

[Pipleline operation (for above example)]



2 Dangerous Code Sequences

The following table lists instruction variants and instruction sequences that must be strictly avoided because they lead to faulty operation.

Leading instruction	Following instruction	Relative position	Recommend	Section	Cores
Load/store instruction (d8,An), (d16,An),(d32,An), (d8,SP), (16,SP),(d32, SP), MOV/MOVBU/MOVB/ MOVHU/MOVH with addressing mode of (Di, An)	---	---	The result of the address calculation must be in the same memory address space as the address.	2-1	AM30 AM31
IE/IM writing instruction AND imm16,PSW, OR imm16, PSW, MOV Dm, PSW	---	---	Instructions writing to IE and IM bits (after 2 cycles)	2-2	AM30 AM31 AM32
Flag update instruction CLR,ADD(except imm,SP) ADDC,SUB,SUBC,MULU, DIV,DIVU,INC Dn,CMP, AND(except imm16,PSW), OR(except imm16,PSW), XOR, NOT, BTST, BSET, BCLR, ASR, LSR, ASL, ASL2, ROR, ROL, Users' defined updating flags	Flag reference ADDC Dm,Dn, SUBC Dm,Dn ROR Dm, ROL Dn, Users' optional for flag reference*1	following	required: more than 1 cycle	2-3	AM30 AM31 AM32
Flag writing instruction AND imm16, PSW, OR imm16, PSW, MOV Dm, PSW	Flag reference ADDC Dm,Dn SUBC Dm,Dn ROR Dn ROL Dn Users' defned for flag reference*	following	required: more than 2 cycles	2-4	AM30 AM31 AM32
A0 writing instruction MOV with A0 writing ADD writing result in A0 SUB writing result in A0, INC, A0, INC4 A0	MUL/MULU	following	required: more than 2 cycles	2-5	AM30
CALLS label, JSR label	---	---	CALLS(d16,PC), JSR(d16,PC) is not available.	2-6	AM31
MOVM [regs], (SP)	users' optional	following	required: more than 1 cycle	2-7	AM31
BSET/BCLR	---	---	bus unlock: cachable region /	2-8	AM31 AM32

* Availability varies with model.

The descriptions in this section have the following components.

[Description]

These portions describe the problems and include pipeline operation diagrams.

[Icons]

Warning: This icon indicates warnings that must always be observed.

AM30 AM31 AM32 These indicate the applicable microcontroller cores.

[Example]

These portions illustrate the problems with specific coding examples—both problematic versions that do not execute as expected and revised versions guaranteed to execute properly.

[Applicable Instructions]

These portions list the instruction variants and instruction sequences exhibiting the problem.

The coding examples labeled “problematic” below do not necessarily always fail to yield the expected results. If two instructions, A and B, must always be separated by at least one cycle, they will still execute as expected together if memory location, interrupts, and other factors conspire to delay the supply of instruction B to the pipeline. For “worst case” reliability, however, always use the revised version instead.

For ease of clarification, the coding examples below use NOPs to separate instructions, but these are not the only candidates. Any instruction in the vicinity that can be moved into the gap without changing program logic or presenting ordering problems of its own makes a better candidate.

2.1 Load/Store Instructions

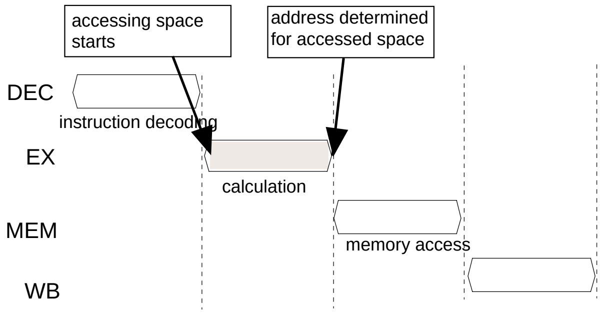
[Description]

Notes

AM30

AM31

To speed memory access, load/store instructions assume that the memory space accessed is that of the base address, so determine it from that in an operation that runs parallel to the address calculation. If the result is in a different memory space, however, faulty operation results. To avoid this problem, always make sure that the base address and the result are in the same memory space.



[Example]

Problematic Version

```
mov   0x20, a0
mov   (0x80000000,a0), d0
```

	base address	address calculation result
	a0=0x00000020	a0+0x8000000 =0x8000020
ROM/RAM/Flash memory	internal data RAM space	external memory space
with Cache	cachable region for data	uncachable region

different

This code does not execute as expected because the base address and the result are in different memory spaces.

Revised Version

```
mov   0x80000000, a0
mov   (0x20, a0), d0
```

	base address	address calculation result
	a0=0x00000020	a0+0x8000000 =0x8000020
ROM/RAM/Flash memory	internal data RAM space	external memory space
with Cache	cachable region for data	uncachable region

same

This code executes without problem because the base address and the result are in the same memory space.

[Applicable Instructions]

MOV, MOVBU, MOVB, MOVHU, and MOVH with the following addressing modes

Addressing Modes	Base Address
(d8,An)/(d16,An)/(d32,An)	An
(d8,SP)/(d16,SP)/(d32,SP)	SP
(Di,An)	An

2.2 Instructions Writing to IE and IM Bits

Notes

AM30

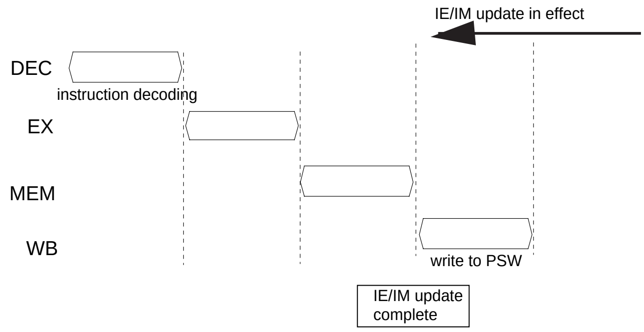
AM31

AM32

[Description]

The use of a pipeline means that instructions writing to the interrupt enable (IE) and interrupt mask (IM) bits in the Processor Status Word (PSW) do not take effect until the write-back stage, two cycles after the execution stage. To put it another way, these instruction must precede by two cycles the code segment where such changes—enabling or disabling interrupts, for example—are to take effect.

Within two cycles after executing an instruction to change the IE and IM bits, an interrupt can be accepted according to the setting before the changes. In this case, the values of PSW saved in the stack on the interrupt are based on the values after the instruction changed.



[Example]

Problematic Version

```
        and     0xf7ff,psw
        mov     (a0),d0         or
        and     0xf7ff,psw
        nop
        mov     (a0),do
```

Although the intention is to disable interrupts, because of the delay in updating the IE bit in the PSW, interrupts are still enabled when the MOV instruction starts executing.

Revised Version

```
        and     0xf7ff,psw
        nop
        nop
        mov     (a0),do
```

Here the MOV instruction can never be pre-empted by an interrupt.

[Applicable Instructions]

AND imm16,PSW, OR imm16,PSW, MOV Dm,PSW

2.3 Sequences Updating and Referencing PSW Flags

Notes

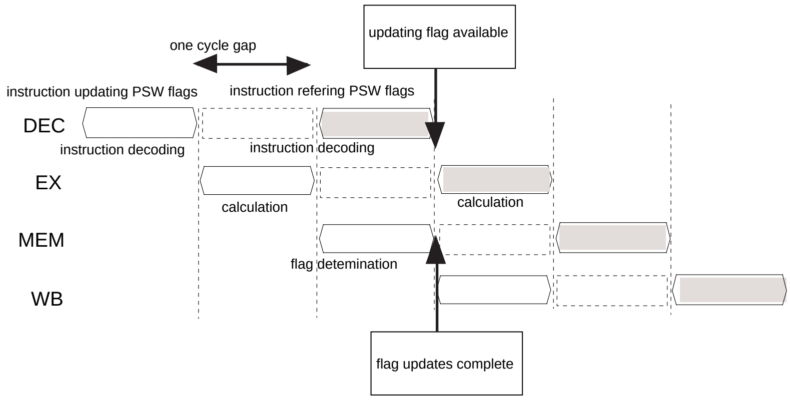
AM30

AM31

AM32

[Description]

The use of a pipeline means that instructions updating PSW flags do not take effect until the memory access stage, one cycle after the execution stage. There must, therefore, be a gap of at least one cycle before any instructions can correctly reference those PSW flags.



[Example]

[Problematic Version]

```
add     d0,d1
addc    d2,d3
```

Because the ADDC instruction starts executing before the ADD instruction has updated the carry flag, it uses the old contents of the carry flag.

[Revised Version]

```
add     d0,d1
nop
addc    d2,d3
```

The ADDC instruction uses the properly updated contents of the carry flag.

[Applicable Instructions]

<Instructions setting PSW flags>	CLR; ADDC; SUB; SUBC; MUL; MULU; DIV; DIVU; CMP; XOR; NOT; BTST; BSET; BCLR; ASR; LSR; ASL; ASL2; ROR; ROL; ADD (except ADD imm, SP) AND (except AND imm16, PSW) INC Dn OR (except OR imm16, PSW) User defined instructions setting PSW flags*
<Instructions referencing PSW flags>	User defined instructions referencing PSW flags*
	(Availability varies with model.)

2.4 Sequences Writing to and Referencing PSW Flags

Notes

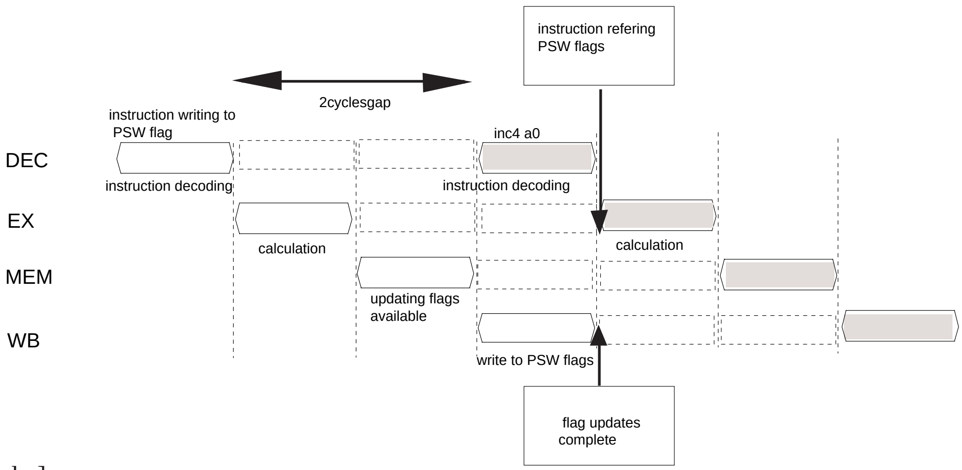
AM30

AM31

AM32

[Description]

The use of a pipeline means that instructions writing to PSW flags do not take effect until the write-back stage, two cycles after the execution stage. There must, therefore, be a gap of at least two cycles before any instructions can correctly reference those PSW flags.



[Example]

Problematic Version

```
mov     d0,psw
addc    d2,d3          or

mov     d0, psw
nop
addc    d2,d3
```

Because the ADDC instruction starts executing before the MOV instruction has updated the carry flag, it uses the old contents of the carry flag.

[Revised Version]

```
mov     d0,psw
nop
nop
addc    d2,d3
```

The ADDC instruction uses the properly updated contents of the carry flag.

[Applicable Instructions]

<Instructions writing to PSW flags>	AND imm16, PSW; OR imm16, PSW; MOV Dm, PSW
<Instructions referencing PSW flags>	User defined instructions referencing PSW flags*
	(* Availability varies with model.)

2.5 MUL/MULU after Write to A0

Notes

AM30

[Description]

To boost execution speed, the AM30 versions of the MUL and MULU instructions modify pipeline operation in a way that interacts with the address register A0. As a result, there must be a gap of at least two cycles after any instruction that writes to A0 before these two instructions. Example

[Example]

[Problematic Version]

```
mov     0x12345678,a0
mul     d0,d1                or

mov     0x12345678,a0
nop
mul     d0,d1
```

Assembling the source file (program.asm) containing the above code with the -mno-nopmul command line option produces object code reflecting the source code exactly as written—

```
as103 -mno-popmul program.asm
```

that is, no additional NOPs. This object code, however, does not always perform as intended.

[Revised Version]

```
mov     0x12345678,a0
nop
nop
mul     d0,d1
```

Inserting the necessary NOPs or assembling the source file with the -mnopmul command line option produces object code with the necessary two-cycle gap and thus guaranteed to execute properly.

```
as103 -mnopmul program.asm
```

specifying these options is to insert two NOPs between Instruction writing to Ao and MUL/NULU and generate the object files.

The assembler defaults to the -mnopmul command line option.

[Applicable Instructions]

<Instruction writing to A0> all MOV, ADD, and SUB instructions writing to A0; INC A0; INC4 A0

2.6 Displacements with CALLS and JSR (AM31 Only)

Notes

AM31

[Description]

To boost execution speed, the AM31 versions of the CALLS and JSR instructions modify pipeline operation in a way that prevents the use of 16-bit displacements. The CALLS (d16, PC) and JSR (d16, PC) variants are not available, so use the d32 variants instead.

The command line option for disabling the use of such 16-bit displacements is -mlongcalls.

[Example]

Assembling the source file (program.asm) containing the above code with the -mlongcalls command line option produces object code reflecting the source code exactly as written—

```
as103  -mlongcalls  program.asm
```

specifying these option is to generate the object files as CALL(d32,PC) or JSR(d32,PC) without using CALLS(d16,PC) ir JSR(d16,PC)

[Applicable Instructions]

CALLS (d16,PC), JSR (d16,PC)

2.7 User Defined Instructions after MOVM [regs], (SP) (AM31 Only)

Notes

AM31

[Description]

To boost execution speed, the AM31 version of MOVM [regs], (SP) modifies pipeline operation in a way that introduces a structural dependency with immediately following user defined instructions (UDFnn and UDFUnn). As a result, there must be a gap of at least one cycle between the two.

[Example]

[Problematic Version]

```
movm     [regs], (SP)
udf00    d0, d1
```

Assembling the source file (program.asm) containing the above code with the -mno-nopmovm command line option produces object code reflecting the source code exactly as written—

```
as103  -mno-nopmovm  program.asm
```

that is, no intervening NOP. This object code, however, does not always perform as intended.

[Revised Version]

```
movm     [regs], (SP)
nop
udf00    d0, d1
```

Inserting the necessary NOP or assembling the source file with the -mnopmovm command line option produces object code with the necessary one-cycle gap and thus guaranteed to execute properly.

```
as103  -mnopmovm  program.asm
```

Specifying these options is to insert one NOP between MOVM[regs],(SP) and UDFnn/UDFUnn and generates the object files.

The assembler defaults to the -mnopmovm command line option.

[Applicable Instructions]

<Leading instruction> MOVM[regs], (SP)

<User defined instructions> (UDFnn and UDFUnn)*

(* Availability varies with model.)

2.8 BSET and BCLR with Cachable External Memory

Notes

AM31

AM32

[Description]

The bit manipulation instructions BSET and BCLR, because of their importance to semaphore management, normally lock the bus for exclusive CPU use while they execute. The AM31 and AM32 versions, however, do not lock the bus for operations on data in the cachable region of external memory because the operations bypass the bus, accessing the cache, not the memory itself.

[Applicable Instructions]

BSET, BCLR

3 Code Sequences to Avoid

The following table lists instruction variants and instruction sequences that should be strictly avoided because they lead to pipeline stalls and thus lower throughput.

Leading instruction	Following instruction	Relativeposition	Recommend	Section	Cores
Instructions needs to execute with high-speed	-	-	allocate in cachable region of locked external memory space	3-1	AM30/ AM31 AM32
Loard/Store (d8,An),(d16,An), (d32,An), (d8,SP) (d16,SP),(d32,SP), MOV/MOVB/ MOVB/MOVHU/ MOVH with addressing mode of (Di,An)	-	-	The result of the address calculation should be in the same memory address space as the address in the base register.	3-2	AM32
Lcc	all instructions	target(branch)	Arrange short instructions with low per-cycle consumption rates at branch target (start of loop) at 8n-4+m for small values of m.	3-3(1)	AM30/ AM31 AM32
target(branch except Lcc				3-3(2)	AM30/ AM31AM32
Bcc and Lcc		following	Arrange short instructions with low per-cycle consumption rates at branch target at 8n+m for small values of m.	3-3(3)	AM30/ AM31 AM32
all except target (branch)			Arrange short instructions with low per-cycle consumption rates immediately after branch instruction.	3-3(4)	AM30/ AM31 AM32
Loard (DCBYPS=0 in cachable region of the external memory space)	instructions using loaded data, without imm32/d32/abs32	following	insert 2 cycles	3-4(1)	Am31/ AM32
	instructions using loaded data, with d32		insert 1 cycle	3-4(2)	Am31/ AM32
	instructions storing loaded data without d32/abs32		insert 1 cycle	3-4(3)	Am31/ AM32

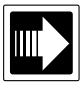
Leading instruction	Following instruction	Relativeposition	Recommend	Section	Cores
Loard (DCBYPS=1 in cachable region of the extenal memory space, uncachable region of the extenal memory, ROM space, RAM space, I/O space, or uncachable in the external memory space)	instructions using word data loaded by LDUSE bit=1 , without imm32/d32/abs32	following	insert 2 cycles	3-4(4)	AM30/AM31 AM32
	instructions using word data loaded by LDUSE bit=1, with d32		insert 1 cycle	3-4(5)	AM30/AM31 AM32
	instructions using word data loaded,by LDUSE bit=0, without imm32/d32/abs32		insert 1 cycle	3-4(6)	AM30/AM31 AM32
	instructions using byte/half-byte data loaded without imm32/d32/ abs32		insert 2 cycles	3-4(7)	AM30/AM31 AM32
	instructions using byte/half-byte data loaded with d32		insert 1 cycle	3-4(8)	AM30/AM31 AM32
	instructions storing loaded data, withd32/abs32		insert 1 cycle	3-4(9)	AM30/AM31 AM32
DIV/DIVU (dividend equals as 0)	instructionsusing contents of Dn for leading DIV/DIVU, without imm32 d32/abs32	following	insert 2 cycle	3-5(1)	AM30/AM31 AM32
	instructions using contents of Dn for leading DIV/DIVU		insert 1 cycle	3-5(2)	AM30/AM31 AM32
	instructions storing contents of Dn for leading DIV/DIVU, without imm32/d32/abs32		insert 1 cycle	3-5(3)	AM30/AM31 AM32
SETLB	Lcc	following	insert 3 cycles	3-6(1)	AM30/AM31 AM32
MOVM(SP),regs		following	insert 3 cycles	3-6(2)	AM30/AM31 AM32
RET/RETF		return			
MOVM(SP), regs	SETLB	following	insert 1 cycle	3-7	AM30/AM31 AM32
RET/RETF		return			
MOV Dm,MDR EXT, MUL/MULU	RETF	following	insert 3 cycles	3-8(1)	AM30/AM31 AM32
DIV/DIVU		following	insert 2 cycles	3-8(2)	AM30/AM31 AM32
CALL/CALLS		target (branch)			
MOVM(SP),regs		following	insert 3 cycles	3-8(3)	AM30/AM31 AM32
RET/RETF		return			
CALL/CALLS	MOV MDR,Dn DIV/DIVU	target(branch)	insert 2 cycles	3-9	AM30/AM31 AM32

The descriptions in this section have the following components.

[Description]

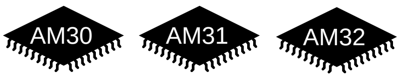
These portions describe the recommendations—with pipeline operation diagrams as necessary.

[Icons]



High-speed

High-speed: This icon indicates a recommendation for boosting throughput by avoiding pipeline stalls.



These indicate the applicable microcontroller cores.

[Example]

These portions illustrate the recommendations with specific coding examples—both problematic versions that produce pipeline stalls and revised versions that run faster by avoiding pipeline stalls.

[Applicable Instructions]

These portions list the instruction variants and instruction sequences covered by the recommendation.

For ease of clarification, the coding examples below avoid pipeline stalls by moving INC instructions, but these are not the only candidates. Any instruction in the vicinity that can be moved into the gap without changing program logic or presenting ordering problems of its own makes a suitable candidate.

3.1 Time-Critical Code

High-speed

AM30 AM31 AM32

[Description]

To extract maximum performance from the pipeline, the hardware must be able to fetch instructions fast enough to keep the instruction queue supplied. This means, in turn, locating interrupt handlers and other portions requiring the fastest response as well as libraries and other frequently used portions in the internal program ROM, internal RAM, or cachable portions of external memory. Models with onboard cache, for example, can enjoy a performance boost by locking the cache with the external memory containing such code.

[Example]

```
_TEXT      section     CODE, PUBLIC,1
sub_func
           mov         0,d0
rts
end
```

The following command line relocates the machine code for the above source code (program.asm) to boost its execution speed.

```
as103-o program.rf program.asm
ld103-T@CODE=50000000 -o program.ex program.rf
```

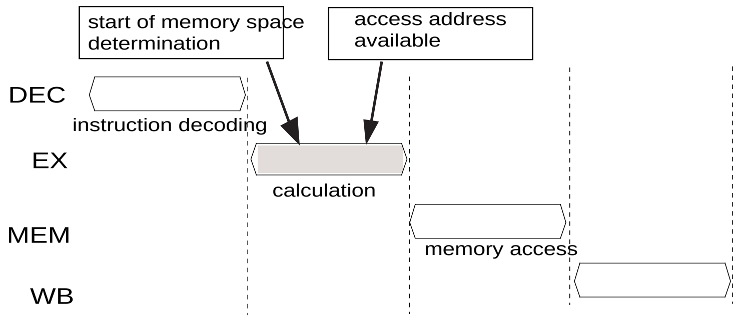
The above command line generates an executable file (program.ex) with the corresponding machine code starting at address 0x50000000. For a model with onboard ROM, RAM, and Flash memory, the best choice of address is within the internal program ROM; for a model with onboard cache, in the cachable portions of external memory.

3.2 Load/Store Instructions

High-speed

[Description]

To speed memory access, load/store instructions assume that the memory space accessed is that of the base address, so determine it from that in an operation that runs parallel to the address calculation. If the result is in a different memory space, however, the hardware must repeat the memory space determination, a step that takes an additional cycle. For maximum throughput, always make sure that the base address and the result are in the same memory space.



[Example]

[Problematic Version]

```
        mov        0x20,a0
        mov        (0x80000000,a0),d0
```

	base address	address calculation result
	a0=0x00000020	a0+0x8000020
ROM/RAM/Flash memory	internal data RAM space	external memory space
with Cache	cachable region for data	uncachable region

different

This code executes slower because the base address and the result are in different memory spaces.

[Revised Version]

```
        mov        0x80000000,a0
        mov        (0x20,a0).d0
```

	base address	address calculation result
	a0=0x80000000	0x20+a0=0x80000020
ROM/RAM/Flash memory	internal data RAM space	external memory space
with Cache	cachable region for data	uncachable region

same

This code executes at maximum speed because the base address and the result are in the same memory space.

[Applicable Instructions]

MOV, MOVBU, MOVB, MOVHU, and MOVH with the following addressing modes

Addressing mode	base address
(d8,An)/(d16,An)/(d32,An)	An
(d8,SP)/(d16,SP)/(d32,SP)	SP
(Di,An)	An

3.3 Instructions Following Branch and Other Instructions

High-speed

AM30

AM31

AM32

(1) Lcc Branch Targets

[Description]

Because the instruction fetch hardware uses 8-byte alignment and the Loop Instruction Register (LIR) holds the first four bytes of the loop, arranging short instructions with low per-cycle consumption rates at the branch target (start of loop) at addresses 8n-4+m for small values of m produces higher throughput.

[Example]

```
        align       8
        mov         0x00,a0
        clr         d0
        setlb
        mov         d0,(a0)          :instruction at address 8n-4
        mov         d0,(0x10,a0)
        mov         d0,(0x20,a0)     :instruction at address 8n
        mov         d0,(ox30,a0)
        inc4        a0
        cmp         0x10,a0
        llt
```

Instruction at address 8n The above example shows a branch target at an address four bytes before a fetch boundary (8n-4). The SETLB instruction therefore loads the Loop Instruction Register (LIR) with the two instructions MOV D0, (A0) and MOV D0, (0x10,A0) and the Loop Address Register (LAR) with the boundary address (8n). When the Lcc instruction (LLT in this example) starts executing these instructions, it also simultaneously triggers a fetch of a full eight bytes, ensuring an ample supply of instructions and thus preventing a pipeline stall—for models using onboard ROM for instructions and onboard RAM for data and onboard cache models with no hits, anyway.

[Applicable Instructions]

<Leading instruction>   Lcc
<Target(branch)>   any instructions

High-speed

AM30

AM31

AM32

(2) Branch Targets Other than Lcc

[Description]

Because the instruction fetch hardware uses 8-byte alignment, arranging short instructions with low per-cycle consumption rates at the branch target at addresses 8n+m for small values of m produces higher throughput.

[Example]

```
            align   8
            nop
            nop
            nop
            nop
            nop
            mov     0x00,a0
            clr     d0
LABEL       mov     d0,(a0)          :instruction at address 8n
            mov     d0,(0x10,a0)
            mov     d0,(0x20,a0)
            mov     d0,(0x30,a0)
            inc4    a0
            cmp     0x10,a0
            bit     LABEL
```

The above example shows a branch target at a fetch boundary (8n). When the branch instruction (BLT in this example) starts executing these instructions, it triggers a fetch of a full eight bytes, ensuring an ample supply of instructions and thus preventing a pipeline stall—for models using onboard ROM for instructions and onboard RAM for data and onboard cache models with no hits, anyway.

[Applicable Instructions]

<Leading instruction> Bcc, JMP, CALL, CALLS, RET, RETF, RETS, RTI, TRAP

<Target(branch)> Any instructions

High-speed

AM30 AM31 AM32

(3) Instructions Following Bcc or Lcc

[Description]

Arranging short instructions with low per-cycle consumption rates after a conditional branch instruction maximizes supply, producing higher throughput.

[Example]

```
blt     LABEL
mul     d0,d1     Two-byte instruction requiring multiple cycles to execute
```

The above example shows a MUL instruction executed when the branch instruction (BLT in this example) fails. Because the instruction requires multiple cycles, the core has time to fetch more instruction bytes, ensuring a better supply and thus higher throughput.

[Applicable Instructions]

< Leading instruction> Bcc or Lcc

<Target(branch)> Any instructions

High-speed

AM30 AM31 AM32

(4)Instructions Following Non-Branch Instructions

[Description]

Grouping long instructions together risks having instruction queue consumption outstrip supply. For higher throughput, try to even out the consumption rate.

[Example]

```
mov     0x12345678,d0     six-byte instruction
mov     (a0),d1           one byte instruction
inc     d0                one byte instruction
add     d0,d1             one byte instruction
mov     0x9abcdef0,d2     six-byte instruction
```

The above example shows such a separation. This arrangement evens out the consumption rate, preventing pipeline stalls with the longer instructions and thus delivering higher throughput.

[Applicable Instructions]

<Leading instruction> Any instruction except Bcc, Lcc, JMP, CALL, CALLS, RET, RETF, RETS, RTI,TRAP

<Target(branch) Any instructions

3.4 Instructions Following Load Instructions

Load instructions normally require two cycles (memory access and write-back stages) to retrieve the data from memory, so closely following instructions requiring that data for operands can stall the pipeline.
Load instructions accessing cachable external memory initiate the cache access during the memory access stage and retrieve the data from the cache during the write-back stage. Those accessing other memory types (uncachable external memory, internal ROM, internal RAM, internal I/O region, or external memory for models without onboard cache) retrieve the data from memory during the memory access stage and align, extend, and write that data to a register during the write-back stage.

Setting the DCBYPS bit in the core's Memory Control Register (MEMCTRC) to “1” can, for models with operating frequencies low enough, shorten the cache initiate and retrieve process to a single cycle in certain circumstances. For further details, refer to the documentation for the particular device.
Setting the LDUSE bit in MEMCTRC to “0” makes a word (imm32, d32, or abs32) available for a calculation (address or otherwise) in following instructions after a single cycle for the following memory types (abbreviated to “any” in the rest of this section): cachable external memory with DCBYPS = 1, uncachable external memory, internal ROM, internal RAM, internal I/O region, and external memory for models without onboard cache—in other words, all types except cachable external memory with DCBYPS = 0. For further details, refer to the documentation for the particular device.

High-speed

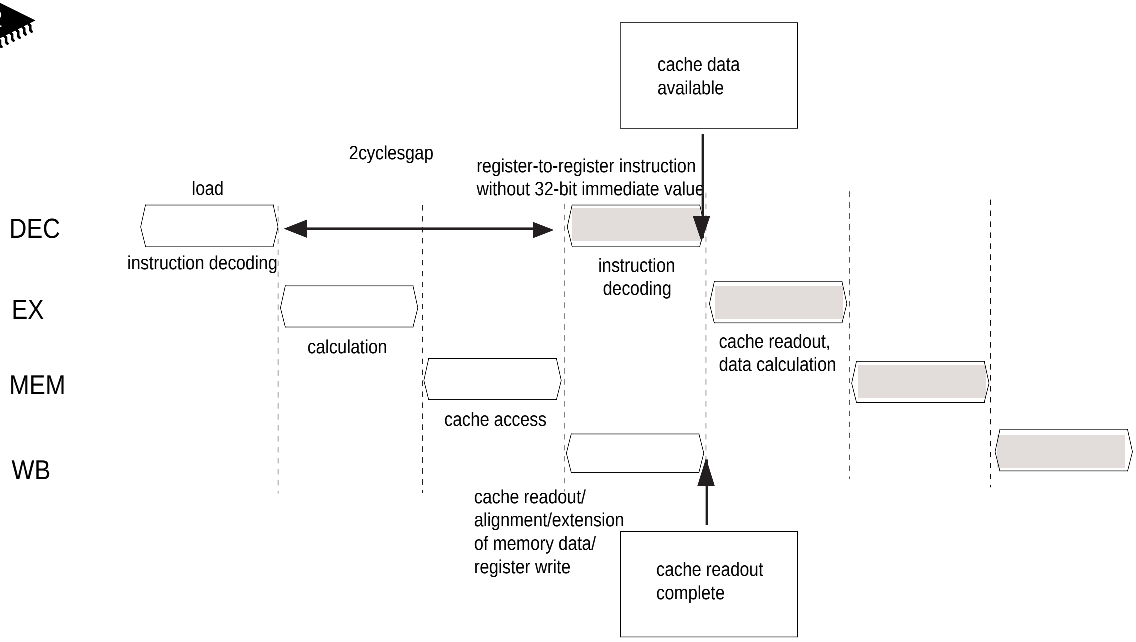
AM31

AM32

(1) Cachable Memory, DCBYPS = 0, Follower without imm32/d32/abs32

[Description]

To prevent pipeline stalls, we recommend inserting at least two cycles when the DCBYPS bit is “0,” the leading instruction accesses cachable external memory, and the following instruction uses the loaded data other than as imm32, d32, or abs32 in its calculations.



[Example]

Problematic Version

```
inc     a2
mov     (a0),d0     : Load instruction
inc     a3
add     d1,d0       : Instruction using loaded data other than as imm32, d32 or abs32
                    or
inc     a2
inc     a3
mov     (a0),d0     : Load instruction
add     d1,d0       I: nstruction using loaded data other than as imm32, d32 or abs32
```

Retrieving and aligning the data from the cache takes two cycles, so the pipeline stalls until the data loaded with the MOV instruction is available to the ADD instruction.

Revised Version

```
mov     (a0),d0     : Load instruction
inc     a2
inc     a3
add     d1,d0       : Instruction using loaded data other than as imm32, d32 or abs32
```

The data loaded with the MOV instruction is available to the ADD instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> All load variants of MOV, MOVBU, MOVB,* MOVHU, and MOVH*
(* The MOVB Mem, Reg and MOVH Mem, Reg variants require only a one-cycle gap.)

<Following instruction> Any instruction using the loaded data other than as imm32, d32, or abs32 in its calculations except the following; MOV PSW,Dn, MOV Dm,PSW,
AND imm16,PSW, OR imm16,PSW, Bcc, Lcc, JMP, TRAP, NOP.

Here cachable external memory refers to the cachable portions of AM31 or AM32 external memory.

High-speed

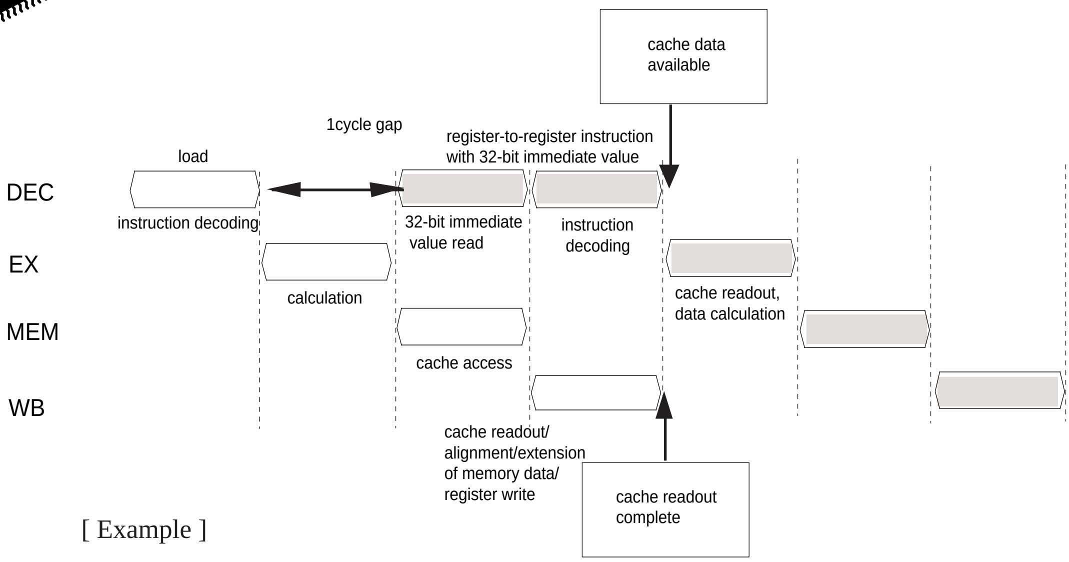
AM31

AM32

(2) Cachable Memory, DCBYPS = 0, Follower with d32

[Description]

To prevent pipeline stalls, we recommend inserting at least one cycle when the DCBYPS bit is “0,” the leading instruction accesses cachable external memory, and the following instruction uses the data together with a 32-bit displacement [imm32?] operand.



[Example]

[Problematic Version]		
inc	a1	
mov	(a0),d0	Load instruction
add	0x12345678,d0	Instruction using loaded data and 32-bit displacement d32 operand

Retrieving and aligning the data from the cache takes one cycle, so the pipeline stalls until the data loaded with the MOV instruction is available to the ADD instruction.

[Revised Version]		
mov	(a0),d0	Load instruction
inc	a1	
add	0x12345678,d0	Instruction using loaded data and 32-bit displacement d32 operand

The data loaded with the MOV instruction is available to the ADD instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> All load variants of MOV, MOVBU, and MOVHU

<Following instruction> MOV, MOVBU, and MOVHU variants with 32-bit operands (d32).

Here cachable external memory refers to the cachable portions of AM31 or AM32 external memory.

High-speed

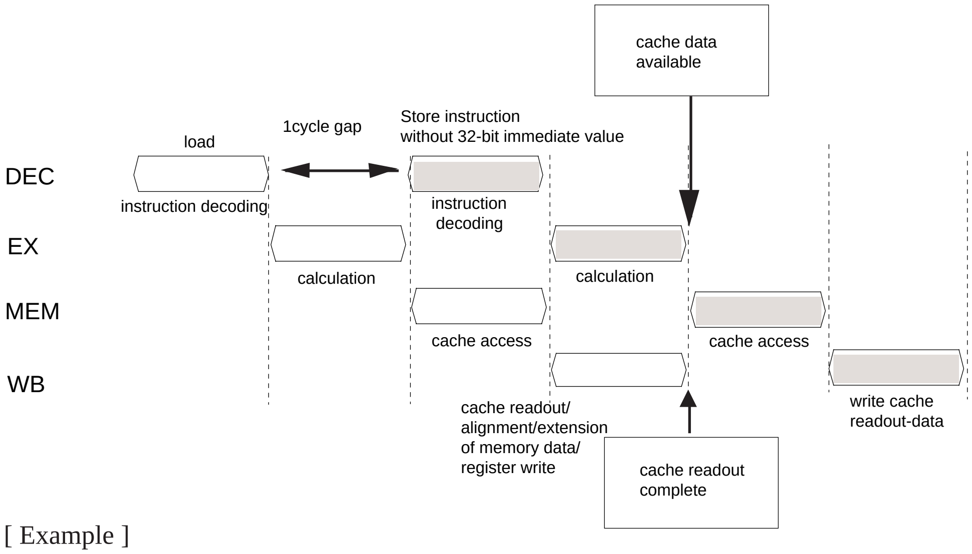
AM31

AM32

(3) Cachable Memory, DCBYPS = 0, Follower Storing without d32/abs32

[Description]

To prevent pipeline stalls, we recommend inserting at least one cycle when the DCBYPS bit is “0,” the leading instruction accesses cachable external memory, and the following instruction stores the data without using a d32 or abs32 operand.



[Example]

[Problematic Version]

```
inc    a1
mov    (a0),d0    Load instruction
mov    d0,(a1)    Instruction storing loaded data without d32/abs32 operand
```

Retrieving and aligning the data from the cache takes one cycle, so the pipeline stalls until the data loaded with the first MOV instruction is available to the second MOV instruction.

[Revised Version]

```
mov    (a0),d0    Load instruction
inc    a1
mov    d0,(a1)    Instruction storing loaded data without d32/abs32 operand
```

The data loaded with the first MOV instruction is available to the second MOV instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> All load variants of MOV, MOVBU, and MOVHU

<Following instruction> MOV, MOVBU, MOVB, MOVHU, and MOVH store without d32/abs32 operands

Here cachable external memory refers to the cachable portions of AM31 or AM32 external memory.

High-speed

AM30

AM31

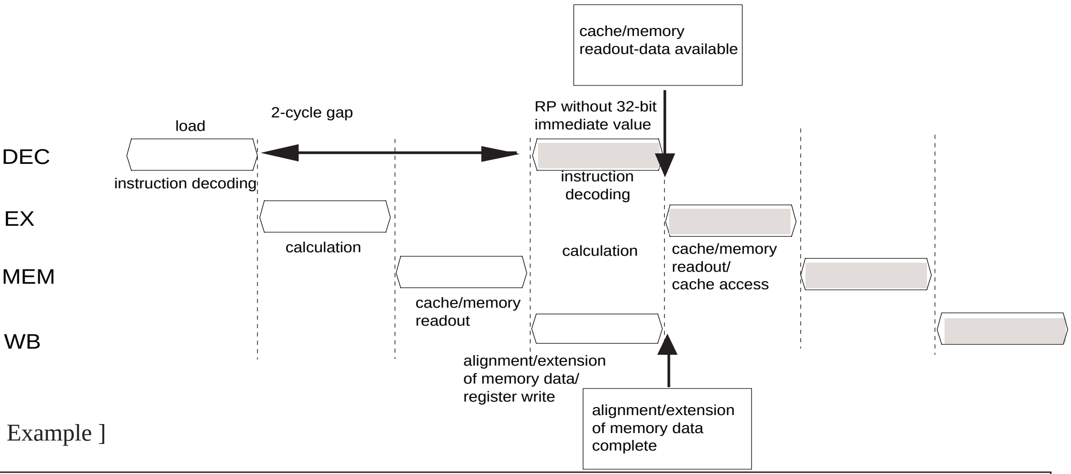
AM32

(4) Other Memory, LDUSE = 1, Word Data, Follower without imm32/d32/abs32

Here “other” means all memory types except cachable external memory with DCBYPS = 0.

[Description]

To prevent pipeline stalls, we recommend inserting at least two cycles when the LDUSE bit is “1,” the leading instruction loads a word, and the following instruction uses the loaded data other than as imm32, d32, or abs32 in its calculations.



[Example]

[Problematic Version]

```
inc     a2
mov     (a0),d0   Load instruction
inc     a3
add     d1,d0     Instruction using loaded data other than as imm32, d32 or
                  abs32
                  or
inc     a2
inc     a3
mov     (a0),d0   Load instruction
add     d1,d0     Instruction using loaded data other than as imm32, d32 or
                  abs32
```

Retrieving and aligning the data from the cache or memory takes two cycles, so the pipeline stalls until the data loaded with the MOV instruction is available to the ADD instruction.

[Revised Version]

```
mov     (a0),d0   Load instruction
inc     a2
inc     a3
add     d1,d0     Instruction using loaded data other than as imm32, d32 or
                  abs32
```

The data loaded with the MOV instruction is available to the ADD instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction>	All load variants of MOV
<Following instruction>	Any instruction using the loaded data other than as imm32, d32, or abs32 in its calculations except the following; MOV PSW,Dn , MOV Dm,PSW, AND imm16,PSW, Bcc, Lcc, JMP, TRAP, NOP.

High-speed

AM30

AM31

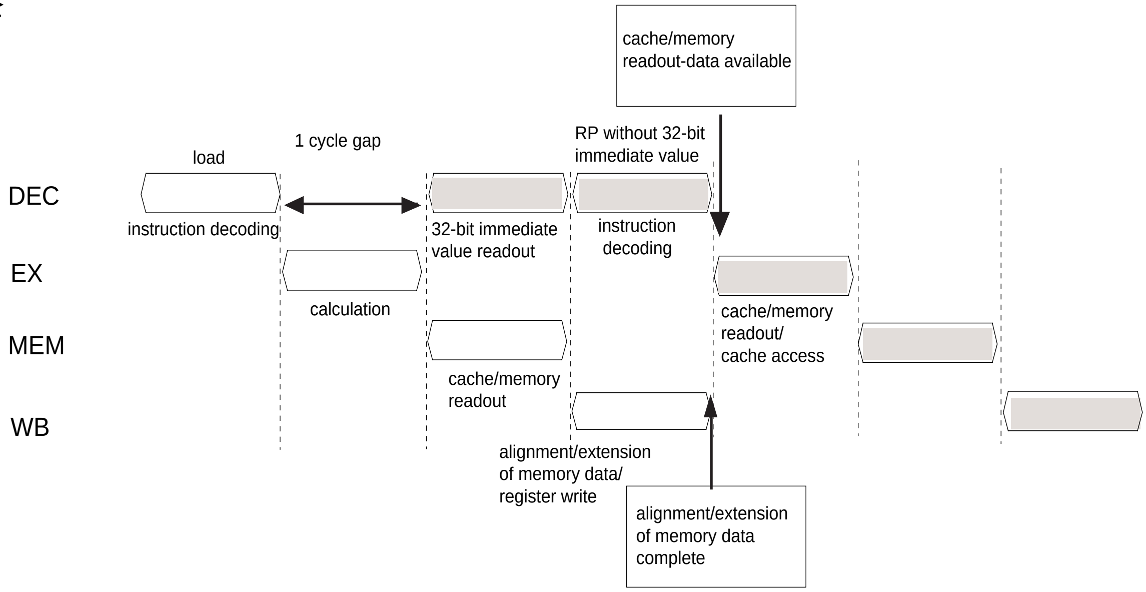
AM32

(5) Other Memory, LDUSE = 1, Word Data, Follower with d32

Here “other” means all memory types except cachable external memory with DCBYPS = 0.

[Description]

To prevent pipeline stalls, we recommend inserting at least one cycle when the LDUSE bit is “1,” the leading instruction loads a word, and the following instruction uses the data together with a 32-bit displacement [imm32?] operand.



[Example]

[Problematic Version]

```
inc     a1
mov     (a0),d0           Load instruction
add     0x12345678,d0     Instruction using loaded data and 32-bit displacement
                          operand
```

Retrieving and aligning the data from the cache or memory takes one cycle, so the pipeline stalls until the data loaded with the MOV instruction is available to the ADD instruction.

[Revised Version]

```
mov     a(0),d0           Load instruction
inc     a1
add     0x12345678,d0     Instructions using loaded data and 32-bit
                          displacement operand
```

The data loaded with the MOV instruction is available to the ADD instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction>	All load variants of MOV
<Following instruction >	MOV, MOVBU, and MOVHU variants with 32-bit displacement operands, ADD imm32,Dn, OR imm32,Dn, XOR imm32,Dn, BTST imm32,Dn, UDFnn imm32,Dn, UDFUnn imm32,Dn.

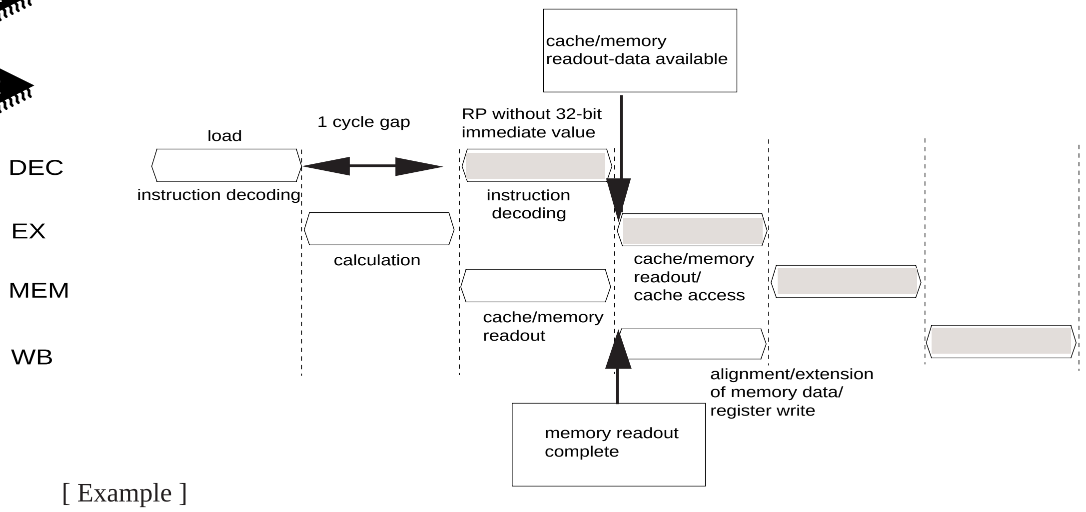
High-speed

(6) Other Memory, LDUSE = 0, Word Data, Follower without imm32/d32/abs32

Here “other” means all memory types except cachable external memory with DCBYPS = 0.

[Description]

To prevent pipeline stalls, we recommend inserting at least one cycle when the LDUSE bit is “0,” the leading instruction loads a word, and the following instruction uses the loaded data other than as imm32, d32, or abs32 in its calculations.



[Example]

[Problematic Version]

```
inc     a1
mov     (a0),d0     Load instruction
add     d1,d0       Instruction using loaded data other than as imm32, d32
                    or abs32.
```

Retrieving and aligning the data from the cache or memory takes two cycles, so the pipeline stalls until the data loaded with the MOV instruction is available to the ADD instruction.

[Revised Version]

```
mov     (a0),d0     Load instruction
inc     a1
add     d1,d0       Instruction using loaded data other than as imm32, d32
                    or abs32.
```

The data loaded with the MOV instruction is available to the ADD instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> All load variants of MOV

<Following instruction> Any instruction using the loaded data other than as imm32, d32, or abs32 in its calculations except the following; MOV PSW,Dn, MOV Dm,PSW, AND imm16,PSW, OR imm16,PSW, Bcc, Lcc, JMP, TRAP NOP.

High-speed

AM30

AM31

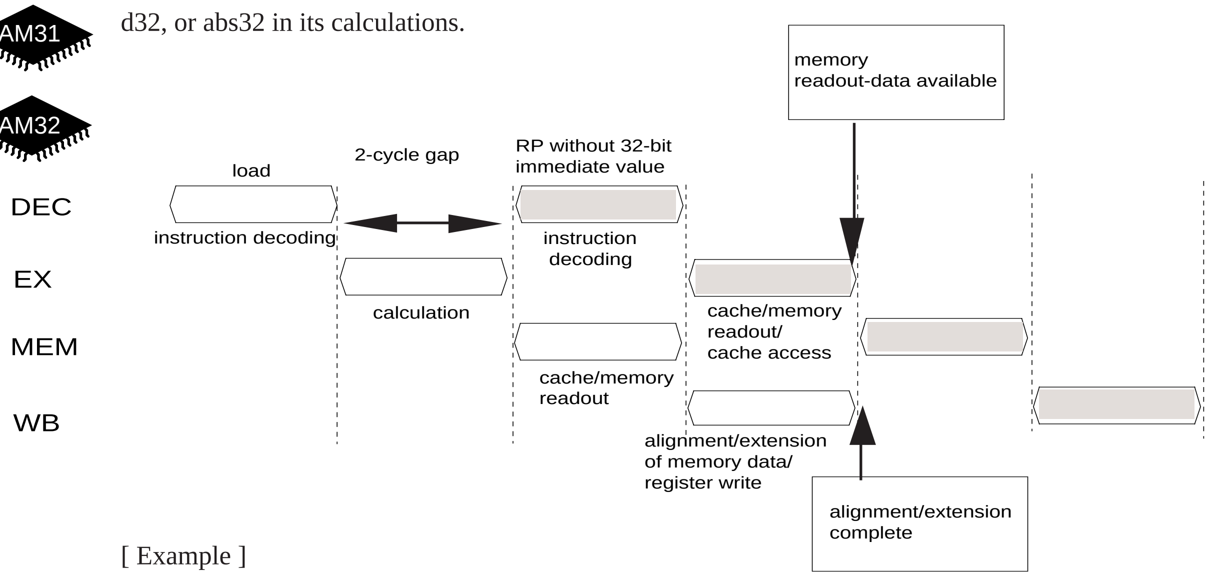
AM32

(7) Other Memory, Non-Word Data, Follower without imm32/d32/abs32

Here “other” means all memory types except cachable external memory with DCBYPS = 0.

[Description]

To prevent pipeline stalls, we recommend inserting at least two cycles when the leading instruction loads a byte or half-word, and the following instruction uses the loaded data oth
d32, or abs32 in its calculations.



[Example]

[Problematic Version]

```
inc     a1
movbu   (a0),d0    Load instruction
inc     a2
add     d1,d0      Instruction using loaded data other than as imm32, d32
or
                   abs32
                   or
inc     a1
inc     a2
movbu   (a0),d0    Load instruction
add     d1,d0      Instruction using loaded data other than as imm32, d32
                   or abs32
```

Retrieving and aligning the data from the cache or memory takes two cycles, so the pipeline stalls until the data loaded with the MOV instruction is available to the ADD instruction.

[Revised Version]

```
movbu (a0),d0   Load instruction
inc   a1
inc   a2
add   d1,d0  Instruction using loaded data other than as imm32, d32 or abs32.
```

The data loaded with the MOV instruction is available to the ADD instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> All load variants of MOV, MOVBU, MOVB,* MOVHU, and MOVH* (* The MOVB Mem, Reg and MOVH Mem, Reg variants require only a one-cycle gap.)

<Following instruction> Any instruction using the loaded data other than as imm32, d32, or abs32 in its calculations except the following; MOV PSW,Dn, MOV Dm,PSW, AND imm16,PSW, OR imm16,PSW, Bcc, Lcc, JMP, TRAP, NOP.

High-speed

AM30

AM31

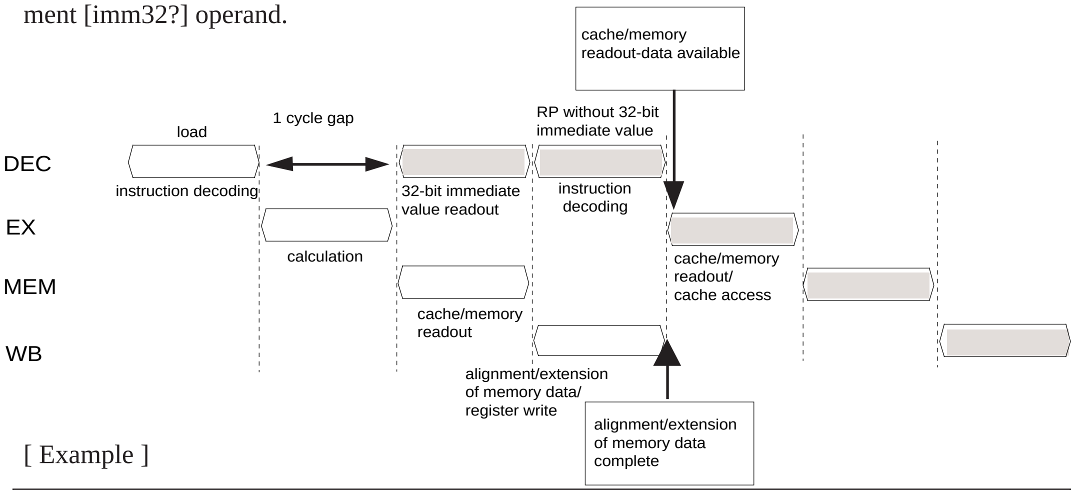
AM32

(8) Other Memory, Non-Word Data, Follower with d32

Here “other” means all memory types except cachable external memory with DCBYPS = 0.

[Description]

To prevent pipeline stalls, we recommend inserting at least one cycle when the leading instruction loads a byte or half-word, and the following instruction uses the data tc -bit displacement [imm32?] operand.



[Example]

[Problematic Version]

inc	a1	
movbu	(a0),d0	Load instruction
add	0x12345678,d0	Instruction using loaded data and 32-bit displacement operand

Retrieving and aligning the data from the cache or memory takes one cycle, so the pipeline stalls until the data loaded with the MOVBU instruction is available to the ADD instruction.

[Revised Version]

movbu	(a0),d0	Load instruction
inc	a1	
add	0x12345678,d0	Instruction using loaded data and 32-bit displacement operand

The data loaded with the MOVBU instruction is available to the ADD instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> All load variants of MOVBU and MOVHU

<Following instruction> MOVBU, MOVBU, and MOVHU variants with 32-bit displacement operands; ADD imm32,Dn/An/SP, SUB imm32,Dn/An, CMP imm32,Dn/An, AND imm32,Dn, OR imm32,Dn, XOR imm32,Dn, BTST imm32,Dn, UDFnn imm32,Dn, UDFUnn imm32,Dn

High-speed

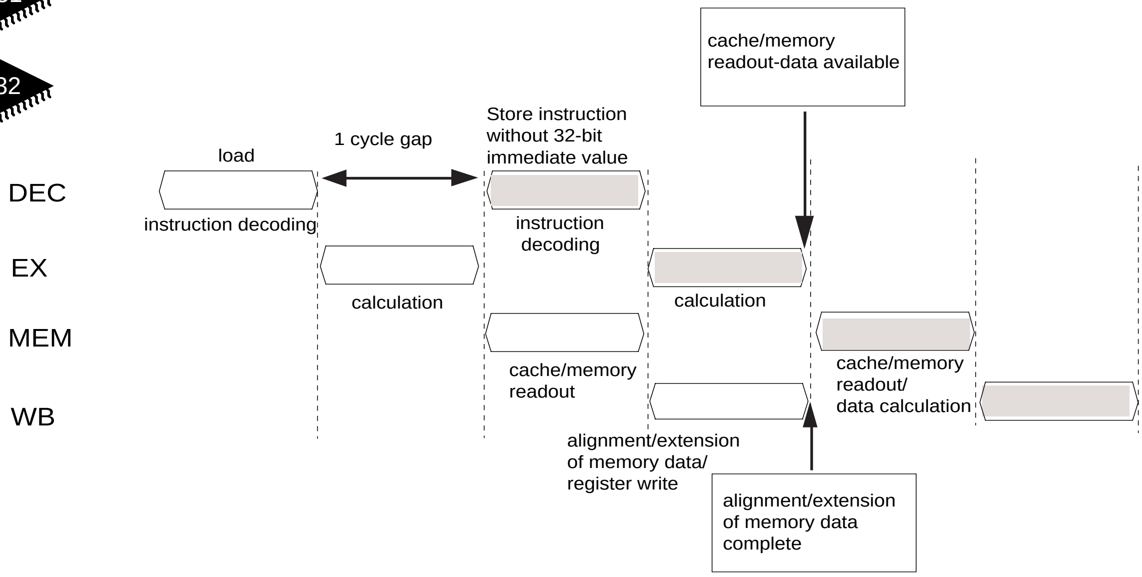
AM30 AM31 AM32

(9) Other Memory, Follower Storing without d32/abs32

Here “other” means all memory types except cachable external memory with DCBYPS = 0.

[Description]

To prevent pipeline stalls, we recommend inserting at least one cycle when the following instruction stores the data without using a d32 or abs32 operand.



[Example]

[Problematic Version]

```
inc     a1
movbu   (a0),d0    Load instruction
mov     d0,(a1)    Instruction storing loaded data without d32/abs32 operand
```

Retrieving and aligning the data from the cache or memory takes one cycle, so the pipeline stalls until the data loaded with the MOVBU instruction is available to the MOV instruction.

[Revised Version]

```
movbu   (a0),d0    Load instruction
inc     a1
mov     d0,(a1)    Instruction storing loaded data without d32/abs32 operand
```

The data loaded with the MOVBU instruction is available to the MOV instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> All load variants of MOVBU and MOVHU

<Following instruction> MOV, MOVBU, MOVB, MOVHU, and MOVH store without d32/abs32 operands.

3.5 Instructions Following DIV/DIVU with Zero Dividend

The DIV and DIVU instructions minimize their execution times by minimizing the size of the dividend in bytes, but they still require a minimum of two cycles to store, during their write-back stage, the result in the destination data register (Dn)—even when the dividend, originally in Dn, is 0 and the result, stored in the same register, is the same.

High-speed

AM30

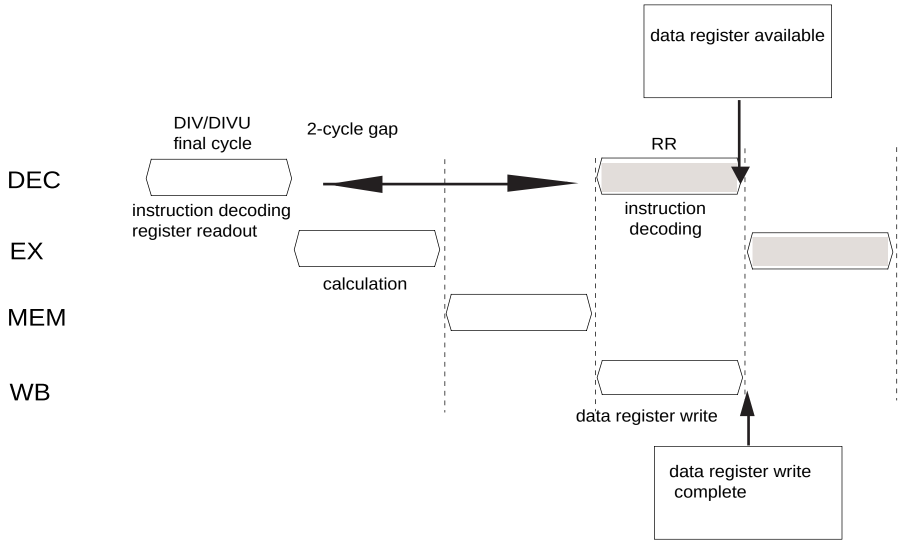
AM31

AM32

(1) Follower without imm32/d32/abs32

[Description]

To prevent pipeline stalls, we recommend inserting at least two cycles when the leading divide instruction has a zero dividend and the following instruction uses the result (Dn) other than as imm32, d32, or abs32 in its calculations.



[Example]

[Problematic Version]

```
clr     d0
mov     d0,mdr
mov     0x05,d1
inc     a0
div     d1,d0     DIV/DIVU
inc     a1
add     d1,d0     Instruction using result(dn) other than as imm32, d32 or
                  abs32
                  or
clr     d0
mov     d0,mdr
mov     0x05,d1
inc     a0
inc     a0
inc     a1
div     d1,d0     DIV/DIVU
add     d1,d0     Instruction using result(Dn) other than as imm32, d32 or
                  abs32
```

Storing the result takes two cycles, so the pipeline stalls until the result of the divide instruction is available to the ADD instruction.

[Revised Version]

```
clr     d0
mov     d0,mdr
mov     0x05,d1
div     d1,do      DIV/DIVU
inc     a0
inc     a1
add     d1,d0      Instruction using result (Dn) other than as imm32, d32
                   or abs32
```

The result of the divide instruction is available to the ADD instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> DIV/DIVU instruction with zero dividend

<Following instruction> Any instruction using the result (Dn) other than as imm32, d32, or abs32 in its calculations except the following; MOV PSW, Dn, MOV Dm,PSW, AND imm16,PSW, OR imm16,PSW, Bcc, Lcc, JMP, TRAP, NOP.

(2) Follower with d32

High-speed

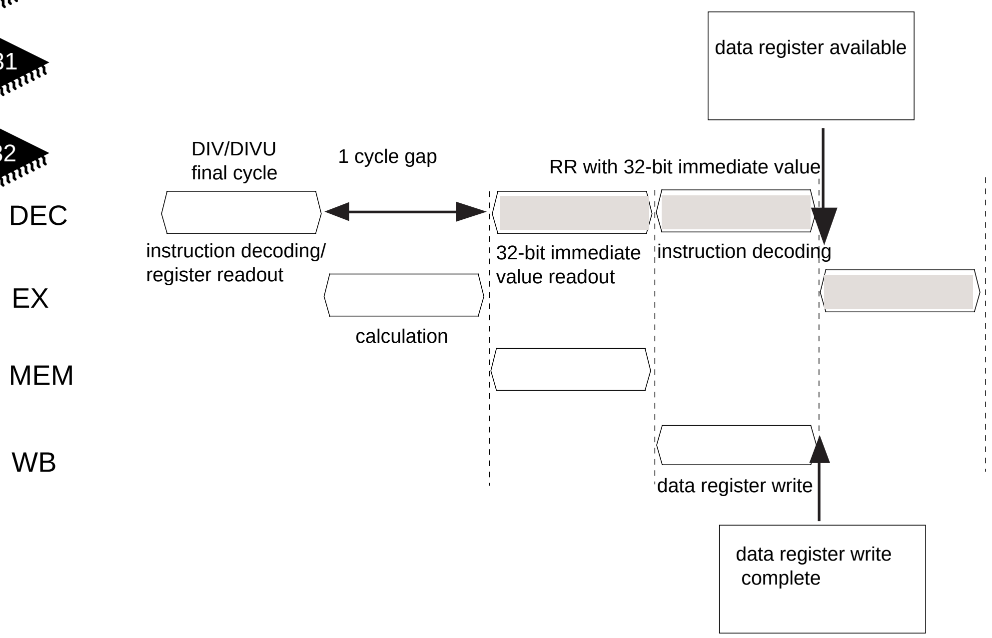
AM30

AM31

AM32

[Description]

To prevent pipeline stalls, we recommend inserting at least one cycle when the leading divide instruction has a zero dividend and the following instruction uses the result (Dn) together with a 32-bit displacement [imm32?] operand.



[Example]

[Problematic Version]

```
clr     d0
mov     d0,mdr
mov     0x05,d1
inc     a0
div     d1,d0              DIV/DIVU
add     0x12345678,d0      Instruction using result(Dn) and 32-bit
                           displacement operand
```

Storing the result takes one cycle, so the pipeline stalls until the result of the divide instruction is available to the ADD instruction.

[Revised Version]

```
clr     d0
mov     d0,mdr
mov     0x05,d1
div     d1,d0              DIV/DIVU
inc     a0
add     0x12345678,d0      Instruction using result(dn) and 32-bit displacement
                           operand
```

The result of the divide instruction is available to the ADD instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction>	DIV/DIVU instruction with zero dividend
<Following instruction>	MOV, MOVBU, and MOVHU variants with 32-bit displacement operands; ADD imm32,Dn/An/SP, SUB imm32,Dn/An, CMP imm32,Dn/An, AND imm32,Dn, OR imm32,Dn, XOR imm32,Dn, BTST imm32,Dn, UDFnn imm32,Dn, UDFUnn imm32,Dn

(3) Follower Storing without d32/abs32

High-speed

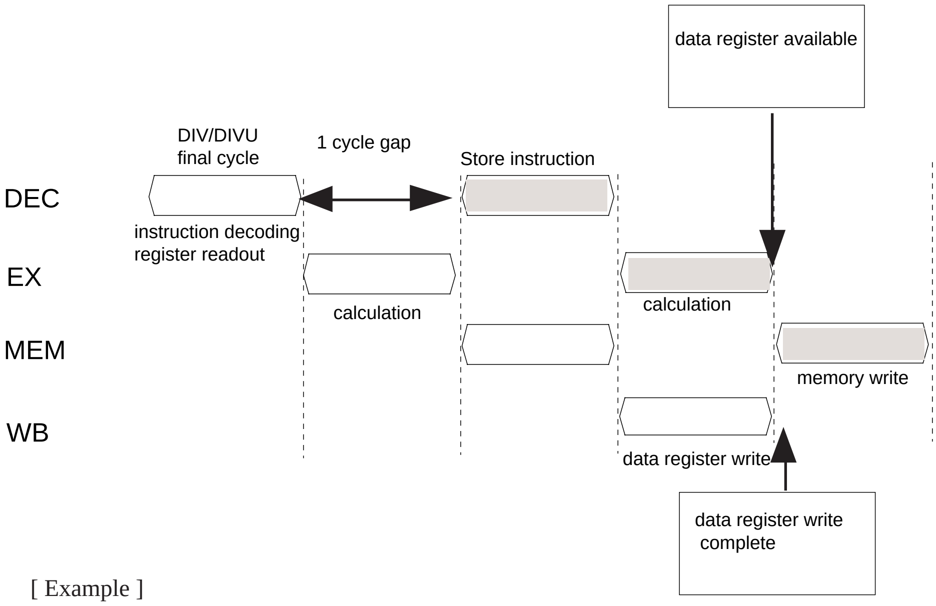
AM30

AM31

AM32

[Description]

To prevent pipeline stalls, we recommend inserting at least one cycle when the leading divide instruction has a zero dividend and the following instruction stores the result (Dn) without using an imm32, d32, or abs32 operand.



[Example]

[Problematic Version]

```
clr     d0
mov     d0,mdr
mov     0x05,d1
inc     a0
div     d1,d0       DIV/DIVU
mov     d0,(a1)     Instruction storing result(Dn) without d32 or abs32
                    operand
```

Storing the result takes one cycle, so the pipeline stalls until the result (Dn) is available to the MOV instruction.

[Revised Version]

```
clr     d0
mov     d0,mdr
mov     0x05,d1
div     d1,d0       DIV/DIVU
inc     a0
mov     d0,(a1)     Instruction storing result(Dn) without imm32, d32 or
                    abs32 operand
```

[Applicable Instructions]

<Leading instruction> DIV/DIVU instruction with zero dividend

<Following instruction> Any instruction using the result (Dn) other than as imm32, d32, or abs32 in its calculations except the following; MOV PSW,Dn, MOV Dm,PSW, AND imm16,PSW, OR imm16,PSW, Bcc, Lcc, JMP, TRAP, NOP.

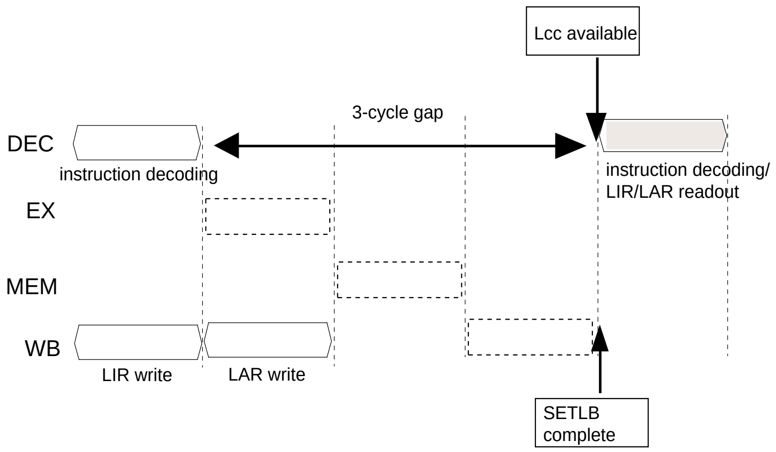
3.6 Instructions Preceding Lcc

The Lcc instruction bypasses the five-stage pipeline with additional hardware to increase the execution speed. To ensure proper interaction between the two, it must therefore wait for certain preceding instructions to complete.

(1) SETLB

[Description]

To ensure smooth pipeline operation, we recommend inserting at least three cycles between the Lcc instruction and the preceding SETLB instruction setting up the two registers that it must reference: Loop Instruction Register (LIR) and Loop Address (LAR) Register.



[Example]

[Problematic Version]

```
        inc      d0
        setlb             SETLB
        inc      d1
        inc      d2
        lgt               Lcc             or
        inc      d0
        inc      d1
        setlb             SETLB
        inc      d2
        lgt               Lcc             or
        inc      d0
        inc      d1
        inc      d2
        setlb             SETLB
        lgt               Lcc
```

Loading the LIR and LAR registers takes three cycles, so the pipeline stalls until their contents are available to the LGT instruction.

[Revised Version]

```
setlb              SETLB
ubc      d0
inc      d1
inc      d2
lgt                Lcc
```

The updated contents of the LIR and LAR registers are available to the LGT instruction, so the pipeline does not stall.

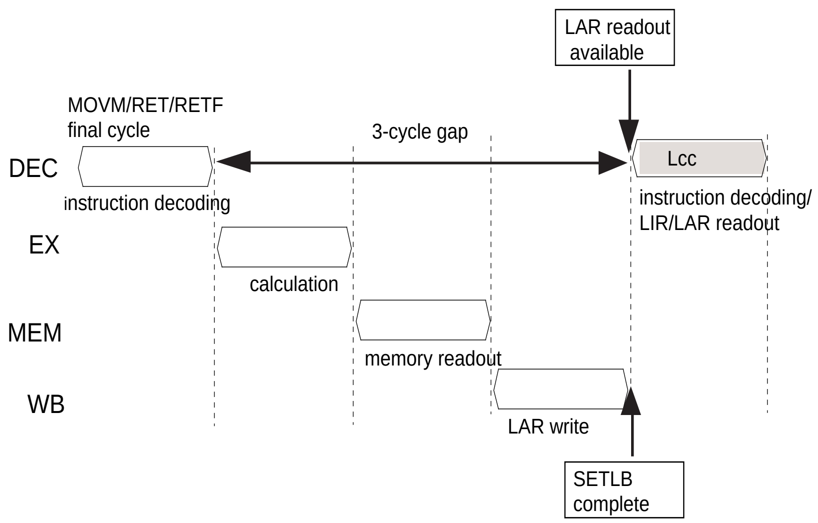
[Applicable Instructions]

<Leading instruction>	SETLB
<Following instruction>	Lcc

(2) Other Instructions Modifying LIR and LAR

[Description]

To ensure smooth pipeline operation, we recommend inserting at least three cycles between the Lcc instruction and other preceding instructions (MOVM, RET, and RETF) modifying the two registers that it must reference: Loop Instruction Register (LIR) and Loop Address (LAR) Register.



[Example]

[Problematic Version]

```
inc     d0
movm    (sp),[other]    Instruction modifying LIR and  LAR
inc     d1
inc     d2
lgt                     Lcc                             or

inc     d0
inc     d1
movm    (sp),[other]    Instruction modifying LIR and LAR
inc     d2
lgt     Lcc                                             or

inc     d0
inc     d1
inc     d2
movm    (sp),pother]    Instruction modifying LIE and LAE
lgt                     Lcc
```

Loading the LIR and LAR registers takes three cycles, so the pipeline stalls until their contents are available to the LGT instruction.

[Revised Version]

```
movm  (sp),[other]   Instruction modifying LIR and LAR
inc   d0
inc   d1
inc   d2
lgt          Lcc
```

The updated contents of the LIR and LAR registers are available to the LGT instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction>   MOVM (SP),regs, RET, RETF
<Following instruction>   Lcc

High-speed

AM30

AM31

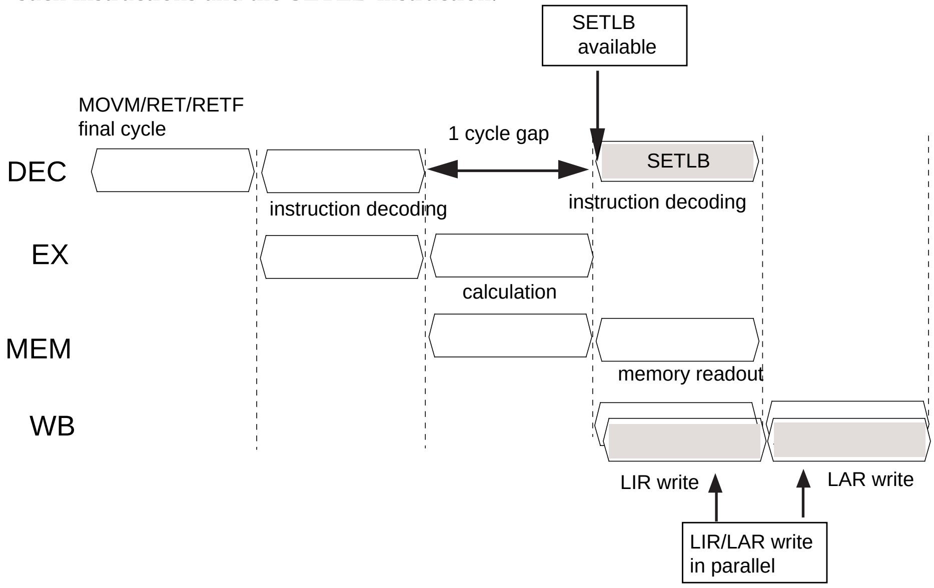
AM32

3.7 Instructions Preceding SETLB

The SETLB instruction bypasses the five-stage pipeline with additional hardware to increase the execution speed. To ensure proper interaction between the two, it must wait for preceding instructions (MOVM, RET, and RETF) modifying the two registers that it loads: Loop Instruction Register (LIR) and Loop Address (LAR) Register.

[Description]

To ensure smooth pipeline operation, we therefore recommend inserting at least one cycle between such instructions and the SETLB instruction.



[Example]

```
[ Problematic Version ]
        inc       d0
        movm      (sp),[other]     Instruction modifying LIR and LAR
        setlb                      SETLB
```

Loading the LIR and LAR registers takes one cycle, so the pipeline stalls until their contents are available to the SETLB instruction.

```
[ Revised Version ]
        movm      (sp),[other]     Instruction modifying LIR and LAR
        inc       d0
        setlb                      SETLB
```

The updated contents of the LIR and LAR registers are available to the SETLB instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> MOVM (SP),regs; RET; RETF

<Following instruction> SETLB

3.8 Instructions Preceding RETF

The RETF instruction takes its return address from the Multiply/Divide Register (MDR), not the stack, so it must wait for preceding instructions modifying that register to complete. How long depends on the position of the MDR update relative to the end of the preceding instruction.

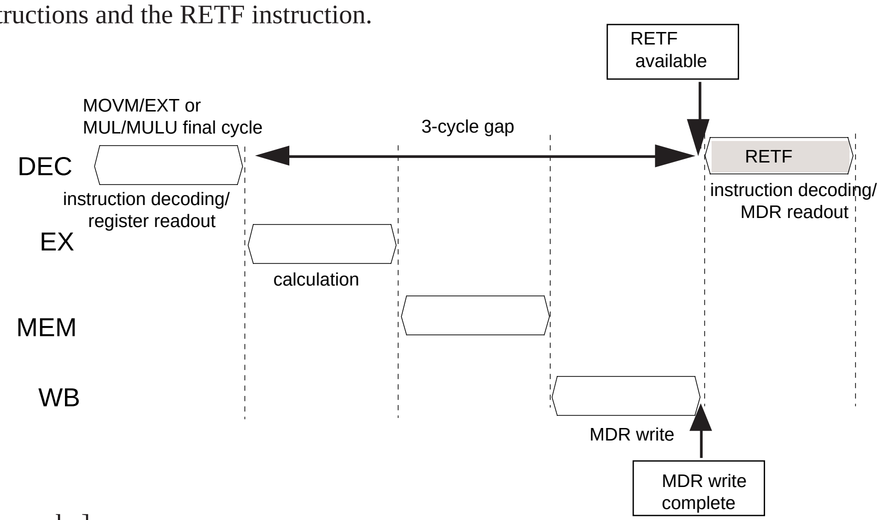
High-speed

(1) MDR Update in Last Cycle

[Description]

To ensure smooth pipeline operation, we recommend inserting at least three cycles between such instructions and the RETF instruction.

AM30
AM31
AM32



[Example]

```
[ Problematic Version ]
            _func
            _0func      FUNCINFO    _func,8,[ ]
                        inc         a0
                        mov         d0,mdr      Instruction modifying MDR in last cycle
                        inc         a1
                        inc         a2
                        retf                    RETF                          or

            _func
            _0func                  FUNCINFO    _func,8,[ ]
                        inc         a0
                        inc         a1
                        mov         d0,mdr      Instruction modifying MDR in last cycle
                        inc         a2
                        retf                    RETF                          or
            _func
            _0func      FUNCINFO    _func,8,[ ]
                        inc         a0
                        inc         a1
                        inc         a2
                        mov         d0,mdr      Instruction modifying MDR in last cycle
                        retf                    RETF
```

Loading the MDR register with the MOV instruction takes three cycles, so the pipeline stalls until that register's contents are available to the RETF instruction.

[Revised Version]

```
_func
_funco      FUNCINFO   _func,8,[ ]
            mov        d0,mdr       Instruction modifying MDR in last cycle
            inc        a0
            inc        a1
            inc        a2
            retf                    retf
```

The updated contents of the MDR register are available to the RETF instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction>	MOVM Dm, MDR, EXT, MUL, MULU
<Following instruction>	RETF

For a detailed description of FUNCINFO syntax, refer to the Cross Assembler User's Manual.

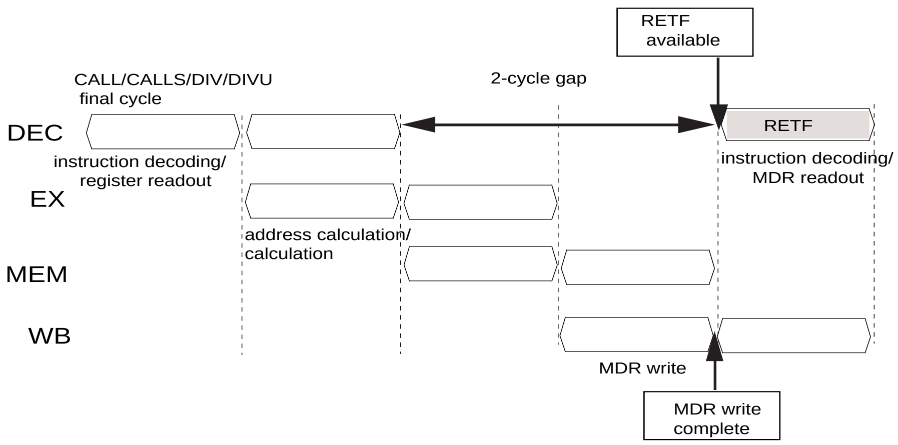
High-speed

AM30 AM31 AM32

(2) MDR Update in Second Last Cycle

[Description]

To ensure smooth pipeline operation, we recommend inserting at least two cycles between such instructions and the RETF instruction.



[Example]

[Problematic Version]

```
            _lab
            LABEL     FUNCINFO   _lab,8,[ ]
                      inc        a0
                      div        d1,d0      Instruction modifyint MDR in second last cycle
                      inc        a1
                      retf                  RETF               or
            _lab
            LABEL     FUNCINFO   _lab,8,[ ]
                      inc        a0
            inc       a1
                      div        d1,d0      Instruction modifying MDR in second last cycle
                      retf                  RETF
```

Updating the MDR register with the DIV instruction takes two cycles, so the pipeline stalls until that register's contents are available to the RETF instruction.

Revised Version

```
            _lab
            LABEL     FUNCINFO   _lab,8,[ ]
                      div        d1,d0      Instruction modifying MDR in second last cycle
                      inc        a0
                      inc        a1
                      retf                  RETF
```

The updated contents of the MDR register are available to the RETF instruction, so the pipeline does not stall.

Applicable Instructions

<Leading instruction>   DIV, DIVU, CALL, CALLS
<Following instruction>   RETF

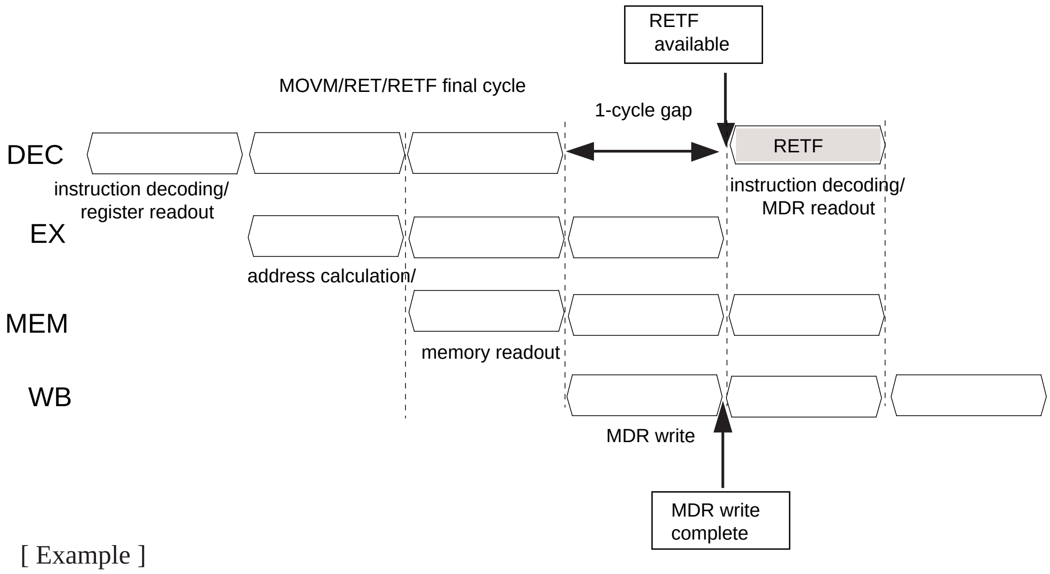
For a detailed description of FUNCINFO syntax, refer to the Cross Assembler User's Manual.

High-speed AM30 AM31 AM32

(3) MDR Update in Third Last Cycle

[Description]

To ensure smooth pipeline operation, we recommend inserting at least one cycle between such instructions and the RETF instruction.



[Example]

[Problematic Version]

```
_lab
LABEL      FUNCINFO   _lab,8,[ ]
           inc        d0
           movm       (sp), [other]   Instruction modifying MDR in third last
                                       cycle
           retf       RETF
```

RETF instruction Reloading the MDR register with the MOVM instruction takes one cycle, so the pipeline stalls until that register's contents are available to the RETF instruction.

[Revised Version]

```
_lab
LABEL      FUNCINFO   _lab,8,[ ]
           movm       (sp),[other]   Instruction modifying MDR in third last
                                      cycle
           inc        d0
           retf                      RETF
```

The updated contents of the MDR register are available to the RETF instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> MOVM (SP),regs, RET, RETF

<Following instruction> RETF

For a detailed description of FUNCINFO syntax, refer to the Cross Assembler User's Manual.

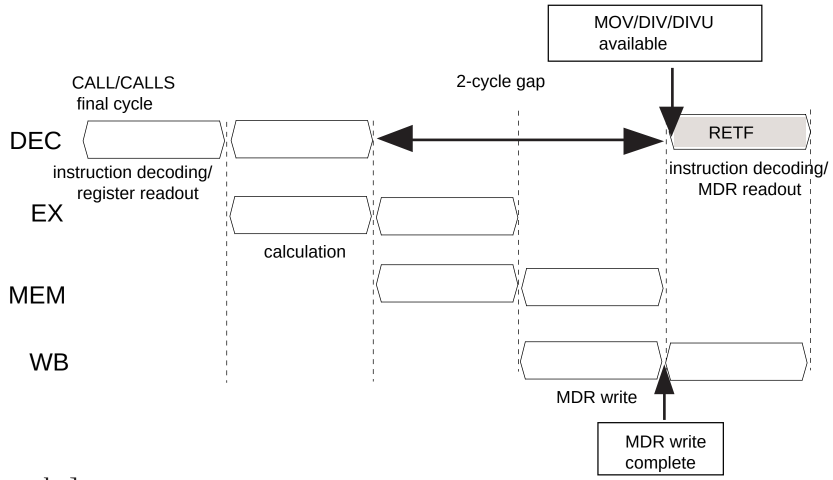
3.9 Instructions at CALL/CALLS Targets

High-speed

AM30 AM31 AM32

[Description]

To speed returns with the RETF instruction, the CALL and CALLS instructions store the return address in the Multiply/Divide Register (MDR), not on the stack. We therefore recommend inserting at least two cycles at the start of the subroutine to allow them time to complete the MDR update before using instructions (MOV, DIV, and DIVU) that reference that register.



[Example]

[Problematic Version]

```
                call    LABEL     CALL/CALLS
                :
                :
        LABEL   inc     a0
                mov     mdr,d0    Instruction referring MDR    or
                call    LABEL     CALL/CALLS
                :
                :
        LABEL   mov     mdr,d0    Instruction referring MDR
                inc     a0
                inc     a1
```

Loading the MDR register with the CALL/CALLS instruction takes two cycles, so the pipeline stalls until that register's contents are available to the MOV instruction.

[Revised Version]

```
                call    LABEL     CALL/CALLS
                :
                :
        LABEL   inc     a0
                inc     a1
                mov     mdr,d0    Instruction referring MDR
```

The updated contents of the MDR register are available to the MOV instruction, so the pipeline does not stall.

[Applicable Instructions]

<Leading instruction> CALL or CALLS
<Following instruction> MOV MDR, Dn, DIV, DIVU

The above coding examples omit the FUNCINFO directives required with each CALL instruction. For a detailed description of FUNCINFO syntax, refer to the Cross Assembler User’s Manual.

---

4 Boiler Plate Code Sequences

4.1 Reset Routine

Boiler plate

[Description]

A reset start leaves all registers with indeterminate contents, so always start by initiating registers. The Stack Pointer (SP) is particularly important.
If the user application program uses level interrupts, initialize the interrupt vector registers as well.

[Example]

```
IVAR0        equ      0x20000000
IVAR1        equ      0x20000004
IVAR2        equ      0x20000008
IVAR3        equ      0x2000000c
IVAR4        equ      0x20000010
IVAR5        equ      0x20000014
IVAR6        equ      0x20000018
;
Reset routine
;
             org      0x40000000          ; Entry point for reset
RESET:
             jmp      STARTUP
             org      0x40000008          ; Entry point for nonmaskable interrupt
NMIROUTINE:
             ....
STARTUP:
             mov      STCKTP,a0           ; Initialize Stack Pointer(SP) to word boundary
                                          ; (divisible by 4)
             mov      a0,sp
             clr      d0                  ; Clear all registers
             mov      d0,d1
             mov      d0,d2
             mov      d0,d3
             mov      d0,a0
             mov      d0,a0
             mov      a0,a1
             mov      a0,a2
             mov      a0,a3
             mov      IRQ0ROUTINE,a0      ; Set to lower 16 bits of entry point for level interrupt
handler
             movhu    a0,(IVATo)
             mov      IRQ1ROUTINE,a0
             movhu    a0,(IVAR1)
             mov      IRQ2ROUTINE,a0
             movhu    a0,a0,(IVAR2)
```

```
                mov         IRQ3ROUTINE,a0
                movhu       a0,(IVAR3)
                mov         IRQ4ROUTINE,a0
                movhu       a0,(IVAR4)
                mov         IRQ5ROUTINE,a0
                movhu       a0,(IVAR5)
                mov         IRQ6ROUTINE,a0
                movhu       a0,(IVAR6)
```

4.2 Interrupt Handlers

Boiler plate

[Description]

There are two types of interrupt handlers: for nonmaskable interrupts and for level interrupts.
For level interrupts, the Interrupt Group Register (IAGR) gives the group number. Assigning only one interrupt source to an interrupt level speeds response by eliminating the need for differentiation logic.
With store buffer models, always ensure clearing of the interrupt source by separating the write to the interrupt control register and the RTI instruction--with a vacuous read of that register, for example.
For the serial debugger's special requirements for the nonmaskable interrupt handler, refer to the MN10300 Series C Source Code Debugger User's Manual.

[Example]

```
;
Interrupt handler for NMI, level 0 interrupts and level 1 interrupts
;
IAGR      equ   0x34000200
IVAR0     equ   0x20000000
IVAR1     equ   0x20000004
IVAR2     equ   0x20000008
IVAR3     equ   0x2000000c
IVAR4     equ   0x20000010
IVAR5     equ   0x20000014
IVAR6     equ   0x20000018
G01CR     equ   0x34000100
G31CR     equ   0x3400010c
;
Nonmaskable interrupt handler
;
     org   0x40000008   ; Entry point for nonmaskable interrupts
NMIROUTINE:
     add   -8,sp
     mov   a0,(4,sp)    ; Save registers used by handler
     mov   d0,(sp)
     mov   G01CR,a0
     mov   (ao),d0      ; Get NMI source
```

```
                ....                                    ; Processing according to source
                mov             0x0f,d0         ; Clear NMI source flag inG01CR
                movbu           d0,(a0)
                mov             (sp),d0         ; Restore registers
                mov             (4,sp),a0
                add             8,sp
                rti
;
; Level 0 interrupt handler
; Here G3ICR lists only one source for level 0 interrupts
;
IRQ0ROUTINE:                                    ; Level 0 interrupt entry point from IVAR0
                add             -8,sp
                mov             a0,(4,sp)       ; Save registers used by handler
                ....                            ; Process according to source
                mov             0x01,d0         ; Clear source flag in G3ICR
                mov             d0,(G3ICR)      ; Writing "1" to a bit clears corresonding source
                mov             (sp),d0         ; Restore registers
                mov             (4,sp),a0
                add             8,sp
                rti
;
; Level 1 interrupt handler
; Level 1 interrupts have multiple sources
;
IRQ1ROUTINE:                    ; Level 1 interrupt entry point from IVARI
                add             -16,sp
                mov             a0,(12,sp)      ; Save registers used by handler
                mov             a1,(8,sp)
                mov             d0,(sp)
                movhu           (IAGR),a0       ; Get interrupt source
                mov             a0,a1
                add             G01CR,a0        ; Get address for group's interrupt register
                add             IRQ_TBL,a1      ; Get entry point for group's handler
                movhu           (a0),d0         ; Get level 1 interrupt source
#IFDEFMULTIRQ
                or              0x0800,psw      ; Enable nested interrupts(IE=1)
#ENDIF
                mov             (a1),al         ;Get handlers entry point
                calls           (a1)            ; Call handler, determine corresponding source
                                                ; f;or level 1 interrupt and process all corresponding
                                                ; sources
                mov             0x0f,d0         ; Clear source flog in corresponding interrupt
                                                ;  register
                movbu           d0,(a0)         ; Writing "1" to a bit clears corresponding source
                mov             (sp),d0         ; Restore registers
                mov             (4,sp),a1
                mov             (8,sp),a0
                add             12,sp
                rti
```

4.3 Function Called with CALL Only

Boiler plate

[Description]

The CALL, RET, and RETF instruction require FUNCINFO directives to specify the registers to save.

If the subroutine does not modify the Multiply/Divide Register (MDR), the RETF instruction provides a faster return to the caller by using the return address in MDR.

For the procedures for calling C language functions from assembly language functions and assembly language functions from C language functions, refer to the C Compiler User's Manual for Operation.

[Example]

```
;
; Calling function
;
_MAIN:
          add       -12,sp          ; Secure space for subroutine parameters and return
value
          .....
          mov       d0,(8,sp)       ; Save parameters on stack
          mov       d1,(4,sp)
          call      _0FUNC          ; Call subroutine
          ....
          add       12,sp
;
; Subroutine
; Subroutine uses 28 bytes of local storage for saving A2 and other purposes
;
_0FUNC    FUNCINFO _FUNC,28,[a2]    ; Entry point for CALL instructions
          ....      ; Subroutine body
          ret       ; RETF may be used if subroutine does not modify MDR
```

4.4 Function Called with Both CALL and CALLS

Boiler plate

[Description]

A function called with both the CALL and CALLS instructions must provide two entry points because CALLS does not provide CALL's register saving and local storage allocation features. The CALLS entry point must provide them.

If the subroutine does not modify the Multiply/Divide Register (MDR), the RETF instruction provides a faster return to the caller by using the return address in MDR.

For the procedures for calling C language functions from assembly language functions and assembly language functions from C language functions, refer to the C Compiler User's Manual for Operation.

[Example]

```
;
; Calling function
;
_MAIN:
            add         -12,sp          ; Secure space for subroutine parameters and return
            ....
            mov         d0,(8,sp)       ; Save parameters on stack
            mov         d1,(4,sp)
            call        _FUNC           ; Call subroutine with CALL instruction
            ....
            add         12,sp
            ....
            ....
            add         -12,sp          ; Secure space for fubroutine parameters and return
            ....
            mov         d0,(8,sp)       ; Save parameters on stack
            mov         d1,(4,sp)
            calls       _FUNC           ; Call subroutine with CALL instruction
            ....
            add         12,sp
;
; Subroutine
; Subroutine uses 28 bytes of local storage for saving A2 and other purposes
; Subroutine may be called with CALL or CALLS instruction
;
_FUNC:
            movm        [a2],(sp)       ; CALLS instruction entry point must save registers and
                                        ;  allocate local storage
            add         -24,sp
_0FUNC      FUNCINFO _FUNC,28[a2]       ; Entry point for CALL instruction
            ....        ; Subroutine body
            ret         ; RETF may be used if subroutine does not modify MDR
```

Appendix 4

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For -mat	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
MOV	MOV Dm,Dn	Dm → Dn	-	-	-	-	1	1	S0	1000	DmDn														26
	MOV Dm,An	Dm → An	-	-	-	-	2	1	D0	1111	0001	1110	DmAn												26
	MOV Am,Dn	Am → Dn	-	-	-	-	2	1	D0	1111	0001	1101	AmDn												26
	MOV Am,An	Am → An	-	-	-	-	1	1	S0	1001	AmAn														26
	MOV SP,An	SP → An	-	-	-	-	1	1	S0	0011	11An														26
	MOV Am,SP	An → SP	-	-	-	-	2	1	D0	1111	0010	1111	Am00												26
	MOV PSW,Dn	PSW(zero_ext) → Dn	-	-	-	-	2	1	D0	1111	0010	1110	01Dn												26
	MOV Dm,PSW	Dm → PSW	●	●	●	●	2	1	D0	1111	0010	1111	Dm11												26
	MOV MDR,Dn	MDR → Dn	-	-	-	-	2	1	D0	1111	0010	1110	00Dn												26
	MOV Dm,MDR	Dm → MDR	-	-	-	-	2	1	D0	1111	0010	1111	Dm11												26
	MOV (Am),Dn	mem32(Am) → Dn	-	-	-	-	1	1	S0	0111	DnAm														27
	MOV (d8,Am),Dn	mem32(d8(sign_ext)+Am) → Dn	-	-	-	-	3	1	D1	1111	1000	0000	DnAm	<d8.	...>										27
	MOV (d16,Am),Dn	mem32(d16(sign_ext)+Am) → Dn	-	-	-	-	4	1	D2	1111	1010	0000	DnAm	<d16	....	....	....>								27
	MOV (d32,Am),Dn	mem32(d32+Am) → Dn	-	-	-	-	6	2	D4	1111	1100	0000	DnAm	<d32	....	....	....	....	....	....	....>				27
	MOV (d8,SP),Dn	mem32(d8(zero_ext)+SP) → Dn	-	-	-	-	2	1	S1	0101	10Dn	<d8	....>												27
	MOV (d16,SP),Dn	mem32(d16(zero_ext)+SP) → Dn	-	-	-	-	4	1	D2	1111	1010	1011	01Dn	<d16	....	....	....>								27
	MOV (d32,SP),Dn	mem32(d32+SP) → Dn	-	-	-	-	6	2	D4	1111	1100	1011	01Dn	<d32	....	....	....	....	....	....	....>				27
	MOV (Di,Am),Dn	mem32(Di+Am) → Dn	-	-	-	-	2	1	D0	1111	0011	00Dn	DiAm												27
	MOV (abs16),Dn	mem32(abs16(zero_ext)) → Dn	-	-	-	-	3	1	S2	0011	00Dn	<abs16....		....	....>										27
	MOV (abs32),Dn	mem32(abs32) → Dn	-	-	-	-	6	2	D4	1111	1100	1010	01Dn	<abs32....		....	....	....	....	....	....>				27
	MOV (Am),An	mem32(Am) → An	-	-	-	-	2	1	D0	1111	0000	0000	AnAm												27
	MOV (d8,Am),An	mem32(d8(sign_ext)+Am) → An	-	-	-	-	3	1	D1	1111	1000	0010	AnAm	<d8	....>										27
	MOV (d16,Am),An	mem32(d16(sign_ext)+Am) → An	-	-	-	-	4	1	D2	1111	1010	0010	AnAm	<d16	....	....	....>								27
	MOV (d32,Am),An	mem32(d32+Am) → An	-	-	-	-	6	2	D4	1111	1100	0010	AnAm	<d32	....	....	....	....	....	....	....>				27
	MOV (d8,SP),An	mem32(d8(zero_ext)+SP) → An	-	-	-	-	2	1	S1	0101	11An	<d8	....>												27
	MOV (d16,SP),An	mem32(d16(zero_ext)+SP) → An	-	-	-	-	4	1	D2	1111	1010	1011	00An	<d16	....	....	....>								27
	MOV (d32,SP),An	mem32(d32+SP) → An	-	-	-	-	6	2	D4	1111	1100	1011	00An	<d32	....	....	.....	....	....	....	....>				27
	MOV (Di,Am),An	mem32(Di+Am → An	-	-	-	-	2	1	D0	1111	0011	10An	DiAm												27
	MOV (abs16),An	mem32(abs16(zero_ext)) → An	-	-	-	-	4	1	D2	1111	1010	1010	00An	<abs16....		....	....>								27
	MOV (abs32),An	mem32(abs32) → An	-	-	-	-	6	2	D4	1111	1100	1010	00An	<abs32....		....	....	....	....	....	....>				27
	MOV (d8,Am),SP	mem32(d8(sign_ext)+Am) → SP	-	-	-	-	3	1	D1	1111	1000	1111	00Am	<d8	....>										27
	MOV Dm,(An)	Dm → mem32(An)	-	-	-	-	1	1	S0	0110	DmAn														28
	MOV Dm,(d8,An)	Dm → mem32(d8(sign_ext)+An)	-	-	-	-	3	1	D1	1111	1000	0111	DmAn	<d8	....>										28
	MOV Dm,(d16,An)	Dm → mem32(d16(sign_ext)+An)	-	-	-	-	4	1	D2	1111	1010	0001	DmAn	<d16	....	....	....>								28
	MOV Dm,(d32,An)	Dm → mem32(d32+An)	-	-	-	-	6	2	D4	1111	1100	0001	DmAn	<d32	....	....	....	....	....	....	....>				28
	MOV Dm,(d8,SP)	Dm → mem32(d8(zero_ext)+SP)	-	-	-	-	2	1	S1	0100	Dm10	<d8	....>												28
	MOV Dm,(d16,SP)	Dm → mem32(d16(zero_ext)+SP)	-	-	-	-	4	1	D2	1111	1010	1001	Dm01	<d16	....	....	....>								28
	MOV Dm,(d32,SP)	Dm → mem32(d32+SP)	-	-	-	-	6	2	D4	1111	1100	1001	Dm01	<d32.	....	....	....	....	....	....	....>				28
	MOV Dm,(Di,An)	Dm → mem32(Di+An)	-	-	-	-	2	2	D0	1111	0011	01Dm	DiAn												28
	MOV Dm,(abs16)	Dm → mem32(abs16(zero_ext))	-	-	-	-	3	1	S2	0000	Dm01	<abs16....		....	....>										28
	MOV Dm,(abs32)	Dm → mem32(abs32)	-	-	-	-	6	2	D4	1111	1100	1000	Dm01	<abs32....		....	....	....	....	....	....>				28

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	CF	NF	ZF	Code Size	Cycle	For -mat	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
MOV	MOV Am,(An)	Am → mem32(An)	-	-	-	-	2	1	D0	1111	0000	0001	AmAn												28
	MOV Am,(d8,An)	Am → mem32(d8(sign_ext)+An)	-	-	-	-	3	1	D1	1111	1000	0011	AmAn	<d8	....>										28
	MOV Am,(d16,An)	Am → mem32(d16(sign_ext)+An)	-	-	-	-	4	1	D2	1111	1010	0011	AmAn	<d16	....	....	....>								28
	MOV Am,(d32,An)	Am → mem32(d32+An)	-	-	-	-	6	2	D4	1111	1100	0011	AmAn	<d32	....	....	....	....	....	....	....>				28
	MOV Am,(d8,SP)	Am → mem32(d8(zero_ext)+SP)	-	-	-	-	2	1	S1	0100	Am11	<d8	....>												28
	MOV Am,(d16,SP)	Am → mem32(d16(zero_ext)+SP)	-	-	-	-	4	1	D2	1111	1010	1001	Am00	<d16	....	....	....>								28
	MOV Am,(d32,SP)	Am → mem32(d32+SP)	-	-	-	-	6	2	D4	1111	1100	1001	Am00	<d32	....	....	....	....	....	....	....>				28
	MOV Am,(Di,An)	Am → mem32(Di+An)	-	-	-	-	2	2	D0	1111	0011	11Am	DiAn												28
	MOV Am,(abs16)	Am → mem32(abs16(zero_ext))	-	-	-	-	4	1	D2	1111	1010	1000	Am00	<abs16....	....	....>									28
	MOV Am,(abs32)	Am → mem32(abs32)	-	-	-	-	6	2	D4	1111	1100	1000	Am00	<abs32....	....	....	....	....	....	....>					28
	MOV SP,(d8,An)	SP → mem32(d8(sign_ext)+An)	-	-	-	-	3	1	D1	1111	1000	1111	01An	<d8	....>										28
	MOV imm8,Dn	imm8(sign_ext) → Dn	-	-	-	-	2	1	S1	1000	DnDn	<imm8 ...>													29
	MOV imm16,Dn	imm16(sign_ext) → Dn	-	-	-	-	3	1	S2	0010	11Dn	<imm16	....	....>											29
	MOV imm32,Dn	imm32 → Dn	-	-	-	-	6	2	D4	1111	1100	1100	10Dn	<imm32...	....	....	....	....	....	....>					29
	MOV imm8,An	imm8(zero_ext] → An	-	-	-	-	2	1	S1	1001	AnAn	<imm8 ....>													29
	MOV imm16,An	imm16(zero_ext) → An	-	-	-	-	3	1	S2	0010	01An	<imm16....	....	....>											29
	MOV imm32,An	imm32 → An	-	-	-	-	6	2	D4	1111	1100	1101	11An	<imm32...	....	....	....	....	....	....>					29
MOVBU	MOVBU (Am),Dn	mem8(Am)(zero_ext) → Dn	-	-	-	-	2	1	D0	1111	0000	0100	DnAm												30
	MOVBU (d8,Am),Dn	mem8(d8(sign_ext)+Am)(zero_ext) → Dn	-	-	-	-	3	1	D1	1111	1000	0100	DnAm	<d8	....>										30
	MOVBU (d16,Am),Dn	mem8(d16(sign_ext)+Am)(zero_ext) → Dn	-	-	-	-	4	1	D2	1111	1010	0100	DnAm	<d16	....	....	....>								30
	MOVBU (d32,Am),Dn	mem8(d32+Am)(zero_ext] → Dn	-	-	-	-	6	2	D4	1111	1100	0100	DnAm	<d32	....	....	....	....	....	....	....>				30
	MOVBU (d8,SP),Dn	mem8(d8(zero_ext)+SP)(zero_ext) → Dn	-	-	-	-	3	1	D1	1111	1000	1011	10Dn	<d8	....>										30
	MOVBU (d16,SP),Dn	mem8(d16(zero_ext)+SP)(zero_ext) → Dn	-	-	-	-	4	1	D2	1111	1010	1011	10Dn	<d16	....	....	....>								30
	MOVBU (d32,SP),Dn	mem8(d32+SP)(zero_ext) → Dn	-	-	-	-	6	2	D4	1111	1100	1011	10Dn	<d32	....	....	....	....	....	....	....>				30
	MOVBU (Di,Am),Dn	mem8(Di+Am)(zero_ext) → Dn	-	-	-	-	2	1	D0	1111	0100	00Dn	DiAm												30
	MOVBU (abs16),Dn	mem8(abs16(zero_ext))(zero_ext] → Dn	-	-	-	-	3	1	S2	0011	01Dn	<abs16	....	....>											30
	MOVBU (abs32),Dn	mem8(abs32)(zero_ext) → Dn	-	-	-	-	6	2	D4	1111	1100	1010	10Dn	<abs32	....	....	....	....	....	....>					30
	MOVBU Dm,(An)	Dm → mem8(An)	-	-	-	-	2	1	D0	1111	0000	0101	DmAn												31
	MOVBU Dm,(d8,An)	Dm → mem8(d8+An)	-	-	-	-	3	1	D1	1111	1000	0101	DmAn	<d8	....>										31
	MOVBU Dm,(d16,An)	Dm → mem8(d16+An)	-	-	-	-	4	1	D2	1111	1010	0101	DmAn	<d16	....	....	....>								31
	MOVBU Dm,(d32,An)	Dm → mem8(d32+An)	-	-	-	-	6	2	D4	1111	1100	0101	DmAn	<d32	....	....	....	....	....	....	....>				31
	MOVBU Dm,(d8,SP)	Dm → mem8(d8(zero_ext)+SP)	-	-	-	-	3	1	D1	1111	1000	1001	Dm10	<d8	....>										31
	MOVBU Dm,(d16,SP)	Dm → mem8(d16(zero_ext)+SP)	-	-	-	-	4	1	D2	1111	1010	1001	Dm10	<d16	....	....	....>								31
	MOVBU Dm,(d32,SP)	Dm → mem8(d32+SP)	-	-	-	-	6	2	D4	1111	1100	1001	Dm10	<d32	....	....	....	....	....	....	....>				31
	MOVBU Dm,(Di,An)	Dm → mem8(Di+An)	-	-	-	-	2	2	D0	1111	0100	01Dm	DiAn												31
	MOVBU Dm,(abs16)	Dm → mem8(abs16(zero_ext))	-	-	-	-	3	1	S2	0000	Dm10	<abs16	....	....>											31
	MOVBU Dm,(abs32)	Dm → mem8(abs32)	-	-	-	-	6	2	D4	1111	1100	1000	Dm10	<abs32	....	....	....	....	....	....>					31

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
MOVB	MOVB (Am),Dn	mem8(Am)(sign_ext) → Dn	-	-	-	-	3	2																	32
	MOVB (d8,Am),Dn	mem8(d8(sign_ext)+Am)(sign_ext) → Dn	-	-	-	-	4	2																	32
	MOVB (d16,Am),Dn	mem8(d16(sign_ext)+Am)(sign_ext) → Dn	-	-	-	-	5	2																	32
	MOVB (d32,Am),Dn	mem8(d32+Am)(sign_ext) → Dn	-	-	-	-	7	3																	32
	MOVB (d8,SP),Dn	mem8(d8(zero_ext)+SP)(sign_ext) → Dn	-	-	-	-	4	2																	32
	MOVB (d16,SP),Dn	mem8(d16(zero_ext)+SP)(sign_ext) → Dn	-	-	-	-	5	2																	32
	MOVB (d32,SP),Dn	mem8(d32+SP)(sign_ext) → Dn	-	-	-	-	7	3																	32
	MOVB (Di,Am),Dn	mem8(Di+Am)(sign_ext) → Dn	-	-	-	-	3	2																	32
	MOVB (abs16),Dn	mem8(abs16(zero_ext))(sign_ext) → Dn	-	-	-	-	4	2																	32
	MOVB (abs32),Dn	mem8(abs32)(sign_ext) → Dn	-	-	-	-	7	3																	32
	MOVB Dm,(An)	Dm → mem8(An)	-	-	-	-	2	1																	33
	MOVB Dm,(d8,An)	Dm → mem8(d8(sign_ext)+An)	-	-	-	-	3	1																	33
	MOVB Dm,(d16,An)	Dm → mem8(d16(sign_ext)+An)	-	-	-	-	4	1																	33
	MOVB Dm,(d32,An)	Dm → mem8(d32+An)	-	-	-	-	6	2																	33
	MOVB Dm,(d8,SP)	Dm → mem8(d8(zero_ext)+SP)	-	-	-	-	3	1																	33
	MOVB Dm,(d16,SP)	Dm → mem8(d16(zero_ext)+SP)	-	-	-	-	4	1																	33
	MOVB Dm,(d32,SP)	Dm → mem8(d32+SP)	-	-	-	-	6	2																	33
	MOVB Dm,(Di,An)	Dm → mem8(Di+An)	-	-	-	-	2	2																	33
	MOVB Dm,(abs16)	Dm → mem8(abs16(zero_ext))	-	-	-	-	3	1																	33
	MOVB Dm,(abs32)	Dm → mem8(abs32)	-	-	-	-	6	2																	33
MOVHU	MOVHU (Am),Dn	mem16(Am)(zero_ext) → Dn	-	-	-	-	2	1	D0	1111	0000	0110	DnAm												34
	MOVHU (d8,Am),Dn	mem16(d8(sign_ext)+Am)(zero_ext) → Dn	-	-	-	-	3	1	D1	1111	1000	0110	DnAm	<d8	....>										34
	MOVHU (d16,Am),Dn	mem16(d16(sign_ext)+Am)(zero_ext) → Dn	-	-	-	-	4	1	D2	1111	1010	0110	DnAm	<d16	....	....	....>								34
	MOVHU (d32,Am),Dn	mem16(d32+Am)(zero_ext) → Dn	-	-	-	-	6	2	D4	1111	1100	0110	DnAm	<d32	....	....	....	....	....	....	....>				34
	MOVHU (d8,SP),Dn	mem16(d8(zero_ext)+SP)(zero_ext) → Dn	-	-	-	-	3	1	D1	1111	1000	1011	11Dn	<d8	....>										34
	MOVHU (d16,SP),Dn	mem16(d16(zero_ext)+SP)(zero_ext) → Dn	-	-	-	-	4	1	D2	1111	1010	1011	11Dn	<d16	....	....	....>								34
	MOVHU (d32,SP),Dn	mem16(d32+SP)(zero_ext) → Dn	-	-	-	-	6	2	D4	1111	1100	1011	11Dn	<d32	....	....	....	....	....	....	....>				34
	MOVHU (Di,Am),Dn	mem16(Di+Am)(zero_ext) → Dn	-	-	-	-	2	1	D0	1111	0100	10Dn	DiAm												34
	MOVHU (abs16),Dn	mem16(abs16(zero_ext))(zero_ext) → Dn	-	-	-	-	3	1	S2	0011	10Dn	<abs16	....	....	....>										34
	MOVHU (abs32),Dn	mem16(abs32)(zero_ext) → Dn	-	-	-	-	6	2	D4	1111	1100	1010	11Dn	<abs32	....	....	....	....	....	....	....>				34
	MOVHU Dm,(An)	Dm → mem16(An)	-	-	-	-	2	1	D0	1111	0000	0111	DmAn												35
	MOVHU Dm,(d8,An)	Dm → mem16(d8(sign_ext)+An)	-	-	-	-	3	1	D1	1111	1000	0111	DmAn	<d8	....>										35
	MOVHU Dm,(d16,An)	Dm → mem16(d16(sign_ext)+An)	-	-	-	-	4	1	D2	1111	1010	0111	DmAn	<d16	....	....	....>								35
	MOVHU Dm,(d32,An)	Dm → mem16(d32+An)	-	-	-	-	6	2	D4	1111	1100	0111	DmAn	<d32	....	....	....	....	....	....	....>				35
	MOVHU Dm,(d8,SP)	Dm → mem16(d8(zero_ext)+SP)	-	-	-	-	3	1	D1	1111	1000	1001	Dm11	<d8	....>										35
	MOVHU Dm,(d16,SP)	Dm → mem16(d16(zero_ext)+SP)	-	-	-	-	4	1	D2	1111	1010	1001	Dm11	<d16	....	....	....>								35
	MOVHU Dm,(d32,SP)	Dm → mem16(d32+SP)	-	-	-	-	6	2	D4	1111	1100	1001	Dm11	<d32	....	....	....	....	....	....	....>				35
	MOVHU Dm,(Di,An)	Dm → mem16(Di+An)	-	-	-	-	2	2	D0	1111	0100	11Dm	DiAn												35
	MOVHU Dm,(abs16)	Dm → mem16(abs16(zero_ext))	-	-	-	-	3	1	S2	0000	Dm11	<abs16	....	....	....>										35
	MOVHU Dm,(abs32)	Dm → mem16(abs32)	-	-	-	-	6	2	D4	1111	1100	1000	Dm11	<abs32	....	....	....	....	....	....	....>				35

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	Format	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
MOVH	MOVH (Am),Dn	mem16(Am)(sign_ext) → Dn	-	-	-	-	3	2																	36
	MOVH (d8,Am),Dn	mem16(d8(sign_ext)+Am)(sign_ext) → Dn	-	-	-	-	4	2																	36
	MOVH (d16,Am),Dn	mem16(d16(sign_ext)+Am)(sign_ext) → Dn	-	-	-	-	5	2																	36
	MOVH (d32,Am),Dn	mem16(d32+Am)(sign_ext) → Dn	-	-	-	-	7	3																	36
	MOVH (d8,SP),Dn	mem16(d8(zero_ext)+SP)(sign_ext) → Dn	-	-	-	-	4	2																	36
	MOVH (d16,SP),Dn	mem16(d16(zero_ext)+SP)(sign_ext) → Dn	-	-	-	-	5	2																	36
	MOVH (d32,SP),Dn	mem16(d32+SP)(sign_ext) → Dn	-	-	-	-	7	3																	36
	MOVH (Di,Am),Dn	mem16(Di+Am)(sign_ext) → Dn	-	-	-	-	3	2																	36
	MOVH (abs16),Dn	mem16(abs16(zero_ext))(sign_ext) → Dn	-	-	-	-	4	2																	36
	MOVH (abs32),Dn	mem16(abs32)(sign_ext) → Dn	-	-	-	-	7	3																	36
	MOVH Dm,(An)	Dm → mem16(An)	-	-	-	-	2	1																	37
	MOVH Dm,(d8,An)	Dm → mem16(d8(sign_ext)+An)	-	-	-	-	3	1																	37
	MOVH Dm,(d16,An)	Dm → mem16(d16(sign_ext)+An)	-	-	-	-	4	1																	37
	MOVH Dm,(d32,An)	Dm → mem16(d32+An)	-	-	-	-	6	2																	37
	MOVH Dm,(d8,SP)	Dm → mem16(d8(zero_ext)+SP)	-	-	-	-	3	1																	37
	MOVH Dm,(d16,SP)	Dm → mem16(d16(zero_ext)+SP)	-	-	-	-	4	1																	37
	MOVH Dm,(d32,SP)	Dm → mem16(d32+SP)	-	-	-	-	6	2																	37
	MOVH Dm,(Di,An)	Dm → mem16(Di+An)	-	-	-	-	2	2																	37
	MOVH Dm,(abs16)	Dm → mem16(abs16(zero_ext))	-	-	-	-	3	1																	37
	MOVH Dm,(abs32)	Dm → mem16(abs32)	-	-	-	-	6	2																	37
MOVM	MOVM (SP),[reg1,.,regn]	PC+2 → PC	-	-	-	-	2	1	S1	1100	1110	<regs	....>											registers specified with regs = 0	38
		mem32(SP) → reg,SP+4 → SP	-	-	-	-	2	2																registers specified with regs = 1	38
		mem32(SP+4) → reg1,mem32(SP) → reg2, SP+8 → SP	-	-	-	-	2	3																registers specified with regs = 2(*1)	38
		mem32(SP+8) → reg1,mem32(SP+4) → reg2 mem32(SP) → reg3,SP+12 → SP	-	-	-	-	2	4																registers specified with regs = 3(*1)	38
		mem32(SP+12) → D2,mem32(SP+8) → D3, mem32(SP+4) → A2,mem32(SP) → A3, SP+16 → SP	-	-	-	-	2	5																registers specified with regs = =4(*1)	38
		mem32(SP+28) → D0,mem32(SP+24) → D1, mem32(SP+20) → A0,mem32(SP+16) → A1, mem32(SP+12) → MDR,mem32(SP+8) → LIR, mem32(SP+4) → LAR,SP+32 → SP	-	-	-	-	2	8																registers specified with regs = 7	38
		mem32(SP+32) → reg,mem32(SP+28) → D0, mem32(SP+24) →D1,mem32(SP+20) →A0, mem32(SP+16) →A1,mem32(SP+12) →MDR, mem32(SP+8) →LIR,mem32(SP+4) →LAR, SP+36 →SP	-	-	-	-	2	9																registers specified with regs = 8	38
		mem32(SP+36)→reg1, mem32(SP+32)→reg2, mem32(SP+28)→D0,mem32(SP+24)→D1, mem32(SP+20)→A0,mem32(SP+16)→A1, mem32(SP+12)→MDR,mem32(SP+8)→LIR, mem32(SP+4)→LAR,SP+40 →SP	-	-	-	-	2	10																registers specified with regs = 9(*1)	38

*1: registers specified with regn are returned in the order; D2, D3, A2 and A3 no matter when the assembler srites these registers. Skip the registers which is not specified.

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	CF	NF	ZF	Code Size	Cycle	For-mat	Machine Code (1 2 3 4 5 6 7 8 9 10 11 12 13 14)	Notes	Page
MOVM	MOVM (SP),[reg1,.,regn]	mem32(SP+40)→reg1,mem32(SP+36)→reg2, mem32(SP+32)→reg3,mem32(SP+28)→D0, mem32(SP+24)→D1,mem32(SP+20)→A0, mem32(SP+16)→A1,mem32(SP+12)→MDR, mem32(SP+8)→LIR,mem32(SP+4)→LAR, SP+44→SP	-	-	-	-	2	11	S1	1100 1110 <regs ....>	registers specified with regs=10(*1)	38
		mem32(SP+44)→D2,mem32(SP+40)→D3, mem32(SP+36)→A2,mem32(SP+32)→A3, mem32(SP+28)→D0,mem32(SP+24)→D1, mem32(SP+20)→A0,mem32(SP+16)→A1, mem32(SP+12)→MDR,mem32(SP+8)→LIR, mem32(SP+4)→LAR,SP+48→SP	-	-	-	-	2	12			rregisters specified with regs=11	38
	MOVM [reg1,.,regn],(SP)	PC+2→PC	-	-	-	-	2	1	S1	1100 1111 <regs ....>	registers specified with regs =0	39
		reg→mem32(SP-4),SP-4→SP	-	-	-	-	2	1			registers specified with regs=1	39
		reg1→mem32(SP-4),reg2→mem32(SP-8), SP-8→SP	-	-	-	-	2	2			registers specified with regs= 2(*2)	39
		reg1→mem32(SP-4),reg2→mem32(SP-8), reg3→mem32(SP-12),SP-12→SP	-	-	-	-	2	3			registers specified with regs= 3(*2)	39
		D2→mem32(SP-4),D3→mem32(SP-8), A2→mem32(SP-12),A3→mem32(SP-16), SP-16→SP	-	-	-	-	2	4			registers specified with regs= 4	39
		D0→mem32(SP-4),D1→mem32(SP-8), A0→mem32(SP-12),A1→mem32(SP-16), MDR→mem32(SP-20),LIR→mem32(SP-24), LAR→mem32(SP-28),SP-32→SP	-	-	-	-	2	8			registers specified with regs= 7	39
		reg→mem32(SP-4),D0→mem32(SP-8), D1→mem32(SP-12),A0→mem32(SP-16), A1→mem32(SP-20),MDR→mem32(SP-24), LIR→mem32(SP-28),LAR→mem32(SP-32), SP-36→SP	-	-	-	-	2	9			registers specified with regs= 8(*2)	39
		reg1→mem32(SP-4),reg2→mem32(SP-8), D0→mem32(SP-12),D1→mem32(SP-16), A0→mem32(SP-20),A1→mem32(SP-24), MDR→mem32(SP-28),LIR→mem32(SP-32), LAR→mem32(SP-36),SP-40→SP	-	-	-	-	2	10			registers specified with regs= 9(*2)	39
		reg1→mem32(SP-4),reg2→mem32(SP-8), reg3→mem32(SP-12),D0→mem32(SP-16), D1→mem32(SP-20),A0→mem32(SP-24), A1→mem32(SP-28),MDR→mem32(SP-32), LIR→mem32(SP-36),LAR→mem32(SP-40),, SP-44→SP	-	-	-	-	2	11			registers specified with regs= 10(*2)	39

*1: Registers specified with regn are returned in the order; D2, D3, A2 and A3 no matter when the assembler writes theses registers. Skip the registers which is not specified

*2: Registers specified with regn are saved in the order; D2, D3, A2 and A3 no matter when the assembler write these registers. Skip the registers which is not specified.

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
MOVM	MOVM regs,(SP)	D2 → mem32(SP-4),D3 → mem32(SP-8) A2 → mem32(SP-12),A3 → mem32(SP-16) D0 → mem32(SP-20),D1 → mem32(SP-24) A0 → mem32(SP-28),A1 → mem32(SP-32) MDR → mem32(SP-36),LIR → mem32(SP-40) LAR → mem32(SP-44),SP-48 → SP	-	-	-	-	2	12	S1	1100	1111	<regs ....>												Registers specified with regs =11	39
EXT	EXT Dn	IF (Dn.bp31=0), 0x00000000 → MDR IF (Dn.bp31=1), 0xFFFFFFFF → MDR	-	-	-	-	2	1	D0	1111	0010	1101	00Dn												40
EXTB	EXTB Dn	IF (Dn.bp7=0), Dn & 0x000000FF → Dn IF (Dn.bp7=1), Dn | 0xFFFFFF00 → Dn	-	-	-	-	1	1	S0	0001	00Dn														41
EXTBU	EXTBU Dn	Dn & 0x000000FF → Dn	-	-	-	-	1	1	S0	0001	01Dn														42
EXTH	EXTH Dn	IF (Dn.bp15=0), Dn & 0x0000FFFF → Dn IF (Dn.bp15=1), Dn | 0xFFFF0000 → Dn	-	-	-	-	1	1	S0	0001	10Dn														43
EXTHU	EXTHU Dn	Dn & 0x0000FFFF → Dn	-	-	-	-	1	1	S0	0001	11Dn														44
CLR	CLR Dn	0 → Dn	0	0	0	1	1	1	S0	0000	Dn00														45

Arithmetic Operation Instructions

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
ADD	ADD Dm,Dn	Dm + Dn → Dn	●	●	●	●	1	1	S0	1110	DmDn														46
	ADD Dm,An	Dm + An → An	●	●	●	●	2	1	D0	1111	0001	0101	AmDn												46
	ADD Am,Dn	Am + Dn → Dn	●	●	●	●	2	1	D0	1111	0001	0110	DmAn												46
	ADD Am,An	Am + An → An	●	●	●	●	2	1	D0	1111	0001	0111	AmAn												46
	ADD imm8,Dn	imm8(sign_ext) + Dn → Dn	●	●	●	●	2	1	S1	0010	10Dn	<imm8 ....>													47
	ADD imm16,Dn	imm16(sign_ext) + Dn → Dn	●	●	●	●	4	1	D2	1111	1010	1100	00Dn	<imm16....		....	....>								47
	ADD imm32,Dn	imm32 + Dn → Dn	●	●	●	●	6	2	D4	1111	1100	1100	00Dn	<imm32....		....	....	....	....	....	....>				47
	ADD imm8,An	imm8(sign_ext) + An → An	●	●	●	●	2	1	S1	0010	00An	<imm8 ....>													47
	ADD imm16,An	imm16(sign_ext) + An → An	●	●	●	●	4	1	D2	1111	1010	1101	00An	<imm16....		....	....>								47
	ADD imm32,An	imm32 + An → An	●	●	●	●	6	2	D4	1111	1100	1101	00An	<imm32....		....	....	....	....	....	....>				47
	ADD imm8,SP	imm8(sign_ext) + SP → SP	-	-	-	-	3	1	D1	1111	1000	1111	1110	<imm8 ....>											47
	ADD imm16,SP	imm16(sign_ext) + SP → SP	-	-	-	-	4	1	D2	1111	1010	1111	1110	<imm16....		....	....>								47
	ADD imm32,SP	imm32 + SP → SP	-	-	-	-	6	2	D4	1111	1100	1111	1110	<imm32....		....	....	....	....	....	....>				47
ADDC	ADDC Dm,Dn	Dm + Dn + CF → Dn	●	●	●	●	2	1	D0	1111	0001	0100	DmDn												48
SUB	SUB Dm,Dn	Dn - Dm → Dn	●	●	●	●	2	1	D0	1111	0001	0000	DmDn												49
	SUB Dm,An	An - Dm → An	●	●	●	●	2	1	D0	1111	0001	0010	DmAn												49
	SUB Am,Dn	Dn - Am → Dn	●	●	●	●	2	1	D0	1111	0001	0001	AmDn												49
	SUB Am,An	An - Am → An	●	●	●	●	2	1	D0	1111	0001	0011	AmAn												49
	SUB imm32,Dn	Dn - imm32 → Dn	●	●	●	●	6	2	D4	1111	1100	1100	10Dn	<imm32....		....	....	....	....	....	....>				50
	SUB imm32,An	An - imm32 → An	●	●	●	●	6	2	D4	1111	1100	1101	01An	<imm32....		....	....	....	....	....	....>				50
SUBC	SUBC Dm,Dn	Dn - Dm - CF → Dn	●	●	●	●	2	1	D0	1111	0001	1000	DmDn												51
MUL	MUL Dm,Dn	(Dn* Dn) → { MDR , Dn }	?	?	●	●	2	3	D0	1111	0010	0100	DmDn											Dn =0	52
								13																Dn =value by 1 byte	52
								21																Dn =value by 2-byte	52
								29																Dn =value by 3-byte	52
								34																Dn =value by 4-byte	52

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	Machine Code (1–14)	Notes	Page
MULU	MULU Dm,Dn	(Dn* Dn) → { MDR , Dn }	?	?	●	●	2	3	D0	1111 0010 0101 DmDn	Dn = 0	53
								13			Dn = value by 1 byte	53
								21			Dn = value by 2-byte	53
								29			Dn = value by 3-byte	53
								34			Dn = value by 4-byte	53
DIV	DIV Dm,Dn	((MDR<< 32) & 0xFFFFFFFF00000000 + Dn)	0/1	?/?	●/?	●/?	2	4	D0	1111 0010 0110 DmDn	{MDR,Dn} = 0	54
		/ Dm → Dn						14			{MDR,Dn}= value by 1 byte	54
		((MDR<< 32) & 0xFFFFFFFF00000000 + Dn)						22			{MDR,Dn} = value by 2- byte	54
		% Dm → MDR						30			{MDR,Dn} = value by 3-byte	54
								38			{MDR,Dn} = value by 4- byte or more	54
DIVU	DIVU Dm,Dn	((MDR<< 32) & 0xFFFFFFFF00000000 + Dn)	0/1	?/?	●/?	●/?	2	4	D0	1111 0010 0111 DmDn	{MDR,Dn} = 0	55
		/ Dm → Dn						14			{MDR,Dn} = value by 1 byte	55
		((MDR<< 32) & 0xFFFFFFFF00000000+ Dn)						22			{MDR,Dn} = value by 2- byte	55
		% Dm → MDR						30			{MDR,Dn} = value by 3-byte	55
								38			{MDR,Dn} = value by 4- byte or more	55
INC	INC Dn	Dn + 1 → Dn	●	●	●	●	1	1	S0	0100 Dn00		56
	INC An	An + 1 → An	-	-	-	-	1	1	S0	0100 An01		56
INC4	INC4 An	An + 4 → An	-	-	-	-	1	1	S0	0101 00An		57

Comparative Instructions

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	Machine Code (1–14)	Notes	Page
CMP	CMP Dm,Dn	Dn - Dm : PSW	●	●	●	●	1	1	S0	1010 DmDn		58
	CMP Dm,An	An - Dm : PSW	●	●	●	●	2	1	D0	1111 0001 1010 DmAn		58
	CMP Am,Dn	Dn - Am : PSW	●	●	●	●	2	1	D0	1111 0001 1001 AmDn		58
	CMP Am,An	An - Am : PSW	●	●	●	●	1	1	S0	1011 AmAn		58
	CMP imm8,Dn	Dn - imm8(sign_ext) : PSW	●	●	●	●	2	1	S1	1010 DnDn <imm8 ....>		58
	CMP imm16,Dn	Dn - imm16(sign_ext) : PSW	●	●	●	●	4	1	D2	1111 1010 1100 10Dn <imm16 ....> ....>		58
	CMP imm32,Dn	Dn - imm32 : PSW	●	●	●	●	6	2	D4	1111 1100 1100 10Dn <imm32		58
	CMP imm8,An	An - imm8(zero_ext) : PSW	●	●	●	●	2	1	S1	1011 AnAn <imm8 ....>		58
	CMP imm16,An	An - imm16(zero_ext) : PSW	●	●	●	●	4	1	D2	1111 1010 1101 10An <imm16 ....> ....>		58
	CMP imm32,An	An - imm32 : PSW	●	●	●	●	6	2	D4	1111 1100 1101 10An <imm32		58

Logical Operation Instructions

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	Machine Code (1–14)	Notes	Page
AND	AND Dm,Dn	Dm & Dn → Dn	0	0	●	●	2	1	D0	1111 0010 0000		59
	AND imm8,Dn	imm8(zero_ext) & Dn → Dn	0	0	●	●	3	1	D1	1111 1000 1110 00Dn <imm8 ....>		59
	AND imm16,Dn	imm16(zero_ext) & Dn → Dn	0	0	●	●	4	1	D2	1111 1010 1110 00Dn <imm16 ....> ....>		59
	AND imm32,Dn	imm32 & Dn → Dn	0	0	●	●	6	2	D4	1111 1100 1110 00Dn <imm32		59
	AND imm16,PSW	imm16(zero_ext) & PSW → PSW	●	●	●	●	4	1	D2	1111 1010 1111 1100 <imm16 ....>		60
OR	OR Dm,Dn	Dm | Dn → Dn	0	0	●	●	2	1	D0	1111 0010 0001 DmDn		61
	OR imm8,Dn	imm8(zero_ext) | Dn → Dn	0	0	●	●	3	1	D1	1111 1000 1110 01Dn <imm8 ....>		61
	OR imm16,Dn	imm16(zero_ext) | Dn → Dn	0	0	●	●	4	1	D2	1111 1010 1110 01Dn <imm16 ....> ....>		61
	OR imm32,Dn	imm32 | Dn → Dn	0	0	●	●	6	2	D4	1111 1100 1110 0 Dn <imm32		61
	OR imm16,PSW	imm16(zero_ext) | PSW → PSW	●	●	●	●	4	1	D2	1111 1010 1111 1101 <imm16 ....>		62

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag				Code Size	Cycle	For -mat	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
			VF	CF	NF	ZF																			
XOR	XOR Dm,Dn	Dm ^ Dn → Dn	0	0	●	●	2	1	D0	1111	0010	0010	DmDn												63
	XOR imm16,Dn	imm16(zero_ext) ^ Dn → Dn	0	0	●	●	4	1	D2	1111	1010	1110	10Dn	<imm16		....	....>								63
	XOR imm32,Dn	imm32 ^ Dn → Dn	0	0	●	●	6	2	D4	1111	1100	1110	10Dn	<imm32		....	....	....	....	....	....>				63
NOT	NOT Dn	Dn ^ 0xFFFFFFFF → Dn	0	0	●	●	2	1	D0	1111	0010	0011	00Dn												64
Bit Instructions																									
BTST	BTST imm8,Dn	imm8(zero_ext) & Dn : PSW	0	0	●	●	3	1	D1	1111	1000	1110	11Dn	<imm8 ....>											65
	BTST imm16,Dn	imm16(zero_ext) & Dn : PSW	0	0	●	●	4	1	D2	1111	1010	1110	11Dn	<imm16		....	....>								65
	BTST imm32,Dn2	imm32 & Dn : PSW	0	0	●	●	6	2	D4	1111	1100	1110	11Dn	<imm32		....	....	....	....	....	....>				65
	BTST imm8,(d8,An)	imm8(zero_ext) & mem8(d8(sign_ext)+An)(zero_ext) : PSW	0	0	0	●	4	4	D2	1111	1010	1111	10An	<d8	....>	<imm8 ....>									65
	BTST imm8,(abs16)	imm8(zero_ext) & mem8(abs16(zero_ext))(zero_ext) : PSW	0	0	0	●	5	4	D3	1111	1110	1000	0010	<abs16		....	....>	<imm8 ....>						Not used for AM30/AM31	65
	BTST imm8,(abs32)	imm8(zero_ext) & mem8(abs32)(zero_ext) : PSW	0	0	0	●	7	5	D5	1111	1110	0000	0010	<abs32		....	....	....	....	....	....>	<imm8 ....>			65
BSET	BSET Dm,(An)	mem8(An)(zero_ext) → temp, temp & Dm : PSW, temp | Dm → mem8(An)	0	0	0	●	2	5	D0	1111	0000	1000	DmAn												66
	BSET imm8,(d8,An)	mem8(d8(sign_ext) + An)(zero_ext) → temp, temp & imm8(zero_ext): PSW, temp | imm8(zero_ext) → mem8(d8(sign_ext) + An)	0	0	0	●	4	5	D2	1111	1010	1111	00An	<d8	....>	<imm8 ....>									67
	BSET imm8,(abs16)	mem8(abs16(zero_ext))(zero_ext) → temp , temp & imm8(zero_ext) : PSW, temp | imm8(zero_ext) → mem8(abs16(zero_ext))	0	0	0	●	5	5	D3	1111	1110	1000	0000	<abs16		....	....>	<imm8 ....>						Not used for AM30/AM31	67
	BSET imm8,(abs32)	mem8(abs32)(zero_ext) → temp , temp & imm8(zero_ext) : PSW, temp | imm8(zero_ext) → mem8(abs32)	0	0	0	●	7	6	D5	1111	1110	0000	0000	<abs32		....	....	....	....	....	....>	<imm8 ....>			67
BCLR	BCLR Dm,(An)	mem8(An)(zero_ext) → temp , temp & Dm : PSW, temp & (Dm ^ 0xFFFFFFFF) → mem8(An)	0	0	0	●	2	5	D0	1111	0000	1001	DmAn												68
	BCLR imm8,(d8,An)	mem(d8(sign_ext) + An)(zero_ext) → temp, temp & imm8(zero_ext) : PSW, temp & (imm8(zero_ext) ^ 0xFFFFFFFF) → mem8(d8(sign_ext) + An)	0	0	0	●	4	5	D2	1111	1010	1111	01An	<d8	....>	<imm8 ....>									69
	BCLR imm8,(abs16)	mem8(abs16(zero_ext))(zero_ext) → temp , temp & imm8(zero_ext) : PSW, temp & (imm8(zero_ext) ^ 0xFFFFFFFF) → mem8(abs16(zero_ext))	0	0	0	●	5	5	D3	1111	1110	1000	0001	<abs16		....	....>	<imm8 ....>						Not used for AM30/AM31	69
	BCLR imm8,(abs32)	mem8(abs32)(zero_ext) → temp , temp & imm8(zero_ext) : PSW, temp & (imm8(zero_ext) ^ 0xFFFFFFFF) → mem8(abs32)	0	0	0	●	7	6	D5	1111	1110	0000	0001	<abs32		....	....	....	....	....	....>	<imm8 ....>			69

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
Shift Instructions																									
ASR	ASR Dm,Dn	IF ((Dm&0x0000001F) ≠ 0) , Dn.lsb → CF, (Dn >> (Dm & 0x0000001F))(sign_ext) → Dn	?	●	●	●	2	1	D0	1111	0010	1011	DmDn												70
		IF ((Dm&0x0000001F)=0),PC + 2 → PC	?	?	●	●																			
	ASR imm8,Dn	IF ((imm8 & 0x1F) ≠0) , Dn.lsb → CF, (Dn >> (imm8 & 0x1F))(sign_ext) → Dn	?	●	●	●	3	1	D1	1111	1000	1100	10Dn	<imm8 ....>											71
		IF ((imm8 & 0x1F)=0) , PC + 3 → PC	?	?	●	●																			
	ASR Dn	Dn.lsb → CF, (Dn >> 1)(sign_ext) → Dn	?	●	●	●	3	1																	72
LSR	LSR Dm,Dn	IF ((Dm&0x0000001F) ≠ 0),Dn.lsb → CF, (Dn >> (Dm & 0x0000001F))(zero_ext) → Dn	?	●	●	●	2	1	D0	1111	0010	1010	DmDn												73
		IF ((Dm&0x0000001F)=0), PC + 2 → PC	?	?	●	●																			
	LSR imm8,Dn	IF ((imm8 & 0x1F) ≠ 0),Dn.lsb → CF, (Dn >> (imm8 & 0x1F))(zero_ext) → Dn	?	●	●	●	3	1	D1	1111	1000	1100	01Dn	<imm8 ....>											74
		IF ((imm8 & 0x1F)=0), PC + 3 → PC	?	?	●	●																			
	LSR Dn	Dn.lsb → CF, (Dn >> 1)(zero_ext) → Dn	?	●	●	●	3	1																	75
ASL	ASL Dm,Dn	IF ((Dm & 0x0000001F) ≠ 0), Dn << (Dm & 0x0000001F) → Dn IF ((Dm & 0x0000001F)=0), PC + 2 → PC	?	?	●	●	2	1	D0	1111	0010	1001	DmDn												76
	ASL imm8,Dn	IF ((imm8 & 0x1F) ≠0), Dn << (imm8 & 0x1F) → Dn IF ((imm8 & 0x1F)=0), PC + 3 → PC	?	?	●	●	3	1	D1	1111	1000	1100	00Dn	<imm8 ....>											77
ASL2	ASL2 Dn	(Dn << 2) & 0xFFFFFFFC → Dn	?	?	●	●	1	1	S0	0101	01Dn														78
ROR	ROR Dn	CF << 31 → temp, Dn.lsb → CF, (Dn >> 1)(zero_ext) | temp → Dn	0	●	●	●	2	1	D0	1111	0010	1000	01Dn												79
ROL	ROL Dn	CF → temp, Dn.msb → CF, (Dn << 1) | temp → Dn	0	●	●	●	2	1	D0	1111	0010	1000	00Dn												80

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	CF	NF	ZF	Code Size	Cycle	For-mat	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
Branch Instructions																									
Bcc	BEQ (d8,PC)	IF (ZF=1), PC + d8(sign_ext)→ PC IF (ZF=0),PC + 2 → PC	-	-	-	-	2	3/1*	S1	1100	1000	<d8	....>											Branch enable/disable	81
	BNE (d8,PC)	IF (ZF=0), PC + d8(sign_ext) → PC IF (ZF=1), PC + 2 → PC	-	-	-	-	2	3/1*	S1	1100	1001	<d8	....>											Branch enable/disable	81
	BGT (d8,PC)	IF ((ZF | (NF^VF))=0),PC + d8(sign_ext) → PC IF ((ZF | (NF^VF))=1),PC + 2 → PC	-	-	-	-	2	3/1*	S1	1100	0001	<d8	....>											Branch enable/disable	81
	BGE (d8,PC)	IF ((NF ^ VF)=0),PC + d8(sign_ext) → PC IF ((NF ^ VF)=1),PC + 2 → PC	-	-	-	-	2	3/1*	S1	1100	0010	<d8	....>											Branch enable/disable	81
	BLE (d8,PC)	IF ((ZF | (NF^VF))=1),PC + d8(sign_ext) → PC IF ((ZF | (NF^VF))=0),PC + 2 → PC	-	-	-	-	2	3/1*	S1	1100	0011	<d8	....>											Branch enable/disable	81
	BLT (d8,PC)	IF ((NF ^ VF)=1),PC + d8(sign_ext) →PC IF ((NF ^ VF)=0),PC + 2 → PC	-	-	-	-	2	3/1*	S1	1100	0000	<d8	....>											Branch enable/disable	81
	BHI (d8,PC)	IF ((CF | ZF)=0),PC + d8(sign_ext) → PC IF ((CF | ZF)=1),PC + 2 →PC	-	-	-	-	2	3/1*	S1	1100	0101	<d8	....>											Branch enable/disable	81
	BCC (d8,PC)	IF (CF = 0),PC + d8(sign_ext) → PC IF (CF = 1),PC + 2 → PC	-	-	-	-	2	3/1*	S1	1100	0110	<d8	....>											Branch enable/disable	81
	BLS (d8,PC)	IF ((CF | ZF)=1),PC + d8(sign_ext) → PC IF ((CF | ZF)=0),PC + 2 → PC	-	-	-	-	2	3/1*	S1	1100	0111	<d8	....>											Branch enable/disable	81
	BCS (d8,PC)	IF (CF = 1),PC + d8(sign_ext) → PC IF (CF = 0),PC + 2 → PC	-	-	-	-	2	3/1*	S1	1100	0100	<d8	....>											Branch enable/disable	81
	BVC (d8,PC)	IF (VF = 0),PC + d8(sign_ext) → PC IF (VF = 1),PC + 3 → PC	-	-	-	-	3	4/2*	D1	1111	1000	1110	1000	<d8	....>									Branch enable/disable	81
	BVS (d8,PC)	IF (VF = 1),PC + d8(sign_ext) → PC IF (VF = 0),PC + 3 → PC	-	-	-	-	3	4/2*	D1	1111	1000	1110	1001	<d8	....>									Branch enable/disable	81
	BNC (d8,PC)	IF (NF = 0),PC + d8(sign_ext) → PC IF (NF = 1),PC + 3 → PC	-	-	-	-	3	4/2*	D1	1111	1000	1110	1010	<d8	....>									Branch enable/disable	81
	BNS (d8,PC)	IF (NF = 1),PC + d8(sign_ext) → PC IF (NF = 0),PC + 3 → PC	-	-	-	-	3	4/2*	D1	1111	1000	1110	1011	<d8	....>									Branch enable/disable	81
	BRA (d8,PC)	PC + d8(sign_ext) →PC	-	-	-	-	2	3	S1	1100	1010	<d8	....>												81
Lcc	LEQ	IF (ZF=1), LAR - 4 →PC IF (ZF=0),PC + 1 →PC	-	-	-	-	1	1/2*	S0	1101	1000													Branch enable/disable	82
	LNE	IF (ZF=0), LAR - 4 →PC IF (ZF=1), PC + 1 →PC	-	-	-	-	1	1/2*	S0	1101	1001													Branch enable/disable	82

*:Depends on the status of Instruction queue.

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag				Code Size	Cycle	For -mat	Machine Code														Notes	Page
			VF	CF	NF	ZF				1	2	3	4	5	6	7	8	9	10	11	12	13	14		
Lcc	LGT	IF ((ZF | (NF^VF))=0), LAR - 4 → PC IF ((ZF | (NF^VF))=1),PC + 1 →PC	-	-	-	-	1	1/2*	S0	1101	0001													Branch enable/disable	82
	LGE	IF ((NF ^ VF)=0), LAR - 4 →PC IF ((NF ^ VF)=1),PC + 1 →PC	-	-	-	-	1	1/2*	S0	1101	0010													Branch enable/disable	82
	LLE	IF ((ZF | (NF ^ VF))=1),LAR - 4 →PC IF ((ZF | (NF ^ VF))=0),PC + 1 →PC	-	-	-	-	1	1/2*	S0	1101	0011													Branch enable/disable	82
	LLT	IF ((NF ^ VF)=1),LAR - 4 →PC IF ((NF ^ VF)=0),PC + 1 →PC	-	-	-	-	1	1/2*	S0	1101	0000													Branch enable/disable	82
	LHI	IF ((CF | ZF)=0), LAR - 4 →PC IF ((CF | ZF)=1),PC + 1 →PC	-	-	-	-	1	1/2*	S0	1101	0101													Branch enable/disable	82
	LCC	IF (CF = 0), LAR - 4 →PC IF (CF = 1),PC + 1 → PC	-	-	-	-	1	1/2*	S0	1101	0110													Branch enable/disable	82
	LLS	IF ((CF | ZF)=1), LAR - 4 →PC IF ((CF | ZF)=0),PC + 1 →PC	-	-	-	-	1	1/2*	S0	1101	0111													Branch enable/disable	82
	LCS	IF (CF = 1), LAR - 4 →PC IF (CF = 0),PC + 1 →PC	-	-	-	-	1	1/2*	S0	1101	1001													Branch enable/disable	82
	LRA	LAR - 4 →PC	-	-	-	-	1	1	S0	1101	1010														82
SETLB	SETLB	mem32 (PC + 1) →LIR , PC + 5 →LAR	-	-	-	-	1	1	S0	1101	1011														83
JMP	JMP (An)	An → PC	-	-	-	-	2	3	D0	1111	0000	1111	01An												84
	JMP label	IF (label = (d16,PC)),PC + d16(sign_ext) →PC	-	-	-	-	3	2	S2	1100	1100	<d16.	....	....	....>										84
		IF (label = (d32,PC)),PC + d32 →PC	-	-	-	-	5	3**	S4	1101	1100	<d32	....	....	....	....	....	....	....>					**4 cycyes for AM30	84
CALL	CALL label	PC + 5 →mem32(SP), SP - imm8(zero_ext) →SP,PC + 5 →MDR, PC + d16(sign_ext) →PC	-	-	-	-	5	2	S4	1100	1101	<d16.	....	....	....>	<regs	....>	<imm8	....>					If label = (d16,PC), registers specified with regs = 0	85
		PC + 5 →mem32(SP),reg1 →mem32(SP-4), SP - imm8(zero_ext) →SP,PC + 5 →MDR, PC + d16(sign_ext) →PC	-	-	-	-	5	3																If label = (d16,PC), registers specified with regs = 1	85
		PC + 5 →mem32(SP),reg1→mem32(SP-4), reg2→mem32(SP),SP - imm8(zero_ext) →SP, PC + 5 →MDR,PC + d16(sign_ext) →PC	-	-	-	-	5	4																If label = (d16,PC), registers specified with regs = 2	85
		PC + 5 →mem32(SP),reg1→mem32(SP-4), reg2→mem32(SP-8),reg3→mem32(SP), SP - imm8(zero_ext) →SP,PC + 5 →MDR, PC + d16(sign_ext) →PC	-	-	-	-	5	5																If label = (d16,PC), registers specified with regs = 3	85
		PC + 5 →mem32(SP),D2→mem32(SP-4), D3→mem32(SP-8),A2→mem32(SP-12), A3→mem32(SP-16), SP - imm8(zero_ext) →SP,PC + 5 →MDR, PC + d16(sign_ext) →PC	-	-	-	-	5	6																If label = (d16,PC), registers specified with regs = 4	85

*: Depends on the status of Instruction queue.

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag				Code Size	Cycle	For -mat	Machine Code														Notes	Page
			VF	CF	NF	ZF				1	2	3	4	5	6	7	8	9	10	11	12	13	14		
CALL	CALL label	PC + 5 → mem32(SP),D0 → mem32(SP-4), D1→ mem32(SP-8),A0→ mem32(SP-12), A1→ mem32(SP-16),MDR→ mem32(SP-20), LIR→ mem32(SP-24),LAR→ mem32(SP-28), SP - imm8(zero_ext) → SP,PC + 5 → MDR, PC + d16(sign_ext) → PC	-	-	-	-	5	9	S4	1100	1101	<d16.	....	....	....>	<regs	....>	<imm8	....>					If label = (d16,PC), registers specified with regs = 7	85
		PC + 5 → mem32(SP),reg1→ mem32(SP-4), D0→ mem32(SP-8),D1→ mem32(SP-12), A0→ mem32(SP-16),A1→ mem32(SP-20), MDR→ mem32(SP-24),LIR→ mem32(SP-28), LAR→ mem32(SP-32), SP - imm8(zero_ext) → SP,PC + 5 → MDR, PC + d16(sign_ext) → PC	-	-	-	-	5	10																If label = (d16,PC), registers specified with regs = 8	85
		PC + 5 → mem32,(SP)reg1→ mem32(SP-4), reg2 → mem32(SP-8),D0→ mem32(SP-12), D1→ mem32(SP-16),A0→ mem32(SP-20), A1→ mem32(SP-24),MDR→ mem32(SP-28), LIR→ mem32(SP-32),LAR→ mem32(SP-36), SP - imm8(zero_ext) → SP,PC + 5 → MDR, PC + d16(sign_ext) → PC	-	-	-	-	5	11																If label = (d16,PC), registers specified with regs = 9	85
		PC + 5 → mem32(SP),reg1→ mem32(SP-4), reg2→ mem32(SP-8),reg3→ mem32(SP-12), D0→ mem32(SP-16),D1→ mem32(SP-20), A0→ mem32(SP-24),A1→ mem32(SP-28), MDR→ mem32(SP-32),LIR→ mem32(SP-36), LAR→ mem32(SP-40), SP - imm8(zero_ext) → SP,PC + 5 → MDR, PC + d16(sign_ext) → PC	-	-	-	-	5	12																If label = (d16,PC), registers specifiied with regs = 10	85
		PC + 5 →mem32(SP),D2→ mem32(SP-4), D3→ mem32(SP-8),A2→ mem32(SP-12), A3→ mem32(SP-16),D0→ mem32(SP-20), D1→ mem32(SP-24),A0→ mem32(SP-28), A1→ mem32(SP-32),MDR→ mem32(SP-36), LIR→ mem32(SP-40),LAR→ mem32(SP-44) SP - imm8(zero_ext) → SP,PC + 5 → MDR, PC+d16(sign_ext) → PC,	-	-	-	-	5	13																If label = (d16,PC), registers specified with regs = 11	85
		PC+7 → mem32(SP), SP - imm8(zero_ext) → SP,PC + 7 → MDR, PC + d32 → PC	-	-	-	-	7	4*	S6	1101	1101	<d32	....	....	....	....	....	....	....>	<regs	....>	<imm8	....>	If label = (d32,PC), registers specified with regs = 0 *: 5 cycyles for AM30	85
		PC + 7 → mem32(SP),reg1 → mem32(SP-4), SP - imm8(zero_ext) → SP,PC + 7 → MDR, PC + d32 → PC	-	-	-	-	7	4*																If label = (d32,PC), register specified with regs = 1 *: 5 cycles for AM30	85

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
CALL	CALL label	PC + 7 → mem32(SP),reg1→ mem32(SP-4), reg2→ mem32(SP),SP - imm8(zero_ext) → SP, PC + 7 → MDR,PC + d32 → PC	-	-	-	-	7	5*	S6	1101	1101	<d32	....	....	....	....	....	....	....>	<regs	....>	<imm8	....>	If label = (d32,PC), registers specified with regs = 2 *: 6 cycyles for AM30	85
		PC + 7 → mem32(SP), reg1→ mem32(SP-4),reg2→ mem32(SP-8), reg3→ mem32(SP),SP - imm8(zero_ext) → SP, PC + 7 → MDR,PC + d32 → PC	-	-	-	-	7	6*																If label = (d32,PC), registers specified wtih regs = 3 *: 7 cycles for AM30	85
		PC + 7 → mem32(SP),D2→ mem32(SP-4), D3→ mem32(SP-8),A2→ mem32(SP-12), A3→ mem32(SP-16), SP - imm8(zero_ext) → SP,PC + 7 → MDR, PC + d32 → PC	-	-	-	-	7	7*																If label = (d32,PC), registers specified with regs = 4 *: 8 cycles for AM30	85
		PC + 7 → mem32(SP),D0→ mem32(SP-4), D1→ mem32(SP-8),A0→ mem32(SP-12), A1→ mem32(SP-16),MDR→ mem32(SP-20), LIR→ mem32(SP-24),LAR→ mem32(SP-28), SP - imm8(zero_ext) → SP,PC + 7 → MDR, PC + d32 → PC	-	-	-	-	7	10*																If label = (d32,PC), registers specified with regs = 7 *: 11 cycles for AM30	85
		PC + 7 → mem32(SP),reg1→ mem32(SP-4), D0→ mem32(SP-8),D1→ mem32(SP-12), A0→ mem32(SP-16),A1→ mem32(SP-20), MDR→ mem32(SP-24),LIR→ mem32(SP-28), LAR→ mem32(SP-32), SP - imm8(zero_ext) → SP,PC + 7 → MDR, PC + d32 → PC	-	-	-	-	7	11*																If label = (d32,PC), registers specified with regs = 8 *: 12 cycles for AM30	85
		PC + 7 → mem32(SP),reg1→ mem32(SP-4), reg2 → mem32(SP-8),D0→ mem32(SP-12), D1→ mem32(SP-16),A0→ mem32(SP-20), A1→ mem32(SP-24),MDR→ mem32(SP-28), LIR→ mem32(SP-32),LAR→ mem32(SP-36), SP - imm8(zero_ext) → SP,PC + 7 → MDR, PC + d32 → PC	-	-	-	-	7	12*																If label = (d32,PC), registers specified with regs = 9 *: 13 cycles for AM30	85
		PC + 7 → mem32(SP),reg1→ mem32(SP-4), reg2→ mem32(SP-8),reg3→ mem32(SP-12), D0→ mem32(SP-16),D1→ mem32(SP-20), A0→ mem32(SP-24),A1→ mem32(SP-28), MDR→ mem32(SP-32),LIR→ mem32(SP-36), LAR→ mem32(SP-40), SP - imm8(zero_ext) → SP,PC + 7 → MDR, PC + d32 → PC	-	-	-	-	7	13*																If label = (d32,PC), registers specified with regs =10 * 14 cycles for AM30	85

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag				Code Size	Cycle	For -mat	Machine Code														Notes	Page
			VF	CF	NF	ZF				1	2	3	4	5	6	7	8	9	10	11	12	13	14		
CALL	CALL label	PC + 7 → mem32(SP),D2→ mem32(SP-4), D3→ mem32(SP-8),A2→ mem32(SP-12), A3→ mem32(SP-16),D0→ mem32(SP-20), D1→ mem32(SP-24),A0→ mem32(SP-28), A1→ mem32(SP-32),MDR→ mem32(SP-36), LIR→ mem32(SP-40),LAR→ mem32(SP-44) SP - imm8(zero_ext) → SP,PC + 7 → MDR, PC + d32 → PC	-	-	-	-	7	14*	S6	1101	1101	<d32	....	....	....	....	....	....	....>	<regs	....>	<imm8 ....>		If label = (d32,PC), registersspecified with regs =11 *: 5 cycles for AM30	85
CALLS	CALLS (An)	PC + 2 → mem32(SP),PC + 2 → MDR, An → PC	-	-	-	-	2	3	D0	1111	0000	1111	00An												87
	CALLS label	IF (label = (d16,PC)), PC + 4 → mem32(SP), PC + 4 → MDR, PC +d16 (sign_ext) → PC	-	-	-	-	4	3	D2	1111	1010	1111	1111	<d16	....	....	....>							*: 4 cycles for AM30	88
		IF ((label = (d32,PC)), PC + 6 → mem32(SP), PC + 6 → MDR, PC + d32 → PC	-	-	-	-	6	3*	D4	1111	1100	1111	1111	<d32	....	....	....	....	....	....	....>			registers specified with regs =0 *:4 cycles for AM30	88
RET	RET	SP + imm8(zero_ext) → SP,mem32(SP) → PC	-	-	-	-	3	5*	S2	1101	1111	<regs	....>	<imm8 ....>										registers specified with regs =1 *:4 cycles for AM30	89
		SP + imm8(zero_ext) → SP, mem32(SP-4)→ reg, mem32(SP) → PC	-	-	-	-	3	5*																registers specified with regs =2 *:4 cycles for AM30	89
		SP + imm8(zero_ext) → SP, mem32(SP-4)→ reg1,mem32(SP-8)→ reg2, mem32(SP) → PC	-	-	-	-	3	5*																registers specified with regs =3 *:4 cycles for AM30	89
		SP + imm8(zero_ext) → SP, mem32(SP-4)→ reg1,mem32(SP-8)→ reg2, mem32(SP-12)→ reg3,mem32(SP) → PC	-	-	-	-	3	5*																registers specified with regs =4	89
		SP + imm8(zero_ext) → SP, mem32(SP-4)→ D2,mem32(SP-8)→ D3, mem32(SP-12)→ A2,mem32(SP-16)→ A3, mem32(SP) → PC	-	-	-	-	3	5																registers specified with regs =7	89
		SP + imm8(zero_ext) → SP, mem32(SP-4)→ D0,mem32(SP-8)→ D1, mem32(SP-12)→ A0,mem32(SP-16)→ A1, mem32(SP-20)→ MDR,mem32(SP-24)→ LIR, mem32(SP-28)→ LAR,mem32(SP) → PC	-	-	-	-	3	8																registers specified with regs =8	89
		SP + imm8(zero_ext) → SP, mem32(SP-4)→ reg1,mem32(SP-8)→ D0, mem32(SP-12)→ D1,mem32(SP-16)→ A0, mem32(SP-20)→ A1,mem32(SP-24)→ MDR, mem32(SP-28)→ LIR,mem32(SP-32)→ LAR, mem32(SP) → PC	-	-	-	-	3	9																	89

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For -mat	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
RET	RET	SP + imm8(zero_ext) → SP, mem32(SP-4)→ reg1,mem32(SP-8)→ reg2, mem32(SP-12)→ D0,mem32(SP-16)→ D1, mem32(SP-20)→ A0,mem32(SP-24)→ A1, mem32(SP-28)→ MDR,mem32(SP-32)→ LIR, mem32(SP-36)→ LAR,mem32(SP) → PC	-	-	-	-	3	10	S2	1101	1111	<regs ....>		<imm8 ....>										registers specified with regs = 9	89
		SP + imm8(zero_ext) → SP, mem32(SP-4)→ reg1,mem32(SP-8)→ reg2, mem32(SP-12)→ reg3,mem32(SP-16)→ D0, mem32(SP-20)→ D1,mem32(SP-24)→ A0, mem32(SP-28)→ A1,mem32(SP-32)→ MDR, mem32(SP-36)→ LIR,mem32(SP-40)→ LAR, mem32(SP) → PC	-	-	-	-	3	11																registers specified with regs= 10	89
		SP + imm8(zero_ext) → SP, mem32(SP-4)→ D2,mem32(SP-8)→ D3, mem32(SP-12)→ A2,mem32(SP-16)→ A3, mem32(SP-20)→ D0,mem32(SP-24)→ D1, mem32(SP-28)→ A0,mem32(SP-32)→ A1, mem32(SP-36)→ MDR,mem32(SP-40)→ LIR, mem32(SP-44)→ LAR,mem32(SP) → PC	-	-	-	-	3	12																registers specified with regs= 11	89
RETF	RETF	SP + imm8(zero_ext) → SP,MDR → PC,	-	-	-	-	3	2	S2	1101	1110	<regs ....>		<imm8 ....>										register specified with regs = 0	90
		SP + imm8(zero_ext) → SP,MDR → PC, mem32(SP-4)→ reg	-	-	-	-	3	2																register specified with regs= 1	90
		SP + imm8(zero_ext) → SP,MDR → PC, mem32(SP-4)→ reg1,mem32(SP-8)→ reg2,	-	-	-	-	3	3																registers specified with regs = 2	90
		SP + imm8(zero_ext) → SP,MDR → PC, mem32(SP-4)→ reg1,mem32(SP-8)→ reg2, mem32(SP-12)→ reg3	-	-	-	-	3	4																registers specified with regs= 3	90
		SP + imm8(zero_ext) → SP,MDR → PC, mem32(SP-4)→ D2,mem32(SP-8)→ D3, mem32(SP-12)→ A2,mem32(SP-16)→ A3,	-	-	-	-	3	5																registers specified with regs= 4	90
		SP + imm8(zero_ext) → SP,MDR → PC, mem32(SP-4)→ D0,mem32(SP-8)→ D1, mem32(SP-12)→ A0,mem32(SP-16)→ A1, mem32(SP-20)→ MDR,mem32(SP-24)→ LIR, mem32(SP-28)→ LAR	-	-	-	-	3	8																registers specified with regs = 7	90
		SP + imm8(zero_ext) → SP,MDR → PC, mem32(SP-4)→ reg1,mem32(SP-8)→ D0, mem32(SP-12)→ D1,mem32(SP-16)→ → A0, mem32(SP-20)→ A1,mem32(SP-24)→ MDR, mem32(SP-28)→ LIR,mem32(SP-32)→ LAR,	-	-	-	-	3	9																registers specified with regs = 8	90

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	Machine Code (1–14)	Notes	Page
RETF	RETF	SP + imm8(zero_ext) → SP,MDR → PC, mem32(SP-4)→ reg1,mem32(SP-8)→ reg2, mem32(SP-12)→ D0,mem32(SP-16)→ D1, mem32(SP-20)→ A0,mem32(SP-24)→ A1, mem32(SP-28)→ MDR,mem32(SP-32)→ LIR, mem32(SP-36)→ LAR	-	-	-	-	3	10	S2	1101 1110 <regs ....> <imm8 ....>	registers specified with regs = 9	90
		SP + imm8(zero_ext) → SP,MDR → PC, mem32(SP-4)→ reg1,mem32(SP-8)→ reg2, mem32(SP-12)→ reg3,mem32(SP-16)→ D0, mem32(SP-20)→ D1,mem32(SP-24)→ A0, mem32(SP-28)→ A1,mem32(SP-32)→ MDR, mem32(SP-36)→ LIR,mem32(SP-40)→ LAR,	-	-	-	-	3	11			registers specified with regs= 10	90
		SP + imm8(zero_ext) → SP,MDR → PC, mem32(SP-4)→ D2,mem32(SP-8)→ D3, mem32(SP-12)→ A2,mem32(SP-16)→ A3, mem32(SP-20)→ D0,mem32(SP-24)→ D1, mem32(SP-28)→ A0,mem32(SP-32)→ A1, mem32(SP-36)→ MDR,mem32(SP-40)→ LIR, mem32(SP-44)→ LAR	-	-	-	-	3	10			registers specified with regs= 11	90
RETS	RETS	mem32(SP) → PC	-	-	-	-	2	5*	D0	1111 0000 1111 1100	*: 4 cycles for AM30	91
JSR	JSR (An)	SP - 4 → SP,PC + 2 → mem32(SP) PC + 2 → MDR,An → PC, (execute subroutine) SP + 4 → SP	●	●	●	●	8	5				92
	JSR label	IF (label = (d16,PC)), SP - 4 → SP,PC + 4 → mem32(SP), PC + 4 → MDR,PC + d16 (sign_ext) → PC (execute subroutine) SP+4 → SP	●	●	●	●	10	5				93
		IF (label = (d32,PC)), SP - 4 → SP,PC + 6 → (SP+3), PC + 6 → MDR,PC + d32 → PC (execute subroutine) SP+4 → SP	●	●	●	●	12	5*			*: 6 cycles for AM30	93
RTS	RTS	mem32(SP) → PC	-	-	-	-	2	4	D0	1111 0000 1111 1101		94
RTI	RTI	mem16(SP) → PSW,mem32(SP+4) → PC, SP + 8 → SP	●	●	●	●	2	4				95
TRAP	TRAP	PC + 2 → mem32(SP),0x40000010 → PC	-	-	-	-	2	4	D0	1111 0000 1111 1110		96
NOP	NOP	PC + 1 → PC	-	-	-	-	1	1	S0	1100 1011		97

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag				Code Size	Cycle	For -mat	Machine Code														Notes	Page
			VF	CF	NF	ZF				1	2	3	4	5	6	7	8	9	10	11	12	13	14		
Extension Instructions																									
UDF	UDF00 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	0000	DmDn												98
	UDF01 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	0001	DmDn												98
	UDF02 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	0010	DmDn												98
	UDF03 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	0011	DmDn												98
	UDF04 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	0100	DmDn												98
	UDF05 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	0101	DmDn												98
	UDF06 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	0110	DmDn												98
	UDF07 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	0111	DmDn												98
	UDF08 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	1000	DmDn												98
	UDF09 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	1001	DmDn												98
	UDF10 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	1010	DmDn												98
	UDF11 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	1011	DmDn												98
	UDF12 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	1100	DmDn												98
	UDF13 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	1101	DmDn												98
	UDF14 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	1110	DmDn												98
	UDF15 Dm,Dn	Dm op Dn → Dn	*	*	*	*	2	*	D0	1111	0110	1111	DmDn												98
	UDF20 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	0000	DmDn												98
	UDF21 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	0001	DmDn												98
	UDF22 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	0010	DmDn												98
	UDF23 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	0011	DmDn												98
	UDF24 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	0100	DmDn												98
	UDF25 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	0101	DmDn												98
	UDF26 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	0110	DmDn												98
	UDF27 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	0111	DmDn												98
	UDF28 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	1000	DmDn												98
	UDF29 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	1001	DmDn												98
	UDF30 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	1010	DmDn												98
	UDF31 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	1011	DmDn												98
	UDF32 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	1100	DmDn												98
	UDF33 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	1101	DmDn												98
	UDF34 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	1110	DmDn												98
	UDF35 Dm,Dn	Dm op Dn	-	-	-	-	2	*	D0	1111	0101	1111	DmDn												98
	UDF00 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	0000	00Dn	<imm8 ....>											99
	UDF01 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	0001	00Dn	<imm8 ....>											99
	UDF02 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	0010	00Dn	<imm8 ....>											99
	UDF03 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	0011	00Dn	<imm8 ....>											99
	UDF04 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	0100	00Dn	<imm8 ....>											99
	UDF05 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	0101	00Dn	<imm8 ....>											99
	UDF06 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	0110	00Dn	<imm8 ....>											99

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	CF	NF	ZF	Code Size	Cycle	For-mat	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
UDFnn	UDF07 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	0111	00Dn	<imm8 ....>											99
	UDF08 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	1000	00Dn	<imm8 ....>											99
	UDF09 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	1001	00Dn	<imm8 ....>											99
	UDF10 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	1010	00Dn	<imm8 ....>											99
	UDF11 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	1011	00Dn	<imm8 ....>											99
	UDF12 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	1100	00Dn	<imm8 ....>											99
	UDF13 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	1101	00Dn	<imm8 ....>											99
	UDF14 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	1110	00Dn	<imm8 ....>											99
	UDF15 imm8,Dn	imm8(sign_ext) op Dn → Dn	*	*	*	*	3	*	D1	1111	1001	1111	00Dn	<imm8 ....>											99
	UDF20 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0000	10Dn	<imm8 ....>										Not used for AM30	99
	UDF21 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0001	10Dn	<imm8 ....>										Not used for AM30	99
	UDF22 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0010	10Dn	<imm8 ....>										Not used for AM30	99
	UDF23 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0011	10Dn	<imm8 ....>										Not used for AM30	99
	UDF24 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0100	10Dn	<imm8 ....>										Not used for AM30	99
	UDF25 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0101	10Dn	<imm8 ....>										Not used for AM30	99
	UDF26 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0110	10Dn	<imm8 ....>										Not used for AM30	99
	UDF27 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0111	10Dn	<imm8 ....>										Not used for AM30	99
	UDF28 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1000	10Dn	<imm8 ....>										Not used for AM30	99
	UDF29 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1001	10Dn	<imm8 ....>										Not used for AM30	99
	UDF30 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1010	10Dn	<imm8 ....>										Not used for AM30	99
	UDF31 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1011	10Dn	<imm8 ....>										Not used for AM30	99
	UDF32 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1100	10Dn	<imm8 ....>										Not used for AM30	99
	UDF33 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1101	10Dn	<imm8 ....>										Not used for AM30	99
	UDF34 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1110	10Dn	<imm8 ....>										Not used for AM30	99
	UDF35 imm8,Dn	imm8(sign_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1111	10Dn	<imm8 ....>										Not used for AM30	99
	UDF00 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0000	00Dn	<imm16...		....	....>								99
	UDF01 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0001	00Dn	<imm16...		....	....>								99
	UDF02 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0010	00Dn	<imm16...		....	....>								99
	UDF03 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0011	00Dn	<imm16...		....	....>								99
	UDF04 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0100	00Dn	<imm16...		....	....>								99
	UDF05 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0101	00Dn	<imm16...		....	....>								99
	UDF06 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0110	00Dn	<imm16...		....	....>								99
	UDF07 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0111	00Dn	<imm16...		....	....>								99
	UDF08 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1000	00Dn	<imm16...		....	....>								99
	UDF09 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1001	00Dn	<imm16...		....	....>								99
	UDF10 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1010	00Dn	<imm16...		....	....>								99
	UDF11 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1011	00Dn	<imm16...		....	....>								99
	UDF12 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1100	00Dn	<imm16...		....	....>								99
	UDF13 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1101	00Dn	<imm16...		....	....>								99
	UDF14 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1110	00Dn	<imm16...		....	....>								99
	UDF15 imm16,Dn	imm16(sign_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1111	00Dn	<imm16...		....	....>								99

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag				Code	Cycle	For	Machine Code														Notes	Page
			VF	CF	NF	ZF	Size		-mat	1	2	3	4	5	6	7	8	9	10	11	12	13	14		
UDFnn	UDF20 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0000	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF21 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0001	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF22 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0010	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF23 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0011	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF24 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0100	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF25 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0101	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF26 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0110	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF27 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0111	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF28 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1000	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF29 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1001	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF30 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1010	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF31 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1011	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF32 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1100	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF33 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1101	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF34 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1110	10Dn	<imm16...		....	....>							Not used for AM30	99
	UDF35 imm16,Dn	imm16(sign_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1111	10Dn	<imm16...		....	....>			....				Not used for AM30	99
	UDF00 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0000	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF01 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0001	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF02 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0010	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF03 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0011	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF04 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0100	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF05 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0101	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF06 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0110	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF07 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0111	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF08 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1000	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF09 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1001	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF10 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1010	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF11 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1011	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF12 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1100	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF13 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1101	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF14 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1110	00Dn	<imm32...		....	....	....	....	...	....>				99
	UDF15 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1111	00Dn	<imm32...		....	....	....	....	....	....>				99
	UDF20 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0000	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF21 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0001	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF22 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0010	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF23 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0011	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF24 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0100	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF25 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0101	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF26 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0110	10Dn	<imm32...		....	....	....	....	.....	....>			Not used for AM30	99
	UDF27 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0111	10Dn	<imm32...		....	....	....	....		....>			Not used for AM30	99

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
																....									
UDFnn	UDF28 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1000	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF29 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1001	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF30 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1010	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF31 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1011	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF32 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1100	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF33 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1101	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF34 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1110	10Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	99
	UDF35 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1111	10Dn	<imm32...			....	....	....	....	....>			Not used for AM30	99
UDFUnn	UDFU00 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	0000	01Dn	<imm8 ....>											100
	UDFU01 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	0001	01Dn	<imm8 ....>											100
	UDFU02 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	0010	01Dn	<imm8 ....>											100
	UDFU03 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	0011	01Dn	<imm8 ....>											100
	UDFU04 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	0100	01Dn	<imm8 ....>											100
	UDFU05 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	0101	01Dn	<imm8 ....>											100
	UDFU06 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	0110	01Dn	<imm8 ....>											100
	UDFU07 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	0111	01Dn	<imm8 ....>											100
	UDFU08 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	1000	01Dn	<imm8 ....>											100
	UDFU09 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	1001	01Dn	<imm8 ....>											100
	UDFU10 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	1010	01Dn	<imm8 ....>											100
	UDFU11 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	1011	01Dn	<imm8 ....>											100
	UDFU12 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	1100	01Dn	<imm8 ....>											100
	UDFU13 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	1101	01Dn	<imm8 ....>											100
	UDFU14 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	1110	01Dn	<imm8 ....>											100
	UDFU15 imm8,Dn	imm8(zero_ext) op Dn →Dn	*	*	*	*	3	*	D1	1111	1001	1111	01Dn	<imm8 ....>											100
	UDFU20 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0000	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU21 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0001	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU22 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0010	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU23 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0011	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU24 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0100	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU25 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0101	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU26 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0110	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU27 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	0111	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU28 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1000	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU29 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1001	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU30 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1010	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU31 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1011	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU32 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1100	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU33 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1101	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU34 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1110	11Dn	<imm8 ....>										Not used for AM30	100
	UDFU35 imm8,Dn	imm8(zero_ext) op Dn	-	-	-	-	3	*	D1	1111	1001	1111	11Dn	<imm8 ....>										Not used for AM30	100

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For -mat	Machine Code 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
UDFUnn	UDFU00 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0000	01Dn	<imm16...		....	....>								100
	UDFU01 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0001	01Dn	<imm16...		....	....>								100
	UDFU02 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0010	01Dn	<imm16...		....	....>								100
	UDFU03 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0011	01Dn	<imm16...		....	....>								100
	UDFU04 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0100	01Dn	<imm16...		....	....>								100
	UDFU05 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0101	01Dn	<imm16...		....	....>								100
	UDFU06 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0110	01Dn	<imm16...		....	....>								100
	UDFU07 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	0111	01Dn	<imm16...		....	....>								100
	UDFU08 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1000	01Dn	<imm16...		....	....>								100
	UDFU09 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1001	01Dn	<imm16...		....	....>								100
	UDFU10 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1010	01Dn	<imm16...		....	....>								100
	UDFU11 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1011	01Dn	<imm16...		....	....>								100
	UDFU12 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1100	01Dn	<imm16...		....	....>								100
	UDFU13 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1101	01Dn	<imm16...		....	....>								100
	UDFU14 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1110	01Dn	<imm16...		....	....>								100
	UDFU15 imm16,Dn	imm16(zero_ext) op Dn → Dn	*	*	*	*	4	*	D2	1111	1011	1111	01Dn	<imm16...		....	....>								100
	UDFU20 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0000	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU21 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0001	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU22 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0010	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU23 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0011	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU24 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0100	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU25 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0101	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU26 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0110	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU27 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	0111	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU28 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1000	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU29 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1001	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU30 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1010	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU31 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1011	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU32 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1100	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU33 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1101	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU34 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1110	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU35 imm16,Dn	imm16(zero_ext) op Dn	-	-	-	-	4	*	D2	1111	1011	1111	11Dn	<imm16...		....	....>							Not used for AM30	100
	UDFU00 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0000	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU01 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0001	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU02 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0010	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU03 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0011	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU04 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0100	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU05 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0101	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU06 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0110	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU07 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	0111	01Dn	<imm32...		....	....	....	....	....	....>				100

MN1030/MN103S SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag VF	Flag CF	Flag NF	Flag ZF	Code Size	Cycle	For-mat	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Notes	Page
UDFUnn	UDFU08 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1000	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU09 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1001	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU10 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1010	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU11 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1011	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU12 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1100	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU13 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1101	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU14 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1110	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU15 imm32,Dn	imm32 op Dn → Dn	*	*	*	*	6	*	D4	1111	1101	1111	01Dn	<imm32...		....	....	....	....	....	....>				100
	UDFU20 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0000	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU21 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0001	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU22 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0010	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU23 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0011	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU24 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0100	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU25 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0101	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU26 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0110	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU27 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	0111	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU28 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1000	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU29 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1001	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU30 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1010	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU31 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1011	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU32 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1100	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU33 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1101	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU34 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1110	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100
	UDFU35 imm32,Dn	imm32 op Dn	-	-	-	-	6	*	D4	1111	1101	1111	11Dn	<imm32...		....	....	....	....	....	....>			Not used for AM30	100

MN1030/MN103S SERIES INSTRUCTION SET

INSTRUCTION SET

■ Description

Dn,Dm,Di	data registers
An,Am	address registers
MDR	multiply/divide register
PSW	processor status word
PC	program counter
SP	stack pointer
LIR	loop instruction registers
LAR	loop address registers
imm8,imm16,imm32	immediate value(8, 16 or 32 bits)
d8,d16,d32	displacement(8, 16 or 32 bits)
abs16,abs32	absolute address (16 or 32 bits)
mem8(An)	lower 8-bit data in memory referred by () address
mem16(An)	lower 16bit data in memory referred by ()address
mem32(An)	lower 32-bit data in memory referred by () address
regs	registers
.lsb,.msb	bit location(lowest/highest)
&	logical AND
|	logical OR
^	exclusive OR
~	bit inverted
op	operation defined by users
<<,>>	bit shift(right/left) performsa bit shift for specified value
VF	overflow flags
CF	carry flags
NF	negative flags
ZF	zero flags
temp	temporary registers
→	move
:	reflects operation result
(sign_ext)	sign-extend
(zero_ext)	zero-extend
{MDR,Dn}	64-bit data defined whose upper 32-bit data are in MDR and lower 32-bit in register Dn within "{}".
0x....	hexadecimal(hexadecimal following to 0x.)

■ Flag

l	changes
-	no changes
0	always 0
1	alwayas 1
?	not defined
*	defined by users

■ CodeSize

byte:

■ Cycles

Cycles may be changed the status of the pipline, memory space to access.

Cycles are calculated on those conditions;

(1) no pipeline installation

(2) Instruction queue: 2 cycles

data load/store: 1 cycle

(ROM/RAM/ internal flash:

Instructions: access to internal ROM/RAM space

data: access to internal RAM space

with cache

Instructions/data :access to cachable area and hit the chache)

Please see the LSI manuals for how the pipeline installation affects the cycles.

If using extended instructions, the users define the cycles.

■ Format

Please refer to Chapter 1 Overview

■ IInstructions replaced to other instructions by Assembler

Format or Mchine Codeare not written

usable CodeSize and Cycles are written

MOVB Reg,Mem , MOVH Reg,Mem,

ASR Dn , LSR Dn , RTS

■ Instructions replaced to multiple instructions by Assembler

Format or Mchine Code are not written

usable CodeSize and Cycles are written

MOVB Mem,Reg , MOVH Reg,Mem ,

JSR (An) , JSR label

1st byte

Upper/Lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	CLR D0	MOV D0,(abs16)	MOVBU D0,(abs16)	MOVHU D0,(abs16)	CLR D1	MOV D1,(abs16)	MOVBU D1,(abs16)	MOVHU D1,(abs16)	CLR D2	MOV D2,(abs16)	MOVBU D2,(abs16)	MOVHU D2,(abs16)	CLR D3	MOV D3,(abs16)	MOVBU D3,(abs16)	MOVHU D3,(abs16)
1	EXTB Dn				EXTBU Dn				EXTH Dn				EXTHU Dn			
2	ADD imm8,An				MOV imm16,An				ADD imm8,Dn				MOV imm16,Dn			
3	MOV (abs16),Dn				MOVBU (abs16),Dn				MOVHU (abs16),Dn				MOV SP,An			
4	INC D0	INC A0	MOV D0,(d8,SP)	MOV A0,(d8,SP)	INC D1	INC A1	MOV D1,(d8,SP)	MOV A1,(d8,SP)	INC D2	INC A2	MOV D2,(d8,SP)	MOV A2,(d8,SP)	INC D3	INC A3	MOV D3,(d8,SP)	MOV A3,(d8,SP)
5	INC4 An				ASL2 Dn				MOV (d8,SP),Dn				MOV (d8,SP),An			
6	MOV Dm,(An)															
7	MOV (Am),Dn															
8	MOV Dm,Dn (If m=nMOV, imm8,Dn)															
9	MOV Am,An (If m=nMOV, imm8,An)															
A	CMP Dm,Dn (If m=n, CMP imm8,Dn)															
B	CMP Am,An (If m=n, CMP imm8,An)															
C	BLT (d8,PC)	BGT (d8,PC)	BGE (d8,PC)	BLE (d8,PC)	BCS (d8,PC)	BHI (d8,PC)	BCC (d8,PC)	BLS (d8,PC)	BEQ (d8,PC)	BNE (d8,PC)	BRA (d8,PC)	NOP	JMP (d16,PC)	CALL (d16,PC)	MOVM (SP),regs	MOVM regs,(SP)
D	LLT	LGT	LGE	LLE	LCS	LHI	LCC	LLS	LEQ	LNE	LRA	SETLB	JMP (d32,PC)	CALL (d32,PC)	RETF	RET
E	ADD Dm,Dn															
F	Code extension (2-byte)								Code extension (3-byte)		Code extension		Code extension (6-byte)		Code extension (7-byte)	

2nd byte (1st byte:F0) Instruction for 2-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	MOV (Am),An															
1	MOV Am,(An)															
2																
3																
4	MOVBU (Am),Dn															
5	MOVBU Dm,(An)															
6	MOVHU (Am),Dn															
7	MOVHU Dm,(An)															
8	BSET Dm,(An)															
9	BCLR Dm,(An)															
A																
B																
C																
D																
E																
F	CALLS (An)				JMP (An)								RETS	RTI	TRAP	

2nd byte (1st byte: F1)) Instruction for 2-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	SUB Dm,Dn															
1	SUB Am,Dn															
2	SUB Dm,An															
3	SUB Am,An															
4	ADDC Dm,Dn															
5	ADD Am,Dn															
6	ADD Dm,An															
7	ADD Am,An															
8	SUBC Dm,Dn															
9	CMP Am,Dn															
A	CMP Dm,An															
B																
C																
D	MOV Am,Dn															
E	MOV Dm,An															
F																

2nd byte(1st byte:F2) Instruction for 2-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	AND Dm,Dn															
1	OR Dm,Dn															
2	XOR Dm,Dn															
3	NOT Dn															
4	MUL Dm,Dn															
5	MULU Dm,Dn															
6	DIV Dm,Dn															
7	DIVU Dm,Dn															
8	ROL Dn				ROR Dn											
9	ASL Dm,Dn															
A	LSR Dm,Dn															
B	ASR Dm,Dn															
C																
D	EXT Dn															
E	MOV MDR,Dn				MOV PSW,Dn											
F	MOV A0,SP		MOV D0,MDR	MOV D0,PSW	MOV A1,SP		MOV D1,MDR	MOV D1,PSW	MOV A2,SP		MOV D2,MDR	MOV D2,PSW	MOV A3,SP		MOV D3,MDR	MOV D3,PSW

2nd byte (1st byte: F3) Instructions for 2-byte

<table>
<tr><th>Upper/lower</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr>
<tr><td>0</td><td rowspan="4" colspan="16">MOV (Di,Am),Dn</td></tr>
<tr><td>1</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td><td rowspan="4" colspan="16">MOV Dm,(Di,An)</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td><td rowspan="4" colspan="16">MOV (Di,Am),An</td></tr>
<tr><td>9</td></tr>
<tr><td>A</td></tr>
<tr><td>B</td></tr>
<tr><td>C</td><td rowspan="4" colspan="16">MOV Am,(Di,An)</td></tr>
<tr><td>D</td></tr>
<tr><td>E</td></tr>
<tr><td>F</td></tr>
</table>

2nd byte(1st byte: F4) Instruction for 2-byte)

<table>
<tr><th>Upper/lower</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr>
<tr><td>0</td><td rowspan="4" colspan="16">MOVBU (Di,Am),Dn</td></tr>
<tr><td>1</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td><td rowspan="4" colspan="16">MOVBU Dm,(Di,An)</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td><td rowspan="4" colspan="16">MOVHU (Di,Am),Dn</td></tr>
<tr><td>9</td></tr>
<tr><td>A</td></tr>
<tr><td>B</td></tr>
<tr><td>C</td><td rowspan="4" colspan="16">MOVHU Dm,(Di,An)</td></tr>
<tr><td>D</td></tr>
<tr><td>E</td></tr>
<tr><td>F</td></tr>
</table>

2nd byte(1st byte:F5) Instruction for 2-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	UDF20 Dm,Dn															
1	UDF21 Dm,Dn															
2	UDF22 Dm,Dn															
3	UDF23 Dm,Dn															
4	UDF24 Dm,Dn															
5	UDF25 Dm,Dn															
6	UDF26 Dm,Dn															
7	UDF27 Dm,Dn															
8	UDF28 Dm,Dn															
9	UDF29 Dm,Dn															
A	UDF30 Dm,Dn															
B	UDF31 Dm,Dn															
C	UDF32 Dm,Dn															
D	UDF33 Dm,Dn															
E	UDF34 Dm,Dn															
F	UDF35 Dm,Dn															

2nd byte (1st byte: F6) Instruction for 2-byte)

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	UDF00 Dm,Dn															
1	UDF01 Dm,Dn															
2	UDF02 Dm,Dn															
3	UDF03 Dm,Dn															
4	UDF04 Dm,Dn															
5	UDF05 Dm,Dn															
6	UDF06 Dm,Dn															
7	UDF07 Dm,Dn															
8	UDF08 Dm,Dn															
9	UDF09 Dm,Dn															
A	UDF10 Dm,Dn															
B	UDF11 Dm,Dn															
C	UDF12 Dm,Dn															
D	UDF13 Dm,Dn															
E	UDF14 Dm,Dn															
F	UDF15 Dm,Dn															

2nd byte (1st byte:F8) Instruction for 3-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	MOV (d8,Am),Dn															
1	MOV Dm,(d8,An)															
2	MOV (d8,Am),An															
3	MOV Am,(d8,An)															
4	MOVBU (d8,Am),Dn															
5	MOVBU Dm,(d8,An)															
6	MOVHU (d8,Am),Dn															
7	MOVHU Dm,(d8,An)															
8																
9			MOVBU D0,(d8,SP)	MOVHU D0,(d8,SP)			MOVBU D1,(d8,SP)	MOVHU D1,(d8,SP)			MOVBU D2,(d8,SP)	MOVHU D2,(d8,SP)			MOVBU D3,(d8,SP)	MOVHU D3,(d8,SP)
A																
B									MOVBU (d8,SP),Dn				MOVHU (d8,SP),Dn			
C	ASL imm8,Dn				LSR imm8,Dn				ASR imm8,Dn							
D																
E	AND imm8,Dn				OR imm8,Dn				BVC (d8,PC)	BVS (d8,PC)	BNC (d8,PC)	BNS (d8,PC)	BTST imm8,Dn			
F	MOV (d8,An),SP				MOV SP,(d8,An)										ADD imm8,SP	

2nd byte (1st byte:F9) Instruction for 3-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	UDF00 imm8,Dn				UDFU00 imm8,Dn				UDF20 imm8,Dn*				UDFU20 imm8,Dn*			
1	UDF01 imm8,Dn				UDFU01 imm8,Dn				UDF21 imm8,Dn*				UDFU21 imm8,Dn*			
2	UDF02 imm8,Dn				UDFU02 imm8,Dn				UDF22 imm8,Dn*				UDFU22 imm8,Dn*			
3	UDF03 imm8,Dn				UDFU03 imm8,Dn				UDF23 imm8,Dn*				UDFU23 imm8,Dn*			
4	UDF04 imm8,Dn				UDFU04 imm8,Dn				UDF24 imm8,Dn*				UDFU24 imm8,Dn*			
5	UDF05 imm8,Dn				UDFU05 imm8,Dn				UDF25 imm8,Dn*				UDFU25 imm8,Dn*			
6	UDF06 imm8,Dn				UDFU06 imm8,Dn				UDF26 imm8,Dn*				UDFU26 imm8,Dn*			
7	UDF07 imm8,Dn				UDFU07 imm8,Dn				UDF27 imm8,Dn*				UDFU27 imm8,Dn*			
8	UDF08 imm8,Dn				UDFU08 imm8,Dn				UDF28 imm8,Dn*				UDFU28 imm8,Dn*			
9	UDF09 imm8,Dn				UDFU09 imm8,Dn				UDF29 imm8,Dn*				UDFU29 imm8,Dn*			
A	UDF10 imm8,Dn				UDFU10 imm8,Dn				UDF30 imm8,Dn*				UDFU30 imm8,Dn*			
B	UDF11 imm8,Dn				UDFU11 imm8,Dn				UDF31 imm8,Dn*				UDFU31 imm8,Dn*			
C	UDF12 imm8,Dn				UDFU12 imm8,Dn				UDF32 imm8,Dn*				UDFU32 imm8,Dn*			
D	UDF13 imm8,Dn				UDFU13 imm8,Dn				UDF33 imm8,Dn*				UDFU33 imm8,Dn*			
E	UDF14 imm8,Dn				UDFU14 imm8,Dn				UDF34 imm8,Dn*				UDFU34 imm8,Dn*			
F	UDF15 imm8,Dn				UDFU15 imm8,Dn				UDF35 imm8,Dn*				UDFU35 imm8,Dn*			

AM30

*: Installed for AM30/AM32. Not used for AM30.

2nd byte (1st byte: F4) Instruction for 4-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	MOV (d16,Am),Dn															
1	MOV Dm,(d16,An)															
2	MOV (d16,Am),An															
3	MOV Am,(d16,An)															
4	MOVBU (d16,Am),Dn															
5	MOVBU Dm,(d16,An)															
6	MOVHU (d16,Am),Dn															
7	MOVHU Dm,(d16,An)															
8	MOV A0,(abs16)				MOV A1,(abs16)				MOV A2,(abs16)				MOV A3,(abs16)			
9	MOV A0,(d16,SP)	MOV D0,(d16,SP)	MOVBU D0,(d16,SP)	MOVHU D0,(d16,SP)	MOV A1,(d16,SP)	MOV D1,(d16,SP)	MOVBU D1,(d16,SP)	MOVHU D1,(d16,SP)	MOV A2,(d16,SP)	MOV D2,(d16,SP)	MOVBU D2,(d16,SP)	MOVHU D2,(d16,SP)	MOV A3,(d16,SP)	MOV D3,(d16,SP)	MOVBU D3,(d16,SP)	MOVHU D3,(d16,SP)
A	MOV (abs16),An															
B	MOV (d16,SP),An				MOV (d16,SP),Dn				MOVBU (d16,SP),Dn				MOVHU (d16,SP),Dn			
C	ADD imm16,Dn								CMP imm16,Dn							
D	ADD imm16,An								CMP imm16,An							
E	AND imm16,Dn				OR imm16,Dn				XOR imm16,Dn				BTST imm16,Dn			
F	BSET imm8,(d8,An)				BCLR imm8,(d8,An)				BTST imm8,(d8,An)				AND imm16,PSW	OR imm16,PSW	ADD imm16,SP	CALLS (d16,PC)

2nd byte (1st byte: FB) Instruction for 4-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	UDF00 imm16,Dn				UDFU00 imm16,Dn				UDF20 imm16,Dn*				UDFU20 imm16,Dn*			
1	UDF01 imm16,Dn				UDFU01 imm16,Dn				UDF21 imm16,Dn*				UDFU21 imm16,Dn*			
2	UDF02 imm16,Dn				UDFU02 imm16,Dn				UDF22 imm16,Dn*				UDFU22 imm16,Dn*			
3	UDF03 imm16,Dn				UDFU03 imm16,Dn				UDF23 imm16,Dn*				UDFU23 imm16,Dn*			
4	UDF04 imm16,Dn				UDFU04 imm16,Dn				UDF24 imm16,Dn*				UDFU24 imm16,Dn*			
5	UDF05 imm16,Dn				UDFU05 imm16,Dn				UDF25 imm16,Dn*				UDFU25 imm16,Dn*			
6	UDF06 imm16,Dn				UDFU06 imm16,Dn				UDF26 imm16,Dn*				UDFU26 imm16,Dn*			
7	UDF07 imm16,Dn				UDFU07 imm16,Dn				UDF27 imm16,Dn*				UDFU27 imm16,Dn*			
8	UDF08 imm16,Dn				UDFU08 imm16,Dn				UDF28 imm16,Dn*				UDFU28 imm16,Dn*			
9	UDF09 imm16,Dn				UDFU09 imm16,Dn				UDF29 imm16,Dn*				UDFU29 imm16,Dn*			
A	UDF10 imm16,Dn				UDFU10 imm16,Dn				UDF30 imm16,Dn*				UDFU30 imm16,Dn*			
B	UDF11 imm16,Dn				UDFU11 imm16,Dn				UDF31 imm16,Dn*				UDFU31 imm16,Dn*			
C	UDF12 imm16,Dn				UDFU12 imm16,Dn				UDF32 imm16,Dn*				UDFU32 imm16,Dn*			
D	UDF13 imm16,Dn				UDFU13 imm16,Dn				UDF33 imm16,Dn*				UDFU33 imm16,Dn*			
E	UDF14 imm16,Dn				UDFU14 imm16,Dn				UDF34 imm16,Dn*				UDFU34 imm16,Dn*			
F	UDF15 imm16,Dn				UDFU15 imm16,Dn				UDF35 imm16,Dn*				UDFU35 imm16,Dn*			

AM30

*: Installed for AM31/AM32. Not used for AM30.

2nd byte (1st byte: FC) Instruction for 6-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	MOV (d32,Am),Dn															
1	MOV Dm,(d32,An)															
2	MOV (d32,Am),An															
3	MOV Am,(d32,An)															
4	MOVBU (d32,Am),Dn															
5	MOVBU Dm,(d32,An)															
6	MOVHU (d32,Am),Dn															
7	MOVHU Dm,(d32,An)															
8	MOV A0,(abs32)	MOV D0,(abs32)	MOVBU D0,(abs32)	MOVHU D0,(abs32)	MOV A1,(abs32)	MOV D1,(abs32)	MOVBU D1,(abs32)	MOVHU D1,(abs32)	MOV A2,(abs32)	MOV D2,(abs32)	MOVBU D2,(abs32)	MOVHU D2,(abs32)	MOV A3(abs32)	MOV D3,(abs32)	MOVBU D3,(abs32)	MOVHU D3,(abs32)
9	MOV A0,(d32,SP)	MOV D0,(d32,SP)	MOVBU D0,(d32,SP)	MOVHU D0,(d32,SP)	MOV A1,(d32,SP)	MOV D1,(d32,SP)	MOVBU D1,(d32,SP)	MOVHU D1,(d32,SP)	MOV A2,(d32,SP)	MOV D2,(d32,SP)	MOVBU D2,(d32,SP)	MOVHU D2,(d32,SP)	MOV A3,(d32,SP)	MOV D3,(d32,SP)	MOVBU D3,(d32,SP)	MOVHU D3,(d32,SP)
A	MOV (abs32),An				MOV (abs32),Dn				MOVBU (abs32),Dn				MOVHU (abs32),Dn			
B	MOV (d32,SP),An				MOV (d32,SP),Dn				MOVBU (d32,SP),Dn				MOVHU (d32,SP),Dn			
C	ADD imm32,Dn				SUB imm32,Dn				CMP imm32,Dn				MOV imm32,Dn			
D	ADD imm32,An				SUB imm32,An				CMP imm32,An				MOV imm32,An			
E	AND imm32,Dn				OR imm32,Dn				XOR imm32,Dn				BTST imm32,Dn			
F															ADD imm32,SP	CALLS (d32,PC)

2nd byte (1st byte: FD) Instruction for 6-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	UDF00 imm32,Dn				UDFU00 imm32,Dn				UDF20 imm32,Dn*				UDFU20 imm32,Dn*			
1	UDF01 imm32,Dn				UDFU01 imm32,Dn				UDF21 imm32,Dn*				UDFU21 imm32,Dn*			
2	UDF02 imm32,Dn				UDFU02 imm32,Dn				UDF22 imm32,Dn*				UDFU22 imm32,Dn*			
3	UDF03 imm32,Dn				UDFU03 imm32,Dn				UDF23 imm32,Dn*				UDFU23 imm32,Dn*			
4	UDF04 imm32,Dn				UDFU04 imm32,Dn				UDF24 imm32,Dn*				UDFU24 imm32,Dn*			
5	UDF05 imm32,Dn				UDFU05 imm32,Dn				UDF25 imm32,Dn*				UDFU25 imm32,Dn*			
6	UDF06 imm32,Dn				UDFU06 imm32,Dn				UDF26 imm32,Dn*				UDFU26 imm32,Dn*			
7	UDF07 imm32,Dn				UDFU07 imm32,Dn				UDF27 imm32,Dn*				UDFU27 imm32,Dn*			
8	UDF08 imm32,Dn				UDFU08 imm32,Dn				UDF28 imm32,Dn*				UDFU28 imm32,Dn*			
9	UDF09 imm32,Dn				UDFU09 imm32,Dn				UDF29 imm32,Dn*				UDFU29 imm32,Dn*			
A	UDF10 imm32,Dn				UDFU10 imm32,Dn				UDF30 imm32,Dn*				UDFU30 imm32,Dn*			
B	UDF11 imm32,Dn				UDFU11 imm32,Dn				UDF31 imm32,Dn*				UDFU31 imm32,Dn*			
C	UDF12 imm32,Dn				UDFU12 imm32,Dn				UDF32 imm32,Dn*				UDFU32 imm32,Dn*			
D	UDF13 imm32,Dn				UDFU13 imm32,Dn				UDF33 imm32,Dn*				UDFU33 imm32,Dn*			
E	UDF14 imm32,Dn				UDFU14 imm32,Dn				UDF34 imm32,Dn*				UDFU34 imm32,Dn*			
F	UDF15 imm32,Dn				UDFU15 imm32,Dn				UDF35 imm32,Dn*				UDFU35 imm32,Dn*			

AM30 *: Installed for AM31/AM32. Not used for AM30.

2nd byte (1st byte: FE) Instruction for 7/5-byte

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	BSET imm8, (abs32)	BCLR imm8, (abs32)	BTST imm8, (abs32)													
1																
2																
3																
4																
5																
6																
7																
8	BSET imm8, (abs16)*	BCLR imm8, (abs16)*	BTST imm8, (abs16)*													
9																
A																
B																
C																
D																
E																
F																

AM30 AM31 * : Installed for AM32. Not used for AM30/AM31.

2nd byte (1st byte:F7) reserved map

Upper/lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

Index

5

Index

J

L

M

N

O

R

MN1030/MN103S Series
Instruction Manual

January, 2003 4th Edition

Issued by Matsushita Electric Industrial Co., Ltd.

Semiconductor Company, Matsushita Electric Industrial Co., Ltd.

Nagaokakyo, Kyoto 617-8520, Japan
Tel: (075) 951-8151
http://www.panasonic.co.jp/semicon/

SALES OFFICES

■ NORTH AMERICA

●U.S.A. Sales Office:
Panasonic Industrial Company [PIC]
• New Jersey Office:
Two Panasonic Way Secaucus, New Jersey 07094 U.S.A.
Tel: 1-201-348-5257 Fax:1-201-392-4652
• Chicago Office:
1707 N. Randall Road Elgin, Illinois 60123-7847 U.S.A.
Tel: 1-847-468-5720 Fax:1-847-468-5725
• Milpitas Office:
1600 McCandless Drive Milpitas, California 95035 U.S.A.
Tel: 1-408-942-2912 Fax:1-408-946-9063
• Atlanta Office:
1225 Northbrook Parkway Suite 1-151 Suwanee, GA 30024 U.S.A.
Tel: 1-770-338-6953 Fax:1-770-338-6849
• San Diego Office:
9444 Balboa Avenue, Suite 185, San Diego, California 92123 U.S.A.
Tel: 1-619-503-2903 Fax:1-858-715-5545

●Canada Sales Office:
Panasonic Canada Inc. [PCI]
5770 Ambler Drive 27 Mississauga, Ontario, L4W 2T3 CANADA
Tel: 1-905-238-2315 Fax:1-905-238-2414

■ LATIN AMERICA

●Mexico Sales Office:
Panasonic de Mexico, S.A. de C.V. [PANAMEX]
Amores 1120 Col. Del Valle Delegacion Benito Juarez C.P. 03100 Mexico, D.F. MEXICO
Tel: 52-5-488-1000 Fax:52-5-488-1073
• Guadalajara Office:
SUCURSAL GUADALAJARA
Av. Lazaro Cardenas 2305 Local G-102 Plaza Comercial Abastos; Col. Las Torres Guadalajara, Jal. 44920 MEXICO
Tel: 52-3-671-1205 Fax:52-3-671-1256

●Brazil Sales Office:
Panasonic do Brasil Ltda. [PANABRAS]
Caixa Postal 1641, Sao Jose dos Campos, Estado de Sao Paulo
Tel: 55-12-335-9000 Fax:55-12-331-3789

■ EUROPE

●Europe Sales Office:
Panasonic Industrial Europe GmbH [PIE]
• U.K. Sales Office:
Willoughby Road, Bracknell, Berks., RG12 8FP, THE UNITED KINGDOM
Tel: 44-1344-85-3671 Fax:44-1344-85-3853
• Germany Sales Office:
Hans-Pinsel-Strasse 2 85540 Haar, GERMANY
Tel: 49-89-46159-119 Fax:49-89-46159-195

■ ASIA

●Singapore Sales Office:
Panasonic Semiconductor of South Asia [PSSA]
300 Beach Road, #16-01, The Concourse, Singapore 199555 THE REPUBLIC OF SINGAPORE
Tel: 65-6390-3688 Fax:65-6390-3689

●Malaysia Sales Office:
Panasonic Industrial Company (M) Sdn. Bhd. [PICM]
• Head Office:
Tingkat 16B, Menara PKNS Petaling Jaya, No.17, Jalan Yong Shook Lin 46050 Petaling Jaya, Selangor Darul Ehsan, MALAYSIA
Tel: 60-3-7951-6601 Fax:60-3-7954-5968
• Penang Office:
Suite 20-07,20th Floor, MWE Plaza, No.8, Lebuh Farquhar,10200 Penang, MALAYSIA
Tel: 60-4-201-5113 Fax:60-4-261-9989
• Johore Sales Office:
Menara Pelangi, Suite8.3A, Level8, No.2, Jalan Kuning Taman Pelangi, 80400 Johor Bahru, Johor, MALAYSIA
Tel: 60-7-331-3822 Fax:60-7-355-3996

●Thailand Sales Office:
Panasonic Industrial (THAILAND) Ltd. [PICT]
252-133 Muang Thai-Phatra Complex Building, 31st Fl. Rachadaphisek Rd., Huaykwang, Bangkok 10320, THAILAND
Tel: 66-2-693-3428 Fax:66-2-693-3422

●Philippines Sales Office: [PISP]
Panasonic Indsutrial Sales Philippines Division of Matsushita Electric Philippines Corporation
102 Laguna Boulevard,Bo.Don Jose Laguna Technopark, Santa. Rosa, Laguna 4026 PHILIPPINES
Tel: 63-2-520-8615 Fax:63-2-520-8629

●India Sales Office:
National Panasonic India Ltd. [NPI]
E Block, 510, International Trade Tower Nehru Place, New Delhi_110019 INDIA
Tel: 91-11-629-2870 Fax:91-11-629-2877

●Indonesia Sales Office:
P.T.MET & Gobel [M&G]
JL. Dewi Sartika (Cawang 2) Jakarta 13630, INDONESIA
Tel: 62-21-801-5666 Fax:62-21-801-5675

●China Sales Office:
Panasonic Industrial (Shanghai) Co., Ltd. [PI(SH)]
Floor 6, Zhong Bao Mansion, 166 East Road Lujian Zui, PU Dong New District, Shanghai, 200120 CHINA
Tel: 86-21-5866-6114 Fax:86-21-5866-8000
Panasonic Industrial (Tianjin) Co., Ltd. [PI(TJ)]
Room No.1001, Tianjin International Building 75, Nanjin Road, Tianjin 300050, CHINA
Tel: 86-22-2313-9771 Fax:86-22-2313-9770
Panasonic SH Industrial Sales (Shenzhen) Co., Ltd. [PSI(SZ)]
7A-107, International Bussiness & Exhibition Centre, Futian Free Trade Zone, Shenzhen 518048, CHINA
Tel: 86-755-8359-8500 Fax:86-755-8359-8516
Panasonic Shun Hing Industrial Sales (Hong Kong) Co., Ltd. [PSI(HK)]
11th Floor, Great Eagle Center 23 Harbour Road, Wanchai, HONG KONG
Tel: 852-2529-7322 Fax:852-2865-3697

●Taiwan Sales Office:
Panasonic Industrial Sales (Taiwan) Co.,Ltd. [PIST]
• Head Office:
6F, 550, Sec. 4, Chung Hsiao E. RD. Taipei, 110, TAIWAN
Tel: 886-2-2757-1900 Fax:886-2-2757-1906
• Kaohsiung Office:
6th Floor, Hsin Kong Bldg. No.251, Chi Hsien 1st Road Kaohsiung 800, TAIWAN
Tel: 886-7-346-3815 Fax:886-7-236-8362

●Korea Sales Office:
Panasonic Industrial Korea Co., Ltd. [PIKL]
Kukje Center Bldg. 11th Fl., 191 Hangangro 2ga, Youngsan-ku, Seoul 140-702, KOREA
Tel: 82-2-795-9600 Fax:82-2-795-1542

220103

Printed in JAPAN