eSL/eSLS Series (+ eSLZ000)

16 Bits DSP Sound Processor

PROGRAMMING GUIDE

Doc. Version 1.5

ELAN MICROELECTRONICS CORP.

April 2009

Trademark Acknowledgments:

IBM is a registered trademark and PS/2 is a trademark of IBM.

Windows is a trademark of Microsoft Corporation.

EASY SOUND is a registered trademark of ELAN Microelectronics Corp.

ELAN and ELAN logo are trademarks of ELAN Microelectronics Corporation.

The contents of this publication are subject to change without further notice. ELAN Microelectronics assumes no responsibility concerning the accuracy, adequacy, or completeness of this publication. ELAN Microelectronics makes no commitment to update, or to keep current the information and material contained in this publication. Such information and material may change to conform to each confirmed order.

In no event shall ELAN Microelectronics be made responsible for any claims attributed to errors, omissions, or other inaccuracies in the information or material contained in this publication. ELAN Microelectronics shall not be liable for direct, indirect, special incidental, or consequential damages arising out of the use of such information or material.

The software (if any) described in this publication is furnished under a license or nondisclosure agreement, and may be used or copied only in accordance with the terms of such agreement.

ELAN Microelectronics products are not intended for use in life support appliances, devices, or systems. Use of ELAN Microelectronics product in such applications is not supported and is prohibited.
NO PART OF THIS PUBLICATION MAY BE REPRODUCED OR TRANSMITTED IN ANY FORM OR BY ANY MEANS WITHOUT THE EXPRESSED WRITTEN PERMISSION OF ELAN MICROELECTRONICS.

ELAN MICROELECTRONICS CORPORATION

Headquarters:
No. 12, Innovation Road 1
Hsinchu Science Park
Hsinchu, Taiwan 30077
Tel: +886 3 563-9977
Fax: +886 3 563-9966
http://www.emc.com.tw

Hong Kong:
Elan (HK) Microelectronics Corporation, Ltd.
Flat A, 19/F, World Tech Centre
95 How Ming Street, Kwun Tong
Kowloon , HONG KONG
Tel: +852 2723-3376
Fax: +852 2723-7780
elanhk@emc.com.hk

USA:
Elan Information Technology Group (USA)
1821 Saratoga Ave., Suite 250
Saratoga, CA 95070
USA
Tel: +1 408 366-8225
Fax: +1 408 366-8220

Shenzhen:
Elan Microelectronics Shenzhen, Ltd.
3F, SSMEC Bldg., Gaoxin S. Ave. I
Shenzhen Hi-tech Industrial Park
(South Area), Shenzhen, CHINA
Tel: +86 755 2601-0565
Fax: +86 755 2601-0500

Shanghai:
Elan Microelectronics Shanghai, Ltd.
23/Bldg. #115 Lane 572, Bibo Road
Zhangjiang Hi-Tech Park
Shanghai, CHINA
Tel: +86 21 5080-3866
Fax: +86 21 5080-4600

Contents

Programming Guide Revision History

Doc. Version	Revision Description	Date
1.0	Initial Offical Release with new revised examples	2006/12/15
1.1	Add comment in Section 2.3.1 Modified descriptions and figure in Section 1.15	2007/03/29
1.2	Modified long MOV instruction in Section 2.3.1, 2.4.36	2007/08/10
1.3	Modified BSR description in Section 1.4.2 Modified data direct address mode in Section 1.5.6 Modified mov instruction in Section 1.11.1	2008/08/10
1.4	Modified ADC Description in Section 2.4.1 Modified ADD Description in Section 2.4.2 Modified RETI Description in Section 2.4.48 Modified TRAP Description in Section 2.4.59 Modified example description inSection 2.5.5	2009/02/05
1.5	Modify Algorithm-related information in Chapter 1	2009/04/15

Chapter 1
Introduction

1.1 Introduction to eSL/eSLS Series and eSLZ000 ICs

The eSL/eSLS Series and eSLZ000 ICs (or "eSL Series" for short) differ from each other in the following manner:

- eSL ICs fully comply with all features of the eSL Series.
- eSLS ICs is the simplified version of the eSL ICs. Hence, these chips have simpler performance than the eSL ICs.
- eSLZ000 IC is the eSLZ000 ICE kernel chip used to emulate the eSL/eSLS Series.

ELAN eSL Series ICs are 16-bit DSP Sound Processor with multi-channel speech and instrument playback based on Elan 16-bit DSP platform. The series has a powerful 16-bit DSP architecture that handles most of the speech/melody functions. Speech and melody can be played back simultaneously with the speech synthesis implemented by software. A wide range of compression bit rates and various volume levels are supported. eSL Series chips are equipped with real instrument waveform which enable the chips to obtain good quality melody. ELAN eSL peripherals include RTC, Timer, WDT, DAC, PWM, etc.

The eSL Series ICs offer FAST, SLEEP, GREEN, and SLOW modes of operation. The use of GREEN and SLOW mode further reduces power consumption. Moreover, GREEN mode also provides RTC function for wake-up propose.

The chips are designed as a cost effective processors offering optimized performance and are ideal for such applications as high compression rate digital voice signal, high quality instrument melody, voice recognition, digital sound effect, etc. The eSL Series constructive features motivate exploration into wide variety of new creative ideas for more innovative products.

ELAN eSL Series perform extremely well in speech application based on powerful DSP architecture and are endowed with good algorithm for audio compression.

1.2 Features

- **MCU**
  - 16-bit RISC CPU architecture
  - CPU clock: 20MHz @ 3.3V (eSL and eSLS only)
  - CPU clock: 18MHz @ 3.3V (eSLZ000 only)
  - Programmable PLL
  - 4 CPU operation modes (Fast, Slow, Green, & Sleep)
  - Powerful DSP Instruction Set (MAC, DIV, RPT, LOOP)
  - Saturation mode supported
  - 8 general purpose registers (GPR)
  - 21 interrupt sources with 2-level priority (eSL and eSLZ000 only)
  - 18 interrupt source with 2-level priority (eSLS only)
- **Memory**
  - 32K-word program memory
  - 2K-word data RAM (eSL and eSLS only)
  - 8K-word data RAM (eSLZ000 only)
  - 128/256/512K-word data ROM (eSL and eSLS only)
  - External data ROM up to 32MB (eSLZ000 only)
- **Peripherals**
  - Real Time Clock (RTC) with wake up function
  - Four 8-bit timers, two general purpose timer, two multiple-function timer
  - 8-bit Watch Dog Timer (WDT) with general purpose timer capability
  - 40 GPIO + 8 Output (eSL and eSLZ000 only)
  - 24 GPIO (eSLS only)
  - Serial Peripheral Interface (eSL and eSLZ000 only)
  - 12-bit Analog to Digital Converter with touch panel and MIC inputs (eSL and eSLZ000 only)
  - Built-in regulator

1.3 Block Diagram

As shown in the block diagram below, ELAN eSL Series (eSL/eSLS Series and eSLZ000) utilize a modified Harvard architecture in such a way that the memory is organized into two separated fields; Program ROM and Data RAM. As the memory is separated, the central processing units can read/write data at the same time. Furthermore, the I/O space has an independent address, i.e., the I/O-mapped I/O. The different configurations of each domain are explained in this chapter.

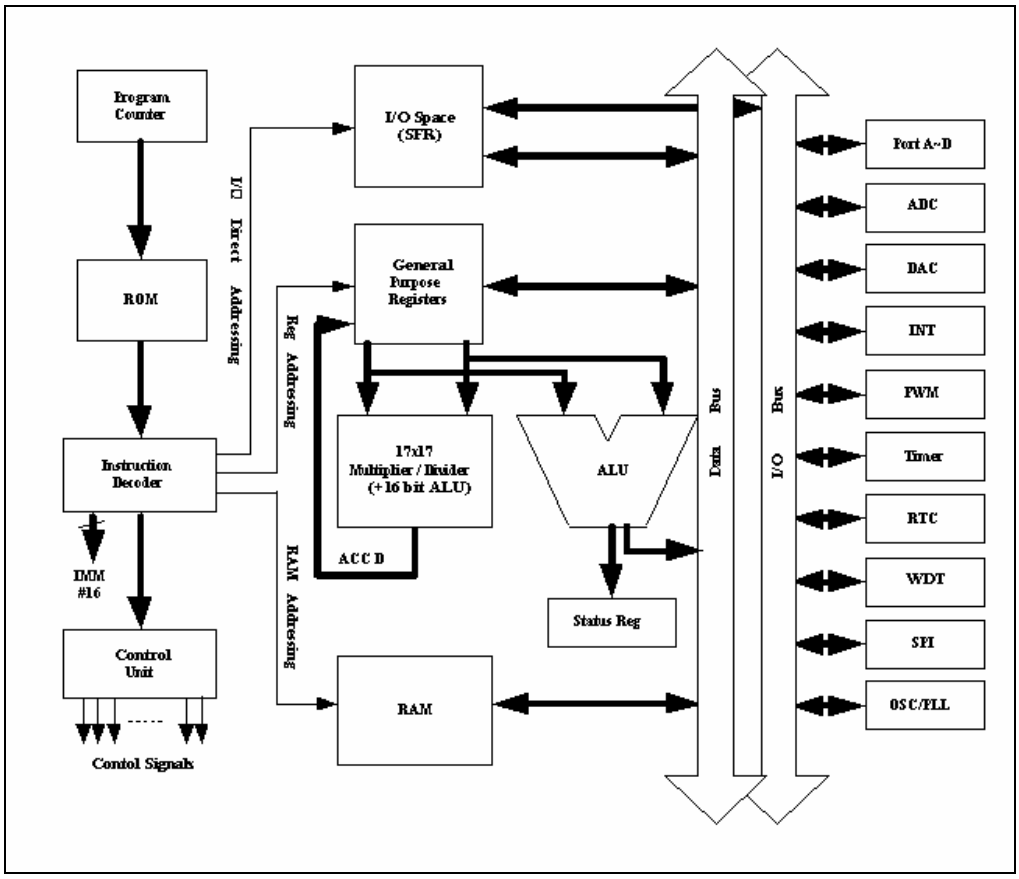


Figure 1-1 eSL Series System Block Diagram

1.4 Program ROM and Data RAM Description

1.4.1 Program ROM/RAM

It includes 32K * 16-bit on-chip ROM (for both eSL & eSLS Series) or RAM (eSLZ000 only) for your program and general data storage utilization. Program counter (PC) is the dedicated counter for program address, and is automatically modified by control flow processing. The eight general purpose registers can be used as Program ROM or RAM pointers.

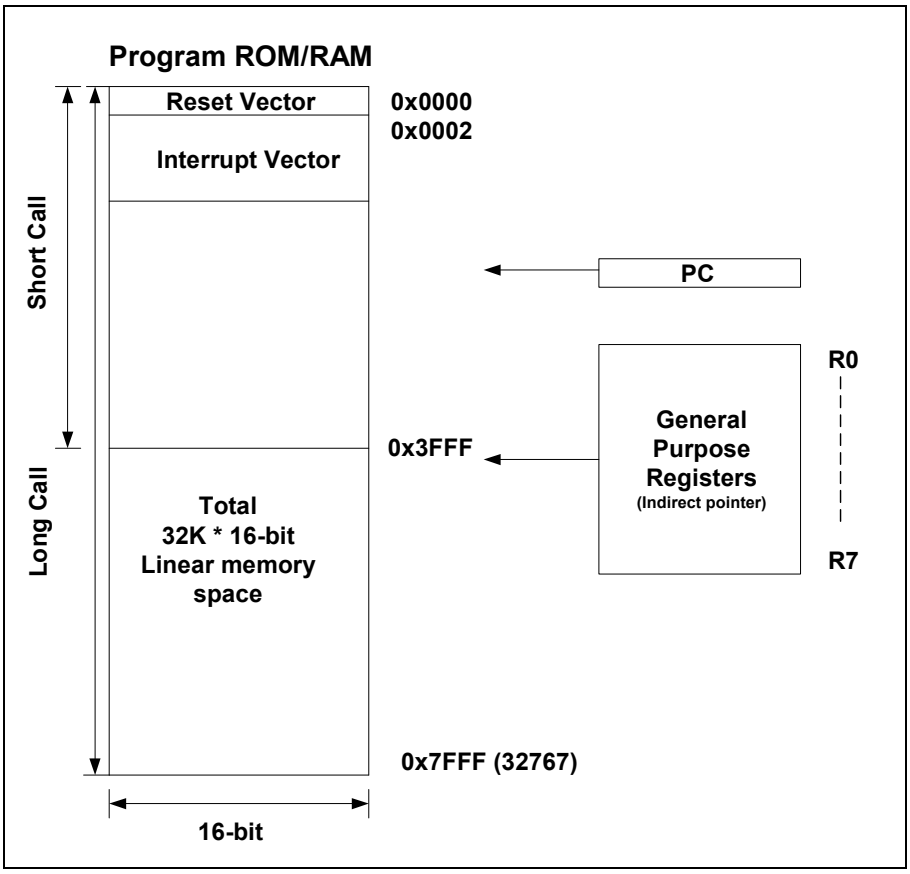


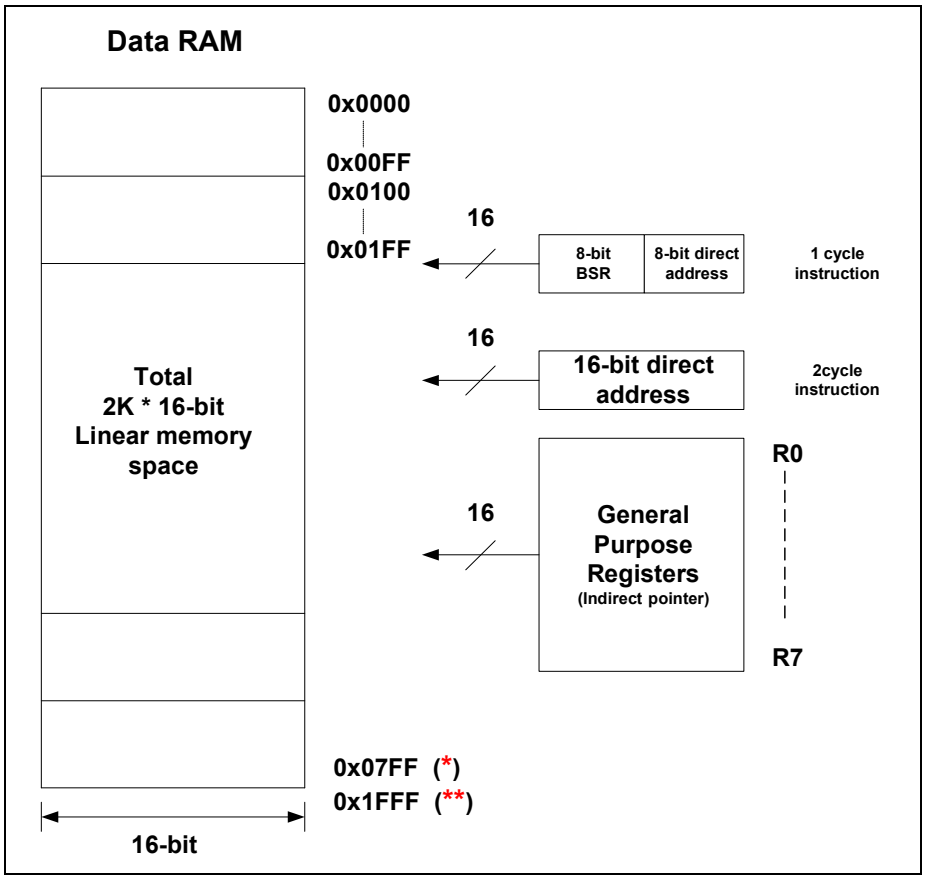
Figure 1-2 eSL Series Program ROM Block Diagram

1.4.2 Data RAM

The block diagram below shows the architecture in the eSL data RAM. It consisted of six different addressing modes for the data memory cover, namely;

- 16-bit direct
- 8-bit direct
- Indirect with Displacement
- Indirect
- Indirect with Post-decrement
- Indirect with Post-increment

BSR (Bank Select Register) is used for MOV instruction. When user use 8-bit MOV instruction, they must make sure the BSR is correct. User doesn't care the BSR if they use 16-bit MOV instruction (L). Please see the data transfer instruction and appendix about code optimization.



* *Applies to eSL and eSLS*
** *Applies to eSLZ000*

Figure 1-3 eSL Series Data RAM Block Diagram

1.5 Addressing Mode

ELAN eSL Series supports powerful and efficient addressing modes. A lot of instructions use several addressing modes. The following sections will describe the available eSL Series addressing modes.

1.5.1 Register Direct Addressing

The operands are in the register file.

Example: R1 = R2 + R3

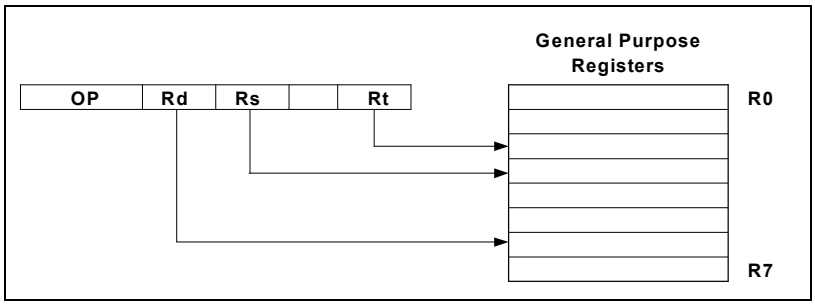


Figure 1-4 Register Direct Addressing Operation Diagram

1.5.2 Register Indirect Addressing

Operand address is the contents of the registers used when accessing the RAM or ROM.

Example: R3 = [R2] – R1 – B and R3 = P[R1]

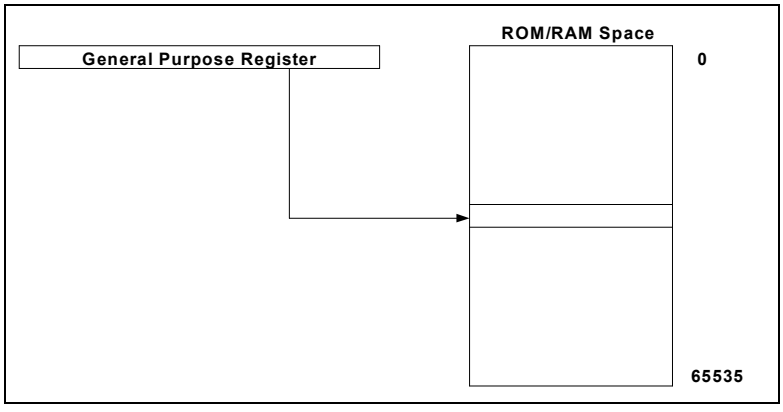


Figure 1-5 Register Indirect Addressing Operation Diagram

1.5.3 Indirect Addressing with Post-Decrement

The indirect register pointer is decremented by 1 after each operation.

Example: R3 = [R5--] and D = R5 * [R6--] (US)

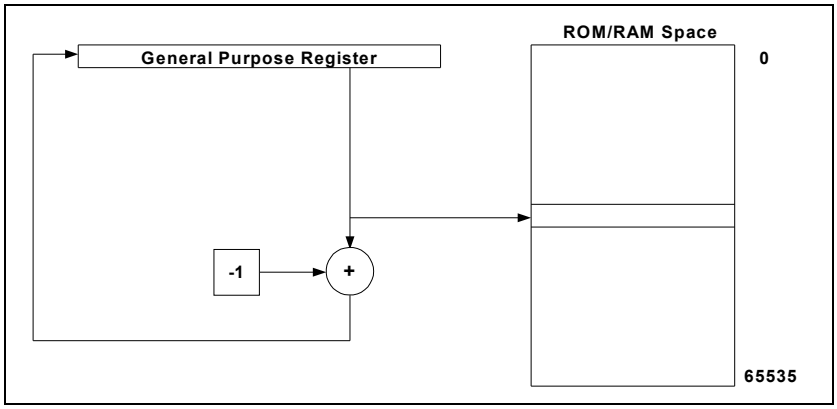


Figure 1-6 Indirect Addressing with Post-Decrement Operation Diagram

1.5.4 Indirect Addressing with Post-Increment

The indirect register pointer is incremented by 1 after each operation. The addressing mode is quite powerful when in bulk operation or during operation that needs a lot of memory access. The purpose of the addressing mode is to keep high the MAC data path utilization.

Example: D = D + [R3++] * P[R4++] and [R1++] = P[R5++].

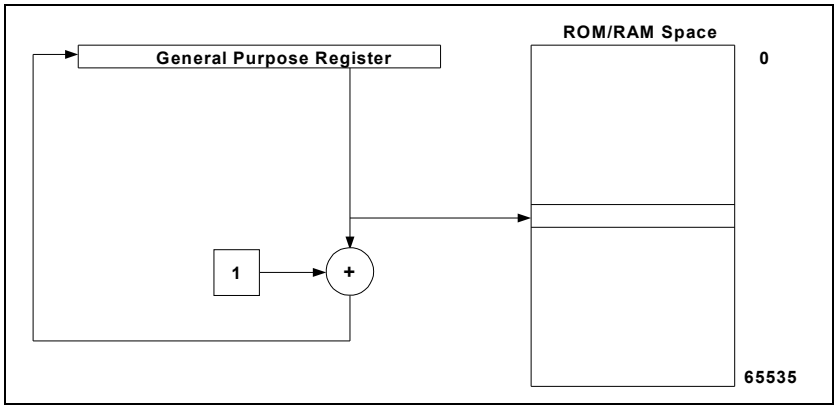


Figure 1-7 Indirect Addressing with Post-Increment Operation Diagram

1.5.5 I/O Direct Addressing

The address is contained in a 7-bit instruction word. The second operand is either Rd[1] or Rs[2] (destination or source register respectivley) used by IN and OUT instructions to read from or write to the I/O registers.

Example: R6 = IO[PORTA] and POP IO[PORTC].

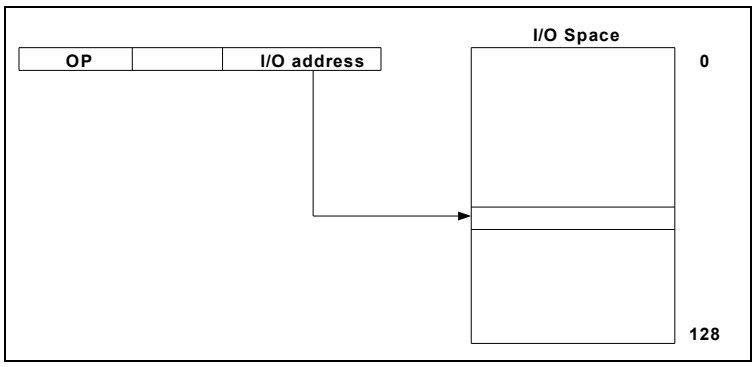


Figure 1-8 I/O Indirect Addressing Operation Diagram

1.5.6 RAM (Data) Direct Addressing

The 8-bit data address is contained in the 1-word instruction. Rd or Rs specifies the destination or source register respectively.

The 16-bit data address is contained in the 16 LSBs of a 2-word instruction. Rd or Rs specifies the destination or source register respectively.

Example: R = RAM8

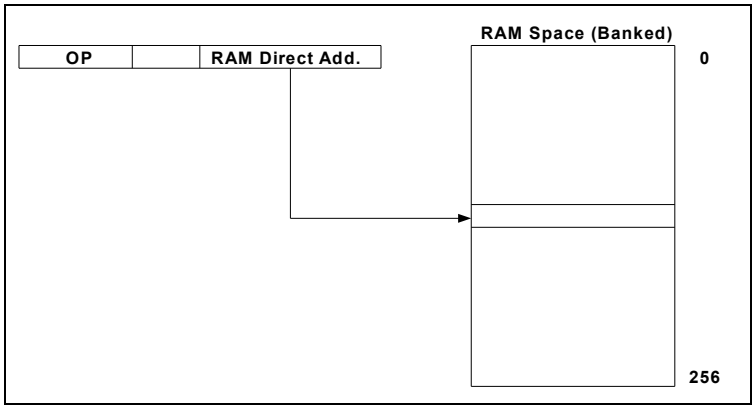


Figure 1-9a RAM (8-Bit Data) Direct Addressing Operation Diagram

[1] **Rd** is the Destination Register of General Purpose Registers.
[2] **Rs** is the Source Register of General Purpose Registers.

Example: R = RAM16 (L)

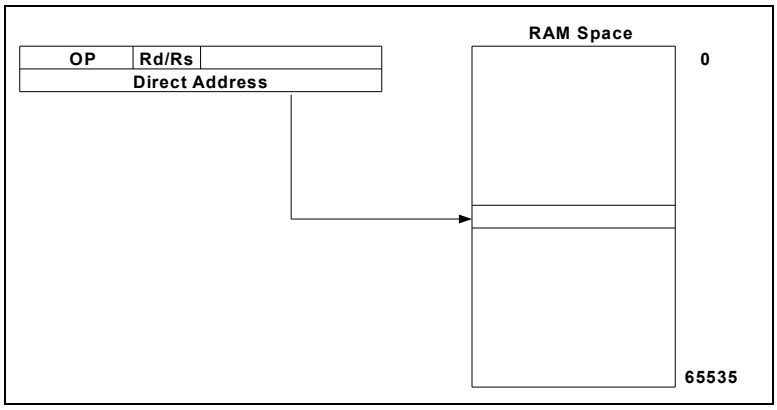


Figure 1-9b RAM (16-Bit Data) Direct Addressing Operation Diagram

1.5.7 Immediate Addressing

The 16-bit program address is contained in the 16 LSBs of a 2-word instruction.

Example: CALL label and JMP label

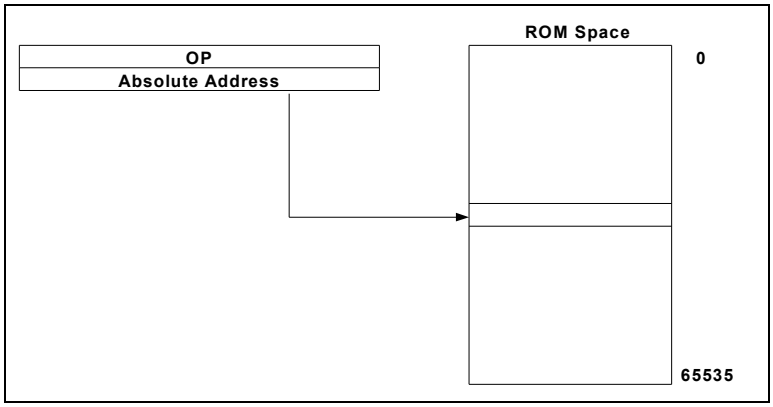


Figure 1-10 Immediate Addressing Operation Diagram

1.5.8 Relative Program Addressing

Program execution continues at address PC + offset + 1. The offset is contained in the instruction word. Short Conditional Branch instructions can only reach –256 to 255 locations away from the current address. However, Long Branch instructions can reach the entire program memory from every location.

NOTE

Long Range Conditional Branch can reach 0 to 65,535 locations, but it needs two words instruction.

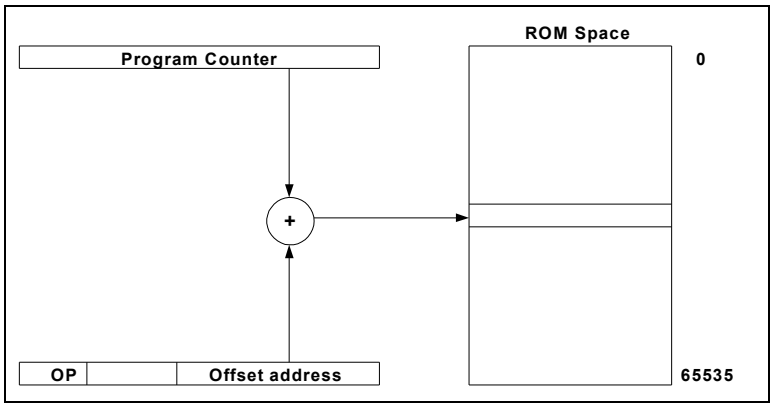


Figure 1-11 Relative Program Addressing Operation Diagram

1.5.9 Data Indirect Addressing with Displacement

Operand address is the result of the register contents added to the address contained in 5 bits of the instruction word. For example, R1 = [R3 -#10]; R3 is the only register that can be the base register.

Example: R1 = [R3 – #10]

NOTE

R3 *is the only register available that can be used as base register*

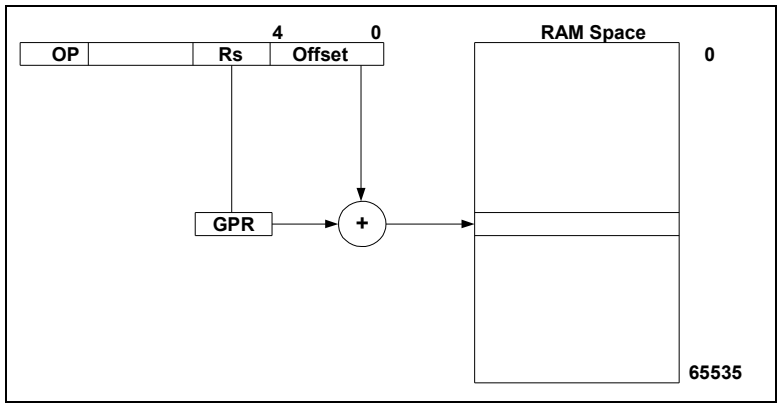


Figure 1-12a Data Indirect with 5-Bit Displacement Addressing Operation Diagram

Operand address is the result of the register contents added to another register.

Example**:** [R3 – R1] = R2

NOTE

R3 *is the only register available that can be used as base register*

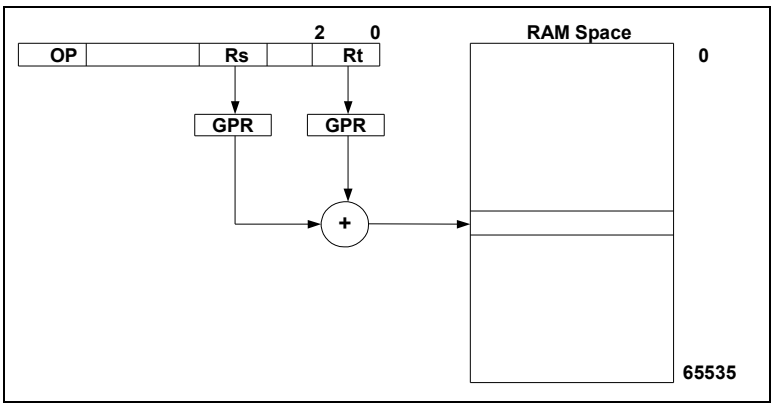


Figure 1-12b Data Indirect with 5-Bit Displacement Addressing Operation Diagram

1.6 Register Model

Figure below shows the eSL Series register configuration. Registers are discussed in further details in the following sections.

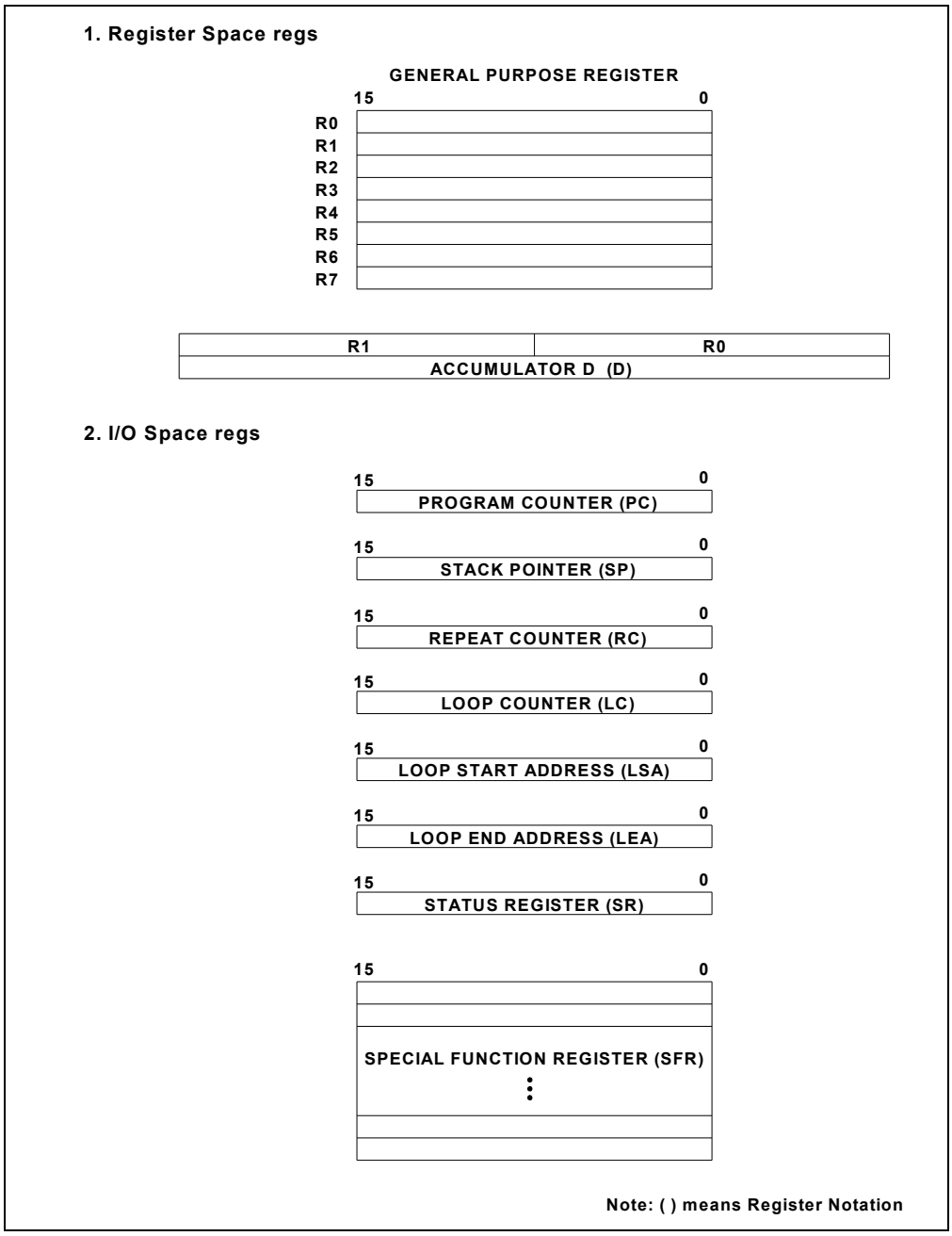


Figure 1-13 The eSL Series Register Configuration

1.6.1 General Purpose Registers

The General Purposes Registers' register space consisted of 8 x 16-bit. They are used as data, address, or as offset registers. The address can be up to 64 Kbytes (ROM, RAM) without any segmentation (bank).

The Registers R0 and R1 are assigned with other functions in addition to their general usage. These two registers are treated as a single double-word (32-bit) accumulator called Accumulator D that holds the operands and the results of arithmetic calculations or data manipulations.

1.6.2 Program Counter

The Program Counter (PC) is a 16-bit wide register that holds the address of the next instruction to be executed. Therefore, the PC can address up to 64K instruction (read only) words.

1.6.3 Stack Pointer

The Stack Pointer (SP) holds the 16-bit address of the last stack location used. It is automatically modified by interrupt processing, subroutine calls, and returns. You may reprogram the SP during initialization to any location within data (RAM) space. The SP can also be used by your program as PUSH and POP instructions. However, be aware that the SP is also used by the CPU.

1.6.4 Repeat and Loop Registers

These four registers; Repeat Counter, Loop Counter, Loop Start Address, and Loop End Address are used as temporary registers when executing repeat or loop instruction. The Repeat and Loop Counter store the repeat times, as well as the start and end address in loop operation.

1.6.5 Status Register

The Status Register (SR) contains the following system status bits:

Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
GIE	x*	X*	x*	x*	SME	S6R	F/I	x*	x*	x*	T	N	Z	V	C

**x* = Don't care. Reserved for future enhancements*

Where:

- **Carry (C) Flag**

C is set when a carry or borrow occurs during an arithmetic operation. The Carry Flag bit is set or reset, depending on the operation that is performed.

For **ADD** instructions:

C = 1: Carry occurs
C = 0: No carry occurs

For **SUBTRACT** instructions:

C = 1: No borrow occurs
C = 0: Borrow occurs

For **COMPARE** instructions:

Same as **SUBTRACT** instruction (above)

For **ROTATION** instructions:

The Carry flag is used as a link between the least significant bit (LSB) and most significant bit (MSB).

- **Overflow (V) Flag**

V is set when a two's complement overflow occurs as a result of an operation.

V = 1: Overflow occurred
V = 0: No overflow occurred

- **Zero (Z) Flag**

The Z bit is set when all the bits of the result are 0's.

Z = 1: The result equals zero after operation.
(R1:R0 = 0 when in MAC operation)

- **Negative (N) Flag**

The negative flag stores the state of the most significant bit of the output result.

N = 1: The result of the operation is negative.
N = 0: The result of the operation is positive.

- **Test (T) Flag**

T is used by Bit test operation instruction (BTEST)

T = 1: The tested bit is 1
T = 0: The tested bit is 0

- **Fractional / Integer (F/I) Flag**

F/I is used by Fractional and Integer modes

F/I = 1: Fractional mode
F/I = 0: Integer mode

■ Shift 6-Bit (S6R) Flag

S6R is used by Shift 6-bit or otherwise

S6R = 1 Shift 6-bit

S6R = 0: Shift 0 bit (F/I = 0)

Shift 1 bit (F/I = 1)

■ Saturation Mode (SME) Flag

SME is used by Saturation mode

SME = 1: Saturation mode enabled

SME = 0: Saturation mode disabled

■ Global Interrupt Enable (GIE) Flag

The Global Interrupt Enable bit must be set (=1) in order for the interrupts to be enabled. If reset, all maskable interrupts are disabled. The GIE bit is cleared by interrupts and restored by the RETI instruction.

GIE = 1: Interrupts are enabled

GIE = 0: Interrupts are disabled

■ Division and Multiplication Modes

S6R	F/I	Division	Multiplication
0	0	Integer	Integer
0	1	Fractional	Fractional
1	0	Integer	Shift 6 bit
1	1	Fractional	Shift 6 bit

■ ALU Saturation Mode - 16BIT

SME	Overflow (V)	Carry (C)	ALU Output
0	X	X	ALU Output
1	0	0	ALU Output
1	0	1	ALU Output
1	1	0	0111111111111111
1	1	1	1000000000000000

■ MAC/MAS Saturation Mode - 32BIT

SME	Overflow (V)	Carry (C)	MAC Output
0	X	X	MAC Output
1	0	0	MAC Output
1	0	1	MAC Output
1	1	0	0X7FFFFFFF
1	1	1	0X80000000

1.7 Instruction Set Description

The eSL Series Instruction Set definitions were prepared and provided under the following considerations**:**

- That the Instruction Set must be complete with no missing functionality.
- That the instructions must be orthogonal, that is, not unnecessarily redundant.

The eSL Series has a 16-bit instruction set (1 or 2 words). It is organized into instruction categories grouped by function as shown in table blow. It's very easy to remember and can be easily incorporated into your program.

Function Groups	Instructions
Logic and Mathematic Instructions	AND, OR, XOR, COM, NEG, CMP, CLR, ADD, ADC, SUB, SUBB, INC, DEC
Branch instructions	JCC, JCS, JLS, JGE,…. (S)JMP, (L)JMP
Shift instructions	SHL, SHR, ROL, ROR, ASR
Data transfer instructions	MOV [R = R; R.l = #8 ; R.h = #8; R = RAM[In,m]; R = ROM[In,m]; RAM[In,m] = R; R = RAMname; RAMname = R IN [R=IO <Add>]; OUT [IO <Add>=R] PUSH R;IO ; POP R;IO; SWAP [R.h = R (Low Byte), R.l = R (High Byte)] R = [R+n]
Bit operation instructions	BS, BC, BTEST, BTG IO; Register; RAM
Control instructions	(S)CALL, (L)CALL, NOP, RET, RETI, RPT, LOOP, TRAP
DSP instructions	MUL.UU, MUL.US, MUL.SU, MUL.SS, MAC, MAS, DIV, DIVS

1.8 Logic and Mathematic Instructions

The eSL has a full set of 6-bit (1 word) and 16-bit(2 words) logic and mathematic instructions.

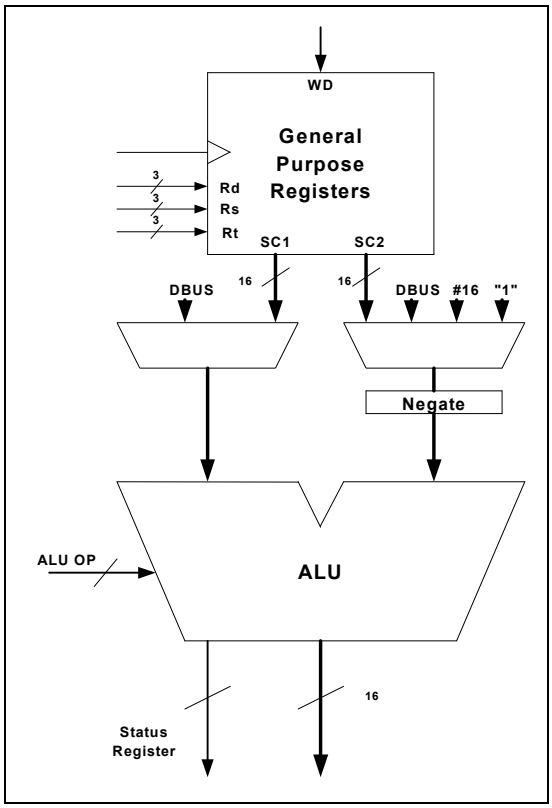


Figure 1-14 The eSL Series Arithmetic Logic Unit

1.8.1 Logic and Mathematic Instruction Definition

Mnemonic	Description	Operand
ADD	Addition without carry	Rn; [Rn]; #6imm; #16imm
ADC	Add with Carry	Rn; [Rn]; #6imm; #16imm
SUB	Subtraction without borrow	Rn; [Rn]; #6imm; #16imm
SUBB	Subtraction with borrow	Rn; [Rn]; #6imm; #16imm
AND	Logical AND	Rn; [Rn]; #6imm; #16imm
OR	Logical OR	Rn; [Rn]; #6imm; #16imm
XOR	Logical exclusive OR	Rn; [Rn]; #6imm; #16imm
CMP	Compare	Rn
NEG	2's complement	Rn
COM	1's complement	Rn

1.9 Conditional Branch Instruction

Conditional Jumps support program branching relative to the program counter. The numeric range of short condition branch is 9-bit offset values (-256 to 255). Long range condition branch can reach 0 to 65,535, but it needs two words instruction.

The Conditional Branch Instructions supported by the eSL Series are shown in the table blow. When a specified condition is met, a signed 9-bit is added to the value in the Program Counter. Or the Program Counter is replaced by 16-bit absolute address.

1.9.1 Conditional Branch Instruction Definitions

Mnemonic	Description	Operation	Comment
IF CC JMP	Jump if carry (C) clear	If (C==0) then jump to PC +n+ 1	Simple
IF CS JMP	Jump if carry (C) set	If (C==1) then jump to PC +n+ 1	Simple
IF VC JMP	Jump if overflow (V) clear	If (V==0) then jump to PC +n+ 1	Simple
IF VS JMP	Jump if overflow (V) set	If (V==1) then jump to PC +n+ 1	Simple
IF NE JMP	Jump if not equal	If (Z==0) then jump to PC +n+ 1	Simple
IF EQ JMP	Jump if equal	If (Z==1) then jump to PC +n+ 1	Simple
IF PL JMP	Jump if plus	If (N==0) then jump to PC +n+ 1	Simple
IF MI JMP	Jump if minus	If (N==1) then jump to PC +n+ 1	Simple
IF TC JMP	Jump if test (T) clear	If (T==0) then jump to PC +n+ 1	Simple
IF TS JMP	Jump if test (T) set	If (T==1) then jump to PC +n+ 1	Simple
IF LO JMP	Jump if lower than	If (C==0) then jump to PC +n+ 1	Unsigned
IF HS JMP	Jump if higher or same	If (C==1) then jump to PC +n+ 1	Unsigned
IF LS JMP	Jump if lower or same	If (N^V==0) then jump to PC +n+ 1	Unsigned
IF GE JMP	Jump if greater than or equal	If (N^V==1) then jump to PC +n+ 1	Signed
IF GT JMP	Jump if greater than	If (Z|(N^V)==0) then jump to PC +n+ 1	Signed
IF LE JMP	Jump if less than or equal	If (Z|(N^V)==1) then jump to PC +n+ 1	Signed
IF LT JMP	Jump if less than	If (C==0 | Z==1) then jump to PC +n+ 1	Signed
JMP	Always Jump	Always jump to PC +n+ 1	Simple

The instruction code fetch and the program counter increment technique end with the following formula:

- PC_new = PC_old + 1 + Offset (when taking a short branch)
- PC_new = 16-bit (LSB) absolute address (when taking a long branch)

NOTE

You can use Conditional Branch simply by keying the mnemonic. The eSL assembler will choose the short or long branch depending on the offset range.

1.10 Shift and Rotation Instructions

Shifts and Rotation instructions are used to shift and rotate the general purpose register respectively. The shifter is capable of performing one bit shifting functions and all pass the shifted-out bit through the C flag bit. Use arithmetic shift right (ASR) for keeping the sign bit.

1.10.1 Shift and Rotation Instruction Definitions

Mnemonic	Description
SHL	Shift Left
SHR	Shift Right
ROL	Rotate Left
ROR	Rotate Right
ASR	Arithmetic Shift Right

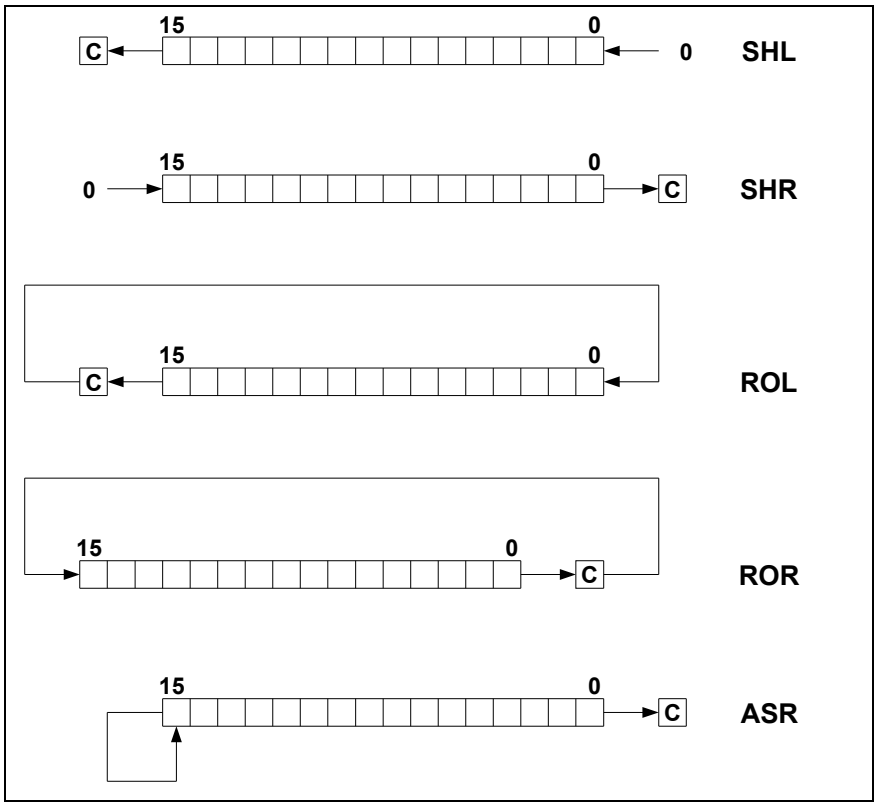


Figure 1-15 Shift & Rotation Instructions Flow Diagram

1.11 Data Transfer Instruction

The Data Transfer instruction moves data from a source to a destination. It provides indirect auto-increase or decrease mode for moving large block of data around main memory. The following table lists the available functions in the Data Transfer instruction.

1.11.1 Data Transfer Instruction Description

Mnemonic	Description
MOV	Move from RHS to LHS[1]
IN	Input from I/O
OUT	Output to I/O
PUSH	Push to TOS
POP	Pop from TOS[2]

[1] ***RHS:*** *Right Hand Side;* ***LHS:*** *Left Hand Side.* [2] ***TOS:*** *Top Of Stack*

MOV instruction provides the data transfer facilities to move data from memory to register (LOAD), or from register to memory (STORE). IN and OUT instructions are capable of moving data via I/O space. Furthermore, PUSH and POP instructions provide a data transfer channel between register and stack (or I/O and stack).

There are two kind data memory mov instruction, one is 8-bit mov instruction, user need to set BSR and 8-bit direct address to do the mov operation, the other is 16-bit long mov instruction, user just need to set 16-bit direct address. Please see the instruction table to understand the performance and space between this two instructions.

1.11.2 Data Transfer Addressing Categories

As the eSL Series architecture are register based, the chips are powerful in moving data from register to any space as demonstrated in the following table.

Source / Destinat'n	Register	RAM Direct	RAM Indirect (with Inc/Dec)	ROM Indirect (with Inc/Dec)	I/O Space (IN/OUTPUT)	Stack (PUSH/POP)	Immediate
Register	Available	Available	Available	Available	Available	Available	Available
RAM Direct	Available	-	-	-	-	-	-
RAM Indirect	Available	-	-	Available	-	-	Available
I/O Space	Available	-	-	-	-	Available	-
Stack	Available	-	-	-	Available	-	-

1.11.3 Data Transfer Programming Examples

Syntax	Description
Rd = Rs	Rs Register → Rd Register
Rd = P[Rs++]	ROM address [Rs] → Rd Register; then Rs=Rs+1
[Rd] = Rs	Rs Register → RAM address [Rd]
Rn.l = #0x5a	Load #imm8 → Rd.l; 0→ Rd.h
Rn.h = #0xa5	Load#imm8→Rd.h; Rd.l un-change
Rd = IO[18]	Read IO port 18 to Rd Register
IO[0xA] = Rs	Write Rs Register to IO port 10(0xA)
PUSH IO[7]	Save the content of IO port 7 on the stack
POP IO[8]	Read stack to IO port 8
PUSH Rs	Write Rs register to stack
POP Rd	Restore Rd from stack

1.12 Bit Operation Instruction

The Operations instruction uses a mask value to test or change the value of individual bits in I/O, RAM, or registers.

■ The Space Definitions

Space
Register
RAM Direct (0x0000 ~ 0x0007)
I/O [0x00 ~ 0x0F]

■ The Bit Operating Definitions

Mnemonic	Description	Operation
BS	Bit b set as 1	Mask OR (b=1; others=0)
BC	Bit b clear as 0	Mask AND (b=0; others=1)
BTEST	Bit b test as 1	Mask AND (b=1; others=0)
BTG	Bit b toggle	Mask XOR (b=1; others=0)

1.13 Control Instructions

The instructions in this group are used for the program control flow. CALL, RET, and RETI instructions provide subroutine and interrupt execution. Unconditional overhead free program loop constructs are supported using the RPT and LOOP instructions.

- **CALL:** Before jumping to target address, the 16-bit return address (PC+1) is pushed onto the stack.
- **RET:** Pop the return address to PC then return from subroutine.
- **RETI:** Pop the return address to PC then return from interrupt service routine.
- **RPT:** Repeat the next instruction
- **LOOP:** Zero-overhead LOOP. (Must include at least 3 instruction; but the last instruction cannot be JMP, CALL, RETURN or RPT instruction. See Section 1.13.2 below).

The repeat (RPT) function can use instructions, such as Multiply/Accumulate and Block Move to increase the execution speed of these instructions. These multicycle instruction effectively become single-cycle instructions after the first iteration of a repeat instruction.

1.13.1 Operating Instruction Example Syntax

Mnemonic	Example Syntax	Original Cycle*
MOV	[R5] = P[R7++]	2
MUL.UU	D = R2 * P[R3++] (UU)	2
MUL.US	D = R5 * P[R7++] (US)	2
MUL.SU	D = R3 * [R5 ++] (SU)	1
MUL.SS	D = R4 * P[R3++] (SS)	2
MAC	D = D + R2 * P[R1++]	2
MAS	D = D – R2 * P[R6++]	2
ADD	R1 = [R2] + R3	1
SUB	R3 = [R2] - R1	1

* Number of cycles when instruction is not repeated

1.13.2 RPT and LOOP Instructions Limitations

Some instructions cannot be repeated with RPT instruction and cannot be the last instruction in a LOOP. These instructions are as below.

Mnemonic	Description
CALL	Unconditional call
JMP instructions	Branch instruction (Include unconditional or condition brach)
RET	Return from subroutine
RETI	Return from interrupt
RPT	Repeat next instruction

1.14 DSP Instruction

The hardware multiplier module supports four types of multiplication. Multiplication is possible for:

- 16-Bit (unsigned) x 16-Bit (unsigned)
- 16-Bit (unsigned) x 16-Bit (signed)
- 16-Bit (signed) x 16-Bit (unsigned)
- 16-Bit (signed) x 16-Bit (signed)

The multiplier deals with 16-bit signed/unsigned numbers; 17 bits are needed to represent the operand in both modes in 2's complement. It can multiplex its output using a scaler (controlled by I/O instruction) to support either fractional or integer results. Under the fractional operation, the result is shifted one bit to the left. Under the integer operation, the result is not shifted.

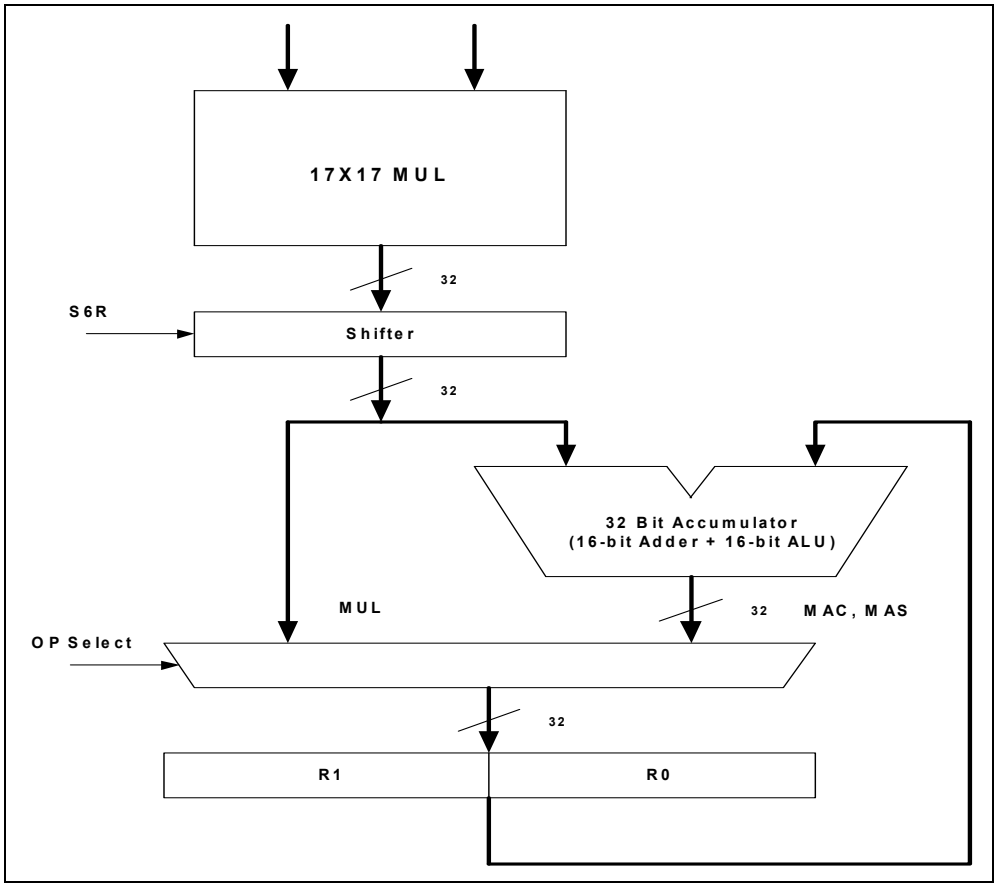


Figure 1-16 DSP Architecture Diagram

The same multiplier is used to support the MAC and MAS instructions. The 16-bit adder combines with 16-bit ALU to perform 32-bit operations.

NOTE

*The multiplier performs [signed * signed] when executing MAC or MAS*

Division is more complex than multiplication. Some algorithms use division a lot. Hence, the eSL Series implemented the division in the hardware. For low-cost implementations where chip size must be minimized, Sequential Division architecture is applied. The most common techniques used in Sequential Division are the Restoring Divide and Non-Restoring Divide. The Restoring Division has timing issues problem. For this reason, eSL Series implemented the Non-Restoring conditional add/subtract division architecture. The division can be signed or unsigned. To perform the division, R1 and R0 store the 32-bit dividend, Rs stores the 16-bit divisor. Then execute the following operation (see Figure 1-17 below)**:**

1. Use XOR result to determine if the dividend should be added to or subtracted from the divisor (Rs). If the result is 0, execute subtraction. Otherwise, addition is executed. Initial XOR result is 0.
2. Shift the register pair (R1, R0) one bit to the left. Move the inverted result of XOR operation into the LSB.
3. Compare the divisor and result sign bit (XOR operation).

NOTE

In Signed Division, the sign bit of Quotient is determined by XOR operation input (Divisor and Dividend sign bits) during the first loop at which time, the Dividends bypass the ALU.

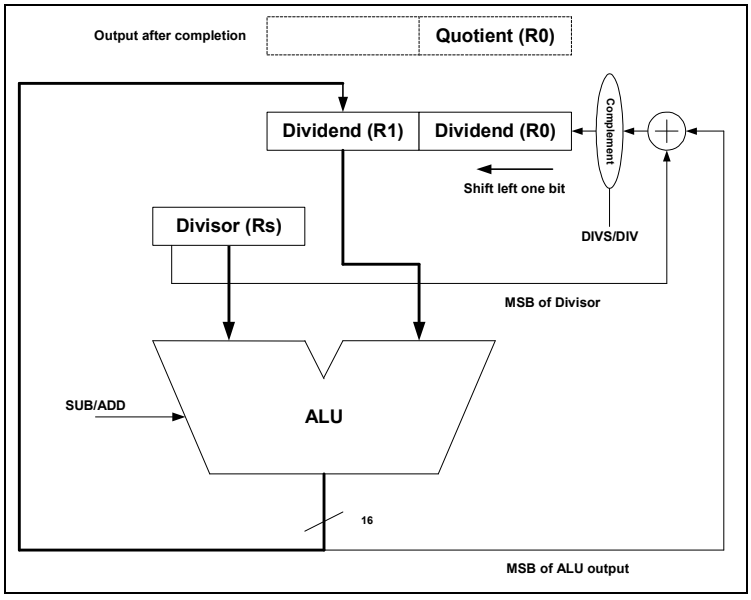


Figure 1-17 Division Architecture Diagram

After repeating the process (Steps 1 to 3) 16 times, the R0 register will contain the quotient. The eSL series will then perform 32-bit by 16-bit division in a fractional format. You can use the following instructions**:**

- **DIV:** For unsigned division
- **DIVS:** For signed division

In the fractional division, the valid results are obtained only when the Dividend (R1, R0) is less or equal to Divisor (Rs). Ensures the magnitude of the quotient is less than one (1.0). To perform the integer division, you must shift the Dividend one bit to the left before dividing.

NOTE

More notes on eSL Series division:

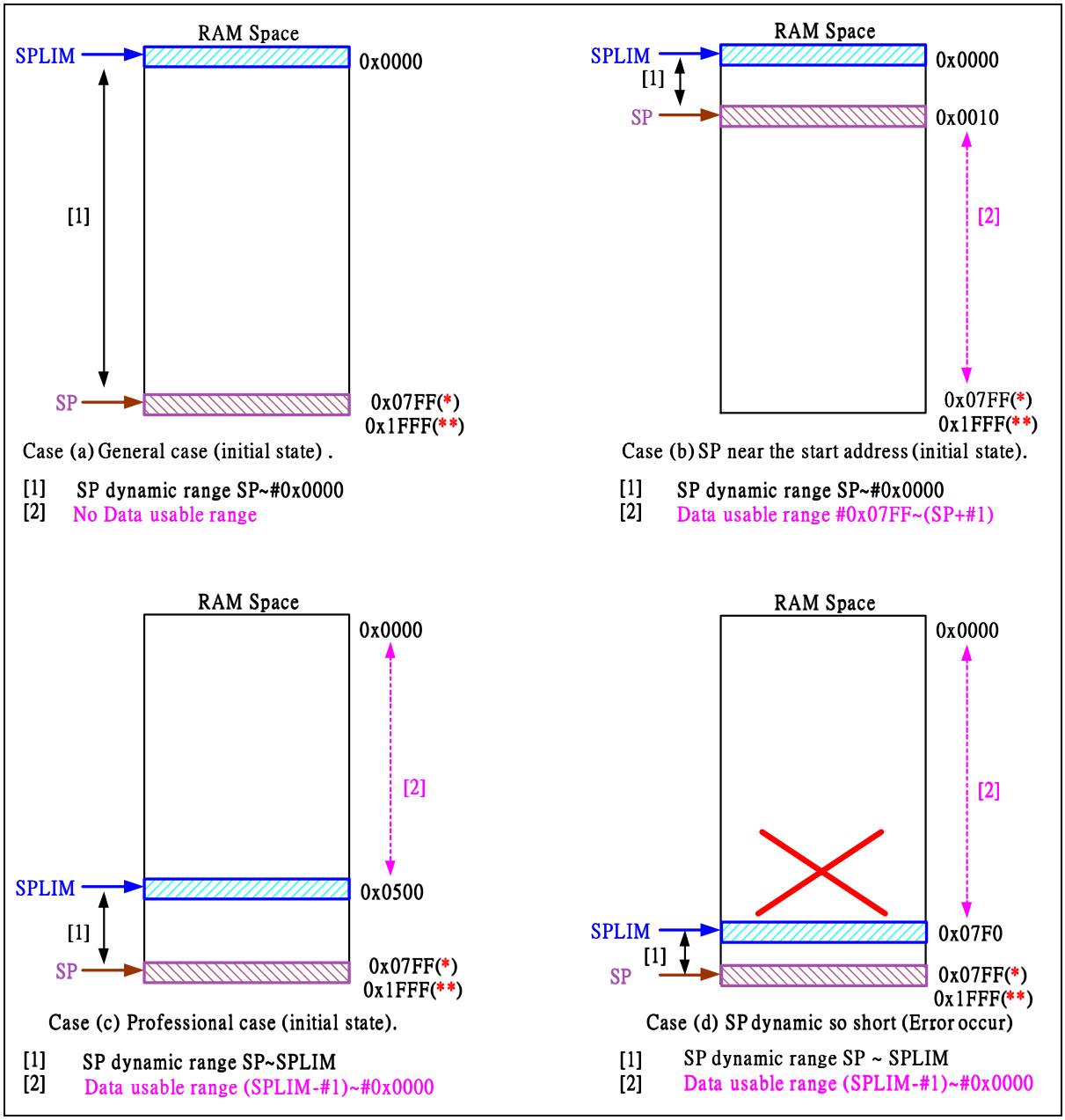
- *eSL Series hardware can NOT check division overflow and division with zero divisor. You must preclude these conditions through software manipulation.*
- *The quotient produced by a division with a negative divisor will generally be one LSB less than the correct result.*
- *Input operands must be of the same type (signed or unsigned) and produce a result of the same type.*
- *In division, the result of quotient is correct. But, the final result of remainder is not correct.*
- *Unsigned divisions can produce erroneous results if the divisor is greater than 0x7FFF. Dividend must be smaller than 0x7ffe8001 (7fff*ffff.).*
- *In signed divisions, the divisor cannot be a negative number (<= 0x7fff). Dividend must be smaller than 0x3FFF0001 (7fff*7fff).*

1.15 Stack Pointer Limit (SPLIM)

1.15.1 General Description

Generally, Stack Pointer Address (SPA) register is used as the last address pointer of RAM when system is at the initial state of general application. So, it is not easy to subtract the SPA register value to less than or equal to address 0x0000 of RAM. For ideal case, you should not use Stack Pointer (SP) to point the last address of RAM, or will hold most of the RAM memory for data variable use. Under this condition, the Stack Point Limit (SPLIM) is used to limit the value of SP in order to optimize memory management.

1.15.2 SPLIM Block Diagram



* *Applicable to eSL and eSLS chips*

** *Applicable to eSLZ000 ICE chip*

Figure 1-18 Stack Pointer Limit (SPLIM) Control Register Operation Block Diagram

1.15.3 Register Description

The Stack Pointer Limit (SPLIM) register, as defined in the table below; shows its default value to be the Address 0x0000 of RAM. If the value of Stack Pointer equals with SPLIM, then SPLIMIF is set, and the interrupt occurs.

SPLIM	Bit	DIR.	Description	Reset Value
SPLIM	[15:0]	R/W	The SPLIM range: 0 ~ 0x07FF (eSL and eSLS Series) 0 ~ 0x1FFF (eSLZ000 only)	0x0000

1.15.4 Operation Description

In reference to Figure 1-18 above, Cases (a) to (c) illustrate the program initial value setup before program starts. Case (d) shows the SPA setup with SPLIM operation constraint for the reason that its SPA dynamic operational range is limited (too short).

■ **Case (a)**

The first case shows a good example for keeping the SPA value at maximum which ensures value will not negatively exceeds to less than Address 0x0000 of RAM memory. The two reasons behind it are, first; the dynamic operative range of SPA is large enough for the required usage. The other is that the SPLIM will restrict the SP from exceeding the Address 0x0000 (default value). If the SPA value equals SPLIM, interrupt will occur. It alerts programmer which SPA operation is overused. However, for large SPA dynamic range as in this case, such condition rarely occurs. If for some reasons you have indeed overused the RAM memory, error will result with the SPA value and the program control flow.

■ **Case (b)**

In this case, the SPA dynamic operation range area of RAM memory is located at lower address and is separated from the area for user general usage. This arrangement is fine for RAM memory allocated to user usage, but for SPA dynamic operation range, the area is not big enough. Taken for granted that the area is still enough for SPA dynamic operation range, the area nonetheless, will not be able to accommodate the BS and BC instructions which needs RAM Address 0x0000 to 0x0007 to operate. This is because its memory management is dynamic.

■ Case (c) & (d)

The best arrangement for SPA and SPLIM is illustrated in Case (c) where you already know how much SPA dynamic operation range is needed by setting SPLIM (not SP) and used the remaining and most of the RAM memory for your general usage. However, if you provide inadequate space for SPA dynamic range (as in of Case d), SPLIM interrupt will occur frequently. Therefore, Case (c) must be used carefully and is recommended for professional programmers only. Furthermore, under Case (c), you may use BS and BC in RAM Address 0x0000 to 0x0007 as its memory management is user definable.

CAUTION !!

SP value can't equal to SPLIM value in Case (c) and (d), otherwise the data in data usable range will damage.

■ Example:

```
/************************************************
 * Set the value of SP and SPLIM as in Case(a)
************************************************/******
R0 = #0x0000
IO[SPLIM] = R0          // SPLIM = #0x0000
R0 = #0x07FF
IO[SPA] = R0            // SP = #0x07FF

/************************************************
 * Set the value of SPAR and SPLIM as in Case(b)
************************************************
R0 = #0x0000
IO[SPLIM] = R0          // SPLIM = #0x0000
R0 = #0x0010
IO[SPA] = R0            // SP = #0x0010

/************************************************
 * Set the value of SPAR and SPLIM as in Case(c)
************************************************
R0 = #0x0500
IO[SPLIM] = R0          // SPLIM = #0x0500
R0 = #0x07FF
IO[SPA] = R0            // SP = #0x07FF
/************************************************
 * Set the value of SPAR and SPLIM as in Case(d)
************************************************
R0 = #0x07F0
IO[SPLIM] = R0          // SPLIM = #0x07F0
R0 = #0x07FF
IO[SPA] = R0            // SP = #0x07FF
```

Chapter 2
Instruction Set

2.1 Introduction

The eSL Series provides powerful instructions. The numeric representation is listed in the following table. The characters are none case-sensitive.

Type	Number			
Decimal	0	1	8	15
Binary	0b0000	0b0001	0b1000	0b1111
Hexadecimal	0x0	0x1	0x8	0xF

2.1.2 Symbol Summary

■ **General Symbol**

Symbol	Description
Address Generator	Address generator for data RAM
ALU	16-bit signed/unsigned arithmetic logic unit
Multiplier	17×17 hardware multiplier
Program Counter	Program counter with 15 bits for 32K program ROM address
Peripheral Control Registers	Peripheral control registers are useful for peripheral comment, include ADC/DAC/INT….etc.
RAM	2k word internal RAM (0x0000 ~ 0x07FFF)
SR	Status register contains carry/zero/overflow… flag status
Stack	16 bits address software stacks for subroutine call and interrupt
W/C	Word / Cycle

■ **Operator**

SYMBOL	DESCRIPTION
+, -, *, / *	The four fundamental operations of arithmetic
[ ]	Point to Data RAM

* *The division instruction (/) takes 16 cycles when in fractional mode, and takes 17 cycles when in integer mode.*

■ Operand

Symbol	Description
R0~R7	General purpose register
Rd	Destination register (R0 ~ R7)
Rs	Source register (R0 ~ R7)
Rt	Second Source register (R0 ~ R7)
Rn	The number of Repeat or Loop counter (R0 ~ R7)
.b	Specific operation bit of a Word (ex: R0.15)(b:0~15)
#imm6	6 Bits immediate Data (0~63)
#imm8	8 Bits immediate Data (0~255)
#imm16	16 Bits immediate Data (0~65535)
RAM16	The value of 16 Bits RAM Direct addressing
RAM8	The value of 8 Bits RAM Direct addressing
B	$\overline{C}$, the Invert of Carrier
P[ ]	Point to Program ROM
IO[ ]	Point to I/O space
<abs ADDR>	Absolute address (16 bits)

■ Flag Status (SR)

Symbol	Description
T	Test Flag
N	Negative Flag
Z	Zero Flag
V	Overflow Flag
C	Carry Flag
+	Flag is affected by instruction operation
-	Flag is un-change by instruction operation
*****	Flag is un-defined (Don't care) by instruction operation

■ Auto-Operation Symbol Description

These symbols described herewith cannot operate as instruction.

Symbol	Description
$addr16	The value of 16 Bits RAM address (not the RAM address itself)
$addr8	The value of 8 Bits RAM address (not the RAM address itself)
Long_addr	16 bits absolute address for long jump
Short_addr	16 bits address with PC + 1 + 9#offset for short jump
L_addr	16 bit absolute address for long call
S_addr	14 bit address for short call
+, -, *, /	The four fundamental operations of arithmetic (Operator)
&, |, ^, ~	Logic operation as AND, OR, XOR, and 1's complement respectively (Operator)
==	Equal Operator
→, ←→	Direction Move Operator and swap respectively
RC	Repeat Counter
PC	Program Counter
TOS	Top of Stack
GIE	Global Interrupt Enable flag

2.2 ALU Instruction Operation

2.2.1 Introduction

The two prime concerns when defining the instruction sets are**:**

- The instruction set must be complete with no missing functionality.
- The instructions must be orthogonal, that is; not unnecessarily redundant.

The eSL Series has a 16-bit instruction set (1 or 2 words). It is organized into instruction categories grouped by function as shown in the following table. The instruction is very easy to remember and is built into your program.

2.2.2 ALU Operation Flag Alternation

Instruction	Resulting SR	Note
a + d > 65535	C / V = 1,N / Z = 0	
a + d =0	C / V / N = 0, Z = 1	0 + 0 => C = 0
65535 ≧ a + d > 0	C / Z / V / N = 0	
a - d > 0	C = 1, Z / V / N = 0	
a - d = 0	C / Z = 1, V / N = 0	0 - 0 or x - x => C = 1
a - d < 0	C / Z / V = 0, N = 1	
CMP R0, R1 > 0	C = 1, Z / V / N = 0	
CMP R0, R1 = 0	C / Z = 1, V / N = 0	0 - 0 or x - x => C = 1
CMP R0, R1 < 0	C / Z / V = 0, N = 1	
D = D + Rs * Rt > 4294967295 (MAC)	C / V = 1, Z / N = 0	
D = D + Rs * Rt = 0 (MAC)	C / V / N = 0, Z = 1	
4294967295 ≧ D = D + Rs * Rt > 0 (MAC)	C / Z / V / N = 0	
D = D - Rs * Rt < 0 (MAS)	C / Z / V = 0, N = 1	
D = D - Rs * Rt = 0 (MAS)	C / Z = 1, V / N = 0	
4294967295 ≧ D = D - Rs * Rt > 0 (MAS)	C = 1, Z / V / N = 0	

2.3 Instruction Summary

2.3.1 Assembly Instruction Set

The eSL Series Assembly Instruction Set are separated into “Data Transfer Instructions,” “Arithmetic Operation Instructions,” “Logic Operation Instructions,” “Bit Operation Instructions,” and “Program Jump Instructions.” Each of these Instructions is detailed in the following tables.

■ Data Transfer Instructions

Function	Algebra Assembly Syntax	Operation	W/C	T	N	Z	V	C
MOV	Rd=Rs	Rs→Rd	1/1	-	-	-	-	-
	Rd=[Rs]	[Rs]→Rd	1/1	-	-	-	-	-
	Rd=[Rs++]	[Rs++]→Rd	1/1	-	-	-	-	-
	Rd=[Rs--]	[Rs--]→Rd	1/1	-	-	-	-	-
	Rd=P[Rs]	P[Rs]→Rd	1/2(1)[1]	-	-	-	-	-
	Rd=P[Rs++]	P[Rs++]→Rd	1/2(1)[1]	-	-	-	-	-
	[Rd++]=P[Rs--]	P[Rs--]→[Rd++]	1/2(1)[1]	-	-	-	-	-
	[Rd++]=P[Rs++]	P[Rs++]→[Rd++]	1/2(1)[1]	-	-	-	-	-
	Rd=P[Rs--]	P[Rs--]→Rd	1/2(1)[1]	-	-	-	-	-
	[Rd]=P[Rs++]	P[Rs++]→[Rd]	1/2(1)[1]	-	-	-	-	-
	[Rd--]=P[Rs--]	P[Rs--]→[Rd--]	1/2(1)[1]	-	-	-	-	-
	[Rd]=P[Rs]	P[Rs]→[Rd]	1/2(1)[1]	-	-	-	-	-
	[Rd++]=P[Rs]	P[Rs]→[Rd++]	1/2(1)[1]	-	-	-	-	-
	[Rd]=Rs	[Rs]→[Rd]	1/1	-	-	-	-	-
	[Rd++]=Rs	Rs]→ [Rd++]	1/1	-	-	-	-	-
	[Rd--]=Rs	[Rs]→[Rd--]	1/1	-	-	-	-	-
	Rd=RAM16 (L)	$addr16 → Rd	2/2	-	-	-	-	-
	RAM16=Rs (L)	Rs→ $addr16	2/2	-	-	-	-	-
	Rd=#imm16	#imm16 → Rd	2/2	-	-	-	-	-
	[Rd]=#imm16	#imm16 → [Rd]	2/2	-	-	-	-	-
	Rd.l=#imm8	#imm8→Rd.l; 0→Rh	1/1	-	-	-	-	-
	Rd.h=#imm8	#imm8→Rd.h	1/1	-	-	-	-	-
	Rd=RAM8	$addr8 → Rd	1/1	-	-	-	-	-
	RAM8=Rs	Rs→ $addr8	1/1	-	-	-	-	-
	Rd=[R3-#imm6]	[R3-#imm6]→Rd	1/1	-	-	-	-	-
	Rd=[R3-Rt]	[R3-Rt]→Rd	1/1	-	-	-	-	-
	[R3-#imm6]=Rs	Rs→[R3-#imm6]	1/1	-	-	-	-	-
	[R3-Rt]=Rs	Rs→[R3-Rt]	1/1	-	-	-	-	-
IN	Rd = IO[Addr]	IO[Addr] → Rd	1/1	-	-	-	-	-
OUT	IO[Addr]=Rs	Rs → IO[Addr]	1/1	-	-	-	-	-
PUSH	PUSH Rn	1. Rn→ TOS 2. SP-1 → SP	1/1	-	-	-	-	-
	PUSH IO[Add]	1. I/O → TOS 2. SP-1 → SP	1/1	-	-	-	-	-
POP	POP Rn	1. SP+1 → SP 2. TOS → Rn	1/1	-	-	-	-	-
	POP IO[Addr]	1. SP+1 → SP 2. TOS → IO[Addr]	1/1	-	-	-	-	-
SP (+)	SP=SP+#imm6	SP+#imm6 → SP	1/1	-	-	-	-	-
SP (–)	SP=SP–#imm6	SP-#imm6 →SP	1/1	-	-	-	-	-

[1] *Using RPT instruction to perform this operation only needs 1 cycle*

■ Arithmetic Operation Instructions

Function	Algebra Assembly Syntax	Operation	W/C	T	N	Z	V	C
ADD	Rd = Rs+Rt	Rs+Rt → Rd	1/1	-	+	+	+	+
	[Rd] = Rs+Rt	Rs+Rt → [Rd]	1/1	-	+	+	+	+
	Rd = [Rs]+Rt	[Rs]+Rt → Rd	1/1	-	+	+	+	+
	Rd = Rd+#imm6	Rd+#imm6 → Rd	1/1	-	+	+	+	+
	Rd = Rs+#imm16	Rs+#imm16 → Rd	2/2	-	+	+	+	+
	[Rd] = Rs+#imm16	Rs+#imm16 → [Rd]	2/2	-	+	+	+	+
	Rd = Rs+RAM16	Rs+ $addr16 → Rd	2/2	-	+	+	+	+
	Rd++	Rd + 1 → Rd	1/1	-	+	+	+	+
	Rd - -	Rd – 1 → Rd	1/1	-	+	+	+	+
ADC	Rd = Rs+Rt+C	Rs+Rt+C → Rd	1/1	-	+	+	+	+
	[Rd] = Rs+Rt+C	Rs+Rt+C → [Rd]	1/1	-	+	+	+	+
	Rd = [Rs]+Rt+C	[Rs]+Rt+C → Rd	1/1	-	+	+	+	+
	Rd = Rd+#imm6+C	Rd+#imm6+C → Rd	1/1	-	+	+	+	+
	Rd = Rs+#imm16+C	Rs+#imm16+C → Rd	2/2	-	+	+	+	+
	[Rd] = Rs+#imm16+C	Rs+#imm16+C → [Rd]	2/2	-	+	+	+	+
	Rd = Rs+RAM16+C	Rs+ $addr16+C → Rd	2/2	-	+	+	+	+
SUB	Rd = Rs-Rt	Rs-Rt → Rd	1/1	-	+	+	+	+
	[Rd] = Rs-Rt	Rs-Rt → [Rd]	1/1	-	+	+	+	+
	Rd = [Rs]-Rt	[Rs]-Rt → Rd	1/1	-	+	+	+	+
	Rd = Rd-#imm6	Rd-#imm6 → Rd	1/1	-	+	+	+	+
	Rd = Rs-#imm16	Rs-#imm16 → Rd	2/2	-	+	+	+	+
	[Rd] = Rs-#imm16	Rs-#imm16 → [Rd]	2/2	-	+	+	+	+
	Rd = Rs-RAM16	Rs- $addr16 → Rd	2/2	-	+	+	+	+
SUBB	Rd = Rs-Rt-B	Rs-Rt-/C → Rd	1/1	-	+	+	+	+
	[Rd] = Rs-Rt-B	Rs-Rt-/C → [Rd]	1/1	-	+	+	+	+
	Rd = [Rs]-Rt-B	[Rs]-Rt-/C → Rd	1/1	-	+	+	+	+
	Rd = Rd-#imm6-B	Rd-#imm6-/C → Rd	1/1	-	+	+	+	+
	Rd = Rs-#imm16-B	Rs-#imm16-/C → Rd	2/2	-	+	+	+	+
	[Rd] = Rs-#imm16-B	Rs-#imm16-/C → [Rd]	2/2	-	+	+	+	+
	Rd = Rs-RAM16-B	Rs- $addr16-/C → Rd	2/2	-	+	+	+	+
MUL.SS	D=Rs*Rt(SS)	Rs.S*Rt.S → D	1/1	-	-	-	-	-
	D=Rs*[Rt] (SS)	Rs.S*[Rt].S → D	1/1	-	-	-	-	-
	D=Rs*[Rt++](SS)	Rs.S*[Rt++].S → D	1/1	-	-	-	-	-
	D=Rs*[Rt--](SS)	Rs.S*[Rt--].S → D	1/1	-	-	-	-	-
	D=Rs*P[Rt](SS)	Rs.S*P[Rt].S → D	1/2(1) [1]	-	-	-	-	-
	D=Rs*P[Rt++](SS)	Rs.S*P[Rt++].S→D	1/2(1) [1]	-	-	-	-	-
	D=[Rs++]*P[Rt--](SS)	[Rs++].S*P[Rt--].S→D	1/2(1) [1]	-	-	-	-	-
	D=[Rs++]*P[Rt++](SS)	[Rs++].S*P[Rt++].S→D	1/2(1) [1]	-	-	-	-	-

[1] *Using RPT instruction to perform this operation only needs 1 cycle*

(Continued)

Function	Algebra Assembly Syntax	Operation	W/C	T	N	Z	V	C
MUL.SU	D=Rs*Rt(SU)	Rs.S*Rt.U→ D	1/1	-	-	-	-	-
	D=Rs*[Rt] (SU)	Rs.S*[Rt].U→ D	1/1	-	-	-	-	-
	D=Rs*[Rt++](SU)	Rs.S*[Rt++].U→ D	1/1	-	-	-	-	-
	D=Rs*[Rt--](SU)	Rs.S*[Rt--].U→ D	1/1	-	-	-	-	-
	D=Rs*P[Rt](SU)	Rs.S*P[Rt].U→ D	1/2(1) [1]	-	-	-	-	-
	D=Rs*P[Rt++](SU)	Rs.S*P[Rt++].U→ D	1/2(1) [1]	-	-	-	-	-
	D=[Rs++]*P[Rt--](SU)	[Rs++].S*P[Rt--].U→ D	1/2(1) [1]	-	-	-	-	-
	D=[Rs++]*P[Rt++](SU)	[Rs++].S*P[Rt++].U→D	1/2(1) [1]	-	-	-	-	-
MUL.US	D=Rs*Rt(US)	Rs.U*Rt.S → D	1/1	-	-	-	-	-
	D=Rs*[Rt] (US)	Rs.U*[Rt].S → D	1/1	-	-	-	-	-
	D=Rs*[Rt++](US)	Rs.U*[Rt++].S → D	1/1	-	-	-	-	-
	D=Rs*[Rt--](US)	Rs.U*[Rt--].S → D	1/1	-	-	-	-	-
	D=Rs*P[Rt](US)	Rs.U*P[Rt].S → D	1/2(1) [1]	-	-	-	-	-
	D=Rs*P[Rt++](US)	Rs.U*P[Rt++].S → D	1/2(1) [1]	-	-	-	-	-
	D=[Rs++]*P[Rt--](US)	[Rs++].U*P[Rt--].S → D	1/2(1) [1]	-	-	-	-	-
	D=[Rs++]*P[Rt++](US)	[Rs++].U*P[Rt++].S→D	1/2(1) [1]	-	-	-	-	-
MUL.UU	D=Rs*Rt(UU)	Rs.U*Rt.U→ D	1/1	-	-	-	-	-
	D=Rs*[Rt] (UU)	Rs.U*[Rt].U→ D	1/1	-	-	-	-	-
	D=Rs*[Rt++](UU)	Rs.U*[Rt++].U→ D	1/1	-	-	-	-	-
	D=Rs*[Rt--](UU)	Rs.U*[Rt--].U→ D	1/1	-	-	-	-	-
	D=Rs*P[Rt](UU)	Rs.U*P[Rt].U→ D	1/2(1) [1]	-	-	-	-	-
	D=Rs*P[Rt++](UU)	Rs.U*P[Rt++].U→ D	1/2(1) [1]	-	-	-	-	-
	D=[Rs++]*P[Rt--](UU)	[Rs++].U*P[Rt--].U→D	1/2(1) [1]	-	-	-	-	-
	D=[Rs++]*P[Rt++](UU)	[Rs++].U*P[Rt++].U→D	1/2(1) [1]	-	-	-	-	-
DIV.S	D=D/Rs(S)	Signed [R1:R0] / Rs →R0 = Quotient	1/17(16) [2]	-	*	*	*	*
DIV.U	D=D/Rs	Unsigned [R1:R0] / Rs →R0 = Quotient	1/17(16) [2]	-	*	*	*	*
MAC	D=D+Rs*Rt	D+Rs*Rt → D	1/1	-	+	+	+	+
	D=D+Rs*[Rt]	D+Rs*[Rt] → D	1/1	-	+	+	+	+
	D=D+Rs*[Rt++]	D+Rs*[Rt++] → D	1/1	-	+	+	+	+
	D=D+Rs*[Rt--]	D+Rs*[Rt--] → D	1/1	-	+	+	+	+
	D=D+Rs*P[Rt]	D+Rs*P[Rt] → D	1/2(1) [1]	-	+	+	+	+
	D=D+Rs*P[Rt++]	D+Rs*P[Rt++] → D	1/2(1) [1]	-	+	+	+	+
	D=D+[Rs++]*P[Rt--]	D+[Rs++]*P[Rt--] → D	1/2(1) [1]	-	+	+	+	+
	D=D+[Rs++]*P[Rt++]	D+[Rs++]*P[Rt++] → D	1/2(1) [1]	-	+	+	+	+
MAS	D=D-Rs*Rt	D-Rs*Rt → D	1/1	-	+	+	+	+
	D=D-Rs*[Rt]	D-Rs*[Rt] → D	1/1	-	+	+	+	+
	D=D-Rs*[Rt++]	D-Rs*[Rt++] → D	1/1	-	+	+	+	+
	D=D-Rs*[Rt--]	D-Rs*[Rt--] → D	1/1	-	+	+	+	+
	D=D-Rs*P[Rt]	D-Rs*P[Rt] → D	1/2(1) [1]	-	+	+	+	+
	D=D-Rs*P[Rt++]	D-Rs*P[Rt++] → D	1/2(1) [1]	-	+	+	+	+
	D=D-[Rs++]*P[Rt--]	D-[Rs++]*P[Rt--] → D	1/2(1) [1]	-	+	+	+	+
	D=D-[Rs++]*P[Rt++]	D-[Rs++]*P[Rt++] → D	1/2(1) [1]	-	+	+	+	+

[1] *Using RPT instruction to perform this operation only needs 1 cycle*

[2] *The division instruction (1/17(16)) needs 17 cycles in integer mode and 16 cycles is needed in fractional mode.*

■ Logic Operation Instructions

Function	Algebra Assembly Syntax	Operation	W/C	T	N	Z	V	C
AND	Rd = Rs AND Rt	Rs&Rt → Rd	1/1	–	+	+	*	*
	[Rd] = Rs AND Rt	Rs&Rt → [Rd]	1/1	–	+	+	*	*
	Rd = [Rs] AND Rt	[Rs]&Rt → Rd	1/1	–	+	+	*	*
	Rd = Rd AND #imm6	Rd&#imm6 → Rd	1/1	–	+	+	*	*
	Rd = Rs AND #imm16	Rs&#imm16 → Rd	2/2	–	+	+	*	*
	[Rd] = Rs AND #imm16	Rs&#imm16 → [Rd]	2/2	–	+	+	*	*
	Rd = Rs AND RAM16	Rs& $addr16 → Rd	2/2	–	+	+	*	*
OR	Rd = Rs OR Rt	Rs | Rt → Rd	1/1	–	+	+	*	*
	[Rd] = Rs OR Rt	Rs | Rt → [Rd]	1/1	–	+	+	*	*
	Rd = [Rs] OR Rt	[Rs] | Rt → Rd	1/1	–	+	+	*	*
	Rd = Rd OR #imm6	Rd | #imm6 → Rd	1/1	–	+	+	*	*
	Rd = Rs OR #imm16	Rs | #imm16 → Rd	2/2	–	+	+	*	*
	[Rd] = Rs OR #imm16	Rs | #imm16 → [Rd]	2/2	–	+	+	*	*
	Rd = Rs OR RAM16	Rs | $addr16 → Rd	2/2	–	+	+	*	*
XOR	Rd = Rs XOR Rt	Rs^Rt → Rd	1/1	–	+	+	*	*
	[Rd] = Rs XOR Rt	Rs^Rt → [Rd]	1/1	–	+	+	*	*
	Rd = [Rs] XOR Rt	[Rs]^Rt → Rd	1/1	–	+	+	*	*
	Rd = Rd XOR #imm6	Rd^#imm6 → Rd	1/1	–	+	+	*	*
	Rd = Rs XOR #imm16	Rs^#imm16 → Rd	2/2	-	+	+	*	*
	[Rd] = Rs XOR #imm16	Rs^#imm16 → [Rd]	2/2	-	+	+	*	*
	Rd = Rs XOR RAM16	Rs^$addr16 → Rd	2/2	-	+	+	*	*
ROL	Rd=ROL Rs	1.Rs → Rd 2.C→Rd[0] Rd[14:0]→Rd[15:1] Rd[15]→C	1/1	-	+	+	+	+
SHL	Rd=SHL Rs	1.Rs → Rd 2. 0→Rd[0] Rd[14:0]→Rd[15:1] Rd[15] → C	1/1	-	+	+	+	+
ROR	Rd=ROR Rs	1.Rs → Rd 2.C→Rd[15] Rd[15:1]→Rd[14:0] Rd[0]→C	1/1	-	+	+	+	+
SHR	Rd=SHR Rs	1.Rs → Rd 2.0→Rd[15] Rd[15:1]→Rd[14:0] Rd[0] → C	1/1	-	+	+	+	+
ASR	Rd=ASR Rs	1.Rs → Rd 2. Rd.15→Rd[15] Rd[15:1]→Rd[14:0] Rd[0]→C	1/1	-	+	+	+	+
COM	Rd=COM Rs	~Rs → Rd	1/1	-	+	+	+	+
NEG	Rd=NEG Rs	~Rs+1 → Rd	1/1	-	+	+	+	+
SWAP	SWAP Rs	Rs[15:8] ←→ Rs[7:0]	1/1	-	–	–	–	–

(Continued)

Function	Algebra Assembly Syntax	Operation	W/C	T	N	Z	V	C
CMP	CMP Rs,Rt	Rs–Rt, update SR register	1/1	-	+	+	+	+
	CMP Rs,[Rt]	Rs–[Rt], update SR register	1/1	-	+	+	+	+
	CMP Rs,[Rt++]	Rs–[Rt++], update SR register	1/1	-	+	+	+	+
	CMP Rs,[Rt--]	Rs–[Rt--], update SR register	1/1	-	+	+	+	+
	CMP Rs,P[Rt]	Rs–P[Rt], update SR register	1/2(1) [1]	-	+	+	+	+
	CMP Rs,P[Rt++]	Rs–P[Rt++], update SR register	1/2(1) [1]	-	+	+	+	+
	CMP [Rs++],P[Rt--]	[Rs++]–P[Rt--], update SR register	1/2(1) [1]	-	+	+	+	+
	CMP [Rs++],P[Rt++]	[Rs++]–P[Rt++], update SR register	1/2(1) [1]	-	+	+	+	+

[1] *Using RPT instruction to perform this operation only needs 1 cycle*

■ Bit Operation Instructions*

Function	Algebra Assembly Syntax	Operation	W/C	T	N	Z	V	C
BS	BS Rd.b	1→ Rd[bit b]	1/1	-	-	-	-	-
	BS IO[IO_addr].b	1→ IO_addr[bit b]	1/1	-	-	-	-	-
	BS RAM_addr.b	1→ RAM_addr[bit b]	1/1	-	-	-	-	-
BC	BC Rd.b	0→ Rd[bit b]	1/1	-	-	-	-	-
	BC IO[IO_addr].b	0→ IO_addr[bit b]	1/1	-	-	-	-	-
	BC RAM_addr.b	0→ RAM_addr[bit b]	1/1	-	-	-	-	-
BTG	BTG Rd.b	~ Rd.b → Rd.b	1/1	-	-	-	-	-
	BTG IO[IO_addr].b	~ IO_addr.b → IO_addr.b	1/1	-	-	-	-	-
	BTG RAM_addr.b	~RAM_addr.b →RAM_addr.b	1/1	-	-	-	-	-
BTEST	BTEST Rd.b	Test Rd.b→Test Flag	1/1	+	-	-	-	-
	BTEST IO[IO_addr].b	Test IO_addr.b→Test Flag	1/1	+	-	-	-	-
	BTEST RAM_addr.b	Test RAM_addr.b →Test Flag	1/1	+	-	-	-	-

* ***IO_addr:*** *0x00~0x0F*

RAM_addr: *0x0000~0x0007*

■ Program Jump Instructions

Function	Algebra Assembly Syntax	Operation	W/C	T	N	Z	V	C
CALL	CALL Rd	1. SP -1 → SP 2. PC+1 → TOS 3. Rd→ PC	1/2	-	-	-	-	-
	CALL Long_addr	Call absolute address	2/2	-	-	-	-	-
	CALL Short_addr	PC← 00nnnnnnnnnnnn	1/2	-	-	-	-	-
RET	RET	Return from subroutine	1/2	-	-	-	-	-
RETI	RETI	Return from interrupt	1/2	-	-	-	-	-
RPT[1]	RPT Rn	Repeat next inst (Rn+1) times	1/1	-	-	-	-	-
	RPT #imm6	Repeat next inst (#imm6+1) times	1/1	-	-	-	-	-
LOOP	LOOP Rn .ENDL	Do loop to .ENDL, (Rn+1) times	2/2	-	-	-	-	-
	LOOP #imm6 .ENDL	Do loop to .ENDL, (#imm6+1) times	2/2	-	-	-	-	-
JMP	JMP Rs	Rd→ PC	1/2	-	-	-	-	-
	JMP Long_addr	Long_addr→PC	2/2	-	-	-	-	-
	JMP Short_addr	PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JCS	IF CS JMP Long_addr	If C==1, Long_add→PC	2/2	-	-	-	-	-
	IF CS JMP Short_addr	If C==1, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JCC	IF CC JMP Long_addr	If C==0, Long_addr→PC	2/2	-	-	-	-	-
	IF CC JMP Short_addr	If C==0, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JVS	IF VS JMP Long_addr	If V==1, Long_addr→PC	2/2	-	-	-	-	-
	IF VS JMP Short_addr	If V==1, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JVC	IF VC JMP Long_addr	If V==0, Long_addr→PC	2/2	-	-	-	-	-
	IF VC JMP Short_addr	If V==0, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JMI	IF MI JMP Long_addr	If N==1, Long_addr→PC	2/2	-	-	-	-	-
	IF MI JMP Short_addr	If N==1, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-

[1] *These multi-cycle instructions (Word /Cycle 1/2(1)) become a single-cycle instructions after the first iteration of a repeat "RPT" instruction*

[2] *The short jump instruction (1/1(2)) needs two cycles if jump is carried out and one cycle is needed if no jump is carried out.*

(Continued)

Function	Algebra Assembly Syntax	Operation	W/C	T	N	Z	V	C
JPL	IF PL JMP Long_addr	If N==0, Long_addr→PC	2/2	-	-	-	-	-
	IF PL JMP Short_addr	If N==0, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JTS	IF TS JMP Long_addr	If T==1, Long_addr→PC	2/2	-	-	-	-	-
	IF TS JMP Short_addr	If T==1, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JTC	IF TC JMP Long_addr	If T==0, Long_addr→PC	2/2	-	-	-	-	-
	IF TC JMP Short_addr	If T==0, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JGT	IF GT JMP Long_addr	If [Z | (N^V) == 0], Long_addr→PC	2/2	-	-	-	-	-
	IF GT JMP Short_addr	If [Z | (N^V) == 0], PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JGE	IF GE JMP Long_addr	If [(N ^ V) == 0], Long_addr→PC	2/2	-	-	-	-	-
	IF GE JMP Short_addr	If [(N ^ V) == 0], PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JHS	IF HS JMP Long_addr	If C==1, Long_addr→PC	2/2	-	-	-	-	-
	IF HS JMP Short_addr	If C==1, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JEQ	IF EQ JMP Long_addr	If Z==1, Long_addr→PC	2/2	-	-	-	-	-
	IF EQ JMP Short_addr	If Z==1, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JNE	IF NE JMP Long_addr	If Z== 0, Long_addr→PC	2/2	-	-	-	-	-
	IF NE JMP Short_addr	If Z == 0, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JLE	IF LE JMP Long_addr	If [Z | (N^V) == 1], Long_addr→PC	2/2	-	-	-	-	-
	IF LE JMP Short_addr	If [Z | (N^V) == 1], PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-
JLO	IF LO JMP Long_addr	If C==0, Long_addr→PC	2/2	-	-	-	-	-
	IF LO JMP Short_addr	If C==0, PC+1+Offset→PC	1/1(2)[2]	-	-	-	-	-

[2] *The short jump instruction (1/1(2)) needs two cycles if jump is carried out and one cycle is needed if no jump is carried out.*

(Continued)

Function	Algebra Assembly Syntax	Operation	W/C	T	N	Z	V	C
JLS	IF LS JMP Long_add	If [(C == 0) | (Z == 1)], Long_addr→PC	2/2	-	-	-	-	-
	IF LS JMP Short_add	If [(C == 0) | (Z == 1)], PC+1+Offset→PC	1/1(2) [2]	-	-	-	-	-
JLT	IF LT JMP Long_add	If [(N ^ V) == 1], Long_addr→PC	2/2	-	-	-	-	-
	IF LT JMP Short_add	If [(N ^ V) == 1], PC+1+Offset→PC	1/1(2) [2]	-	-	-	-	-
TRAP	TRAP #imm6	1.PC+1 → TOS 2.SP-1 → SP 3.(#imm6 vector number*2) → PC 4.GIE → 0	1/2	-	-	-	-	-
NOP	NOP	No operation	1/1	-	-	-	-	-

[2] *The short jump instruction (1/1(2)) needs two cycles if jump is carried out and one cycle is needed if no jump is carried out.*

2.4 Assembly Instruction Descriptions

2.4.1 ADC – Addition with Carry

Syntax		
Algebra Mode	**Operation**	**W/C**
[Rd]=Rs+Rt+C	Rs+Rt+C → [Rd]	1/1
Rd=[Rs]+Rt+C	[Rs]+Rt+C → Rd	1/1
Rd=Rd+#imm6+C	Rd+#imm6+C → Rd	1/1
Rd=Rs+#imm16+C	Rs+#imm16+C → Rd	2/2
[Rd]=Rs+#imm16+C	Rs+#imm16+C → [Rd]	2/2
Rd=Rs+RAM16+C	Rs+$addr16+C → Rd	2/2
Description		
Adds the two operands and the Carry flag using binary addition. The first operand is the destination register into which the result is stored.		
Flags Affected		
N Z V C		

Example:

```
CARRIER   .EQU   0
;-------------------------------------------------
R0 = #0X0000
R1 = #0X031F
BS IO[SR].CARRIER
R2=R1+R0+C                      ; R2 : #0X0320
BS IO[SR].CARRIER
[R0]=R1+R0+C                    ; [R0] : #0X0320
BS IO[SR].CARRIER
R3=[R0]+R0+C                    ; R3 : #0X0321
BS IO[SR].CARRIER
R1=R1+#0x20+C                   ; R1 : #0X0340
BS IO[SR].CARRIER
R4=R3+#0x1000+C                 ; R4 : #0X1322
BS IO[SR].CARRIER
[R0]=R3+#0x1000+C               ; [R0] : #0X1322
BS IO[SR].CARRIER
[R0] = R0
R5=R4+0x0000+C                  ; R5 : #0X1323
```

2.4.2 ADD – Addition without Carry

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd=Rs+Rt+C	Rs+Rt+C → Rd	1/1
[Rd]=Rs+Rt+C	Rs+Rt+C → [Rd]	1/1
Rd=[Rs]+Rt+C	[Rs]+Rt+C → Rd	1/1
Rd=Rd+#imm6+C	Rd+#imm6+C → Rd	1/1
Rd=Rs+#imm16+C	Rs+#imm16+C → Rd	2/2
[Rd]=Rs+#imm16+C	Rs+#imm16+C → [Rd]	2/2
Rd=Rs+RAM16+C	Rs+$addr16+C → Rd	2/2
Description		
Adds the two operands using binary addition. The first operand is the destination register into which the result is stored.		
Flags Affected		
N Z V C		

Examples:

```
R0 = #0X0000
R1 = #0X031F
R2 = R1+R0                  ; R2 = #0X031F
[R0] = R1+R0                ; [R0] : #0X031F
R3 = [R0]+R0                ; R3 = #0X031F
R1 = R1+#0x20               ; R1 = #0X033F
R4 = R3+#0x1000             ; R4 = #0X131F
[R0] = R3+#0x1000           ; R0 = #0X0000,   [R0] : #0X131F
R1 = #0x0000
R5 = R0 + [R1]              ; R5=: #0X131F
```

2.4.3 AND – Logical AND

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd=Rs AND Rt	Rs&Rt → Rd	1/1
[Rd]=Rs AND Rt	Rs&Rt → [Rd]	1/1
Rd=[Rs] AND Rt	[Rs]&Rt → Rd	1/1
Rd=Rd AND #imm6	Rd&#imm6 → Rd	1/1
Rd =Rs AND #imm16	Rs&#imm16 → Rd	2/2
[Rd]=Rs AND #imm16	Rs&#imm16 → [Rd]	2/2
Rd = Rs AND RAM16	Rs&$addr16→ Rd	2/2
Description		
Performs a logical AND between the two operands. The first operand is the destination register into which the result is stored.		
Flags Affected		
N Z		

Example:

```
R0 = #0X00FF
R1 = #0X031F
R2=R1 AND R0              ; R2 : #0X001F
[R0]=R1 AND R0            ; [R0] : #0X001F
R3=[R0] AND R0            ; R3 : #0X001F
R1=R1 AND #0x20           ; R1 : #0X0000
R4=R3 AND #0x0011         ; R4 : #0X0011
R0= #0
[R0]=R3 AND #0x0011       ; [R0] : #0X0011
R0=#0x0011
R1 = #0x0000
R5= R0 AND [R1]           ; R5 : #0X0011
```

2.4.4 ASR – Arithmetic Shift Right

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd = ASR Rs	1.Rs → Rd 2.Rd.15→Rd[15] Rd[15:1]→Rd[14:0] Rd[0]→C	1/1
Description		
Performs an arithmetic right shift of one position of the source operand. The result is stored in the destination register. The result is sign extended: the most significant bit of the result is filled with the most significant bit of the operand. The Carry flag takes the value of the least significant bit of the operand.		
Flags Affected		
N Z V C		

Example:

```
R1 = #0X8001
R0 = ASR R1              ; R0 : #0XC000, N = 1, C = 1
```

2.4.5 BC – Bit Clear

Syntax		
Algebra Mode	**Operation**	**W/C**
BC Rd.b	0→ Rd[bit b]	1/1
BC IO[IO_addr].b	0→ IO_addr[bit b]	1/1
BC RAM_addr.b	0→ RAM_addr[bit b]	1/1
Description		
Performs AND between the content of destination address and corresponding bits in bit mask field. (1) RAM address only supports 0x0000~0x0007 for BC instruction. (2) IO address only supports 0x0000~0x000F for BC instruction.		
Flags Affected		
None		

Example:

```
R0 = #0x031F
IO[PDIRA] = R0
R0 = #0x00FF
0x0000 = R0
R1 = #0x031F
BC R1.4                  ; R1 : #0X030F
BC IO[PDIRA].1           ; IO[PDIRA] : #0X031D
BC 0x0000.4              ; [0x00] : #0X00EF
```

2.4.6 BS – Bit Set

Syntax		
Algebra Mode	**Operation**	**W/C**
BS Rd.b	1→ Rd[bit b]	1/1
BS IO[IO_addr].b	1→ IO_addr[bit b]	1/1
BS RAM_addr.b	1→ RAM_addr[bit b]	1/1
Description		
Performs OR between the content of destination address and corresponding bits in bit mask field. (1) RAM address only supports 0x0000~0x0007 for BS instruction. (2) IO address only supports 0x0000~0x000F for BS instruction.		
Flags Affected		
None		

Example:

```
R0 = #0x031D
IO[PDIRA] = R0
R0 = #0x00EF
0x0000 = R0
R1 = #0X030F
BS R1.4                    ; R1 : #0X031F
BS IO[PORTA].1             ; IO[PORTA] : #0X031F
BS 0x0000.4                ; [0x00] : #0X00FF
```

2.4.7 BTEST – Bit Test

Syntax		
Algebra Mode	**Operation**	**W/C**
BTEST Rd.b	Test Rd.b → Test Flag	1/1
BTEST IO[IO_addr].b	Test IO_addr.b → Test Flag	1/1
BTEST.RAM_addr.b	Test RAM_addr.b → Test Flag	1/1
Description		
Performs AND between the content of destination address and corresponding bits in bit mask field. (1) RAM address only supports 0x0000~0x0007 for BTEST instruction. (2) IO address only supports 0x0000~0x000F for BTEST instruction.		
Flags Affected		
T		

Example:

```
R1 = #0X030F
BEGIN:
BTEST R1.0          ; test R1[0] is 1 or 0
IF TS JMP BEGIN     ; R1.0=1, so program will jump to BEGIN address
```

2.4.8 BTG – Bit Toggle

Syntax		
Algebra Mode	**Operation**	**W/C**
BTG Rd.b	~ Rd.b → Rd.b	1/1
BTG IO[IO_addr].b	~ IO_addr.b → IO_add.b	1/1
BTG.RAM_addr.b	~ RAM_addr.b → RAM_add.b	1/1
Description		
Performs XOR between the content of destination address and corresponding bits in bit mask field. (1) RAM address only supports 0x0000~0x0007 for BTEST instruction. (2) IO address only supports 0x0000~0x000F for BTEST instruction.		
Flags Affected		
None		

Example:

```
R0 = #0x031D
IO[PDIRA] = R0
R0 = #0x00EF
0x0000 = R0

R1 = #0X030F
BTG R1.4                    ; R1 : #0X031F
BTG IO[PDIRA].1             ; IO[PDIRA] : #0X031F
BTG 0x0000.4                ; [0x00] : #0X00FF
```

2.4.9 CALL – Jump to Subroutine

Syntax		
Algebra Mode	**Operation**	**W/C**
CALL Rd	1. SP -1 → SP 2. PC+1 → TOS 3. Rd→ PC	1/2
CALL L_addr	1. SP -1 → SP 2. PC+1 → TOS 3. 16 bit address→ PC	2/2
CALL S_addr	1. SP -1 → SP 2. PC+1 → TOS 3. {00,14 bit address}→ PC	1/2
Description		
Jumps to the subroutine whose address is given in the operand. The program memory address of the instruction immediately follows the CALL instruction and is pushed on top of stack.		
Flags Affected		
None		

Example:

```
R2=#TB1
CALL R2
R4=R0                    ; R4 :  #0X0001
CALL TB4
R3=R0                    ; R3 :  #0X0001
CALL TB2
R2=R0                    ; R2 :  #0X0001
TB1:
R0=#0x0001
RET
TB2:
;STATEMENT
RET
TB4:
;STATEMENT
RET
```

2.4.10 CMP – Compare

Syntax		
Algebra Mode	**Operation**	**W/C**
CMP Rs,Rt	Rs–Rt, update SR register	1/1
CMP Rs,[Rt]	Rs–[Rt], update SR register	1/1
CMP Rs,[Rt++]	Rs–[Rt++], update SR register	1/1
CMP Rs,[Rt--]	Rs–[Rt--], update SR register	1/1
CMP Rs,P[Rt]	Rs–P[Rt] , update SR register	1/2(1)
CMP Rs,P[Rt++]	Rs–P[Rt++], update SR register	1/2(1)
CMP [Rs++],P[Rt--]	[Rs++]–P[Rt--], update SR register	1/2(1)
CMP [Rs++],P[Rt++]	[Rs++]–P[Rt++], update SR register	1/2(1)
Description		
This instruction performs a compare between Registers Rd and Rs (Rd-Rs). None of the registers is changed. Only the flags are affected.		
Flags Affected		
N Z V C		

Example:

```
;INITIAL_VALUE :
; P[R6] : 0x031F
;-----------------------------
R1 = #0X031F
R0 = R1
CMP R0,R1               ; R0=R1, Z flag set
R2 = IO[SR]
R2 = R2 AND #0x0004     ; R2 : #0X0004
R0 = #0
CMP  R1,[R0]            ; [R0]=R1, Z flag set
R2 = IO[SR]
R2 = R2 AND #0x0004     ; R2 : #0X0004
R0 = #0
CMP R1,[R0 ++]          ; [R0]=R1, Z flag set, R0=R0+#0x01
R2 = IO[SR]
R2 = R2 AND #0x0004     ; R2 : #0X0004
R0 = #0
CMP R1,P[R6]
R2 = IO[SR]
R2 = R2 AND #0x0004     ; R2 : #0X0004
CMP R1,P[R6++]          ; R6=R6+#0x01
R2 = IO[SR]
R2 = R2 AND #0x0004     ; R2 : #0X0004
```

2.4.11 COM – One's Complement

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd = COM Rs	~Rs → Rd	1/1
Description		
Performs the One's Complement of source register then store the result into destination register.		
Flags Affected		
N Z V C		

Example:

```
R1 = #0X5555
R0 = COM R1              ; R0 = ~R1 = #0XAAAA.
```

2.4.12 DIV.S – Signed Division

Syntax		
Algebra Mode	**Operation**	**W/C**
D=D/Rs(S)	Signed [R1:R0] / Rs → R0 = Quotient	1/17(16)
Description		
Performs an extended 32-bit by 16-bit division of the two words stored in the R1 and R0 (accumulator D) registers by the source word operand Rs for signed division. The quotient is then stored in the low order word of the R0 register. R2 is limited in #1 ~ #0x7FFF		
Flags Affected		
None		

NOTE

- *eSL Series hardware can NOT check division overflow and division with zero divisor. You must preclude these conditions through software manipulation.*
- *The quotient produced by a division with a negative divisor will generally be one LSB less than the correct result.*
- *Input operands must be of the same type (signed or unsigned) and produce a result of the same type.*
- *In division, the result of quotient is correct. But, the final result of remainder is not correct.*
- *In signed Divisions, the divisor cannot be a negative number (<= 0x7fff). Dividend must be smaller than 0x3FFF0001 (7fff*7fff).*

Example:

```
R0 = #0X0001
R1 = #0X3FFF
R2 = #0x7FFF
D = D/R2 (S)             ; R0 : #0X7FFF
```

2.4.13 DIV.U – Unsigned Division

Syntax		
Algebra Mode	**Operation**	**W/C**
D=D/Rs	Unsigned [R1:R0] / Rs → R0 = Quotient	1/17(16)
Description		
Performs an extended 32-bit by 16-bit division of the two words stored in the R1 and R0 (accumulator D) registers by the source word operand Rs for unsigned division. The quotient is then stored in the low order word of the R0 register		
Flags Affected		
None		

NOTE

- *eSL Series hardware can NOT check division overflow and division with zero divisor. You must preclude these conditions through software manipulation.*
- *The quotient produced by a division with a negative divisor will generally be one LSB less than the correct result.*
- *Input operands must be of the same type (signed or unsigned) and produce a result of the same type.*
- *In division, the result of quotient is correct. But, the final result of remainder is not correct.*
- *Unsigned Divisions can produce erroneous results if the divisor is greater than 0x7FFF. Dividend must be smaller than 0x7ffe8001 (7fff*ffff).*

Example:

```
R0 = #0x8001
R1 = #0x7FFE
R2 = #0x7FFF;
D = D/R2                    ; R0=0xFFFF
```

2.4.14 IN – Load an I/O Space to Register

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd = IO[Addr]	IO[Add] → Rd	1/1
Description		
Loads data from the I/O space into register Rd.		
Flags Affected		
None		

Example:

```
R0 = #0x031F
IO[PDIRA] = R0
R1=IO[PDIRA]              ; R1 = #0X031F
```

2.4.15 JCC –Jump Upon Condition C Flag Clear

Syntax		
Algebra Mode	**Operation**	**W/C**
IF CC JMP Long_addr	If C==0, Long_addr→PC	2/2
IF CC JMP Short_addr	If C==0, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by C == 0 is true.		
Flags Affected		
None		

Example:

```
R2 = #0x00F0
R2 = SHR R2          ; CARRY CLEAR
IF CC JMP TB4        ; JUMP TB4
TB4:
R3 =R2               ; R3 :  #0x0078
R2 = SHR R2          ; CARRY CLEAR
IF CC JMP TB2        ; JUMP TB2
TB2:
R4 = R2              ; R4 :  #0x003C
```

2.4.16 JCS – Jump Upon Condition C Flag Set

Syntax		
Algebra Mode	**Operation**	**W/C**
IF CS JMP Long_addr	If C==1, Long_addr→PC	2/2
IF CS JMP Short_addr	If C==1, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by C == 1 is true.		
Flags Affected		
None		

Example:

```
R2 = #0x000F
R2 = SHR R2          ; CARRY SET
IF CS JMP TB4        ; JUMP TB4, OUT OF 16KBYTE ADDRESS
TB4:
R3 = R2              ; R3 :  #0X0007
R2 = SHR R2          ; CARRY SET
IF CS JMP TB2        ; JUMP TB2, INSIDE 16KBYTE ADDRESS
TB2:
R4 = R2              ; R4 :  #0X0003
```

2.4.17 JEQ – Jump Upon Condition Z Flag Set

Syntax		
Algebra Mode	**Operation**	**W/C**
IF EQ JMP Long_addr	If Z==1, Long_addr→PC	2/2
IF EQ JMP Short_addr	If Z==1, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by Z == 1 is true.		
Flags Affected		
None		

Example:

```
R2 = #0x031F
R7 = #0x031F
CMP R2,R7           ; ZERO SET
IF EQ JMP TB4       ; JUMP TB4
NOP
NOP
TB4:
R3 = R0
```

2.4.18 JGE – Jump Upon Condition Greater Than or EQUAL

Syntax		
Algebra Mode	**Operation**	**W/C**
IF GE JMP Long_addr	If [(N ^ V) == 0], Long_addr→PC	2/2
IF GE JMP Short_addr	If [(N ^ V) == 0], PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by (N ^ V) == 0 is true.		
Flags Affected		
None		

Example:

```
R2 = #0x031F
R7 = #0x031E
CMP R2,R7          ; (N^V) == 0
IF GE JMP TB4      ; JUMP TB4
NOP
NOP
TB4:
R2 = #0x031F
R7 = #0x031F
CMP R2,R7          ; (N^V) == 0
IF GE JMP TB2      ; JUMP TB2
NOP
NOP
TB2:
```

2.4.19 JGT – Jump Upon Condition Greater Than

Syntax		
Algebra Mode	**Operation**	**W/C**
IF GT JMP Long_addr	If [Z | (N^V) == 0], Long_addr→PC	2/2
IF GT JMP Short_addr	If [Z | (N^V) == 0], PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by Z | (N^V) == 0 is true.		
Flags Affected		
None		

Example:

```
R2 = #0x031F
R7 = #0x031E
CMP R2,R7           ; Z|(N^V) == 0
IF GT JMP TB4       ; JUMP TB4
NOP
NOP
TB4:
R2 = #0x031E
R7 = #0x031E
CMP R2,R7           ; Z|(N^V) == 0
IF GT JMP TB2       ; NO JUMP
NOP
NOP
TB2:
```

2.4.20 JHS – Jump Upon Condition Higher or Same

Syntax		
Algebra Mode	**Operation**	**W/C**
IF HS JMP Long_addr	If C==1, Long_addr→PC	2/2
IF HS JMP Short_addr	If C==1, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by C == 1 is true. This is same with JCS (Section 2.4.16).		
Flags Affected		
None		

Example:

```
R2 = #0x031F
R7 = #0x031E
CMP R2,R7          ; CARRIER SET
IF HS JMP TB4      ; JUMP TB4
NOP
NOP
TB4:
R2 = #0x031F
R7 = #0x031F
CMP R2,R7
IF HS JMP TB2      ; JUMP TB2
NOP
NOP
TB2:
```

2.4.21 JLE – Jump Upon Condition Less Than or Equal

Syntax		
Algebra Mode	**Operation**	**W/C**
IF LE JMP Long_addr	If [Z | (N^V) == 1], Long_addr→PC	2/2
IF LE JMP Short_addr	If [Z | (N^V) == 1], PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by Z | (N^V) == 1 is true.		
Flags Affected		
None		

Example:

```
R2 = #0x031F
R7 = #0x031E
CMP R2,R7           ; Z|(N^V) == 1
IF LE JMP TB4       ; JUMP TB4
NOP
NOP
TB4:
R2 = #0x031F
R7 = #0x031F
CMP R2,R7           ; Z|(N^V) == 1
IF LE JMP TB2       ; JUMP TB2
NOP
NOP
TB2:
```

2.4.22 JLO – Jump upon Condition Lower Than

Syntax		
Algebra Mode	**Operation**	**W/C**
IF LO JMP Long_addr	If C==0, Long_addr→PC	2/2
IF LO JMP Short_addr	If C==0, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by C == 0 is true. This is same with JCC (Section 2.4.15)		
Flags Affected		
None		

Example:

```
R2 = #0x031F
R7 = #0x031E
CMP R7,R2          ; CARRIER CLEAR
IF LO JMP TB4      ; JUMP TB4
NOP
NOP
TB4:
```

2.4.23 JLS – Jump upon Condition Lower or Same

Syntax		
Algebra Mode	**Operation**	**W/C**
IF LS JMP Long_addr	If [(C == 0) | (Z == 1)], Long_addr→PC	2/2
IF LS JMP Short_addr	If [(C == 0) | (Z == 1)], PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by (C == 0) | (Z == 1) is true.		
Flags Affected		
None		

Example:

```
R2 = #0x031F
R7 = #0x031E
CMP R7,R2          ; CARRY CLEAR OR ZERO SET
IF LS JMP TB4      ; J UMP TB4
NOP
NOP
TB4:
R2 = #0x031F
R7 = #0x031F
CMP R7,R2          ; CARRY CLEAR OR ZERO SET
IF LS JMP TB2      ; JUMP TB2
NOP
NOP
TB2:
```

2.4.24 JLT – Jump Upon Condition Less Than

Syntax		
Algebra Mode	**Operation**	**W/C**
IF LT JMP Long_addr	If [(N ^ V) == 1], Long_addr→PC	2/2
IF LT JMP Short_addr	If [(N ^ V) == 1], PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by (N ^ V) == 1 is true.		
Flags Affected		
None		

Example:

```
R2 = #0x031F
R7 = #0x031E
CMP R7,R2           ; (N^V) == 1
IF LT JMP TB4       ; JUMP TB4
NOP
NOP
TB4:
```

2.4.25 JMI – Jump Upon Condition N Flag Set

Syntax		
Algebra Mode	**Operation**	**W/C**
IF MI JMP Long_addr	If N==1, Long_addr→PC	2/2
IF MI JMP Short_addr	If N==1, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by N == 1 is true.		
Flags Affected		
None		

Example:

```
R2 = #0x031F
R7 = #0x031E
R1 = R7-R2          ; N=1
IF MI JMP TB4       ; JUMP TB4
NOP
NOP
TB4:
```

2.4.26 JMP – Jump Always

Syntax		
Algebra Mode	**Operation**	**W/C**
JMP Rs	Rd→ PC	1/2
JMP Long_addr	Long_addr→PC	2/2
JMP Short_addr	PC+1+Offset→PC	1/1(2)
Description		
Unconditionally jumps to the specified address.		
Flags Affected		
None		

Example:

```
BEGIN:
R0 = #TB1
JMP R0
NOP
NOP
TB1:
NOP
JMP TB2
NOP
NOP
TB2:
```

2.4.27 JNE – Jump Upon Condition Z Flag Clear

Syntax		
Algebra Mode	**Operation**	**W/C**
IF NE JMP Long_addr	If Z== 0, Long_addr→PC	2/2
IF NE JMP Short_addr	If Z == 0, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by Z == 0 is true.		
Flags Affected		
None		

Example:

```
R2=#0x031F
R7=#0x031E
CMP R2,R7          ; ZERO CLEAR
IF NE JMP TB4      ; JUMP TB4
NOP
NOP
TB4:
```

2.4.28 JPL – Jump Upon Condition N Flag Clear

Syntax		
Algebra Mode	**Operation**	**W/C**
IF PL JMP Long_addr	If N==0, Long_addr→PC	2/2
IF PL JMP Short_addr	If N==0, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by N == 0 is true.		
Flags Affected		
None		

Example:

```
R2=#0x031F
R7=#0x031E
R1=R2-R7            ; NEGATIVE FLAG CLEAR
IF PL JMP TB4       ; JUMP TB4, OUT OF 16KBYTE ADDRESS
NOP
NOP
TB4:
```

2.4.29 JTC – Jump Upon Condition T Flag Clear

Syntax		
Algebra Mode	**Operation**	**W/C**
IF TC JMP Long_addr	If T==0, Long_addr→PC	2/2
IF TC JMP Short_addr	If T==0, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by T == 0 is true.		
Flags Affected		
None		

Example:

```
R0=#0
R2=#0x031F
BTEST R0.5          ; TEST T FLAG
IF TC JMP TB4       ; JUMP TB4
NOP
NOP
TB4:
BTEST R2.6          ; TEST T FLAG
IF TC JMP TB2       ; JUMP TB2NOP
NOP
TB2:
```

2.4.30 JTS – Jump Upon Condition T Flag Set

Syntax		
Algebra Mode	**Operation**	**W/C**
IF TS JMP Long_addr	If T==1, Long_addr→PC	2/2
IF TS JMP Short_addr	If T==1, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by T == 1 is true.		
Flags Affected		
None		

Example:

```
R0=#0
R2=#0x031F
BTEST R2.1          ; TEST T FLAG
IF TS JMP TB4       ; JUMP TB4
NOP
NOP
TB4:
```

2.4.31 JVC – Jump Upon Condition V Flag Clear

Syntax		
Algebra Mode	**Operation**	**W/C**
IF VC JMP Long_addr	If V==0, Long_addr→PC	2/2
IF VC JMP Short_addr	If V==0, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by V == 0 is true.		
Flags Affected		
None		

Example:

```
R0=#0
R2=#0x2FFF
R2=R2+R2            ; OVERFLOW FLAG CLEAR
IF VC JMP TB4       ; JUMP TB4, OUT OF 16KBYTE
NOP
NOP
TB4:
R2=#0x2FFF
R2=R2+R2            ; OVERFLOW FLAG CLEAR
IF VC JMP TB2       ; JUMP TB2
NOP
NOP
TB4:
```

2.4.32 JVS – Jump Upon Condition V Flag SET

Syntax		
Algebra Mode	**Operation**	**W/C**
IF VS JMP Long_addr	If V==1, Long_addr→PC	2/2
IF VS JMP Short_addr	If V==1, PC+1+Offset→PC	1/1(2)
Description		
Conditionally jumps to the specified address if the state defined by V == 1 is true.		
Flags Affected		
None		

Example:

```
R2 = #0xAFFF
R2 = R2+R2          ; OVERFLOW FLAG CLEAR
IF VS JMP TB4       ; JUMP TB4, OUT OF 16KBYTE
NOP
NOP
TB4:
R2 = #0xFFFF
R2 = R2+R2          ; OVERFLOW FLAG CLEAR
IF VS JMP TB2       ; NO JUMP
NOP
NOP
TB2:
```

2.4.33 LOOP – Initial Hardware Loop

Syntax		
Algebra Mode	**Operation**	**W/C**
LOOP Rn .ENDL	Rn →LC PC + 1→LSA End_loop→LEA	2/2
LOOP #imm6 .ENDL	#imm6→LC PC + 1→LSA End_loop→LEA	2/2
Description		
Initiate a zero-overhead hardware loop which is executed “Rn” (the content of register) times or “#imm6” times. When this instruction is executed, the loop start address register (LSA) is loaded with PC + 1 and the loop end address register (LEA) is loaded with the LSB 16-bit address.		
Flags Affected		
None		

Example:

```
R0 = #0x60
R1 = #0x3F
LOOP R0
  R1++
R0 = R1              ; R0 = #0xA0
NOP
ENDL
LOOP #0x3F
R1++
R0 = R1              ; R0 = #0xE0
NOP
ENDL
```

2.4.34 MAC – 16X16-Bit Signed-Signed Multiply and Accumulate with D

Syntax		
Algebra Mode	**Operation**	**W/C**
D=D+Rs*Rt	D+Rs*Rt → D	1/1
D=D+Rs*[Rt]	D+Rs*[Rt] → D	1/1
D=D+Rs*[Rt++]	D+Rs*[Rt++] → D	1/1
D=D+Rs*[Rt--]	D+Rs*[Rt--] → D	1/1
D=D+Rs*P[Rt]	D+Rs*P[Rt] → D	1/2(1)
D=D+Rs*P[Rt++]	D+Rs*P[Rt++] → D	1/2(1)
D=D+[Rs++]*P[Rt--]	D+[Rs++]*P[Rt--] → D	1/2(1)
D=D+[Rs++]*P[Rt++]	D+[Rs++]*P[Rt++] → D	1/2(1)
Description		
Multiply the two operands using 16x16-bit signed-signed and add with D. Then store the 32-bit result into accumulator D [R1:R0].		
Flags Affected		
N Z V C		

Example:

```
R1 = #0x0000
R2 = #0x031F
R3 = #0x0003
R4 = #0x0000
R0 = #0x0100
[R4] = R2
R4++
[R4] = R2
R4++
R7 = #0x5163
[R4] = R7
R4++
R7 = #0x8342
[R4] = R7
R4 = #0x0000
D=D+R2*R3                          ; R0 : #0X0A5D
R0=#0x0100
D=D+R3*[R4]                        ; R0 : #0X0A5D
R0=#0x0100
D=D+R3*[R4++]                      ; R0 : #0X0A5D
R0=#0x0100
D=D+R3*[R4--]                      ; R0 : #0X0A5D
R0=#0x0100
R2=#TESTDATA
D=D+R3*P[R2]                       ; R0 : #0X0A5D
R0=#0x0100
R2=#TESTDATA
D=D+R3*P[R2++]                     ; R0 : #0X0A5D
R0=#0x0100
R3 = #0x0001
R2 = #TESTDATA
D=D+[R3++]*P[R2++]                 ; R0 : #0xBEC1
R0=#0x0100
D=D+[R3++]*P[R2--]                 ; R0 : #0X04FD

TESTDATA  .DW
0x031F, 0x031F, 0x5163, 0x8342
```

2.4.35 MAS – 16X16-Bit Signed-Signed Multiply and Subtract with D

Syntax		
Algebra Mode	**Operation**	**W/C**
D=D–Rs*Rt	D–Rs*Rt → D	1/1
D=D–Rs*[Rt]	D–Rs*[Rt] → D	1/1
D=D–Rs*[Rt++]	D–Rs*[Rt++] → D	1/1
D=D–Rs*[Rt--]	D–Rs*[Rt--] → D	1/1
D=D–Rs*P[Rt]	D–Rs*P[Rt] → D	1/2(1)
D=D–Rs*P[Rt++]	D–Rs*P[Rt++] → D	1/2(1)
D=D– [Rs++]*P[Rt--]	D– [Rs++]*P[Rt--] → D	1/2(1)
D=D– [Rs++]*P[Rt++]	D– [Rs++]*P[Rt++] → D	1/2(1)
Description		
Multiply the two operands using 16x16-bit signed-signed and subtract with D. Then store the 32-bit result into accumulator D [R1:R0].		
Flags Affected		
N Z V C		

Example:

```
R1 = #0x0000
R2 = #0x031F
R3 = #0x0003
R4 = #0x0000
R0 = #0x1000
[R4] = R2
R4++
[R4] = R2
R4++
R7 = #0x5163
[R4] = R7
R4++
R7 = #0x8342
[R4] = R7
R4 = #0x0000

D=D-R2*R3                    ; R0 : #0X06A3
R0=#0x1000
D=D-R3*[R4]                  ; R0 : #0X06A3
R0=#0x1000
D=D-R3*[R4++]                ; R0 : #0X06A3
R0=#0x1000
D=D-R3*[R4--]                ; R0 : #0X06A3
R0=#0x1000
R2=#TESTDATA
D=D-R3*P[R2]                 ; R0 : #0X06A3
R0=#0x1000
R2=#TESTDATA
```

```
D=D-R3*P[R2++]                          ; R0 : #0X06A3
R0=#0x0100
R3 = #0x0001
R2 = #TESTDATA
D=D+[R3++]*P[R2++]                      ; R0 : #0xBEC1
R0=#0x0100
D=D+[R3++]*P[R2--]                      ; R0 : #0X04FD

TESTDATA:
.DW     0x031F, 0x031F, 0x5163, 0x8342
```

2.4.36 MOV – Data Move

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd=Rs	Rs→Rd	1/1
Rd=[Rs]	[Rs]→Rd	1/1
Rd=[Rs++]	[Rs++]→Rd	1/1
Rd=[Rs--]	[Rs--]→Rd	1/1
Rd=P[Rs]	P[Rs]→Rd	1/2(1)
Rd=P[Rs++]	P[Rs++]→Rd	1/2(1)
[Rd++]=P[Rs--]	P[Rs--]→[Rd++]	1/2(1)
[Rd++]=P[Rs++]	P[Rs++]→[Rd++]	1/2(1)
Rd=P[Rs--]	P[Rs--]→Rd	1/2(1)
[Rd]=P[Rs++]	P[Rs++]→[Rd]	1/2(1)
[Rd--]=P[Rs--]	P[Rs--]→[Rd--]	1/2(1)
[Rd]=P[Rs]	P[Rs]→[Rd]	1/2(1)
[Rd++]=P[Rs]	P[Rs]→[Rd++]	1/2(1)
[Rd]=Rs	[Rs]→[Rd]	1/1
[Rd++]=Rs	Rs]→ [Rd++]	1/1
[Rd--]=Rs	[Rs]→[Rd--]	1/1
Rd=RAM16 (L)	$addr16 → Rd	2/2
RAM16=Rs (L)	Rs→ $addr16	2/2
[Rd]=#imm16	#imm16 → [Rd]	2/2
Rd.l=#imm8	#imm8→Rd.l; 0→Rh	1/1
Rd.h=#imm8	#imm8→Rd.h	1/1
Rd=RAM8	$addr8 → Rd	1/1
RAM8=Rs	Rs→ $addr8	1/1
Rd=[R3-#imm6]	[R3-#imm6]→Rd	1/1
Rd=[R3-Rt]	[R3-Rt]→Rd	1/1
[R3-#imm6]=Rs	Rs→[R3-#imm6]	1/1
[R3-Rt]=Rs	Rs→[R3-Rt]	1/1
Description		
Moves 8-bit or 16-bit value from the source to the destination. (RHS → LHS) Direct move from the data memory to the data memory is NOT possible.		
Flags Affected		
None		

Example:

```
.CODE
R0 = #0x031F
0x0000 = R0
0x1000 = R0 (L)
R0 = #0X0000
R1 = #0X031F
R2 = #TESTDATA
R3=R1                ; R3: #0x031F
R3=[R0]              ; R3: #0x031F
R3=[R0++]            ; R3: #0x031F
R0--
R3=[R0--]            ; R3: #0x031F
R3=P[R2]             ; R3: #0x031F
R3=P[R2++]           ; R3: #0x031F
R3=P[R2--]           ; R3: #0x031F
R0++
[R0++]=P[R2++]       ; RAM[R0] : #0x031F
R3=P[R2--]           ; R3: #0x031F
[R0]=P[R2++]         ; RAM[R0] : #0x031F
[R0--]=P[R2--]       ; RAM[R0] : #0x031F
[R0]=P[R2]           ; RAM[R0] : #0x031F
[R0++]=P[R2]         ; RAM[R0] : #0x031F
[R0++]=R1            ; RAM[R0] : #0x031F
[R0--]=R1            ; RAM[R0] : #0x031F
R3=0x1000 (L)        ; R3: #0x031F
0x1000=R1 (L)        ; RAM(L): #0x031F
[R0]=#0x031F         ; RAM[R0] : #0x031F
R3.l=#0x1F           ; R3: #0x001F
R3.h=#0x03           ; R3: #0x031F
R3=0x0000            ; R3: #0x031F
0x0000=R1            ; RAM(S): #0x031F
R3=#0x001F
R3=[R3-#0x1F]        ; R3: #0x031F
R3=[R3-R1]           ; R3: #0x031F
[R3-#0x1F]=R1        ; RAM[R3-#imm6] : #0x031F
[R3-R1]=R1           ; RAM[R3-R1]: #0x031F

TESTDATA:
.DW 0x031F,0x031F
```

2.4.37 MUL.SS – 16X16-Bit Signed-Signed Multiply

Syntax		
Algebra Mode	**Operation**	**W/C**
D=Rs*Rt(SS)	Rs.S*Rt.S → D	1/1
D=Rs*[Rt] (SS)	Rs.S*[Rt].S → D	1/1
D=Rs*[Rt++](SS)	Rs.S*[Rt++].S → D	1/1
D=Rs*[Rt--](SS)	Rs.S*[Rt--].S → D	1/1
D=Rs*P[Rt](SS)	Rs.S*P[Rt].S → D	1/2(1)
D=Rs*P[Rt++](SS)	Rs.S*P[Rt++].S → D	1/2(1)
D=[Rs++]*P[Rt--](SS)	[Rs++].S*P[Rt--].S → D	1/2(1)
D=[Rs++]*P[Rt++](SS)	[Rs++].S*P[Rt++].S → D	1/2(1)
Description		
Multiply the two operands using 16x16-bit signed-signed. Then store the 32-bit result into accumulator D [R1:R0].		
Flags Affected		
None		

Example:

```
R0 = #0x0000
R1 = #0xFFDE
[R0] = R1
R0++
[R0] = R1
R2 = #0XFFDE
R3 = # MULSSDATA
D = R2*R2(SS)                ; D[R1,R0] : [#0X0000,#0X0484]
R4 = #0
D = R2*[R4](SS)              ; D[R1,R0] : [#0X0000,#0X0484]
R4 = #0
D = R2*[R4++](SS)            ; D[R1,R0] : [#0X0000,#0X0484]
R4 = #0
D = R2*[R4--](SS)            ; D[R1,R0] : [#0X0000,#0X0484]
D = R2*P[R3](SS)             ; D[R1,R0] : [#0X0000,#0X0484]
D = R2*P[R3++](SS)           ; D[R1,R0] : [#0X0000,#0X0484]
R4 = #0
D = [R4++]*P[R3--](SS)       ; D[R1,R0] : [#0X0000,#0X0484]
D = [R4++]*P[R3++](SS)       ; D[R1,R0] : [#0X0000,#0X0484]

MULSSDATA:
.DW 0XFFDE, 0XFFDE, 0XFFDE, 0XFFDE
```

2.4.38 MUL.SU – 16X16-Bit Signed-Unsigned Multiply

Syntax		
Algebra Mode	**Operation**	**W/C**
D=Rs*Rt(SU)	Rs.S*Rt.U→ D	1/1
D=Rs*[Rt] (SU)	Rs.S*[Rt].U→ D	1/1
D=Rs*[Rt++](SU)	Rs.S*[Rt++].U→ D	1/1
D=Rs*[Rt--](SU)	Rs.S*[Rt--].U→ D	1/1
D=Rs*P[Rt](SU)	Rs.S*P[Rt].U→ D	1/2(1)
D=Rs*P[Rt++](SU)	Rs.S*P[Rt++].U→ D	1/2(1)
D=[Rs++]*P[Rt--](SU)	[Rs++].S*P[Rt--].U→ D	1/2(1)
D=[Rs++]*P[Rt++](SU)	[Rs++].S*P[Rt++].U→ D	1/2(1)
Description		
Multiply the two operands using 16x16-bit signed-unsigned. Then store the 32-bit result into accumulator D [R1:R0].		
Flags Affected		
None		

Example:

```
R0 = #0x0000
R1 = #0x0022
[R0] = R1
R0 ++
[R0] = R1
R2 = #0XFFDE
R3 = #0
R5 = #0X0022
D = R2*R5(SU)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
R4 = #0
D = R2*[R4](SU)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
D = R2*[R4++](SU)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
D = R2*[R4--](SU)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
R3 = #MULDATA
D = R2*P[R3](SU)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
D = R2*P[R3++](SU)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
R0 = #0xFFDE
R1 = #0
[R1++] = R0
[R1--] = R0
D = [R4++]*P[R3--](SU)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
D = [R4++]*P[R3++](SU)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
MULDATA:
.DW 0X0022, 0X0022, 0X0022, 0X0022
```

2.4.39 MUL.US – 16X16-Bit Unsigned-Signed Multiply

Syntax		
Algebra Mode	**Operation**	**W/C**
D=Rs*Rt(US)	Rs.U*Rt.S → D	1/1
D=Rs*[Rt] (US)	Rs.U*[Rt].S → D	1/1
D=Rs*[Rt++](US)	Rs.U*[Rt++].S → D	1/1
D=Rs*[Rt--](US)	Rs.U*[Rt--].S → D	1/1
D=Rs*P[Rt](US)	Rs.U*P[Rt].S → D	1/2(1)
D=Rs*P[Rt++](US)	Rs.U*P[Rt++].S → D	1/2(1)
D=[Rs++]*P[Rt--](US)	[Rs++].U*P[Rt--].S → D	1/2(1)
D=[Rs++]*P[Rt++](US)	[Rs++].U*P[Rt++].S → D	1/2(1)
Description		
Multiply the two operands using 16x16-bit unsigned-signed. Then store the 32-bit result into accumulator D [R1:R0].		
Flags Affected		
None		

Example:

```
R0 = #0x0000
R1 = #0xFFDE
[R0] = R1
R0 ++
[R0] = R1
R2 = #0X0022
R3 = #0
R5 = #0XFFDE
D = R2*R5(US)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
R4 = #0
D = R2*[R4](US)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
D = R2*[R4++](US)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
D = R2*[R4--](US)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
R3 = #MULDATA
D = R2*P[R3](US)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
D = R2*P[R3++](US)
R0 = NEG R0                    ; D[R1,R0] : [#0XFFFF,#0X0484]
R0 = #0x0022
R1 = #0
[R1++] = R0
[R1--] = R0
D = [R4++]*P[R3--](US)         ; D[R1,R0] : [#0XFFFF,#0X0484]
R0 = NEG R0
D = [R4++]*P[R3++](US)         ; D[R1,R0] : [#0XFFFF,#0X0484]
R0 = NEG R0

MULDATA:
.DW 0XFFDE, 0XFFDE, 0XFFDE, 0XFFDE
```

2.4.40 MUL.UU – 16X16-bit Unsigned-Unsigned Multiply

Syntax		
Algebra Mode	**Operation**	**W/C**
D=Rs*Rt(UU)	Rs.U*Rt.U→ D	1/1
D=Rs*[Rt] (UU)	Rs.U*[Rt].U→ D	1/1
D=Rs*[Rt++](UU)	Rs.U*[Rt++].U→ D	1/1
D=Rs*[Rt--](UU)	Rs.U*[Rt--].U→ D	1/1
D=Rs*P[Rt](UU)	Rs.U*P[Rt].U→ D	1/2(1)
D=Rs*P[Rt++](UU)	Rs.U*P[Rt++].U→ D	1/2(1)
D=[Rs++]*P[Rt--](UU)	[Rs++].U*P[Rt--].U→ D	1/2(1)
D=[Rs++]*P[Rt++](UU)	[Rs++].U*P[Rt++].U → D	1/2(1)
Description		
Multiply the two operands using 16x16-bit unsigned-unsigned. Then store the 32-bit result into accumulator D [R1:R0].		
Flags Affected		
None		

Example:

```
R0 = #0x0000
R1 = #0x0022
[R0] = R1
R0 ++
[R0] = R1
R2 = #0X0022
R3 = #0
D=R2*R2(UU)                 ; D[R1,R0] : [#0X0000,#0X0484]
R4=#0
D=R2*[R4](UU)               ; D[R1,R0] : [#0X0000,#0X0484]
D=R2*[R4++](UU)             ; D[R1,R0] : [#0X0000,#0X0484]
D=R2*[R4--](UU)             ; D[R1,R0] : [#0X0000,#0X0484]
R3 = #MULDATA
D=R2*P[R3](UU)              ; D[R1,R0] : [#0X0000,#0X0484]
D=R2*P[R3++](UU)            ; D[R1,R0] : [#0X0000,#0X0484]
D=[R4++]*P[R3--](UU)        ; D[R1,R0] : [#0X0000,#0X0484]
D=[R4++]*P[R3++](UU)        ; D[R1,R0] : [#0X0000,#0X0484]

MULDATA:
.DW 0X0022, 0X0022, 0X0022, 0X0022
```

2.4.41 NEG – Two's Complement

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd=NEG Rs	~Rs+1 → Rd	1/1
Description		
Performs the two's complement then store the result in destination register.		
Flags Affected		
N Z V C		

Example:

```
R0 = #0x031F
R1 = NEG R0          ; R1 : #0XFCE1
```

2.4.42 NOP – No Operation

Syntax		
Algebra Mode	**Operation**	**W/C**
NOP	PC+1→ PC	1/1
Description		
No operation.		
Flags Affected		
None		

Example:

```
R0 = #5
LOOP R0
  NOP
  NOP
  NOP
.ENDL
```

2.4.43 OR – Logical OR

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd=Rs OR Rt	Rs | Rt → Rd	1/1
[Rd]=Rs OR Rt	Rs | Rt → [Rd]	1/1
Rd=[Rs] OR Rt	[Rs] | Rt → Rd	1/1
Rd=Rd OR #imm6	Rd | #imm6 → Rd	1/1
Rd=Rs OR #imm16	Rs | #imm16 → Rd	2/2
[Rd]=Rs OR #imm16	Rs | #imm16 → [Rd]	2/2
Rd=Rs OR RAM16	Rs | $addr16 → Rd	2/2
Description		
Performs a logical OR between the two operands. The first operand is the destination register into which the result is stored		
Flags Affected		
N Z		

Example:

```
.DATA
TEMPDATA    .DS    1
.CODE
R0 = #0X00FF
R1 = #0X031F
R2 = R1 OR R0              ; R2 : #0X03FF
[R0] = R1 OR R0            ; [R0] : #0X03FF
R3 = [R0] OR R0            ; R3 : #0X03FF
R1 = R1 OR #0x20           ; R1 : #0X033F
R4 = R3 OR #0x0011         ; R4 : #0X03FF
R0 = #0x0000
[R0] = R3 OR #0x0011       ; [R0] : #0X03FF
R5 = R0 OR TEMPDATA        ; R5 : #0X03FF
```

2.4.44 OUT – Store Register to I/O Location

Syntax		
Algebra Mode	**Operation**	**W/C**
IO[Addr]=Rs	Rs → IO[Addr]	1/1
Description		
Output data to the I/O space.		
Flags Affected		
None		

Example:

```
R0 = #0x031F
IO[PDIRA] = R0             ; IO[PDIRA] : 0X031F
```

2.4.45 POP – Pop Register or I/O from Stack

Syntax		
Algebra Mode	**Operation**	**W/C**
POP Rn	1. SP+1 → SP 2. TOS → Rn	1/1
POP IO[Add]	1. SP+1 → SP 2. TOS → IO[Add]	1/1
Description		
Retrieves the 16-bit value from the top of the stack and stores it into the register or I/O.		
Flags Affected		
None		

Example:

```
R0 = #0x031F
R1 = #0x031E
PUSH R0
PUSH R1
POP R0              ; R0 : #0X031E
POP IO[PDIRA]       ; IO[PDIRA] : #0X031F
```

2.4.46 PUSH – Push Register or I/O onto Stack

Syntax		
Algebra Mode	**Operation**	**W/C**
PUSH Rn	1. Rn→ TOS 2. SP-1 → SP	1/1
PUSH IO[Add]	1. I/O → TOS 2. SP-1 → SP	1/1
Description		
Stores the value of the register or the content of I/O address on the top of the hardware stack.		
Flags Affected		
None		

Example:

```
R0 = #0x031E
IO[PDIRA] = R0
R0 = #0x1000
IO[SPA] = R0
R0= #0X031F
PUSH  R0            ; [0X00FF] : #0X031F
PUSH  IO[PDIRA]     ; [0X00FE] : #0X031E
```

2.4.47 RET – Return from Subroutine

Syntax		
Algebra Mode	**Operation**	**W/C**
RET	1. Top of stack → PC 2. SP+1→SP	1/2
Description		
Returns from execution of a subroutine. The returned address is popped from the top of stack and stored into the program counter.		
Flags Affected		
None		

Example:

See example for CALL instruction in Section 2.4.9.

2.4.48 RETI – Return from Interrupt

Syntax		
Algebra Mode	**Operation**	**W/C**
RETI	1. Top of stack → PC 2. SP+1→SP 3. 0→ GIE	1/2
Description		
Returns from an interrupt subroutine execution. The returned address is popped from the top of the stack and stored into the Program Counter.		
Flags Affected		
None		

Example:

```
.Vector
LJMP POWERON          ; 0x00 0x01
NOP                   ; 0x02
NOP                   ; 0x03
NOP                   ; 0x04
NOP                   ; 0x05
LJMP ExternalINT0     ; 0x06 0x07

.CODE
POWERON:
; STATEMENT
JMP POWERON
// Interrupt sub routine
ExternalINT0:
; STATEMENT
RETI
```

2.4.49 ROL – Rotate Left through Carry

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd=ROL Rs	1. Rs → Rd 2. C→Rd[0] Rd[14:0]→Rd[15:1] Rd[15]→C	1/1
Description		
Rotates the source operand position one-step to the left. The result is stored in the destination register. The Carry flag takes the value of the most significant bit of the operand.		
Flags Affected		
N Z V C		

Example:

```
R0 = #0x001F
R3 = R0
R2 = IO[SR]
LOOP #0x000F
R0 = SHL R0
R2 = IO[SR]
R2 = R2 XOR R1
.ENDL
IO[SR] = R2
R0 = ROL R3          ; R0 : #0X003F
```

2.4.50 ROR – Rotate Right through Carry

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd=ROR Rs	1. Rs → Rd 2. C→Rd[15] Rd[15:1]→Rd[14:0] Rd[0]→C	1/1
Description		
Rotates the source operand position one-step to the right. The result is stored in the destination register. The Carry flag takes the value of the most significant bit of the operand.		
Flags Affected		
N Z V C		

Example:

```
R0 = #0x001F
R3=SHL R0
R2=IO[SR]
LOOP #0x000F
R0=SHL R0
R2=IO[SR]
R2=R2 XOR R1
.ENDL
IO[SR]=R2
R0=ROR R3            ; R0 : #0X801F
```

2.4.51 RPT – Repeat Next Instruction

Syntax		
Algebra Mode	**Operation**	**W/C**
RPT Rn	Rn→RC	1/1
RPT #imm6	#imm6→RC	1/1
Description		
The repeat counter (RC) is loaded with the number of immediate data or with the content of register when this instruction is executed.		
Flags Affected		
None		

Example:

```
R0 = #0x60
R1 = #0x3F
RPT R0
R1++
R0 = R1                ; R0 = #0xA0
RPT #0x3F
R1++
R0 = R1                ; R0 = #0xE0
```

2.4.52 SHL – Logical Shift Left with Carry

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd = SHL Rs	1. Rs → Rd 2. 0→Rd[0] Rd[14:0]→Rd[15:1] Rd[15] → C	1/1
Description		
Shift the source operand position one-step to logical left. The result is stored in the destination register. The least significant bit of the result is cleared. The Carry flag takes the value of the most significant bit of the operand.		
Flags Affected		
N Z V C		

Example:

```
R0 = #0X031F
R2=IO[SR]
LOOP #0x000F
R0=SHL R0
R1=IO[SR]
R2 = R2 XOR R1
.ENDL
R2 = R2 AND #0x0001   ; R2 : #0X0001
```

2.4.53 SHR – Logical Shift Right with Carry

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd = SHR Rs	1. Rs → Rd 2. 0→Rd[15] Rd[15:1]→Rd[14:0] Rd[0] → C	1/1
Description		
Shift the source operand position one-step to logical right. The result is stored in the destination register. The least significant bit of the result is cleared. The Carry flag takes the value of the most significant bit of the operand.		
Flags Affected		
N Z V C		

Example:

```
R0 : #0X003B
R1=R0
LOOP #0x0006
R1=SHR R1
R0= R0 XOR R1
NOP                    ; There must be at least 3 instructions
.ENDL
R0=R0 AND #0x0001      ; R0 : #0X0001
```

2.4.54 SP (–) – Stack Point (Decrease)

Syntax		
Algebra Mode	**Operation**	**W/C**
SP=SP–#imm6	SP–#imm6→SP	1/1
Description		
Subtract stack pointer (SP) and 6-bit immediate data, then store result in stack pointer address (SPA) register.		
Flags Affected		
None		

Example:

```
;-------------------------------;
;INITIAL_VALUE
;SP : #0X000F
;-------------------------------;
SP=SP-#0x000F
R0=IO[SPA]             ; R0 : #0X0000
```

2.4.55 SP (+) – Stack Point (Increase)

Syntax		
Algebra Mode	**Operation**	**W/C**
SP=SP+#imm6	SP+#imm6→SP	1/1
Description		
Add stack pointer (SP) and 6-bit immediate data, then store result in stack pointer address (SPA) register.		
Flags Affected		
None		

Example:

```
;------------------------------;
;INITIAL_VALUE
;SP : #0X00EF
;------------------------------;
SP = SP+#0x10
R0 = IO[0x11]          ; R0 = #0X00FF
```

2.4.56 SUB – Subtract without Borrow

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd=Rs-Rt	Rs-Rt → Rd	1/1
[Rd]=Rs-Rt	Rs-Rt → [Rd]	1/1
Rd=[Rs]-Rt	[Rs]-Rt → Rd	1/1
Rd=Rd-#imm6	Rd-#imm6 → Rd	1/1
Rd=Rs-#imm16	Rs-#imm16 → Rd	2/2
[Rd]=Rs-#imm16	Rs-#imm16 → [Rd]	2/2
Rd=Rs-RAM16	Rs - $addr16 → Rd	2/2
Description		
Subtracts the Operand 2 (Rt) from the Operand 1 (Rs). The first operand (Rd) is the destination register into which the result is stored.		
Flags Affected		
N Z V C		

Example:

```
.DATA
TEMPDATA    .DS    1
.CODE
R0 = #0X0000
R1 = #0X031F
R2 = R1-R0             ; R2 : #0X031F
[R0] = R1-R0           ; [R0] : #0X031F
R3 = [R0]-R0           ; R3 : #0X031F
R1 = R1-#0x0020        ; R1 : #0X02FF
R4 = R3-#0x0100        ; R4 : #0X021F
[R0] = R3-#0x0100      ; [R0] : #0X021F
R5 = R0-TEMPDATA       ; R5 : #0XFDE1
```

2.4.57 SUBB – Subtract with Borrow

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd=Rs-Rt-B	Rs-Rt-/C → Rd	1/1
[Rd]=Rs-Rt-B	Rs-Rt-/C → [Rd]	1/1
Rd=[Rs]-Rt-B	[Rs]-Rt-/C → Rd	1/1
Rd=Rd-#imm6-B	Rd-#imm6-/C → Rd	1/1
Rd=Rs-#imm16-B	Rs-#imm16-/C → Rd	2/2
[Rd]=Rs-#imm16-B	Rs-#imm16-/C → [Rd]	2/2
Rd=Rs-RAM16-B	Rs- $addr16-/C → Rd	2/2
Description		
Subtracts the two operands and the Borrow flag using binary addition. The first operand is the destination register into which the result is stored.		
Flags Affected		
N Z V C		

Example:

```
.DATA
TEMPDATA    .DS    1
.CODE
R0 = #0X0000
R1 = #0X031F
BC IO[SR].0
R2 = R1-R0-B          ; R2 =#0X031E
BC IO[SR].0
[R0] = R1-R0-B        ; [R0] : #0X031E
BC IO[SR].0
R3 = [R0]-R0-B        ; R3 =: #0X031D
BC IO[SR].0
R1 = R1-#0x0020-B     ; R1 = #0X02FE
BC IO[SR].0
R4 = R3-#0x0100-B     ; R4 = #0X021C
BC IO[SR].0
[R0] = R3-#0x0100-B   ; [R0] = #0X021C
BC IO[SR].0
R5 = R0-TEMPDATA-B    ; R5 = #0XFDE3
```

2.4.58 SWAP – SWAP

Syntax		
Algebra Mode	**Operation**	**W/C**
SWAP Rs	Rs[15:8] ←→Rs[7:0]	1/1
Description		
Performs the Rs's high byte interchange with Rs's low byte.		
Flags Affected		
None		

Example:

```
R0 = #0x1234
SWAP R0               ; R0 = #0X3412
```

2.4.59 TRAP – Call Interrupt Service Routine

Syntax		
Algebra Mode	**Operation**	**W/C**
TRAP #imm6	1. PC+1→TOS 2. SP–1→SP 3. (#imm6 vector number*2) →PC 4. GIE→0	1/2
Description		
Call interrupt service routine via immediate trap number (imm6). This instruction allows you to use your program to execute any interrupt service routine.		
Flags Affected		
None		

Example:

```
.VECTOR
JMP START              ; 0x00
NOP                    ; 0x01
NOP                    ; 0x07
JMP T0INT              ; 0x08 0x09
JMP T1INT              ; 0x0A 0x0B
.CODE
START:
R0 = #0x031F
R1 = #0xFCE1
;TRAP #imm6
TRAP #0x03
TRAP #0x04
JMP  START
;-------------------------------------------------;
; FINAL _VALUE :                                  ;
; R2 : #0X0000                                    ;
; OPERATION : CALL T0INT AND T1INT SERVICE ROUTINE ;
;-------------------------------------------------;
T0INT:
R2 = R0 OR R1
RET
T1INT:
R2 = R2+#0x0001
RET
```

2.4.60 XOR – Logical XOR

Syntax		
Algebra Mode	**Operation**	**W/C**
Rd=Rs XOR Rt	Rs^Rt → Rd	1/1
[Rd]=Rs XOR Rt	Rs^Rt → [Rd]	1/1
Rd=[Rs] XOR Rt	[Rs]^Rt → Rd	1/1
Rd=Rd XOR #imm6	Rd^#imm6 → Rd	1/1
Rd=Rs XOR #imm16	Rs^#imm16 → Rd	2/2
[Rd]=Rs XOR #imm16	Rs^#imm16 → [Rd]	2/2
Rd=Rs XOR RAM16	Rs^ $addr16 → Rd	2/2
Description		
Performs a logical XOR between the two operands. The first operand is the destination register into which the result is stored.		
Flags Affected		
N Z		

Example:

```
TEMPDATA   .DS   1
;-----------------------------------------------------
R0 = #0X00FF
R1 = #0X031F
R2 = R1 XOR R0           ; R2 : #0X03E0
[R0] = R1 XOR R0         ; [R0] : #0X03E0
R3 = [R0] XOR R0         ; R3 : #0X031F
R1 = R1 XOR #0x20        ; R1 : #0X033F
R4 = R3 XOR #0x0011      ; R4 : #0X030E
[R1] = R3 XOR #0x0011    ; [R1] : #0X030E
R7 = #0
TEMPDATA = R7
R5 = R0 XOR TEMPDATA     ; R5 : #0X00FF
```

2.5 Code Optimization Examples

2.5.1 Optimizing Continuous Shift Operation

The following examples show how the continuous shift method performance (left column) could be improved by implementing the multiplication method (right column). However, there are cases where this optimization method may not be possible. Hence you need to compare the execution performance of both methods.

Example 1

```
R0 = #0x0FF0                    R0 = #0x0FF0
RPT  #0x0003        ->          R2 = #0x0010
R0 = SHL R0                     D  = R0 * R2 (UU)
```

Example 2

```
R1 = #0x0FF0                    R1 = #0x0FF0
RPT  #0x0007        ->          R2 = #0x0100
R1 = SHR R1                     D  = R1 * R2 (UU)
```

2.5.2 Optimizing Divisor of n Power of 2 (2n)

If R2 divisor is not a fixed number and R2 (divisor) is 2^n, then multiplication method could improve its execution performance. However, there are cases where this optimization method may not be possible. Hence you need to compare the execution performance of both methods.

Example

```
R0 = #0xF000                    R0 = #0xF000
R1 = #0x000F                    R1 = #0x000F
R2 = #0x0100        ->          R2 = #0x0100
D  = D/ R2                      R5 = R1

                                D  = R0 * R2 (UU)
                                R4 = R1
                                D  = R5 * R2 (UU)

                                R0 = R0 OR R4
```

2.5.3 Optimizing Multi-Level Loop Operation

During multi-level loop operation, apply LOOP/.ENDL method to the innermost loop level only and implement at conditional branch to the rest of the loop levels. This will significantly improve system performance.

Where there is only a single level loop, the LOOP/.ENDL operation is also recommended.

Example

```
R0 = #0x0002
R1 = #0x0002
R2 = #0x0000

LOOP R0
PUSH IO[LCR]
PUSH IO[LSA]
PUSH IO[LEA]
LOOP R1
R2 = R2 + #1
R2 = R2 + #1
R2 = R2 + #1
.ENDL
POP IO[LEA]
POP IO[LSA]
POP IO[LCR]
NOP
NOP
NOP
.ENDL
```

```
R3 = #0x0001
R0 = #0x0002
R1 = #0x0002
R2 = #0x0000
R0 ++
LOOP_EXAMPLE:

LOOP R1
R2 = R2 + R3
R2 = R2 + R3
R2 = R2 + R3
.ENDL
R0 = R0 - R3
IF NE JMP
LOOP_EXAMPLE
```

2.5.4 Optimizing Variables in the Same Algorithms

All the variables in the same algorithms should be stored in BANK A to reduce PROM access and machine execution cycles, as well as improve performance.

Example:

```
R0 = #0
IO[BSR]=R0
ALGO:
.....
```

2.5.5 Optimizing Usage of Offset Numbers

When using offset numbers such as 0xFFFF, 0x7FFF, and 0x8000, these offset numbers should be stored into variables under Bank A.

Example:

```
R0 = #0
IO[BSR] = R0
0x0000 = #0xFFFF      ; 2  clk
0x0001 = #0x7FFF
0x0002 = #0x8000
R0 = R0 + 0x0000      ; 1 clk
R0 = R0 + #0xFFFF     ; 2 clk
```