ARCompact™ ISA for the ARC™ 600

Programmer's Reference

5052-005

ARCompact™ ISA for the ARC™ 600 Programmer's Reference

ARC™ International

European Headquarters	North American Headquarters
ARC House	2025 Gateway Place, Suite 140
The Waterfront	San Jose, CA 95110 USA
Elstree Road	Tel. 408.437.3400
Elstree, Herts WD6 3BS UK	Fax 408.437.3401
Tel. +44 (0) 20.8236.2800	
Fax +44 (0) 20.8236.2801	

www.arc.com

Confidential and proprietary information

Notice

This document contains confidential and proprietary information of ARC International and is protected by copyright, trade secret, and other state, federal, and international laws. Its receipt or possession does not convey any rights to reproduce, disclose its contents, or manufacture, use, or sell anything it may describe. Reproduction, disclosure, or use without specific written authorization of ARC International is strictly forbidden.

The product described in this manual is licensed, not sold, and may be used only in accordance with the terms of an end-user license agreement (EULA) applicable to it. Use without an EULA, in violation of the EULA, or without paying the license fee is unlawful.

Every effort is made to make this manual as accurate as possible. However, ARC International shall have no liability or responsibility to any person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this manual, including but not limited to any interruption of service, loss of business or anticipated profits, and all direct, indirect, and consequential damages resulting from the use of this manual. The entire ARC International warranty and liability in respect of use of the product are set out in the EULA.

ARC International reserves the right to change the specifications and characteristics of the product described in this manual, from time to time, without notice to users. For current information on changes to the product, users should read the readme file and/or release notes that are contained in the distribution media. Use of the product is subject to the warranty provisions contained in the EULA.

Trademark acknowledgments—ARC, the ARC logo, ARCangel, ARCform, ARChitect, ARCompact, ARCtangent, BlueForm, CASSEIA, High C/C++, High C++, iCon186, MetaDeveloper, Precise Solution, Precise/BlazeNet, Precise/EDS, Precise/MFS, Precise/MQX, Precise/MQX Test Suites, Precise/MQXsim, Precise/RTCS, Precise/RTCSsim, SeeCode, TotalCore, Turbo186, Turbo86, V8 µ-RISC, V8 microRISC, and VAutomation are trademarks of ARC International. High C and MetaWare are registered under ARC International. All other trademarks are the property of their respective owners.

5052-005 September-2003

Contents

List of Figures

List of Tables

Chapter 1 — Introduction

This document is aimed at programmers of the ARCompact™ ISA for theARC™ 600 family of processors.

Information on additional extensions or customizations that have been implemented in the target ARC 600 system are contained in other manuals.

The ARCompact ISA is designed to reduce code size and maximize the opcode space available to extension instructions.

In the ARCompact ISA, compact 16-bit encodings of frequently used statically occurring 32-bit instructions are defined. These can be freely intermixed with the 32-bit instructions.

Key Features

Data Paths

- 32-Bit Data Bus
- 32-Bit Load/Store Address Bus
- 16/32-Bit Instruction Bus
- 32-Bit Instruction Address Bus

Registers

- 32 General Purpose Core Registers
- Loop Count Register
- Auxiliary Register Set

Load/Store Unit

- Delayed Load mechanism with Register Scoreboard
- Buffered Store

- Address Register Write-Back
- Pre and Post Address Register Write-Back
- Stack Pointer Support
- Scaled Data Size Addressing Mode
- PC-relative addressing

Program Flow

- 5 Stage Pipeline
- Single Cycle Instructions
- Conditional ALU Instructions
- Single Cycle Immediate Data
- Jumps and Branches with Single Instruction Delay Slot
- Combined compare-and-branch instructions
- Delay Slot Execution Modes
- Zero Overhead Loops

Interrupts and Exceptions

- Levels of Exception
- Non-Maskable Exceptions
- Maskable External Interrupts in basecase ARC 600

Extensions

- 111 Extension Dual Operand Instruction Codes
- 64 User Extension Dual Operand Instruction Codes
- 28 Extension Core Registers
- 32-Bit addressable Auxiliary Register Set
- 16 Extension Condition Codes
- Build Configuration Registers

System Customizations

- Host Interface
- Separate Memory Controller
- Separate Load/Store Unit
- Separate Interrupt Unit

Host Interface Debug Features

- Start, stop and single step the ARC 600 processor via special registers
- Check and change the values in the register set and ARC 600 memory
- Communicate via the semaphore register and shared memory
- Perform code profiling by reading the status register
- Breakpoint Instruction

Power Management

- Sleep Mode
- Clock Gating Option

Architectural Overview

The ARC 600 processor is a RISC architecture, containing a four stage pipeline. The ARCompact instruction set is orthogonal, allowing ALU and load/store operations on any of the general purpose core registers.

The architecture is extandable in the instruction set and registers. These extensions will be touched upon in this document but covered fully in other documents.

This document describes the a minimal configuration of an ARC 600 processor.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Programmer’s Model

The programmer’s model is common to all implementations of ARC 600 processor and allows upward compatibility of code.

Logically, the ARC 600 processor is based around a 3 (or 4)-port core register file with many of the instructions having two source operands and 1 destination register. Other registers are contained in the auxiliary register set and are accessed with the LOAD-REGISTER/STORE-REGISTER commands or other special commands.

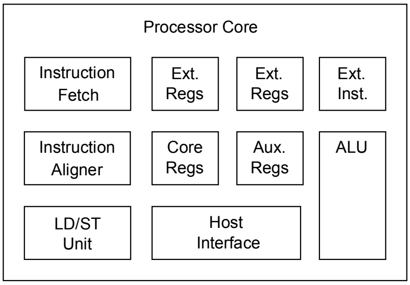


Figure 1.1 Block diagram of the ARC 600 architecture

Core register set

The general purpose registers (r0-r28)can be used for any purpose by the programmer. Some of these core registers have recommended functionality like stack pointers. The remaining core registers have special purposes like link registers and loop counters.

Auxiliary register set

The auxiliary register set contains special status and control registers. Auxiliary registers occupy a special address space that is accessed using special load and store instructions, or other special commands.

16-bit and 32-bit Instructions

Short immediate values are implied by the various instruction formats. 32-bit long immediate data (limm) is indicated by using r62 as a source register.

Register r63 (PCL) is a read-only value of the 32-bit PC (32-bit aligned) for use as a source operand in all instructions allowing PC-relative addressing.

There are compact 16-bit encodings of frequent statically occurring 32-bit instructions. Compressed 16-bit instructions typically use:

- Frequently used instructions only
- Reduced register range (most frequent 8 registers: r0-r3, r12-r15)
- Implied registers (BLINK, SP, GP, FP, PC)
- Typically only 1 or 2 operand registers specified (destination and source register are the same)
- Reduced immediate data size
- Reduced branch range
- No branch delay slot execution modes
- No conditional execution
- No flag setting option (only a few instructions will set flags e.g. BTST_S, CMP_S and TST_S)

The Host

The ARC 600 processor was developed with an integrated host interface to support communications with a host system. The ARC 600 processor can be started, stopped and communicated by the host system using special registers. Further information is contained in later sections of this manual.

Most of the techniques outlined here will be handled by the software debugging system, and the programmer, in general, need not be concerned with these specific details.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Extensions

The ARC 600 processor is designed to be extendable according to the requirements of the system in which it is used. These extensions include more core and auxiliary registers, new instructions, and additional condition code tests. This section is intended to inform the programmer of the ARC 600 processor where these extensions occur and how they affect the programmer's view of the ARC 600 processor.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Extension core registers

The core register set has a total of 64 different addressable registers. The first 29 are general purpose basecase registers, the next 3 are the link registers and the last 4 are the loop count register, long immediate data indicator, read-only program counter and a reserved register. This leaves registers 32 to 59 for extension purposes. The extension registers are referred to as r32, r33,..etc....,r59. The core register map is shown in Figure 3.1.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Extension auxiliary registers

The auxiliary registers are accessed with 32-bit addresses and are long word data size only. Extensions to the auxiliary register set can be anywhere in this address space except those positions defined as basecase for auxiliary registers. They are referred to using the load from auxiliary register (LR) and store to auxiliary register (SR) instructions or special extension instructions. The reserved auxiliary register addresses are shown in Figure 3.2 Auxiliary Register Map.

NOTE Note: If an auxiliary register position that does not exist is read, then the ID register value is returned.

The auxiliary register address region 0x60 up to 0x80, is reserved for the Build Configuration Registers (BCR) that can be used by embedded software or host debug software to detect the configuration of the ARC 600 hardware. The Build Configuration Registers contain the version of each ARC 600 extension, as well as configuration information that is build specific.

NOTE The Build Configuration Registers are fully described in the associated documentation.

The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Extension instructions

Instruction are encoded within an instruction word using a 5 bit binary field. The first 8 encodings define 32-bit instructions, the remaining 24 encodings define 16-bit instructions. Two extension encodings are reserved in the 32-bit instruction set and another encodings in the 16-bit instruction set. User extension instructions are provided by one encoding in the 32-bit instruction set and two encodings in the 16-bit instruction set. Each encoding can contain dual operand instructions (**a ← b op c**), single operand instructions (**a ← op b**) and zero operand instructions (**op c**).

Extension instructions can be used in the same way as the normal ALU instructions, except an external ALU is used to obtain the result for write-back to the core register set.

Extension condition codes

The condition code test on an instruction is encoded using a 5 bit binary field. This gives 32 different possible conditions that can be tested. The first 16 codes (0x00-0x0F) are those condition codes defined in the basecase version of ARC 600 processor which use only the internal condition flags from the status register (Z, N, C, V), see Table 6.2 Condition codes.

The remaining 16 condition codes (10-1F) are available for extensions and are used to:

- provide additional tests on the internal condition flags or
- test extension status flags from external sources or
- test a combination external and internal flags

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

System Customization

As well as the extensions mentioned in the previous section, ARC 600 processor can be additionally customized to match memory, cache, and interrupt requirements.

Load store unit

The load store unit contains the register scoreboard for marking which registers are waiting to be written from the result of delayed loads. The size of the scoreboard is changed according to the number of delayed loads that the memory controller can accommodate at any given time. The load store unit can additionally be modified to provide result write-back and register scoreboard for multi-cycle extension instructions.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Interrupt Unit

The interrupt unit contains the exception and interrupt vector positions, the logic to tell the ARC 600 processor which of the 3 levels of interrupt has occurred, and the arbitration between the interrupts and exceptions. The interrupt unit can be modified to alter the priority of interrupts, the vector positions and the number of interrupts.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Debugging Features

It is possible for the ARC 600 processor to be controlled from a host processor using special debugging features. The host is able to:

- start and stop the ARC 600 processor via the status and debug register
- single step the ARC 600 processor via the debug register
- check and change the values in the register set and ARC 600 memory
- communicate with the ARC 600 processor via the semaphore register and shared memory

- perform code profiling by reading the status register
- enable software breakpoints by using BRK instruction

With these abilities it is possible for the host to provide software breakpoints, single stepping and program tracing of the ARC 600 processor.

It is possible for the ARC 600 processor to halt itself with the FLAG instruction. The self halt bit (SH) in the debug register is set if the ARC 600 processor halts itself.

Power Management

The ARC 600 processor has special power management features. The SLEEP instruction halts the ARC 600 processor and halts the pipeline until an interrupt or a restart occurs. Sleep mode stalls the core pipeline and disables any on-chip RAM.

Optional clock gating is provided which will switch off all non-essential clocks when the ARC 600 processor is halted or the ARC 600 processor is in sleep mode. This means the internal ARC 600 control unit is not active and major blocks are disabled. The host interface, interrupt unit and memory interfaces are always left enabled to allow host accesses and "wake" feature. The following diagram shows a summary of the clock gating and sleep circuitry.

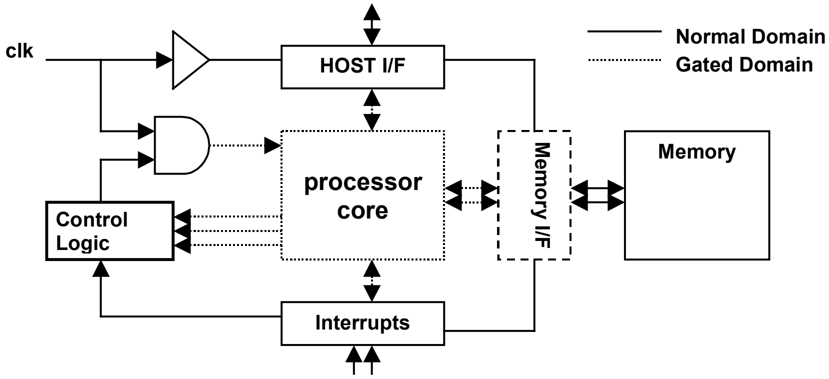


Figure 1.2 Power Management Block Diagram

Chapter 2 — Data Organization and Addressing

This chapter describes the data organization and addressing used with the ARC 600 processor.

Address Space

The ARC 600 processor has independent Instruction and Data paths that may be configured in either a von Neumann or Harvard configuration.

- The 32-bit Program Counter supports a 4GB address space for code.
- Data transfer instructions support 32-bit addressing for load/store data operations, providing a 4GB data space.
- An Auxiliary address space provides an additional 4G long word locations for memory and/or peripherals.

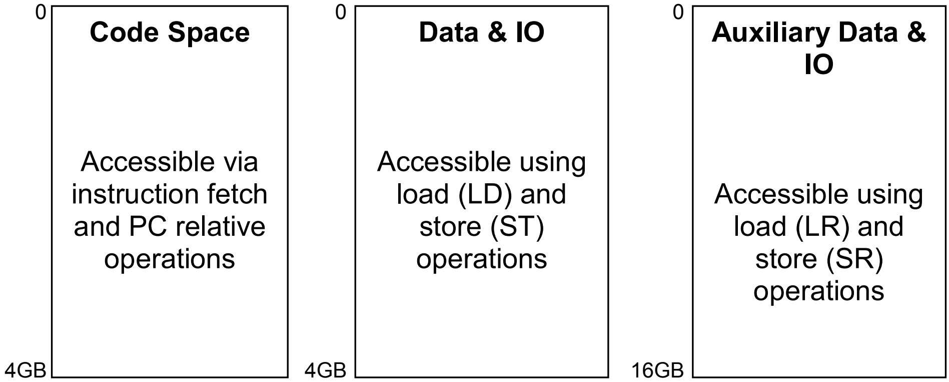

0
Code Space
Accessible via instruction fetch and PC relative operations
4GB
0
Data & IO
Accessible using load (LD) and store (ST) operations
4GB
0
Auxiliary Data & IO
Accessible using load (LR) and store (SR) operations
16GB

Figure 2.1 Address Space Model

Data Formats

The basecase ARC 600 processor is, by default, a little-endian architecture. The ARC 600 processor can operate on data of various sizes. The memory operations (load and store) can have data of 32 bit (long word), 16 bit (word) or 8 bit(byte) wide. Byte operations use the low order 8 bits and may extend the sign of the byte across the rest of the long word depending on the load/store instruction. The same applies to the word operations with the word occupying the low order 16 bits. Data memory is accessed using byte addresses, which means long word or word accesses can be supplied with non-aligned addresses. The following data alignments are supported:

- long words on long word boundaries
- words on word boundaries
- bytes on byte boundaries

There is no "unaligned access exception" available in the ARC 600 processor.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

32-bit data

All load/store, arithmetic and logical operations support 32-bit data.

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">Byte 3</td><td colspan="8">Byte 2</td><td colspan="8">Byte 1</td><td colspan="8">Byte 0</td></tr>
</table>

Figure 2.2 Register Containing 32-bit Data

<table>
<tr><td>Address</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>(</td></tr>
<tr><td>N</td><td colspan="8">Byte 0</td></tr>
<tr><td>N+1</td><td colspan="8">Byte 1</td></tr>
<tr><td>N+2</td><td colspan="8">Byte 2</td></tr>
<tr><td>N+3</td><td colspan="8">Byte 3</td></tr>
</table>

Figure 2.3 32-bit Register Data in Byte-Wide Memory

16-bit data

Load/store and some multiplication instructions support 16-bit data. 16-bit data can be converted to 32-bit data by using unsigned extend (EXTW) or signed extend (SEXW) instructions.

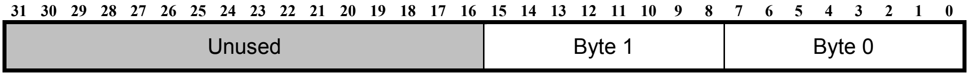


Figure 2.4 Register containing 16-bit data

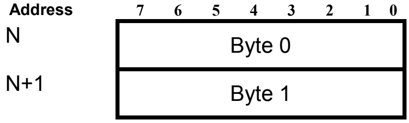


Figure 2.5 16-bit Register Data in Byte-Wide Memory

8-bit data

Load/store operations support 8-bit data. 8-bit data can be converted to 32-bit data by using unsigned extend (EXTB) or signed extend (SEXB) instructions.

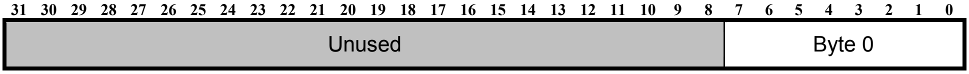


Figure 2.6 Register containing 8-bit data

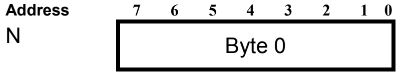


Figure 2.7 8-bit Register Data in Byte-Wide Memory

1-bit data

The ARC 600 processor supports single bit operations on data stored in the core registers. A bit manipulation instruction includes an immediate value specifying the bit to operate on. Bit manipulation instructions can operate on 8, 16 or 32 bit data located within core registers, as each bit is individually addressable.

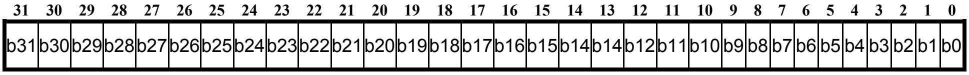

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 b16 b15 b14 b14 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0

Figure 2.8 Register containing 1-bit data

Instruction Formats

Both 32-bit and 16-bit instructions are available in the ARCompact ISA. Since a 32 bit instruction word is always fetched from the memory controller, certain alignment operations occur on the 32-bit memory word before instruction word is in the correct format for interpretation by the ARC 600 processor.

16-bit and 32-bit instructions can be mixed in memory, which can lead to 32-bit instructions and 32-bit immediate data becoming misaligned (i.e. the 32-bit value is on a 16-bit address boundary).

A special packed instruction format is used to enable the correct operation of the instruction fetch interface of the ARC 600 processor.

The following instruction information can be contained in the 32-bit memory value:

- 32-bit instruction word
- Two 16-bit instruction words
- One 16-bit instruction word and the first part of a 32-bit instruction word
- The second part of a 32-bit instruction word and one 16-bit instruction word
- The second part of a 32-bit instruction word and the first part of the following 32-bit instruction word.
- 32-bit long immediate data in the same position as a 32-bit instruction word

Packed middle-endian instruction format

The basecase ARC 600 processor is, by default, a little-endian architecture. However, the packed instruction format allows the instruction fetch mechanism to determine the address of the next PC when a 32-bit memory word contains a 16-bit instruction. Part of this mechanism is to ensure that any misaligned 32-bit instruction provides the opcode field in the first 16-bits that are retrieved from memory. For the ARC 600 this means that the upper 16-bits of the 32-bit instruction must be provided first, even in a little-endian memory system, hence the term middle-endian. Once an instruction is unpacked into its full 32-bit instruction word the fields are interpreted as documented in the following chapters.

32-bit Instruction or 32-bit immediate data

Assuming a little-endian memory representation, a packed 32-bit instruction, or 32-bit immediate data will be stored in memory as illustrated below.

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
Byte 3	Byte 2	Byte 1	Byte 0

Figure 2.9 32-bit Instruction byte representation

Address	7 6 5 4 3 2 1 0
N	Byte 2
N+1	Byte 3
N+2	Byte 0
N+3	Byte 1

Figure 2.10 32-bit instruction in Byte-Wide memory

Two 16-bit Instructions

Assuming a little-endian memory representation, two packed 16-bit instruction will be stored in memory as illustrated below.

Instruction 1

15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
Ins 1 Byte 1	Ins1 Byte 0

Instruction 2

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

Figure 2.11 16-bit Instruction byte representation

Address	7 6 5 4 3 2 1 0
N	Ins 1 Byte 0
N+1	Ins 1 Byte 1
N+2	Ins 2 Byte 0
N+3	Ins 2 Byte 1

Figure 2.12 Two 16-bit instructions in Byte-Wide memory

16-bit instruction followed by 32-bit instruction

Assuming a little-endian memory representation, a 16-bit instruction followed by a 32-bit instruction will be stored in memory as illustrated below.

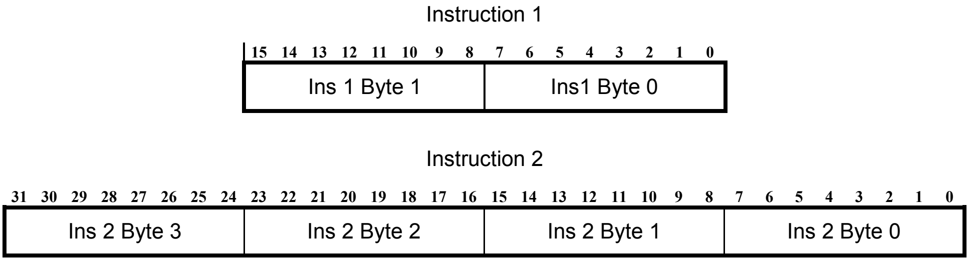


Figure 2.13 16-bit and 32-bit Instruction byte representation

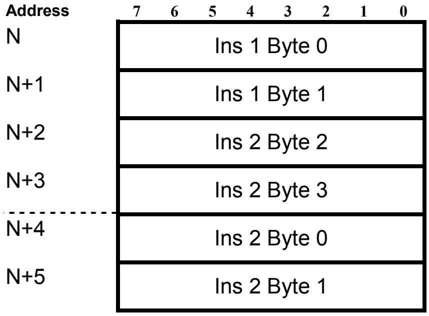
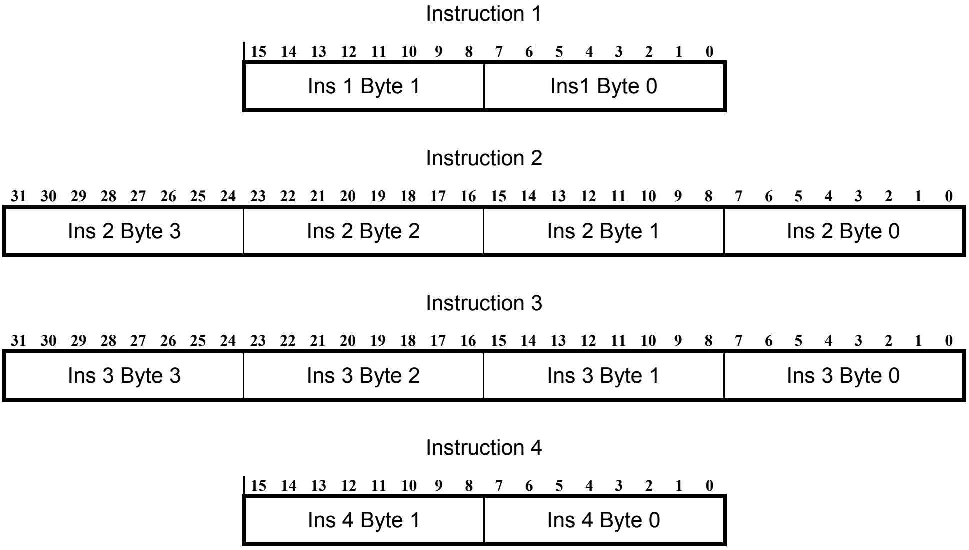


Figure 2.14 16-bit and 32-bit instructions in Byte-Wide Memory

Series of 16-bit and 32-bit instructions

Assuming a little-endian memory representation, a 16-bit and 32-bit instruction sequence will be stored in memory as illustrated below.



Instruction 1

15–8	7–0
Ins 1 Byte 1	Ins1 Byte 0

Instruction 2

31–24	23–16	15–8	7–0
Ins 2 Byte 3	Ins 2 Byte 2	Ins 2 Byte 1	Ins 2 Byte 0

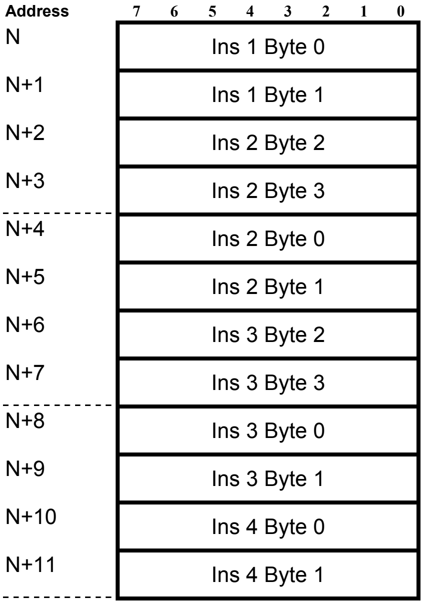
Instruction 3

31–24	23–16	15–8	7–0
Ins 3 Byte 3	Ins 3 Byte 2	Ins 3 Byte 1	Ins 3 Byte 0

Instruction 4

15–8	7–0
Ins 4 Byte 1	Ins 4 Byte 0

Figure 2.15 16-bit and 32-bit instruction sequence, byte representation



Address	7 6 5 4 3 2 1 0
N	Ins 1 Byte 0
N+1	Ins 1 Byte 1
N+2	Ins 2 Byte 2
N+3	Ins 2 Byte 3
N+4	Ins 2 Byte 0
N+5	Ins 2 Byte 1
N+6	Ins 3 Byte 2
N+7	Ins 3 Byte 3
N+8	Ins 3 Byte 0
N+9	Ins 3 Byte 1
N+10	Ins 4 Byte 0
N+11	Ins 4 Byte 1

Figure 2.16 16-bit and 32-bit instruction sequence, in Byte-Wide memory

Addressing Modes

There are six basic addressing modes supported by the architecture:

Register Direct	operations are performed on values stored in registers
Register Indirect	operations are performed on locations specified by the contents of registers
Register Indirect with offset	operations are performed on locations specified by the contents of a register plus an offset value (in another register, or as immediate data)
Immediate	operations are performed using constant data stored within the opcode
PC relative	operations are performed relative to the current value of the Program Counter (usually branch or PC relative loads)
Absolute	operations are performed on data at a location in memory specified by a constant value in the opcode.

The instruction formats for each addressing mode are specified in the following sections. The descriptions use a format defined below. An instruction is described by the operation (op), including optional flags, then the operand list.

Operation

<.f>	writeback to status register flags
<.cc>	condition code field (e.g. conditional branch)
<.d>	delay slot follows instruction (used for branch & jump)
<.zz>	size definition (Byte, Word. Long)
<.x>	perform sign extension
<.di>	data cache bypass (load and store operations)
<.aa>	address writeback

Operand	
a, b & c	General Purpose registers (note reduced range for 16-bit instructions)
h	General Purpose register, full range for 16-bit instructions.
u<X>	unsigned immediate values of size <X>-bits
s<X>	signed immediate values of size <X> bits
limm	long immediate value of size 32-bits (stored as a second opcode)

Null instruction format

The ARCompact ISA supports a special type of instruction format, where the destination of the operation is defined as null (0). When this instruction format is used the result of the operation is discarded, but the condition codes may be set – this allows any instruction to act in a manner similar to compare.

For example:

```
ADD.F  r1, r2, r3    ;the result of r2+r3
                     ;is written to r1 and
                     ;the flags are updated

ADD.F  0,r2,r3       ;the result of r2+r3 is
                     ;used to update the
                     ;flags, but is not saved.

MOV    0,0           ;recommended NOP equivalent
```

As all 32-bit instruction formats support this mode, it is not explicitly defined in the addressing modes for the processor. The 16-bit instruction set provides a no-operation instruction, NOP_S.

Conditional execution

A number of the 32-bit instructions in the ARCompact ISA support conditional execution. A 5-bit condition code field allows up to 32 independent conditions to be tested for before execution of the instruction. Sixteen conditions are defined by default, with the remainder available for customer definition, as required.

Conditional branch instruction

Both the 32-bit and 16-bit instructions support conditional branch (Bcc) operations. The 32-bit instructions also include conditional jump and jump and link (Jcc and JLcc respectively), whereas the 16-bit instruction set provides unconditional jumps only.

Compare and branch instruction

The ARCompact ISA includes two forms of instruction, which integrate compare/test and branch.

The compare and branch conditionally (BRcc) command is the juxtaposition of compare (CMP) and conditional branch (Bcc) instructions. These instructions are available in both 32-bit and limited 16-bit versions.

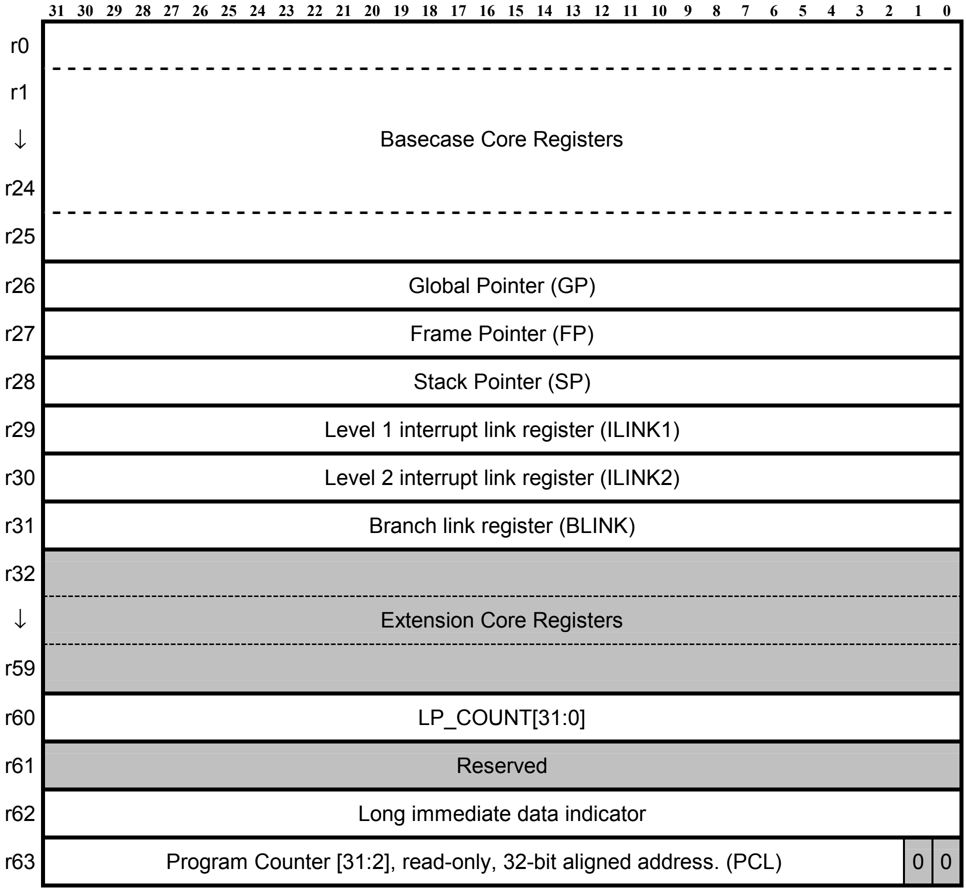
The Branch if bit set/clear (BBIT0, BBIT1) instructions provide the operation of the bit test (BTST) and branch if equal/not equal (BEQ/BNE) instructions. These instructions are only available as 32-bit instructions.

For performance reasons it is preferable to use a negative displacement with a frequently taken BRcc, BBIT0 or BBIT1 instruction, and a positive displacement with one that is rarely taken.

Chapter 3 — Register Set Details

Core Register Set

The following figure shows a summary of the core register set.



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

r0	Basecase Core Registers		
r1			
↓			
r24			
r25			
r26	Global Pointer (GP)		
r27	Frame Pointer (FP)		
r28	Stack Pointer (SP)		
r29	Level 1 interrupt link register (ILINK1)		
r30	Level 2 interrupt link register (ILINK2)		
r31	Branch link register (BLINK)		
r32	Extension Core Registers		
↓			
r59			
r60	LP_COUNT[31:0]		
r61	Reserved		
r62	Long immediate data indicator		
r63	Program Counter [31:2], read-only, 32-bit aligned address. (PCL)	0	0

Figure 3.1 Core Register Map Summary

The default implementation of the core provides 32 general purpose 32-bit core registers, users can increase the amount of available registers up to 60 if required. When executing 32-bit instructions, the full range of core registers is available. 16-bit instructions have a limited access to core registers, as shown in Table 3.1.

Register	32-bit Instruction Function and Default Usage	16-bit Instruction Access to Register
r0	General Purpose	Default Access
r1	General Purpose	Default Access
r2	General Purpose	Default Access
r3	General Purpose	Default Access
r4 - r11	General Purpose	*MOV_S, CMP_S & ADD_S*
r12	General Purpose	Default Access
r13	General Purpose	Default Access
r14	General Purpose	Default Access
r15	General Purpose	Default Access
r16 - r25	General Purpose	*MOV_S, CMP_S & ADD_S*
r26 (GP)	Global Pointer	*LD_S, MOV_S, CMP_S & ADD_S*
r27 (FP)	Frame Pointer (default)	*MOV_S, CMP_S & ADD_S*
r28 (SP)	Stack Pointer	*PUSH_S, POP_S, SUB_S, LD_S, ST_S, MOV_S, CMP_S & ADD_S*
r29 (ILINK1)	Level 1 Interrupt Link	*MOV_S, CMP_S & ADD_S*
r30 (ILINK2)	Level 2 Interrupt Link	*MOV_S, CMP_S & ADD_S*
r31 (BLINK)	Branch Link Register	*JL_S, BL_S, J_S, PUSH_S, POP_S, MOV_S, CMP_S &*

Register	32-bit Instruction Function and Default Usage	16-bit Instruction Access to Register
		ADD_S
r32 - r59	Extension Core Registers	*MOV_S, CMP_S & ADD_S*
r60 (LP_COUNT)	Loop Counter	*MOV_S, CMP_S & ADD_S*
r61	*Reserved*	*Reserved*
r62	Long Immediate	*MOV_S, CMP_S & ADD_S*
r63 (PCL)	Program Counter (32-bit aligned)	*MOV_S, CMP_S & ADD_S, LD_S*

Table 3.1 Core Register Set

Core register mapping used in 16-bit Instructions

The 16-bit instructions use only 3 bits for register encoding. However, the 16-bit move (MOV_S), the 16-bit compare (CMP_S) and the 16-bit add (ADD_S) instructions are capable of accessing the full set of core registers, this facilitates copy and manipulation of data stored in registers not accessible to other 16-bit instructions.

The most frequently used registers according to the ARCompact application binary interface (ABI) are r0-r3 (ABI call argument registers), r12 (temporary register) and r13-r15 (ABI call saved registers).

16-bit instruction register encoding	32-bit instruction register
0	r0
1	r1
2	r2
3	r3
4	r12
5	r13
6	r14
7	r15

Table 3.2 16-bit instruction register encoding

Register	Use
r0-r7	argument regs
r8-r12	temp regs
r13-r25	saved regs
r26	GP (global pointer.)
r27	FP (frame pointer)
r28	SP (stack pointer)
r29	ILINK1
r30	ILINK2
r31	BLINK

Table 3.3 Current ABI register usage

Pointer registers

The ARCompact application binary interface (ABI) defines 3 pointer registers: Global Pointer (GP), Frame Pointer (FP) and Stack Pointer (SP) which use registers r26, r27 and r28 respectively. The global pointer (GP) is used to point to small sets of shared data throughout execution of a program. The stack pointer (SP) register points to the lowest used address of the stack. The frame pointer

(FP) register points to a back-trace data structure that can be used to back-trace through function calls.

Link registers

The link registers (ILINK1, ILINK2, BLINK) are used to provide links back to the position where an interrupt or branch occurred. They can also be used as general purpose registers, but if interrupts or branch-and-link or jump-and-link are used, then these are reserved for that purpose.

Loop count register

The loop count register (LP_COUNT) is used for zero delay loops. Because LP_COUNT is decremented if the program counter equals the loop end address and also LP_COUNT does not have next cycle bypass like the other core registers, it is not recommended that LP_COUNT be used as a general purpose register, also LD instructions that use LP_COUNT as a destination is not permitted. See section Zero Overhead Loop for further details.

Immediate data indicator

Register position 62 is reserved for encoding long (32-bit) immediate data addressing modes onto instruction words. It is reserved for that purpose and are not available to the programmer as a general purpose register.

Extension core registers

The register set is extandable in register positions 32-59 (r32-r59). Results of accessing the extension register region are undefined in the basecase version of the ARC 600 processor. If a core register is read that is not implemented, then an unknown value is returned. No exception is generated. Writes to non-implemented core registers are ignored.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Multiply result registers

Table 3.4 shows the defined extension core registers for the optional multiply.

Register	Name	Use
r57	MLO	Multiply low 32 bits, read only
r58	MMID	Multiply middle 32 bits, read only
r59	MHI	Multiply high 32 bits, read only

Table 3.4 Multiply Result Registers

Auxiliary Register Set

The following figure shows a summary of the auxiliary register set.

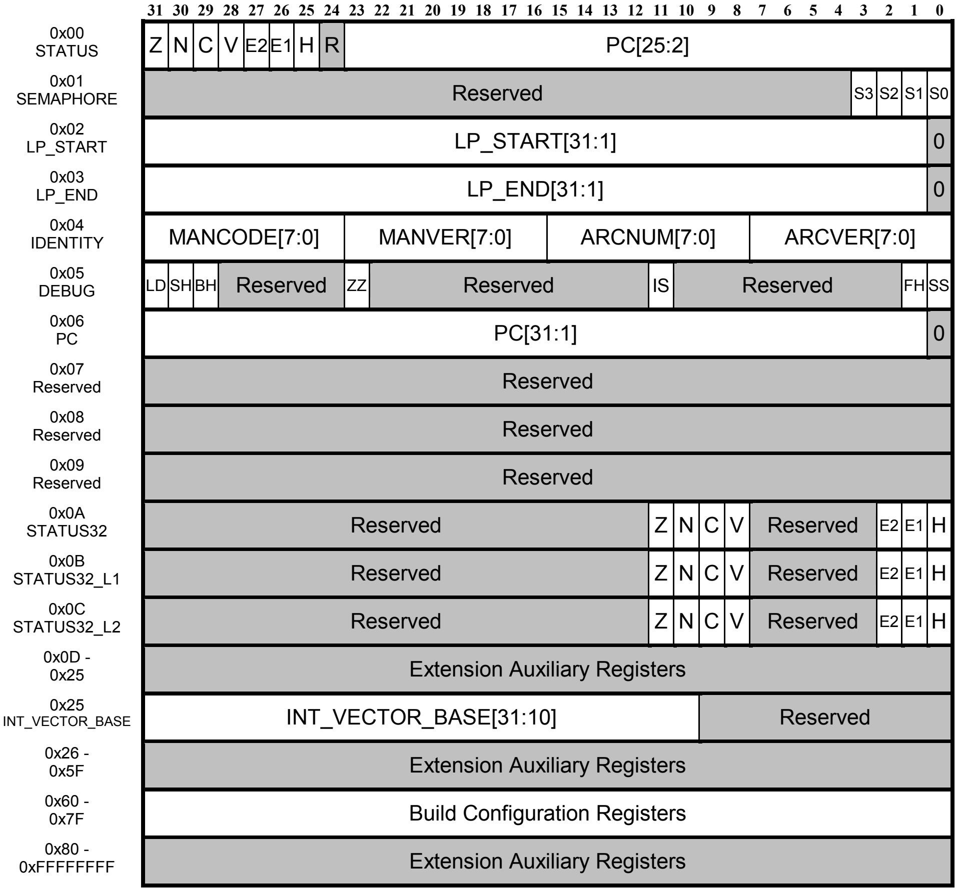


Figure 3.2 Auxiliary Register Map

The basecase ARC 600 processor uses a small set of of status and control registers and reserves registers 0x60 to 0x7F, leaving the remaining 2^{32} registers for extension purposes.

Number	Auxiliary register name	LR/SR r/w	Description
0x0	STATUS	r	Status register (Original ARCtangent-A4 processor format)
0x1	SEMAPHORE	r/w	Inter-process/Host semaphore register
0x2	LP_START	r/w	Loop start address (32-bit)
0x3	LP_END	r/w	Loop end address (32-bit)
0x4	IDENTITY	r	Processor Identification register
0x5	DEBUG	r	Debug register
0x6	PC	r	PC register (32-bit)
0xA	STATUS32	r	Status register (32-bit)
0xB	STATUS32_L1	r/w	Status register save for level 1 exceptions
0xC	STATUS32_L2	r/w	Status register save for level 2 exceptions
0x25	INT_VECTOR_BASE	r/w	Interrupt Vector Base address
0x60 - 0x7F	*RESERVED*		*Build Configuration Registers*

Table 3.5 Auxiliary Register Set

Status register (obsolete)

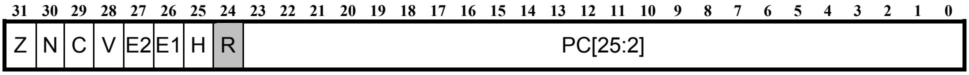


Figure 3.3 STATUS Register (Obsolete)

The status register (STATUS) is used for legacy code that may be recompiled to use the ARCompact ISA. Full status and program counter information is provided in the PC register (PC) and 32-bit status register (STATUS_32)

Semaphore register

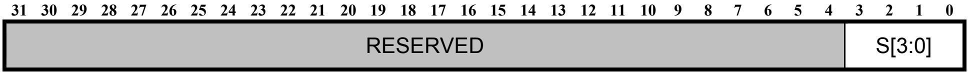


Figure 3.4 Semaphore Register

The SEMAPHORE register, Figure 3.4, is used to control inter-process or ARC 600 processor to host communication. The basecase ARC 600 processor has at least 4 semaphore bits (S[3:0]). The remaining bits of the semaphore register are reserved for future versions of ARC 600 processors.

Each semaphore bit is independent of the others and is claimed using a *set-and-test* protocol. The semaphore register can be read at any time by the host or ARC 600 processor to see which semaphores it currently owns.

To claim a semaphore bit

Write ‘1’ to the semaphore bit.

Read back the semaphore bit. Then:

If returned value is ‘1’ then semaphore has been obtained.

If returned value is ‘0’ then the host has the bit.

To release a semaphore bit.

Write a ‘0’ to the semaphore bit.

Mutual exclusion is provided between the ARC 600 processor and the host. In other words, if the host claims a particular semaphore bit, the ARC 600 processor will not be able to claim that same semaphore bit until the host has released it. Conversely, if the ARC 600 processor claims a particular semaphore bit, the host will not be able to claim that same semaphore bit until the ARC 600 processor has released it.

The semaphore bits are cleared to 0 after a reset, which is the state where neither the ARC 600 processor nor the host have claimed any semaphore bits. When claiming a semaphore bit (i.e. setting the semaphore bit to a ‘1’), care should be taken not to clear the remaining semaphore bits. Keeping a local copy, or reading the semaphore register, and OR-ing that value with the bit to be claimed before writing back to the semaphore register could accomplish this.

Example:

```
 .equ  SEMBIT0,1  ; constant to indicate semaphore bit 0
 .equ  SEMBIT1,2  ; constant to indicate semaphore bit 1
 .equ  SEMBIT2,4  ; constant to indicate semaphore bit 2
 .equ  SEMBIT3,8  ; constant to indicate semaphore bit 3

LR r2,[SEMAPHORE]   ; r2 <= semaphore pattern already attained
OR r2,r2,SEMBIT1    ; r2 <= semaphore pattern attained and wanted
SR r2,[SEMAPHORE]   ; attempt to get the semaphore bit
LR r2,[SEMAPHORE]   ; read back semaphore register
AND.F 0,r2,SEMBIT1  ; test for the semaphore bit being set
                    ; EQ means semaphore not attained
                    ; NE means semaphore attained
```

NOTE Replacing the statement OR r2,r2,SEMBIT1 with BIC r2,r2,SEMBIT1 will release the semaphore, leaving any previously attained semaphores in their attained state.

Loop control registers

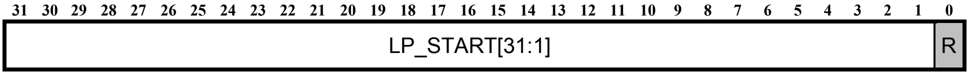


Figure 3.5 LP_START Register

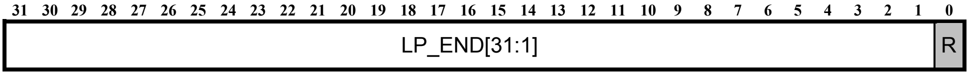


Figure 3.6 LP_END Register

The loop start (LP_START) and loop end (LP_END) registers contain the addresses for the zero delay loop mechanism. Figure 3.5 and Figure 3.6 show the format of these registers. The loop start and loop end registers can be set up with the special loop instruction (LP) or can be manipulated with the auxiliary register access instructions (LR and SR).

LP_START and LP_END registers follow the auxiliary PC register (PC) format - Bit 0 is reserved. The effect of setting Bit 0 is implementation dependant.

In the ARCompact v1 implementation Bit 0 is ignored and should always be set to zero. Programming cautions exist when using the loop control registers, see section: Zero Overhead Loop.

Identity register

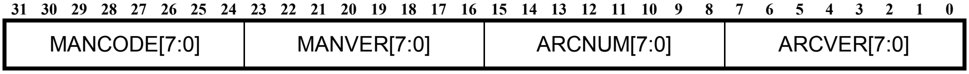


Figure 3.7 Identity Register

Figure 3.7 shows the identity register (IDENTITY). It contains the manufacturer code (MANCODE[7:0]), manufacturer version number (MANVER[7:0]), the additional identity number (ARCNUM[7:0]) and the ARC 600 basecase version number (ARCVER[7:0]).

The format for ARCVER[7:0] is

- 0x00 to 0x0F = ARCtangent-A4 processor family (Original 32-Bit only processor cores)
- 0x10 to 0x1F = ARCtangent-A5 processor family (ARCompact 16/32 Bit processor cores).
- 0x20 to 0x2F = ARC 600 processor family (ARCompact 16/32 Bit processor cores).
- 0x30 to 0xFF = Reserved

Debug register

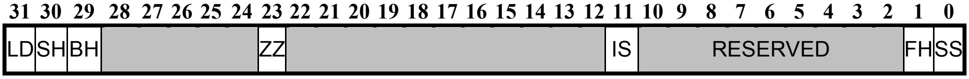


Figure 3.8 Debug Register

The debug register (DEBUG) contains seven bits: load pending bit (LD); self halt (SH); breakpoint halt (BH); sleep mode (ZZ): single instruction step (IS); force halt (FH) and single step (SS).

LD can be read at any time by either the host or the ARC 600 processor and indicates that there is a delayed load waiting to complete. The host should wait for this bit to clear before changing the state of the ARC 600 processor.

SH indicates that the ARC 600 processor has halted itself with the FLAG instruction, this bit is cleared whenever the H bit in the STATUS register is cleared (i.e. The ARC 600 processor is running or a single step has been executed).

Breakpoint Instruction Halt (BH) bit is set when a breakpoint instruction has been detected in the instruction stream at stage one of the pipeline. A breakpoint halt is set when BH is ‘1’. This bit is cleared when the H bit in the status register is cleared, e.g. single stepping or restarting the ARC 600 processor.

ZZ bit indicates that the ARC 600 processor is in "sleep" mode. The ARC 600 processor enters sleep mode following a SLEEP instruction. ZZ is cleared whenever the ARC 600 processor "wakes" from sleep mode.

The force halt bit (FH) is a foolproof method of stopping the ARC 600 processor externally by the host. The host setting this bit does not have any side effects when the ARC 600 processor is halted already. FH is not a mirror of the STATUS register H bit:- clearing FH will *not* start the ARC 600. FH always returns 0 when it is read.

Single stepping is provided through the use of IS and SS. Single instruction step (IS) is used in combination with SS. When IS and SS are both set by the host the ARC 600 processor will execute one full instruction.

SS is a write only bit that when set by the host will cause the ARC 600 processor to single cycle step. The single cycle step function enables the processor for one cycle. It should be noted that this does not necessarily correspond to one instruction being executed, since stall conditions may be present. In order to execute a single instruction, the remote system must repeatedly single-step the ARC 600 processor until the values change in either the program counter or loop count register, or use SS in combination with IS.

Program counter

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PC[31:1]																															R

Figure 3.9 PC Register

The PC register contains the 32-Bit program counter. Bit 0 is reserved (ignored) however it should be set to zero.

Status Register 32-bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED																				Z	N	C	V	R	R	R	R	R	E2	E1	H

Figure 3.10 STATUS32 Register

The status register contains the status flags. The status register (STATUS32), shown in Figure 3.10, contains the following status flags: zero (Z), negative (N), carry (C) and overflow (V); the interrupt mask bits (E[2:1]); and the halt bit (H).

CAUTION	There must be at least one instruction between a FLAG instruction and a "J.F<.D> [ILINK1]" or "J.F<.D> [ILINK2]" instruction.

```
FLAG    0x100
NOP
J.F     [ilink1]
```

Status Link Registers

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="20">RESERVED</td><td>Z</td><td>N</td><td>C</td><td>V</td><td>R</td><td>R</td><td>R</td><td>R</td><td>R</td><td>E2</td><td>E1</td><td>R</td></tr>
</table>

Figure 3.11 STATUS32_L1, STATUS32_L2 Registers

A level 1 or level 2 interrupt/exception will save the current status register STATUS32 in auxiliary register STATUS32_L1 or STATUS32_L2.

If J.F<.D> [ILINK1] or J.F<.D> [ILINK2] instructions are executed to return from level 1 or level 2 interrupts/exceptions then the current status register STATUS32 will be restored from auxiliary register STATUS32_L1 or STATUS32_L2 accordingly.

CAUTION	There must be at least one instruction between writing to STATUS32_L1 or STATUS32_L2 using an SR instruction and a "J.F<.D> [ILINK1]" or "J.F<.D> [ILINK2]" instruction.

```
SR      r0,[STATUS32_L1]
NOP
J.F     [ilink1]
```

Interrupt Vector Base Register

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="22">INT_VECTOR_BASE[31:10]</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Figure 3.12 INT_VECTOR_BASE Registers

The Interrupt Vector Base register (INT_VECTOR_BASE) contains the base address of the interrupt vectors. On reset the interrupt vector base address is loaded with a value from the interrupt system. This value can be read from INT_VECTOR_BASE at any time. During program execution the interrupt vector base can be changed by writing to INT_VECTOR_BASE. The interrupt vector base address can be set to any 1Kbyte-aligned address. The bottom 10 bits are ignored for writes and will return 0 on reads.

Extension auxiliary registers

The auxiliary register set is extandable up to the full 2^{32} register space. Results of accessing the extension auxiliary register region are undefined in the basecase version of the ARC 600 processor. If an auxiliary register is read that is not implemented, then an unknown value is returned. No exception is generated. Writes to non-implemented auxiliary registers are ignored.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Optional extensions auxiliary registers

The following table summarizes the auxiliary registers that are used by the optional extensions:

Number	Name	r/w	Description
0x12	MULHI	w	High part of multiply to restore multiply state
0x7B	MULTIPLY_BUILD	r	Build configuration for: multiply
0x7C	SWAP_BUILD	r	Build configuration for: swap
0x7D	NORM_BUILD	r	Build configuration for: normalize
0x7F	BARREL_BUILD	r	Build configuration for: barrel shift

Chapter 4 — Interrupts

Introduction

The ARC 600 interrupt mechanism is such that 3 levels of interrupts are provided.

- Exceptions like Reset, Memory Error and Invalid Instruction (high priority)
- level 2 (mid priority) interrupts which are maskable.
- level 1 (low priority) interrupts which are maskable

The exception set has the highest priority, level 2 set has middle priority and level 1 the lowest priority.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

ILINK and Status Save Registers

When an interrupt occurs, the appropriate link register (ILINK1 or ILINK2) is loaded with the value of next PC, the associated status save register (STATUS32_L1 or STATUS32_L2) is also updated with the status register (STATUS32); the PC is then loaded with the relevant address for servicing the interrupt.

Link register ILINK2 and status save register STATUS32_L2 are associated with the level 2 set of interrupts and the two exceptions: memory error and instruction error. ILINK1 and status save register STATUS32_L1 are associated with the level 1 set of interrupts.

Interrupt Vectors

In the basecase ARC 600 processor, there are three exceptions and each exception has it's own vector position, an alternate interrupt unit may be implemented, see section Alternate Interrupt Unit.

The ARC 600 processor does not implement interrupt vectors as such, but rather a table of jumps. When an interrupt occurs the ARC 600 processor jumps to fixed addresses in memory, which contain a jump instruction to the interrupt handling code. The start of these interrupt vectors is dependent on the particular ARC 600 system and is often a set of contiguous jump vectors. The INT_VECTOR_BASE register can be read at any time to determine the start of the interrupt vectors, and can be used to change the base of the interrupt vectors during program execution, see section Interrupt Vector Base Register.

Example vector offsets are shown in the following table. Two long-words are reserved for each interrupt line to allow room for a jump instruction with a long immediate address.

Vector	Name	Link register	Byte Offset	Priority
0	reset	-	0x00	1 Highest
1	memory exception	ILINK2	0x08	2
2	instruction error	ILINK2	0x10	3
3 – n	*irq3-irqn*	*ILINK1 or ILINK2*	0x18 – 0x08*n	*System dependant*

Table 4.1 Interrupt Summary

It is possible to execute the code for servicing the last interrupt in the interrupt vector table without using the jump mechanism. An example set of vectors showing the last interrupt vector is shown in the following code.

```
             ;Start of exception vectors
reset: JAL   res_service   ;vector 0
mem_ex:      JAL    mem_service  ;vector 1, ilink2
ins_err:     JAL    instr_service       ;vector 2, ilink2
ivect3:      JAL    iservice3           ;vector 3,   ilink1
ivect4:      ;vector 4, interrupt,      ilink1
             ;start of interrupt service code for
             ;ivect4
```

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Interrupt Enables

The level 1 set and level 2 set of interrupts are maskable. The interrupt enable bits E2 and E1 in the status register (STATUS32) are used to enable level 2 set and level 1 set of interrupts respectively. Interrupts are enabled or disabled with the flag instruction.

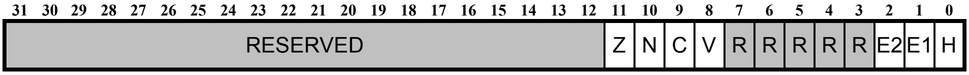


Figure 4.1 STATUS32 Register

Example:

```
.equu  EI,6     ; constant to enable both interrupts
```

```
.equ   EI1,2    ; constant to enable level 1 interrupt only
.equ   EI2,4    ; constant to enable level 2 interrupt only
.equ   DI,0     ; constant to disable both interrupts

FLAG   EI       ; enable interrupts and clear other flags

FLAG   DI       ; disable interrupts and clear other flags
```

Returning from Interrupts

When the interrupt routine is entered, the interrupt enable flags are cleared for the current level and any lower priority level interrupts. Hence, when a level 2 interrupt occurs, both the interrupt enable bits in the status register are cleared at the same time as the PC is loaded with the address of the appropriate interrupt routine.

Returning from an interrupt is accomplished by jumping to the contents of the appropriate link register, using the JAL [ILINK*n*] instruction. With the flag bit enabled on the jump instruction, the status register is also loaded from the associate STATUS32_L*n* register, thus returning the flags to their state at point of interrupt, including of course the interrupt enable bits E1 and E2, one or both of which will have been cleared on entry to the interrupt routine.

There are 2 link registers ILINK1 (r29) and ILINK2 (r30) for use with the maskable interrupts, memory exception and instruction error. These link registers correspond to levels 1 and 2 and the interrupt enable bits E1 and E2 for the maskable interrupts.

For example, if there was no interrupt service routine for interrupt number 5, the arrangement of the vector table would be:

```
ivect4:      JAL    iservice4  ;vector 4
ivect5:      JAL.F  [ILINK1]        ;vector 5 (jump to ilink1)
             NOP              ;instruction padding
ivect6:      JAL    iservice6  ;vector 6
```

Reset

A reset is an external reset signal that causes the ARC 600 processor to perform a “hard” reset. Upon reset, various internal states of the ARC 600 processor are pre-set to their initial values: the pipeline is flushed; interrupts are disabled; status register flags are cleared; the semaphore register is cleared; loop count, loop start and loop end registers are cleared; the scoreboard unit is cleared; pending load flag is cleared; and program execution resumes at the interrupt vector base address (offset 0x00) which is the basecase ARC 600 reset vector position. The core registers are not initialized except loop count (which is cleared). A jump to the reset vector, a “soft” reset, will *not* pre-set any of the internal states of the ARC 600 processor.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Memory Error

A memory error can be caused by an instruction fetch from, a load from or a store to an invalid part of memory. In the basecase ARC 600 processor, this exception is non-recoverable in that the instruction that caused the error cannot be returned to.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Instruction Error

If an invalid instruction is fetched that the ARC 600 processor cannot execute, then an instruction error is caused. In the basecase ARC 600 processor, this exception is non-recoverable in that the instruction that caused the error cannot be returned to. The mechanism checks all major opcodes and sub-opcodes to determine whether the instruction is valid.

The software interrupt instruction (SWI) will also generate an instruction error exception when executed.

Interrupt Timing

Interrupts are held off when a compound instruction has a dependency on the following instruction or is waiting for immediate data from memory. This occurs during a branch, jump or simply when an instruction uses long immediate data. Interrupts are also held off when a predicted branch is in the pipeline, or when a flag instruction is being processed. The time taken to service an interrupt is basically a jump to the appropriate vector and then a jump to the routine pointed to by that vector. The timings of interrupts according to the type of instruction in the pipeline is given later in this documentation.

The time it takes to service an interrupt will also depend on the following:

- Whether a jump instruction is contained in the interrupt vector table
- Allowing stage 1 to stage 2 dependencies to complete
- Whether a predicted branch is being processed
- Returning loads using write-back stage
- An I- Cache miss causing the I-Cache to reload in order to service the interrupt
- The number of register push items onto a software stack at the start of the interrupt service routine
- Whether an interrupt of the same or higher level is already being serviced
- An interruption by higher level interrupt

Alternate Interrupt Unit

It should be assumed that ARC 600 adheres to the interrupt mechanism according to this chapter. It is possible, however, that an alternate interrupt unit may be provided on a particular system. The interrupt unit contains the exception and interrupt vector positions, the logic to tell the ARC 600 which of the 3 levels of interrupt has occurred, and the arbitration between the interrupts and exceptions.

The interrupt unit can be modified to alter the priority of interrupts, the vector positions and the number of interrupts. The 3 levels of interrupt which are set with the status register and the return mechanism through link registers can not be altered. Further masking bits and extra link registers can be provided by the use of extensions in the auxiliary and core register set. How this would be done is entirely system dependent.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Interrupt Flow

The following diagram illustrates the process involved when and interrupt or exception occurs during program execution. The priority for each level of interrupt is shown, but the interrupt priority within each level set is system dependent.

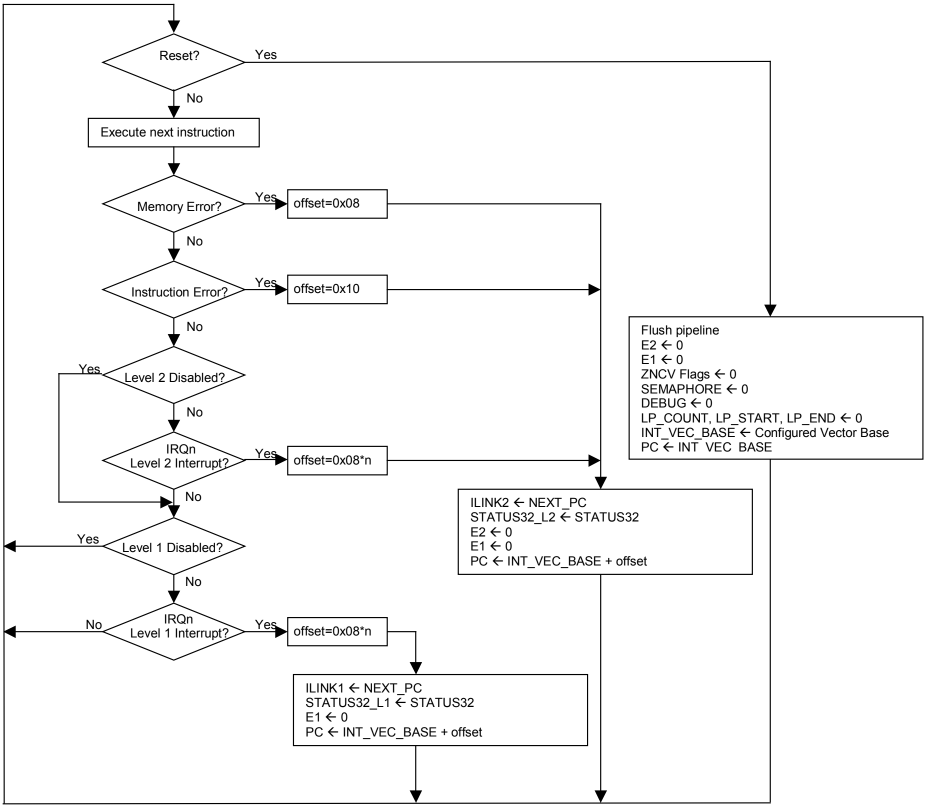


Figure 4.2 Interrupt Execution

Chapter 5 — Instruction Set Summary

This chapter contains an overview of the types of instructions in the ARCompact ISA.

Both 32-bit and 16-bit instructions are available in the ARCompact ISA and are indicated using particular suffixes on the instruction as illustrated by the following syntax:

OP	implies 32-bit instruction
OP_L	indicates 32-bit instruction.
OP_S	indicates 16-bit instruction

If no suffix is used on the instruction then the implied instruction is 32-bit format. Note that 16-bit instructions have a reduced range of source and target core registers unless indicated otherwise.

The following notation is used for the syntax of operations.

a	destination register (reduced range for 16-bit instruction.)
b	source operand 1 (reduced range for 16-bit instruction.)
c	source operand 2 (reduced range for 16-bit instruction.)
h	full register range for 16-bit instructions
cc	condition code
<.cc>	optional condition code
Z	Zero flag
N	Negative flag
C	Carry flag
V	Overflow flag
<.f>	optional set flags
<.aa>	optional address writeback
<.d>	optional delay slot mode
<.di>	optional direct data cache bypass
<.x>	optional sign extend
<zz>	optional data size
u	unsigned immediate, number indicates field size
s	signed immediate, number indicates field size
limm	long immediate

Table 5.1 Instruction Syntax Convention

List of Instructions

The ARCompact ISA has 32 base instruction opcodes with additional variations (including NOP) that provide a set of 86 arithmetic and logical instructions, load/store, and branch/jump instructions. 51 instructions are 32-bit and the remaining 35 instructions are 16-bit. The extension options provide an additional 4 instructions of 32-bit formats and 1 instruction in 16-bit format, giving a total of 91 instructions.

The following table summarizes the 32-bit alongside the 16-bit instructions supported by the ARCompact ISA.

32-Bit Instructions		16-Bit Instructions	
Instruction	**Operation**	**Instruction**	**Operation**
ABS	Absolute value	ABS_S	Absolute value
ADC	Add with carry		
ADD	Add	ADD_S	Add
ADD1	Add with left shift by 1 bit	ADD1_S	Add with left shift by 1 bits
ADD2	Add with left shift by 2 bits	ADD2_S	Add with left shift by 2 bits
ADD3	Add with left shift by 3 bits	ADD3_S	Add with left shift by 3 bits
AND	Logical AND	AND_S	Logical AND
ASL	Arithmetic Shift Left	ASL_S	Arithmetic Shift Left
ASR	Arithmetic Shift Right	ASR_S	Arithmetic Shift Right
BBIT0	Branch if bit cleared to 0		
BBIT1	Branch if bit set to 1		
Bcc	Branch if condition true	Bcc_S	Branch if condition true
BCLR	Clear specified bit (to 0)	BCLR_S	Clear specified bit (to 0)
BIC	Bit-wise inverted AND	BIC_S	Bit-wise inverted AND
BLcc	Branch and Link	BL_S	Branch and Link
BMSK	Bit Mask	BMSK_S	Bit Mask
BRcc	Branch on compare	BRcc_S	Branch on compare
BSET	Set specified bit (to 1)	BSET_S	Set specified bit (to 1)
BTST	Test value of specified bit	BTST_S	Test value of specified bit
BXOR	Bit XOR		
CMP	Compare	CMP_S	Compare
EXT	Unsigned extend	EXT_S	Unsigned extend
FLAG	Write to Status Register		
Jcc	Jump	J_S	Jump
JLcc	Jump and Link	JL_S	Jump and Link
LD	Load from memory	LD_S	Load from memory
LP	Loop (zero-overhead loops)		

32-Bit Instructions		16-Bit Instructions	
Instruction	**Operation**	**Instruction**	**Operation**
LR	Load from Auxiliary memory		
LSR	Logical Shift Left	LSR_S	Logical Shift Right
MAX	Return Maximum		
MIN	Return Minimum		
MOV	Move (copy) to register	MOV_S	Move (copy) to register
MUL64	32 x 32 Multiply	MUL64_S	32 x 32 Multiply
MULU64	32 x 32 Unsigned Multiply		
NORM	Normalize to 32 bits		
NORMW	Normalize to 16 bits		
NOT	Logical bit inversion	NOT_S	Logical bit inversion
OR	Logical OR	OR_S	Logical OR
RCMP	Reverse Compare		
RLC	Rotate Left through Carry		
ROR	Rotate Right		
RRC	Rotate Right through Carry		
RSUB	Reverse Subtraction		
SBC	Subtract with carry	NEG_S	Negate
SEX	Signed extend	SEX_S	Signed extend
SLEEP	Put processor in sleep mode		
SR	Store to Auxiliary memory		
ST	Store to memory	ST_S	Store to memory
SUB	Subtract	SUB_S	Subtract
SUB1	Subtract with left shift by 1 bit		
SUB2	Subtract with left shift by 2 bits		
SUB3	Subtract with left shift by 3 bits		
SWAP	Swap 16 x 16		
SWI	Software Interrupt		
TST	Test	TST_S	Test
XOR	Logical Exclusive-OR	XOR_S	Logical Exclusive-OR
		BRK_S	Break (halt) processor
		NOP_S	No Operation
		POP_S	Restore register value from stack
		PUSH_S	Store register value on stack

Table 5.2 List of Instructions

Arithmetic and Logical Operations

These operations are of the form **a ←b op c** where the destination (a) is replaced by the result of the operation (op) on the operand sources (b and c). The ordering of the operands is important for some non-commutative operations (for example: SUB, SBC, BIC, ADD1/2/3, SUB1/2/3) All arithmetic and logical instructions can be conditional and/or set the flags.

If the destination register is set to an absolute value of "0" then the result is discarded and the operation acts like a NOP instruction. A long immediate (limm) value can be used for either source operand 1 or source operand 2.

Summary of basecase ALU instructions

Instruction	Operation	Description
ADD	a ← b + c	add
ADC	a ← b + c + C	add with carry
SUB	a ← b – c	subtract
SBC	a ← (b – c) - C	subtract with carry
AND	a ← b and c	logical bitwise AND
OR	a ← b or c	logical bitwise OR
BIC	a ← b and not c	logical bitwise AND with invert
XOR	a ← b exclusive-or c	logical bitwise exclusive-OR
MAX	a ← b max c	larger of 2 signed integers
MIN	a ← b min c	smaller of 2 signed integers
MOV	b ← c	move
TST	b and c	test
CMP	b - c	compare
RCMP	c - b	reverse compare
RSUB	a ← c - b	reverse subtract
BSET	a ← b or (1<<c)	bit set

Instruction	Operation	Description
BCLR	a ← b and not (1<<c)	bit clear
BTST	b and (1<<c)	bit test
BXOR	a ← b xor (1<<c)	bit xor
BMSK	a ← b and ((1<<(c+1))-1)	bit mask
ADD1	a ← b + (c << 1)	add with left shift by 1
ADD2	a ← b + (c << 2)	add with left shift by 2
ADD3	a ← b + (c << 3)	add with left shift by 3
SUB1	a ← b - (c << 1)	subtract with left shift by 1
SUB2	a ← b - (c << 2)	subtract with left shift by 2
SUB3	a ← b - (c << 3)	subtract with left shift by 3
ASL	a ← b asl c	arithmetic shift left
ASR	a ← b asr c	arithmetic shift right
LSR	a ← b lsr c	logical shift right
ROR	a ← b ror c	rotate right

Table 5.3 Basecase ALU Instructions

Syntax for arithmetic and logical operations

Including "0" as destination value and a limm as either source operand 1 or source operand 2 expands the generic syntax for standard arithmetic and logical instructions. The generic instruction syntax is used for the following arithmetic and logic operations:

> SUB; AND; OR; BIC; XOR; ADD1; ADD2; ADD3; ASL; ASR and LSL

The following instructions have the same generic instruction format, but do not have a 16 bit instruction (op_S b,b,c) equivalent.

> ADC; SBC; RSUB; SUB1; SUB2; SUB3; ROR; MIN and MAX.

The full generic instruction syntax is:

op<.f>	a,b,c	
op<.f>	a,b,u6	
op<.f>	b,b,s12	
op<.cc><.f>	b,b,c	
op<.cc><.f>	b,b,u6	
op<.f>	a,limm,c	*(if b=limm)*
op<.f>	a,b,limm	*(if c=limm)*
op<.cc><.f>	b,b,limm	
op<.f>	0,b,c	*;if a=0*
op<.f>	0,b,u6	
op<.f>	0,b,limm	*(if a=0, c=limm)*
op<.cc><.f>	0,limm,c	*(if a=0, b=limm)*
op_S	b,b,c	*(reduced register range)*

For example, the syntax for AND is:

AND<.f>	a,b,c	*(a = b and c)*
AND<.f>	a,b,u6	*(a = b and u6)*
AND<.f>	b,b,s12	*(b = b and s12)*
AND<.cc><.f>	b,b,c	*(b = b and c)*
AND<.cc><.f>	b,b,u6	*(b = b and u6)*
AND<.f>	a,limm,c	*(a = limm and c)*
AND<.f>	a,b,limm	*(a = b and limm)*
AND<.cc><.f>	b,b,limm	*(b = b and limm)*
AND<.f>	0,b,c	*(b and c)*
AND<.f>	0,b,u6	*(b and u6)*
AND<.cc><.f>	0,b,limm	*(b and limm)*
AND<.cc><.f>	0,limm,c	*(limm and c)*
AND_S	b,b,c	*(b = b and c)*

Add instruction

The ADD instruction extends the generic instruction syntax for 16-bit instruction formats to allow access to stack pointer (SP) and global pointer (GP), along with further immediate modes. The syntax for ADD is:

ADD<.f>	a,b,c	*(a = b+c)*
ADD<.f>	a,b,u6	*(a = b+u6)*
ADD<.f>	b,b,s12	*(b = b+s12)*
ADD<.cc><.f>	b,b,c	*(b = b+c)*
ADD<.cc><.f>	b,b,u6	*(b = b+u6)*
ADD<.f>	a,limm,c	*(a = limm+c)*
ADD<.f>	a,b,limm	*(a = b+limm)*
ADD<.cc><.f>	b,b,limm	*(b = b+limm)*
ADD<.f>	0,b,c	*(b+c)*
ADD<.f>	0,b,u6	*(b+u6)*
ADD<.cc><.f>	0,b,limm	*(b+limm)*
ADD<.cc><.f>	0,limm,c	*(limm+c)*
ADD_S	a, b, c	*(a = b + c, reduced set of regs)*
ADD_S	c, b, u3	*(c = b + u3, reduced set of regs)*
ADD_S	b, b, u7	*(b = b + u7, reduced set of regs)*
ADD_S	b, b, h	*(b = b + h, full set of regs for h)*
ADD_S	b, b, limm	*(b = b + limm)*
ADD_S	r0, GP, s11	*(32-bit aligned offset)*
ADD_S	b, SP, u7	*(u7 offset is 32-bit aligned)*
ADD_S	SP, SP, u7	*(u7 offset is 32-bit aligned)*

Subtract instruction

The subtract instruction extends the generic instruction syntax for 16-bit instruction formats to allow access to stack pointer (SP) and further immediate modes. The syntax variants for SUB are:

SUB<.f>	a,b,c	*(a = b-c)*
SUB<.f>	a,b,u6	*(a = b-u6)*
SUB<.f>	b,b,s12	*(b = b-s12)*
SUB<.cc><.f>	b,b,c	*(b = b-c)*
SUB<.cc><.f>	b,b,u6	*(b = b-u6)*
SUB<.f>	a,limm,c	*(a = limm-c)*
SUB<.f>	a,b,limm	*(a = b-limm)*
SUB<.cc><.f>	b,b,limm	*(b = b-limm)*
SUB<.f>	0,b,c	*(b-c)*
SUB<.f>	0,b,u6	*(b-u6)*
SUB<.cc><.f>	0,b,limm	*(b-limm)*
SUB<.cc><.f>	0,limm,c	*(limm-c)*
SUB_S	b,b,c	*(b = b-c, reduced set of regs)*
SUB_S.NE	b,b,b	*(If Z=0 Clear b, reduced set of regs)*
SUB_S	b, b, u5	*(b = b-u5, reduced set of regs)*
SUB_S	c, b, u3	*(c = b-u3, reduced set of regs)*
SUB_S	SP, SP, u7	*(u7 offset is 32-bit aligned)*

Reverse subtract instruction

The Reverse Subtract instruction (RSUB) is special in that the source1 and source2 operands are swapped over by the ARC 600 processor ALU before the subtract operation.

The syntax of RSUB, however, stays the same as that for the generic ALU operation:

RSUB<.f>	a,b,c	*(a = c-b)*
RSUB<.f>	a,b,u6	*(a = u6-b)*
RSUB<.f>	b,b,s12	*(b = s12-b)*

RSUB<.cc><.f>	b,b,c	*(b = c-b)*
RSUB<.cc><.f>	b,b,u6	*(b = u6-b)*
RSUB<.f>	a,limm,c	*(a = c-limm)*
RSUB<.f>	a,b,limm	*(a = limm-b)*
RSUB<.cc><.f>	b,b,limm	*(b = limm-b)*
RSUB<.f>	0,b,c	*(c-b)*
RSUB<.f>	0,b,u6	*(u6-b)*
RSUB<.cc><.f>	0,b,limm	*(limm-b)*
RSUB<.cc><.f>	0,limm,c	*(c-limm)*

Test and compare instructions

TST, CMP and RCMP have special instruction encoding in that the destination is always ignored and the instruction result is always discarded. The flags are always set according to the instruction result (implicit ".f", and encoded with F=1). RCMP is special in that the source1 and source2 operands are swapped over by the ARC 600 processor ALU before the subtract operation.

Register-register

The *General operations register-register* format is implemented and provides the following redundant formats for TST, CMP and RCMP:

op	b,c	*(b=source 1, c=source 2.Redundant format see Conditional Register format)*
op	b,limm	*(b=source 1, c=limm=source 2. Redundant format see Conditional Register format)*
op	limm,c	*(limm=source 1, c=source 2. Redundant format see Conditional Register format)*
op	limm,limm	*(limm=source 1, limm=source 2.. Redundant format see Conditional Register format)*

Register with unsigned 6-bit immediate

The *General operations register with unsigned 6-bit immediate* format is implemented and provides the following redundant formats for TST, CMP and RCMP:

op	b,u6	*(b=source 1, u6=source 2. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*
op	limm,u6	*(limm=source 1, u6=source 2. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*

Register with signed 12-bit immediate

The *General operations register with signed 12-bit immediate* format provides the following syntax for TST, CMP and RCMP:

op	b,s12	*(b=source 1, s12=source 2)*
op	limm,s12	*(limm=source 1, s12=source 2. Not useful format)*

Conditional Register

The *General operations conditional register* format provides the following syntax for TST, CMP and RCMP:

op<.cc>	b,c	*(b=source 1, c=source 2)*
op<.cc>	b,limm	*(b=source 1, c=limm=source 2)*
op<.cc>	limm,c	*(limm=source 1, c=source 2)*
op<.cc>	limm,limm	*(limm=source 1, limm=source 2. Not useful format)*

Conditional register with unsigned 6-bit immediate

The *General operations conditional register with unsigned 6-bit immediate* format provides the following syntax for TST, CMP and RCMP:

op<.cc>	b,u6	*(b=source 1, u6=source 2)*
op<.cc>	limm,u6	*(limm=source 1, u6=source 2. Not useful format)*

The syntax for test and compare instructions is therefore:

TST	b,s12	*(b & s12)*
TST<.cc>	b,c	*(b & c)*
TST<.cc>	b,u6	*(b & u6)*
TST<.cc>	b,limm	*(b & limm)*

TST<.cc>	limm,c	*(limm & c)*
TST_S	b,c	*(b&c, reduced set of regs)*

CMP	b,s12	*(b-s12)*
CMP<.cc>	b,c	*(b-c)*
CMP<.cc>	b,u6	*(b-u6)*
CMP<.cc>	b,limm	*(b-limm)*
CMP<.cc>	limm,c	*(limm-c)*
CMP_S	b, h	*(b-h, full set of regs for h)*
CMP_S	b, limm	*(b-limm, full set of regs for h)*
CMP_S	b, u7	*(b-u7, reduced set of regs)*

RCMP	b,s12	*(s12-b)*
RCMP<.cc>	b,c	*(c-b)*
RCMP<.cc>	b,u6	*(u6-b)*
RCMP<.cc>	b,limm	*(limm-b)*
RCMP<.cc>	limm,c	*(c-limm)*

Bit Test Instruction

The BTST instruction only requires two source operands. BTST has a special instruction encoding in that the destination is always ignored and the instruction result is always discarded. The second source operand selects the bit position to test (0 to 31), which can be covered by a u6 immediate number. The status flags are always set according to the instruction result (implicit ".f", and encoded with F=1).

Register-register

The *General operations register-register* format is implemented and provides the following redundant formats for BTST:

BTST	b,c	*(b=source 1, c=source 2.Redundant format see Conditional Register format)*

BTST	b,limm	*(b=source 1, c=limm=source 2. Redundant format see Conditional register with unsigned 6-bit immediate format)*
BTST	limm,c	*(limm=source 1, c=source 2. Redundant format see Conditional Register format)*
BTST	limm,limm	*(limm=source 1, limm=source 2.. Redundant format see Conditional register with unsigned 6-bit immediate format)*

Register with unsigned 6-bit immediate

The *General operations register with unsigned 6-bit immediate* format is implemented and provides the following redundant formats for BTST:

BTST	b,u6	*(b=source 1, u6=source 2. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*
BTST	limm,u6	*(limm=source 1, u6=source 2. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*

Register with signed 12-bit immediate

The *General operations register with signed 12-bit immediate* format provides the following redundant syntax for BTST:

BTST	b,s12	*(b=source 1, s12=source 2. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*
BTST	limm,s12	*(limm=source 1, s12=source 2. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*

Conditional Register

The *General operations conditional register* format provides the following syntax for BTST:

BTST<.cc>	b,c	*(b=source 1, c=source 2)*
BTST<.cc>	b,limm	*(b=source 1, c=limm=source 2. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*

BTST<.cc>	limm,c	*(limm=source 1, c=source 2)*
BTST<.cc>	limm,limm	*(limm=source 1, limm=source 2. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*

Conditional register with unsigned 6-bit immediate

The *General operations conditional register with unsigned 6-bit immediate* format provides the following syntax for BTST:

BTST<.cc>	b,u6	*(b=source 1, u6=source 2)*
BTST<.cc>	limm,u6	*(limm=source 1, u6=source 2. Not useful format)*

Single bit instructions

The single bit instructions (BSET, BCLR, BXOR and BMSK) instructions require two source operands and one destination operand. The second source operand selects the bit position to test (0 to 31) which can be covered by a u6 immediate number.

BSET, BCLR, BXOR and BMASK are bit-set, bit-clear, bit-xor and bit-mask instructions, respectively.

Register-register

The *General operations register-register* format is implemented and provides the following formats for BSET, BCLR, BXOR and BMSK:

op<.f>	a,b,c	
op<.f>	a,limm,c	*(if b=limm)*
op<.f>	a,b,limm	*(if c=limm. Redundant format see Register with unsigned 6-bit immediate format)*
op<.f>	a,limm,limm	*(if b=c=limm. Redundant format see Register with unsigned 6-bit immediate format)*
op<.f>	0,b,c	*(if a=0)*
op<.f>	0,limm,c	*(Redunant format, see Conditional Register format)*
op<.f>	0,b,limm	*(if a=0, c=limm. Redundant format see Register with unsigned 6-bit immediate format)*

op<.f>	0,limm,limm	*(if a=0, b=c=limm. Redundant format see Conditional register with unsigned 6-bit immediate format)*

Register with unsigned 6-bit immediate

The *General operations register with unsigned 6-bit immediate* format is implemented and provides the following formats for BSET, BCLR, BXOR and BMSK:

op<.f>	a,b,u6	
op<.f>	a,limm,u6	*(Not useful format)*
op<.f>	0,b,u6	
op<.f>	0,limm,u6	*(Redundant format see Conditional register with unsigned 6-bit immediate format)*

Register with signed 12-bit immediate

The *General operations register with signed 12-bit immediate* format provides the following redundant syntax for BSET, BCLR, BXOR and BMSK:

op<.f>	b,b,s12	*(Redundant format see Conditional register with unsigned 6-bit immediate format)*
op<.f>	0,limm,s12	*(Redundant format see Conditional register with unsigned 6-bit immediate format)*

Conditional Register

The *General operations conditional register* format provides the following syntax for BSET, BCLR, BXOR and BMSK:

op<.cc><.f>	b,b,c	
op<.cc><.f>	0,limm,c	
op<.cc><.f>	b,b,limm	*(Redundant format see Conditional register with unsigned 6-bit immediate format)*
op<.cc><.f>	0,limm,limm	*(Redundant format see Conditional register with unsigned 6-bit immediate format)*

Conditional register with unsigned 6-bit immediate

The *General operations conditional register with unsigned 6-bit immediate* format provides the following syntax for BSET, BCLR, BXOR and BMSK:

op<.cc><.f>	b,b,u6	
op<.cc><.f>	0,limm,u6	*(Not useful format)*

The syntax for the single bit operations is therefore:

BSET<.f>	a,b,c	*(a = b | (1<<c))*
BSET<.cc><.f>	b,b,c	*(b = b | (1<<c))*
BSET<.f>	a,b,u6	*(a = b | (1<<u6))*
BSET<.cc><.f>	b,b,u6	*(b = b | (1<<u6))*
BSET_S	b, b, u5	*(uses reduced set of regs)*
BCLR<.f>	a,b,c	*(a = b & ~(1<<c))*
BCLR<.cc><.f>	b,b,c	*(b = b & ~(1<<c))*
BCLR<.f>	a,b,u6	*(a = b & ~(1<<u6))*
BCLR<.cc><.f>	b,b,u6	*(b = b & ~(1<<u6))*
BCLR_S	b, b, u5	*(uses reduced set of regs)*
BTST<.cc>	b,c	*(b & (1<<c))*
BTST<.cc>	b,u6	*(b & (1<<u6))*
BTST_S	b, u5	*(uses reduced set of regs)*
BXOR<.f>	a,b,c	*(a = b xor (1<<c))*
BXOR<.cc><.f>	b,b,c	*(b = b xor (1<<c))*
BXOR<.f>	a,b,u6	*(a = b xor (1<<u6))*
BXOR<.cc><.f>	b,b,u6	*(b = b xor (1<<u6))*
BMSK<.f>	a,b,c	*(a = b & ((1<<(c+1))-1))*
BMSK<.cc><.f>	b,b,c	*(b = b & ((1<<(c+1))-1))*
BMSK<.f>	a,b,u6	*(a= b & ((1<<(u6+1))-1))*
BMSK<.cc><.f>	b,b,u6	*(b= b & ((1<<(u6+1))-1))*

BMSK_S	b, b, u5	*(uses reduced set of regs)*

Barrel Shift/Rotate

The barrel shifter provides a number of instructions that will allow any operand to be shifted left or right by up to 32 positions in one cycle, the result being available for write-back to any core register. Single bit shift instructions are also provided as single operand instructions as shown in Table 5.6.

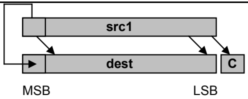
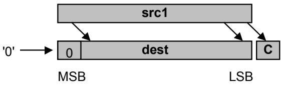
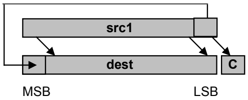
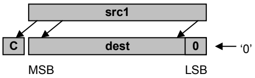
Instruction	Operation	Description
ASR		multiple arithmetic shift right, sign filled
LSR		multiple logical shift right, zero filled
ROR		multiple rotate right
ASL		multiple arithmetic shift left, zero filled

Table 5.4 Barrel Shift Operations

The ROR instruction does not have any 16 bit instruction (op_S *a*,*b*,*c*) equivalent. The ASR, LSR and ASL instructions extend the generic instruction syntax to include:

op_S	b,b,u5
op_S	b,b,c

ASR and LSR additionally provide the following syntax

op_S	c,b,u3

The syntax for the barrel shifter is:

ASL<.f>	a,b,c	*(a = b<<c)*
ASL<.f>	a,b,u6	*(a = b<<u6)*
ASL<.f>	b,b,s12	*(b = b<<s12)*

ASL<.cc><.f>	b,b,c	*(b = b<<c)*
ASL<.cc><.f>	b,b,u6	*(b = b<<u6)*
ASL<.f>	a,limm,c	*(a = limm<<c)*
ASL<.f>	a,b,limm	*(a = b<<limm)*
ASL<.cc><.f>	b,b,limm	*(b = b<<limm)*
ASL<.f>	0,b,c	*(b<<c)*
ASL<.f>	0,b,u6	*(b<<u6)*
ASL<.cc><.f>	0,limm,c	*(limm<<c)*
ASL_S	c,b,u3	*(c = b<<u3)*
ASL_S	b,b,c	*(b = b<<c)*
ASL_S	b,b,u5	*(b=b<<u5)*

ASR<.f>	a,b,c	*(a = b>>c)*
ASR<.f>	a,b,u6	*(a = b>>u6)*
ASR<.f>	b,b,s12	*(b = b>>s12)*
ASR<.cc><.f>	b,b,c	*(b = b>>c)*
ASR<.cc><.f>	b,b,u6	*(b = b>>u6)*
ASR<.f>	a,limm,c	*(a = limm>>c)*
ASR<.f>	a,b,limm	*(a = b>>limm)*
ASR<.cc><.f>	b,b,limm	*(b = b>>limm)*
ASR<.f>	0,b,c	*(b>>c)*
ASR<.f>	0,b,u6	*(b>>u6)*
ASR<.cc><.f>	0,limm,c	*(limm>>c)*
ASR_S	c,b,u3	*(c = b>>u3)*
ASR_S	b,b,c	*(b = b>>c)*
ASR_S	b,b,u5	*(b=b>>u5)*

LSR<.f>	a,b,c	*(a = b>>c)*
LSR<.f>	a,b,u6	*(a = b>>u6)*
LSR<.f>	b,b,s12	*(b = b>>s12)*
LSR<.cc><.f>	b,b,c	*(b = b>>c)*
LSR<.cc><.f>	b,b,u6	*(b = b>>u6)*
LSR<.f>	a,limm,c	*(a = limm>>c)*
LSR<.f>	a,b,limm	*(a = b>>limm)*
LSR<.cc><.f>	b,b,limm	*(b = b>>limm)*
LSR<.f>	0,b,c	*(b>>c)*
LSR<.f>	0,b,u6	*(b>>u6)*
LSR<.cc><.f>	0,limm,c	*(limm>>c)*
LSR_S	b,b,c	*(b = b>>c)*
LSR_S	b,b,u5	*(b = b>>u6)*

ROR<.f>	a,b,c	*(a = (b<<(31-c)):(b>>c))*
ROR<.f>	a,b,u6	*(a = (b<<(31-u6)):(b>>u6))*
ROR<.f>	b,b,s12	*(b = (b<<(31-s12)):(b>>s12))*
ROR<.cc><.f>	b,b,c	*(b = (b<<(31-c)):(b>>c))*
ROR<.cc><.f>	b,b,u6	*(b = (b<<(31-u6)):(b>>u6))*
ROR<.f>	a,limm,c	*(a = (limm<<(31-c)):(limm>>c))*
ROR<.f>	a,b,limm	*(a = (b<<(31-limm)):(b>>limm))*
ROR<.cc><.f>	b,b,limm	*(b = (b<<(31-limm)):(b>>limm)*
ROR<.f>	0,b,c	*((b<<(31-c)):(b>>c))*
ROR<.f>	0,b,u6	*((b<<(31-u6)):(b>>u6))*
ROR<.cc><.f>	0,limm,c	*((b<<(31-limm)):(limm>>c))*

Single Operand Instructions

Some instructions require just a single source operand. These include sign-extend and rotate instructions. These instructions are of the form **b ← op c** where the destination (b) is replaced by the operation (op) on the operand source (c). Single operand instructions can set the flags.

The following table shows the move, extend and negate operations.

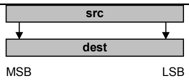
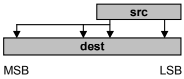
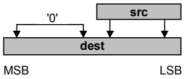
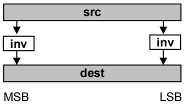
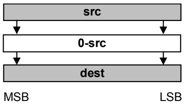
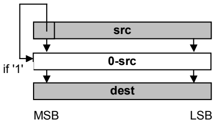
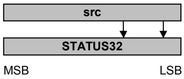
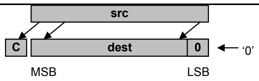
Instruction	Operation	Description
MOV		Move
SEX		Sign extend byte or word
EXT		Zero extend byte or word
NOT		Logical NOT
NEG		Negate
ABS		Absolute
FLAG		Set flags

Table 5.5 Single operand: moves and extends

The following table shows the rotate and shift operations.

Instruction	Operation	Description
ASL		Arithmetic shift left by one

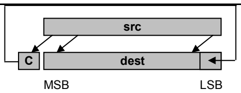
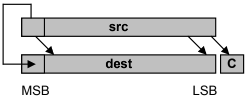
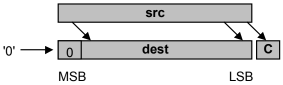
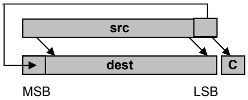
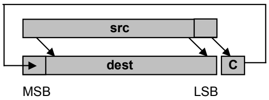
RLC		Rotate left through carry
ASR		Arithmetic shift right by one
LSR		Logical shift right by one
ROR		Rotate right
RRC		Rotate right through carry

Table 5.6 Single operand: Rotates and Shifts

The following instructions do not have a 16 bit instruction (op_S b,c) equivalent.

ROR, RRC and RLC;

Single operand instruction syntax is:

```
op<.f>      b,c
op<.f>      b,u6
op<.f>      b,limm
op<.f>      0,c
op<.f>      0,u6
op<.f>      0,limm
op_S        b,c
```

Move to register instruction

The move instruction, MOV, has a wider syntax than other single operand instructions by being encoded as a general ALU instruction. The first operand is only used as the destination register; the final operand is used as the source operand. Using the limm encoding in the first operand field is ignored in just the same way as it is if used in the destination of other instructions, causing the MOV instruction result to be discarded.

Register-register

The *General operations register-register* format is implemented, where the destination field A is ignored and the B field is used instead as the destination register. The MOV instruction provides the following redundant formats:

MOV<.f>	b,c	*(b=destination, c=source. Redundant format, see Conditional Register format.)*
MOV<.f>	b,limm	*(b=destination, c=limm=source. Redundant format, see Conditional Register format.)*
MOV<.f>	0,c	*(b=limm, c=source. Redundant format, see Conditional Register format.)*
MOV<.f>	0,limm	*(if b=limm, b= c=limm=source. Redundant format, see Conditional Register format.)*

Register with unsigned 6-bit immediate

The *General operations register with unsigned 6-bit immediate* format is implemented, where the destination field A is ignored and the B field is used instead as the destination register. The MOV instruction provides the following redundant formats:

MOV<.f>	b,u6	*(b=destination, u6=source. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*
MOV<.f>	0,u6	*(b=limm, u6=source. Redundant format, see Conditional register with unsigned 6-bit immediate format)*

Register with signed 12-bit immediate

The *General operations register with signed 12-bit immediate* format provides the following syntax for the MOV instruction:

MOV<.f>	b,s12	*(b=destination, s12=source)*
MOV<.f>	0,s12	*(b=limm, s12=source)*

Conditional Register

The *General operations conditional register* format provides the following syntax for the MOV instruction:

MOV<.cc><.f>	b,c	*(b=destination, c=source)*
MOV<.cc><.f>	b,limm	*(b=destination, c=limm=source)*
MOV<.cc><.f>	0,c	*(b=limm, c=source)*
MOV<.cc><.f>	0,limm	*(if b=limm, b= c=limm=source)*

Conditional register with unsigned 6-bit immediate

The *General operations conditional register with unsigned 6-bit immediate* format provides the following syntax for the MOV instruction:

MOV<.cc><.f>	b,u6	*(b=destination, u6=source)*
MOV<.cc><.f>	0,u6	*(b=limm, u6=source)*

16-bit instruction, move with high register

The *Mov/Cmp/Add with High Register, 0x0E, [0x00 - 0x03]* format provides the following syntax for the MOV instruction:

MOV_S	b, h	*(b = destination, h=source. Full range of regs for h)*
MOV_S	b, limm	*(b = destination, limm=source)*
MOV_S	h, b	*(h = destination, b = source. Full range of regs for h)*

16-bit instruction, move immediate

The *Move Immediate, 0x1B* format provides the following syntax for the MOV instruction

MOV_S	b, u8	*(b = destination, u8 = source. Reduced set of regs for b)*

Flag instruction

The FLAG instruction has a special syntax that ignores the destination field. The FLAG instruction always updates the status flags.

Register-register

The *General operations register-register* format is implemented, where the destination field A is ignored, the B field is ignored and the C field is used as the source register. The FLAG instruction provides the following redundant formats:

FLAG	c	*(a = ignored, b= ignored, c=source. Redundant format, see Conditional Register format.)*
FLAG	b,limm	*(a = ignored, b= ignored, c=limm=source. Redundant format, see Conditional Register format.)*

Register with unsigned 6-bit immediate

The *General operations register with unsigned 6-bit immediate* format is implemented, where the destination field A is ignored, the B field is ignored and the u6 immediate field is used as the source value. The FLAG instruction provides the following redundant formats:

FLAG	u6	*(a = ignored, b= ignored, u6=source. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*

Register with signed 12-bit immediate

The *General operations register with signed 12-bit immediate* format provides the following syntax for the FLAG instruction:

FLAG	s12	*(b = ignored, s12=source)*

Conditional Register

The *General operations conditional register* format provides the following syntax for the FLAG instruction:

FLAG<.cc>	c	*(b=ignored, c=source)*
FLAG<.cc>	limm	*(b=ignored, c=limm=source)*

Conditional register with unsigned 6-bit immediate

The *General operations conditional register with unsigned 6-bit immediate* format provides the following syntax for the FLAG instruction:

FLAG<.cc>	b,u6	*(b=ignored, u6=source)*

Negate operation

Negate is a separate instruction in 16-bit instruction format and is provided in 32-bit instruction format by using reverse subtract.

The syntax for negate operations is:

NEG_S	b,c	*(b = 0-c, reduced set of regs)*
RSUB<.f>	a,b,0	*(a = 0-b)*
RSUB<.cc><.f>	b,b,0	*(b = 0-b)*

Zero Operand Instructions

Some instructions require no source operands or destinations. The ARCompact ISA supports these instructions using the form **op c** where the operand source c supplies information for the instruction. Zero operand instructions can set the flags.

Instruction	Description
SLEEP	Sleep
SWI	Software interrupt
BRK_S	Breakpoint Instruction
NOP	Null Instruction

Table 5.7 Basecase ZOP instructions

Zero operand instruction syntax is:

SLEEP	*(encoded as SLEEP 0, i.e "sleep" with u6=0)*
SWI	*(encoded as SWI 0, i.e. "swi" with u6=0)*
NOP	*(encoded as MOV 0,0)*
NOP_S	*(16-bit instruction form)*
BRK_S	*(Breakpoint instruction)*
op<.f>	*c*
op<.f>	*u6*
op<.f>	*limm*

Breakpoint instruction

The breakpoint instruction is a single operand basecase instruction that halts the program code when it is decoded at stage one of the pipeline. This is a very basic debug instruction, which stops the ARC 600 processor from performing any instructions beyond the breakpoint. The pipeline is also flushed upon decode of this instruction.

To restart the ARC 600 processor at the correct instruction the old instruction is rewritten into main memory, immediately followed by an invalidate instruction cache line command (even if an instruction cache has not been implemented) to ensure that the correct instruction is loaded into the cache before being executed by the ARC 600 processor. The program counter must also be rewritten in order to generate a new instruction fetch, which reloads the instruction. Most of the work is performed by the debugger with regards to insertion, removal of instructions with the breakpoint instruction.

The program flow is not interrupted when employing the breakpoint instruction, and there is no need for implementing a breakpoint service routine. There is also no limit to the number of breakpoints you can insert into a piece of code.

NOTE The breakpoint instruction sets the BH bit (refer to section Chapter 1 — Programmer's Model) in the Debug register when it is decoded at stage one of the pipeline. This allows the debugger to determine what caused the ARC 600 processor to halt. The BH bit is cleared when the Halt bit in the Status register is cleared, e.g. by restarting or single–stepping the ARC 600 processor.

A breakpoint instruction may be inserted into any position.

```
MOV    r0, 0x04
ADD    r1, r0, r0
XOR.F  0, r1, 0x8
BRK_S  ;<----- break here
SUB    r2, r0, 0x3
ADD.NZ r1, r0, r0
JZ.D   [r8]
OR     r5, r4, 0x10
```

Figure 5.1 ARCompact Assembly code with BRK_S instruction

Breakpoints are primarily inserted into the code by the host so control is maintained at all times by the host. The BRK_S instruction may however be used in the same way as any other ARCompact instruction.

The breakpoint instruction can be placed anywhere in a program, except immediately following any branch or jump instruction. The breakpoint instruction is decoded at stage two of the pipeline which consequently stalls stages one and two, and allows instructions in stages three, four and five to continue, i.e. flushing the pipeline.

BRK Instruction following a Jump or Branch Instruction

Due to stage 2 to stage 1 dependencies, the breakpoint instruction behaves differently when it is placed following a Branch or Jump instruction. In these cases, the ARC 600 will stall stages one and two of the pipeline while allowing instructions in subsequent stages (three and four) to proceed to completion.

The following example shows BRK_S following a conditional jump instruction.

```
MOV    r0, 0x04
ADD    r1, r0, r0
XOR.F  0, r1, 0x8
SUB    r2, r0, 0x3
ADD.NZ r1, r0, r0
JZ.D   [r8]
BRK_S  ;<---- break inserted
       ;      into here
OR     r5, r4, 0x10
```

Figure 5.2 A Breakpoint following a jump instruction

The link register is not updated for Branch and Link, BL, (or Jump and Link, JL) instruction when the BRK_S instruction immediately follows. When the ARC 600 processor is started, the link register will update as normal.

Interrupts are treated in the same manner by the ARC 600 processor as Branch, and Jump instructions when a BRK_S instruction is detected. Therefore, an interrupt that reaches stage two of the pipeline when a BRK_S instruction is in stage one will keep it in stage two, and flush the remaining stages of the pipeline. It is also important to note that an interrupt that occurs in the same cycle as a breakpoint is held off as the breakpoint is of a higher priority. An interrupt at stage three is allowed to complete when a breakpoint instruction is in stage one.

Sleep instruction

The sleep mode is entered when the ARC 600 processor encounters the SLEEP instruction. It stays in sleep mode until an interrupt or restart occurs. Power consumption is reduced during sleep mode since the pipeline ceases to change state, and the RAMs are disabled. More power reduction is achieved when clock gating option is used, whereby all non-essential clocks are switched off.

The SLEEP instruction can be put anywhere in the code, as in the example below:

```
SUB r2, r2, 0x1
ADD r1, r1, 0x2
SLEEP
...
```

The SLEEP instruction is a single operand instruction without flags or operands. The SLEEP instruction is decoded in pipeline stage 2. If a SLEEP instruction is detected, then the sleep mode flag (ZZ) is immediately set and the pipeline stage 1 is stalled. A flushing mechanism assures that all earlier instructions are executed until the pipeline is empty. The SLEEP instruction itself leaves the pipeline during the flushing.

When in sleep mode, the sleep mode flag (ZZ) is set and the pipeline is stalled, but not halted. The host interface operates as normal allowing access to the DEBUG and the STATUS registers and it can halt the processor. The host cannot clear the sleep mode flag, but it can wake the ARC 600 processor by halting then restarting ARC 600 processor. The program counter, PC, points to the next instruction in sequence after the sleep instruction.

The ARC 600 processor will wake from sleep mode on an interrupt or when the ARC 600 processor is restarted. If an interrupt wakes up the ARC 600, the ZZ flag is cleared and the instruction in pipeline stage 1 is killed. The interrupt routine is serviced and execution resumes at the instruction in sequence after the SLEEP instruction. When the ARC 600 processor is started after having been halted, the ZZ flag is cleared.

Using SLEEP with an RTOS type application

The SLEEP and FLAG instructions can be used in RTOS type applications by ensuring that interrupts are enabled when SLEEP is executed. The reccomended code sequence is FLAG followed by SLEEP. This allows interrupts to be re-enabled at the same time as SLEEP is entered. Note that interrupts remain disabled until FLAG has completed its update of the flag registers in stage 4 of the ARC 600 pipeline. Hence, if SLEEP follows into the pipeline immediately behind FLAG, then no interrupt can be taken between the FLAG and SLEEP. However, this mechanism requires that there is no instruction-cache miss on the SLEEP instruction.

The following example shows the code sequence to ensure successful use of the SLEEP instruction for RTOS type applications.

```
.equu  EI,0x06       ; Constant to enable both interrupt levels
.align  8            ; ensure cache alignment is to 8 bytes
FLAG   EI            ; Enable interrupts
SLEEP                ; Put processor into sleep mode
```

Figure 5.3 A SLEEP following a FLAG instruction

SLEEP Instruction following a Jump or Branch Instruction

A SLEEP instruction can follow a branch or jump instruction as in the following code example:

```
BAL.D after_sleep
SLEEP
...
after_sleep:
ADD r1,r1,0x2
```

Figure 5.4 SLEEP in an enabled and executed delay slot

In this example, the ARC 600 processor goes to sleep after the branch instruction has been executed. When the ARC 600 processor is sleeping, the PC points to the “add" instruction after the label "after_sleep". When an interrupt occurs, the ARC 600 processor wakes up, executes the interrupt service routine and continues with the “add” instruction.

If the delay slot is not enabled or not executed (i.e. killed), as in the following code example, the SLEEP instruction that follows is never executed:

```
BAL.ND after_sleep
SLEEP
       ...
after_sleep:
ADD r1,r1,0x2
```

Figure 5.5 SLEEP in killed Delay Slot

The SLEEP instruction cannot immediately follow a BRcc or BBITn instruction.

SLEEP Instruction In Single Step Mode

The SLEEP instruction acts as a NOP during single step mode, because every single-step is a restart and the ARC 600 processor wakes up at the next single-step. Consequently, the SLEEP instruction behaves exactly like a NOP propagating through the pipeline.

Software interrupt instruction

The execution of an undefined extension instruction in ARC 600 processors raises an instruction error exception. A new basecase instruction is introduced that also raises this exception. Once executed, the control flow is transferred from the user program to the system instruction error exception handler.

The SWI instruction is a single operand instruction in the same class as the SLEEP and BRK instructions and takes no operands or flags. The SWI instruction cannot immediately follow a BRcc or BBITn instruction.

Branch Instructions

Due to the pipeline in the ARC 600 processor, the branch instruction does not take effect immediately, but after a one cycle delay. The execution of the immediately following instruction after the branch can be controlled. The following instruction is said to be in the *delay slot*. The modes for specifying the execution of the delay slot instruction are indicated by the optional .d field according to the following table.

Mode	Operation
ND	Only execute the next instruction when *not* jumping (default)
D	Always execute the next instruction

Since the execution of the instruction that is in the delay slot is controlled by the delay slot mode, it should never be the target of any branch or jump instruction.

The condition codes that are available for conditional branch instructions are shown in Table 6.2.

Branch instructions

Conditional Branch (Bcc) has a branch range of ±1MB, whereas unconditional branch (B) has larger range of ±16MB. The branch target address is 16-bit aligned.

The syntax of the branch instruction is shown below.

Bcc<.d>	s21	*(branch if condition is true)*
B<.d>	s25	*(unconditional branch far)*
B_S	s10	*(unconditional branch)*
BEQ_S	s10	
BNE_S	s10	
BGT_S	s7	
BGE_S	s7	
BLT_S	s7	
BLE_S	s7	
BHI_S	s7	
BHS_S	s7	
BLO_S	s7	
BLS_S	s7	

Branch and link instructions

Conditional Branch and Link (BLcc) has a branch range of ±1MB, whereas unconditional Branch and Link (BL) has larger range of ±16MB. The target address must be 32-bit aligned.

The syntax of the branch and link instruction is shown below.

BLcc<.d>	s21	*(branch if condition is true)*
BL<.d>	s25	*(unconditional branch far)*
BL_S	s13	*(unconditional branch)*

Branch on compare/bit test register-register

Branch on Compare (BRcc) and Branch on Bit Test (BBIT0, BBIT1) have a branch range of ±256B. The branch target address is 16-bit aligned.

The BRcc instruction is similar in execution to a normal compare instruction (CMP) with the addition that a branch occurs if the condition is met. No flags are updated and no ALU result is written back to the register file. A limited set of condition code tests are available for the BRcc instruction as shown in the following table. Note that additional condition code tests are available through the effect of reversing the operands, as shown at the end of the table.

Mnemonic	**Condition**
BREQ	Branch if b-c is equal
BRNE	Branch if b-c is not equal
BRLT	Branch if b-c is less than
BRGE	Branch if b-c is greater than or equal
BRLO	Branch if b-c is lower than
BRHS	Branch if b-c is higher than or same
BBIT0	Branch if bit c in register b is clear
BBIT1	Branch if bit c in register b is set

Table 5.8 Branch on compare/test mnemonics

Mnemonic	**Condition**
BRGT b,u6,s9	*Branch if b-c is greater than (encode as BRLT c,b,s9)*
BRLE b,u6,s9	*Branch if b-c is less than or equal (encode as BRGE c,b,s9)*
BRHI b,u6,s9	*Branch if b-c is higher than (encode as BRLO c,b,s9)*
BRLS b,u6,s9	*Branch if b-c is lower than or same (encode as BRHS c,b,s9)*

Table 5.9 Branch on compare pseudo mnemonics, register-register

Assembler pseudo-instructions for missing conditions using immediate data, are shown below. Note that these versions have a reduced immediate range of 0 to 62 instead of 0 to 63:

Mnemonic	Condition
BRGT b,u6,s9	*Branch if b-u6 is greater than (encode as BRGE b,u6+1,s9)*
BRLE b,u6,s9	*Branch if b-u6 is less than or equal (encode as BRLT b,u6+1,s9)*
BRHI b,u6,s9	*Branch if b-u6 is higher than (encode as BRHS b,u6+1,s9)*
BRLS b,u6,s9	*Branch if b-u6 is lower than or same (encode as BRLO b,u6+1,s9)*

Table 5.10 Branch on compare pseudo mnemonics, register-immediate

The syntax of the branch on compare and branch on bit test instructions are shown below.

BRcc<.d>	b,c,s9	*(branch if reg-reg compare is true, swap regs if inverse condition required)*
BRcc<.d>	b,u6,s9	*(branch if reg-immediate compare is true, use "immediate+1" if a missing condition is required)*
BRcc	b,limm,s9	*(branch if reg-limm compare is true)*
BRcc	limm,c,s9	*(branch if limm-reg compare is true)*
BREQ_S	b,0,s8	*(branch if register is 0)*
BRNE_S	b,0,s8	*(branch if register is non-zero)*
BBIT0<.d>	b,u6,s9	*(branch if bit u6 in reg b is clear)*
BBIT1<.d>	b,u6,s9	*(branch if bit u6 in reg b is set)*
BBIT0<.d>	b,c,s9	*(branch if bit c in reg b is clear)*
BBIT1<.d>	b,c,s9	*(branch if bit c in reg b is set)*

Example showing delay slot usage:

```
        ; if r0=2, r1=2
        brne           r0,r1,ok1      ; r0=r1, no branch to "ok1"
        add            r2,r2,1        ; delay slot 1 - executed
        add            r3,r3,1        ; delay slot 2 - executed
        add            r4,r4,1        ; executed
ok1:
        ; if r0=2, r1=3
        brne           r0,r1,ok2      ; r0 != r1, branch to "ok2"
        add            r2,r2,1        ; delay slot 1 - killed
        add            r3,r3,1         ; delay slot 2 - killed
        add            r4,r4,1         ; not fetched
ok2:
        ; if r0=2, r1=2
        brne.d  r0,r1,ok3              ; r0=r1, no branch to "ok3"
        add            r2,r2,1         ; delay slot 1 - executed
        add            r3,r3,1         ; delay slot 2 - executed
        add            r4,r4,1         ; executed
ok3:
        ; if r0=2, r1=3
        brne.d  r0,r1,ok4              ; r0 != r1, branch to "ok4"
        add            r2,r2,1         ; delay slot 1 - executed
        add            r3,r3,1         ; delay slot 2 - killed
        add            r4,r4,1         ; not fetched
ok4:
```

Jump Instructions

A jump instruction does not take effect immediately, but after a delay of one instruction. The execution of the immediately following instruction after the jump can be controlled. The following instruction is said to be in the *delay slot*. The modes for specifying the execution of the delay slot instruction are indicated by the optional .d field according to the following table.

Mode	Operation
ND	Only execute the next instruction when *not* jumping (default)
D	Always execute the next instruction

Since the execution of the instruction that is in the delay slot is controlled by the delay slot mode, it should never be the target of any branch or jump instruction.

NOTE If the jump instruction is used with long immediate data then the delay slot execution mechanism does not apply.

When source registers ILINK1 and ILINK2 are used with the Jump instruction they are treated in a special way to allow flag restoring when returning from interrupt handling routines or exceptions handling routines.

Summary of jumps and special format instructions

Instruction	Operation	Description
Jcc	pc ← c	jump
Jcc.D	pc ← c	jump with delay slot
JLcc	blink ← next_pc; pc ← c	jump and link

JLcc.D	blink ← next_pc; pc ← c	jump and link with delay slot

Table 5.11 Basecase Jump Instructions

Syntax for jumps and special format instructions

Jump instructions can target any address within the full memory address map, but the target address is 16-bit aligned.

The syntax for the jump and special format instructions is similar to the basecase ALU operation syntax, but only source operand 2 is used.

The Jump instruction syntax is:

Jcc<.d>	[c]	*(PC = c)*
Jcc	limm	*(PC = limm)*
Jcc<.d>	u6	*(PC = u6)*
J<.d>	s12	*(PC = s12)*
Jcc.F	[ILINK1]	*(PC = ILINK1: STATUS32 = STATUS32_L1)*
Jcc.F	[ILINK2]	*(PC = ILINK2: STATUS32 = STATUS32_L2)*
J_S<.d>	[b]	*(reduced set of registers)*
J_S<.d>	[blink]	*(PC = BLINK)*
JEQ_S	[blink]	*(PC = BLINK)*
JNE_S	[blink]	*(PC = BLINK)*

Jump and Link instruction syntax is:

JLcc<.d>	[c]	*(PC = c: BLINK = next_pc)*
JLcc	limm	*(PC = limm: BLINK = next_pc)*
JLcc<.d>	u6	*(PC = u6: BLINK = next_pc)*
JL<.d>	s12	*(PC = s12: BLINK = next_pc)*
JL_S<.d>	[b]	*(reduced set of registers)*

Zero Overhead Loop

The ARC 600 processor has the ability to perform loops without any delays being incurred by the count decrement or the end address comparison. Zero delay loops are set up with the registers LP_START, LP_END and LP_COUNT. LP_START and LP_END can be directly manipulated with the LR and SR instructions and LP_COUNT can be manipulated in the same way as registers in the core register set.

The special instruction LP is used to set up the LP_START and LP_END in a single instruction. The LP instruction is similar to the branch instruction. Loops can be conditionally entered. If the condition code test for the LP instruction returns *false,* then a branch occurs to the address specified in the LP instruction. The branch target address is 16-bit aligned. If the condition code test is *true,* then the address of the next instruction is loaded into LP_START register and the LP_END register is loaded with the address defined in the LP instruction.

The loop mechanism is always active and the registers used by the loop mechanism are set up with the LP instruction. As LP_END is set to 0 upon reset, it is not advisable to execute an instruction placed at the end of program memory space (0xFFFFFFFC or 0xFFFFFFFE) as this will trigger the LP mechanism if no other LP has been set up since reset. Also, caution is needed if code is copied or overlaid into memory. Before executing such code LP_END should be initialised to a safe value (i.e. 0) to prevent accidental LP triggering.

The processor determines the next address from which to fetch an instruction according to whether there is a branch or jump being executed and whether the current program counter (cPC) has reached the last instruction of a zero overhead loop. If a branch or jump instruction is taken then the target of that instruction always defines the next PC. Whenever current PC reaches the last instruction of a zero overhead loop the LP_COUNT register is decremented. This happens regardless of whether the loop will iterate or whether the loop will terminate.

On reaching the last instruction of a zero overhead loop the processor will examine the LP_COUNT register. If it is not equal to either 0 or 1, and there is no taken branch at that location, then the program counter will be set to LP_START.

This is illustrated in Figure 5.6.

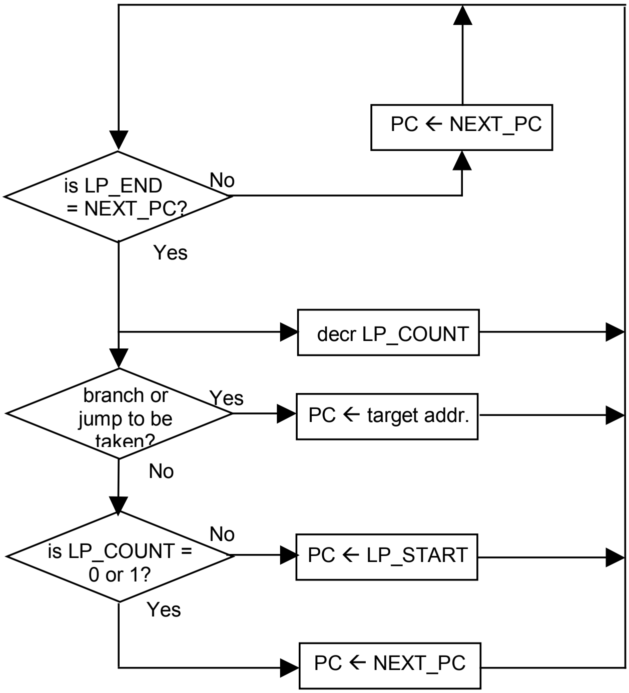


Figure 5.6 Loop Detection and Update Mechanism

Loop Instruction

The loop instruction, LP, has a special syntax that ignores the destination field, and only requiures one source operand. The source operand is a 16-bit aligned target address value.

Register-register

The *General operations register-register* format is not implemented for the LP instruction.

Register with unsigned 6-bit immediate

The *General operations register with unsigned 6-bit immediate* format is implemented, where the destination field A is ignored, the B field is ignored and the immediate field is used as the source value. The source value is a 16-bit aligned address, which provides the following syntax for the LP instruction:

LP u7 *(a = ignored, b= ignored, u7=source. Redundant format, see Conditional register with unsigned 6-bit immediate format.)*

Register with signed 12-bit immediate

The *General operations register with signed 12-bit immediate* format is implemented, where the B field is ignored and the immediate field is used as the source value. The source value is a 16-bit aligned address, which provides the following syntax for the LP instruction:

LP s13 *(b = ignored, s13=source. aux_reg[LP_END] = pc + s13 and aux_reg[LP_START] = next_pc)*

Conditional Register

The *General operations conditional register* format is not implemented for the LP instruction.

Conditional register with unsigned 6-bit immediate

The *General operations conditional register with unsigned 6-bit immediate* format is implemented. where the B field is ignored and the immediate field is used as the source value. The source value is a 16-bit aligned address, which provides the following syntax for the LP instruction:

LP<.cc> u7 *(b=ignored, u7=source. if cc false pc = pc + u7; if cc true aux_reg[LP_END] = pc + u7 and aux_reg[LP_START] = next_pc)*

The use of zero delay loops is illustrated in Figure 5.7.

```
MOV    LP_COUNT,2    ; do loop 2 times (flags not set)
LP     loop_end      ; set up loop mechanism to work
```

```
                                 ; between loop_in and loop_end
loop_in:     LR     r0,[r1]      ; first instruction
                                 ; in loop
             ADD    r2,r2,r0     ; sum r0 with r2
             BIC    r1,r1,4      ; last instruction
                                 ; in loop
loop_end:
             ADD    r19,r19,r20  ; first instruction after loop
```

Figure 5.7 Example loop code

In order that the zero delay loop mechanism works as expected, there are certain affects that the user should be aware of.

Single instruction loops

The LP instruction can only set up loops containing at least two instruction words. The LP instruction can be used to set up a loop containing a single instruction that references long immediate data – since it has two instruction words:

```
             LP     loop_end           ;
loop_in:     ADD    r22,r22,0x00010000 ; single instruction in
loop
loop_end:
             ADD    r19,r19,r20        ; first instruction after
loop
```

Figure 5.8 Valid Single Instruction Loop

The LP instruction must not be used to set up loops with a single instruction word.

However, if the user wishes to set up a loop containing only a single instruction word, then the LP_START and LP_END registers can be set explicitly using SR instructions. Figure 5.9 gives an example. The loop rules specify that a minimum of three instruction words must be *fetched* after an SR write to LP_START or LP_END and the end of the loop – hence in this case two NOP instructions are included for padding.

```
             MOV    LP_COUNT,5     ; no. of times to do loop
             MOV    r0,doploop     ; load START loop address
             MOV    r1,doploopend ; load END loop address
             SR     r0,[LP_START]; set up loop START register
             SR     r1,[LP_END]  ; set up loop END register
             NOP                 ; allow time to update regs
             NOP                 ; can move useful instrs. here
doploop:     OR     r21,r22,r23  ; single instruction in loop
doploopend: ADD r19,r19,r20      ; first instruction after loop
```

Figure 5.9 Setting up a Single Instruction Loop

Special care must be taken when directly manipulating LP_START and LP_END to ensure that the values written refer to the first address occupied by an instruction. Unpredictable behavior will result when LP_START or LP_END are set to point to any other locations.

Loop count register

Unlike other core registers, the loop count register does not support short cutting (data forwarding). In order to guarantee the new value is read, there must be at least 1 instruction word fetched between an instruction writing LP_COUNT and one reading LP_COUNT.

```
      MOV    LP_COUNT,r0   ; update loop count register
      MOV    r1,LP_COUNT   ; old value of LP_COUNT
      MOV    r1,LP_COUNT   ; new value of LP_COUNT
      MOV    r1,LP_COUNT   ; new value of LP_COUNT
```

Figure 5.10 Reading Loop Counter after Writing

In order for the loop mechanism to work properly, the loop count register must be set up with at least 4 instruction words fetched after the writing instruction and before the end of the loop. In the example in Figure 5.11, the MOV instruction does not comply with the rule – there are only three instruction words (LP, OR, AND) fetched before the end of the loop. The MOV instruction must be followed by a NOP to ensure predictable behavior.

```
             MOV    LP_COUNT,r0; do loop r0 times (flags not set)
             LP    loop_end       ; set up loop mechanism
loop_in:     OR    r21,r22,r23    ; first instruction in loop
             AND   0,r21,23       ; last instruction in loop
loop_end:
             ADD   r19,r19,r20 ; first instruction after loop
```

Figure 5.11 Invalid Loop Count set up

```
             MOV   LP_COUNT,r0 ; do loop r0 times (flags not set)
             NOP               ; allow time for loop count set up
             LP    loop_end       ; set up loop mechanism
loop_in:     OR    r21,r22,r23    ; first instruction in loop
             AND   0,r21,23       ; last instruction in loop
loop_end:
             ADD   r19,r19,r20 ; first instruction after loop
```

Figure 5.12 Valid Loop Count set up

Note the emphasis on the number of instructions *fetched* between the LP_COUNT setup and the end of the loop. Since code flow is not always linear, the programmer must ensure that the rules are complied with even when a branch forms part of the code sequence between the write to LP_COUNT and the end of the loop:

```
             MOV   LP_COUNT,r0 ; do loop r0 times
             BAL    loop_last
             ..
             ..
             LP    loop_end      ; set up loop mechanism
loop_in:     OR    r21,r22,r23   ; first instruction in loop
loop_last:   AND   0,r21,23      ; last instruction in loop
loop_end:
             ADD   r19,r19,r20 ; first instruction after loop
```

Figure 5.13 Invalid Loop Count set up with branch

```
             MOV    LP_COUNT,r0 ; do loop r0 times
             NOP                ; 1
             NOP                ; 2
             BAL    loop_last   ; 3 (loop_last is 4)
             ..
             ..
             LP    loop_end      ; set up loop mechanism
loop_in:     OR    r21,r22,r23   ; first instruction in loop
loop_last:   AND   0,r21,23      ; last instruction in loop
loop_end:
             ADD   r19,r19,r20 ; first instruction after loop
```

Figure 5.14 Valid Loop Count set up with branch

Reading the LP_COUNT register inside a loop is hazardous – multiple rules are overlaid. A previous paragraph describes that the value read from the LP_COUNT will be unpredictable for two instructions following the write. When reading LP_COUNT inside a loop, an additional complication is that the result will be unpredictable if read from the last instruction word position in the loop:

```
         ...
         MOV   r0,LP_COUNT  ; loop count for this iteration
         MOV   r0,LP_COUNT  ; loop count for next iteration
loop_end:
         ADD   r19,r19,r20  ; first instruction after loop
```

Figure 5.15 Reading Loop Counter near Loop Mechanism

LP_COUNT must not be loaded directly from memory

The LP_COUNT register must not be used as the destination of a load or POP instruction. Instead, an intermediary register should be used, as follows:

```
LD     r1,[r0]       ; register loaded from memory
MOV    LP_COUNT, r1  ; LP_COUNT loaded from register
```

Figure 5.16 Correct set-up of LP_COUNT via a register

The example loads a value into an intermediate register before being transferred to LP_COUNT.

Branch and jumps in loops

Jumps and branches without linking or branch delay slots may be used at any position in the loop. The programmer must however be aware of the side-effects on the LP_COUNT register of using branches within a loop, and also of the positions within loops where certain other branch or jump instructions may not be used.

When a branch is used for early termination of a loop, the value of the loop count register after loop exit is undefined under certain circumstances:

1. When a BRcc or BBITn instruction is separated from the end of the loop by less than three instruction word fetches.
2. When any other branch/jump instruction is separated from the end of the loop by less than two instruction word fetches.

There are also rules about where certain branch/jump instructions may be placed within zero-overhead loops.

Loops within loops

One zero-overhead loop may be used inside another provided that the inner loop saves and restores the context of the outer loop and complies with all other rules. An additional rule is that a loop instruction may not be used in either of the last two instruction slots before the end of an existing loop.

Valid instruction regions in loops

To summarize the effect that the loop mechanism has on these special cases see the tables below.

Notes:

- Instruction numbers Insn-N refer to the sequence of instructions slots within a loop – which is not the same as the instruction positions if branches are used within the loop.
- Two instruction slots are taken by instructions with long immediate data – The first position (to which the rules apply) is the instruction, the second is the long immediate data word.

Key:

?^{1} Writes to the loop count register – the number of loop iterations executed before the loop count mechanism takes account of the change is undefined.

?^{2} Reads from the loop count register – the value returned may not be the number of the current loop iteration.

! Loop count register value unpredictable when branch taken to exit early from the loop.

✕ An instruction of this type may not be executed in this instruction slot.

✕^{2} A branch or jump may be placed in this position provided its target is outside the loop. Upon exit the value of LP_COUNT will be one less than the number of iterations executed. A branch or jump may not be placed in this position if its target is inside the loop. If this rule is violated the loop may execute an undefined number of iterations.

n/a Instructions using long immediate data take two slots. Hence the instruction itself cannot be present in the last instruction slot.

The following table covers loop setup, early exit and use of long immediate data:

Loop setups	**Loop Set Up**	**Writing**	**Reading**	**Writing**	**Reading**	**Long**
Immediates	`LP loop_end`	`LP_COUNT`	`LP_COUNT`	`LP_END, LP_START`	`LP_END, LP_START`	**Immediate**
`Loop_st:`						
`Ins1`	...	...	...	...	...	...
`Ins2`	...	...	...	...	...	...
`Ins3`	...	...	...	...	...	...
`...`	...	...	...	...	...	...
`...`	...	...	...	...	...	...
`Insn-4`	...	...	...	...	...	...
`Insn-3`	...	**?**^{1}	...	...	...	...
`Insn-2`	...	**?**^{1}	...	**✕**	...	...
`Insn-1`	**✕**	**?**^{1}	...	**✕**	...	...
`Insn`	**✕**	**?**^{1}	**?**^{2}	**✕**	...	n/a
`Loop_end:`						
`Outins1`						
`Outins2`						

Table 5.12 Loop setup, early exit and long immediate data

The following tables cover use of branch and jump instructions:

Flow (1)	Bcc Jcc [Rn]	BRcc BBITn	BLcc JLcc	Bcc.d Jcc.d J_S.d	BLcc.d JLcc.d JL_S.d	BRcc.d BBITn.d
Loop_st:						
Ins1	...	...	...	...	...	...
Ins2	...	...	...	...	...	...
Ins3	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
Insn-4	...	...	...	...	...	...
Insn-3	...	...	...	...	...	...
Insn-2	...	...	...	...	...	...
Insn-1	...	...	...	✗[2]	✗	✗[2]
Insn	✗[2]	✗[2]	✗	✗	✗	✗
Loop_end:						
Outins1						
Outins2						

Table 5.13 Branch and Jumps in loops, flow(1)

Flow (2)	Jcc limm	JLcc limm	LP other_loop
Loop_st:			
Ins1	...	...	...
Ins2	...	...	...
...	...	...	...
...	...	...	...
Insn-2	...	...	...
Insn-1	✗[2]	✗	✗
Insn	✗	✗	✗
Loop_end:			
Outins1			
Outins2			

Table 5.14 Branch and Jumps in loops, flow(2)

Auxiliary Register Operations

The access to the auxiliary register set is accomplished with the special load register and store register instructions (LR and SR). They work in a similar way to the normal load and store instructions except that the access is accomplished in a single cycle due to the fact that address computation is not carried out and the scoreboard unit is not used. The LR and SR instruction do not cause stalls like the normal load and store instructions but in the same cases that arithmetic and logic instructions would cause a stall.

Access to the auxiliary registers are limited to 32 bit (long word) only and the instructions are *not* conditional.

Instruction	Operation	Description
LR	b ← aux.reg[c]	load from auxiliary register
SR	aux.reg[c] ← b	store to auxiliary register

Table 5.15 Auxiliary Register Operations

Load from auxiliary register

The load from auxiliary register instruction, LR, has one source and one destination register. The LR instruction is not a conditional instruction and uses the *General operations register-register* and *General operations register with signed 12-bit immediate* formats to provide the following syntax:

```
LR      b,[c]
LR      b,[limm]
LR      b,[s12]
```

Store to auxiliary register

The store to auxiliary register instruction, SR, has two source registers only. The SR instruction is not a conditional instruction and uses the *General operations register-register* and *General operations register with signed 12-bit immediate* formats to provide the following syntax:

```
SR      b,[c]
```

SR	b,[limm]	*(c=limm)*
SR	b,[s12]	
SR	limm,[c]	*(b=limm)*
SR	limm,[s12]	*(b=limm)*

Load/Store Instructions

The transfer of data to and from memory is accomplished with the load and store commands (LD, ST). It is possible for these instructions to write the result of the address computation back to the address source register, pre or post calculation. This is accomplished with the optional address write-back suffices: .A, .AW, and .AB. Addresses are interpreted as byte addresses unless the scaled address mode is used, as indicated by the address suffix .AS. Note that using the scaled address mode with 8-bit data size (LDB.AS or STB.AS) has undefined behavior and should not be used. Note that if the offset is not required during a load or store, the value encoded will be set to 0.

The size of the data for a Load or Store is indicated by Load-Byte instruction (LDB), Load-Word instruction (LDW), Store-Byte instruction (STB) and Store-Word instruction (STW). LD or ST with no size suffix indicates 32-bit data. Byte and word loads are zero or sign extended to 32-bits by using the sign extend suffix: .X. Note that using the sign extend suffix on the LD instruction with a 32-bit data size is undefined and should not be used.

Loads are passed to the memory controller with the appropriate address, and the register that is the destination of the load is tagged to indicate that it is waiting for a result, as loads take a minimum of one cycle to complete. If an instruction references the tagged register before the load has completed, the pipeline will stall until the register has been loaded with the appropriate value. For this reason it is not recommended that loads be immediately followed by instructions that reference the register being loaded. Delayed loads from memory will take a variable amount of time depending upon the presence of cache and the type of memory that is available to the memory controller. Consequently, the number of instructions to be executed in between the load and the instruction using the register will be application specific.

Stores are passed to the memory controller, which will store the data to memory when it is possible to do so. The pipeline may be stalled if the memory controller cannot accept any more buffered store requests.

If a data-cache is available in the memory controller, the load and store instructions can bypass the use of that cache. When the suffix .DI is used the cache is bypassed and the data is loaded directly from or stored directly to the memory. This is particularly useful for shared data structures in main memory, for the use of memory-mapped I/O registers, or for bypassing the cache to stop the cache being updated and overwriting valuable data that has already been loaded in that cache.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

The address write-back modes are shown in the following table.

AA bits	Address mode	Memory address used	Register Value write-back
00	No write-back	Reg + offset	no write-back
01	.A or .AW	Reg + offset	Reg + offset Register updated pre memory transaction.
10	.AB	Reg	Reg + offset Register updated post memory transaction.
11	.AS Scaled , no write-back .AS	Reg + (offset << data_size) Note that using the scaled address mode with 8-bit data size (LDB.AS or STB.AS) has undefined behavior and should not be used.	no write-back

Table 5.16 LD/ST <.aa> address mode bits

Load

Unlike basecase ALU operations, the load instruction cannot target a long immediate value as the target register. Two syntaxes are available depending on how the address is calculated: register-register and register-offset. The syntax for the load instruction is:

LD<zz><.x><.aa><.di>	a,[b]	*(uses ld a,[b,0])*
LD<zz><.x><.aa><.di>	a,[b,s9]	
LD<zz><.x><.di>	a,[limm]	*(= ld a,[limm,0])*
LD<zz><.x><.aa><.di>	a,[b,c]	
LD<zz><.x><.aa><.di>	a,[b,limm]	
LD<zz><.x><.di>	a,[limm,c]	
LD_S	a, [b, c]	
LDB_S	a, [b, c]	
LDW_S	a, [b, c]	
LD_S	c, [b, u7]	*(u7 offset is 32-bit aligned)*
LDB_S	c, [b, u5]	
LDW_S<.x>	c, [b, u6]	*(u6 offset is 16-bit aligned)*
LD_S	b, [SP, u7]	*(u7 offset is 32-bit aligned)*
LDB_S	b, [SP, u7]	*(u7 offset is 32-bit aligned)*
LD_S	r0, [GP, s11]	*(s11 offset is 32-bit aligned)*
LDB_S	r0, [GP, s9]	
LDW_S	r0, [GP, s10]	*(s10 offset is 16-bit aligned)*
LD_S	b, [PCL, u10]	*(u10 offset is 32-bit aligned)*

Store register with offset

Store register+offset instruction syntax:

ST<zz><.aa><.di>	c,[b]	*(use st c,[b,0])*
ST<zz><.aa><.di>	c,[b,s9]	
ST<zz><.di>	c,[limm]	*(= st c,[limm,0])*
ST<zz><.aa><.di>	limm,[b,s9]	
ST_S	b, [SP, u7]	*(u7 offset is 32-bit aligned)*

STB_S	b, [SP, u7]	*(u7 offset is 32-bit aligned)*
ST_S	c, [b, u7]	*(u7 offset is 32-bit aligned)*
STB_S	c, [b, u5]	
STW_S	c, [b, u6]	*(u6 offset is 16-bit aligned)*

Stack pointer operations

The ARC 600 processor provides stack pointer functionality through the use of the stack pointer core register (SP). Push and pop operations are provided through normal Load and Store operations in the 32-bit instruction set, and specific instructions in the 16-bit instruction set. The instructions syntax for push operations on the stack is:

ST.AW	c,[SP,-4]	*(Push c onto the stack)*
PUSH_S	b	*(Push b onto the stack)*
PUSH_S	BLINK	*(Push BLINK onto the stack)*

The instructions syntax for pop operations on the stack is:

LD.AB	a,[SP,+4]	*(Pop top item of stack to a)*
POP_S	b	*(Pop top item of stack to b)*
POP_S	BLINK	*(Pop top item of stack to BLINK)*

The following instructions are also available in 16-bit instruction format, for working with the stack:

LD_S, LDB_S, ST_S, STB_S, ADD_S, SUB_S, MOV_S, and CMP_S.

ARCompact Extension Instructions

These operations are generally of the form **a ←b op c** where the destination (a) is replaced by the result of the operation (op) on the operand sources (b and c). All extension instructions can be conditional and/or set the flags.

Syntax for generic extension instructions

If the destination register is set to an absolute value of "0" then the result is discarded and the operation acts like a NOP instruction. A long immediate (limm) value can be used for either source operand 1 or source operand 2. The generic extension instruction format is:

op<.f>	a,b,c	
op<.f>	a,b,u6	
op<.f>	b,b,s12	
op<.cc><.f>	b,b,c	
op<.cc><.f>	b,b,u6	
op<.f>	a,limm,c	*(if b=limm)*
op<.f>	a,limm,u6	
op<.f>	0,limm,s12	
op<.cc><.f>	0,limm,c	
op<.cc><.f>	0,limm,u6	
op<.f>	a,b,limm	*(if c=limm)*
op<.cc><.f>	b,b,limm	
op<.f>	a,limm,limm	*(if b=c=limm)*
op<.cc><.f>	0,limm,limm	
op<.f>	0,b,c	*(if a=0)*
op<.f>	0,b,u6	
op<.f>	0,limm,c	*(if a=0, b=limm)*
op<.f>	0,limm,u6	
op<.f>	0,b,limm	*(if a=0, c=limm)*
op<.f>	0,limm,limm	*(if a=0, b=c=limm)*
op_S	b,b,c	

Syntax for Single operand extension instructions

Single source operand instructions are supported for extension instructions. ingle operand instruction syntax is:

```
op<.f>          b,c
op<.f>          b,u6
op<.f>          b,limm
op<.f>          0,c
op<.f>          0,u6
op<.f>          0,limm
op_S            b,c
```

Syntax for Zero operand extension instructions

Zero operand instruction syntax is:

```
op<.f>          c
op<.f>          u6
op<.f>          limm
op_S
```

Optional Extensions Library

The extensions library consists of a number of components that can be used to add functionality to the ARC 600 processor.

These components are function units, which are interfaced to the ARC 600 processor through the use of extension instructions or registers.

The library currently consists of the following components:

- 32 bit Multiplier, small (10 cycle) implementation
- 32 bit Multiplier, fast (4 cycle) implementation
- Normalize (find-first-bit) instruction
- Swap instruction

Multiply 32 X 32

Two versions of the scoreboarded 32x32 multiplier function are available, 'fast' and 'small', taking four and ten cycles respectively. The full 64-bit result is available to be read from the core register set. The middle 32 bits of the 64-bit result are also available. The multiply is scoreboarded in such a way that if a multiply is being carried out, and if one of the result registers is required by another ARC 600 instruction, the processor will stall until the multiply has finished. The destination is always ignored for the multiply instrcution and thus the syntax for the multiply instructions can optionally supply a "0" as the destination register.

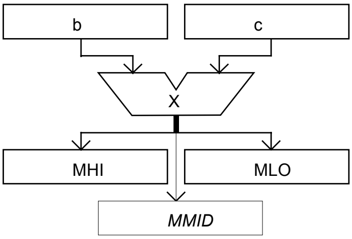


Figure 5.17 32x32 Multiply

Register-register

The *General operations register-register* format is implemented for the multiply instructions. The destination register is always encoded as an immediate operand.

The following redundant syntax formats are provided for the multiply instructions:

MUL64	<0,>b,c	*(a = limm, b = source 1, c = source 2. Redundant format see Conditional Register format)*
MUL64	<0,>b,limm	*(a = limm, b limm, c = source 2. Redundant format see Conditional Register format)*
MUL64	<0,>limm,c	*(a = limm, b = source 1, c = limm. Redundant format see Conditional Register format)*
MUL64	<0,>limm,limm	*(a = limm, b = limm, c = limm. Redundant format see Conditional Register format)*
MULU64	<0,>b,c	*(a = limm, b = source 1, c = source 2. Redundant format see Conditional Register format)*
MULU64	<0,>b,limm	*(a = limm, b = limm, c = source 2. Redundant format see Conditional Register format)*
MULU64	<0,>limm,c	*(a = limm, b = source 1, c = limm. Redundant format see Conditional Register format)*
MULU64	<0,>limm,limm	*(a = limm, b = limm, c = limm. Redundant format see Conditional Register format)*

Register with unsigned 6-bit immediate

The *General operations register with unsigned 6-bit immediate* format is implemented for the multiply instructions. The destination register is always encoded as an immediate operand. The following redundant syntax formats are provided for the multiply instructions:

MUL64	<0,>b,u6	*(a = limm, b = source 1, u6 = source 2. Redundant format see Conditional register with unsigned 6-bit immediate format)*
MUL64	<0,>limm,u6	*(a = limm, b = limm, u6 = source 2. Redundant format see Conditional register with unsigned 6-bit immediate format)*
MULU64	<0,>b,u6	*(a = limm, b = source 1, u6 = source 2. Redundant format see Conditional register with unsigned 6-bit immediate format)*

MUL64	<0,>limm,u6	*(a = limm, b = limm, u6 = source 2. Redundant format see Conditional register with unsigned 6-bit immediate format)*

Register with signed 12-bit immediate

The *General operations register with signed 12-bit immediate* format provides the following syntax for the multiply instructions:

MUL64	<0,>b,s12	*(b = source 1, s12 = source 2)*
MUL64	<0,>limm,s12	*(b = limm, s12 = source 2)*
MULU64	<0,>b,s12	*(b = source 1, s12 = source 2)*
MULU64	<0,>limm,s12	*(b = limm, s12 = source 2)*

Conditional Register

The *General operations conditional register* format provides the following syntax for the multiply instructions:

MUL64<.cc>	<0,>b,c	*(b = source 1, c = source 2)*
MUL64<.cc>	<0,>b,limm	*(b = source 1, c = limm)*
MUL64<.cc>	<0,>limm,c	*(b = limm, c = source 2)*
MUL64<.cc>	<0,>limm,limm	*(b = limm, c = limm)*
MULU64<.cc>	<0,>b,c	*(b = source 1, c = source 2)*
MULU64<.cc>	<0,>b,limm	*(b = source 1, c = limm)*
MULU64<.cc>	<0,>limm,c	*(b = limm, c = source 2)*
MULU64<.cc>	<0,>limm,limm	*(b = limm, c = limm. Not useful format)*

Conditional register with unsigned 6-bit immediate

The *General operations conditional register with unsigned 6-bit immediate* format provides the following syntax for the multiply instructions:

MUL64<.cc>	<0,>b,u6	*(b = source 1, u6 = source 2)*
MUL64<.cc>	<0,>limm,u6	*(b = limm, u6 = source 2. Not useful format)*

16-bit instruction, multiply

The unsigned multiply operation does not have a 16-bit instruction equivalent. The *General Register Format Instructions, 0x0F, [0x00 - 0x1F]* format provides the following syntax for the signed multiply

MUL64_S <0,>b,c

Quick exit for conditional multiplies

If an instruction condition placed on a MUL64 or MULU64 is found to be false, the multiply will not be performed, and the instruction will complete on the same cycle without affecting the values stored in the multiply result registers.

Reading and Pre-loading the 32X32 multiply results

The results are accessed via the read-only extension core registers MLO, MMID and MHI.

The extension auxiliary register MULHI is used to restore the multiply result register if the multiply has been used, for example, by an interrupt service routine. Note, that no interlock is provided to stall writes when a multiply is taking place. For this reason, the user must ensure that the multiply has completed before writing the MULHI register. Reading one of the scoreboarded multiplier result registers can easily do this.

The lower part of the multiply result register can be restored by multiplying the desired value by 1.

To read the upper and lower parts of the multiply results

```
MOV           r1,mlo ;put lower result in r1
MOV           r2,mhi ;put upper result in r2
```

To restore the multiply results

```
MULU64   r1,1   ;restore lower result
MOV    0,mlo  ;wait until multiply complete. N.B causes
       ;processor to stall,until multiplication is
                ;finished
SR  r2,[mulhi]  ;restore upper result
```

NORM Instruction

The NORM instruction gives the normalization integer for the signed value in the operand. The normalization integer is the amount by which the operand should be shifted left to normalize it as a 32-bit signed integer. To find the normalization integer of a 32-bit register by using software without a NORM instruction, requires many ARC 600 instruction cycles.

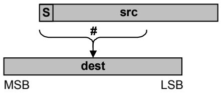


Figure 5.18 Norm Instruction

Uses for the NORM instruction include:

- Acceleration of single bit shift division code, by providing a fast 'early out' option.
- Reciprocal and multiplication instead of division
- Reciprocal square root and multiplication instead of square root

The syntax for the normalize instruction is:

```
NORM<.f>        b,c
NORM<.f>        b,u6
NORM<.f>        b,limm
NORM<.f>        0,c
NORM<.f>        0,u6
NORM<.f>        0,limm

NORMW<.f>       b,c
NORMW<.f>       b,u6
NORMW<.f>       b,limm
NORMW<.f>       0,c
NORMW<.f>       0,u6
NORMW<.f>       0,limm
```

SWAP Instruction

The swap instruction is a very simple extension that can be used with the multiply-accumulate block. It exchanges the upper and lower 16-bit of the source value, and stores the result in a register. This is useful to prepare values for multiplication, since the multiply-accumulate block takes its 16-bit source values from the upper 16 bits of the 32-bit values presented.

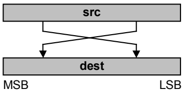


Figure 5.19 SWAP Instruction

The syntax for the swap instruction is:

SWAP<.f>	b,c
SWAP<.f>	b,u6
SWAP<.f>	b,limm
SWAP<.f>	0,c
SWAP<.f>	0,u6
SWAP<.f>	0,limm

Chapter 6 — 32-bit Instruction Formats Reference

This chapter shows the available encoding formats for the 32-bit instructions. Some encodings define instructions that are also defined in other encoding formats. Chapter 5 — *Instruction Set Summary* lists and notes the redundant formats. The processor implements all redundant encoding formats. A listing of syntax and encoding that excludes the redundant formats is contained in Chapter 9 — *Instruction Set Details*.

A complete list of the major opcodes is shown in Table 6.1.

Major Opcode	**Instruction and/or type**	**Notes**	**Type**
0x00	Bcc	Branch	32-bit
0x01	BLcc, BRcc	Branch and link conditional Compare-branch conditional	32-bit
0x02	LD *register + offset*	Delayed load	32-bit
0x03	ST *register + offset*	Buffered store	32-bit
0x04	op a,b,c	ARC 32-bit basecase instructions	32-bit
0x05	op a,b,c	ARC 32-bit extension instructions	32-bit
0x06	op a,b,c	ARC 32-bit extension instructions	32-bit
0x07	op a,b,c	User 32-bit extension instructions	32-bit
0x08	op_S b,b,c	ARC 16-bit extension instructions	16-bit
0x09	op_S b,b,c	ARC 16-bit extension instructions	16-bit
0x0A	op_S b,b,c	User 16-bit extension instructions	16-bit
0x0B	op_S b,b,c	User 16-bit extension instructions	16-bit
0x0C	LD_S / LDB_S / LDW_S / ADD_S a,b,c	Load/add register-register	16-bit
0x0D	ADD_S / SUB_S / ASL_S / LSR_S c,b,u3	Add/sub/shift immediate	16-bit

0x0E	MOV_S / CMP_S / ADD_S b,h / b,b,h		One dest/source can be any of r0-r63	16-bit
0x0F	op_S	b,b,c	General ops/ single ops	16-bit
0x10	LD_S	c,[b,u7]	Delayed load (32-bit aligned offset)	16-bit
0x11	LDB_S	c,[b,u5]	Delayed load (8-bit aligned offset)	16-bit
0x12	LDW_S	c,[b,u6]	Delayed load (16-bit aligned offset)	16-bit
0x13	LDW_S.X	c,[b,u6]	Delayed load (16-bit aligned offset)	16-bit
0x14	ST_S	c,[b,u7]	Buffered store (32-bit aligned offset)	16-bit
0x15	STB_S	c,[b,u5]	Buffered store (8-bit aligned offset)	16-bit
0x16	STW_S	c,[b,u6]	Buffered store (16-bit aligned offset)	16-bit
0x17	*OP*_S	b,b,u5	Shift/subtract/bit ops	16-bit
0x18	LD_S / LDB_S / ST_S / STB_S / ADD_S / PUSH_S / POP_S		Sp-based instructions	16-bit
0x19	LD_S / LDW_S / LDB_S / ADD_S		Gp-based ld/add (data aligned offset)	16-bit
0x1A	LD_S	b,[PCL,u10]	Pcl-based ld (32-bit aligned offset)	16-bit
0x1B	MOV_S	b,u8	Move immediate	16-bit
0x1C	ADD_S / CMP_S	b,u7	Add/compare immediate	16-bit
0x1D	BRcc_S	b,0,s8	Branch conditionally on reg z/nz	16-bit
0x1E	Bcc_S	s10/s7	Branch conditionally	16-bit
0x1F	BL_S	s13	Branch and link unconditionally	16-bit

Table 6.1 Major opcode Map, 32-bit and 16-Bit instructions

Condition Code Tests

The following table shows the codes used for condition code tests.

Code	**Mnemonic**	**Condition**	**Test**
Q field			
0x00	AL, RA	Always	1
0x01	EQ , Z	Zero	Z
0x02	NE , NZ	Non-Zero	/Z
0x03	PL , P	Positive	/N

Code Q field	Mnemonic	Condition	Test
0x04	MI , N	Negative	N
0x05	CS , C, LO	Carry set, lower than (unsigned)	C
0x06	CC , NC, HS	Carry clear, higher or same (unsigned)	/C
0x07	VS , V	Over-flow set	V
0x08	VC , NV	Over-flow clear	/V
0x09	GT	Greater than (signed)	(N and V and /Z) or (/N and /V and /Z)
0x0A	GE	Greater than or equal to (signed)	(N and V) or (/N and /V)
0x0B	LT	Less than (signed)	(N and /V) or (/N and V)
0x0C	LE	Less than or equal to (signed)	Z or (N and /V) or (/N and V)
0x0D	HI	Higher than (unsigned)	/C and /Z
0x0E	LS	Lower than or same (unsigned)	C or Z
0x0F	PNZ	Positive non-zero	/N and /Z

Table 6.2 Condition codes

Branch Jump Delay Slot Modes

The following table shows the codes used for delay slot modes on Branch and Jump instructions.

N Bit	Mode	Operation
0	ND	Only execute the next instruction when *not* jumping (default)
1	D	Always execute the next instruction

Table 6.3 Delay Slot Modes

Load Store Address Write-back Modes

The following table shows the codes used for address write-back modes in Load and Store instructions.

AA bits	Address mode	Memory address used	Register Value write-back
00	No write-back	Reg + offset	no write-back
01	.A or .AW	Reg + offset	Reg + offset Register updated pre memory transaction.
10	.AB	Reg	Reg + offset Register updated post memory transaction.
11	.AS Scaled , no write-back .AS	Reg + (offset << data_size) Note that using the scaled address mode with 8-bit data size (LDB.AS or STB.AS) has undefined behavior and should not be used.	no write-back

Table 6.4 Address Write-back Modes

Load Store Direct to Memory Bypass Mode

The following table shows the codes used for direct to memory bypass modes in Load and Store instructions.

Di bit	Di Suffix	Access mode
0		Default access to memory
1	DI	Direct to memory, bypassing data-cache (if available)

Table 6.5 Direct to Memory Bypass Mode

Load Store Data Size Mode

The following table shows the codes used for data size modes in Load and Store instructions.

ZZ Code	ZZ Suffix	Access mode
00		Default, Long word
01	B	Byte
10	W	Word
11		*Reserved*

Table 6.6 Load Store Data Size Mode

Load Data Extend Mode

The following table shows the codes used data extend modes in Load instructions.

X bit	X Suffix	Access mode
0		If size is not long word then data is zero extended
1	X	If size is not long word then data is sign extended

Table 6.7 Load Data Extend Mode

Branch Conditionally, 0x00, [0x0]

The target address is 16-bit aligned to target 16-bit aligned instructions. See Table 6.2 for information on condition code test encoding, and Table 6.3 for delay slot mode encoding.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	S[10:1]										0	S[20:11]										N	Q[4:0]				

Syntax:

Bcc<.d>   s21   *(branch if condition is true)*

Branch Unconditional Far, 0x00, [0x1]

The target address is 16-bit aligned to target 16-bit aligned instructions. See Table 6.3 for information on delay slot mode encoding.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	S[10:1]										1	S[20:11]										N	R	S[24:21]			

Syntax:

B<.d>	s25	*(unconditional branch far)*

Branch on Compare Register-Register, 0x01, [0x1, 0x0]

The target address is 16-bit aligned to target 16-bit aligned instructions. See Table 6.3 for information on delay slot mode encoding.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	B[2:0]			S[7:1]							1	S8	B[5:3]			C[5:0]						N	0	I[3:0]			

Syntax:

BRcc<.d>	b,c,s9	*(branch if reg-reg compare is true, swap regs if inverse condition required)*
BRcc	b,limm,s9	*(branch if reg-limm compare is true)*
BRcc	limm,c,s9	*(branch if limm-reg compare is true)*
BBIT0<.d>	b,c,s9	*(branch if bit c in reg b is clear)*
BBIT1<.d>	b,c,s9	*(branch if bit c in reg b is set)*

Sub-opcode **I field (4 bits)**	**Instruction**	**Operation**	**Description**
0x00	BREQ	b - c	Branch if reg-reg is equal
0x01	BRNE	b - c	Branch if reg-reg is not equal
0x02	BRLT	b - c	Branch if reg-reg is less than
0x03	BRGE	b - c	Branch if reg-reg is greater than or equal
0x04	BRLO	b - c	Branch if reg-reg is lower than

Sub-opcode I field (4 bits)	Instruction	Operation	Description
0x05	BRHS	b - c	Branch if reg-reg is higher than or same
0x06			*Reserved*
0x07			*Reserved*
0x08			*Reserved*
0x09			*Reserved*
0x0A			*Reserved*
0x0B			*Reserved*
0x0C			*Reserved*
0x0D			*Reserved*
0x0E	BBIT0	(b and 1<<c) == 0	Branch if bit c in register b is clear
0x0F	BBIT1	(b and 1<<c) != 0	Branch if bit c in register b is set

Table 6.8 Branch on compare/bit test register-register

Branch on Compare/Bit Test Register-Immediate, 0x01, [0x1, 0x1]

The target address is 16-bit aligned to target 16-bit aligned instructions. See Table 6.3 for information on delay slot mode encoding.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	B[2:0]			S[7:1]							1	S8	B[5:3]			U[5:0]						N	1	I[3:0]			

Syntax:

BRcc<.d>	b,u6,s9	*(branch if reg-immediate compare is true, use "immediate+1" if a missing condition is required)*
BBIT0<.d>	b,u6,s9	*(branch if bit u6 in reg b is clear)*
BBIT1<.d>	b,u6,s9	*(branch if bit u6 in reg b is set)*

Sub-opcode **I field (4 bits)**	**Instruction**	**Operation**	**Description**
0x00	BREQ	b - u6	Branch if reg-imm is equal
0x01	BRNE	b - u6	Branch if reg-imm is not equal
0x02	BRLT	b - u6	Branch if reg-imm is less than
0x03	BRGE	b - u6	Branch if reg-imm is greater than or equal
0x04	BRLO	b - u6	Branch if reg-imm is lower than
0x05	BRHS	b - u6	Branch if reg-imm is higher than or same
0x06			*Reserved*
0x07			*Reserved*
0x08			*Reserved*
0x09			*Reserved*
0x0A			*Reserved*
0x0B			*Reserved*
0x0C			*Reserved*
0x0D			*Reserved*
0x0E	BBIT0	(b and 1<<u6) == 0	Branch if bit u6 in register b is clear

0x0F	BBIT1	(b and 1<<u6) != 0	Branch if bit u6 in register b is set

Table 6.9 Branch Conditionally/bit test on register-immediate

Branch and Link Conditionally, 0x01, [0x0, 0x0]

The target address must be 32-bit aligned. See Table 6.2 for information on condition code test encoding, and Table 6.3 for delay slot mode encoding.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	S[10:2]									0	0	S[20:11]										N	Q[4:0]				

Syntax:

BLcc<.d> s21 *(branch if condition is true)*

Branch and Link Unconditional Far, 0x01, [0x0, 0x1]

The target address must be 32-bit aligned. See Table 6.3 for information on delay slot mode encoding.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	S[10:2]									1	0	S[20:11]										N	R	S[24:21]			

Syntax:

BL<.d> s25 *(unconditional branch far)*

Load Register with Offset, 0x02

See Table 6.4, Table 6.5, Table 6.6 and Table 6.7 for information on encoding the Load instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	B[2:0]			S[7:0]								S8	B[5:3]			Di	A	A	Z	Z	X	A[5:0]					

Syntax:

LD<zz><.x><.aa><.di>	a,[b,s9]	
LD<zz><.x><.di>	a,[limm,s9]	*(use ld a,[limm])*
LD<zz><.x><.di>	a,[limm]	*(= ld a,[limm,0])*

Store Register with Offset, 0x03

See Table 6.4, Table 6.5 and Table 6.6 for information on encoding the Store instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	B[2:0]			S[7:0]								S8	B[5:3]			C[5:0]						Di	A	A	Z	Z	R

Syntax:

ST<zz><.aa><.di>	c,[b,s9]	
ST<zz><.di>	c,[limm]	*(= st c,[limm,0])*
ST<zz><.aa><.di>	limm,[b,s9]	

General operations, 0x04, [0x00 - 0x3F]

General operations register-register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	B[2:0]			0	0	I[5:0]						F	B[5:3]			C[5:0]						A[5:0]					

Syntax:

op<.f>	a,b,c	
op<.f>	a,limm,c	*(if b=limm)*
op<.f>	a,b,limm	*(if c=limm)*
op<.f>	a,limm,limm	*(if b=c=limm. Not useful format)*
op<.f>	0,b,c	*(if a=0)*

op<.f>	0,limm,c	*(Redunant format, see General operations conditional register format)*
op<.f>	0,b,limm	*(if a=0, c=limm)*
op<.f>	0,limm,limm	*(if a=0, b=c=limm. Not useful format)*
op<.f>	b,c	*(SOP instruction)*
op<.f>	b,limm	*(SOP instruction)*
op<.f>	0,c	*(SOP instruction)*
op<.f>	0,limm	*(SOP instruction)*
op<.f>	c	*(ZOP instruction)*
op<.f>	limm	*(ZOP instruction)*

General operations register with unsigned 6-bit immediate

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	B[2:0]			0	1	I[5:0]						F	B[5:3]			U[5:0]						A[5:0]					

Syntax:

op<.f>	a,b,u6	
op<.f>	a,limm,u6	*(Not useful format)*
op<.f>	0,b,u6	
op<.f>	0,limm,u6	*(Not useful format)*
op<.f>	b,u6	*(SOP instruction)*
op<.f>	0,u6	*(SOP instruction)*
op<.f>	u6	*(ZOP instruction)*

General operations register with signed 12-bit immediate

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	B[2:0]			1	0	I[5:0]						F	B[5:3]			S[5:0]						S[11:6]					

Syntax:

op<.f>	b,b,s12	
op<.f>	0,limm,s12	*(Not useful format)*

General operations conditional register

31	30	29	28	27	26 25 24	23	22	21 20 19 18 17 16	15	14 13 12	11 10 9 8 7 6	5	4 3 2 1 0
0	0	1	0	0	B[2:0]	1	1	I[5:0]	F	B[5:3]	C[5:0]	0	Q[4:0]

Syntax:

op<.cc><.f>	b,b,c	
op<.cc><.f>	0,limm,c	
op<.cc><.f>	b,b,limm	
op<.cc><.f>	0,limm,limm	*(Not useful format)*

General operations conditional register with unsigned 6-bit immediate

31	30	29	28	27	26 25 24	23	22	21 20 19 18 17 16	15	14 13 12	11 10 9 8 7 6	5	4 3 2 1 0
0	0	1	0	0	B[2:0]	1	1	I[5:0]	F	B[5:3]	U[5:0]	1	Q[4:0]

Syntax:

op<.cc><.f>	b,b,u6	
op<.cc><.f>	0,limm,u6	*(Not useful format)*

Long immediate with general operations

The first long-word is the instruction that contains the long immediate data indicator (register r62). The second long-word is the long immediate (limm) data itself.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Limm[31:0]

Syntax:

limm

Load register-register, 0x04, [0x30 - 0x37]

Load register+register instruction, LD, is specially encoded in major opcode 0x04 in that the normal "F and two mode bits" are replaced by the "D and two A bits" in the instruction word bit[15] and bits[23:22]. The normal "conditional/immediate" mode bits are replaced by addressing mode bits.(N.B. using an immediate value in the "a" field is not allowed):

See Table 6.4, Table 6.5, Table 6.6 and Table 6.7 for information on encoding the Load instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	B[2:0]			A	A	1	1	0	Z	Z	X	Di	B[5:3]			C[5:0]						A[5:0]					

Syntax:

LD<zz><.x><.aa><.di>	a,[b,c]
LD<zz><.x><.aa><.di>	a,[b,limm]
LD<zz><.x><.di>	a,[limm,c]

ALU operations, 0x04, [0x00-0x1F]

Sub-opcode I field (6 bits)	Instruction	Operation	Description
0x00	ADD	a ← b + c	add
0x01	ADC	a ← b + c + C	add with carry
0x02	SUB	a ← b – c	subtract
0x03	SBC	a ← (b – c) - C	subtract with carry
0x04	AND	a ← b and c	logical bitwise AND
0x05	OR	a ← b or c	logical bitwise OR
0x06	BIC	a ← b and not c	logical bitwise AND with invert
0x07	XOR	a ← b exclusive-or c	logical bitwise exclusive-OR

Sub-opcode **I field (6 bits)**	**Instruction**	**Operation**	**Description**
0x08	MAX	a ← b max c	larger of 2 signed integers
0x09	MIN	a ← b min c	smaller of 2 signed integers
0x0A	MOV	b ← c	move. See section: *Move to register instruction*
0x0B	TST	b and c	test
0x0C	CMP	b - c	compare
0x0D	RCMP	c - b	reverse compare
0x0E	RSUB	a ← c - b	reverse subtract
0x0F	BSET	a ← b or 1<<c	bit set
0x10	BCLR	a ← b and not 1<<c	bit clear
0x11	BTST	b and 1<<c	bit test
0x12	BXOR	a ← b xor 1<<c	bit xor
0x13	BMSK	a ← b and ((1<<(c+1))-1)	bit mask
0x14	ADD1	a ← b + (c << 1)	add with left shift by 1
0x15	ADD2	a ← b + (c << 2)	add with left shift by 2
0x16	ADD3	a ← b + (c << 3)	add with left shift by 3
0x17	SUB1	a ← b - (c << 1)	subtract with left shift by 1
0x18	SUB2	a ← b - (c << 2)	subtract with left shift by 2

Sub-opcode I field (6 bits)	Instruction	Operation	Description
0x19	SUB3	a ← b - (c << 3)	subtract with left shift by 3
0x1A			*Reserved*
0x1B			*Reserved*
0x1C			*Reserved*
0x1D			*Reserved*
0x1E			*Reserved*
0x1F			*Reserved*

Table 6.10 ALU Instructions

Special format instructions, 0x04, [0x20 - 0x3F]

Sub-opcode I field (6 bits)	Instruction	Operation	Description
0x20	Jcc	pc ← c	jump
0x21	Jcc.D	pc ← c	jump with delay slot
0x22	JLcc	blink ← next_pc; pc ← c	jump and link
0x23	JLcc.D	blink ← next_pc; pc ← c	jump and link with delay slot
0x24			*Reserved*
0x25			*Reserved*
0x26			*Reserved*
0x27			*Reserved*

Sub-opcode I field (6 bits)	Instruction	Operation	Description

0x28	LPcc	aux.reg[LP_END] ← pc + c
		aux.reg[LP_START] ← next_pc
			loop (16-bit aligned target address)

0x29	FLAG	aux.reg[STATUS32] ← c	set status flags
0x2A	LR	b ← aux.reg[c]	load from auxiliary register. See section:*Load from auxiliary register*
0x2B	SR	aux.reg[c] ← b	store to auxiliary register. See section: *Store to auxiliary register*
0x2C			*Reserved*
0x2D			*Reserved*
0x2E			*Reserved*
0x2F	SOPs	*A field is sub-opcode2*	See section: *Single operand instructions*
0x30...0x37	LD	*Load register-register*	See section: *Load register-register, 0x04, [0x30 - 0x37]*
0x38...0x3F			*Reserved*

Table 6.11 Special Format Instructions

Single operand instructions, 0x04, [0x2F, 0x00 - 0x3F]

The sub-opcode 2 (destination 'a' field) is reserved for defining single source operand instructions when sub-opcode 1 of 0x2F is used.

Sub-opcode2 **A field (6 bits)**	**Instruction**	**Operation**	**Description**
0x00	ASL	b ← c+c	Arithmetic shift left by one
0x01	ASR	b ← asr(c)	Arithmetic shift right by one
0x02	LSR	b ← lsr(c)	Logical shift right by one
0x03	ROR	b ← ror(c)	Rotate right
0x04	RRC	b ← rrc(c)	Rotate right through carry
0x05	SEXB	b ← sexb(c)	Sign extend byte
0x06	SEXW	b ← sexw(c)	Sign extend word
0x07	EXTB	b ← extb(c)	Zero extend byte
0x08	EXTW	b ← extw(c)	Zero extend word
0x09	ABS	b ← abs(c)	Absolute
0x0A	NOT	b ← not(c)	Logical NOT
0x0B	RLC	b ← rlc(c)	Rotate left through carry
0x0C			*Reserved*
0x0D			*Reserved*
0x0E			*Reserved*
0x0F			*Reserved*
0x10			*Reserved*
...			*Reserved*
0x3F	ZOPs	*B field is* *sub-opcode3*	See Zero operand (ZOP) table

Table 6.12 Single Operand Instructions

Zero operand instructions, 0x04, [0x2F, 0x3F, 0x00 - 0x3F]

The sub-opcode 3 (source operand b field) is reserved for defining zero operand instructions when sub-opcode 2 of 0x3F is used.

Sub-opcode3 B field (6 bits)	Instruction	Operation	Description
0x00			*Reserved*
0x01	SLEEP	Sleep	Sleep
0x02	SWI	Swi	Software interrupt
...			*Reserved*
0x3F			*Reserved*

Table 6.13 Zero Operand Instructions

32-bit Extension Instructions, 0x05 - 0x07

Three sets of extension instructions are available as shown in the following table:

Major Opcode	Sub Opcode1	Sub Opcode2	Sub Opcode3	Instruction Usage
[31:27]	**[21:16]** **i-field**	**[5:0]** **a-field**	**[14:12]: [26:24]** **b-field**	
0x05	0x00-0x2E			ARC Cores extension instructions
"	0x30- 0x3F			ARC Cores extension instructions

Major Opcode	Sub Opcode1	Sub Opcode2	Sub Opcode3	Instruction Usage
[31:27]	[21:16] i-field	[5:0] a-field	[14:12]:[26:24] b-field	
"	0x2F	0x00-0x3E		ARC Cores single operand extension instructions
"	"	0x3F	0x00-0x3F	ARC Cores zero operand extension instructions
0x06	0x00-0x2E			ARC Cores extension instructions
"	0x30-0x3F			ARC Cores extension instructions
"	0x2F	0x00-0x3E		ARC Cores single operand extension instructions
"	"	0x3F	0x00-0x3F	ARC Cores zero operand extension instructions
0x07	0x00-0x2E			User extension instructions
"	0x30-0x3F			User extension instructions
"	0x2F	0x00-0x3E		User single operand extension instructions
"	"	0x3F	0x00-0x3F	User zero operand extension instructions

Table 6.14 Summary of Extension Instruction Encoding

Dual Operand Extension Instructions, 0x05, [0x00-0x2E and 0x30-0x3F]

Using major opcode 0x05, the syntax op<.f> a,b,c is encoded as shown below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	B[2:0]			0	0	I[5:0]						F	B[5:3]			C[5:0]						A[5:0]					

Figure 6.1 Extension ALU Operation, register-register

The syntax of op<.f> a,b,u6 is encoded as shown below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	B[2:0]			0	1	I[5:0]						F	B[5:3]			U[5:0]						A[5:0]					

Figure 6.2 Extension ALU Operation, register with unsigned 6-bit immediate

The syntax of op<.f> b,b,s12 is encoded as shown in below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	B[2:0]			1	0	I[5:0]						F	B[5:3]			S[5:0]						S[11:6]					

Figure 6.3 Extension ALU Opration, register with signed 12-bit immediate

The syntax of op<.cc><.f> b,b,c is encoded as shown in below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	B[2:0]			1	1	I[5:0]						F	B[5:3]			C[5:0]						0	Q[4:0]				

Figure 6.4 Extension ALU Operation, conditional register

The syntax of op<.cc><.f> b,b,u6 is encoded as shown below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	B[2:0]			1	1	I[5:0]						F	B[5:3]			U[5:0]						1	Q[4:0]				

Figure 6.5 Extension ALU Operation, cc register with unsigned 6-bit immediate

The syntax follows the same structure as the arithmetic and logical operations.

Sub-opcode I field (6 bits)	Instruction	Operation	Description
0x00	ASL	a ← b asl c	Multiple arithmetic shift left
0x01	LSR	a ← b lsr c	Multiple logicatl shift right
0x02	ASR	a ← b asr c	Multiple arithmetic shift right
0x03	ROR	a ← b ror c	Multiple rotate right
0x04	MUL64	mulres ← b * c	32 X 32 signed multiply
0x05	MULU64	mulres ← b * c	32 X 32 unsigned multiply
0x06			*Reserved*
...			*Reserved*
0x2F	SOPs	*A field is sub-opcode2*	See Single operand SOP table
...			*Reserved*
0x3F			*Reserved*

Table 6.15 Extension ALU Instructions

Single operand extension instructions, 0x05, [0x2F, 0x00 - 0x3F]

The sub-opcode 2 (destination 'a' field) is reserved for defining single source operand instructions when sub-opcode 1 of 0x2F is used.

Sub-opcode2 A field (6 bits)	Instruction	Operation	Description
0x00	SWAP	b ← swap(c)	Swap words
0x01	NORM	b ← norm(c)	Normalize

Sub-opcode2 A field (6 bits)	Instruction	Operation	Description
0x02			*Reserved*
...			*Reserved*
0x07			*Reserved*
0x08	NORMW	b ← norm(c)	Normalize word
0x09			*Reserved*
...			*Reserved*
0x3F	ZOPs	*B field is sub-opcode3*	See Zero operand (ZOP) table

Table 6.16 Extension Single Operand Instructions

Single operand instruction syntax is:

op<.f>	b,c
op<.f>	b,u6
op<.f>	b,limm
op<.f>	0,c
op<.f>	0,u6
op<.f>	0,limm

Zero operand extension instructions, 0x05, [0x2F, 0x3F, 0x00 - 0x3F]

The sub-opcode 3 (source operand b field) is reserved for defining zero operand instructions when sub-opcode 2 of 0x3F is used.

Sub-opcode3 **B field** **(6 bits)**	**Instruction**	**Operation**	**Description**
0x00			*Reserved*
0x01			*Reserved*
0x02			*Reserved*
...			*Reserved*
0x3F			*Reserved*

Table 6.17 Extension Zero Operand Instructions

Zero operand instruction syntax is:

```
op<.f>      c
op<.f>      u6
op<.f>      limm
```

User extension instructions

64 user extension slots are available in "op a,b,c" format, when using major opcode 0x07. See Table 6.14.

Chapter 7 — 16-bit Instruction Formats Reference

This chapter shows the available encoding formats for the 16-bit instructions. Some encodings define instructions that are also defined in other encoding formats. Chapter 5 — *Instruction Set Summary* lists and notes the redundant formats. The processor implements all redundant encoding formats. A listing of syntax and encoding that excludes the redundant formats is contained in Chapter 9 — *Instruction Set Details*.

A complete list of the major opcodes is shown in Table 6.1.

Load /Add Register-Register, 0x0C, [0x00 - 0x03]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	b[2:0]			c[2:0]			i[1:0]		a[2:0]		

Syntax:

LD_S	a, [b, c]
LDB_S	a, [b, c]
LDW_S	a, [b, c]
ADD_S	a, b, c

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x00	LD_S	a ← mem[b + c].l	Load long word (reg.+reg.)
0x01	LDB_S	a ← mem[b + c].b	Load unsigned byte (reg.+reg.)

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x02	LDW_S	a ← mem[b + c].w	Load unsigned word (reg.+reg.)
0x03	ADD_S	a ← b + c	Add

Table 7.1 16-Bit, LD / ADD Register-Register

Add/Sub/Shift Register-Immediate, 0x0D, [0x00 - 0x03]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	b[2:0]			c[2:0]			i[1:0]		u[2:0]		

Syntax:

ADD_S	c, b, u3
SUB_S	c, b, u3
ASL_S	c, b, u3
ASR_S	c, b, u3

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x00	ADD_S	c ← b + u3	Add
0x01	SUB_S	c ← b + u3	Subtract
0x02	ASL_S	c ← b asl u3	Multiple arithmetic shift left
0x03	ASR_S	c ← b asr u3	Multiple arithmetic shift right

Table 7.2 16-Bit, ADD/SUB Register-Immediate

Mov/Cmp/Add with High Register, 0x0E, [0x00 - 0x03]

15	14	13	12	11	10 9 8	7 6 5	4 3	2 1 0
0	1	1	1	0	b[2:0]	h[2:0]	i[1:0]	h[5:3]

Syntax:

ADD_S	b, b, h	
MOV_S	b, h	
CMP_S	b, h	
MOV_S	h, b	
MOV_S	0, b	*(h=limm)*

Sub-opcode **i field (2 bits)**	**Instruction**	**Operation**	**Description**
0x00	ADD_S	b ← b + h	Add
0x01	MOV_S	b ← h	Move
0x02	CMP_S	b - h	Compare
0x03	MOV_S	h ← b	Move

Table 7.3 16-Bit MOV/CMP/ADD with High Register

Long immediate with Mov/Cmp/Add

The instruction that contains the long immediate data indicator (register r62) as a high register, is followed by a long-word that is the long immediate (limm) data itself.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Limm[31:0]

Syntax:

limm

General Register Format Instructions, 0x0F, [0x00 - 0x1F]

15	14	13	12	11	10 9 8	7 6 5	4 3 2 1 0
0	1	1	1	1	b[2:0]	c[2:0]	i[4:0]

Syntax:

SUB_S	b, b, c
AND_S	b, b, c
OR_S	b, b, c
BIC_S	b, b, c
XOR_S	b, b, c
TST_S	b, c
SEXB_S	b, c
SEXW_S	b, c
EXTB_S	b, c
EXTW_S	b, c
ABS_S	b, c
NOT_S	b, c
NEG_S	b, c
ADD1_S	b, b, c
ADD2_S	b, b, c
ADD3_S	b, b, c
ASL_S	b, c
ASL_S	b, b, c

ASR_S	b, c	
ASR_S	b, b, c	
LSR_S	b, c	
LSR_S	b, b, c	
BRK_S		
MUL64_S	b, c	*(extension option)*

Sub-opcode **i field (5 bits)**	**Instruction**	**Operation**	**Description**
0x00	SOPs	*c field is sub-opcode2*	See Single ops table
0x01			*Reserved*
0x02	SUB_S	b ← b - c	Subtract
0x03			*Reserved*
0x04	AND_S	b ← b and c	Logical bitwise AND
0x05	OR_S	b ← b or c	Logical bitwise OR
0x06	BIC_S	b ← b and not c	Logical bitwise AND with invert
0x07	XOR_S	b ← b exclusive-or c	Logical bitwise exclusive-OR
0x08			*Reserved*
0x09			*Reserved*
0x0A			*Reserved*
0x0B	TST_S	b and c	Test
0x0C	MUL64_S	mulres ← b * c	32 X 32 Multiply
0x0D	SEXB_S	b ← sexb(c)	Sign extend byte

Sub-opcode **i field (5 bits)**	**Instruction**	**Operation**	**Description**
0x0E	SEXW_S	b ← sexw(c)	Sign extend word
0x0F	EXTB_S	b ← extb(c)	Zero extend byte
0x10	EXTW_S	b ← extw(c)	Zero extend word
0x11	ABS_S	b ← abs(c)	Absolute
0x12	NOT_S	b ← not(c)	Logical NOT
0x13	NEG_S	b ←neg(c)	Negate
0x14	ADD1_S	b ← b + (c << 1)	Add with left shift by 1
0x15	ADD2_S	b ← b + (c << 2)	Add with left shift by 2
0x16	ADD3_S	b ← b + (c << 3)	Add with left shift by 3
0x17			
0x18	ASL_S	b ← b asl c	Multiple arithmetic shift left
0x19	LSR_S	b ← b lsr c	Multiple logical shift right
0x1A	ASR_S	b ← b asr c	Multiple arithmetic shift right
0x1B	ASL_S	b ← c + c	Arithmetic shift left by one
0x1C	ASR_S	b ← c asr 1	Arithmetic shift right by one
0x1D	LSR_S	b ← c lsr 1	Logical shift right by one
0x1E			*Reserved*
0x1F	BRK_S	Break	Break (Encoding is 0x7FFF)

Table 7.4 16-Bit General Operations

Jumps and special format instructions, 0x0F, [0x00, 0x00 - 0x07]

Syntax:

J_S<.d>	[b]
JL_S<.d>	[b]
SUB_S.ne	b,b,b

Sub-opcode2 **c field (3 bits)**	**Instruction**	**Operation**	**Description**
0x00	J_S	pc ← b	Jump
0x01	J_S.D	pc ← b	Jump delayed
0x02	JL_S	blink ← pc; pc ← b	Jump and link
0x03	JL_S.D	blink ← pc; pc ← b	Jump and link delayed
0x04			*Reserved*
0x05			*Reserved*
0x06	SUB_S.NE	if (flags.Z==0) then b ← b - b	If Z flag is 0, clear register
0x07	ZOPs	*b field is* *sub-opcode3*	See Zero operand (ZOP) table

Table 7.5 16-Bit Special Format Instructions

Zero operand instructions, 0x0F, [0x00, 0x07, 0x00 - 0x07]

Syntax:

NOP_S	
J_S<.d>	[blink]
JEQ_S	[blink]
JNE_S	[blink]

Sub-opcode3 b field (3 bits)	Instruction	Operation	Description
0x00	NOP_S	nop	No operation
0x01			*Reserved*
0x02			*Reserved*
0x03			*Reserved*
0x04	JEQ_S [blink]	pc ← blink	Jump using blink register if equal
0x05	JNE_S [blink]	pc ← blink	Jump using blink register if not equal
0x06	J_S [blink]	pc ← blink	Jump using blink register
0x07	J_S.D [blink]	pc ← blink	Jump using blink register delayed

Table 7.6 16-Bit Single and Zero Operand Instructions

Load/Store with Offset, 0x10 - 0x16

The offset u[4:0] is data size aligned. Syntactically u7 should be multiples of 4, and u6 should be multiples of 2.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					b[2:0]			c[2:0]			u[4:0]				

Syntax:

LD_S	c, [b, u7]	*(u7 must be 32-bit aligned)*
LDB_S	c, [b, u5]	
LDW_S	c, [b, u6]	*(u6 must be 16-bit aligned)*
LDW_S.X	c, [b, u6]	*(u6 must be 16-bit aligned)*
ST_S	c, [b, u7]	*(u7 must be32-bit aligned)*
STB_S	c, [b, u5]	
STW_S	c, [b, u6]	*(u6 must be 16-bit aligned)*

Major opcode **I field (5 bits)**	**Instruction**	**Operation**	**Description**
0x10	LD_S	c ← mem[b + u7].l	Load long word
0x11	LDB_S	c ← mem[b + u5].b	Load unsigned byte
0x12	LDW_S	c ← mem[b + u6].w	Load unsigned word
0x13	LDW_S.X	c ← mem[b + u6].wx	Load signed word
0x14	ST_S	mem[b + u7].l ← c	Store long word
0x15	STB_S	mem[b + u5].b ← c	Store unsigned byte
0x16	STW_S	mem[b + u6].w ← c	Store unsigned word

Table 7.7 16-Bit Load and Store with Offset

Shift/Subtract/Bit Immediate, 0x17, [0x00 - 0x07]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	b[2:0]			i[2:0]			u[4:0]				

Syntax:

SUB_S	b, b, u5
BSET_S	b, b, u5
BCLR_S	b, b, u5
BMSK_S	b, b, u5
BTST_S	b, u5
ASL_S	b, b, u5
LSR_S	b, b, u5
ASR_S	b, b, u5

Sub-opcode2 i field (3 bits)	Instruction	Operation	Description
0x00	ASL_S	b ← b asl u5	Multiple arithmetic shift left
0x01	LSR_S	b ← b lsr u5	Multiple logical shift left
0x02	ASR_S	b ← b asr u5	Multiple arithmetic shift right
0x03	SUB_S	b ← b - u5	Subtract
0x04	BSET_S	b ← b or 1<<u5	Bit set
0x05	BCLR_S	b ← b and not 1<<u5	Bit clear
0x06	BMSK_S	b ← b and ((1<<(u5+1))-1)	Bit mask
0x07	BTST_S	b and 1<<u5	Bit test

Table 7.8 16-Bit Shift/SUB/Bit Immediate

Stack Pointer Based Instructions, 0x18, [0x00 - 0x07]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	b[2:0]			i[2:0]			u[4:0]				

Syntax:

LD_S	b, [SP, u7]	*(u7 offset is 32-bit aligned)*
LDB_S	b, [SP, u7]	*(u7 offset is 32-bit aligned)*
ST_S	b, [SP, u7]	*(u7 offset is 32-bit aligned)*
STB_S	b, [SP, u7]	*(u7 offset is 32-bit aligned)*
ADD_S	b, SP, u7	*(u7 offset is 32-bit aligned)*
ADD_S	SP, SP, u7	*(u7 offset is 32-bit aligned)*
SUB_S	SP, SP, u7	*(u7 offset is 32-bit aligned)*
POP_S	b	
POP_S	BLINK	
PUSH_S	b	
PUSH_S	BLINK	

Sub-opcode **i field (3 bits)**	**Instruction**	**Operation**	**Description**
0x00	LD_S	b ← mem[SP + u7].l	Load long word sp-rel.
0x01	LDB_S	b ← mem[SP + u7].b	Load unsigned byte sp-rel.
0x02	ST_S	mem[SP + u7].l ← b	Store long word sp-rel.
0x03	STB_S	mem[SP + u7].b ← b	Store unsigned byte sp-rel.

Sub-opcode i field (3 bits)	Instruction	Operation	Description
0x04	ADD_S	b ← SP + u7	Add
0x05	ADD_S /SUB_S	sp ← sp +- u7	Add (b=0) /subtract (b=1) (b=2 to b=7 are undefined)

0x06	POP_S b	b ← mem[SP].l
		SP ← SP + 4
			Pop register from stack (u=1)

0x06	POP_S blink	blink ← mem[SP].l
		SP ← SP + 4
			Pop blink from stack
			(u=17, b=reserved)

0x07	PUSH_S b	SP ← SP - 4
		mem[SP].l ← b
			Push register to stack
			(u=1)

0x07	PUSH_S blink	SP ← SP - 4
		mem[SP].l ← blink
			Push blink to stack
			(u=17, b=reserved)

Table 7.9 16-Bit Stack Pointer based Instructions

Load/Add GP-Relative, 0x19, [0x00 - 0x03]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	i[1:0]		s[8:0]								

The offset (s[8:0]) is shifted accordingly to provide the appropriate data size alignment.

Syntax:

LD_S	r0, [GP, s11]	*(32-bit aligned offset)*
LDB_S	r0, [GP, s9]	*(8-bit aligned offset)*

LDW_S	r0, [GP, s10]	*(16-bit aligned offset)*
ADD_S	r0, GP, s11	*(32-bit aligned offset)*

Sub-opcode **i field (2 bits)**	**Instruction**	**Operation**	**Description**
0x00	LD_S	r0 ← mem[GP + s11].l	Load gp-relative (32-bit aligned) to r0
0x01	LDB_S	r0 ← mem[GP + s9].b	Load unsigned byte gp-relative (8-bit aligned) to r0
0x02	LDW_S	r0 ← mem[GP +s10].w	Load unsigned word gp-relative (16-bit aligned) to r0
0x03	ADD_S	r0 ← GP + s11	Add gp-relative (32-bit aligned) to r0

Table 7.10 16-Bit GP Relative Instructions

Load PCL-Relative, 0x1A

The offset (u[7:0]) is shifted accordingly to provide the appropriate 32-bit data size alignment.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	b[2:0]			u[7:0]							

Syntax:

LD_S	b, [PCL, u10]	*(32-bit aligned offset)*

Move Immediate, 0x1B

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	b[2:0]			u[7:0]							

Syntax:

MOV_S b, u8

ADD/CMP Immediate, 0x1C, [0x00 - 0x01]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	b[2:0]			i	u[6:0]						

Syntax:

ADD_S b, b, u7

CMP_S b, u7

Sub-opcode i field (1 bit)	Instruction	Operation	Description
0x00	ADD_S	b ← b + u7	Add
0x01	CMP_S	b - u7	Compare

Table 7.11 16-Bit ADD/CMP Immediate

Branch on Compare Register with Zero, 0x1D, [0x00 - 0x01]

The target address is 16-bit aligned.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	b[2:0]			i	s[7:1]						

Syntax:

BREQ_S b, 0, s8

BRNE_S b, 0, s8

Sub-opcode i field (1 bit)	Instruction	Operation	Description
0x00	BREQ_S		Branch if register is zero
0x01	BRNE_S		Branch if register is non-zero

Table 7.12 16-Bit Branch on Compare

Branch Conditionally, 0x1E, [0x00 - 0x03]

The target address is 16-bit aligned.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i[1:0]		s[9:1]								

Syntax:

B_S	s10
BEQ_S	s10
BNE_S	s10

Sub-opcode i field (2 bits)	Instruction	Operation	Description
0x00	B_S		Branch always
0x01	BEQ_S		Branch if equal
0x02	BNE_S		Branch if not equal
0x03	Bcc_S		See Bcc table

Table 7.13 16-Bit Branch, Branch Conditionally

Branch conditionally with cc field, 0x1E, [0x03, 0x00 - 0x07]

The target address is 16-bit aligned.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	i[2:0]			s[6:1]					

Syntax:

BGT_S	s7
BGE_S	s7
BLT_S	s7
BLE_S	s7
BHI_S	s7
BHS_S	s7
BLO_S	s7
BLS_S	s7

Sub-opcode i field (3 bits)	Instruction	Operation	Description
0x00	BGT_S		Branch if greater than
0x01	BGE_S		Branch if greater than or equal
0x02	BLT_S		Branch if less than
0x03	BLE_S		Branch if less than or equal
0x04	BHI_S		Branch if higher than
0x05	BHS_S		Branch if higher or the same
0x06	BLO_S		Branch if lower than
0x07	BLS_S		Branch if lower or the same

Table 7.14 16-Bit Branch Conditionally

Branch and Link Unconditionally, 0x1F

The target address can only target 32-bit aligned instructions.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	s[12:2]										

Syntax:

BL_S s13

16-bit Extension Instructions, 0x08 - 0x0B

Four sets of extension instructions are available in 16-bit instruction format. When the sub opcode format matches the structure of the basecase ARCompact ISA, the sets are shown in the following table:

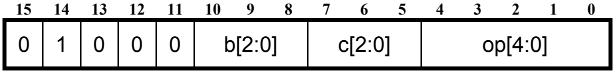
[15:11]	**[4:0]**	**[7:5]**	**[10:8]**	
Major Opcode	**i field** **Sub Opcode 1**	**c field** **Sub Opcode 2**	**b field** **Sub Opcode 3**	**Instruction Usage**
0x08	0x01-0x1F			ARC Cores extension instructions
"	0x00	0x00-0x06		ARC Cores single operand extension instructions
"	"	0x07	0x00-0x07	ARC Cores zero operand extension instructions
0x09	0x01-0x1F			ARC Cores extension instructions
"	0x00	0x00-0x06		ARC Cores single operand extension instructions
"	"	0x07	0x00-0x07	ARC Cores zero operand extension instructions

[15:11] Major Opcode	[4:0] i field Sub Opcode 1	[7:5] c field Sub Opcode 2	[10:8] b field Sub Opcode 3	Instruction Usage
0x0A	0x01-0x1F			User extension instructions
"	0x00	0x00-0x06		User single operand extension instructions
"	"	0x07	0x00-0x07	User zero operand extension instructions
0x0B	0x01-0x1F			User extension instructions
"	0x00	0x00-0x06		User single operand extension instructions
"	"	0x07	0x00-0x07	User zero operand extension instructions

Table 7.15 16-Bit Extension Instruction Encoding Summary

The following syntax examples use major opcode 0x08.

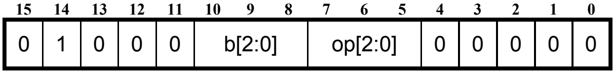
The syntax op_S b,b,c or op_S b,c is encoded as shown below.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	b[2:0]			c[2:0]			op[4:0]				

Figure 7.1 16-Bit Extension ALU Operation, register-register

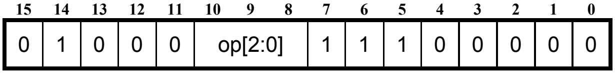
The syntax op_S b or op_S b,b is encoded as shown below.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	b[2:0]			op[2:0]			0	0	0	0	0

Figure 7.2 16-Bit Extension ALU Operation, register

The syntax op_S is encoded as shown below.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	op[2:0]			1	1	1	0	0	0	0	0

Figure 7.3 16-Bit Extension ALU Operation, no registers

Chapter 8 — Condition Codes

Introduction

The ARC 600 processor has an extensive instruction set most of which can be carried out conditionally and/or set the flags. Those instructions using short immediate data can not have a condition code test.

Branch, loop and jump instructions use the same condition codes as instructions. However, the condition code test for these jumps is carried out one stage earlier in the pipeline than other instructions. Therefore, a single cycle stall will occur if a jump is immediately preceded by an instruction that sets the flags.

Status Register

The status register contains the status flags. The status register (STATUS32), shown in Figure 8.1, contains the following status flags: zero (Z), negative (N), carry (C) and overflow (V); the interrupt mask bits (E[2:1]); and the halt bit (H).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED																				Z	N	C	V	R	R	R	R	R	E2	E1	H

Figure 8.1 Status Register 32-Bit

Status Flags Notation

In the instruction set details in the following chapters the following notation is used for status flags:

Z	= Set if result is zero
N	= Set if most significant bit of result is set
C	= Set if carry is generated
V	= Set if overflow is generated

The convention used for the effect of an operation on the status flags is:

●	= Set according to the result of the operation
	= Not affected by the operation
0	= Bit cleared after the operation
1	= Bit set after the operation

Condition Code Test

Table 8.1 shows condition names and the conditions they test.

Mnemonic	Condition	Test	Code
AL, RA	Always	1	0x00
EQ, Z	Zero	Z	0x01
NE, NZ	Non-Zero	/Z	0x02
PL, P	Positive	/N	0x03
MI, N	Negative	N	0x04
CS, C, LO	Carry set, lower than (unsigned)	C	0x05
CC, NC, HS	Carry clear, higher or same (unsigned)	/C	0x06
VS, V	Over-flow set	V	0x07
VC, NV	Over-flow clear	/V	0x08
GT	Greater than (signed)	(N and V and /Z) or (/N and /V and /Z)	0x09
GE	Greater than or equal to (signed)	(N and V) or (/N and /V)	0x0A
LT	Less than (signed)	(N and /V) or (/N and V)	0x0B
LE	Less than or equal to (signed)	Z or (N and /V) or (/N and V)	0x0C
HI	Higher than (unsigned)	/C and /Z	0x0D
LS	Lower than or same (unsigned)	C or Z	0x0E
PNZ	Positive non-zero	/N and /Z	0x0F

Table 8.1 Condition codes

NOTE PNZ does not have an inverse condition.

The remaining 16 condition codes (10-1F) are available for extension and are used to:

- provide additional tests on the internal condition flags or
- test extension status flags from external sources or
- test a combination external and internal flags

If an extension condition code is used that is not implemented, then the condition code test will always return false (i.e. the opposite of AL - always).

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

Chapter 9 — Instruction Set Details

Instruction Set Details

This chapter lists the available instruction set in alphabetic order. The syntax and encoding examples list full syntax for each instruction, but excludes the redundant encoding formats. A full list of encoding formats can be found in Chapter 5 — *Instruction Set Summary*.

Both 32-bit and 16-bit instructions are available in the ARCompact ISA and are indicated using particular suffixes on the instruction as illustrated by the following syntax:

OP	implies 32-bit instruction
OP_L	indicates of 32-bit instruction.
OP_S	indicates 16-bit instruction

If no suffix is used on the instruction then the implied instruction is 32-bit format.

The following notation is used for the operations.

Key for 32-bit instruction formats:

I	instruction major opcode
i	instruction subopcode
A	destination register
b	lower bits source/destination register
B	upper bits source/destination register
C	source/destination register
Q	condition code

u	unsigned immediate (number indicates bit field size)
s	lower bits signed immediate (number indicates bit field size)
S	upper bits signed immediate (number indicates bit field size)
T	upper bits signed immediate (branch unconditionally far)
N	delay slot mode
F	Flag Setting
R	reserved
D	.di direct data cache bypass
A	.a address writeback mode
Z	.zz data size
X	.x sign extend

Table 9.1 32-bit Addressing Modes and Conventions

Key for 16-bit instruction formats:	
_s	suffix for 16-bit instruction
i	instruction sub-opcode
a	source/destination register (r0-3,r12-15)
b	source/destination register (r0-3,r12-15)
c	source/destination register (r0-3,r12-15)
h	source/destination register high (r0-r63)
q	condition code
u	unsigned immediate (number indicates bit field size)
s	signed immediate (number indicates bit field size)

Table 9.2 16-bit Addressing Modes and Conventions

Instruction Code Layout

The instruction encoding for each instruction type and format is shown in the instructions per-page section.

All fields that correspond to an instruction word for a particular format are shown. Fields that have pre-defined values assigned to them are illustrated, and fields that are encoded by the assembler are represented as letters.

32-bit general register formats

Register-Register

```
3130292827262524232221201918171615141312111009080706050403020100
00100bbb00iiiiiiFBBBCCCCCCAAAAAA
 I[4:0]    b[2:0]   00    i[5:0]    F  B[5:3]    C[5:0]       A[5:0]
```

Register with Unsigned 6-Bit Immediate

```
3130292827262524232221201918171615141312111009080706050403020100
00100bbb01iiiiiiFBBBUUUUUUAAAAAA
 I[4:0]    b[2:0]   01    i[5:0]    F  B[5:3]    U[5:0]       A[5:0]
```

Register with Signed 12-Bit Immediate

```
3130292827262524232221201918171615141312111009080706050403020100
00100bbb10iiiiiiFBBBssssssSSSSSS
 I[4:0]    b[2:0]   10    i[5:0]    F  B[5:3]    s[5:0]       S[11:6]
```

Conditional Register with Unsigned 6-Bit Immediate

```
3130292827262524232221201918171615141312111009080706050403020100
00100bbb11iiiiiiFBBBUUUUUU1QQQQQ
  I[4:0]    b[2:0]   11    i[5:0]    F  B[5:3]    U[5:0]    1    Q[4:0]
```

Conditional Register

```
3130292827262524232221201918171615141312111009080706050403020100
00100bbb11iiiiiiFBBBCCCCCC0QQQQQ
  I[4:0]    b[2:0]   11    i[5:0]    F  B[5:3]    C[5:0]    0    Q[4:0]
```

32-bit memory instruction formats

Load Register-Register

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
00100bbbaa110ZZXDBBBCCCCCCAAAAAA
I[4:0]  b[2:0]  a[1:0]  110  Z[1:0]  X  D  B[5:3]  C[5:0]  A[5:0]
```

Long Immediate Data

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
LIMM[31:0]
```

Load Register with Offset

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
00010bbbssssssssSBBBDaaZZXAAAAAA
I[4:0]  b[2:0]  S[7:0]  S  B[5:3]  D  A[1:0]  Z[1:0]  X  A[5:0]
```

Store Register with Offset

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
00011bbbssssssssSBBBCCCCCCDAAZZR
I[4:0]  b[2:0]  S[7:0]  S  B[5:3]  C[5:0]  D  A[1:0]  Z[1:0]  R
```

32-bit branch instruction formats

Branch (16-bit aligned target address)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
00000ssssssssss0SSSSSSSSSSNQQQQQ
I[4:0]  s[10:1]  0  S[20:11]  N  Q[4:0]
```

Branch Unconditional Far (16-bit aligned target address)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
00000ssssssssss1SSSSSSSSSSNRTTTT
I[4:0]  s[10:1]  1  S[20:11]  N  R  T[24:21]
```

Branch on Compare Register-Register (16-bit aligned target address)

```
31302928272625242322212019181716151413121110090807060504030201 00
00001bbbsssssss1SBBBCCCCCCN0iiii
 I[4:0]  b[2:0]   S[7:1]   1 S8 B[5:3]  C[5:0]  N 0  i[3:0]
```

Branch on Compare/Bit Test Register-Immediate (16-bit aligned target address)

```
31302928272625242322212019181716151413121110090807060504030201 00
00001bbbsssssss1SBBBUUUUUUN1iiii
 I[4:0]  b[2:0]   S[7:1]   1 S8 B[5:3]  C[5:0]  N 1  i[3:0]
```

Branch and Link (32-bit aligned target address)

```
31302928272625242322212019181716151413121110090807060504030201 00
00001sssssssss00SSSSSSSSSSNQQQQQ
 I[4:0]     s[10:2]    00     S[20:11]     N   Q[4:0]
```

Branch and Link Unconditional Far (32-bit aligned target address)

```
31302928272625242322212019181716151413121110090807060504030201 00
00001sssssssss10SSSSSSSSSSNRTTTT
 I[4:0]     s[10:2]    10     S[20:11]     N R  T[24:21]
```

16-bit general register formats

Undefined

```
15141312111009080706050403020100
010RRRRRRRRRRRRR
 I[4:0]       R[10:0]
```

General Register Format

```
15141312111009080706050403020100
01111bbbccciiiii
 I[4:0]  b[2:0] c[2:0]  I[4:0]
```

16-bit memory register formats

Load & Add Register-Register (LD/LDB/LDW/ADD)

```
15141312111009080706050403020100
01100bbbcccciiaaa
I[4:0]  b[2:0] C[2:0] i[1:0] a[2:0]
```

Load/Store with Offset LD/LDB/LDW.X/ST/STW

```
15141312111009080706050403020100
IIIIIbbbcccuuuuu
 I[4:0]  b[2:0] c[2:0]  u[4:0]
```

Stack Based Instructions

```
15141312111009080706050403020100
11000bbbiiiuuuuu
 I[4:0]  b[2:0] i[2:0]  u[4:0]
```

Load/Add GP Relative

```
15141312111009080706050403020100
11001iisssssssss
I[4:0]  i[1:0]     S[8:0]
```

Load PCL Relative

```
15141312111009080706050403020100
11010bbbuuuuuuuu
 I[4:0]  b[2:0]     u[7:0]
```

16-bit arithmetic & logical formats

Add/Sub/Shift Register-Immediate

```
15141312111009080706050403020100
01101bbbccciiuuu
I[4:0]  b[2:0] C[2:0] i[1:0] u[2:0]
```

MOV/CMP/ADD High Register (r0-r63)

```
15141312111009080706050403020100
01110bbbhhhiiHHH
 I[4:0]    b[2:0] h[2:0] i[1:0] H[5:3]
```

Shift/Subtract/Bit Immediate

```
15141312111009080706050403020100
10111bbbiiiuuuuu
  I[4:0]    b[2:0] i[2:0]   u[4:0]
```

ADD/CMP Immediate

```
15141312111009080706050403020100
11100bbbiuuuuuuu
  I[4:0]    b[2:0] i       u[6:0]
```

MOV Immediate

```
15141312111009080706050403020100
11011bbbuuuuuuuu
  I[4:0]    b[2:0]         u[7:0]
```

16-bit branch register formats

Branch Conditionally on Register Test with Zero. EQ/NE

```
15141312111009080706050403020100
11101bbbisssssss
  I[4:0]    b[2:0] i       s[7:1]
```

Branch Conditionally AL/EQ/NE

```
15141312111009080706050403020100
11110iisssssssss
 I[4:0]    i[1:0]         s[9:1]
```

Branch Conditionally GT/GE/LT/LE/HI/HS/LO/LS

```
15141312111009080706050403020100
1111011iiissssss
  I[6:0]    i[2:0]   s[6:1]
```

Branch and Link

```
15141312111009080706050403020100
11111sssssssssss
 I[4:0]          S[12:2]
```

Alphabetic Listing

The instructions are arranged in alphabetical order. The instruction name is given at the top left and top right of the page, along with a brief instruction description, and instruction type.

The following terms are used in the description of each instruction.

Format	Instruction format
Operation	Operation of the instruction
Format Key	Key for instruction operation
Syntax	The syntax of the instruction and supported constructs
Instruction Code	Layout of the field of the instruction
Flag Affected	List of status flags that are affected
Related Instructions	Instructions that are related
Description	Full description of the instruction
Pseudo Code Example	Operation of the instruction described in pseudo code
Assembly Code Example	A simple coding example

This page is intentionally left blank.

ABS **Absolute** ABS

Arithmetic Operation

Format: inst dest, src **Operation:** dest ← ABS(src)

Format Key:

src	=	Source Operand
dest	=	Destination
ABS	=	Take Absolute Value of Source

Syntax:

With Result		**Instruction Code**	
ABS<.f>	b,c	00100bbb00101111FBBBCCCCCC001001	
ABS<.f>	b,u6	00100bbb01101111FBBBuuuuuu001001	
ABS<.f>	b,limm	00100bbb00101111FBBB111110001001	L
ABS_S	b,c	01111bbbccc10001	
Without Result			
ABS<.f>	0,c	0010011000101111F111CCCCCC001001	
ABS<.f>	0,u6	0010011001101111F111uuuuuu001001	
ABS<.f>	0,limm	0010011000101111F111111110001001	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if src = 0x8000 0000
C	●	= MSB of src
V	●	= Set if src = 0x8000 0000

Key:

L	= Limm Data

Related Instructions:

SEXB	EXTB
SEXW	EXTW
NEG	

Description:

Take the absolute value that is found in the source operand (src) and place the result into the destination register (dest). The carry flag reflects the state of the most significant bit found in the source register.

Pseudo Code Example:

```
alu = 0 - src                                     /* ABS */
if src[31]==1 then
 dest = alu
else
 dest = src
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = if src==0x8000_0000 then 1 else 0
 C_flag = src[31]
 V_flag = if src==0x8000_0000 then 1 else 0
```

Assembly Code Example:

```
ABS r1,r2              ; Take the absolute value of
                       ; r2 and write result into r1
```

ADC Addition with Carry ADC

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 + src2 + carry

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition code

Syntax:

With Result		**Instruction Code**	
ADC<.f>	a,b,c	`00100bbb00000001FBBBCCCCCCAAAAAA`	
ADC<.f>	a,b,u6	`00100bbb01000001FBBBuuuuuuAAAAAA`	
ADC<.f>	b,b,s12	`00100bbb10000001FBBBssssssSSSSSS`	
ADC<.cc><.f>	b,b,c	`00100bbb11000001FBBBCCCCCC0QQQQQ`	
ADC<.cc><.f>	b,b,u6	`00100bbb11000001FBBBuuuuuu1QQQQQ`	
ADC<.f>	a,limm,c	`0010011000000001F111CCCCCCAAAAAA`	L
ADC<.f>	a,b,limm	`00100bbb00000001FBBB111110AAAAAA`	L
ADC<.cc><.f>	b,b,limm	`00100bbb11000001FBBB1111100QQQQQ`	L
Without Result			
ADC<.f>	0,b,c	`00100bbb00000001FBBBCCCCCC111110`	
ADC<.f>	0,b,u6	`00100bbb01000001FBBBuuuuuu111110`	
ADC<.f>	0,b,limm	`00100bbb00000001FBBB111110111110`	L
ADC<.cc><.f>	0,limm,c	`0010011011000001F111CCCCCC0QQQQQ`	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if overflow is generated

Key:

L = Limm Data

Related Instructions:

ADD	ADD2
ADD1	ADD3

Description:

Add source operand 1 (src1) and source operand 2 (src2) and carry, and place the result in the destination register.

Pseudo Code Example:

```
if cc==true then                                    /* ADC */
 dest = src1 + src2 + C_flag
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example:

```
ADC r1,r2,r3           ; Add r2 to r3 with carry and
                       ; write result into r1
```

ADD — Addition — ADD

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 + src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition code

Syntax:

With Result		**Instruction Code**	
ADD<.f>	a,b,c	00100bbb00000000FBBBCCCCCCAAAAAA	
ADD<.f>	a,b,u6	00100bbb01000000FBBBuuuuuuAAAAAA	
ADD<.f>	b,b,s12	00100bbb10000000FBBBssssssSSSSSS	
ADD<.cc><.f>	b,b,c	00100bbb11000000FBBBCCCCCC0QQQQQ	
ADD<.cc><.f>	b,b,u6	00100bbb11000000FBBBuuuuuu1QQQQQ	
ADD<.f>	a,limm,c	0010011000000000F111CCCCCCAAAAAA	L
ADD<.f>	a,b,limm	00100bbb00000000FBBB111110AAAAAA	L
ADD<.cc><.f>	b,b,limm	00100bbb11000000FBBB1111100QQQQQ	L
ADD_S	a,b,c	01100bbbccc11aaa	
ADD_S	c,b,u3	01101bbbccc00uuu	
ADD_S	b,b,h	01110bbbhhh00HHH	
ADD_S	b,b,limm	01110bbb11000111	L
ADD_S	b,sp,u7	11000bbb100uuuuu	
ADD_S	sp,sp,u7	11000000101uuuuu	
ADD_S	r0,gp,s11	1100111sssssssss	
ADD_S	b,b,u7	11100bbb0uuuuuuu	
Without Result			
ADD<.f>	0,b,c	00100bbb00000000FBBBCCCCCC111110	
ADD<.f>	0,b,u6	00100bbb01000000FBBBuuuuuu111110	
ADD<.f>	0,b,limm	00100bbb00000000FBBB111110111110	L
ADD<.cc><.f>	0,limm,c	0010011011000000F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if overflow is generated

Key:

L = Limm Data

Related Instructions:

ADD	ADD2
ADD1	ADD3

Description:

Add source operand 1 (src1) to source operand 2 (src2) and place the result in the destination register.

NOTE For the 16-bit encoded instructions that work on the stack pointer (SP) or global pointer (GP) the offset is aligned to 32-bit. For example ADD_S sp, sp. u7 only needs to encode the top 5 bits since the bottom 2 bits of u7 are always zero because of the 32-bit data alignment.

Pseudo Code Example:

```
if cc==true then                                        /* ADD */
 dest = src1 + src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example:

```
ADD r1,r2,r3            ; Add contents of r2 with r3
                        ; and write result into r1
```

ADD1 **Addition with Scaled Source** ADD1

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 + (src2 << 1)

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition code

Syntax:

With Result		**Instruction Code**	
ADD1<.f>	a,b,c	00100bbb00010100FBBBCCCCCCAAAAAA	
ADD1<.f>	a,b,u6	00100bbb01010100FBBBuuuuuuAAAAAA	
ADD1<.f>	b,b,s12	00100bbb10010100FBBBssssssSSSSSS	
ADD1<.cc><.f>	b,b,c	00100bbb11010100FBBBCCCCCC0QQQQQ	
ADD1<.cc><.f>	b,b,u6	00100bbb11010100FBBBuuuuuu1QQQQQ	
ADD1<.f>	a,limm,c	0010011000010100F111CCCCCCAAAAAA	L
ADD1<.f>	a,b,limm	00100bbb00010100FBBB111110AAAAAA	L
ADD1<.cc><.f>	b,b,limm	00100bbb11010100FBBB1111100QQQQQ	L
ADD1_S	b,b,c	01111bbbccc10100	
Without Result			
ADD1<.f>	0,b,c	00100bbb00010100FBBBCCCCCC111110	
ADD1<.f>	0,b,u6	00100bbb01010100FBBBuuuuuu111110	
ADD1<.f>	0,b,limm	00100bbb00010100FBBB111110111110	L
ADD1<.cc><.f>	0,limm,c	0010011011010100F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if overflow is generated from the ADD part of the instruction

Key:

L	= Limm Data

Related Instructions:

ADD	ADD2
ADDC	ADD3

Description:

Add source operand 1 (src1) to a scaled version of source operand 2 (src2) (src2 left shifted by 1). Place the result in the destination register.

Pseudo Code Example:

```
if cc==true then                                         /* ADD1 */
 shiftedsrc2 = src2 << 1
 dest = src1 + shiftedsrc2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = (src1[31] AND shiftedsrc2[31] and NOT
dest[31] ) OR ( NOT src1[31] AND NOT
shiftedsrc2[31] and dest[31])
```

Assembly Code Example:

```
ADD1 r1,r2,r3          ; Add contents of r3 shifted
                       ; left one bit to r2
                       ; and write result into r1
```

ADD2 — Addition with Scaled Source — ADD2

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 + (src2 << 2)

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition code

Syntax:

With Result		**Instruction Code**	
ADD2<.f>	a,b,c	00100bbb00010101FBBBCCCCCCAAAAAA	
ADD2<.f>	a,b,u6	00100bbb01010101FBBBuuuuuuAAAAAA	
ADD2<.f>	b,b,s12	00100bbb10010101FBBBssssssSSSSSS	
ADD2<.cc><.f>	b,b,c	00100bbb11010101FBBBCCCCCC0QQQQQ	
ADD2<.cc><.f>	b,b,u6	00100bbb11010101FBBBuuuuuu1QQQQQ	
ADD2<.f>	a,limm,c	0010011000010101F111CCCCCCAAAAAA	L
ADD2<.f>	a,b,limm	00100bbb00010101FBBB111110AAAAAA	L
ADD2<.cc><.f>	b,b,limm	00100bbb11010101FBBB1111100QQQQQ	L
ADD2_S	b,b,c	01111bbbccc10101	
Without Result			
ADD2<.f>	0,b,c	00100bbb00010101FBBBCCCCCC111110	
ADD2<.f>	0,b,u6	00100bbb01010101FBBBuuuuuu111110	
ADD2<.f>	0,b,limm	00100bbb00010101FBBB111110111110	L
ADD2<.cc><.f>	0,limm,c	0010011011010101F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if overflow is generated from the ADD part of the instruction

Key:

L = Limm Data

Related Instructions:

ADD	ADD1
ADDC	ADD3

Description:

Add source operand 1 (src1) to a scaled version of source operand 2 (src2) (src2 left shifted by 2). Place the result in the destination register.

Pseudo Code Example:

```
if cc==true then                                      /* ADD2 */
 shiftedsrc2 = (src2 << 2)
 dest = src1 + shiftedsrc2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = (src1[31] AND shiftedsrc2[31] and NOT
dest[31] ) OR ( NOT src1[31] AND NOT
shiftedsrc2[31] and dest[31])
```

Assembly Code Example:

```
ADD2 r1,r2,r3          ; Add contents of r3 shifted
                       ; left two bits to r2
                       ; and write result into r1
```

ADD3 — Addition with Scaled Source — ADD3

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 + (src2 << 3)

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition code

Syntax:

With Result		**Instruction Code**	
ADD3<.f>	a,b,c	00100bbb00010110FBBBCCCCCCAAAAAA	
ADD3<.f>	a,b,u6	00100bbb01010110FBBBuuuuuuAAAAAA	
ADD3<.f>	b,b,s12	00100bbb10010110FBBBssssssSSSSSS	
ADD3<.cc><.f>	b,b,c	00100bbb11010110FBBBCCCCCC0QQQQQ	
ADD3<.cc><.f>	b,b,u6	00100bbb11010110FBBBuuuuuu1QQQQQ	
ADD3<.f>	a,limm,c	0010011000010110F111CCCCCCAAAAAA	L
ADD3<.f>	a,b,limm	00100bbb00010110FBBB111110AAAAAA	L
ADD3<.cc><.f>	b,b,limm	00100bbb11010110FBBB1111100QQQQQ	L
ADD3_S	b,b,c	01111bbbccc10110	
Without Result			
ADD3<.f>	0,b,c	00100bbb00010110FBBBCCCCCC111110	
ADD3<.f>	0,b,u6	00100bbb01010110FBBBuuuuuu111110	
ADD3<.f>	0,b,limm	00100bbb00010110FBBB111110111110	L
ADD3<.cc><.f>	0,limm,c	0010011011010110F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if overflow is generated from the ADD part of the instruction

Key:

L = Limm Data

Related Instructions:

ADD	ADD1
ADDC	ADD2

Description:

Add source operand 1 (src1) to a scaled version of source operand 2 (src2) (src2 left shifted by 3). Place the result in the destination register.

Pseudo Code Example:

```
if cc==true then                                           /* ADD3 */
 shiftedsrc2 = src2 << 3
 dest = src1 + shiftedsrc2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = (src1[31] AND shiftedsrc2[31] and NOT
dest[31] ) OR ( NOT src1[31] AND NOT
shiftedsrc2[31] and dest[31])
```

Assembly Code Example:

```
ADD3 r1,r2,r3          ; Add contents of r3 shifted
                       ; left three bits to r2
                       ; and write result into r1
```

AND **Bitwise AND Operation** **Logical Operation** AND

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 AND src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition code

Syntax:

With Result		**Instruction Code**	
AND<.f>	a,b,c	`00100bbb00000100FBBBCCCCCCAAAAAA`	
AND<.f>	a,b,u6	`00100bbb01000100FBBBuuuuuuAAAAAA`	
AND<.f>	b,b,s12	`00100bbb10000100FBBBssssssSSSSSS`	
AND<.cc><.f>	b,b,c	`00100bbb11000100FBBBCCCCCC0QQQQQ`	
AND<.cc><.f>	b,b,u6	`00100bbb11000100FBBBuuuuuu1QQQQQ`	
AND<.f>	a,limm,c	`0010011000000100F111CCCCCCAAAAAA`	L
AND<.f>	a,b,limm	`00100bbb00000100FBBB111110AAAAAA`	L
AND<.cc><.f>	b,b,limm	`00100bbb11000100FBBB1111100QQQQQ`	L
AND_S	b,b,c	`01111bbbccc00100`	
Without Result			
AND<.f>	0,b,c	`00100bbb00000100FBBBCCCCCC111110`	
AND<.f>	0,b,u6	`00100bbb01000100FBBBuuuuuu111110`	
AND<.f>	0,b,limm	`00100bbb00000100FBBB111110111110`	L
AND<.cc><.f>	0,limm,c	`0010011011000100F111CCCCCC0QQQQQ`	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

OR	XOR
BIC	

Description:

Logical bitwise AND of source operand 1 (src1) with source operand 2 (src2) with the result written to the destination register.

Pseudo Code Example:

```
if cc==true then                                   /* AND */
 dest = src1 AND src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

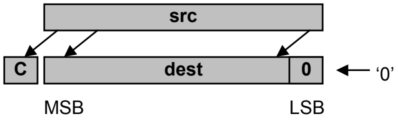
Assembly Code Example:

```
AND r1,r2,r3           ; AND contents of r2 with r3
                       ; and write result into r1
```

ASL — Arithmetic Shift Left — ASL

Logical Operation

Format: inst dest, src **Operation:** dest ← src + src



Format Key:

dest	=	Destination Register
src	=	Source Operand

Syntax:

With Result		**Instruction Code**	
ASL<.f>	b,c	00100bbb00101111FBBBCCCCCC000000	
ASL<.f>	b,u6	00100bbb01101111FBBBuuuuuu000000	
ASL<.f>	b,limm	00100bbb00101111FBBB111110000000	L
ASL_S	b,c	01111bbbccc11011	
Without Result			
ASL<.f>	0,c	00100110 00101111F111CCCCCC000000	
ASL<.f>	0,u6	00100110 01101111F111uuuuuu000000	
ASL<.f>	0,limm	00100110 00101111F111111110000000	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if the sign bit changes after a shift

Key:

L = Limm Data

Related Instructions:

ASR	LSR
ROR	RRC

Description:

Arithmetically left shift the source operand (src) by one and place the result into the destination register (dest). An ASL operation is effectively accomplished by adding the source operand upon itself (src + src), with the result being written into the destination register. All condition code flags are updated as a result of performing the ADD.

Pseudo Code Example:

```
dest = src + src                                    /* ASL */
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 C_flag = Carry()
 V_flag = Overflow()
```

Assembly Code Example:

```
ASL r1,r2              ; Arithmetic shift left
                       ; contents of r2 by one bit
                       ; and write result into r1
```

ASL multiple

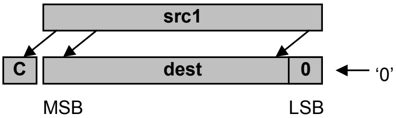
Multiple Arithmetic Shift Left

Logical Operation

ASL multiple

Format: inst dest, src1, src2

Operation: if (cc=true) then dest ← arithmetic shift left of src1 by src2



Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
ASL<.f>	a,b,c	00101bbb00000000FBBBCCCCCCAAAAAA	
ASL<.f>	a,b,u6	00101bbb01000000FBBBuuuuuuAAAAAA	
ASL<.f>	b,b,s12	00101bbb10000000FBBBssssssSSSSSS	
ASL<.cc><.f>	b,b,c	00101bbb11000000FBBBCCCCCC0QQQQQ	
ASL<.cc><.f>	b,b,u6	00101bbb11000000FBBBuuuuuu1QQQQQ	
ASL<.f>	a,limm,c	0010111000000000F111CCCCCCAAAAAA	L
ASL<.f>	a,b,limm	00101bbb00000000FBBB111110AAAAAA	L
ASL<.cc><.f>	b,b,limm	00101bbb11000000FBBB1111100QQQQQ	L
ASL_S	c,b,u3	01101bbbccc10uuu	
ASL_S	b,b,c	01111bbbccc11000	
ASL_S	b,b,u5	10111bbb000uuuuu	
Without Result			
ASL<.f>	0,b,c	00101bbb00000000FBBBCCCCCC111110	
ASL<.f>	0,b,u6	00101bbb01000000FBBBuuuuuu111110	
ASL<.cc><.f>	0,limm,c	0010111011000000F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	•	= Set if result is zero
N	•	= Set if most significant bit of result is set
C	•	= Set if carry is generated
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

ASR	LSR
ROR	RRC

Description:

Arithmetically, shift left src1 by src2 places and place the result in the destination register. Only the bottom 5 bits of src2 are used as the shift value.

Pseudo Code Example:

```
if cc==true then                                     /* ASL */
 dest = src1 << (src2 & 31)                          /* Multiple */
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = if src2==0 then 0 else src1[32-src2]
```

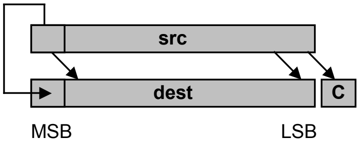
Assembly Code Example:

```
ASL r1,r2,r3           ; Arithmetic shift left
                       ; contents of r2 by r3 bits
                       ; and write result into r1
```

ASR **Arithmetic Shift Right** ASR

Logical Operation

Format: inst dest, src **Operation:** dest ← ASR by 1 (src)



Format Key:

dest = Destination Register
src = Source Operand

Syntax:

With Result		**Instruction Code**	
ASR<.f>	b,c	00100bbb00101111FBBBCCCCCC000001	
ASR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000001	
ASR<.f>	b,limm	00100bbb00101111FBBB111110000001	L
ASR_S	b,c	01111bbbccc11100	
Without Result			
ASR<.f>	0,c	0010011000101111F111CCCCCC000001	
ASR<.f>	0,u6	0010011001101111F111uuuuuu000001	
ASR<.f>	0,limm	0010011000101111F111111110000001	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

ASL
ROR
LSR
RRC

Description:

Arithmetically right shift the source operand (src) by one and place the result into the destination register (dest). The sign of the source operand is retained in the destination register.

Pseudo Code Example:

```
dest = src >> 1                                    /* ASR */
if src[31]==1 then dest[31] = 1
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 C_flag = src[0]
```

Assembly Code Example:

```
ASR r1,r2                ; Arithmetic shift right
                         ; contents of r2 by one bit
                         ; and write result into r1
```

ASR multiple

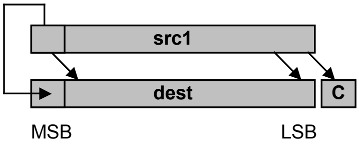
Multiple Arithmetic Shift Right

Logical Operation

ASR multiple

Format: inst dest, src1, src2

Operation: if (cc=true) then dest ← arithmetic shift right of src1 by src2



Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
ASR<.f>	a,b,c	00101bbb00000010FBBBCCCCCCAAAAAA	
ASR<.f>	a,b,u6	00101bbb01000010FBBBuuuuuuAAAAAA	
ASR<.f>	b,b,s12	00101bbb10000010FBBBssssssSSSSSS	
ASR<.cc><.f>	b,b,c	00101bbb11000010FBBBCCCCCC0QQQQQ	
ASR<.cc><.f>	b,b,u6	00101bbb11000010FBBBuuuuuu1QQQQQ	
ASR<.f>	a,limm,c	0010111000000010F111CCCCCCAAAAAA	L
ASR<.f>	a,b,limm	00101bbb00000010FBBB111110AAAAAA	L
ASR<.cc><.f>	b,b,limm	00101bbb11000010FBBB1111100QQQQQ	L
ASR_S	c,b,u3	01101bbbccc11uuu	
ASR_S	b,b,c	01111bbbccc11010	
ASR_S	b,b,u5	10111bbb010uuuuu	
Without Result			
ASR<.f>	0,b,c	00101bbb00000010FBBBCCCCCC111110	
ASR<.f>	0,b,u6	00101bbb01000010FBBBuuuuuu111110	
ASR<.cc><.f>	0,limm,c	0010111011000010F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V		= Unchanged

Key:

L	= Limm Data

Related Instructions:

ASL	LSR
ROR	RRC

Description:

Arithmetically, shift right src1 by src2 places and place the result in the destination register. Only the bottom 5 bits of src2 are used as the shift value.

Pseudo Code Example:

```
if cc==true then                                    /* ASR */
 dest = ((signed)src1) >> (src2 & 31)               /* Multiple */
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = if src2==0 then 0 else src1[src2-1]
```

Assembly Code Example:

```
ASR r1,r2,r3           ; Arithmetic shift right
                       ; contents of r2 by r3 bits
                       ; and write result into r1
```

BBIT0 — Branch on Bit Test Clear — BBIT0

Branch on Register Comparison

Format: inst src1, src2, rd

Operation: if (src1 AND 2^{src2}) = 0 then cPC ← cPCL+rd

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
rd	=	Relative Displacement
cPC	=	Current Program Counter
cPCL	=	Current Program Counter (Address from the 1st byte of the instruction, 32-bit aligned)
nPC	=	Next PC
dPC	=	Next PC + 4 (address of the 2nd following instruction)

Syntax:

		Instruction Code
BBIT0<.d>	b,c,s9	`00001bbbsssssss1SBBBCCCCCCN01110`
BBIT0<.d>	b,u6,s9	`00001bbbsssssss1SBBBuuuuuuN11110`

Delay Slot Modes <.d>:

Delay Slot Mode	N Flag	Description
ND	0	Only execute next instruction when *not* branching (*default, if no <.d> field syntax*)
D	1	Always execute next instruction

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Related Instructions:

BBIT1	BREQ
BRNE	BRLT
BRGE	BRLO
BRHS	

Description:

Test a bit within source operand 1 (src1) to see if it is clear (0). Source operand 2 (src2) explicitly specifies the bit-position that is to be tested within source operand 1 (src1). Only the bottom 5 bits of src2 are used as the bit position. If the condition is true, branch from the current PC (actually PCL) with the displacement value specified in the source operand (rd).

The branch target address can be 16-bit aligned. Since the execution of the instruction that is in the delay slot is controlled by the delay slot mode, it should never be the target of any branch or jump instruction.

CAUTION	The BBIT0 instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D instruction.

Pseudo Code Example:

```
if (src1 & (1 << (src2 & 31)))==0 then          /* BBIT0 */
 if N=1 then
  DelaySlot(nPC)
 KillDelaySlot(dPC)
 PC = cPCL + rd
else
 PC = nPC
```

Assembly Code Example:

```
BBIT0 r1,9,label         ; Branch to label if bit 9
                         ; of r1 is clear
```

BBIT1 — Branch on Bit Test Set / Branch on Register Comparison — BBIT1

Format: inst src1, src2, rd

Operation: if (src1 AND 2^{src2}) = 1 then cPC ← cPCL+rd

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
rd	=	Relative Displacement
cPC	=	Current Program Counter
cPCL	=	Current Program Counter (Address from the 1st byte of the instruction, 32-bit aligned)
nPC	=	Next PC
dPC	=	Next PC + 4 (address of the 2nd following instruction)

Syntax:

		Instruction Code
BBIT1<.d>	b,c,s9	`00001bbbsssssss1SBBBCCCCCCN01111`
BBIT1<.d>	b,u6,s9	`00001bbbsssssss1SBBBuuuuuuN11111`

Delay Slot Modes <.d>:

Delay Slot Mode	N Flag	Description
ND	0	Only execute next instruction when *not* branching (*default, if no <.d> field syntax*)
D	1	Always execute next instruction

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Related Instructions:

BBIT0	BREQ
BRNE	BRLT
BRGE	BRLO
BRHS	

Description:

Test a bit within source operand 1 (src1) to see if it is set (1). Source operand 2 (src2) explicitly specifies the bit-position that is to be tested within source operand 1 (src1). Only the bottom 5 bits of src2 are used as the bit position. If the condition is true, branch from the current PC (actually PCL) with the displacement value specified in the source operand (rd).

The branch target address can be 16-bit aligned. Since the execution of the instruction that is in the delay slot is controlled by the delay slot mode, it should never be the target of any branch or jump instruction.

CAUTION	The BBIT1 instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D instruction.

Pseudo Code Example:

```
if (src1 & (1 << (src2 & 31)))!=0 then          /* BBIT1 */
 if N=1 then
  DelaySlot(nPC)
 KillDelaySlot(dPC)
 PC = cPCL + rd
else
 PC = nPC
```

Assembly Code Example:

```
BBIT1 r1,9,label         ; Branch to label if bit 9
                         ; of r1 is set
```

Bcc　Branch Conditionally　Bcc

Branch Operation

Format: inst rel_addr　　**Operation:** if (cc=true) then cPC ← (cPCL+rd)

Format Key:

rd	=	Relative Displacement
cPC	=	Current Program Counter
cPCL	=	Current Program Counter (Address from the 1st byte of the instruction, 32-bit aligned)
rel_addr	=	cPCL+rd
nPC	=	Next PC
cc	=	Condition Code

Syntax:

Branch		**Instruction Code**
B<cc><.d>	s21	`00000ssssssssss0SSSSSSSSSSNQQQQQ`
Branch Far (Unconditional)		
B<.d>	s25	`00000ssssssssss1SSSSSSSSSSNRtttt`

Delay Slot Modes <.d>:

Delay Slot Mode	N Flag	Description
ND	0	Only execute next instruction when *not* branching (*default, if no <.d> field syntax*)
D	1	Always execute next instruction

Condition Codes <cc>:

Condition Code	Q Field	Description	Test	Condition Code	Q Field	Description	Test
AL, RA	00000	Always	1	VC, NV	01000	Over-flow clear	/V
EQ, Z	00001	Zero	Z	GT	01001	Greater than (signed)	(N and V and /Z) or (/N and /V and /Z)
NE, NZ	00010	Non-Zero	/Z	GE	01010	Greater than or equal to (signed)	(N and V) or (/N and /V)
PL, P	00011	Positive	/N	LT	01011	Less than (signed)	(N and /V) or (/N and V)
MI, N	00100	Negative	N	LE	01100	Less than or equal to (signed)	Z or (N and /V) or (/N and V)
CS, C, LO	00101	Carry set, lower than (unsigned)	C	HI	01101	Higher than (unsigned)	/C and /Z
CC, NC, HS	00110	Carry clear, higher or same	/C	LS	01110	Lower than or same (unsigned)	C or Z

Condition Code	Q Field	Description	Test	Condition Code	Q Field	Description	Test
VS, V	00111	(unsigned) Over-flow set	V	PNZ	01111	Positive non-zero	/N and /Z

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L	= Limm Data

Related Instruction:

BLcc

Bcc_S

Description:

When a conditional branch is used and the specified condition is met (cc = true), program execution is resumed at location PC (actually PCL) + relative displacement, where PC is the address of the Bcc instruction . The conditional branch instruction has a maximum range of +/- 1MByte, and the target address is 16-bit aligned.

The unconditional branch far format has a maximum branch range of +/- 16Mbytes. Since the execution of the instruction that is in the delay slot is controlled by the delay slot mode, it should never be the target of any branch or jump instruction.

CAUTION The Bcc instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D instruction.

Pseudo Code Example:

```
if cc==true then                            /* Bcc */
 if N=1 then
  DelaySlot(nPC)
 PC = cPC + rd
else
 PC = nPC
```

Assembly Code Example:

```
BEQ label               ; Branch to label if Z flag is ; set
                        ; Branch to label and execute
BPL.D label             ; the instruction in the delay
                        ; slot if N flag is clear
```

9 Instruction Set Details

Bcc_S — 16-Bit Branch — Bcc_S

Branch Operation

Format: inst rel_addr **Operation:** if (cc=true) then cPC ← (cPCL+rd)

Format Key:

rd	=	Relative Displacement
cPC	=	Current Program Counter
cPCL	=	Current Program Counter (Address from the 1st byte of the instruction, 32-bit aligned)
rel_addr	=	cPCL+rd
nPC	=	Next PC
cc	=	Condition Code

Syntax:

Branch Conditionally		**Instruction Code**
BEQ_S	s10	`111101sssssssss`
BNE_S	s10	`111110sssssssss`
BGT_S	s7	`1111011000ssssss`
BGE_S	s7	`1111011001ssssss`
BLT_S	s7	`1111011010ssssss`
BLE_S	s7	`1111011011ssssss`
BHI_S	s7	`1111011100ssssss`
BHS_S	s7	`1111011101ssssss`
BLO_S	s7	`1111011110ssssss`
BLS_S	s7	`1111011111ssssss`
Branch Always		
B_S	s10	`111100sssssssss`

Conditions:

Instruction	**Description**	**Branch Condition**
BEQ_S	Branch if Equal	if (Z) then cPC ← (cPCL+rd)
BNE_S	Branch if Not Equal	if (/Z) then cPC ← (cPCL+rd)
BGT_S	Branch if Greater Than	if (N and V and /Z) or (/N and /V and /Z) then cPC ← (cPCL+rd)
BGE_S	Branch if Greater Than or Equal to	if (N and V) or (/N and /V) then cPC ← (cPCL+rd)
BLT_S	Branch if Less Than	if (N and /V) or (/N and V) then cPC ← (cPCL+rd)

Instruction	Description	Branch Condition
BLE_S	Branch if Less Than or Equal	if Z or (N and /V) or (/N and V) then cPC ← (cPCL+rd)
BHI_S	Branch if Higher Than	if (/C and /Z) then cPC ← (cPCL+rd)
BHS_S	Branch if Higher than or the Same	if (/C) then cPC ← (cPCL+rd)
BLO_S	Branch if Lower than	if (C) then cPC ← (cPCL+rd)
BLS_S	Branch if Lower or the Same	if C or Z then cPC ← (cPCL+rd)

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Related Instructions:

Bcc

Description:

A branch is taken from the current PC with the displacement value specified in the source operand (rd) when a condition(s) are met, depending upon the instruction type used.

When using the B_S instruction a branch is always executed from the current PC, 32-bit aligned, with the displacement value specified in the source operand (rd).

For all branch types, the branch target is 16-bit aligned.

CAUTION The Bcc_S instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D instruction.

Pseudo Code Example:

```
if cc==true then                          /* Bcc_S */
 KillDelaySlot(nPC)
 PC = cPCL + rd
else
 PC = nPC
```

Assembly Code Example:

```
BEQ_S label            ; Branch to label if Z flag is ; set
                       ; Branch to label if N flag is
BPL_S label            ; clear
```

BCLR **Bit Clear** BCLR

Logical Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← (src1 AND (NOT 2^{src2}))

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
dest	=	Destination
cc	=	Condition Code

Syntax:

With Result		**Instruction Code**	
BCLR<.f>	a,b,c	00100bbb00010000FBBBCCCCCCAAAAAA	
BCLR<.f>	a,b,u6	00100bbb01010000FBBBuuuuuuAAAAAA	
BCLR<.cc><.f>	b,b,c	00100bbb11010000FBBBCCCCCC0QQQQQ	
BCLR<.cc><.f>	b,b,u6	00100bbb11010000FBBBuuuuuu1QQQQQ	
BCLR<.f>	a,limm,c	00100110000010000F111CCCCCCAAAAAA	L
BCLR_S	b,b,u5	10111bbb101uuuuu	
Without Result			
BCLR<.f>	0,b,c	00100bbb00010000FBBBCCCCCC111110	
BCLR<.f>	0,b,u6	00100bbb01010000FBBBuuuuuu111110	
BCLR<.cc><.f>	0,limm,c	00100110110100000F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L	= Limm Data

Related Instructions:

BSET	BXOR
BTST	BMSK

Description:

Clear (0) an individual bit within the value that is specified by source operand 1 (src1). Source operand 2 (src2) contains a value that explicitly defines the bit-position that is to be cleared in source operand 1 (scr1). Only the bottom 5 bits of src2 are used as the bit value. The result is written into the destination register (dest).

Pseudo Code Example:

```
if cc==true then                                            /* BCLR */
 dest = src1 AND NOT(1 << (src2 & 31))
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
BCLR r1,r2,r3           ; Clear bit r3 of r2
                        ; and write result into r1
```

BIC Bitwise AND Operation with Inverted Source BIC

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 AND NOT src2

Format Key:

dest	=	Destination Register
scr1	=	Source Operand 1
scr2	=	Source Operand 2
cc	=	Condition code

Syntax:

With Result		**Instruction Code**	
BIC<.f>	a,b,c	00100bbb00000110FBBBCCCCCCAAAAAA	
BIC<.f>	a,b,u6	00100bbb01000110FBBBuuuuuuAAAAAA	
BIC<.f>	b,b,s12	00100bbb10000110FBBBssssssSSSSSS	
BIC<.cc><.f>	b,b,c	00100bbb11000110FBBBCCCCCC0QQQQQ	
BIC<.cc><.f>	b,b,u6	00100bbb11000110FBBBuuuuuu1QQQQQ	
BIC<.f>	a,limm,c	00100110000000110F111CCCCCCAAAAAA	L
BIC<.f>	a,b,limm	00100bbb00000110FBBB111110AAAAAA	L
BIC<.cc><.f>	b,b,limm	00100bbb11000110FBBB1111100QQQQQ	L
BIC_S	b,b,c	01111bbbccc00110	
Without Result			
BIC<.f>	0,b,c	00100bbb00000110FBBBCCCCCC111110	
BIC<.f>	0,b,u6	00100bbb01000110FBBBuuuuuu111110	
BIC<.f>	0,b,limm	00100bbb00000110FBBB111110111110	L
BIC<.cc><.f>	0,limm,c	00100110110000110F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

AND OR

XOR

Description:

Logical bitwise AND of source operand 1 (scr1) with the inverse of source operand 2 (src2) with the result written to the destination register.

Pseudo Code Example:

```
if cc==true then                                           /* BIC */
 dest = src1 AND NOT src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
BIC r1,r2,r3           ; AND r2 with the NOT of r3
                       ; and write result into r1
```

BLcc — Branch and Link — BLcc

Branch Operation

Format: inst rel_addr **Operation:** if (cc=true) then (cPC ← cPCL +rd) & (r31 ← nPC or dPC)

Format Key:

rel_addr	=	cPCL + Relative Displacement
rd	=	Relative Displacement
cc	=	Condition Code
cPC	=	Current Program Counter
cPCL	=	Current Program Counter (Address from the 1st byte of the instruction, 32-bit aligned)
nPC	=	Next PC
dPC	=	Next PC + 4 (address of the 2nd following instruction)

Syntax:

Branch and Link (Conditional)		**Instruction Code**
BL<.cc><.d>	s21	`00001ssssssssss00SSSSSSSSSSNQQQQQ`
Branch Far (Unconditional)		
BL<.d>	s25	`00001ssssssssss10SSSSSSSSSSNRtttt`
Branch and Link (Unconditional)		
BL_S	s13	`11111sssssssssss`

Delay Slot Modes <.d>:

Delay Slot Mode	N Flag	Blink (r31)	Description
ND	0	Next PC	Only execute next instruction when *not* branching (*if no <.d> field syntax*)
D	1	2nd following PC	Always execute next instruction

Condition Codes <cc>:

Condition Code	Q Field	Description	Test	Condition Code	Q Field	Description	Test
AL, RA	00000	Always	1	VC, NV	01000	Over-flow clear	/V
EQ, Z	00001	Zero	Z	GT	01001	Greater than (signed)	(N and V and /Z) or (/N and /V and /Z)
NE, NZ	00010	Non-Zero	/Z	GE	01010	Greater than or equal to (signed)	(N and V) or (/N and /V)
PL, P	00011	Positive	/N	LT	01011	Less than (signed)	(N and /V) or (/N and V)

Condition Code	Q Field	Description	Test	Condition Code	Q Field	Description	Test
MI, N	00100	Negative	N	LE	01100	Less than or equal to (signed)	Z or (N and /V) or (/N and V)
CS, C, LO	00101	Carry set, lower than (unsigned)	C	HI	01101	Higher than (unsigned)	/C and /Z
CC, NC, HS	00110	Carry clear, higher or same (unsigned)	/C	LS	01110	Lower than or same (unsigned)	C or Z
VS, V	00111	Over-flow set	V	PNZ	01111	Positive non-zero	/N and /Z

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Related Instructions:

Bcc_s

Description:

When a conditional branch and link is used and the specified condition is met (cc = true), program execution is resumed at location PC, 32-bit aligned, + relative displacement, where PC is the address of the BLcc instruction. Parallel to this, the return address is stored in the link register BLINK (r31). This address is taken either from the first instruction following the branch (current PC) or the instruction after that (next PC) according to the delay slot mode (.d).

CAUTION The BLcc and BL_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D instruction.

The conditional branch and link instruction has a maximum branch range of +/- 1MByte, and the target address is 32-bit aligned. The unconditional branch far format has a maximum branch range of +/- 16Mbytes. Since the execution of the instruction that is in the delay slot is controlled by the delay slot mode, it should never be the target of any branch or jump instruction.

NOTE For the 16-bit encoded instructions the target address is aligned to 32-bit. For example BL_S s13 only needs to encode the top 11 bits since the bottom 2 bits of s13 are always zero because of the 32-bit data alignment.

Pseudo Code Example:

```
if cc==true then                                     /* BLcc */
 if N=1 then
  BLINK = dPC
  DelaySlot(nPC)
 else
  BLINK = nPC
 PC = cPCL + rd
else
 PC = nPC
```

Assembly Code Example:

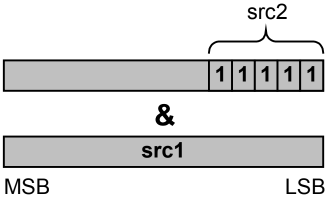
```
BLEQ label             ; if the Z flag is set then
                       ; branch and link to label
                       ; and store the return address in BLINK
```

This page is intentionally left blank.

BMSK **Bit Mask** BMSK

Logical Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 AND $((2^{(src2+1)})-1)$



Format Key:

src1	=	Source Operand 1
scr2	=	Source Operand 2 (Mask Value)
dest	=	Destination
cc	=	Condition Code

Syntax:

With Result		**Instruction Code**	
BMSK<.f>	a,b,c	00100bbb00010011FBBBCCCCCCAAAAAA	
BMSK<.f>	a,b,u6	00100bbb01010011FBBBuuuuuuAAAAAA	
BMSK<.cc><.f>	b,b,c	00100bbb11010011FBBBCCCCCC0QQQQQ	
BMSK<.cc><.f>	b,b,u6	00100bbb11010011FBBBuuuuuu1QQQQQ	
BMSK<.f>	a,limm,c	0010011000010011F111CCCCCCAAAAAA	L
BMSK_S	b,b,u5	10111bbb110uuuuu	
Without Result			
BMSK<.f>	0,b,c	00100bbb00010011FBBBCCCCCC111110	
BMSK<.f>	0,b,u6	00100bbb01010011FBBBuuuuuu111110	
BMSK<.cc><.f>	0,limm,c	0010011011010011F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

BSET BXOR
BTST

Description:

Source operand 2 (src2) specifies the size of a 32-bit mask value in terms of logical 1's starting from the LSB of a 32-bit register up to and including the bit specified by operand 2(src2). Only the bottom 5 bits of src2 are used as the bit value.

A logical AND is performed with the mask value and source operand (src1). The result is written into the destination register (dest).

Pseudo Code Example:

```
if cc==true then                                         /* BMSK */
 dest = src1 AND ((1 << ((src2 & 31)+1)-1)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
BMSK r1,r2,8           ; Mask out the top 24 bits
                       ; of r2 and write result into
                       ; r1
```

BRcc Compare and Branch BRcc

Branch Operation

Format: inst src1, src2, rd **Operation:** if (cc=true) then cPC← (cPCL+rd)

Format Key:

rd	=	Relative displacement
src1	=	Source Operand 1
src2	=	Source Operand 2
cPC	=	Current Program Counter
cPCL	=	Current Program Counter (Address from 1st byte of the instruction, 32-bit aligned)
nPC	=	Next PC
dPC	=	Next PC + 4 (address of the 2nd following instruction)
cc	=	Condition Code

Syntax:

		Instruction Code	
BREQ<.d>	b,c,s9	00001bbbsssssss1SBBBCCCCCCN00000	
BREQ<.d>	b,u6,s9	00001bbbsssssss1SBBBUUUUUUN10000	
BREQ	b,limm,s9	00001bbbsssssss1SBBB111110000000	L
BREQ	limm,c,s9	00001110sssssss1S111CCCCCC000000	L
BRNE<.d>	b,c,s9	00001bbbsssssss1SBBBCCCCCCN00001	
BRNE<.d>	b,u6,s9	00001bbbsssssss1SBBBUUUUUUN10001	
BRNE	b,limm,s9	00001bbbsssssss1SBBB111110000001	L
BRNE	limm,c,s9	00001110sssssss1S111CCCCCC000001	L
BRLT<.d>	b,c,s9	00001bbbsssssss1SBBBCCCCCCN00010	
BRLT<.d>	b,u6,s9	00001bbbsssssss1SBBBUUUUUUN10010	
BRLT	b,limm,s9	00001bbbsssssss1SBBB111110000010	L
BRLT	limm,c,s9	00001110sssssss1S111CCCCCC000010	L
BRGE<.d>	b,c,s9	00001bbbsssssss1SBBBCCCCCCN00011	
BRGE<.d>	b,u6,s9	00001bbbsssssss1SBBBUUUUUUN10011	
BRGE	b,limm,s9	00001bbbsssssss1SBBB111110000011	L
BRGE	limm,c,s9	00001110sssssss1S111CCCCCC000011	L
BRLO<.d>	b,c,s9	00001bbbsssssss1SBBBCCCCCCN00100	
BRLO<.d>	b,u6,s9	00001bbbsssssss1SBBBUUUUUUN10100	
BRLO	b,limm,s9	00001bbbsssssss1SBBB111110000100	L
BRLO	limm,c,s9	00001110sssssss1S111CCCCCC000100	L
BRHS<.d>	b,c,s9	00001bbbsssssss1SBBBCCCCCCN00101	
BRHS<.d>	b,u6,s9	00001bbbsssssss1SBBBUUUUUUN10101	
BRHS	b,limm,s9	00001bbbsssssss1SBBB111110000101	L
BRHS	limm,c,s9	00001110sssssss1S111CCCCCC000101	L

		Instruction Code
BRNE_S	b,0,s8	11101bbb1sssssss
BREQ_S	b,0,s8	11101bbb0sssssss

Delay Slot Modes <.d>:

Delay Slot Mode	N Flag	Description
ND	0	Only execute next instruction when *not* branching (*default, if no <.d> field syntax*)
D	1	Always execute next instruction

Conditions:

Instruction	Description	Branch Condition
BREQ	Branch if Equal	if (src1=src2) then cPC ← (cPC+rd)
BRNE	Branch if Not Equal	if (src1!=src2) then cPC ← (cPC+rd)
BRLT	Branch if Less Than (Signed)	if (src1<src2) then cPC ← (cPC+rd)
BRGE	Branch if Greater Than or Equal (Signed)	if (src1>src2) then cPC ← (cPC+rd)
BRLO	Branch if Lower Than (Unsigned)	if (src1<src2) then cPC ← (cPC+rd)
BRHS	Branch if Higher Than or Same (Unsigned)	if (src1>src2) then cPC ← (cPC+rd)

Related Instructions:

BBIT0 BBIT1

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Description:

A branch is taken from the current PC, 32-bit aligned, with the displacement value specified in the source operand (rd) when source operand 1 (src1) and source operand 2 (src2) conditions are met. All 32-bit compare and branch instructions have two delay slots. The behavior of the 1st delay slot can be controlled by specifying the delay slot mode <.d>, however the following delay slot cannot be controlled, and any instruction present in the 2nd delay slot is killed if the branch is taken.

In the case of the 16-bit compare and branch instructions, BRNE_S compares source operand 1 (src1) against '0', and if scr1 is not equal to zero then a branch is taken from the current PC, 32-bit aligned with the displacement value specified in the source operand (rd).

BREQ_S performs the same comparison, however the branch is taken when source operand 1 (src1) is equal to zero.

The branch target is 16-bit aligned. Since the execution of the instruction that is in the delay slot is controlled by the delay slot mode, it should never be the target of any branch or jump instruction.

CAUTION	The BRcc and BRcc_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D instruction.

Pseudo Code Example:

```
Alu = src1 - src2                                 /* BRcc */
if cc==true then
 if N=1 then
  DelaySlot(nPC)
 KillDelaySlot(dPC)
 PC = cPCL + rd
else
 PC = nPC
```

Assembly Code Example:

```
BREQ r1,r2,label         ; Branch to label if r1 is
                         ; equal to r2
```

This page is intentionally left blank.

BRK_S **Breakpoint** **Debug Operation** BRK_S

Format: inst **Operation:** Halt and flush the processor

Format Key:

inst = Instruction Mnemonic

Syntax:

	Instruction Code
BRK_S	`0111111111111111`

Flag Affected:

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged
BH	●	= 1
H	●	= 1

Key:

L	= Limm Data

Related Instructions:

SLEEP	FLAG

Description:

The breakpoint instruction is decoded at stage one of the pipeline which consequently stalls stage one, and allows instructions in stages two, three and four to continue, i.e. flushing the pipeline.

Due to stage 2 to stage 1 dependencies, the breakpoint instruction behaves differently when it is placed in the delay slots of Branch, and Jump instructions. In these cases, the processor will stall stages one and two of the pipeline while allowing instructions in subsequent stages (three and four) to proceed to completion.

Interrupts are treated in the same manner by the processor as Branch, and Jump instructions when a BRK_S instruction is detected. Therefore, an interrupt that reaches stage two of the pipeline when a BRK_S instruction is in stage one will keep it in stage two, and flush the remaining stages of the pipeline. It is also important to note that an interrupt that occurs in the same cycle as a breakpoint is held off as the breakpoint is of a higher priority. An interrupt at stage three is allowed to complete when a breakpoint instruction is in stage one.

When a debugger performs a restart after a BRK_S instruction has halted the pipeline it must perform an instruction cache invalidation operation. This removes any cached copies of the BRK_S instruction, which will have been overwritten by the debugger in memory with the original instruction at the BRK_S location. The instruction cache invalidation operation is required even in processors that do not have an instruction cache, as it also has the effect of flushing the first stage of the processor pipeline.

NOTE If the H flag is set by the FLAG instruction (`FLAG 1`), three sequential NOP instructions should immediately follow. This means that BRK_S should not immediately follow a `FLAG 1` instruction, but should be separated by 3 NOP instructions.

Pseudo Code Example:

```
FlushPipe()                                /* BRK_S */
DEBUG[BH] = 1
DEBUG[H] = 1
Halt()
```

Assembly Code Example:

```
BRK_S                 ; Breakpoint
```

CAUTION BRK_S is not allowed in the delay slot of any branch or jump instruction.

BSET **Bit Set** BSET

Logical Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← (src1 OR (2^{src2}))

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
dest	=	Destination
cc	=	Condition Code

Syntax:

With Result		**Instruction Code**	
BSET<.f>	a,b,c	00100bbb00001111FBBBCCCCCCAAAAAA	
BSET<.f>	a,b,u6	00100bbb01001111FBBBuuuuuuAAAAAA	
BSET<.cc><.f>	b,b,c	00100bbb11001111FBBBCCCCCC0QQQQQ	
BSET<.cc><.f>	b,b,u6	00100bbb11001111FBBBuuuuuu1QQQQQ	
BSET<.f>	a,limm,c	0010011000001111F111CCCCCCAAAAAA	L
BSET_S	b,b,u5	10111bbb100uuuuu	
Without Result			
BSET<.f>	0,b,c	00100bbb00001111FBBBCCCCCC111110	
BSET<.f>	0,b,u6	00100bbb01001111FBBBuuuuuu111110	
BSET<.cc><.f>	0,limm,c	0010011011001111F110CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

BCLR	BXOR
BTST	BMSK

Description:

Set (1) an individual bit within the value that is specified by source operand 1 (src1). Source operand 2 (src2) contains a value that explicitly defines the bit-position that is to be set in source operand 1 (scr1). Only the bottom 5 bits of src2 are used as the bit value. The result is written into the destination register (dest).

Pseudo Code Example:

```
if cc==true then                                    /* BSET */
 dest = src1 OR (1 << (src2 & 31))
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
BSET r1,r2,r3          ; Set bit r3 of r2
                       ; and write result into r1
```

BTST Bit Test Logical Operation BTST

Format: inst src1, src2 **Operation:** if (cc=true) then **(**src1 AND (2^{src2}))

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition Code

Syntax:

		Instruction Code	
BTST<.cc>	b,c	`00100bbb110100011BBBCCCCCC0QQQQQ`	
BTST<.cc>	b,u6	`00100bbb110100011BBBuuuuuu1QQQQQ`	
BTST<.cc>	limm,c	`00100110110100011111CCCCCC0QQQQQ`	L
BTST_S	b,u5	`10111bbb111uuuuu`	

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

BCLR	BXOR
BSET	BMSK

Description:

Logically AND source operand 1 (src1) with a bit mask specified by source operand 2 (src2). Source operand 2 (src2) explicitly defines the bit that is tested in source operand 1 (src1). Only the bottom 5 bits of src2 are used as the bit value. The flags are updated to reflect the result.

There is no result write-back.

NOTE BTST and BTST_S always set the flags even thought there is no associated flag setting suffix.

Pseudo Code Example:

```
if cc==true then                                 /* BTST */
 alu = src1 AND (1 << (src2 & 31))
 Z_flag = if alu==0 then 1 else 0
 N_flag = alu[31]
```

Assembly Code Example:

```
BTST r1,r2,28          ; Test bit 28 of r2
                       ; and update flags on result
```

BXOR Bit Exclusive OR (Bit Toggle) BXOR

Logical Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← (src1 XOR (2^{src2}))

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
dest	=	Destination
cc	=	Condition Code

Syntax:

With Result		**Instruction Code**	
BXOR<.f>	a,b,c	00100bbb00010010FBBBCCCCCCAAAAAA	
BXOR<.f>	a,b,u6	00100bbb01010010FBBBuuuuuuAAAAAA	
BXOR<.cc><.f>	b,b,c	00100bbb11010010FBBBCCCCCC0QQQQQ	
BXOR<.cc><.f>	b,b,u6	00100bbb11010010FBBBuuuuuu1QQQQQ	
BXOR<.f>	a,limm,c	0010011000010010F111CCCCCCAAAAAA	L
Without Result			
BXOR<.f>	0,b,c	00100bbb00010010FBBBCCCCCC111110	
BXOR<.f>	0,b,u6	00100bbb01010010FBBBuuuuuu111110	
BXOR<.cc><.f>	0,limm,c	0010011011010010F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

BSET	BTST
BCLR	BMSK

Description:

Logically XOR source operand 1 (src1) with a bit mask specified by source operand 2 (src2). Source operand 2 (src2) explicitly defines the bit that is to be toggled in source operand 1 (src1). Only the bottom 5 bits of src2 are used as the bit value. The result is written to the destination register (dest).

Pseudo Code Example:

```
if cc==true then                                        /* BXOR */
 dest = src1 XOR (1 << (src2 & 31))
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
BXOR r1,r2,r3       ; Toggle bit r3 of r2
                    ; and write result into r1
```

CMP **Comparison** CMP

Arithmetic Operation

Format: inst src1, src2 **Operation:** if (cc=true) then src1 – src2

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition Code

Syntax:

		Instruction Code	
CMP	b,s12	`00100bbb100011001BBBssssssSSSSSS`	
CMP<.cc>	b,c	`00100bbb110011001BBBCCCCCC0QQQQQ`	
CMP<.cc>	b,u6	`00100bbb110011001BBBuuuuuu1QQQQQ`	
CMP<.cc>	b,limm	`00100bbb110011001BBB1111100QQQQQ`	L
CMP<.cc>	limm,c	`00100110110011001111CCCCCC0QQQQQ`	L
CMP_S	b,h	`01110bbbhhh10HHH`	
CMP_S	b,limm	`01110bbb11010111`	L
CMP_S	b,u7	`11100bbb1uuuuuuu`	

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if overflow is generated

Key:

L = Limm Data

Related Instructions:

RCMP

Description:

A comparison is performed by subtracting source operand 2 (src2) from source operand 1 (src1) and subsequently updating the flags.

There is no destination register therefore the result of the subtract is discarded.

NOTE CMP and CMP_S always set the flags even thought there is no associated flag setting suffix .

Pseudo Code Example:

```
if cc==true then                                  /* CMP */
 alu = src1 - src2
 Z_flag = if alu==0 then 1 else 0
 N_flag = alu[31]
 C_flag = Carry()
 V_flag = Overflow()
```

Assembly Code Example:

```
CMP r1,r2               ; Subtract r2 from r1
                        ; and set the flags on the
                        ; result
```

EXTB — Zero Extend Byte — EXTB

Arithmetic Operation

Format: inst dest, src **Operation:** dest ← zero extend from byte (src)

Format Key:

src	=	Source Operand
dest	=	Destination

Syntax:

With Result		**Instruction Code**	
EXTB<.f>	b,c	00100bbb00101111FBBBCCCCCC000111	
EXTB<.f>	b,u6	00100bbb01101111FBBBuuuuuu000111	
EXTB<.f>	b,limm	00100bbb00101111FBBB111110000111	L
EXTB_S	b,c	01111bbbccc01111	
Without Result			
EXTB<.f>	0,c	0010011000101111F111CCCCCC000111	
EXTB<.f>	0,u6	0010011001101111F111uuuuuu000111	
EXTB<.f>	0,limm	0010011000101111F111111110000111	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Always Zero
C		= Unchanged
V		= Unchanged

Key:

L	= Limm Data

Related Instructions:

SEXB	ABS
SEXW	EXTW

Description:

Zero extend the byte value in the source operand (src) and write the result into the destination register.

Pseudo Code Example:

```
dest = src & 0xFF                                     /* EXTB */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
EXTB r3,r0             ; Zero extend the bottom 8
                       ; bits of r0 and write
                       ; result to r3
```

EXTW **Zero Extend Word** EXTW

Arithmetic Operation

Format: inst dest, src **Operation:** dest ← zero extend from word (src)

Format Key:

src	=	Source Operand
dest	=	Destination

Syntax:

With Result		**Instruction Code**	
EXTW<.f>	b,c	`00100bbb00101111FBBBCCCCCC001000`	
EXTW<.f>	b,u6	`00100bbb01101111FBBBuuuuuu001000`	
EXTW<.f>	b,limm	`00100bbb00101111FBBB111110001000`	L
EXTW_S	b,c	`01111bbbccc10000`	
Without Result			
EXTW<.f>	0,c	`0010011000101111F111CCCCCC001000`	
EXTW<.f>	0,u6	`0010011001101111F111uuuuuu001000`	
EXTW<.f>	0,limm	`0010011000101111F111111110001000`	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Always Zero
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

SEXB	ABS
SEXW	EXTB

Description:

Zero extend the word value in the source operand (src) and write the result into the destination register.

Pseudo Code Example:

```
dest = src & 0xFFFF                                         /* EXTW */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

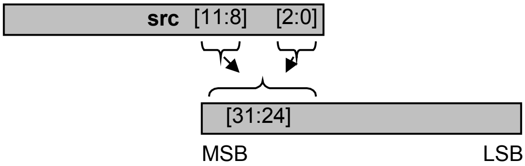
Assembly Code Example:

```
EXTW r3,r0               ; Zero extend the bottom 16
                         ; bits of r0 and write
                         ; result to r3
```

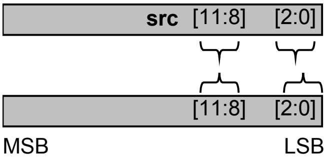
FLAG **Set Flags** FLAG

Control Operation

Format: inst src **Operation:** if (cc=true) then flags ← src



STATUS Register
Auxiliary (0x00)



STATUS32 Register
Auxiliary (0x0A)

Format Key:

src = Source Operand

Syntax:

		Instruction Code	
FLAG<.cc>	c	00100rrr11101001 0RRRCCCCCC0QQQQQ	
FLAG<.cc>	u6	00100rrr11101001 0RRRuuuuuu1QQQQQ	
FLAG<.cc>	limm	00100rrr11101001 0RRR1111100QQQQQ	L
FLAG	s12	00100rrr10101001 0RRRssssssSSSSSS	

Source Operand Flag Positions:

Z	●	= Bit 11 of Source Operand
N	●	= Bit 10 of Source Operand
C	●	= Bit 9 of Source Operand
V	●	= Bit 8 of Source Operand
E2	●	= Bit 2 of Source Operand
E1	●	= Bit 1 of Source Operand
H	●	= Bit 0 of Source Operand (If set ignore all other flags states)

Key:

L = Limm Data

Related Instructions:

SLEEP

Description:

The contents of the source operand (src) are used to set the condition code and processor control flags held in the processor status registers.

NOTE Interrupts are held off until the FLAG instruction completes. This includes a software interrupt instruction (SWI).

CAUTION There should be at least two instructions issued between a FLAG and SWI if the SWI could be followed immediately by a BRK_S instruction. Failure to observe this caution will result in the BRK_S instruction being handled by the processor before the SWI instruction is allowed to interrupt.

Bits [11:8] of the source operand relate to the condition codes, [2:1] relate to the interrupt masks and bit [0] relates to the halt flag. Bits [31:12] and [7:3] are ignored.

The format of the source operand is identical to the format used by the STATUS32 register (auxiliary address 0x0A).

Both the (obsolete) Status Register (auxiliary address 0x00) and STATUS32 register (auxiliary address 0x0A) are updated automatically upon using the FLAG instruction.

NOTE If the H flag is set (halt processor flag), all other flag states are ignored and are not updated.

Pseudo Code Example:

```
if src[0]==1 then                           /* FLAG */
 STATUS32[0] = 1
 Halt()
else
 STATUS32[31:1] = src[31:1]
```

Assembly Code Example:

```
FLAG 1                 ; Halt processor (other flags
                       ; not updated)
NOP                    ; Pipeline Flush
NOP                    ; Pipeline Flush
NOP                    ; Pipeline Flush
FLAG 6                 ; Enable interrupts and clear
                       ; all other flags
```

NOTE If the H flag is set (FLAG 1), three sequential NOP instructions should immediately follow. This ensures that instructions that succeed a FLAG 1 instruction upon a processor restart, execute correctly.

Jcc **Jump Conditionally** Jcc

Jump Operation

Format: inst src **Operation:** if (cc=true) then cPC ← src **

Format Key:

src	=	Source Operand
cPC	=	Current Program Counter
nPC	=	Next PC
cc	=	Condition Code
**	=	Special condition when instruction sets flags (.F) and src = ILINK1 or ILINK2

Syntax:

Jump (Conditional)		**Instruction Code**	
Jcc	[c]	00100RRR111000000RRRCCCCCC0QQQQQ	
Jcc	limm	00100RRR111000000RRR1111100QQQQQ	L
Jcc	u6	00100RRR111000000RRRuuuuuu1QQQQQ	
Jcc.D	u6	00100RRR111000010RRRuuuuuu1QQQQQ	
Jcc.D	[c]	00100RRR111000010RRRCCCCCC0QQQQQ	
Jcc.F	[ilink1]	00100RRR111000001RRR0111010QQQQQ	
Jcc.F	[ilink2]	00100RRR111000001RRR0111100QQQQQ	
JEQ_S	[blink]	0111110011100000	
JNE_S	[blink]	0111110111100000	
Jump (Unconditional)			
J	[c]	00100RRR001000000RRRCCCCCCRRRRRR	
J.D	[c]	00100RRR001000010RRRCCCCCCRRRRRR	
J.F	[ilink1]	00100RRR001000001RRR011101RRRRRR	
J.F	[ilink2]	00100RRR001000001RRR011110RRRRRR	
J	limm	00100RRR001000000RRR111110RRRRRR	L
J	u6	00100RRR011000000RRRuuuuuuRRRRRR	
J.D	u6	00100RRR011000010RRRuuuuuuRRRRRR	
J	s12	00100RRR101000000RRRssssssSSSSSS	
J.D	s12	00100RRR101000010RRRssssssSSSSSS	
J_S	[b]	01111bbb00000000	
J_S.D	[b]	01111bbb00100000	
J_S	[blink]	0111111011100000	
J_S.D	[blink]	0111111111100000	

Delay Slot Modes:

Delay Slot Mode	Description
J/J_S/JEQ_S/JNE_S	Only execute next instruction when *not* branching
Jcc.D/J.D /J_S.D	Always execute next instruction

Condition Codes <cc>:

Condition Code	Q Field	Description	Test	Condition Code	Q Field	Description	Test
AL, RA	00000	Always	1	VC, NV	01000	Over-flow clear	/V
EQ, Z	00001	Zero	Z	GT	01001	Greater than (signed)	(N and V and /Z) or (/N and /V and /Z)
NE, NZ	00010	Non-Zero	/Z	GE	01010	Greater than or equal to (signed)	(N and V) or (/N and /V)
PL, P	00011	Positive	/N	LT	01011	Less than (signed)	(N and /V) or (/N and V)
MI, N	00100	Negative	N	LE	01100	Less than or equal to (signed)	Z or (N and /V) or (/N and V)
CS, C, LO	00101	Carry set, lower than (unsigned)	C	HI	01101	Higher than (unsigned)	/C and /Z
CC, NC, HS	00110	Carry clear, higher or same (unsigned)	/C	LS	01110	Lower than or same (unsigned)	C or Z
VS, V	00111	Over-flow set	V	PNZ	01111	Positive non-zero	/N and /Z

Flags Updated (src=ILINK1\2 & .F)

Z	●	= Set if bit[11] of STATUS_L1 or STATUS_L2 set
N	●	= Set if bit[10] of STATUS_L1 or STATUS_L2 set
C	●	= Set if bit[9] of STATUS_L1 or STATUS_L2 set
V	●	= Set if bit[8] of STATUS_L1 or STATUS_L2 set
E2	●	= Set if bit[2] of STATUS_L1 or STATUS_L2 set
E1	●	= Set if bit[1] of STATUS_L1 or STATUS_L2 set

Key:

L = Limm Data

Related Instructions:

JLcc

Special Conditions:

Source Operand (src)	Operation

src = ILINK1 & .F	pc ← ILINK1
	STATUS32 ← STATUS32_L1

src = ILINK2 & .F	pc ← ILINK2
	STATUS32 ← STATUS32_L2

Description:

If the specified condition is met (cc=true), then the program execution is resumed from the new program counter address that is specified in the source operand (src). Jump instructions have can target any address within the full memory address map, but the target address is 16-bit aligned. Since the execution of the instruction that is in the delay slot is controlled by the delay slot mode, it should never be the target of any branch or jump instruction.

CAUTION The Jcc and Jcc_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D instruction.

When using ILINK1 or ILINK2 as the source operand with Jcc.F or J.F, the contents of the corresponding registers STATUS32_L1 or STATUS32_L2 are automatically copied over to STATUS32.

NOTE A single instruction must separate a FLAG instruction from any type of Jcc.F [ILINK1\2] instruction if they proceed each other. In addition, a single instruction must also separate the auxiliary register write update of STATUS32_L1 or STATUS32_L2 and any type of Jcc.F [ilink1\2] instruction.

Pseudo Code Example:

```
if cc==true then                                            /* Jcc */
 if N==1 then
  DelaySlot(nPC)
 PC = src
 if F==1 and src==ILINK1 then
  STATUS32 = STATUS32_L1
 if F==1 and src==ILINK2 then
  STATUS32 = STATUS32_L2
else
 PC = nPC
```

Assembly Code Example:

```
JEQ [r1]                ; jump to address in r1 if the
                        ; Z flag is set
J.F [ilink1]            ; jump to address in ilink1
                        ; and restore STATUS32 from
                        ; STATUS_L1
```

This page is intentionally left blank.

JLcc — Jump and Link Conditionally — JLcc

Jump Operation

Format: inst src

Operation: if (cc=true) then (cPC ← src) & (BLINK ← nPC)

Format Key:

src	=	Source Operand
cPC	=	Program Counter
cc	=	Condition Code
BLINK	=	Branch and Link Register (r31)
nPC	=	Next PC
dPC	=	Next PC + 4 (address of the 2nd following instruction)

Syntax:

Jump		**Instruction Code**	
JLcc	[c]	`00100RRR111000100RRRCCCCCC0QQQQQ`	
JLcc	limm	`00100RRR111000100RRR1111100QQQQQ`	L
JLcc	u6	`00100RRR111000100RRRuuuuuu1QQQQQ`	
JLcc.D	u6	`00100RRR111000110RRRuuuuuu1QQQQQ`	
JLcc.D	[c]	`00100RRR111000110RRRCCCCCC0QQQQQ`	
Jump (Unconditional)			
JL	[c]	`00100RRR001000100RRRCCCCCCRRRRRR`	
JL.D	[c]	`00100RRR001000110RRRCCCCCCRRRRRR`	
JL	limm	`00100RRR001000100RRR111110RRRRRR`	L
JL	u6	`00100RRR011000100RRRuuuuuuRRRRRR`	
JL.D	u6	`00100RRR011000110RRRuuuuuuRRRRRR`	
JL	s12	`00100RRR101000100RRRssssssSSSSSS`	
JL.D	s12	`00100RRR101000110RRRssssssSSSSSS`	
JL_S	[b]	`01111bbb01000000`	
JL_S.D	[b]	`01111bbb01100000`	

Delay Slot Modes:

Delay Slot Mode	Description
JLcc/JL/JL_S	Only execute next instruction when *not* branching
JLcc.D/JL.D/JL_S.D	Always execute next instruction

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Condition Codes <cc>:

Condition Code	Q Field	Description	Test	Condition Code	Q Field	Description	Test
AL, RA	00000	Always	1	VC, NV	01000	Over-flow clear	/V
EQ, Z	00001	Zero	Z	GT	01001	Greater than (signed)	(N and V and /Z) or (/N and /V and /Z)
NE, NZ	00010	Non-Zero	/Z	GE	01010	Greater than or equal to (signed)	(N and V) or (/N and /V)
PL, P	00011	Positive	/N	LT	01011	Less than (signed)	(N and /V) or (/N and V)
MI, N	00100	Negative	N	LE	01100	Less than or equal to (signed)	Z or (N and /V) or (/N and V)
CS, C, LO	00101	Carry set, lower than (unsigned)	C	HI	01101	Higher than (unsigned)	/C and /Z
CC, NC, HS	00110	Carry clear, higher or same (unsigned)	/C	LS	01110	Lower than or same (unsigned)	C or Z
VS, V	00111	Over-flow set	V	PNZ	01111	Positive non-zero	/N and /Z

Related Instructions:

Jcc

Description:

If the specified condition is met (cc=true), then the program execution is resumed from the new program counter address that is specified in the source operand (src). Jump and link instructions have can target any address within the full memory address map, but the target address is 16-bit aligned. Parallel to this, the program counter address (PC) that immediately follows the jump instruction is written into the BLINK register (r31). Since the execution of the instruction that is in the delay slot is controlled by the delay slot mode, it should never be the target of any branch or jump instruction.

CAUTION The JLcc and JL_S instructions cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D instruction.

Pseudo Code Example:

```
if cc==true then                          /* JLcc */
 if N==1 then
  BLINK = dPC
  DelaySlot(nPC)
 else
  BLINK = nPC
 PC = src
else
 PC = nPC
```

Assembly Code Example:

```
JLEQ [r1]     ; if the Z flag is set then jump and link to address
              ; in r1 and store the return address in BLINK
```

This page is intentionally left blank.

LD Delayed Load from Memory LD

Memory Operation

Format: inst dest, src1, src2 **Operation:** dest ← Result of Memory Load address @ (src1+src2)

Format Key:

src1 = Source Operand 1
src2 = Source Operand 2 (Offset)
dest = Destination

Syntax:

		Instruction Code	
LD<zz><.x><.aa><.di>	a,[b,s9]	00010bbbssssssssSBBBDaaZZXAAAAAA	
LD<zz><.x><.di>	a,[limm]	00010110ssssssssS111DRRZZXAAAAAA	L
LD<zz><.x><.aa><.di>	a,[b,c]	00100bbbaa110ZZXDBBBCCCCCCAAAAAA	
LD<zz><.x><.aa><.di>	a,[b,limm]	00100bbbaa110ZZXDBBB111110AAAAAA	L
LD<zz><.x><.di>	a,[limm,c]	00100110RR110ZZXD111CCCCCCAAAAAA	L
LD_S	a,[b,c]	01100bbbccc00aaa	
LDB_S	a,[b,c]	01100bbbccc01aaa	
LDW_S	a,[b,c]	01100bbbccc10aaa	
LD_S	c,[b,u7]	10000bbbcccuuuuu	
LDB_S	c,[b,u5]	10001bbbcccuuuuu	
LDW_S	c,[b,u6]	10010bbbcccuuuuu	
LDW_S.X	c,[b,u6]	10011bbbcccuuuuu	
LD_S	b,[sp,u7]	11000bbb000uuuuu	
LDB_S	b,[sp,u7]	11000bbb001uuuuu	
LD_S	r0,[gp,s11]	1100100sssssssss	
LDB_S	r0,[gp,s9]	1100101sssssssss	
LDW_S	r0,[gp,s10]	1100110sssssssss	
LD_S	b,[pcl,u10]	11010bbbuuuuuuuu	

Data Size Field <.zz>:

Data Size Syntax	ZZ Field	Description
No Field Syntax	00	Data is a long-word (32-Bits) (*<.x> syntax illegal*)
W	10	Data is a word (16-Bits)
B	01	Data is a byte (8-Bits)
	11	*reserved*

Sign Extend <.x>:

X Flag	Description
0	No sign extension (*default, if no <.x> field syntax*)
1	Sign extend data from most significant bit of data to the most significant bit of long-word

Data Cache Mode <.di>:

D Flag	Description
0	Cached data memory access (*default, if no <.di> field syntax*)
1	Non-cached data memory access (*bypass data cache*)

Address Write-back Mode <.aa>:

Address Write-back Syntax	aa Field	Effective Address	Address Write-Back
No Field Syntax	00	Address = src1+src2 (*register+offset*)	None
.A or .AW	01	Address = src1+src2 (*register+offset*)	src1 ← src1+src2 (*register+offset*)
.AB	10	Address = src1 (*register*)	src1 ← src1+src2 (*register+offset*)

.AS	11	Address = src1+(src2<<1) (<*zz*>= *'10'*)
		Address = src1+(src2<<2) (<*zz*>= *'00'*)
			None*. *Using a byte or signed byte data size is invalid and is a reserved format*

Flag Affected (32-Bit):

Z □ = Unchanged

N □ = Unchanged

C □ = Unchanged

V □ = Unchanged

Key:

L = Limm Data

16-Bit Load Instructions Operation:

Instruction	Format	Operation	Description
LD_S	a, [b,c]	dest ← address[src1+src2].l	Load long word from address calculated by register + register
LDB_S	a, [b,c]	dest ← address[src1+src2].b	Load unsigned byte from address calculated by register + register
LDW_S	a, [b,c]	dest ← address[src1+src2].w	Load unsigned word from address calculated by register + register
LD_S	c, [b,u7]	dest ← address[src1+u7].l	Load long word from address calculated by register + unsigned immediate
LDB_S	c, [b,u5]	dest ← address[src1+u5].b	Load unsigned byte from address calculated by register + unsigned immediate
LDW_S	c, [b,u6]	dest ← address[src1+u6].w	Load unsigned word from address calculated by register + unsigned immediate
LDW_S.X	c, [b,u6]	dest ← address[src1+u6].w	Load signed word from address calculated by register + unsigned immediate
LD_S	b, [sp,u7]	dest ← address[sp+u7].l	Load word from address calculated by Stack Pointer (r28) + unsigned immediate
LDB_S	b, [sp,u7]	dest ← address[sp+u7].b	Load unsigned byte from address calculated by Stack Pointer (r28) + unsigned immediate
LD_S	r0, [gp,s11]	dest ← address[gp+s11].l	Load long word from address calculated by Global Pointer (r26) + signed immediate (signed immediate is 32-bit aligned) and write the result into r0
LDB_S	r0, [gp,s9]	dest ← address[gp+s9].b	Load unsigned byte from address calculated by Global Pointer (r26) + signed immediate (signed immediate is 8-bit aligned) and write the result

Instruction	Format	Operation	Description
			into r0
LDW_S	r0, [gp,s10]	dest ← address[gp+s10].w	Load unsigned word from address calculated by Global Pointer (r26) + signed immediate (signed immediate is 16-bit aligned) and write the result into r0
LD_S	b, [pcl,u10]	dest ← address[pcl+u10]	Load long word from address calculated by longword aligned program counter (pcl) + unsigned immediate (unsigned immediate is 32-bit aligned).

Related Instructions:

ST	SR
LR	

Description:

A memory load occurs from the address that is calculated by adding source operand 1 (src1) with source operand 2 (scr2) and the returning load data is written into the destination register (dest).

NOTE For the 16-bit encoded instructions that work on the stack pointer (SP) or global pointer (GP) the offset is aligned to 32-bit. For example LD_S b,[sp,u7] only needs to encode the top 5 bits since the bottom 2 bits of u7 are always zero because of the 32-bit data alignment.

The size of the requested data is specified by the data size field <.zz> and by default data is zero extended from the most significant bit of the data to the most significant bit of the long-word.

NOTE When a memory controller is employed: Load bytes can be made to any byte alignments, Load words should be made from word aligned addresses and Load longs should be made only from long aligned addresses.

Data can be sign extended by enabling sign extend <.x>. Note that using the sign extend suffix on the LD instruction with a 32-bit data size is undefined and should not be used. If the processor contains a data cache, load requests can bypass the cache by using the <.di> syntax. The address write-back mode can be selected by use of the <.aa> syntax. Note than when using the scaled source addressing mode (.AS), the scale factor is dependent upon the size of the data word requested (.zz).

NOTE LP_COUNT should not be used as the destination of a load. For example the following instruction is *not* allowed: LD LP_COUNT, [r0]

Pseudo Code Example:

```
if AA==0 then address = src1 + src2      /* LD */
if AA==1 then address = src1 + src2
if AA==2 then address = src1
if AA==3 and ZZ==0 then
 address = src1 + (src2 << 2)
if AA==3 and ZZ==2 then
 address = src1 + (src2 << 1)
if AA==1 or AA==2 then
 src1 = src1 + src2
DEBUG[LD] = 1

dest = Memory(address, size)             /* On Returning Load */
if X==1 then
 dest = Sign_Extend(dest, size)
if NoFurtherLoadsPending() then
 DEBUG[LD] = 0
```

Assembly Code Example:

```
LD r0,[r1,4]           ; Load long word from memory
                       ; address r1+4 and write
                       ; result to r0
```

LPcc **Loop Set Up** LPcc

Branch Operation

Format: inst rel_addr **Operation:** if (cc=false) then cPC ← (cPCL+rd) else (LP_END ← cPCL+rd) & (LP_START ← nPC)

Format Key:

rel_addr	=	cPCL + rd
rd	=	Relative Displacement
cc	=	Condition Code
cPC	=	Current Program Counter
cPCL	=	Current Program Counter (Address from the 1st byte of the instruction, 32-bit aligned)
nPC	=	Next PC
LP_START	=	32-Bit Loop Start Auxiliary Register (0x02)
LP_END	=	32-Bit Loop End Auxiliary Register (0x03)

Syntax:

Loop Set Up (Conditional)		**Instruction Code**
LP<cc>	u7	`00100RRR111010000RRRuuuuuu1QQQQQ`
Loop Set Up (Unconditional)		
LP	s13	`00100RRR101010000RRRssssssSSSSSS`

Condition Codes <cc>:

Condition Code	Q Field	Description	Test	Condition Code	Q Field	Description	Test
AL, RA	00000	Always	1	VC, NV	01000	Over-flow clear	/V
EQ, Z	00001	Zero	Z	GT	01001	Greater than (signed)	(N and V and /Z) or (/N and /V and /Z)
NE, NZ	00010	Non-Zero	/Z	GE	01010	Greater than or equal to (signed)	(N and V) or (/N and /V)
PL, P	00011	Positive	/N	LT	01011	Less than (signed)	(N and /V) or (/N and V)
MI, N	00100	Negative	N	LE	01100	Less than or equal to (signed)	Z or (N and /V) or (/N and V)
CS, C, LO	00101	Carry set, lower than (unsigned)	C	HI	01101	Higher than (unsigned)	/C and /Z
CC, NC, HS	00110	Carry clear, higher or same	/C	LS	01110	Lower than or same	C or Z

Condition Code	Q Field	Description	Test	Condition Code	Q Field	Description	Test
VS, V	00111	(unsigned) Over-flow set	V	PNZ	01111	(unsigned) Positive non-zero	/N and /Z

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Loop Operation:

Loop Format	Loop Operation (Conditional Execution <cc>)	
	True	False

LPcc u7	aux_reg[LP_END] = cPCL + u7
	aux_reg[LP_START] = nPC
		cPC ← cPCL + u7

LP s13	aux_reg[LP_END] = cPCL + s13
	aux_reg[LP_START] = nPC
		Always True

Related Instructions:

None

Description:

When the specified condition is *not* met whilst using the LPcc instruction, the relative displacement value (rd) is added to the current PC (actually cPCL) and program execution is subsequently resumed from the new 16-bit aligned cPC. In the event that the condition is met, the auxiliary register LP_END (auxiliary register 0x03) is updated with the resulting address of cPCL + rd. In parallel LP_START (auxiliary register 0x02) is updated with the next PC (nPC).

The non-conditional LP instruction always updates LP_END and LP_START auxiliary registers. Various programming cautions exist for the zero overhead loop mechanism, see section *Zero Overhead Loop* for further details.

CAUTION The LPcc instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D instruction.

Pseudo Code Example:

```
if cc==true then                                 /* LPcc */
 Aux_reg(LP_START) = nPC
 Aux_reg(LP_END) = cPCL + rd
 PC = nPC
else
 PC = cPCL +rd
```

Assembly Code Example:

```
LPNE label           ; if the Z flag is set then
                     ; branch to label else
                     ; set LP_START to address of
```

```
                                ; next instruction and set
                                ; LP_END to label
```

LR — Load from Auxiliary Register — LR

Control Operation

Format: inst dest, src **Operation:** dest ← aux_reg(src)

Format Key:

src	=	Source Operand
dest	=	Destination
aux_reg	=	Auxiliary Register

Syntax:

		Instruction Code	
LR	b,[c]	00100bbb00101010 0BBBCCCCCCRRRRRR	
LR	b,[limm]	00100bbb00101010 0BBB111110RRRRRR	L
LR	b,[s12]	00100bbb10101010 0BBBssssssSSSSSS	

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

SR

Description:

Get the data from the auxiliary register whose number is obtained from the source operand (src) and place the data into the destination register (dest).

Pseudo Code Example:

```
dest = Aux_reg(src)                         /* LR */
```

Assembly Code Example:

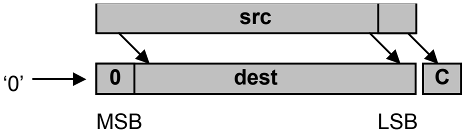
```
LR r1,[r2]              ; Load contents of Aux.
                        ; register r2 into r1
```

This page is intentionally left blank

LSR **Logical Shift Right** LSR

Logical Operation

Format: inst dest, src **Operation:** dest ← LSR by 1 (src)



Format Key:

dest	=	Destination Register
src	=	Source Operand

Syntax:

With Result		**Instruction Code**	
LSR<.f>	b,c	00100bbb00101111FBBBCCCCCC000010	
LSR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000010	
LSR<.f>	b,limm	00100bbb00101111FBBB111110000010	L
LSR_S	b,c	01111bbbccc11101	
Without Result			
LSR<.f>	0,c	0010011000101111F111CCCCCC000010	
LSR<.f>	0,u6	0010011001101111F111uuuuuu000010	
LSR<.f>	0,limm	0010011000101111F111111110000010	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

ASL	ASR
ROR	RRC

Description:

Logically right shift the source operand (src) by one and place the result into the destination register (dest).

The most significant bit of the result is replaced with 0.

Pseudo Code Example:

```
dest = src >> 1                                          /* LSR */
dest[31] = 0
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 C_flag = src[0]
```

Assembly Code Example:

```
LSR r1,r2               ; Logical shift right
                        ; contents of r2 by one bit
                        ; and write result into r1
```

LSR multiple

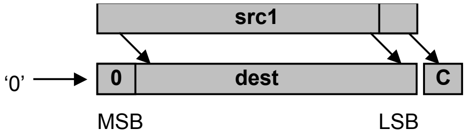
Multiple Logical Shift Right

Logical Operation

LSR multiple

Format: inst dest, src1, src2

Operation: if (cc=true) then dest ← logical shift right of src1 by src2



Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
LSR<.f>	a,b,c	00101bbb00000001FBBBCCCCCCAAAAAA	
LSR<.f>	a,b,u6	00101bbb01000001FBBBuuuuuuAAAAAA	
LSR<.f>	b,b,s12	00101bbb10000001FBBBssssssSSSSSS	
LSR<.cc><.f>	b,b,c	00101bbb11000001FBBBCCCCCC0QQQQQ	
LSR<.cc><.f>	b,b,u6	00101bbb11000001FBBBuuuuuu1QQQQQ	
LSR<.f>	a,limm,c	0010111000000001F111CCCCCCAAAAAA	L
LSR<.f>	a,b,limm	00101bbb00000001FBBB111110AAAAAA	L
LSR<.cc><.f>	b,b,limm	00101bbb11000001FBBB1111100QQQQQ	L
LSR_S	b,b,c	01111bbbccc11001	
LSR_S	b,b,u5	10111bbb001uuuuu	
Without Result			
LSR<.f>	0,b,c	00101bbb00000001FBBBCCCCCC111110	
LSR<.f>	0,b,u6	00101bbb01000001FBBBuuuuuu111110	
LSR<.cc><.f>	0,limm,c	0010111011000001F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

ASL	LSR
ROR	RRC

Description:

Logically, shift right src1 by src2 places and place the result in the destination register. Only the bottom 5 bits of src2 are used as the shift value.

Pseudo Code Example:

```
if cc==true then                                    /* LSR */
 dest = src1 >> (src2 & 31)                         /* Multiple */
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = if src2==0 then 0 else src1[sr2-1]
```

Assembly Code Example:

```
LSR r1,r2,r3           ; Logical shift right
                       ; contents of r2 by r3 bits
                       ; and write result into r1
```

MAX **Return Maximum Value** **Arithmetic Operation** MAX

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← MAX(src1, src2)

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition code
MAX	=	Return Maximum Value

Syntax:

With Result		**Instruction Code**	
MAX<.f>	a,b,c	00100bbb00001000FBBBCCCCCCAAAAAA	
MAX<.f>	a,b,u6	00100bbb01001000FBBBuuuuuuAAAAAA	
MAX<.f>	b,b,s12	00100bbb10001000FBBBssssssSSSSSS	
MAX<.cc><.f>	b,b,c	00100bbb11001000FBBBCCCCCC0QQQQQ	
MAX<.cc><.f>	b,b,u6	00100bbb11001000FBBBuuuuuu1QQQQQ	
MAX<.f>	a,limm,c	0010011000001000F111CCCCCCAAAAAA	L
MAX<.f>	a,b,limm	00100bbb00001000FBBB111110AAAAAA	L
MAX<.cc><.f>	b,b,limm	00100bbb11001000FBBB1111100QQQQQ	L
Without Result			
MAX<.f>	0,b,c	00100bbb00001000FBBBCCCCCC111110	
MAX<.f>	0,b,u6	00100bbb01001000FBBBuuuuuu111110	
MAX<.f>	0,b,limm	00100bbb00001000FBBB111110111110	L
MAX<.cc><.f>	0,limm,c	0010011011001000F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if both source operands are equal
N	●	= Set if most significant bit of result of src1-src2 is set
C	●	= Set if src2 is selected (src2 >= src1)
V	●	= Set if overflow is generated (as a result of src1-src2)

Key:

L = Limm Data

Related Instructions:

MIN

Description:

Return the maximum of the two signed source operands (src1 and src2) and place the result in the destination register (dest).

Pseudo Code Example:

```
if cc==true then                                            /* MAX */
 alu = src1 - src2
 if src2 >= src1 then
  dest = src2
 else
  dest = src1
 if F==1 then
  Z_flag = if alu==0 then 1 else 0
  N_flag = alu[31]
  V_flag = Overflow()
  C_flag = if src2>=src1 then 1 else 0
```

Assembly Code Example:

```
MAX r1,r2,r3           ; Take maximum of r2 and r3
                       ; and write result into r1
```

MIN **Return Minimum Value** MIN

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← MIN(src1, src2)

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition code
MIN	=	Return Minimum Value

Syntax:

With Result		**Instruction Code**	
MIN<.f>	a,b,c	00100bbb00001001FBBBCCCCCCAAAAAA	
MIN<.f>	a,b,u6	00100bbb01001001FBBBuuuuuuAAAAAA	
MIN<.f>	b,b,s12	00100bbb10001001FBBBssssssSSSSSS	
MIN<.cc><.f>	b,b,c	00100bbb11001001FBBBCCCCCC0QQQQQ	
MIN<.cc><.f>	b,b,u6	00100bbb11001001FBBBuuuuuu1QQQQQ	
MIN<.f>	a,limm,c	00100110000001001F111CCCCCCAAAAAA	L
MIN<.f>	a,b,limm	00100bbb00001001FBBB111110AAAAAA	L
MIN<.cc><.f>	b,b,limm	00100bbb11001001FBBB1111100QQQQQ	L
Without Result			
MIN<.f>	0,b,c	00100bbb00001001FBBBCCCCCC111110	
MIN<.f>	0,b,u6	00100bbb01001001FBBBuuuuuu111110	
MIN<.f>	0,b,limm	00100bbb00001001FBBB111110111110	L
MIN<.cc><.f>	0,limm,c	00100110110001001F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if both source operands are equal
N	●	= Set if most significant bit of result of src1-src2 is set
C	●	= Set if src2 is selected (src2 <= src1)
V	●	= Set if overflow is generated (as a result of src1-src2)

Key:

L = Limm Data

Related Instructions:

MAX

Description:

Return the minimum of the two signed source operands (src1 and src2) and place the result in the destination register (dest).

Pseudo Code Example:

```
if cc==true then                                            /* MIN */
 alu = src1 - src2
 if src2 <= src1 then
  dest = src2
 else
  dest = src1
 if F==1 then
  Z_flag = if alu==0 then 1 else 0
  N_flag = alu[31]
  V_flag = Overflow()
  C_flag = if src2<=src1 then 1 else 0
```

Assembly Code Example:

```
MIN r1,r2,r3           ; Take minimum of r2 and r3
                       ; and write result into r1
```

MOV **Move Contents** **Arithmetic Operation** MOV

Format: inst dest, src **Operation:** if (cc=true) then dest ← src

Format Key:

src	=	Source Operand
dest	=	Destination
cc	=	Condition Code

Syntax:

With Result		**Instruction Code**	
MOV<.f>	b,s12	00100bbb10001010FBBBssssssSSSSSS	
MOV<.cc><.f>	b,c	00100bbb11001010FBBBCCCCCC0QQQQQ	
MOV<.cc><.f>	b,u6	00100bbb11001010FBBBuuuuuu1QQQQQ	
MOV<.cc><.f>	b,limm	00100bbb11001010FBBB1111100QQQQQ	L
MOV_S	b,h	01110bbbhhh01HHH	
MOV_S	b,limm	01110bbb11001111	L
MOV_S	h,b	01110bbbhhh11HHH	
MOV_S	b,u8	11011bbbuuuuuuuu	
Without Result			
MOV<.f>	0,s12	0010011010001010F111ssssssSSSSSS	
MOV<.cc><.f>	0,c	0010011011001010F111CCCCCC0QQQQQ	
MOV<.cc><.f>	0,u6	0010011011001010F111uuuuuu1QQQQQ	
MOV<.cc><.f>	0,limm	0010011011001010F1111111100QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

None

Description:

The contents of the source operand (src) are moved to the destination register (dest).

Pseudo Code Example:

```
if cc==true then /* MOV */                              /* MOV */
 dest = src
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

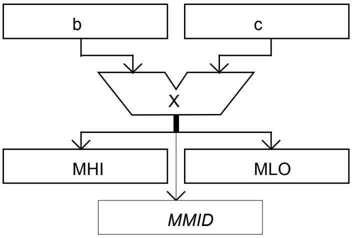
```
MOV r1,r2              ; Move contents of r2 into r1
```

MUL64 — 32 x 32 Signed Multiply Extension Option — MUL64

Format: inst dest, src1, src2

Operation:

MLO ← low part of (src1 * src2)
MHI ← high part of (src1 * src2)
MMID ← middle part of (src1 * src2)



Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

		Instruction Code	
MUL64	<0,>b,c	00101bbb000001000BBBCCCCCC111110	
MUL64	<0,>b,u6	00101bbb010001000BBBuuuuuu111110	
MUL64	<0,>b,s12	00101bbb100001000BBBssssssSSSSSS	
MUL64	<0,>limm,c	00101110000001000111CCCCCC111110	L
MUL64<.cc>	<0,>b,c	00101bbb110001000BBBCCCCCC0QQQQQ	
MUL64<.cc>	<0,>b,u6	00101bbb110001000BBBuuuuuu1QQQQQ	
MUL64<.cc>	<0,>limm,c	00101110110001000111CCCCCC0QQQQQ	L
MUL64<.cc>	<0,>b,limm	00101bbb110001000BBB1111100QQQQQ	L
MUL64_S	<0,>b,c	01111bbbccc01100	

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

MULU64

Description:

Perform a signed 32-bit by 32-bit multiply of operand1 and operand2 then place the most significant 32 bits of the 64-bit result in register MHI, the least significant 32 bits of the 64-bit result in register MLO, and the middle 32 bits of the 64-bit result in register MMID.

Pseudo Code Example:

```
if cc==true then                                                  /* MUL64 */
 mlo = src1 * src2
 mmid = (src1 * src2) >> 16
 mhi = (src1 * src2) >> 32
```

Assembly Code Example:

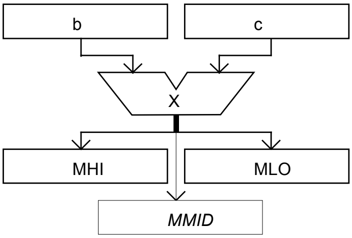
```
MUL64 r2, r3           ; Multiply r2 by r3
                       ; and put the result in the special
                       ; result registers
```

MULU64 — 32 x 32 Unsigned Multiply Extension Option — MULU64

Format: inst dest, src1, src2

Operation:

MLO ← low part of (src1 * src2)
MHI ← high part of (src1 * src2)
MMID ← middle part of (src1 * src2)



Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

		Instruction Code	
MULU64	<0,>b,c	00101bbb000001010BBBCCCCCC111110	
MULU64	<0,>b,u6	00101bbb010001010BBBuuuuuu111110	
MULU64	<0,>b,s12	00101bbb100001010BBBssssssSSSSSS	
MULU64	<0,>limm,c	00101110000001010111CCCCCC111110	L
MULU64<.cc>	<0,>b,c	00101bbb110001010BBBCCCCCC0QQQQQ	
MULU64<.cc>	<0,>b,u6	00101bbb110001010BBBuuuuuu1QQQQQ	
MULU64<.cc>	<0,>limm,c	00101110110001010111CCCCCC0QQQQQ	L
MULU64<.cc>	<0,>b,limm	00101bbb110001010BBB1111100QQQQQ	L

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

MUL64

Description:

Perform an unsigned 32-bit by 32-bit multiply of operand1 and operand2 then place the most significant 32 bits of the 64-bit result in register MHI, the least significant 32 bits of the 64-bit result in register MLO, and the middle 32 bits of the 64-bit result in register MMID.

Pseudo Code Example:

```
if cc==true then                                  /* MULU64 */
 mlo = src1 * src2
 mmid = (src1 * src2) >> 16
 mhi = (src1 * src2) >> 32
```

Assembly Code Example:

```
MULU64 r2, r3           ; Multiply r2 by r3
                        ; and put the result in the special
                        ; result registers
```

NEG_S — Negate — NEG_S

Arithmetic Operation

Format: inst dest, src **Operation:** dest ← 0 - src

Format Key:

src = Source Operand
dest = Destination

Syntax:

		Instruction Code
NEG_S	b,c	01111bbbccc10011

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Related Instructions:

ABS

Description:

The negate instruction subtracts the source operand (src) from zero and places the result into the destination register (dest).

Pseudo Code Example:

```
dest = 0 - src                                  /* NEG */
```

Assembly Code Example:

```
NEG r1,r2               ; Negate r2 and write result
                        ; into r1
```

This page is intentionally left blank.

NOP_S **No Operation** **Control Operation** NOP_S

Format: inst **Operation:** No Operation

Format Key:

inst = Instruction

Syntax:

	Instruction Code
NOP_S	`0111100011100000`

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Related Instructions:

None

Description:

No operation. The state of the processor is not changed by this instruction.

Pseudo Code Example:

```
/* NOP */
```

Assembly Code Example:

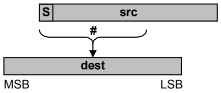
```
NOP                       ; No operation
```

This page is intentionally left blank

NORM **Normalize** NORM

Extension Operation

Format: inst dest, src **Operation:** dest ← normalization integer of src



Format Key:

src = Source Operand
dest = Destination

Syntax:

With Result		**Instruction Code**	
NORM<.f>	b,c	00101bbb00101111FBBBCCCCCC000001	
NORM<.f>	b,u6	00101bbb01101111FBBBuuuuuu000001	
NORM<.f>	b,limm	00101bbb00101111FBBB111110000001	L
Without Result			
NORM<.f>	0,c	0010111000101111F111CCCCCC000001	
NORM<.f>	0,u6	0010111001101111F111uuuuuu000001	
NORM<.f>	0,limm	0010111000101111F111111110000001	L

Flag Affected (32-Bit):

Z	●	= Set if source is zero
N	●	= Set if most significant bit of source is set
C		= Unchanged
V		= Unchanged

Key:

L	= Limm Data

Related Instructions:

EXT
NORMW

SEX

Description:

Gives the normalization integer for the signed value in the operand. The normalization integer is the amount by which the operand should be shifted left to normalize it as a 32-bit signed integer. This function is sometimes referred to as "find first bit". Note that, the returned value for source operand of zero is 0x0000001F. Examples of returned values are shown in the table below:

Operand Value	Returned Value
0x00000000	0x0000001F
0x00000001	0x0000001E
0x1FFFFFFF	0x00000002
0x3FFFFFFF	0x00000001
0x7FFFFFFF	0x00000000
0x80000000	0x00000000
0xC0000000	0x00000001
0xE0000000	0x00000002
0xFFFFFFFF	0x0000001F

Pseudo Code Example:

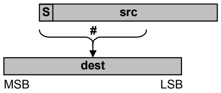
```
dest = NORM(src)                                   /* NORM */
if F==1 then
  Z_flag = if src==0 then 1 else 0
  N_flag = src[31]
```

Assembly Code Example:

```
NORM r1,r2             ; Normalization integer for r2
                       ; write result into r1
```

NORMW **Normalize 16-bit Extension Operation** NORMW

Format: inst dest, src **Operation:** dest ← normalization integer of src



Format Key:

src	=	Source Operand
dest	=	Destination

Syntax:

With Result		**Instruction Code**	
NORMW<.f>	b,c	00101bbb00101111FBBBCCCCCC001000	
NORMW<.f>	b,u6	00101bbb01101111FBBBuuuuuu001000	
NORMW<.f>	b,limm	00101bbb00101111FBBB111110001000	L
Without Result			
NORMW<.f>	0,c	0010111000101111F111CCCCCC001000	
NORMW<.f>	0,u6	0010111001101111F111uuuuuu001000	
NORMW<.f>	0,limm	0010111000101111F111111110001000	L

Flag Affected (32-Bit):

Z	●	= Set if source is zero
N	●	= Set if most significant bit of source is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

EXT SEX

NORM

Description:

Gives the normalization integer for the signed value in the operand. The normalization integer is the amount by which the operand should be shifted left to normalize it as a 16-bit signed integer. When normalizing a 16-bit signed integer the lower 16 bits of the source data (src) is used. This function is sometimes referred to as "find first bit". Note that, the returned value for source operand of zero is 0x000F. Examples of returned values are shown in the table below:

Operand Value	Returned Value
0x0000	0x000F
0x0001	0x000E
0x1FFF	0x0002
0x3FFF	0x0001
0x7FFF	0x0000
0x8000	0x0000
0xC000	0x0001
0xE000	0x0002
0xFFFF	0x000F

Pseudo Code Example:

```
dest = NORMW(src)                                    /* NORMW */
if F==1 then
  Z_flag = if (src & 0x0000FFFF)==0 then 1 else 0
  N_flag = src[15]
```

Assembly Code Example:

```
NORMW r1,r2            ; Normalization integer for r2
                       ; write result into r1
```

9
Instruction Set Details

NOT Logical Bitwise NOT NOT

Logical Operation

Format: inst dest, src **Operation:** dest ← NOT(src)

Format Key:

src	=	Source Operand
dest	=	Destination
NOT	=	Negate Source

Syntax:

With Result		**Instruction Code**	
NOT<.f>	b,c	00100bbb00101111FBBBCCCCCC001010	
NOT<.f>	b,u6	00100bbb01101111FBBBuuuuuu001010	
NOT<.f>	b,limm	00100bbb00101111FBBB111110001010	L
NOT_S	b,c	01111bbbccc10010	
Without Result			
NOT<.f>	0,c	0010011000101111F111CCCCCC001010	
NOT<.f>	0,u6	0010011001101111F111uuuuuu001010	
NOT<.f>	0,limm	0010011000101111F111111110001010	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

ABS

Description:

Logical bitwise NOT (inversion) of the source operand (src) with the result placed into the destination register (dest).

Pseudo Code Example:

```
dest = NOT(src)                                    /* NOT */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
NOT r1,r2              ; Logigal bitwise NOT r2 and
                       ; write result into r1
```

This page is intentionally left blank.

OR — Logical Bitwise OR — OR

Logical Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← (src1 OR src2)

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition code
OR	=	Logical Bitwise OR

Syntax:

With Result		**Instruction Code**	
OR<.f>	a,b,c	`00100bbb00000101FBBBCCCCCCAAAAAA`	
OR<.f>	a,b,u6	`00100bbb01000101FBBBuuuuuuAAAAAA`	
OR<.f>	b,b,s12	`00100bbb10000101FBBBssssssSSSSSS`	
OR<.cc><.f>	b,b,c	`00100bbb11000101FBBBCCCCCC0QQQQQ`	
OR<.cc><.f>	b,b,u6	`00100bbb11000101FBBBuuuuuu1QQQQQ`	
OR<.f>	a,limm,c	`00100110000001 01F111CCCCCCAAAAAA`	L
OR<.f>	a,b,limm	`00100bbb00000101FBBB111110AAAAAA`	L
OR<.cc><.f>	b,b,limm	`00100bbb11000101FBBB1111100QQQQQ`	L
OR_S	b,b,c	`01111bbbccc00101`	
Without Result			
OR<.f>	0,b,c	`00100bbb00000101FBBBCCCCCC111110`	
OR<.f>	0,b,u6	`00100bbb01000101FBBBuuuuuu111110`	
OR<.f>	0,b,limm	`00100bbb00000101FBBB111110111110`	L
OR<.cc><.f>	0,limm,c	`0010011011000101F111CCCCCC0QQQQQ`	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

AND BIC
XOR

Description:

Logical bitwise OR of source operand 1 (src1) with source operand 2 (src2). The result is written into the destination register (dest).

Pseudo Code Example:

```
if cc==true then                                          /* OR */
 dest = src1 OR src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
OR r1,r2,r3             ; Logical bitwise OR contents
                        ; of r2 with r3 and write
                        ; result into r1
```

POP_S **Pop from Stack** **Memory Operation** POP_S

Format: inst dest **Operation:** dest ←Result of Memory Load from Address [sp] then sp ← sp+4

Format Key:

dest	=	Destination Register
sp	=	Stack Pointer (r28)

Syntax:

		Instruction Code
POP_S	b	11000bbb11000001
POP_S	blink	11000RRR11010001

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

PUSH_S

Description:

Perform a long word memory load from the long word aligned address specified in the implicit Stack Pointer (r28) and place the result into the destination register (dest). Subsequently the implicit stack pointer is automatically incremented by 4-bytes (sp=sp+4).

Pseudo Code Example:

```
dest = Memory(SP, 4)                                /* POP */
SP = SP + 4
```

Assembly Code Example:

```
POP r1                  ; Load long word from memory
                        ; at address SP and write
                        ; result to r1 and then add 4
                        ; to SP
```

NOTE The LP_COUNT register should not be used as the destination of a POP_S instruction.

Thispage is intentionally left blank

PUSH_S **Push onto Stack** PUSH_S

Memory Operation

Format: inst src **Operation:** sp ← sp-4 then Memory Write Address [sp] ← src

Format Key:

src	=	Source Operand
sp	=	Stack Pointer (r28)

Syntax:

		Instruction Code
PUSH_S	b	`11000bbb11100001`
PUSH_S	blink	`11000RRR11110001`

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Related Instructions:

POP_S

Description:

Decrement 4-bytes from the implicit stack pointer address found in r28 and perform a long word memory write to that address with the data specified in the source operand (src).

Pseudo Code Example:

```
SP = SP - 4                                          /* PUSH */
Memory(SP, 4) = src
```

Assembly Code Example:

```
PUSH r1                 ; Subtract 4 from SP and then
                        ; store long word from r1
                        ; to memory at address SP
```

This page is intentionally left blank

RCMP **Reverse Comparison Arithmetic Operation** RCMP

Format: inst src1, src2 **Operation:** if (cc=true) then src2 – src1

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition Code

Syntax:

		Instruction Code	
RCMP	b,s12	00100bbb1000110111BBBssssssSSSSSS	
RCMP<.cc>	b,c	00100bbb1100110111BBBCCCCCC0QQQQQ	
RCMP<.cc>	b,u6	00100bbb1100110111BBBuuuuuu1QQQQQ	
RCMP<.cc>	b,limm	00100bbb1100110111BBB1111100QQQQQ	L
RCMP<.cc>	limm,c	001001101100110111111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if overflow is generated

Key:

L = Limm Data

Related Instructions:

CMP

Description:

A reverse comparison is performed by subtracting source operand 1 (src1) from source operand 2 (src2) and subsequently updating the flags.

There is no destination register therefore the result of the subtract is discarded.

NOTE RCMP always sets the flags even thought there is no associated flag setting suffix .

Pseudo Code Example:

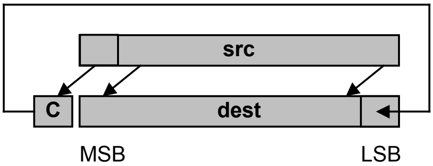
```
if cc==true then                                                /* RCMP */
 alu = src2 - src1
 Z_flag = if alu==0 then 1 else 0
 N_flag = alu[31]
 C_flag = Carry()
 V_flag = Overflow()
```

Assembly Code Example:

```
RCMP r1,r2                ; Subtract r1 from r2
                          ; and set the flags on the
                          ; result
```

RLC Rotate Left Through Carry Logical Operation RLC

Format: inst dest, src **Operation:** dest ← RLC by 1 (src)



Format Key:

src	=	Source Operand
dest	=	Destination
cc	=	Condition Code
RLC	=	Rotate Source Operand Left Through Carry by 1

Syntax:

With Result		**Instruction Code**	
RLC<.f>	b,c	00100bbb00101111FBBBCCCCCC001011	
RLC<.f>	b,u6	00100bbb01101111FBBBuuuuuu001011	
RLC<.f>	b,limm	00100bbb00101111FBBB111110001011	L
Without Result			
RLC<.f>	0,c	0010011000101111F111CCCCCC001011	
RLC<.f>	0,u6	0010011001101111F111uuuuuu001011	
RLC<.f>	0,limm	0010011000101111F111111110001011	L

Flag Affected (32-Bit):

Z	•	= Set if result is zero
N	•	= Set if most significant bit of result is set
C	•	= Set if carry is generated
V		= Undefined

Key:

L	= Limm Data

Related Instructions:

RRC ROR

Description:

Rotate the source operand (src) left by one and place the result in the destination register (dest).

The carry flag is shifted into the least significant bit of the result, and the most significant bit of the source is placed in the carry flag.

Pseudo Code Example:

```
dest = src << 1                                            /* RLC */
dest[0] = C_flag
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 C_flag = src[31]
 V_flag = UNDEFINED
```

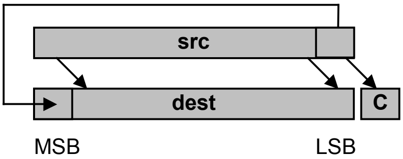
Assembly Code Example:

```
RLC r1,r2              ; Rotate left through carry
                       ; contents of r2 by one bit
                       ; and write result into r1
```

ROR — Rotate Right — ROR

Logical Operation

Format: inst dest, src **Operation:** dest ← ROR by 1 (src)



Format Key:

src	=	Source Operand
dest	=	Destination
cc	=	Condition Code
ROR	=	Rotate Source Operand Right by 1

Syntax:

With Result		**Instruction Code**	
ROR<.f>	b,c	00100bbb00101111FBBBCCCCCC000011	
ROR<.f>	b,u6	00100bbb01101111FBBBuuuuuu000011	
ROR<.f>	b,limm	00100bbb00101111FBBB111110000011	L
Without Result			
ROR<.f>	0,c	00100110001011111F111CCCCCC000011	
ROR<.f>	0,u6	00100110011011111F111uuuuuu000011	
ROR<.f>	0,limm	00100110001011111F111111110000011	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V		= Unchanged

Key:

L	= Limm Data

Related Instructions:

RRC RLC

Description:

Rotate the source operand (src) right by one and place the result in the destination register (dest).

The least significant bit of the source operand is copied to the carry flag.

Pseudo Code Example:

```
dest = src >> 1                                      /* ROR */
dest[31] = src[0]
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 C_flag = src[0]
```

Assembly Code Example:

```
ROR r1,r2               ; Rotate right contents of r2
                        ; by one bit and write result
                        ; into r1
```

ROR multiple

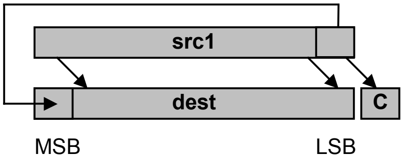
Multiple Rotate Right

Logical Operation

ROR multiple

Format: inst dest, src1, src2

Operation: if (cc=true) then dest ← rotate right of src1 by src2



Format Key:

dest = Destination Register
src1 = Source Operand 1
src2 = Source Operand 2

Syntax:

With Result		**Instruction Code**	
ROR<.f>	a,b,c	00101bbb00000011FBBBCCCCCCAAAAAA	
ROR<.f>	a,b,u6	00101bbb01000011FBBBuuuuuuAAAAAA	
ROR<.f>	b,b,s12	00101bbb10000011FBBBssssssSSSSSS	
ROR<.cc><.f>	b,b,c	00101bbb11000011FBBBCCCCCC0QQQQQ	
ROR<.cc><.f>	b,b,u6	00101bbb11000011FBBBuuuuuu1QQQQQ	
ROR<.f>	a,limm,c	0010111000000011F111CCCCCCAAAAAA	L
ROR<.f>	a,b,limm	00101bbb00000011FBBB111110AAAAAA	L
ROR<.cc><.f>	b,b,limm	00101bbb11000011FBBB1111100QQQQQ	L
Without Result			
ROR<.f>	0,b,c	00101bbb00000011FBBBCCCCCC111110	
ROR<.f>	0,b,u6	00101bbb01000011FBBBuuuuuu111110	
ROR<.cc><.f>	0,limm,c	0010111011000011F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

ASR
RLC
LSR
RRC

Description:

Rotate right src1 by src2 places and place the result in the destination register. Only the bottom 5 bits of src2 are used as the shift value.

Pseudo Code Example:

```
if cc=true then                                             /* ROR */
 dest = src1 >> (src2 & 31)                                 /* Multiple */
 dest [31:(31-src2)] = src1 [(src2-1):0]
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = if src2==0 then 0 else src1[src2-1]
```

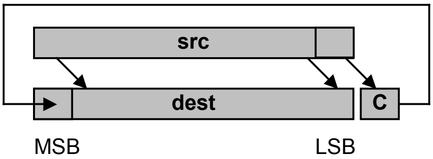
Assembly Code Example:

```
ROR r1,r2,r3           ; Rotate right
                       ; contents of r2 by r3 bits
                       ; and write result into r1
```

RRC Rotate Right through Carry RRC

Logical Operation

Format: inst dest, src **Operation:** dest ← RRC by 1 (src)



Format Key:

src	=	Source Operand
dest	=	Destination
cc	=	Condition Code
RRC	=	Rotate Source Operand Right Through Carry by 1

Syntax:

With Result		**Instruction Code**	
RRC<.f>	b,c	00100bbb00101111FBBBCCCCCC000100	
RRC<.f>	b,u6	00100bbb01101111FBBBuuuuuu000100	
RRC<.f>	b,limm	00100bbb00101111FBBB111110000100	L
Without Result			
RRC<.f>	0,c	0010011000101111F111CCCCCC000100	
RRC<.f>	0,u6	0010011001101111F111uuuuuu000100	
RRC<.f>	0,limm	0010011000101111F111111110000100	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V		= Unchanged

Key:

L	= Limm Data

Related Instructions:

ROR RLC

Description:

Rotate the source operand (src) right by one and place the result in the destination register (dest).

The carry flag is shifted into the most significant bit of the result, and the most significant bit of the source is placed in the carry flag.

Pseudo Code Example:

```
dest = src >> 1                                              /* RRC */
dest[31] = C_flag
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 C_flag = src[0]
```

Assembly Code Example:

```
RRC r1,r2              ; Rotate right through carry
                       ; contents of r2 by one bit
                       ; and write result into r1
```

RSUB **Reverse Subtract** RSUB

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src2 – src1

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition Code

Syntax:

With Result		**Instruction Code**	
RSUB<.f>	a,b,c	`00100bbb00001110FBBBCCCCCCAAAAAA`	
RSUB<.f>	a,b,u6	`00100bbb01001110FBBBuuuuuuAAAAAA`	
RSUB<.f>	b,b,s12	`00100bbb10001110FBBBssssssSSSSSS`	
RSUB<.cc><.f>	b,b,c	`00100bbb11001110FBBBCCCCCC0QQQQQ`	
RSUB<.cc><.f>	b,b,u6	`00100bbb11001110FBBBuuuuuu1QQQQQ`	
RSUB<.f>	a,limm,c	`0010011000001110F111CCCCCCAAAAAA`	L
RSUB<.f>	a,b,limm	`00100bbb00001110FBBB111110AAAAAA`	L
RSUB<.cc><.f>	b,b,limm	`00100bbb11001110FBBB1111100QQQQQ`	L
Without Result			
RSUB<.f>	0,b,c	`00100bbb00001110FBBBCCCCCC111110`	
RSUB<.f>	0,b,u6	`00100bbb01001110FBBBuuuuuu111110`	
RSUB<.f>	0,b,limm	`00100bbb00001110FBBB111110111110`	L
RSUB<.cc><.f>	0,limm,c	`0010011011001110F111CCCCCC0QQQQQ`	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if overflow is generated

Key:

L	= Limm Data

Related Instructions:

SUB
SUB1
SUB2
SUB3
SUB3
SBC

Description:

Subtract source operand 1 (src1) from source operand 2 (src2) and place the result in the destination register.

If the carry flag is set upon performing the subtract, the carry flag should be interpreted as a ‘borrow’.

Pseudo Code Example:

```
if cc==true then                                   /* RSUB */
 dest = src2 - src1
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example:

```
RSUB r1,r2,r3          ; Subtract contents of r2 from
                       ; r3 and write result into r1
```

SBC Subtract with Carry SBC

Arithmetic Operation

Format: inst dest, src1, src2

Operation: if (cc=true) then dest ← (src1 – src2) - C

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition Code
C	=	Carry Flag Value

Syntax:

With Result		**Instruction Code**	
SBC<.f>	a,b,c	`00100bbb00000011FBBBCCCCCCAAAAAA`	
SBC<.f>	a,b,u6	`00100bbb01000011FBBBuuuuuuAAAAAA`	
SBC<.f>	b,b,s12	`00100bbb10000011FBBBssssssSSSSSS`	
SBC<.cc><.f>	b,b,c	`00100bbb11000011FBBBCCCCCC0QQQQQ`	
SBC<.cc><.f>	b,b,u6	`00100bbb11000011FBBBuuuuuu1QQQQQ`	
SBC<.f>	a,limm,c	`0010011000000011F111CCCCCCAAAAAA`	L
SBC<.f>	a,b,limm	`00100bbb00000011FBBB111110AAAAAA`	L
SBC<.cc><.f>	b,b,limm	`00100bbb11000011FBBB1111100QQQQQ`	L
Without Result			
SBC<.f>	0,b,c	`00100bbb00000011FBBBCCCCCC111110`	
SBC<.f>	0,b,u6	`00100bbb01000011FBBBuuuuuu111110`	
SBC<.f>	0,b,limm	`00100bbb00000011FBBB111110111110`	L
SBC<.cc><.f>	0,limm,c	`0010011011000011F111CCCCCC0QQQQQ`	L

Flag Affected (32-Bit):

Z	•	= Set if result is zero
N	•	= Set if most significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if overflow is generated

Key:

L = Limm Data

Related Instructions:

SUB
SUB1
SUB2
RSUB
SUB3

Description:

Subtract source operand 2 (src2) from source operand 1 (src1) and also subtract the state of the carry flag (if set then subtract ‘1’, otherwise subtract ‘0’). Place the result in the destination register.

If the carry flag is set upon performing the subtract, the carry flag should be interpreted as a ‘borrow’.

Pseudo Code Example:

```
if cc==true then                                    /* SBC */
 dest = (src1 - src2) - C_flag
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example:

```
SBC r1,r2,r3           ; Subtract with carry contents
                       ; of r3 from r2 and write
                       ; result into r1
```

SEXB Sign Extend Byte SEXB

Arithmetic Operation

Format: inst dest, src **Operation:** dest ← SEXB(src)

Format Key:

src	=	Source Operand
dest	=	Destination
cc	=	Condition Code
SEXB	=	Sign Extend Byte

Syntax:

With Result		**Instruction Code**	
SEXB<.f>	b,c	00100bbb00101111FBBBCCCCCC000101	
SEXB<.f>	b,u6	00100bbb01101111FBBBuuuuuu000101	
SEXB<.f>	b,limm	00100bbb00101111FBBB111110000101	L
SEXB_S	b,c	01111bbbccc01101	
Without Result			
SEXB<.f>	0,c	0010011000101111F111CCCCCC000101	
SEXB<.f>	0,u6	0010011001101111F111uuuuuu000101	
SEXB<.f>	0,limm	0010011000101111F111111110000101	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

SEXW

Description:

Sign extend the byte contained in the source operand (src) to the most significant bit in a long word and place the result into the destination register (dest).

Pseudo Code Example:

```
dest = src & 0xFF                                  /* SEXB */
dest[31:8] = src[7]
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
SEXB r1,r2              ; Sign extend the bottom 8
                        ; bits of r2 and write
                        ; result to r1
```

SEXW Sign Extend Word Arithmetic Operation SEXW

Format: inst dest, src **Operation:** dest ← SEXW(src)

Format Key:

src	=	Source Operand
dest	=	Destination
cc	=	Condition Code
SEXW	=	Sign Extend Word

Syntax:

With Result		**Instruction Code**	
SEXW<.f>	b,c	`00100bbb00101111FBBBCCCCCC000110`	
SEXW<.f>	b,u6	`00100bbb01101111FBBBuuuuuu000110`	
SEXW<.f>	b,limm	`00100bbb00101111FBBB111110000110`	L
SEXW_S	b,c	`01111bbbccc01110`	
Without Result			
SEXW<.f>	0,c	`0010011000101111F111CCCCCC000110`	
SEXW<.f>	0,u6	`0010011001101111F111uuuuuu000110`	
SEXW<.f>	0,limm	`0010011000101111F111111110000110`	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L	= Limm Data

Related Instructions:

SEXB

Description:

Sign extend the word contained in the source operand (src) to the most significant bit in a long word and place the result into the destination register (dest).

Pseudo Code Example:

```
dest = src & 0xFFFF                                      /* SEXW */
dest[31:16] = src[15]
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
SEXW r1,r2             ; Sign extend the bottom 16
                       ; bits of r2 and write
                       ; result to r1
```

SLEEP — Enter Sleep Mode — SLEEP

Control Operation

Format: inst **Operation:** Enter Processor Sleep Mode

Format Key:

inst = Instruction

Syntax:

	Instruction Code
SLEEP	`00100001011011110000000000111111`

Flag Affected (32-Bit):

Flag		
Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged
ZZ	●	= 1

Key:

L = Limm Data

Related Instructions:

None

Description:

The SLEEP instruction is a single operand instruction without flags or operands. The SLEEP instruction is decoded in pipeline stage 2. If a SLEEP instruction is detected, then the sleep mode flag found in the debug register (auxiliary address 0x05, flag ZZ) is immediately set and the pipeline stage 1 is stalled. A flushing mechanism ensures that all earlier instructions are executed until the pipeline is empty. The SLEEP instruction itself is also flushed from the pipeline.

NOTE If the H flag is set by the FLAG instruction (`FLAG 1`), three sequential NOP instructions should immediately follow. This means that SLEEP should not immediately follow a `FLAG 1` instruction, but should be separated by 3 NOP instructions.

When in sleep mode, the sleep mode flag (ZZ) is set and the pipeline is stalled, but not halted. The host interface operates as normal allowing access to the DEBUG and the STATUS registers and also has the ability to halt the processor. The host cannot clear the sleep mode flag, but it can wake the processor by halting it then restarting it. The program counter PC points to the next instruction in sequence after the sleep instruction.

CAUTION The SLEEP instruction cannot immediately follow a BRcc or BBITn instruction.

The SLEEP instruction can be used in RTOS type applications by using a FLAG 0x06 followed by a SLEEP instruction. This allows interrupts to be re-enabled at the same time as SLEEP is entered.

NOTE The FLAG followed by SLEEP instruction sequence must not encounter an instruction-cache miss. This can be accomplished by ensuring that the FLAG is aligned to the instruction-cache line length.

The processor will wake from sleep mode on an interrupt or when it is restarted. If an interrupt wakes it, the ZZ flag is cleared and the instruction in pipeline stage 1 is killed. The interrupt routine is serviced and execution resumes at the instruction in sequence after the SLEEP instruction. When it is started after having been halted the ZZ flag is cleared.

SLEEP behaves as a NOP during single step mode.

Pseudo Code Example:

```
FlushPipe()                                         /* SLEEP */
DEBUG[ZZ] = 1
WaitForInterrupt()
DEBUG[ZZ] = 0
ServiceInterrupt()
```

Assembly Code Example:

```
.equ  EI,0x06          ; Constant to enable both interrupt levels
.align  8              ; ensure cache alignment is to 8 bytes
FLAG   EI              ; Enable interrupts
SLEEP                  ; put processor into sleep mode
```

SR — Store to Auxiliary Register — SR

Control Operation

Format: inst src1, src2 **Operation:** aux_reg(src2) = src1

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

		Instruction Code	
SR	b,[c]	`00100bbb001010110BBBCCCCCCRRRRRR`	
SR	b,[limm]	`00100bbb001010110BBB111110RRRRRR`	L
SR	b,[s12]	`00100bbb101010110BBBssssssSSSSSS`	
SR	limm,[c]	`00100110001010110111CCCCCCRRRRRR`	L
SR	limm,[s12]	`00100110101010110111ssssssSSSSSS`	L

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

LR

Description:

Store the data that is held in source operand 1 (src1) into the auxiliary register whose number is obtained from the source operand 2 (src2).

Pseudo Code Example:

```
Aux_reg(src2) = src1                                /* SR */
```

Assembly Code Example:

```
SR r1,[r2]             ; Store contents of r1 into
                       ; Aux. register r2
```

ST **Store to Memory** ST

Memory Operation

Format: inst src1, src2, src3 **Operation:** Memory Store Address @ (src2+src3) ← src1

Format Key:

src1 = Source Operand 1
src2 = Source Operand 2
src3 = Source Operand 3 (Offset)

Syntax:

		Instruction Code	
ST<zz><.aa><.di>	c,[b,s9]	00011bbbssssssssSBBBCCCCCCDaaZZR	
ST<zz><.di>	c,[limm]	00011110000000000111CCCCCCDRRZZR	L
ST<zz><.aa><.di>	limm,[b,s9]	00011bbbssssssssSBBB111110DaaZZR	L
ST_S	c,[b,u7]	10100bbbcccuuuuu	
STB_S	c,[b,u5]	10101bbbcccuuuuu	
STW_S	c,[b,u6]	10110bbbcccuuuuu	
ST_S	b,[sp,u7]	11000bbb010uuuuu	
STB_S	b,[sp,u7]	11000bbb011uuuuu	

Data Size Field <zz>:

Data Size Syntax	ZZ Field	Description
No Field Syntax	00	Data is a long-word (32-Bits) *(<.x> syntax illegal*)
W	10	Data is a word (16-Bits)
B	01	Data is a byte (8-Bits)
	11	*reserved*

Data Cache Mode <.di>:

D Flag	Description
0	Cached data memory access (*default, if no <.di> field syntax*)
1	Non-cached data memory access (*bypass data cache*)

Address Write-back Mode <.aa>:

Address Write-back Syntax	aa Field	Effective Address	Address Write-Back
No Field Syntax	00	Address = src2+src3 (*register+offset*)	None
.A or .AW	01	Address = src2+src3 (*register+offset*)	src2 ← src2+src3 (*register+offset*)
.AB	10	Address = src2 (*register*)	src2 ← src2+src3 (*register+offset*)

.AS	11	Address = src2+(src3<<1) (<*zz*>= *'10'*)
		Address = src2+(src3<<2) (<*zz*>= *'00'*)
			None. *Using a byte data size is invalid and is a reserved format*

16-Bit Store Instructions Operation:

Instruction	Format	Operation	Description
ST_S	c, [b,u7]	address[src2+u7].l ← src1	Store long word to address calculated by register + unsigned immediate
STB_S	c, [b,u5]	address[src2+u5].b ← src1	Store unsigned byte to address calculated by register + unsigned immediate
STW_S	c, [b,u6]	address[src2+u6].w ← src1	Store unsigned word to address calculated by register + unsigned immediate
ST_S	b, [sp,u7]	address[sp+u7].l ← src1	Store long word to address calculated by Stack Pointer (r28) + unsigned immediate
STB_S	b, [sp,u7]	address[sp+u7].b ← src1	Store unsigned byte to address calculated by Stack Pointer (r28) + unsigned immediate

Related Instructions:

LD
LR
SR

Description:

Data that is held in source operand 1 (src1) is stored to a memory address that is calculated by adding source operand 2 (src2) with an offset specified by source operand 3 (scr3).

The size of the data written is specified by the data size field <zz> (32-bit instructions).

NOTE When a memory controller is employed: Store bytes can be made to any byte alignments, Store words should be made from word aligned addresses and Store longs should be made only from long aligned addresses.

If the processor contains a data cache, store requests can bypass the cache by using the <.di> syntax.

NOTE For the 16-bit encoded instructions the u offset is aligned accordingly. For example ST_S c, [b. u7] only needs to encode the top 5 bits since the bottom 2 bits of u7 are always zero because of the 32-bit data alignment.

The address write-back mode can be selected by use of the <.aa> syntax.

NOTE When using the scaled source addressing mode (.AS), the scale factor is dependent upon the size of the data word requested (zz).

Pseudo Code Example:

```
if AA==0 then address = src2 + src3                /* ST */
if AA==1 then address = src2 + src3
if AA==2 then address = src2
if AA==3 and ZZ==0 then
 address = src2 + (src3 << 2)
if AA==3 and ZZ==2 then
 address = src2 + (src3 << 1)
Memory(address, size) = src1
if AA==1 or AA==2 then
 src2 = src2 + src3
```

Assembly Code Example:

```
ST r0,[r1,4]          ; Store long word value of
                      ; r0 to memory address
                      ; r1+4
```

This page is intentionally left blank.

SUB **Subtract** SUB

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 – src2

Format Key:

dest = Destination Register
src1 = Source Operand 1
src2 = Source Operand 2
cc = Condition Code

Syntax:

With Result		**Instruction Code**	
SUB<.f>	a,b,c	00100bbb00000010FBBBCCCCCCAAAAAA	
SUB<.f>	a,b,u6	00100bbb01000010FBBBuuuuuuAAAAAA	
SUB<.f>	b,b,s12	00100bbb10000010FBBBssssssSSSSSS	
SUB<.cc><.f>	b,b,c	00100bbb11000010FBBBCCCCCC0QQQQQ	
SUB<.cc><.f>	b,b,u6	00100bbb11000010FBBBuuuuuu1QQQQQ	
SUB<.f>	a,limm,c	00100110000000010F111CCCCCCAAAAAA	L
SUB<.f>	a,b,limm	00100bbb00000010FBBB111110AAAAAA	L
SUB<.cc><.f>	b,b,limm	00100bbb11000010FBBB1111100QQQQQ	L
SUB_S	c,b,u3	01101bbbccc01uuu	
SUB_S	b,b,c	01111bbbccc00010	
SUB_S	b,b,u5	10111bbb011uuuuu	
SUB_S.NE	b,b,b	01111bbb11000000	
SUB_S	sp,sp,u7	11000001101uuuuu	
Without Result			
SUB <.f>	0,b,c	00100bbb00000010FBBBCCCCCC111110	
SUB <.f>	0,b,u6	00100bbb01000010FBBBuuuuuu111110	
SUB <.f>	0,b,limm	00100bbb00000010FBBB111110111110	L
SUB <.cc><.f>	0,limm,c	00100110110000010F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z ● = Set if result is zero
N ● = Set if most significant bit of result is set
C ● = Set if carry is generated
V ● = Set if overflow is generated

Key:

L = Limm Data

Related Instructions:

RSUB	SUB2	SBC
SUB1	SUB3	

Description:

Subtract source operand 2 (src2) from source operand 1 (src1) and place the result in the destination register.

SUB_S.NE is a conditional instruction used to clear a register, and is executed when the Z flag is equal to zero.

If the carry flag is set upon performing the subtract, the carry flag should be interpreted as a ‘borrow’.

NOTE For the 16-bit encoded instructions that work on the stack pointer (SP) the offset is aligned to 32-bit. For example SUB_S sp, sp. u7 only needs to encode the top 5 bits since the bottom 2 bits of u7 are always zero because of the 32-bit data alignment.

Pseudo Code Example:

```
if cc==true then                                    /* SUB */
 dest = src1 - src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example:

```
SUB r1,r2,r3          ; Subtract contents of r3 from
                      ; r2 and write result into r1
```

SUB1 **Subtract with Scaled Source** SUB1

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 – (src2 << 1)

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition Code

Syntax:

With Result		**Instruction Code**	
SUB1<.f>	a,b,c	00100bbb00010111FBBBCCCCCCAAAAAA	
SUB1<.f>	a,b,u6	00100bbb01010111FBBBuuuuuuAAAAAA	
SUB1<.f>	b,b,s12	00100bbb10010111FBBBssssssSSSSSS	
SUB1<.cc><.f>	b,b,c	00100bbb11010111FBBBCCCCCC0QQQQQ	
SUB1<.cc><.f>	b,b,u6	00100bbb11010111FBBBuuuuuu1QQQQQ	
SUB1<.f>	a,limm,c	0010011000010111F111CCCCCCAAAAAA	L
SUB1<.f>	a,b,limm	00100bbb00010111FBBB111110AAAAAA	L
SUB1<.cc><.f>	b,b,limm	00100bbb11010111FBBB1111100QQQQQ	L
Without Result			
SUB1<.f>	0,b,c	00100bbb00010111FBBBCCCCCC111110	
SUB1<.f>	0,b,u6	00100bbb01010111FBBBuuuuuu111110	
SUB1<.f>	0,b,limm	00100bbb00010111FBBB111110111110	L
SUB1<.cc><.f>	0,limm,c	0010011011010111F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if overflow is generated from the SUB part of the instruction

Key:

L	= Limm Data

Related Instructions:

RSUB

SUB

SUB2

SUB3

SBC

Description:

Subtract a scaled version of source operand 2 (src2) (src2 left shifted by 1) from source operand 1 (src1) and place the result in the destination register.

If the carry flag is set upon performing the subtract, the carry flag should be interpreted as a ‘borrow’.

Pseudo Code Example:

```
if cc==true then                                              /* SUB1 */
 dest = src1 - (src2 << 1)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example:

```
SUB1 r1,r2,r3          ; Subtract contents of r3 left
                       ; shifted one bit from r2
                       ; and write result into r1
```

SUB2 Subtract with Scaled Source SUB2

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 – (src2 << 2)

Format Key:

dest = Destination Register
src1 = Source Operand 1
src2 = Source Operand 2
cc = Condition Code

Syntax:

With Result		**Instruction Code**	
SUB2<.f>	a,b,c	00100bbb00011000FBBBCCCCCCAAAAAA	
SUB2<.f>	a,b,u6	00100bbb01011000FBBBuuuuuuAAAAAA	
SUB2<.f>	b,b,s12	00100bbb10011000FBBBssssssSSSSSS	
SUB2<.cc><.f>	b,b,c	00100bbb11011000FBBBCCCCCC0QQQQQ	
SUB2<.cc><.f>	b,b,u6	00100bbb11011000FBBBuuuuuu1QQQQQ	
SUB2<.f>	a,limm,c	0010011000011000F111CCCCCCAAAAAA	L
SUB2<.f>	a,b,limm	00100bbb00011000FBBB111110AAAAAA	L
SUB2<.cc><.f>	b,b,limm	00100bbb11011000FBBB1111100QQQQQ	L
Without Result			
SUB2<.f>	0,b,c	00100bbb00011000FBBBCCCCCC111110	
SUB2<.f>	0,b,u6	00100bbb01011000FBBBuuuuuu111110	
SUB2<.f>	0,b,limm	00100bbb00011000FBBB111110111110	L
SUB2<.cc><.f>	0,limm,c	0010011011011000F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	•	= Set if result is zero
N	•	= Set if most significant bit of result is set
C	•	= Set if carry is generated
V	•	= Set if overflow is generated from the SUB part of the instruction

Key:

L = Limm Data

Related Instructions:

RSUB
SUB1
SUB

SUB3
SBC

Description:

Subtract a scaled version of source operand 2 (src2) (src2 left shifted by 2) from source operand 1 (src1) and place the result in the destination register.

If the carry flag is set upon performing the subtract, the carry flag should be interpreted as a ‘borrow’.

Pseudo Code Example:

```
if cc==true then                                            /* SUB2 */
 dest = src1 - (src2 << 2)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example:

```
SUB2 r1,r2,r3          ; Subtract contents of r3 left
                       ; shifted two bits from r2
                       ; and write result into r1
```

SUB3 — Subtract with Scaled Source — SUB3

Arithmetic Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 – (src2 << 3)

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition Code

Syntax:

With Result		**Instruction Code**	
SUB3<.f>	a,b,c	`00100bbb00011001FBBBCCCCCCAAAAAA`	
SUB3<.f>	a,b,u6	`00100bbb01011001FBBBuuuuuuAAAAAA`	
SUB3<.f>	b,b,s12	`00100bbb10011001FBBBssssssSSSSSS`	
SUB3<.cc><.f>	b,b,c	`00100bbb11011001FBBBCCCCCC0QQQQQ`	
SUB3<.cc><.f>	b,b,u6	`00100bbb11011001FBBBuuuuuu1QQQQQ`	
SUB3<.f>	a,limm,c	`0010011000011001F111CCCCCCAAAAAA`	L
SUB3<.f>	a,b,limm	`00100bbb00011001FBBB111110AAAAAA`	L
SUB3<.cc><.f>	b,b,limm	`00100bbb11011001FBBB1111100QQQQQ`	L
Without Result			
SUB3<.f>	0,b,c	`00100bbb00011001FBBBCCCCCC111110`	
SUB3<.f>	0,b,u6	`00100bbb01011001FBBBuuuuuu111110`	
SUB3<.f>	0,limm,c	`0010011000011001F111CCCCCC111110`	L
SUB3<.cc><.f>	0,limm,c	`0010011011011001F111CCCCCC0QQQQQ`	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	●	= Set if carry is generated
V	●	= Set if overflow is generated from the SUB part of the instruction

Key:

L = Limm Data

Related Instructions:

RSUB	SUB
SUB1	SBC
SUB2	

Description:

Subtract a scaled version of source operand 2 (src2) (src2 left shifted by 3) from source operand 1 (src1) and place the result in the destination register.

If the carry flag is set upon performing the subtract, the carry flag should be interpreted as a ‘borrow’.

Pseudo Code Example:

```
if cc==true then                                            /* SUB3 */
 dest = src1 - (src2 << 3)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = Carry()
  V_flag = Overflow()
```

Assembly Code Example:

```
SUB3 r1,r2,r3           ; Subtract contents of r3 left
                        ; shifted three bits from r2
                        ; and write result into r1
```

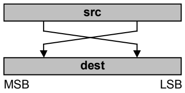
SWAP

Swap words

Extension: Swap instruction

SWAP

Format: inst dest, src **Operation:** dest ← word swap of src



Format Key:

src	=	Source Operand
dest	=	Destination

Syntax:

With Result			
SWAP<.f>	b,c	00101bbb00101111FBBBCCCCCC000000	
SWAP<.f>	b,u6	00101bbb01101111FBBBuuuuuu000000	
SWAP<.f>	b,limm	00101bbb00101111FBBB111110000000	L
Without Result			
SWAP<.f>	0,c	0010111000101111F111CCCCCC000000	
SWAP<.f>	0,u6	0010111001101111F111uuuuuu000000	
SWAP<.f>	0,limm	0010111000101111F111111110000000	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

Description:

Swap the lower 16 bits of the operand with the upper 16 bits of the operand and place the result of that swap in the destination register.

Pseudo Code Example:

```
dest = SWAP(src)                                    /* SWAP */
if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
SWAP r1,r2              ; Swap top and bottom 16 bits of r2
                        ; write result into r1
```

This page is intentionally left blank

SWI Software Interrupt Control Operation SWI

Format: inst **Operation:** Trigger Instruction Error Level Interrupt

Format Key:

inst = Instruction

Syntax:

	Instruction Code
SWI	`00100010011011110000000000111111`

Flag Affected (32-Bit):

Z		= Unchanged
N		= Unchanged
C		= Unchanged
V		= Unchanged
E1	●	= 0
E2	●	= 0

Key:

L	= Limm Data

Related Instructions:

None

Description:

The software interrupt instruction is decoded in stage two of the pipeline and if executed, then it immediately raises the instruction error exception. The instruction error exception will be serviced using the normal interrupt system. ILINK2 is used as the return address in the service routine.

Once an instruction error exception is taken, then the medium and low priority interrupts are masked off so that ILINK2 register can not be updated again as a result of an interrupt thus preserving the return address of the instruction error exception.

NOTE Only the reset and memory error exceptions have higher priorities than the instruction error exception.

CAUTION The SWI instruction cannot immediately follow a BRcc or BBITn instruction.

Pseudo Code Example:

```
ILINK2 = nPc                                              /* SWI */
STATUS32_L2 = STATUS32
STATUS32[E2] = 0
STATUS32[E1] = 0
PC = INT_VECTOR_BACE + 0x10
```

Assembly Code Example:

```
SWI                          ; Software interrupt
```

TST **Test** **Logical Operation** TST

Format: inst src1, src2 **Operation:** if (cc=true) then src1 AND src2

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition Code

Syntax:

		Instruction Code	
TST	b,s12	`00100bbb100010111BBBssssssSSSSSS`	
TST<.cc>	b,c	`00100bbb110010111BBBCCCCCC0QQQQQ`	
TST<.cc>	b,u6	`00100bbb110010111BBBuuuuuu1QQQQQ`	
TST<.cc>	b,limm	`00100bbb110010111BBB1111100QQQQQ`	L
TST<.cc>	limm,c	`00100110110010111111CCCCCC0QQQQQ`	L
TST_S	b,c	`01111bbbccc01011`	

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L	= Limm Data

Related Instructions:

BTST CMP

Description:

Logical bitwise AND of source operand 1 (src1) with source operand 2 (src2) and subsequently updating the flags.

There is no destination register therefore the result of the AND is discarded.

NOTE TST and TST_S always set the flags even thought there is no associated flag setting suffix .

Pseudo Code Example:

```
if cc==true then                                  /* TST */
 alu = src1 AND src2
 Z_flag = if alu==0 then 1 else 0
 N_flag = alu[31]
```

Assembly Code Example:

```
TST r1,r2              ; Logical AND r2 with r1
                       ; and set the flags on the
                       ; result
```

XOR **Logical Bitwise Exclusive OR** XOR

Logical Operation

Format: inst dest, src1, src2 **Operation:** if (cc=true) then dest ← src1 XOR src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition code
XOR	=	Logical Bitwise Exclusive OR

Syntax:

With Result		**Instruction Code**	
XOR<.f>	a,b,c	00100bbb00000111FBBBCCCCCCAAAAAA	
XOR<.f>	a,b,u6	00100bbb01000111FBBBuuuuuuAAAAAA	
XOR<.f>	b,b,s12	00100bbb10000111FBBBssssssSSSSSS	
XOR<.cc><.f>	b,b,c	00100bbb11000111FBBBCCCCCC0QQQQQ	
XOR<.cc><.f>	b,b,u6	00100bbb11000111FBBBuuuuuu1QQQQQ	
XOR<.f>	a,limm,c	0010011000000111F111CCCCCCAAAAAA	L
XOR<.f>	a,b,limm	00100bbb00000111FBBB111110AAAAAA	L
XOR<.cc><.f>	b,b,limm	00100bbb11000111FBBB1111100QQQQQ	L
XOR_S	b,b,c	01111bbbccc00111	
Without Result			
XOR<.f>	0,b,c	00100bbb00000111FBBBCCCCCC111110	
XOR<.f>	0,b,u6	00100bbb01000111FBBBuuuuuu111110	
XOR<.f>	0,b,limm	00100bbb00000111FBBB111110111110	L
XOR<.cc><.f>	0,limm,c	0010011011000111F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V		= Unchanged

Key:

L = Limm Data

Related Instructions:

AND	BIC
OR	

Description:

Logical bitwise exclusive OR of source operand 1 (src1) with source operand 2 (src2). The result is written into the destination register (dest).

Pseudo Code Example:

```
if cc==true then                                            /* XOR */
 dest = src1 XOR src2
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
XOR r1,r2,r3          ; Logical XOR contents of r2
                      ; with r3 and write result
                      ; into r1
```

Chapter 10 — The Host

The Host Interface

The ARC 600 processor was developed with an integrated host interface to support communications with a host system. It can be started, stopped and communicated by the host system using special registers. How the various parts of the ARC 600 processor appear to the host is host interface dependent but an outline of the techniques to control ARC 600 processor are given in this section.

Most of the techniques outlined here will be handled by the software debugging system, and the programmer, in general, need not be concerned with these specific details.

NOTE The implemented ARC 600 system may have extensions or customizations in this area, please see associated documentation.

It is expected that the registers and the program memory of ARC 600 processor will appear as a memory mapped section to the host. For example, Figure 10.1 shows two examples: a) a contiguous part of host memory and b) a section of memory and a section of I/O space.

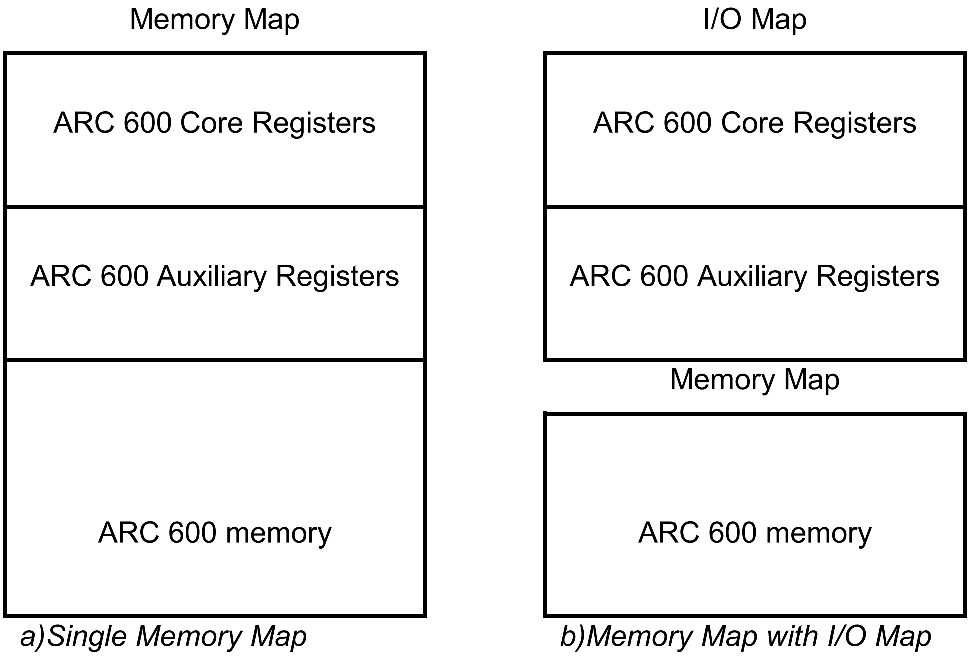


Figure 10.1 Example Host Memory Maps

Once a reset has occurred, the ARC 600 processor is put into a known state and executes the initial reset code. From this point, the host can make the changes to the appropriate part of the ARC 600 processor, depending on whether the ARC 600 processor is running or halted as shown in Table 10.1.

	Running	Halted
Memory	Read/Write	Read/Write
Auxiliary Registers	Mainly No access	Read/Write
Core Registers	No access	Read/Write

Table 10.1 Host Accesses to the ARC 600 processor

Halting

The ARC 600 processor can halt itself with the FLAG instruction or it can be halted by the host. The host halts the ARC 600 processor by setting the H bit in the status register (STATUS32), or by setting the FH bit in the DEBUG register. See Figure 3.8 and Figure 3.10.

When the ARC 600 processor is running only the H bit will change if the host writes to STATUS32 register. However, if the processor halts itself, the *whole* of the STATUS32 register will be updated when the host writes to the STATUS32 register.

The reason for this is that by the time that the host forces a halt, the ARC 600 processor may have halted itself. Therefore, the write of a “halt” number, e.g. 0x01, to the STATUS32 register would overwrite any flag status information that the host required.

In order to force the ARC 600 processor to halt without overwriting the other status flags the additional FH bit in the DEBUG register is provided. See Figure 3.8. The host can test whether the ARC 600 processor has halted by checking the state of the H bit in the STATUS register. Additionally, the SH bit in the debug register is available to test whether the halt was caused by the host, the ARC 600 processor , or an external halt signal. The host should wait for the LD (load pending) bit in the DEBUG register to clear before changing the state of the ARC 600 processor.

Starting

The host starts the ARC 600 processor by clearing the H bit in the STATUS32 register. It is advisable that the host clears any instructions in the pipeline before modifying any registers and re-starting the ARC 600 processor, by sending NOP instructions through, so that any pending instructions that are about to modify any registers in the ARC 600 processor are allowed to complete.

If the ARC 600 processor has been running code, and is to be restarted at a different location, then it will be necessary to put the processor into a state similar to the post-reset condition to ensure correct operation.

- reset the three hardware loop registers to their default values
- flush the pipeline. This is known as ‘pipecleaning’
- disable interrupts, using the status register
- any extension logic should be reset to its default state

If the ARC 600 processor has been running and is to be restarted to CONTINUE where it left off, then the procedure is as follows:

- host reads the status from the STATUS32 Register
- host writes back to the STATUS32 register with *the same* value as was just read, but clearing the H bit
- The ARC 600 processor will continue from where it left off when it was stopped.

NOTE At first glance it appears that the same instruction would be executed twice, but in fact this has been taken care of in the hardware; the pipeline is held stopped for the first cycle after the STATUS32 register has been written and thus the execution starts up again as if there has been no interruption.

Pipecleaning

If the processor is halted whilst it is executing a program, it is possible that the later stages of the pipeline may contain valid instructions. Before re-starting the processor at a new address, these instructions must be cleared to prevent unwanted register writes or jumps from taking place.

If the processor is to be restarted from the point at which it was stopped, then the instructions in the pipeline are to be executed, hence pipecleaning should not be performed.

Pipecleaning is not necessary at times when the pipeline is known to be clean - e.g. immediately after a reset, or if the processor has been stopped by a FLAG instruction followed by three NOPs.

Pipecleaning is achieved as follows:

1. Stop the ARC 600 processor
2. Download a ‘NOP’ instruction into memory.
3. Invalidate instruction cache to ensure that the NOP is loaded from memory
4. Point the PC register to the downloaded NOP
5. Single step until the values in the program counter or loop count register change.
6. Repeat steps 4 and 5 four times

Notice that the program counter is written before each single step, so all branches and jumps, that might be in the pipeline, are overridden, ensuring that the NOP is fetched every time.

It should be noted that the instructions in the pipeline may perform register writes, flag setting, loop set-up, or other operations which change the processor state. Hence, pipecleaning should be performed before any operations which set up the processor state in preparation for the program to be executed - for example loading registers with parameters.

Single Stepping

The Single Step function is controlled by two bits in the DEBUG register. These bits can be set by the debugger to enable the Single Cycle Stepping or Single Instruction Stepping. The two bits, Single Step (SS) and Instruction Step (IS), are write-only by the host and keep their values for one cycle (see Table 10.2).

Field	Description	Access Type
SS	**Single Step**:- Cycle Step enable	Write only from the host
IS	**Instruction Step**:- Instruction Step enable	Write only from the host

Table 10.2 Single Step Flags in Debug Register

Single instruction step

The Single Instruction Step function enables the processor for completion of a whole instruction. The Single Instruction Step function is enabled by setting both the (SS) and (IS) bits in the debug register when ARC 600 processor is halted.

On the next clock cycle the processor is kept enabled for as many cycle as required to complete the instruction. Therefore, any stalls due to register conflicts or delayed loads are accounted for when waiting for an instruction to complete. All earlier instructions in the pipeline are flushed, the instruction that the program counter is pointing to is completed, the next instruction is fetched and the program counter is incremented.

If the stepped instruction was:

- A Branch, Jump or Loop with a delay slot,
  or
- Using Long Immediate data.

Then two instruction fetches are made so that the program counter would be updated appropriately.

The processor halts after the instruction is completed.

SLEEP instruction in single step mode

The SLEEP instruction is treated as a NOP instruction when the processor is in Single Step Mode. This is because every single step acts as a restart or a wake up call. Consequently, the SLEEP instruction behaves exactly like a NOP propagating through the pipeline.

BRK instruction in single step mode

The BRK instruction behaves exactly as when the processor is not in the Single Step Mode. The BRK instruction is detected and kept in stage one forever until removed by the host.

Software Breakpoints

The BRK instruction can also be used to insert a software breakpoint. BRK will halt the ARC 600 processor and flush all previous instructions through the pipe. The host can read the PC register to determine where the breakpoint occurred. As long as the host has access to the ARC 600 processor code memory, it can also replace a ARC 600 processor instruction with a branch instruction. This means that a “software breakpoint” can be set on any instruction, as long as the target breakpoint code is within the branch address range. Since a software breakpoint of this type is a branch instruction, the rules for use of Bcc apply. Care should be taken when setting breakpoints on the last instructions in zero overhead loops and also on instructions in delay slots of jump, branch and loop instructions.

When a debugger performs a restart after a BRK_S instruction has halted the pipeline it must perform an instruction cache invalidation operation. This removes any cached copies of the BRK_S instruction, which will have been overwritten by the debugger in memory with the original instruction at the

BRK_S location. The instruction cache invalidation operation is required even in processors that do not have an instruction cache, as it also has the effect of flushing the first stage of the processor pipeline.

Core Registers

The core registers of ARC 600 processor are available to be read and written by the host. These registers should be accessed by the host once the ARC 600 processor has halted.

Auxiliary Register Set

Some auxiliary registers, unlike the core registers, may be accessed while the ARC 600 processor is running. These dual access registers in the base case are:

STATUS32

The host can read the status register when the ARC 600 processor is running. The main purpose is to see if the processor has halted. See Figure 3.10.

PC

Reading the PC is useful for code profiling. See Figure 3.9.

SEMAPHORE

The semaphore register is used for inter-processor and host to ARC 600 processor communications. Protocols for using shared memory and provision of mutual exclusion can be accomplished with this register. See Figure 3.4.

IDENTITY

The host can determine the version of ARC 600 processor by reading the identity register. See Figure 3.6. Information on extensions added to the ARC 600 processor can be determined through build configuration registers.

NOTE For more information on build configuration registers please refer to associated documentation.

DEBUG

In order to halt the ARC 600 processor, the host needs to set the FH bit of the debug register. The host can determine how the ARC 600 was halted and if there are any pending loads. See Figure 3.8.

Memory

The program memory may be changed by the host. The memory can be changed at any time by the host.

NOTE If program code is being altered, or transferred into ARC 600 memory space, then the instruction cache should be invalidated.