ARC® 700 IP Library

ARCompact™
Instruction Set Architecture

Programmer's Reference

5115-029

ARCompact™ Programmer's Reference

ARC® International

European Headquarters
ARC International,
Verulam Point,
Station Way,
St Albans, Herts, AL1 5HE, UK
Tel. +44 (0) 1727 891400
Fax. +44 (0) 1727 891401

North American Headquarters
3590 N. First Street, Suite 200
San Jose, CA 95134 USA
Tel. +1 408.437.3400
Fax +1 408.437.3401

www.arc.com

ARC Publicly Available Information

Notice

May not be reproduced in any form, including electronic, without specific written permission of ARC International.

The product described in this manual is licensed, not sold, and may be used only in accordance with the terms of a License Agreement applicable to it. Use without a License Agreement, in violation of the License Agreement, 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. ARC International's entire warranty and liability in respect of use of the product are set forth in the License Agreement.

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" and/or "release notes" that are contained in the distribution media. Use of the product is subject to the warranty provisions contained in the License Agreement.

Licensee acknowledges that ARC International is the owner of all Intellectual Property rights in such documents and will ensure that an appropriate notice to that effect appears on all documents used by Licensee incorporating all or portions of this Documentation.

The manual may only be disclosed by Licensee as set forth below.

- Manuals marked "ARC Confidential & Proprietary" may be provided to Licensee's subcontractors under NDA. The manual may not be provided to any other third parties, including manufacturers. Examples--source code software, programmer guide, documentation.
- Manuals marked "ARC Confidential" may be provided to subcontractors or manufacturers for use in Licensed Products. Examples--product presentations, masks, non-RTL or non-source format.
- Manuals marked "Publicly Available" may be incorporated into Licensee's documentation with appropriate ARC permission. Examples--presentations and documentation that do not embody confidential or proprietary information.

The ARCompact instruction set architecture processor is covered by one or more of the following U.S. and international patents: U.S. Patent Nos. 6,178,547, 6,560,754, 6,718,504 and 6,848,074; Taiwan Patent Nos. 155749, 169646, and 176853; and Chinese Patent Nos. ZL 00808459.9 and 00808460.2. U.S., and international patents pending.

U.S. Government Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and/or in similar or successor clauses in the FAR, or the DOD or NASA FAR Supplement.

CONTRACTOR/MANUFACTURER IS ARC International I. P., Inc., 3590 N. First Street, Suite 200, San Jose, CA 95134.

Trademark Acknowledgments

ARCangel, ARChitect, ARCompact, ARCtangent, High C/C++, High C++, the MQX Embedded logo, RTCS, and VRaptor, are trademarks of ARC International. ARC, the ARC logo, High C, MetaWare, MQX, MQX Embedded and VTOC are registered under ARC International. All other trademarks are the property of their respective owners.

5115-029 April-2008

Contents

List of Figures

List of Examples

This page is intentionally left blank.

List of Tables

Chapter 1 — Introduction

This document is aimed at programmers of the ARCompact™ ISA for the ARCtangent™ and ARC® family of processors.

All aspects of the ARCompact ISA are covered in this document, however certain features are only available in specific processor implementations. Features that relate only to specific processor versions are highlighted.

This document covers the instruction set architecture for the following ARCompact based processors:

- ARCtangent-A5 processor
- ARC 600 processor
- ARC 700 processor.

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.

Typographic Conventions

Normal text is displayed using this font.

```
Any code segments are displayed in this mono-space font.
```

TIP Tips point out useful information using this style.

NOTE Notes point out important information.

CAUTION Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

Sections that relate specifically to the ARC 700 processor are marked with this convention.

Sections that relate specifically to the ARC 600 processor are marked with this convention.

Sections that relate specifically to the ARCtangent-A5 processor are marked with this convention.

Sections that relate specifically to both the ARCtangent-A5 and ARC 600 processor are marked with this convention.

Key Features

Instructions

- Freely Intermixed 16/32-Bit Instructions
- User and Kernel Modes

Registers

- General Purpose Core Registers
- Special Purpose Auxiliary Register Set

Load/Store Unit

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

Program Flow

- 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
- Precise Exceptions
- Memory\Instruction\Privilege Exceptions
- Exception Recovery State
- Exception Vectors
- Exception Return Instruction

Multi-Processor Support

- Synchronization and Atomic-exchange instructions

Debug

- Start, stop and single step the processor via special registers
- Full visibility of the processor state via the processors debug interface
- Breakpoint Instruction

Power Management

- Sleep Instruction

Processor Timers

- Two 32-bit programmable timers

ISA Feature Comparison

This document covers the ARCompact ISA definitions for the ARCtangent-A5, ARC 600 and ARC 700 processor implementations.

All processors are upwardly compatible, however due to micro-architectural differences, the timing behavior of each CPU implementation will vary.

Code that is written for processor architectures that make use of all the ARCompact features will not execute correctly on processors that utilize a smaller subset of the ARCompact ISA.

The following table summarizes the key features that are supported by the various processor architectures.

Table 1 Processor Supported Features

ARCompact ISA Features	ARCtangent-A5	ARC 600	ARC 700
Freely Intermixed 16/32-Bit Instructions	●	●	●
General Purpose Core Registers	●	●	●
Auxiliary Register Set	●	●	●
User and Kernel Modes			●
Memory Management Unit Support			●
Extended Arithmetic Instructions	Optional	Optional	●
Register Scoreboard	●	●	●
Address Register Write-Back	●	●	●
Pre and Post Address Register Write-Back	●	●	●
Stack Pointer Support	●	●	●
Scaled Data Size Addressing Mode	●	●	●
PC-relative addressing	●	●	●
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	●	●	●
Levels of Exception	●	●	●

ARCompact ISA Features	ARCtangent-A5	ARC 600	ARC 700
Non-Maskable Exceptions	●	●	●
Maskable External Interrupts	●	●	●
Precise Exceptions			●
Maskable External Interrupts	●	●	●
Memory\Instruction\Privilege Exceptions			●
Exception Recovery State			●
Exception Vectors			●
Exception Return Instruction			●
Synchronization and Atomic-exchange instructions			●
Start, stop and single step the processor via special registers	●	●	●
Full visibility of the processor state via the processors debug interface	●	●	●
Breakpoint Instruction	●	●	●

Programmer's Model

The programmer's model is common to all implementations of the ARCompact based processor and allows upward compatibility of code.

Logically, the ARCompact based processor is based around a general-purpose register file allowing instructions to have two source operands and one destination register. Other registers are contained in the auxiliary register set and are accessed with the LOAD-REGISTER (LR) or STORE-REGISTER (SR) instruction or other special types of instructions.

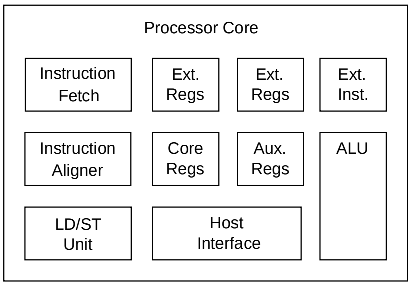


Figure 1 Block diagram of the ARCompact based processor

Core Register Set

The general purpose registers (r0-r28) can be used for any purpose by the programmer. Some of these core registers have defined special purposes like stack pointers, link registers and loop counters. See section Core Register Set on page 39.

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 register and store register instructions, or other special types of instructions. See section Auxiliary Register Set on page 45.

32-bit Instructions

The ARCompact based instruction set, is defined around a 32-bit encoding scheme.

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.

16-bit Instructions

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

- Frequently used instructions only
- Register range reduced from full 64 registers to most frequent 8 registers: r0-r3, r12-r15
- Certain instructions use implied registers like BLINK, SP, GP, FP and PC
- Typically only 1 or 2 operand registers specified (destination and source register are the same)
- Reduced immediate data sizes
- Reduced branch range from maximum offset of ±16MB to maximum offset of ±512B
- 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)

Operating Modes

Operating modes are supported in the ARC 700 processor in order to permit different levels of privilege to be assigned to operating system kernels and user programs – strictly controlling access to 'privileged' system-control instructions and special registers. These operating modes and memory management and protection features combine to ensure that an OS can maintain control of the system at all times, and that both the OS and user tasks can be protected from a malfunctioning or malicious task.

The operating mode is used to determine whether a privileged instruction may be executed. The operating mode is also used by the memory management system to determine whether a specific location in memory may be accessed.

Two operating modes are provided:

- Kernel mode
  - Highest level of privilege
  - Default mode from Reset
  - Access to all machine state, including privileged instructions and privileged registers
- User mode
  - Lowest level of privilege
  - Limited access to machine state

— Any attempt to access privileged machine state causes an exception

Extensions

The ARCompact based 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 where processor extensions occur and how they affect the programmer's view of the ARCompact based processor.

NOTE The implemented 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. Registers r32 to r59 are available for extension purposes. The core register map is shown in Figure 35 on page 39.

NOTE The implemented 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 37 on page 46.

The auxiliary register address region 0x60 up to 0x7F and region 0xC0 up to 0xFF, is reserved for the Build Configuration Registers (BCRs) that can be used by embedded software or host debug software to detect the configuration of the ARCompact based hardware. The Build Configuration Registers contain the version of each ARCompact based extension, as well as configuration information that is build specific.

Some optional components in an ARCompact based based processor system may only provide version information registers to indicate the presence of a given component. These *version registers* are not necessarily part of the Build Configuration Registers set. Optional component version registers may be provided as part of the extension auxiliary register set for a component.

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

Extension Instructions

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

Extension instructions are 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 ARCompact based processor which use only the internal condition flags from the status register (Z, N, C, V), see Table 50 Condition codes on page 135.

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 extension function units or
- test a combination external and internal flags

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

Debugging Features

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

- start and stop the processor via the status and debug register
- single step the processor via the debug register
- check and change the values in the register set and memory
- perform code profiling by reading the status register
- enable software breakpoints by using BRK

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

It is possible for the processor to halt itself with the FLAG instruction.

Power Management

All ARCompact based processors support power management features. The SLEEP instruction halts the pipeline and waits until an interrupt or a restart occurs. Sleep mode stalls the core pipeline and disables any on-chip RAM.

This page is intentionally left blank.

Chapter 2 — Data Organization and Addressing

This chapter describes the data organization and addressing used by the ARCompact based processor.

Address Space

Conceptually the ARCompact ISA has three distinct 32-bit address spaces.

- 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 register accesses.

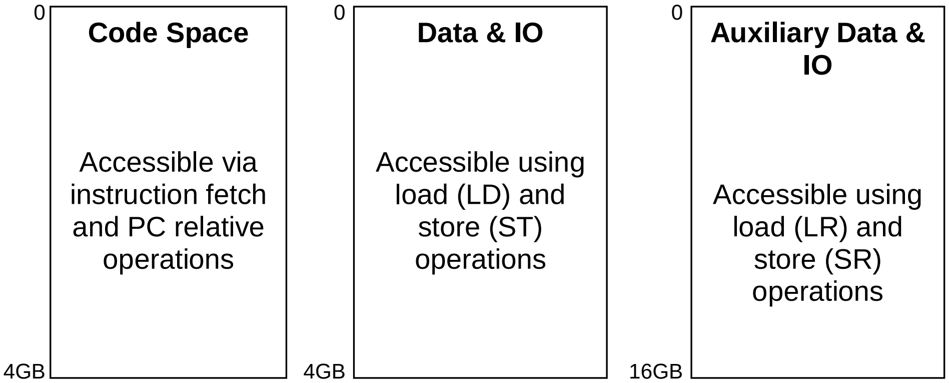

0
Code Space
Accessible via instruction fetch and PC relative operations
4GB
0
Data & IO
Accessible using load (LD) and store (ST) operations
4GB
0
Auxiliary Data & IO
Accessible using load (LR) and store (SR) operations
16GB

Figure 2 Address Space Model

All ARCompact based processors have physically independent Instruction and Data paths that allow for von Neumann or Harvard configurations. However, the default memory configuration for the processor unifies the Data and Instruction memory spaces. A load or store to memory address location *nn* in data memory, will access location *nn* in the instruction memory.

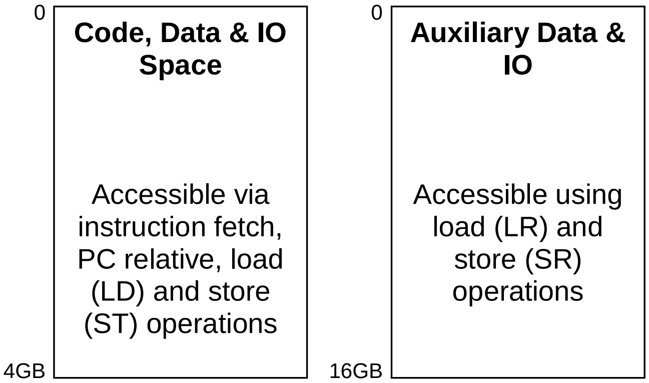

0
Code, Data & IO Space
Accessible via instruction fetch, PC relative, load (LD) and store (ST) operations
4GB
0
Auxiliary Data & IO
Accessible using load (LR) and store (SR) operations
16GB

Figure 3 Unified Address Space Model

Data Formats

All ARCompact based processors by default, support a little-endian architecture. Some configurations of the ARCompact based processor may be big-endian.

The ARCtangent-A5 processor does not support big-endian addressing.

The processor can operate on data of various sizes. The memory operations (load and store type operations) 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

A misaligned data access generates an exception in the ARC 700 processor.

For the ARCtangent-A5 and ARC 600 processors, control of misaligned data access will depend on the configuration of the memory subsystem.

32-bit Data

All load/store, arithmetic and logical operations support 32-bit data. The data representation in a general purpose register is shown in Figure 4 on page 28.

NOTE 32-bit (long word) data should be aligned to 32-bit (long word) boundaries.

Figure 5 on page 28 shows the little-endian representation in byte-wide memory. If the ARCompact based processor supports big-endian addressing then the data would be stored in memory as shown in Figure 6 on page 29.

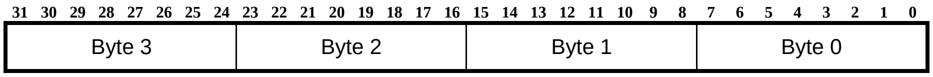


Figure 4 Register Containing 32-bit Data

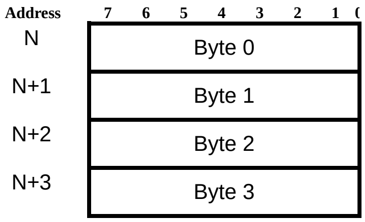


Figure 5 32-bit Register Data in Byte-Wide Memory, Little-Endian

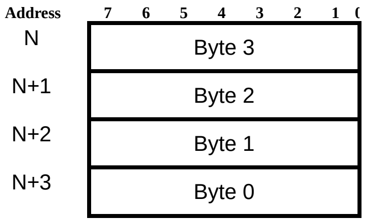


Figure 6 32-bit Register Data in Byte-Wide Memory, Big-Endian

The ARCtangent-A5 processor does not support big-endian addressing.

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. The 16-bit data representation in a general purpose register is shown in Figure 7 on page 29.

For the programmer's model the data is always contained in the lower bits of the core register and the data memory is accessed using a byte address. This model is sometimes referred to as a *data invariance* principle.

NOTE The actual memory bus implementation may have its own representation for data and address. Please see associated documentation.

16-bit (word) data should be aligned to 16-bit (word) boundaries.

Figure 8 on page 29 shows the little-endian representation of 16-bit data in byte-wide memory. If the ARCompact based processor supports big-endian addressing then the 16-bit data would be stored in memory as shown in Figure 9 on page 29.

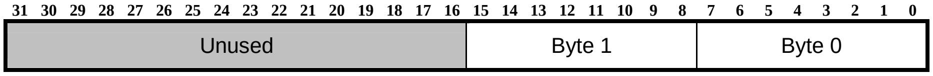


Figure 7 Register containing 16-bit data

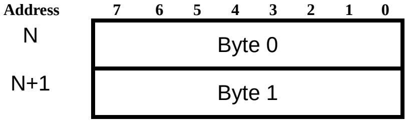


Figure 8 16-bit Register Data in Byte-Wide Memory, Little-Endian

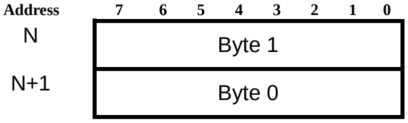


Figure 9 16-bit Register Data in Byte-Wide Memory, Big-Endian

The ARCtangent-A5 processor does not support big-endian addressing.

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. The 8-bit data representation in a general purpose register is shown in Figure 10 on page 30.

For the programmer's model the data is always contained in the lower bits of the core register and the data memory is accessed using a byte address. This model is sometimes referred to as a data invariance principle.

NOTE The actual memory bus implementation may have its own representation for data and address. Please see associated documentation.

Figure 11 on page 30 shows the representation of 8-bit data in byte-wide memory.

Regardless of the endianness of the ARCompact based system, the byte-aligned address, N, of the byte is explicitly given and the byte will be stored or read from that explicit address.

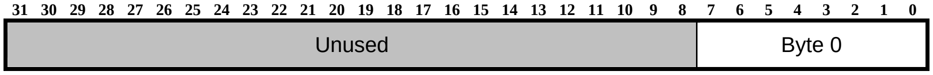


Figure 10 Register containing 8-bit data

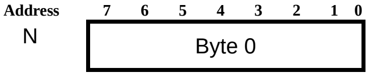


Figure 11 8-bit Register Data in Byte-Wide Memory

1-bit Data

The ARCompact instruction set architecture 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.

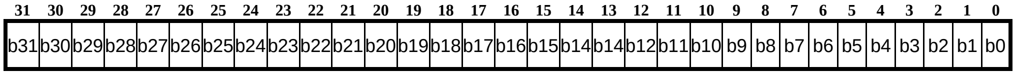

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 b16 b15 b14 b14 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0

Figure 12 Register containing 1-bit data

Extended Arithmetic Data Formats

The ARCtangent-A5 processor supports the extended arithmetic data formats when the optional extended arithmetic instruction library is used.

The ARC 600 processor supports the extended arithmetic data formats when the optional extended arithmetic instruction library is used.

The extended arithmetic instructions are built in to the ARC 700 processor and provide additional data formats.

16-bit Data

16-bit integer or fractional data represented in the high or low parts of the operand. Certain extended arithmetic instructions have specific alignment requirements.

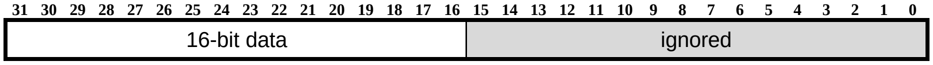


Figure 13 16-bit data format, upper end

or

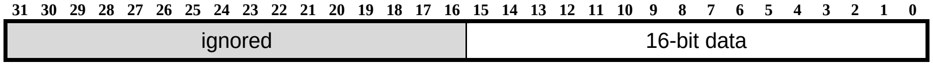


Figure 14 16-bit data format, lower end

Dual 16-bit Data

Two 16-bit integer or fractional data packed as 32-bits. This is the source and destination operand format for the dual 16-bit operations. Channel 1 and channel 2 refer to the high and low parts of the 32-bit data respectively.

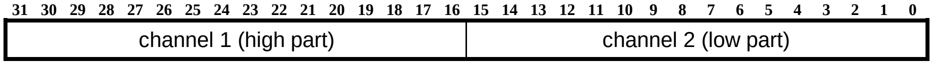


Figure 15 Dual 16 x 16 data format

24-bit Data

24-bit fractional data is represented left justified in 32-bits.

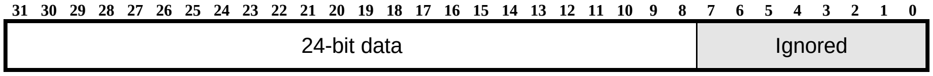


Figure 16 Single 24 x 24 data format

Q Arithmetic

The ‘Q’ mode is used for signed fractional math when using the multiply accumulate units.

Input Format

The input format is: *sign . fraction*

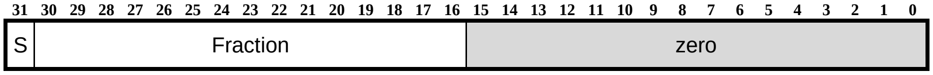


Figure 17 Multiply Accumulate 16-bit Input Data Format

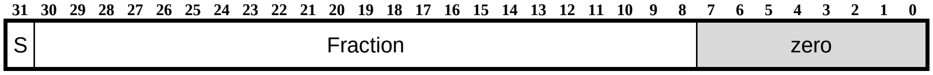


Figure 18 Multiply Accumulate 24-bit Input Data Format

Examples:

Hexadecimal	**Decimal**
0x7FFFFFFF	0.9999..
0x40000000	0.5
0x10000000	0.125
0xC0000000	-0.5
0x80000000	-1.0

Output Format with No Q

When two of fractions are multiplied the result will always be a fractional number less than 1.

However, the sign bit will duplicate giving: *sign sign . fraction*

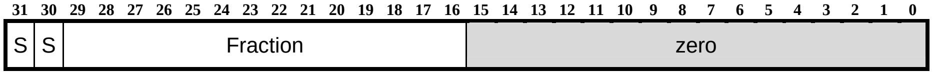


Figure 19 Multiply Accumulate 16-bit Output Data Format with no Q

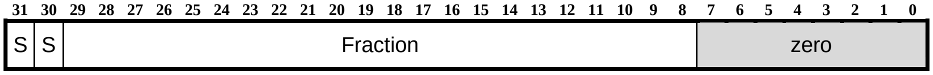


Figure 20 Multiply Accumulate 24-bit Output Data Format with no Q

Output Format with Q

In ‘Q’ arithmetic mode, the multiplier result is shifted left one bit and a zero padded to the right.

The 'Q' arithmetic format is: *sign . fraction*

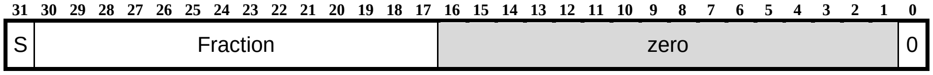


Figure 21 Multiply Accumulate 16-bit Output Data Format with Q

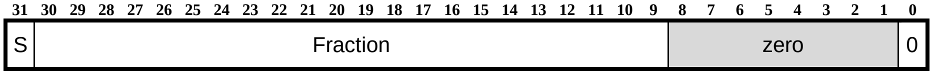


Figure 22 Multiply Accumulate 24-bit Output Data Format with Q

Representation of +1

A special case arises when using Q arithmetic and multiplying –1.0 by –1.0. The result of this operation is +1.0 which can be represented in a accumulator with guard bits enabled but not in a 32-bit register.

For a fractional number represented in a register, the maximum positive number is always 0.99999...... (0x7FFFFFFF), the most positive number that can be represented.

Instruction Formats

The ARCompact instruction set supports freely intermixed 16-bit and 32-bit instructions.

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 containing the major opcode
- 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 containing the major opcode.
- 32-bit long immediate data in the same position as a 32-bit instruction word

Packed Middle-Endian Instruction Format

The basecase ARCompact based 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 ARCompact based 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.

Big-Endian Instruction Format

If the ARCompact based processor has been configured to be big-endian, then no special packing is required since the upper 16-bits of a 32-bit instruction are always provided first.

The ARCtangent-A5 processor does not support big-endian addressing.

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 in Figure 24 on page 33. Assuming a big-endian memory representation, a 32-bit instruction, or 32-bit immediate data will be stored in memory as illustrated in Figure 25 on page 33.

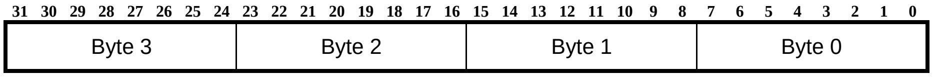


Figure 23 32-bit Instruction byte representation

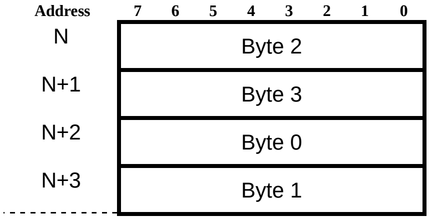


Figure 24 32-bit instruction in Byte-Wide memory, Little-Endian

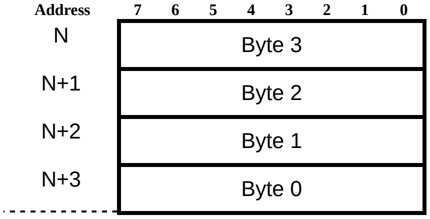


Figure 25 32-bit instruction in Byte-Wide memory, Big-Endian

The ARCtangent-A5 processor does not support big-endian addressing.

Two 16-bit Instructions

Assuming a little-endian memory representation, two packed 16-bit instructions, Figure 26 on page 33, will be stored in memory as illustrated in Figure 27 on page 33. For a big-endian system two 16-bit instructions will be stored in memory as shown in Figure 28 on page 34.

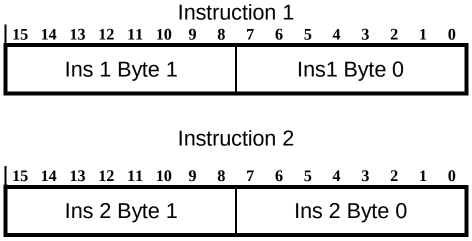


Figure 26 16-bit Instruction byte representation

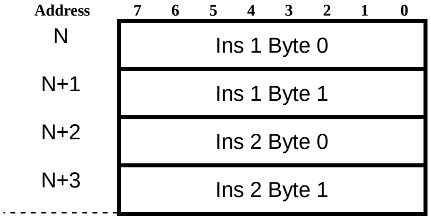
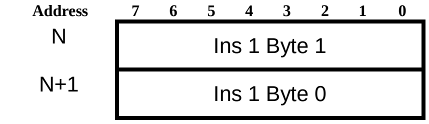


Figure 27 Two 16-bit instructions in Byte-Wide memory, Little-Endian



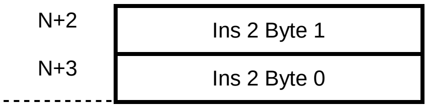


Figure 28 Two 16-bit instructions in Byte-Wide memory, Big-Endian

The ARCtangent-A5 processor does not support big-endian addressing.

16-bit Instruction Followed by 32-bit Instruction

Assuming a little-endian memory representation, a 16-bit instruction followed by a 32-bit instruction, Figure 29 on page 34, will be stored in memory as illustrated in Figure 30 on page 34. For a big-endian system the same instruction sequence will be stored in memory as shown in Figure 31 on page 34.

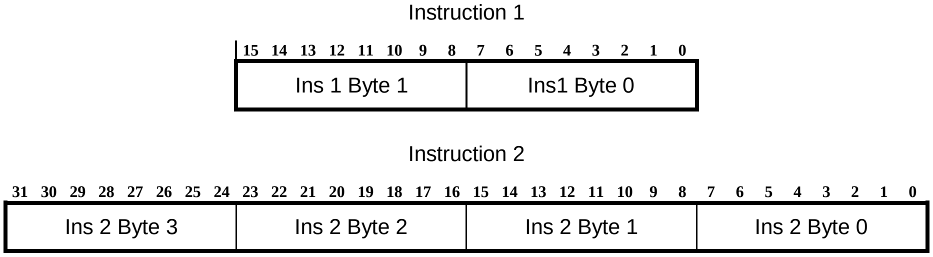


Figure 29 16-bit and 32-bit Instruction byte representation

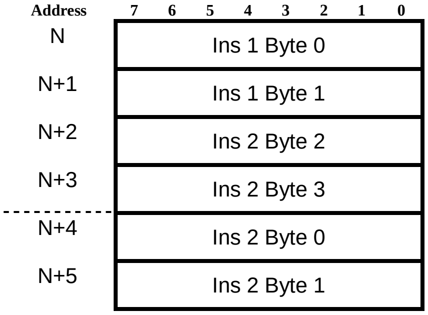


Figure 30 16-bit and 32-bit instructions in Byte-Wide Memory, Little-Endian

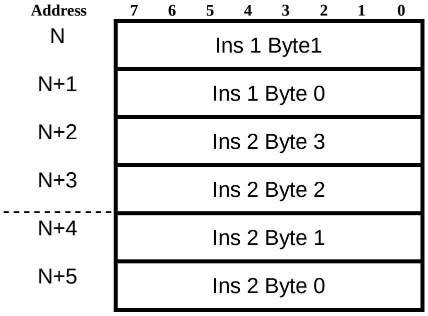


Figure 31 16-bit and 32-bit instructions in Byte-Wide Memory, Big-Endian

The ARCtangent-A5 processor does not support big-endian addressing.

Series of 16-bit and 32-bit Instructions

Assuming a little-endian memory representation, a 16-bit and 32-bit instruction sequence, Figure 32 on page 35, will be stored in memory as illustrated in Figure 33 on page 35. For a big-endian system the same instruction sequence will be stored in memory as shown in Figure 34 on page 36.

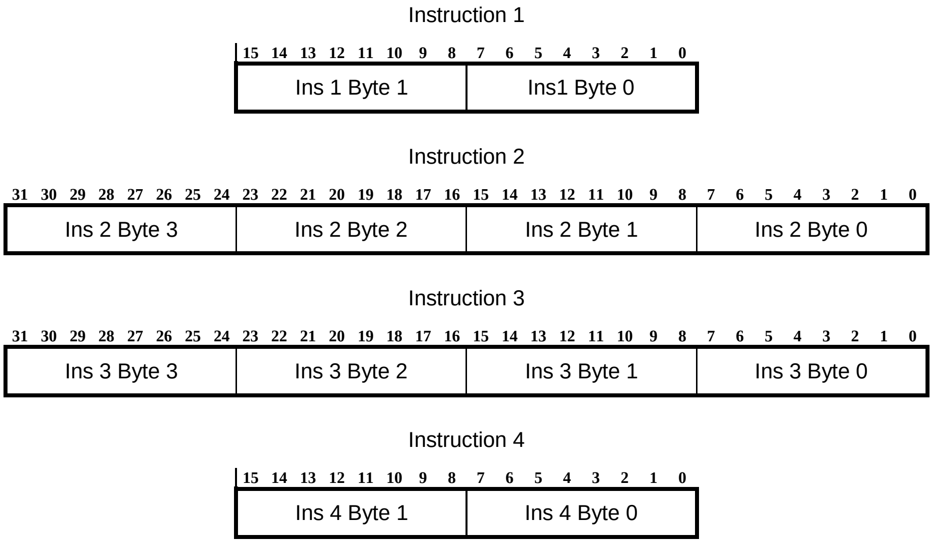


Figure 32 16-bit and 32-bit instruction sequence, byte representation

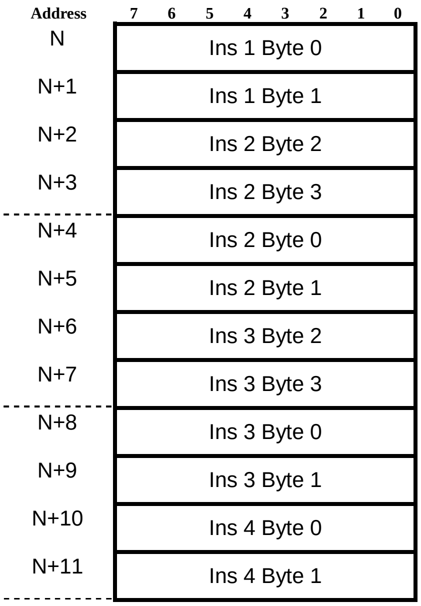


Figure 33 16-bit and 32-bit instruction sequence, in Byte-Wide memory, Little-Endian

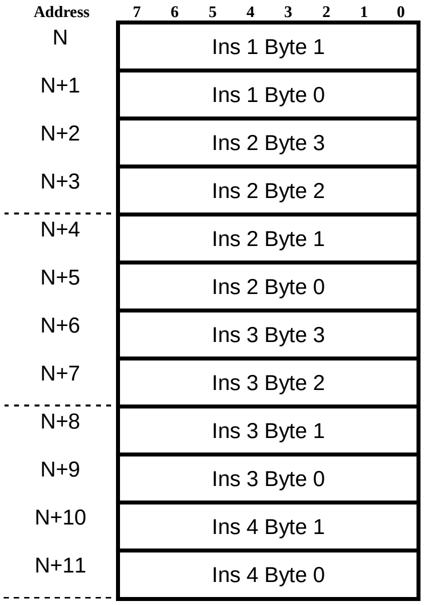


Figure 34 16-bit and 32-bit instruction sequence, in Byte-Wide memory, Big-Endian.

The ARCtangent-A5 processor does not support big-endian addressing.

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.

Example 1 Null Instruction Format

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

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

MOV    0,0           ;Null syntax
                     ;recommended NOP equivalent
```

As all 32-bit instruction formats support this mode, a 32-bit NOP is not explicitly defined. However, the recommended NOP_L equivalent is MOV 0,0. 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.

To take advantage of the ARC 600 branch prediction unit, 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.

Serializing Instructions

Some instructions in the ARCompact based processor are serializing, meaning that they will have full effect before any other instructions can begin execution. Serializing instructions will complete and then flush the pipeline before the next instruction is fetched. BRK and SLEEP are serializing instructions.

In the ARC 700 processor, FLAG, SYNC, and SR are also serializing instructions.

Chapter 3 — Register Set Details

Core Register Set

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

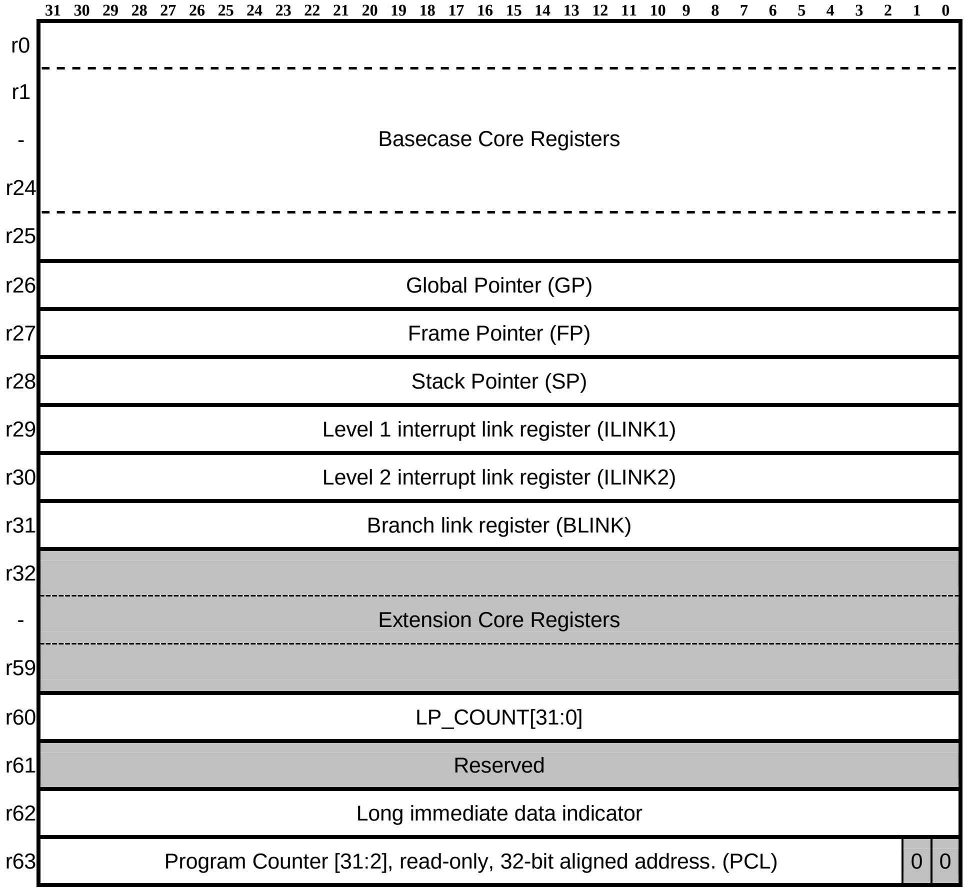


Figure 35 Core Register Map Summary

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 2 on page 39.

Table 2 Core Register Set

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

Register	32-bit Instruction Function and Default Usage	16-bit Instruction Access to Register
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 & 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

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). The special register encoding is shown in Table 3 on page 40 and the ABI usage support is shown in Table 4 on page 41.

Table 3 16-bit instruction register encoding

16-bit instruction register encoding	32-bit instruction register
0	r0
1	r1
2	r2

16-bit instruction register encoding	32-bit instruction register
3	r3
4	r12
5	r13
6	r14
7	r15

Reduced Configuration of Core Registers

The ARC 600 processor can support a reduced set of only 16 core registers. In order to support the ARCompact based ABI the set of reduced registers is indicated in Table 4 on page 41. The RF_BUILD register is used to determine the configuration of core registers.

For the ARC 600 processor writes to non-implemented core registers are ignored, reads return zero, and shortcutting and write-through is disabled. Loads (LD) to non-implemented core registers take place, but the results are discarded. However, this load mechanism should be avoided.

The ARC 700 processor supports the full register set r0 to r31. However, any reference to a non-implemented core register will raise an Instruction Error exception.

Table 4 Current ABI register usage

Register	Use	16-bit Instruction Access	Reduced Configuration
r0-r3	argument regs	●	●
r4-r7	argument regs		
r8-r9	temp regs		
r10-r11	temp regs		●
r12-r15	temp regs	●	●
r16-r25	saved regs		
r26	GP (global pointer.)		●
r27	FP (frame pointer)		●
r28	SP (stack pointer)		●
r29	ILINK1		●
r30	ILINK2		●
r31	BLINK		●

Pointer Registers, GP, r26, FP, r27, SP, r28

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. The ABI usage of core registers is summarized in Table 4 on page 41.

Link Registers, ILINK1, r29, ILINK2, r30, BLINK, r31

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.

For the ARCtangent-A5 and ARC 600 processors ILINK1 or ILINK2 should not be used as targets from multi-cycle instructions.

For the ARC 700 processor ILINK1 and ILINK2 registers are not accessible in user mode. Illegal accesses from user mode to ILINK1 or ILINK2 will cause a Privilege Violation exception and the cause will be indicated in the exception cause register (ECR).

The ILINK1 or ILINK2 registers should not be overwritten by a multi-cycle instruction that retires out-of-order. This is consistent with the restriction ARC 700 already placed on using LP_COUNT and minimises the impact on interrupt response time. Instructions affected include: LD, POP_S, EX, MPY, MPYU, MPYH, MPYHU, and any ARC supplied or user defined extension instructions.

ARC 700 interrupt handling will be delayed until any instruction using ILINK1 or ILINK2 have completed.

Loop Count Register, LP_COUNT, r60

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 it is not recommended that LP_COUNT be used as a general purpose register. See LPcc instruction details on page 247 for further information on the zero delay loop mechanism.

For the ARCtangent-A5 and ARC 600 processor, the LP_COUNT does not have next cycle bypass like the other core registers.

The LP_COUNT register must not be used as the destination of a memory read instruction like LD or POP_S. Instead, an intermediary register should be used, as follows:

Example 2 Correct set-up of LP_COUNT via a register

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

An ARC 700 LD, POP_S or EX instruction to the LP_COUNT register will cause an Instruction Error exception.

The LP_COUNT register must not be used as the destination of multi-cycle instruction. An intermediate register must be used – as with memory accesses to LP_COUNT. A multi-cycle instruction writing to the LP_COUNT register will cause an Instruction Error exception.

The ARC 700 micro architecture ensures that the correct value is always returned when reading the loop count register. The LP_COUNT register can be written at any point within the loop.

The update to the LP_COUNT register will take effect immediately after the writing instruction has finished and after the loop-end mechanism detection has taken place. If the LPcc instruction is in the last position of a loop, any change of program flow required (i.e. jump to LP_START) will be completed before the LP_COUNT register is updated by the instruction.

As a result, writing LP_COUNT from the last instruction in the loop will take effect in the next loop iteration. Writing LP_COUNT from any other position in the loop will take effect in the current loop iteration.

In ARCtangent-A5, in order to guarantee the new value is read, there must be at least 2 instruction words fetched between an instruction writing LP_COUNT and one reading LP_COUNT.

In ARC 600, in order to guarantee the new value is read, there must be at least 1 instruction words fetched between an instruction writing LP_COUNT and one reading LP_COUNT.

Unlike other core registers, the loop count register does not support short cutting (data forwarding).

Example 3 Reading Loop Counter after Writing

```
      MOV    LP_COUNT,r0   ; update loop count register
      MOV    r1,LP_COUNT   ; old value of LP_COUNT
      MOV    r1,LP_COUNT   ; old value of LP_COUNT, ARCtangent-A5
                           ; new value of LP_COUNT, ARC 600
      MOV    r1,LP_COUNT   ; new value of LP_COUNT
```

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 Example 4 on page 43, 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.

Example 4 Invalid Loop Count set up

```
             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
```

Example 5 Valid 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
```

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.

Example 6 Invalid Loop Count set up with branch

```
             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
```

Example 7 Valid 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
```

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:

Example 8 Reading Loop Counter near Loop Mechanism

```
         ...
         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
```

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

Reserved Register, r61

Register r61 is reserved and cannot be used as a general purpose register.

For the ARC 700 processor any reference to the core register r61 will raise an Instruction Error exception.

Immediate Data Indicator, limm, r62

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

Program Counter Long-Word, PCL, r63

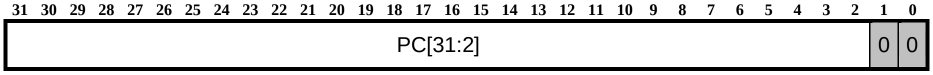


Figure 36 PCL Register

Register r63 (PCL) is a read-only value of the 32-bit Program Counter (PC, 32-bit aligned) for use as a source operand in all instructions allowing PC-relative addressing. The bottom two bits will always return 0.

For the ARCompact based processor the PCL register returns the current instruction address, whereas the PC register returns the the next instruction in sequence.

For the ARC 700 processor, using PCL as a destination register in an instruction will raise an Instruction Error exception.

For the ARC 600 processor, using PCL as a destination register in an instruction will have undefined behavior. Loads to PCL have unpredictable behavior and should also be avoided.

For the ARC 600 processor, PCL should not be used as a source operand in a branch on compare instruction (BBIT0, BBIT1, or BRcc).

Extension Core Registers

The register set is extendible in register positions 32-59 (r32-r59).

Results of accessing the extension register region are undefined for the ARCtangent-A5 and ARC 600 processors. 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. Loads to non-implemented core registers should be avoided.

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

For the ARC 700 processor any reference to a non-implemented core register will raise an Instruction Error exception

Illegal accesses from user mode to implemented core registers will cause a Privilege Violation exception and the cause will be indicated in the exception cause register (ECR).

NOTE When an extension is present but disabled using the XPU register, the exception vector used is Privilege Violation and not Illegal Instruction.

No extension core register can be the target of a load operation (including LD and EX). Thus register values above 31 (with the exception of r62, the limm encoding used as the NULL destination) will cause an Instruction Error exception when used as the destination of a load.

Multiply Result Registers, MLO, MMID, MHI

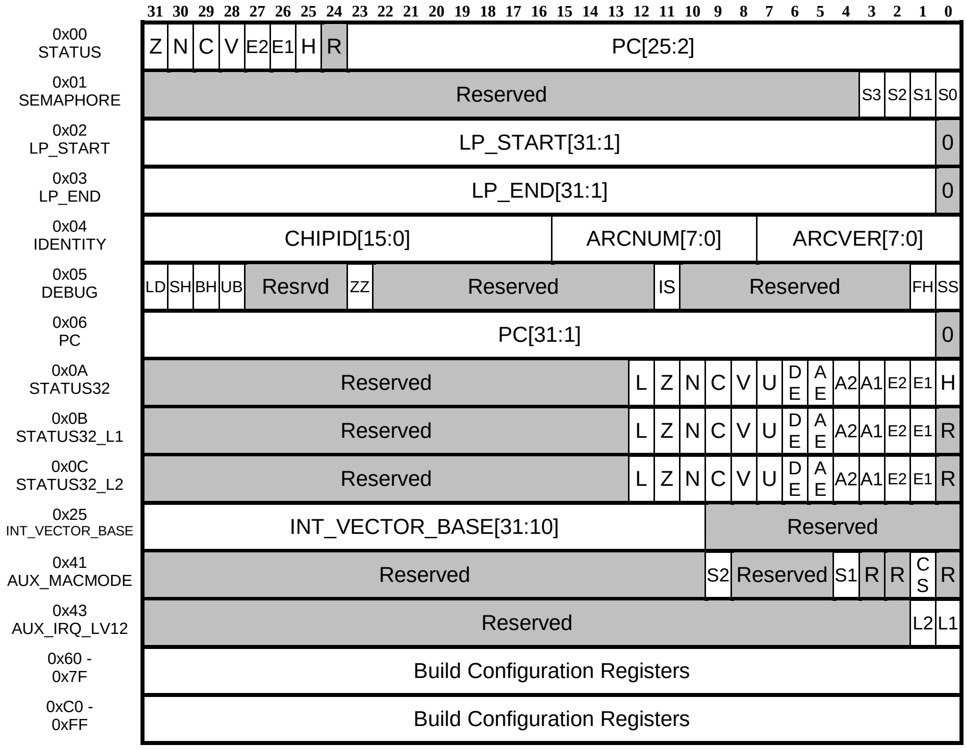
Table 5 on page 45 shows the defined extension core registers for the optional multiply.

Table 5 Multiply Result Registers

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

Auxiliary Register Set

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



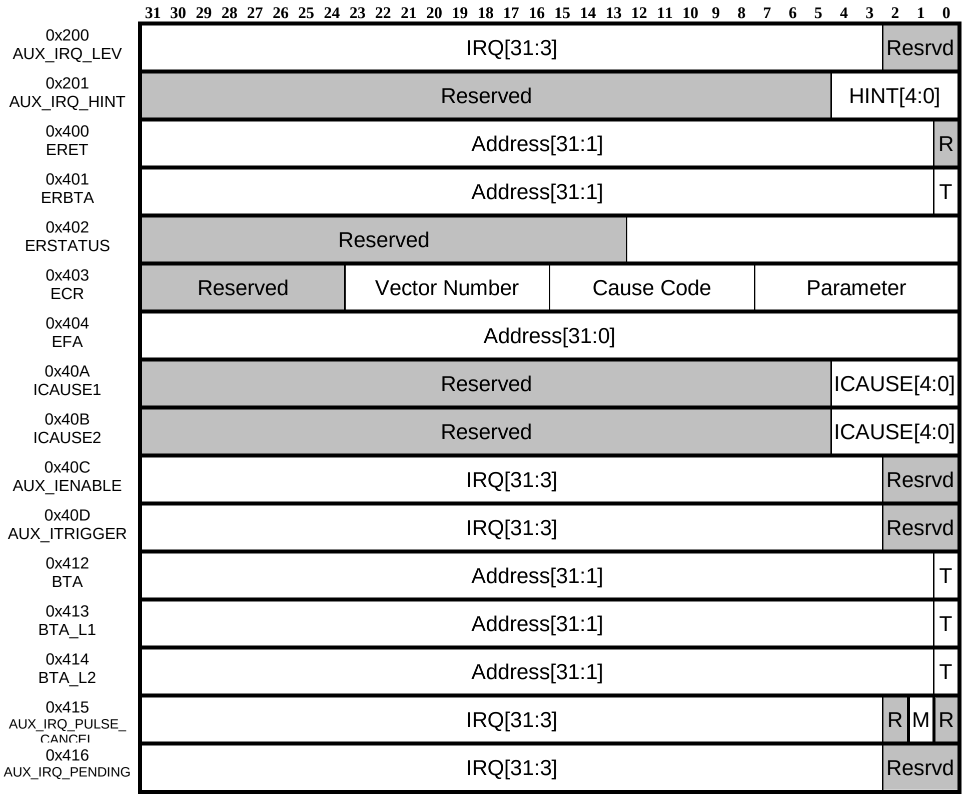


Figure 37 Auxiliary Register Map

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

Table 6 Auxiliary Register Set

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 interrupts
0xC	STATUS32_L2	r/w	Status register save for level 2 interrupts
0x21	COUNT0	r/w	Processor Timer 0 Count value
0x22	CONTROL0	r/w	Processor Timer 0 Control value
0x23	LIMIT0	r/w	Processor Timer 0 Limit value
0x25	INT_VECTOR_BASE	r/w	Interrupt Vector Base address
0x41	AUX_MACMODE	r/w	Extended Arithmetic Status and Mode

Number	Auxiliary register name	LR/SR r/w	Description
0x43	AUX_IRQ_LV12	r/w	Interrupt Level Status
0x60 - 0x7F	RESERVED	r	Build Configuration Registers
0xC0 - 0xFF	RESERVED	r	Build Configuration Registers
0x100	COUNT1	r/w	Processor Timer 1 Count value
0x101	CONTROL1	r/w	Processor Timer 1 Control value
0x102	LIMIT1	r/w	Processor Timer 1 Limit value
0x200	AUX_IRQ_LEV	r/w	Interrupt Level Programming
0x201	AUX_IRQ_HINT	r/w	Software Triggered Interrupt
0x400	ERET	r/w	Exception Return Address
0x401	ERBTA	r/w	Exception Return Branch Target Address
0x402	ERSTATUS	r/w	Exception Return Status
0x403	ECR	r	Exception Cause Register
0x404	EFA	r/w	Exception Fault Address
0x40A	ICAUSE1	r	Level 1 Interrupt Cause Register
0x40B	ICAUSE2	r	Level 2 Interrupt Cause Register
0x40C	AUX_IENABLE	r/w	Interrupt Mask Programming
0x40D	AUX_ITRIGGER	r/w	Interrupt Sensitivity Programming
0x410	XPU	r/w	User Mode Extension Enables
0x412	BTA		Branch Target Address
0x413	BTA_L1	r/w	Level 1 Return Branch Target
0x414	BTA_L2	r/w	Level 2 Return Branch Target
0x415	AUX_IRQ_PULSE_CANCEL	w	Interrupt Pulse Cancel
0x416	AUX_IRQ_PENDING	r	Interrupt Pending Register

Illegal Auxiliary Register Usage

Accessing the extension auxiliary register region in the basecase version of the ARCtangent-A5 processor will return the ID register. If an auxiliary register is read that is defined by an extension but not implemented, then 0 is returned. No exception is generated. Writes to non implemented auxiliary registers are ignored.

If a non existent extension auxiliary register is read in the ARC 600 processor, the value returned is the ID register. If an auxiliary register is read that is defined by an extension but not implemented, then 0 is returned. No exception is generated. Writes to non implemented auxiliary registers are ignored.

For the ARC 700 processor a read or a write of a non existent auxiliary register will raise an Instruction Error exception. Unless otherwise stated in each register description, if a write-only auxiliary register is read, an Instruction Error exception will be raised. Likewise, if a read-only auxiliary register is written, an Instruction Error exception will be raised.

Particular rules apply to Build Configuration Registers.

Status Register (Obsolete), STATUS, 0x00

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
Z	N	C	V	E2	E1	H	R	NEXT_PC[25:2]																							

Figure 38 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 (STATUS32)

Semaphore Register, SEMAPHORE, 0x01

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																												S[3:0]			

Figure 39 Semaphore Register

The SEMAPHORE register, Figure 39 on page 48, is used to control inter-process or ARCompact based processor to host communication. The basecase ARCompact based processor has at least 4 semaphore bits (S[3:0]). The remaining bits of the semaphore register are reserved for future versions of ARCompact based 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 ARCompact based 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 ARCompact based processor and the host. In other words, if the host claims a particular semaphore bit, the ARCompact based processor will not be able to claim that same semaphore bit until the host has released it. Conversely, if the ARCompact based processor claims a particular semaphore bit, the host will not be able to claim that same semaphore bit until the ARCompact based processor has released it.

The semaphore bits are cleared to 0 after a Reset, which is the state where neither the ARCompact based 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 9 Claiming and Releasing Semaphore

```
 .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, LP_START, 0x02, LP_END, 0x03

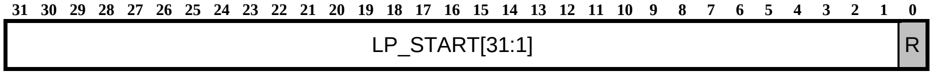


Figure 40 LP_START Register

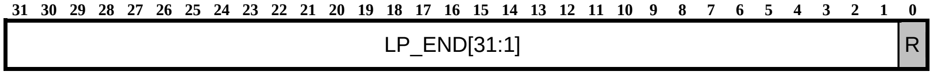


Figure 41 LP_END Register

The loop start (LP_START) and loop end (LP_END) registers contain the addresses for the zero delay loop mechanism. Figure 40 on page 49 and Figure 41 on page 49 show the format of these registers. The loop start and loop end registers can be set up with the special loop instruction (LPcc) 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.

In the ARCompact based processor bit 0 is reserved and should always be set to zero. When reading, bit 0 returns zero. Programming cautions exist when using the loop control registers, See LPcc instruction details on page 247 and Loop Count Register details on page 42 for further information.

Identity Register, IDENTITY, 0x04

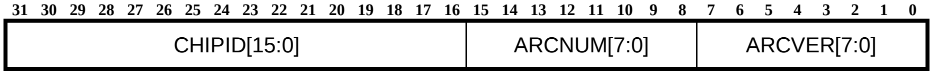


Figure 42 Identity Register

Figure 42 on page 49 shows the identity register (IDENTITY). It contains the unique chip identifier assigned by ARC International (CHIPID[15:0]), the additional identity number (ARCNUM[7:0]) and the ARCompact based 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 = Reserved for ARCtangent-A5 processor family
- 0x20 = Reserved for ARC 600 processor family
- 0x21 = ARC 600 processor family, basecase version 1
- 0x22 = ARC 600 processor family, basecase version 2, supports additional BCR region and accesses to non-existent BCRs will return 0.
- 0x23 to 0x2F = Reserved for ARC 600 processor family
- 0x30 = Reserved for ARC 700 processor family
- 0x31 = ARC 700 processor family, basecase version 1

- 0x32 = ARC 700 processor family, basecase version 2, supports additional BCR region and accesses to BCR region have updated exception model.
- 0x33 to 0x3F = Reserved for ARC 700 processor family
- 0x40 to 0xFF = Reserved

Debug Register, DEBUG, 0x05

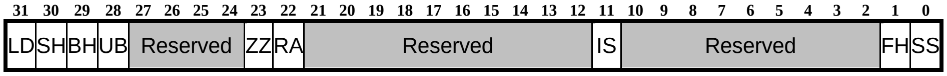


Figure 43 Debug Register

The debug register (DEBUG) contains the following bits:

- load pending bit (LD);
- self halt (SH);
- breakpoint halt (BH);
- sleep mode (ZZ):
- reset applied (RA);
- single instruction step (IS);
- single step (SS);and
- force halt (FH).

LD can be read at any time by either the host (see The Host on page 337) or the ARCompact based 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 ARCompact based processor.

SH indicates that the ARCompact based 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 ARCompact based 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 ARCompact based processor.

The UB bit indicates that BRK is enabled in user mode. This bit is provided to allow an external debugger to debug user-mode tasks. Under all other circumstances, this bit will be set to 0 to ensure that a user-mode task cannot stop the processor by executing a BRK instruction.

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

For the ARC 600 processor, the RA bit is set when a reset has occurred. This bit can be read at any time by either the host (see The Host on page 337) or the ARCompact based processor. The host reads this bit to determine if the system has reset. The bit can only be cleared by the host by writing to the DEBUG register.

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

For the ARC 700 core single instruction stepping is provided through the use of the IS bit, the SS bit is ignored. When the IS bit is set by the host the ARC 700 core will execute one full instruction.

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

Program Counter, PC, 0x06

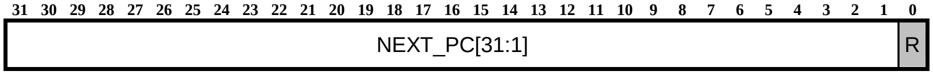


Figure 44 PC Register

The PC register contains the next instruction address based on the 32-Bit program counter. In the ARCompact based processor bit 0 is ignored and should always be set to zero. When reading, bit 0 returns zero.

For the ARCompact based processor the PC register returns the next instruction in sequence, or the target address if the LR instruction is in the delay slot of a branch instruction.

If an LR instruction is in the last instruction position of a zero-overhead loop, the value read from the PC register is undefined.

Status Register 32-bit, STATUS32, 0x0A

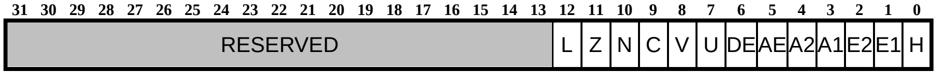


Figure 45 STATUS32 Register

The status register contains the status flags. The status register (STATUS32), shown in Figure 45 on page 51, 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).

The status register is updated by the processor during program flow.The FLAG instruction can be used to set some of the bits in the status register directly, for example to set the Level 1 and Level 2 Interrupt Enables.

The status register can only be read by the processor. However, the status register can be read and written by the host.

When the ARC 700 processor reads the status register in user mode, only the Z, N, C, and V bits are valid. In kernel mode all bits of the register are valid when reading the status register.

The A1 and A2 fields indicate that an interrupt service routine is active. A1 is set on entry to a level 1 interrupt, A2 is set on entry to a level 2 interrupt. Only one bit, A1 or A2, is ever set at any one time in STATUS32. For example, when a level 2 interrupt interrupts a level 1 service routine, A2 is set and A1 is cleared in STATUS32, and the level 2 interrupt status link register STATUS32_L2 will have therefore have A2 cleared and A1 set. When the return from interrupt instruction, RTIE, is executed, A1 and A2 are loaded with values from the selected interrupt status link register (STATUS32_L1 for a level 1 interrupt or STATUS32_L2 for a level 2 interrupt).

The AE bit is set on entry to an exception, and indicates that an exception is active and that the Exception Return Address register (ERET) is valid. When the return from interrupt/exception instruction, RTIE, is executed AE is loaded with the value in the ERSTATUS register.

The DE bit is set in order to indicate that the instruction pointed to by PC32 is the delay slot instruction of a branch. When an instruction completes and this bit is set, the instruction is the delay slot instruction of a branch, irrespective of whether branch or jump is taken. As a result the next instruction required is from the target of the branch. Hence the next PC value is loaded from the Branch Target Address register (BTA). On an exception or interrupt return, the STATUS32 register is reloaded by the RTIE instruction. If the STATUS32[DE] bit is set true as a result of the RTIE operation, the Branch Target Address register (BTA) is simultaneously restored from the Exception Branch Target Address register (ERBTA). The DE bit is only readable by an external debugger or from kernel mode. Using the LR instruction in user mode will return 0 in this bit.

U indicates User mode. User mode restricts access to machine state. Kernel mode, when U is 0, allows an operating system full access. Kernel mode is entered on Reset, interrupt or exception. U is reset to its previous value on interrupt or exception exit when status flags are reloaded from link register.

L indicates whether the zero-overhead loop mechanism is disabled. L is set to 1, indicating loop is disabled on an interrupt or exception. L is reset to its previous value when status flags are reloaded from the link register. L is also cleared when a loop instruction (LPcc) is executed.

The ARCtangent-A5 and ARC 600 processors do not use the A1, A2, AE, DE, U or L fields. These fields will return 0 when read with the LR instruction.

All fields, except the H bit, are set to 0 when the processor is Reset. The H bit is set dependent on the configuration of the processor run state on Reset.

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]
```

Branch Target Address, BTA, 0x412

The BTA register contains the target address of any branch or jump. The value in the BTA register is dependent on whether the branch or jump is taken. The BTA register holds the address to be used after the delay slot has committed in all circumstances.

If the branch or jump is taken the BTA register will contain the target address of the branch or jump. If the branch or jump is not taken the BTA register will contain the address of the instruction that is due to execute immediately after the instruction in the delay slot.

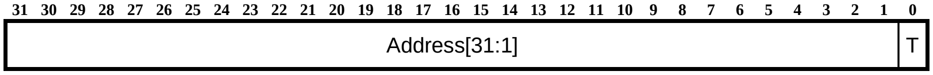


Figure 46 BTA, Branch Target Address

In the ARCompact based processor, the T field (bit 0) when set indicates whether the address field contains the target of a *taken* branch or jump. When the T field is clear, the address field contains address of the next instruction due to execute if the branch/jump is not executed.

Since interrupts are permitted between a branch/jump and an executed delay slot instruction (an exception can also occur on the delay slot instruction), special branch target address registers are used for interrupt and exception handler returns.

When returning from exceptions or interrupts, if the STATUS32[DE] bit will be set true as a result of the RTIE operation, the value in the BTA register will have been restored from the appropriate Interrupt or Exception Return BTA register (ERBTA, BTA_L1 or BTA_L2), allowing the program to resume execution at the correct point.

When returning from an interrupt, the Branch Target Address register (BTA) is loaded from the appropriate high- or low-level Interrupt Return Branch Target Address register (BTA_L1 or BTA_L2).

When returning from an exception, the Branch Target Address register (BTA) is loaded from the Exception Return Branch Target Address (ERBTA) register.

NOTE Certain configurations may not support the BTA, ERBTA, BTA_L1 and BTA_L2 registers. When these registers are not supported, interrupts and exceptions are held off until the instruction in the delay slot commits. The support of these registers is indicated by the BTA_LINK_BUILD configuration register.

Interrupt Status Link Registers, STATUS32_L1, 0x0B, STATUS32_L2, 0x0C

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																			L	Z	N	C	V	U	DE	AE	A2	A1	E2	E1	R

Figure 47 STATUS32_L1, STATUS32_L2 Registers

A level 1 or level 2 interrupt 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 then the current status register STATUS32 will be restored from auxiliary register STATUS32_L1 or STATUS32_L2 accordingly.

In the ARCompact based processor bit 0 is ignored and should always be set to zero. When reading, bit 0 returns zero.

CAUTION For the ARCtangent-A5 and ARC 600 processor 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 Branch Target Link Registers, BTA_L1, 0x413, BTA_L2, 0x414

When returning from an interrupt, the Branch Target Address register (BTA) is loaded from the appropriate high- or low-level Interrupt Return Branch Target Address register (BTA_L1 or BTA_L2).

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
Address[31:1]																															T

Figure 48 BTA_L1 and BTA_L2, Interrupt Return Branch Target Registers

NOTE Certain configurations may not support the BTA, ERBTA, BTA_L1 and BTA_L2 registers. When these registers are not supported, interrupts and exceptions are held off until the instruction in the delay slot commits. The support of these registers is indicated by the BTA_LINK_BUILD configuration register.

Interrupt Vector Base Register, INT_VECTOR_BASE, 0x25

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
INT_VECTOR_BASE[31:10]																						0	0	0	0	0	0	0	0	0	0

Figure 49 INT_VECTOR_BASE Register

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, see Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD on page 80. 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.

Interrupt Level Status Register, AUX_IRQ_LV12, 0x43

After an interrupt has occurred, the level of an interrupt is indicated by the interrupt level status register (AUX_IRQ_LV12) auxiliary register. Two sticky bits are provided to indicate when a level 1 or level 2 interrupt has been taken.

The interrupt level status register is complementary to the A1 and A2 bits of the STATUS32 register.

The sticky bits will stay set until reset by software. Writing '1' to the bit position resets the bits in the interrupt status register, writing a '0' has no effect.

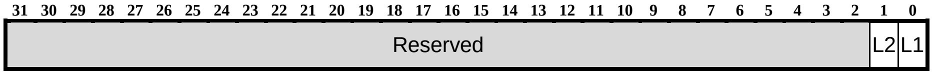


Figure 50 AUX_IRQ_LV12 Interrupt Level Status Register

The level 1 interrupt status bit (L1) is set in hardware if a level 1 interrupt is taken. The L1 bit is cleared in software by writing a '1' to L1. The level 2 interrupt status bit L2 is set in hardware if a level 2 interrupt or exception is taken. The L2 bit is cleared in software by writing a '1' to L2.

Interrupt Level Programming Register, AUX_IRQ_LEV, 0x200

The priority level programming register (AUX_IRQ_LEV) contains the set of interrupts and their priority set. Each interrupt has a corresponding bit position.

A value of '0' in the interrupts bit position represents that the interrupt belongs to priority level 1 set of interrupts and a value of '1' means that the interrupt belongs to priority level 2 set of interrupts.

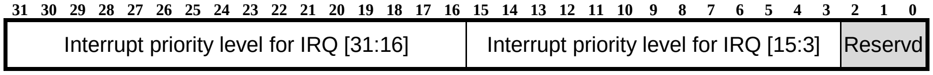


Figure 51 AUX_IRQ_LEV Interrupt Level Programming Register

Bits 0 to 2 are reserved and should be written as 0. Reading from these bits returns 0.

Bits 16 to 31 are only used when the extension interrupts IRQ16-IRQ31 are enabled. If the extension interrupts are not enabled then writing to bits 16 to 31 has no effect and reading returns 0.

After Reset the ARCtangent-A5 processor and ARC 600 processor set all interrupts to their default priority state as shown in the interrupt vector tables, Table 23 and Table 24.

After Reset the ARC 700 processor sets all interrupts to their default priority state as shown in the interrupt vector table, Table 22.

In order to update interrupt priority levels, it is recommended that the AUX_IRQ_LEV register is first read, appropriate bits are updated, and then finally re-written by the ARCompact based code or by the host (see The Host on page 337).

Software Interrupt Trigger, AUX_IRQ_HINT, 0x201

In addition to the SWI/TRAP0 instruction, the interrupt system allows software to generate a specific interrupt by writing to the software interrupt trigger register (AUX_IRQ_HINT). Level 1 and level 2 interrupts (IRQ3 to IRQ31) can be generated through the AUX_IRQ_HINT register. The AUX_IRQ_HINT register can be written through ARCompact based code or from the host (see The Host on page 337).

The software triggered interrupt mechanism can be used even if there are no associated interrupts connected to the ARCompact based processor.

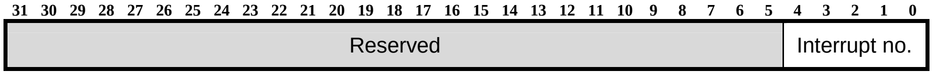


Figure 52 AUX_IRQ_HINT Software Triggered Interrupt

Writing the chosen interrupt value to the AUX_IRQ_HINT register generates a software triggered interrupt. For example a value of 0x09 will trigger an IRQ9 interrupt.

Writing 0x00 to the AUX_IRQ_HINT register clears a software triggered interrupt.

Writing a value greater than 0x1F will clear any software triggered interrupt. Writing values 0x0 to 0x2 have no effect

A read from the AUX_IRQ_HINT register will return the value of the current software triggered interrupt.

A new interrupt should not be generated using the software triggered interrupt system until any outstanding interrupts have been serviced. The AUX_IRQ_HINT register should be read and checked as 0x0 before a new value is written.

If the extension interrupts are not enabled then values outside the range 3 to 15 will clear the AUX_IRQ_HINT register. If extension interrupts are enabled then the valid range of values is extended from 3 to 31.

Since both the host and the ARCompact based code can use the AUX_IRQ_HINT register, a semaphore system needs to be used to control ownership.

The SEMAPHORE register which is available in the ARCtangent-A5 and ARC 600 processor can be used for this purpose.

In the case of pulse sensitive interrupts, no state is kept to indicate what generated the interrupt. It is best practice not to have multiple interrupt sources for pulse sensitive interrupts. For example if an interrupt was generated from both a pulse sensitive interrupt and a software triggered interrupt, then the interrupt service routine would not be able to determine that the pulse sensitive interrupt had also occurred.

It is recommended that the associated interrupt priority level is masked before generating a pulse sensitive interrupts using the AUX_IRQ_HINT register.

For the ARC 700 processor, the AUX_IENABLE register can also be used to mask interrupts generated with AUX_IRQ_HINT.

Interrupt Cause Registers, ICAUSE1, 0x40A, ICAUSE2, 0x40B

When the A1 or A2 bit in the STATUS32 register is true, the associated interrupt cause register (ICAUSE1 or ICAUSE2) will contain the number of the interrupt being handled. Note that a Memory Error interrupt will cause ICAUSE2 to be set to 0x1.

Writing to the Interrupt Cause registers will overwrite any value that has been set by the interrupt system.

The interrupt cause registers, ICAUSE1 and ICAUSE2, are not affected when returning from an interrupt, and when read will return the value of the last interrupt taken.

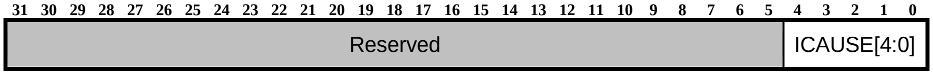


Figure 53 ICAUSE1 and ICAUSE2 Interrupt Cause Registers

Interrupt Mask Programming Register, AUX_IENABLE, 0x40C

The ARC 700 processor uses the AUX_IENABLE register to enable individual masking of each incoming interrupt. Writing a value of '1' in the interrupts bit position enables that particular interrupt. To disable all interrupts, by turning off the interrupt unit, use the FLAG instruction to reset the Level 1 and Level 2 Interrupt Enables.

The AUX_IENABLE register can also be used to mask interrupts generated with the AUX_IRQ_HINT register.

Bits 0 to 2 are reserved and should be written as 0b111. Reading from these bits returns 0b111.

Enable bits for non-present interrupts will return 0, and writes to these bits will be ignored.

If the full set of interrupts are available the AUX_IENABLE register is set to 0xFFFFFFFF when the processor is Reset.

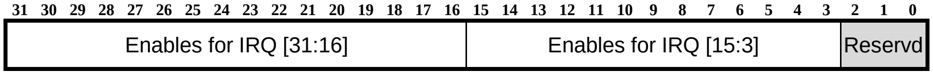


Figure 54 AUX_IENABLE, Interrupt Mask Programming Register

Interrupt Sensitivity Programming Register, AUX_ITRIGGER, 0x40D

The ARC 700 processor uses the AUX_ITRIGGER register to allow an operating system to select whether each interrupt will be level or pulse sensitive.

Bits 0 to 2 are reserved and should be written as 0. Reading from these bits returns 0.

A value of '0' in the interrupts bit position represents that the interrupt is level sensitive and a value of '1' means that the interrupt is pulse sensitive.

This register is set to 0x0 when the processor is Reset which sets all interrupts to be level sensitive.

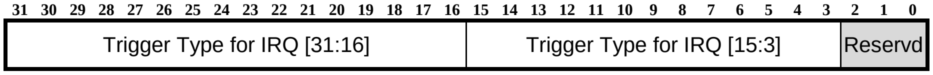


Figure 55 AUX_ITRIGGER, Interrupt Sensitivity Programming Register

Interrupt Pulse Cancel Register, AUX_IRQ_PULSE_CANCEL, 0x415

A write-only 32-bit register, AUX_IRQ_PULSE_CANCEL, is provided to allow the operating system to clear a pulse-triggered interrupt after it has been received, and before it is serviced. Writing '1' to the relevant bit will clear the interrupt if it is set to pulse-sensitivity. If the interrupt is of type level sensitivity, then writing to its relevant bit position will have no effect.

Bits 0 and 2 are reserved and should be written as 0.

Bit 1 is set when a Memory Error interrupt occurs, it is cleared by writing to it.

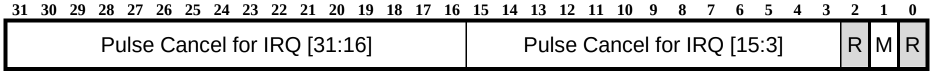


Figure 56 AUX_IRQ_PULSE_CANCEL Interrupt Pulse Cancel Register

Interrupt Pending Register, AUX_IRQ_PENDING, 0x416

The read-only Interrupt Pending register, AUX_IRQ_PENDING, is provided to allow the operating system to determine which interrupts are currently asserted and awaiting service.

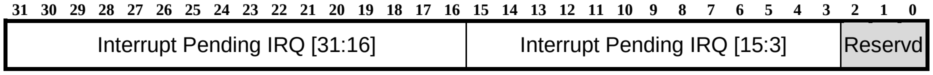


Figure 57 AUX_IRQ_PENDING, Interrupt Pending Register

Reading from bits 0 to 2 bits returns 0.

Exception Return Address, ERET, 0x400

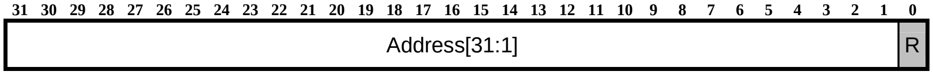


Figure 58 ERET, Exception Return Address

When returning from an exception the program counter (PC) is loaded from the Exception Return Address (ERET) register.

When a fault is detected on an instruction, the exception return address register (ERET) is loaded with the PC value used to fetch the faulting instruction.

If the exception is coerced using a TRAP_S or TRAP0 instruction, the exception return register (ERET) is loaded with the address of the next instruction to be fetched after the TRAP instruction. This value is the architectural PC expected after the TRAP completes – hence pending branches and loops are taken into account.

In the ARCompact based processor bit 0 is ignored and should always be set to zero. When reading, bit 0 returns zero.

Exception Return Branch Target Address, ERBTA, 0x401

When returning from an exception, the Branch Target Address register (BTA) is loaded from the Exception Return Branch Target Address (ERBTA) register.

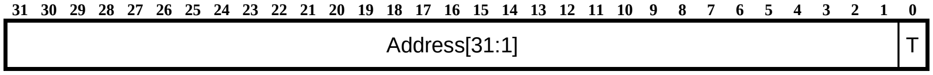


Figure 59 ERBTA, Exception Return Branch Target Address

NOTE Certain configurations may not support the BTA, ERBTA, BTA_L1 and BTA_L2 registers. When these registers are not supported, interrupts and exceptions are held off until the instruction in the delay slot commits. The support of these registers is indicated by the BTA_LINK_BUILD configuration register.

Exception Return Status, ERSTATUS, 0x402

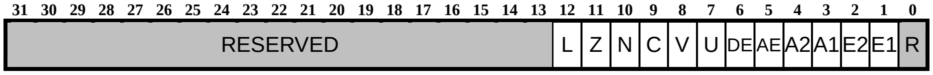


Figure 60 ERSTATUS, Exception Return Status Register

An exception will save the current status register STATUS32 in auxiliary register ERSTATUS.

When the RTIE instruction is executed to return from the exception handler then the current status register STATUS32 will be restored from auxiliary register ERSTATUS.

In the ARCompact based processor bit 0 is ignored and should always be set to zero. When reading, bit 0 returns zero.

Exception Cause Register, ECR, 0x403

The Exception Cause register (ECR) is provided to allow an exception handler access to information about the source of the exception condition. The value in the Exception Cause register is made up as shown in Figure 61 on page 58.

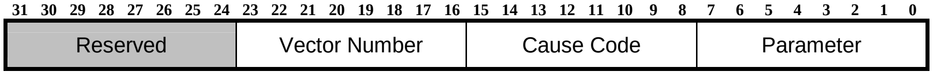


Figure 61 ECR, Exception Cause Register

The Vector Number is an eight-bit number, directly corresponding to the vector number and vector name being used. See Table 25 on page 82 for a list of vector numbers.

Since multiple exceptions share each vector, the eight bit Cause Code is used to identify the exact cause of an exception. See Table 26 on page 87 for a full list of exception cause codes.

The eight bit Parameter is used to pass additional information about an exception that cannot be contained in the previous fields. See Table 26 on page 87 for a full list of exception parameters.

Writing to the Exception Cause register will overwrite any value that has been set by the exception system.

Interrupts do not set the exception cause register. Receipt of interrupts sets the appropriate ICAUSE*n* register to the number of the last taken interrupt.

Exception Fault Address, EFA, 0x404

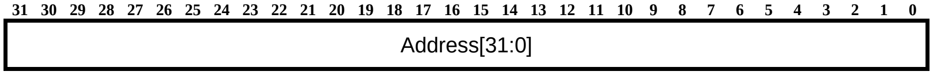


Figure 62 EFA, Exception Fault Address

When an exception occurs, the exception fault address register (EFA) is loaded with the address associated with the fault. For memory data operations, this is the target of the operation. For other faults, the EFA register will be loaded with the PC value used to fetch the faulting instruction.

User Mode Extension Enable Register, XPU, 0x410

The 32-bit register, XPU, is provided to control access to extension instructions and state. The enable bits of the register is used to control groups of extension functions rather than individual instructions or registers. The register allows:

- Disabling of extension functions - for example to permit software emulation of extensions to be tested
- Operating systems to grant user-mode access to extension functions and state
- Intelligent context switching of extension state (lazy context switch)
- Context switching of extension hardware in system containing reconfigurable logic
- Extension enables could be used as part of a power reduction scheme

A group of extensions would be a related set of instructions and registers, for example

- DSP extensions group
- Cryptography extensions group
- etc

All extension functions are assigned to an extensions group.

When an attempt is made to access an extension function (whether instruction or state), the permission bit for the extension group is checked. If the permission is enabled, the access is successful. If the permission is disabled, the CPU will generate a Privilege Violation.

The exception cause register (ECR) is loaded with an appropriate code in order that an OS can:

- Distinguish between an access to a disabled extension and a non-existent extension.
- For a disabled-extension, determine which extension group was accessed

With this functionality, various scenarios are possible for OS control of extensions.

User Mode Extension Enable Register

On Reset, the user mode extensions permission register is set to 0x00000000 in order to disable all extension functions.

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
u31	u30	u29	u28	u27	u26	u25	u24	u23	u22	u21	u20	u19	u18	u17	u16	u15	u14	u14	u12	u11	u10	u9	u8	u7	u6	u5	u4	u3	u2	u1	u0

Figure 63 XPU, User Mode Extension Permission Register

Groups u0 to u15 are reserved for extensions provided by ARC International. Groups u16 to u31 are available for customer use.

Processor Timers Auxiliary Registers

The processor timers are two independent 32-bit timers. Timer 0 and timer 1 are identical in operation, their only difference being that they are connected to different interrupts.

The processor timers are connected to a system clock signal that operates even when the ARCompact based processor is in sleep mode. The timers can be used to generate interrupt signals that will wake the processor from sleep mode.

During ARC 700 debug access, for example when the debug system is reading auxiliary registers or memory, the processor timers are paused so that debug operations are not included in the cycle count.

The processor timers automatically reset and restart their operation after reaching the limit value. The processor timers can be programmed to count only the clock cycles when the processor is not halted. The processor timers can also be programmed to generate an interrupt or to generate a system Reset upon reaching the limit value.

Programming

In order to program a timer *n*, the following sequence should be used:

- Write 0 to the CONTROL*n* register to disable interrupts
- Write the limit value to the timer LIMIT*n* register
- Set up the control flags according to the desired mode of operation by updating the timer CONTROL*n* register
- Write the count value to the timer COUNT*n* register.

Timer *n* starts counting from the COUNT*n* value upwards until it reaches the LIMIT*n* value after which a level type interrupt, if enabled, is generated. Timer *n* then automatically restarts to count from 0 upward until it reaches the limit value again.

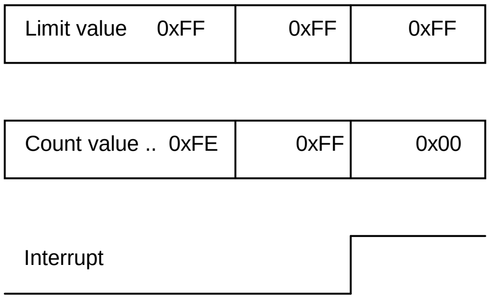


Figure 64 Interrupt Generated after Timer Reaches Limit Value

It is up to the software to clear the timer interrupts. Once an interrupt is generated, writing to CONTROL*n* register clears it. This should be performed during the interrupt service routine.

In Watchdog mode, see The reset signal is activated two cycles after the limit condition has been reached.

Timer 0 Count Register, COUNT0, 0x21

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
Timer Count Value

Figure 65 Timer 0 Count Value Register

The timer count value, COUNT0, is a read/write register. Writing to this register sets the initial count value for the timer, and restarts the timer. Subsequently, the register can be read to reflect the timer 0 count progress.

The COUNT0 register can be updated when the timer is running in which case the internal count register is updated with the new count value and the timer starts counting up from the updated value.

Timer 0 Control Register, CONTROL0, 0x22

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	IP	W	NH	IE

Figure 66 Timer 0 Control Register

The timer control register (CONTROL0) is used to update the control modes of the timer.

Writing to CONTROL0 will de-assert the timer interrupt, but does not stop the timer from counting. The timer continues counting and will independently start the next iteration of counting, setting COUNT0 to 0, when LIMIT0 equals COUNT0.

The Interrupt Enable flag (IE) enables the generation of an interrupt after the timer has reached its limit condition. If this bit is not set then no interrupt will be generated. The IE flag is set to 0 when the processor is Reset.

The Not Halted mode flag (NH) causes cycles to be counted only when the processor is running (not halted). When set to 0 the timer will count every clock cycle. When set to 1 the timer will only count when the processor is running. The NH flag is set to 0 when the processor is Reset.

The Watchdog mode flag (W) enables the generation of a system watchdog reset signal after the timer has reached its limit condition. If this bit is not set then no watchdog reset signal will be generated. The watchdog reset signal is activated two cycles after the limit condition has been reached. The watchdog reset signal can be used to cause a system or processor Reset with appropriate custom logic.

If both the IE and W bits are set then only the watchdog reset is activiated since the ARCompact based processor will be reset and the interrupt will be lost.

If both the IE and W bits are clear then the timer will automatically reset and restart its operation after reaching the limit value.

For the ARC 600 processor, the Interrupt Pending flag (IP) is a read only flag that reflects the value of the timer interrupt line. A 0 indicates the value of the interrupt line is low, a 1 indicates the value of the interrupt line is high.

All of the control flags should be programmed in one write access to the CONTROL0 register.

Timer 0 Limit Register, LIMIT0, 0x23

31 … 0
Timer Limit Value

Figure 67 Timer 0 Limit Value Register

The timer limit value, LIMIT0, is a read/write register. The programmer should write the limit value into this register. The limit value is the value after which an interrupt or reset is to be generated. The timer limit register is set to 0x00FFFFFF when the processor is Reset for backward compatibility to previous processor variants.

Timer 1 Count Register, COUNT1, 0x100

31 … 0
Timer Count Value

Figure 68 Timer 1 Count Value Register

See COUNT0 register on page 60 for field information.

Timer 1 Control Register, CONTROL1, 0x101

31 … 4	3	2	1	0
Reserved	IP	W	NH	IE

Figure 69 Timer 1 Control Register

See CONTROL0 register on page 60 for field information.

Timer 1 Limit Register, LIMIT1, 0x102

31 … 0
Timer Limit Value

Figure 70 Timer 1 Limit Value Register

See LIMIT0 register on page 61 for field information.

Extension Auxiliary Registers

The auxiliary register set is extendible up to the full 2^{32} register space. If an extension auxiliary register is accessed that is not implemented then certain conditions apply. See Illegal Auxiliary Register Usage on page 47.

NOTE The implemented 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.

Table 7 Optional Extension Auxiliary Registers

Number	Name	r/w	Description
0x12	MULHI	w	High part of multiply to restore multiply state

Multiply Restore Register, MULHI, 0x12

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 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.

Example 10 Reading Multiply Result Registers

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

Example 11 Restoring 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
```

Extended Arithmetic Auxiliary Registers

The following table summarizes the auxiliary registers that are used by the extended arithmetic library.

Table 8 Extended Arithmetic Auxiliary Registers

Number	Name	r/w	Description
0x41	AUX_MACMODE	r/w	Extended Arithmetic status and mode register

MAC Status and Mode Register, AUX_MACMODE, 0x41

To support the extended arithmetic library, the AUX_MACMODE register is provided. There are two channels in the AUX_MACMODE registers which correspond to channel 1 data (high 16-bit) and channel 2 data (low 16-bit) respectively in the packed 16-bit data format. See also Dual 16-bit Data on page 30. Both channel 1 and channel 2 flags will be updated when any dual word instruction completes. When a non dual word extended arithmetic instruction saturates both saturation flags S1 and S2 will be set.

The saturation flags, S1 and S2 are sticky and both are cleared by writing to the AUX_MACMODE register and setting the CS bit.

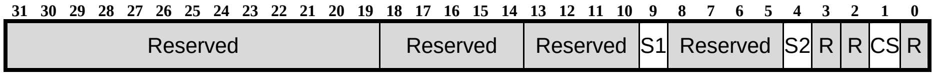


Figure 71 AUX_MACMODE Register

Refer to section Extended Arithmetic Condition Codes on page 170 for discussion of the condition code tests.

Build Configuration Registers

A reserved set of auxiliary registers, called Build Configuration Registers (BCRs), can be used by embedded software or host debug software to detect the configuration of the ARCompact based system. the build configuration registers contain the version of each extension, as well as specific configuration information.

Some optional components in an ARCompact based based processor system may only provide version information registers to indicate the presence of a given component. These *version registers* are not necessarily part of the Build Configuration Registers set. Optional component version registers may be provided as part of the extension auxiliary register set for a component.

Generally each register has two fields, the least significant 8 bits contain the version number of the module, the remaining bits contain configuration information. Any bits within the register that are not required will return zero. The version number field will be set to zero if the module is not implemented in the design, and can therefore be used to detect the presence of the component within the ARCompact based system.

If a non existent extension build configuration register is read in the ARC 600 processor, the value returned is 0. No exception is generated. Writes to build configuration registers are ignored.

For the ARC 700 processor a read of a non existent build configuration register in kernel mode will return 0. No exception is generated. In user mode reads from build configuration registers will raise a Privilege Violation exception. In kernel or user mode writes to build configuration registers will raise an Instruction Error exception.

The following table summarizes the build configuration registers for components that are described in this manual.

Table 9 Build Configuration Registers

Number	Name	r/w	Description
0x60	BCR_VER		Build Configuration Registers Version
0x63	BTA_LINK_BUILD	r	Build configuration for: BTA Registers
0x65	EA_BUILD	r	Build configuration for: Extended Arithmetic
0x68	VECBASE_AC_BUILD	r	Build configuration for: Interrupts
0x6E	RF_BUILD	r	Build configuration for: Core Registers
0x75	TIMER_BUILD	r	Build configuration for: Processor Timers
0x7B	MULTIPLY_BUILD	r	Build configuration for: Multiply

Number	Name	r/w	Description
0x7C	SWAP_BUILD	r	Build configuration for: Swap
0x7D	NORM_BUILD	r	Build configuration for: Normalize
0x7E	MINMAX_BUILD	r	Build configuration for: Min/Max
0x7F	BARREL_BUILD	r	Build configuration for: barrel shift

Build Configuration Registers Version, BCR_VER, 0x60

The BCR version register, BCR_VER, specifies which build configuration register implementation is present.

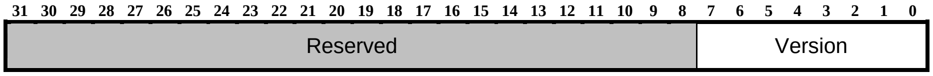


Figure 72 BCR_VER Register

Table 10 BCR_VER field descriptions

Field	Description
Version	Version of Build Configuration Registers 0x00 = Reserved 0x01 = BCR Region at 0x60-0x7F only 0x02 = BCR Region at 0x60-0x7F and 0xC0-0xFF

BTA Configuration Register, BTA_LINK_BUILD, 0x63

The BTA configuration register, BTA_LINK_BUILD, specifies whether the BTA registers are present.

Certain configurations may not support the BTA, ERBTA, BTA_L1 and BTA_L2 registers. When these registers are not supported, interrupts and exceptions are held off until the instruction in the delay slot commits.

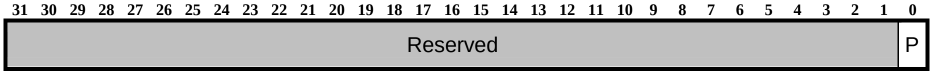


Figure 73 BTA_LINK_BUILD Configuration Register

The field descriptions are shown in the following table.

Table 11 BTA_LINK_BUILD field descriptions

Field	Description
P	Presence of BTA Registers 0x0 = BTA registers are absent 0x1 = BTA registers are present

Extended Arithmetic Configuration Register, EA_BUILD, 0x65

The extended arithmetic configuration register, EA_BUILD, contains the version of the extended arithmetic instructions.

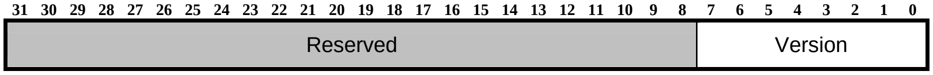


Figure 74 EA_BUILD Configuration Register

Table 12 EA_BUILD field descriptions

Field	Description
Version	Version of Extended Arithmetic 0x00 = Reserved 0x01 = Reserved 0x02 = Current Version

Interrupt Vector Base Address Configuration, VECBASE_AC_BUILD, 0x68

The default base address of the interrupt vector table is fixed when a particular ARCompact based system is created. On Reset the programmable vector base register, INT_VECTOR_BASE is set from the constant value in VECBASE_AC_BUILD .

VECBASE_AC_BUILD is a read only register. Bits 1 to 0 indicate the number of interrupts provided with the interrupt unit.

Bits 10 to 31 show the interrupt vector base address based on the configuration of the interrupt system.

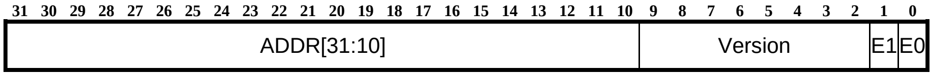


Figure 75 VECBASE_AC_BUILD Configuration Register

Table 13 VECBASE_AC_BUILD field descriptions

Field	Description
Version	Version of Interrupt Unit 0x00 = ARCtangent-A5, ARC 600 Interrupt Unit 0x01 = ARC 700 Interrupt Unit
E[1:0]	Number of interrupts in system 0x0 = 16 interrupts 0x1 = 32 interrupts 0x2 = 8 interrupts (only available in Version 0x01 Interrupt Unit). 0x3 = Reserved
ADDR[31:10]	Interrupt Vector Base Address

Core Register Set Configuration Register, RF_BUILD, 0x6E

The RF_BUILD register is provided to determine whether the base core register set is configured as a 16 or 31 entry set, and whether the register set is cleared on Reset. The RF_BUILD register also indicates whether the register set is made up from a 3 or 4 port register file.

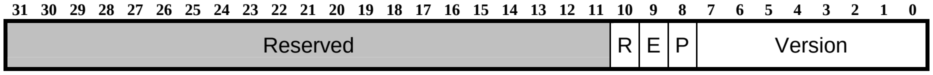


Figure 76 RF_BUILD Configuration Register

The field descriptions are shown in the following table.

Table 14 RF_BUILD field descriptions

Field	Description
Version	Version of Core Register Set 0x01 = Current Version

Field	Description
P	Number of Ports
	0x0 = 3 port register file
	0x1 = 4 port register file
E	Number of Entries
	0x0 = 32 entry register file
	0x1 = 16 entry register file
R	Reset State
	0x0 = Not cleared on reset
	0x1 = Cleared on reset

Processor Timers Configuration Register, TIMER_BUILD, 0x75

The TIMER_BUILD configuration register indicates the presence of the Processor Timers Auxiliary Registers .

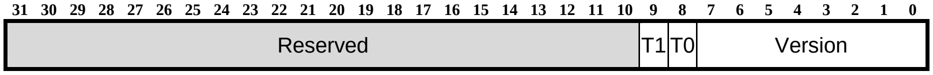


Figure 77 TIMER_BUILD Configuration Register

Table 15 TIMER_BUILD field descriptions

Field	Description
Version	Current version – 0x01 = Version 1 0x02 = ARCtangent-A5 and ARC 700 Processor Timers 0x03 = ARC 600 R3 Processor Timers, with interrupt pending bits
T0	Timer 0 Present 0x0 = no timer 0 0x1 = timer 0 present
T1	Timer 1 Present 0x0 = no timer 1 0x1 = timer 1 present

Multiply Configuration Register, MULTIPLY_BUILD, 0x7B

The multiply configuration register, MULTIPLY_BUILD, contains the version of the multiply instructions.

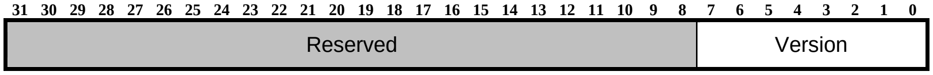


Figure 78 MULTIPLY_BUILD Configuration Register

Table 16 MULTIPLY_BUILD field descriptions

Field	Description
Version	Version of Multiply 0x01 = Multiply 32x32 with special result registers 0x02 = Multiply 32x32 with any result register

Swap Configuration Register, SWAP_BUILD, 0x7C

The multiply configuration register, SWAP_BUILD, contains the version of the SWAP instruction.

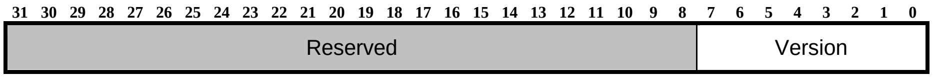


Figure 79 SWAP_BUILD Configuration Register

Table 17 SWAP_BUILD field descriptions

Field	Description
Version	Version of Swap 0x01 = Current Version

Normalize Configuration Register, NORM_BUILD, 0x7D

The multiply configuration register, NORM_BUILD, contains the version of the normalize instructions, NORM and NORMW.

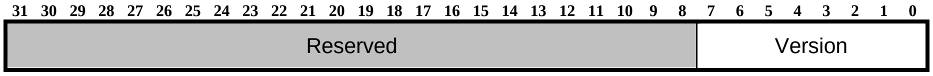


Figure 80 NORM_BUILD Configuration Register

Table 18 NORM_BUILD field descriptions

Field	Description
Version	Version of Swap 0x01 = Reserved 0x02 = Current Version

Min/Max Configuration Register, MINMAX_BUILD, 0x7E

The MIN/MAX configuration register, MINMAX_BUILD, contains the version of the MIN and MAX instructions.

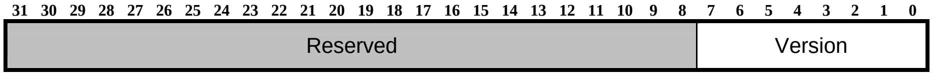


Figure 81 MINMAX_BUILD Configuration Register

Table 19 MINMAX_BUILD field descriptions

Field	Description
Version	Version of Min/Max 0x01 = Reserved 0x02 = Current Version

Barrel Shifter Configuration Register, BARREL_BUILD, 0x7F

The multiply configuration register, BARREL_BUILD, contains the version of the Barrel Shift/Rotate instructions.

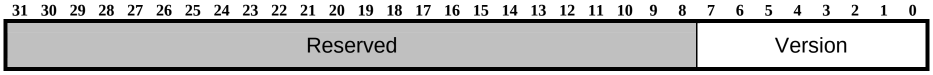


Figure 82 BARREL_BUILD Configuration Register

Table 20 BARREL_BUILD field descriptions

Field	Description
Version	Version of Barrel Shifter 0x01 = Reserved 0x02 = Current Version

This page is intentionally left blank.

Chapter 4 — Interrupts and Exceptions

Introduction

The ARCompact based processor has interrupts and exceptions. Exceptions are synchronous to an instruction. Most exceptions can occur at the same place each time a program is executed (apart from a Memory Error exception that can occur asynchronously), whereas interrupts are typically asynchronous. There are additionally two sets of maskable interrupts: level 2 (mid priority) and level 1 (low priority). The exception set always has the highest priority over the interrupts.

In the ARC 700 processor,. interrupts and exceptions cause the processor to enter into the Kernel operating mode. Depending upon the processor operating mode when an exception or interrupt takes place, the processor can either enter Kernel or User mode upon returning from an interrupt or an exception.

Privileges and Operating Modes

The operating mode, on the ARC 700 processor, is used to determine whether a task may execute a privileged instruction. The operating mode is also used by the memory management system to determine whether a specific location in memory may be accessed.

Two operating modes are provided:

- Kernel mode
- User mode

Various bits in the STATUS32 register are provided in order that kernel mode tasks can determine in which mode they are running, to enable the processor to correctly recover from all legitimate interrupt/exception situations, and to enable the complete processor state to be saved and restored.

Kernel Mode

The ARC 700 kernel mode is the highest level of privilege and is the default mode from Reset. Access to all machine state, including privileged instructions and privileged registers, is provided in Kernel mode.

User Mode

The ARC 700 user mode is the lowest level of privilege and provides limited access to machine state. Any attempt to access privileged machine state, for example privileged instructions or privileged registers, causes an exception.

Privilege Violations

The section describes the privileges available to ARC 700 tasks running in user mode and kernel mode. The following table gives an overview of the differences in privilege between the two modes.

Table 21 Overview of ARC 700 Privileges

Function	User	Kernel
Access to General Purpose Registers	All except ILINK1/2 – no access from user mode	●
Memory management / TLB controls		●
Cache management		●
Access to memory with ASID = User PID	By flag bits in Page Descriptor (PD)	By flag bits in Page Descriptor (PD)
Access to memory with ASID ≠ User PID	If global bit set	If global bit set
Unprivileged instructions	●	●
Privileged instructions		●
Access to Basecase Auxiliary Registers	Only LP_START, LP_END, PC32 and STATUS32[ZNCV]	●
Build Configuration Registers		●
Timer access		●
TRAP_S n, TRAP0	●	●
Interrupt Enable, level selection		●
Extension instructions and state	permissions in XPU	●

Privileged Instructions

All ARC 700 instructions are unprivileged unless specifically defined as privileged. Privileged instructions are:

- SLEEP
- RTIE
- J.F [ILINKn]

These instructions are privileged when STATUS32[UB]=0:

- BRK
- BRK_S

Privileged Registers

Access to the majority of general-purpose registers is not affected by the ARC 700 operating mode. Switching between user and kernel modes does not effect the contents of general-purpose registers. No accesses are permitted to the ILINK1 or ILINK2 registers from user mode. Illegal accesses from user mode to ILINK1 or ILINK2 will cause a Instruction Error exception and the cause will be indicated in the exception cause register (ECR).

Moves to and from auxiliary registers are permitted in both user and kernel mode on the ARC 700 processor. However, in user mode, only a limited set of auxiliary registers may be accessed without causing a protection-violation exception.

Auxiliary registers accessible in user mode include:

- PC
- STATUS32 - ZNCV flags

- LP_START
- LP_END
- Extension Auxiliary Registers - where permitted by extension enables

The remaining auxiliary registers are only accessible in kernel mode.

Switching Between Operating Modes

The ARC 700 processor is set into kernel mode during these transitions:

- TRAP_S, TRAP0
- Interrupt
- Exception
- Reset or Machine check exception
- Write to STATUS32 from debug port when machine is halted

Switching from kernel mode to user mode takes place under the following conditions:

- Return from exception - when machine status register indicates that the last exception was taken from user mode
- Return from interrupt - when machine status register indicates that the highest priority active interrupt was taken from user mode

ARC 700 exception and interrupt handlers may choose to adjust the values in their return address (ERET, ILINK1, ILINK2) and status link registers (ERSTATUS, STATUS32_L1, STATUS32_L2) in order to simultaneously jump to a kernel-mode or user-mode task whilst clearing the relevant interrupt-active or exception-active bits in the status register.

The FLAG instruction cannot be used to change the user or kernel mode state of the ARC 700 processor.

Interrupts

The ARCompact based processor features two level of interrupt:

- level 2 (mid priority) interrupts which are maskable.
- level 1 (low priority) interrupts which are maskable

For the ARC 700 processor, interrupts can be serviced whilst the processor is executing in user mode or kernel mode, and a high-level interrupt can be serviced whilst a low-level interrupt handler is being executed. Although exceptions can be taken in interrupt service routines, interrupts are disabled on entry to exception handling routines.

The interrupt unit is provided with a specific configuration and is programmable.

Interrupt Unit Programming

The interrupt unit allows programming of certain parameters.

Before programming the interrupt unit, all interrupts should be disabled and then all pending interrupts should be dealt with.

For the ARC 700 processor, the AUX_IRQ_PENDING register can be used to ensure that there are no further pending interrupts.

Once the interrupt unit has been programmed accordingly the desired interrupts can be enabled.

Interrupt Unit Configuration

The specific configuration of the interrupt unit can be determined by reading the interrupt vector base configuration register, VECBASE_AC_BUILD.

The sensitivity level of each interrupt is dependent on the specific configuration.

The ARC 700 AUX_ITRIGGER register allows the level or pulse sensitivity to be programmed.

Interrupt Priority

Exceptions, like Reset and Instruction Error, have a higher priority than interrupts, the level 2 interrupt set has middle priority and level 1 the lowest priority.

In addition there is a relative priority in the set of interrupts associated with each level. The interrupt vector table indicates a higher priority with a lower "relative priority" value. For example, a relative priority of M1 has the highest priority within the (mid) priority level 2 set.

For the ARCtangent-A5 processor, for example IRQ7 has the highest priority in the level 2 set and IRQ6 has the lowest priority in the level 2 set.

In general with the ARCtangent-A5 and ARC 600 processors, the higher the interrupt number (IRQn) the higher the priority within the interrupt level set. Note, however, that IRQ7 always has the highest relative priority within its level set in order to ensure backward compatibility to previous ARC processors.

For the ARC 700 processor, the higher the interrupt number the lower the priority.

Programming the AUX_IRQ_LEV auxiliary register can change the level priority of each maskable interrupt.

NOTE The implemented 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

The ARCompact based processor does not implement interrupt vectors as such, but rather a table of jumps. When an interrupt occurs the ARCompact based 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 ARCompact based 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 on page 53.

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.

Example 12 Exception Vector Code

```
       ;Start of exception vectors
reset:        JAL    res_service           ;vector 0
mem_ex:       JAL    mem_service           ;vector 1
ins_err:      JAL    instr_service         ;vector 2
ivect3:       JAL    iservice3             ;vector 3,      ilink1
ivect4:       JAL    iservice4             ;vector 4,      ilink1
ivect5:       JAL    iservice5             ;vector 5,      ilink1
ivect6:       JAL    iservice6             ;vector 6,      ilink2
ivect7:       JAL    iservice7             ;vector 7,      ilink2
                     ;start of interrupt service code for
                     ;ivect7
```

In the ARC 700 interrupt system, there are thirty-two default interrupts/ exceptions associated with vectors 0 to 31, and each has its own vector position.

In the ARCtangent-A5 and ARC 600 configurable interrupt system, there are sixteen default interrupts/ exceptions associated with vectors 0 to 15, each having its own vector position. A further 16 extension interrupts may also be provided.

The 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.

Table 22 ARC 700 Interrupt Vector Summary

Vector	Name	Link register	Priority (Default)	Relative Priority	Byte Offset
0	Reset	-	-	-	0x00
1	Memory Error	ILINK2	level 2 : mid	M1	0x08
2	Instruction Error	-	-	-	0x10
3	IRQ3 (Timer 0)	ILINK1	level 1 : low	L1	0x18
4	IRQ4 (Timer 1)	ILINK1	level 1 : low	L2	0x20
5	IRQ5 (UART)	ILINK1	level 1 : low	L3	0x28
6	IRQ6 (EMAC)	ILINK1	level 1 : low	L4	0x30
7	IRQ7 (XY Memory)	ILINK1	level 1 : low	L5	0x38
8	IRQ8	ILINK1	level 1 : low	L6	0x40
9	IRQ9	ILINK1	level 1 : low	L7	0x48
10	IRQ10	ILINK1	level 1 : low	L8	0x50
11	IRQ11	ILINK1	level 1 : low	L9	0x58
12	IRQ12	ILINK1	level 1 : low	L10	0x60
13	IRQ13	ILINK1	level 1 : low	L11	0x68
14	IRQ14	ILINK1	level 1 : low	L12	0x70
15	IRQ15	ILINK1	level 1 : low	L13	0x78
16	IRQ16	ILINK1	level 1 : low	L14	0x80
17	IRQ17	ILINK1	level 1 : low	L15	0x88

Vector	Name	Link register	Priority (Default)	Relative Priority	Byte Offset
18	IRQ18	ILINK1	level 1 : low	L16	0x90
19	IRQ19	ILINK1	level 1 : low	L17	0x98
20	IRQ20	ILINK1	level 1 : low	L18	0xA0
21	IRQ21	ILINK1	level 1 : low	L19	0xA8
22	IRQ22	ILINK1	level 1 : low	L20	0xB0
23	IRQ23	ILINK1	level 1 : low	L21	0xB8
24	IRQ24	ILINK1	level 1 : low	L22	0xC0
25	IRQ25	ILINK1	level 1 : low	L23	0xC8
26	IRQ26	ILINK1	level 1 : low	L24	0xD0
27	IRQ27	ILINK1	level 1 : low	L25	0xD8
28	IRQ28	ILINK1	level 1 : low	L26	0xE0
29	IRQ29	ILINK1	level 1 : low	L27	0xE8
30	IRQ30	ILINK1	level 1 : low	L28	0xF0
31	IRQ31	ILINK1	level 1 : low	L29	0xF8

Table 23 ARCtangent-A5 and ARC 600 Interrupt Vector Summary

Vector	Name	Link register	Priority (Default)	Relative Priority	Byte Offset
0	Reset	-	high	H1	0x00
1	Memory Error	ILINK2	high	H2	0x08
2	Instruction Error	ILINK2	high	H3	0x10
3	IRQ3 (Timer 0)	ILINK1	level 1 : low	L27	0x18
4	IRQ4 (XY Memory)	ILINK1	level 1 : low	L26	0x20
5	IRQ5 (UART)	ILINK1	level 1 : low	L25	0x28
6	IRQ6 (EMAC)	ILINK2	level 2 : mid	M2	0x30
7	IRQ7 (Timer 1)	ILINK2	level 2 : mid	M1	0x38
8	IRQ8	ILINK1	level 1 : low	L24	0x40
9	IRQ9	ILINK1	level 1 : low	L23	0x48
10	IRQ10	ILINK1	level 1 : low	L22	0x50
11	IRQ11	ILINK1	level 1 : low	L21	0x58
12	IRQ12	ILINK1	level 1 : low	L20	0x60
13	IRQ13	ILINK1	level 1 : low	L19	0x68
14	IRQ14	ILINK1	level 1 : low	L18	0x70
15	IRQ15	ILINK1	level 1 : low	L17	0x78

When the extension interrupts are enabled, a further 16 interrupt lines are provided along with their associated vector addresses. By default all extension interrupts belong to the level 1 interrupt set, and IRQ31 has the highest priority within the level 1 interrupt set. Note, however, that IRQ7 always has the highest relative priority within its level set.

The interrupt vector addresses are added contiguously to the default set of interrupt vectors provided by the configurable interrupt system.

The extension interrupts and their vectors are shown in the following table.

Table 24 ARCtangent-A5 and ARC 600 Extension Interrupt Vector Summary

Vector	Name	Link register	Priority (Default)	Relative Priority	Byte Offset
16	IRQ16	ILINK1	level 1 : low	L16	0x80
17	IRQ17	ILINK1	level 1 : low	L15	0x88
18	IRQ18	ILINK1	level 1 : low	L14	0x90
19	IRQ19	ILINK1	level 1 : low	L13	0x98
20	IRQ20	ILINK1	level 1 : low	L12	0xA0
21	IRQ21	ILINK1	level 1 : low	L11	0xA8
22	IRQ22	ILINK1	level 1 : low	L10	0xB0
23	IRQ23	ILINK1	level 1 : low	L9	0xB8
24	IRQ24	ILINK1	level 1 : low	L8	0xC0
25	IRQ25	ILINK1	level 1 : low	L7	0xC8
26	IRQ26	ILINK1	level 1 : low	L6	0xD0
27	IRQ27	ILINK1	level 1 : low	L5	0xD8
28	IRQ28	ILINK1	level 1 : low	L4	0xE0
29	IRQ29	ILINK1	level 1 : low	L3	0xE8
30	IRQ30	ILINK1	level 1 : low	L2	0xF0
31	IRQ31	ILINK1	level 1 : low	L1	0xF8

Level 1 and Level 2 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 (see Figure 45 on page 51) are used to enable level 2 set and level 1 set of interrupts respectively. Interrupts are enabled or disabled with the FLAG instruction.

Example 13 Enabling Interrupts with the FLAG instruction

```
.equ   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
```

Individual Interrupt Enables

The ARC 700 processor uses the AUX_IENABLE register to enable individual masking of each incoming interrupt. Writing a value of 1 in the interrupts bit position enables that particular interrupt. To disable all interrupts, by turning off the interrupt unit, use the FLAG instruction to reset the Level 1 and Level 2 Interrupt Enables.

Priority Level Programming

The configurable interrupt system provides the ability to change the priority set to which an interrupt belongs. The priority level programming register (AUX_IRQ_LEV) contains the set of interrupts and their priority set. Each interrupt has a corresponding bit position.

After Reset the ARCtangent-A5 processor and ARC 600 processor set all interrupts to their default priority state as shown in the interrupt vector tables, Table 23 and Table 24.

After Reset the ARC 700 processor sets all interrupts to their default priority state as shown in the interrupt vector table, Table 22.

Interrupt Level Status

After an interrupt has occurred, the level of an interrupt is indicated by the interrupt level status register (AUX_IRQ_LV12) auxiliary register. Two sticky bits are provided to indicate if a level 1 or level 2 interrupt has been taken. The interrupt level status register can be used to indicate nested interrupts, i.e. a mid priority level 2 interrupt has interrupted a low priority level 1 interrupt. The sticky bits will stay set until reset by software.

The interrupt level status register is complementary to the A1 and A2 bits of the STATUS32 register

Interrupt Cause Registers

Two bits (A1 and A2) are provided in the STATUS32 register to indicate which interrupt levels are currently being serviced. These are set on entry to the interrupt and overwritten by the values copied from STATUS32_L1 or STATUS32_L2 on exit.

When one of these bits in the STATUS32 register is true, the associated interrupt cause register (ICAUSE1 or ICAUSE2) will contain the number of the interrupt being handled. Note that a Memory Error interrupt will cause ICAUSE2 to be set to 0x1.

The interrupt cause registers, ICAUSE1 and ICAUSE2, are not affected when returning from an interrupt.

Pending Interrupts

The read-only Interrupt Pending register, AUX_IRQ_PENDING, is provided to allow the operating system to determine which interrupts are currently asserted and awaiting service.

Software Triggered Interrupt

In addition to the SWI/TRAP0 instruction, the interrupt system allows software to generate a specific interrupt by writing to the software interrupt trigger register (AUX_IRQ_HINT). Level 1 and level 2 interrupts (IRQ3 to IRQ31) can be generated through the AUX_IRQ_HINT register. The AUX_IRQ_HINT register can be written through ARCompact based code or from the host.

The software triggered interrupt mechanism can be used even if there are no associated interrupts connected to the ARCompact based processor.

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.F [ILINKn] instruction. With the flag bit enabled on the jump instruction, the status register is also loaded from the associate STATUS32_Ln 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.

The RTIE instruction can also be used to return from an interrupt. RTIE allows an interrupt handler to use a single instruction for interrupt exit, without needing to know which interrupt level caused entry to the routine. The values contained in the STATUS32[A1/A2] flags are used to determine which link register pair to use for exit.

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.

If the branch target register, BTA, is available, it will be returned to the value stored in the BTA_L1 or BTA_L2 registers.

The interrupt cause registers, ICAUSE1 and ICAUSE2, are not affected when returning from an interrupt.

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

Example 14 No Interrupt Routine for ivect5

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

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. 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.

Interrupts are also held off when a predicted branch is in the pipeline, or when a flag instruction is being processed.

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
- 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
- Whether a predicted branch is being processed (ARC 600)

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.

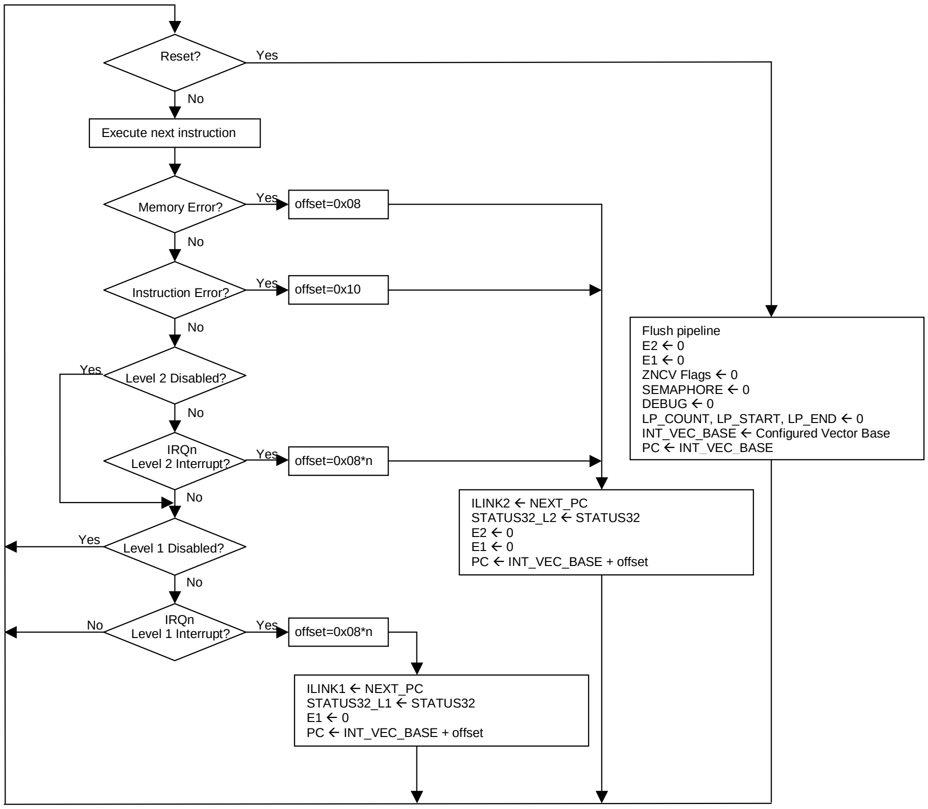


Figure 83 Interrupt Execution

Interrupt Vector Base Address Configuration

The start of the interrupt vectors is dependent on the particular ARCompact based system. On Reset the start of the interrupt vectors is set by the interrupt vector base configuration register, VECBASE_AC_BUILD. This value is also loaded into the interrupt vector base address register, INT_VECTOR_BASE on Reset.

During program execution the start of interrupt vectors can be determined and modified through the interrupt vector base address register, INT_VECTOR_BASE, see Figure 49 on page 54.

Interrupt Sensitivity Level Configuration

The configurable interrupt system can be either pulse sensitive or level sensitive.

An interrupting device that is set to pulse sensitive interrupt, only has to assert the interrupt line once and then de-assert the interrupt line. The fact that a pulse sensitive interrupt has occurred is held until the associated interrupt vector is called. No action is required by the ISR to clear the interrupt.

An interrupting device that is set to level sensitive interrupt must assert and hold the interrupt line until instructed to de-assert the interrupt line by the appropriate interrupt service routine.

The interrupts (IRQ3 to IRQ31) are level sensitive by default, but can be changed to pulse sensitivity depending on the configuration of the interrupt system and configuration of the ARCompact based system.

Interrupt Sensitivity Level Programming

The ARC 700 processor uses the AUX_ITRIGGER register to allow an operating system to select whether each interrupt will be level or pulse sensitive.

Canceling Pulse Triggered Interrupts

A write-only 32-bit register, AUX_IRQ_PULSE_CANCEL, is provided to allow the operating system to clear a pulse-triggered interrupt after it has been received, and before it is serviced. Writing '1' to the relevant bit will clear the interrupt if it is set to pulse-sensitivity. If the interrupt is of type level sensitivity, then writing to its relevant bit position will have no effect.

Exceptions

The processor is designed to allow exceptions to be taken and handled from user mode or kernel mode and from interrupt service routines. An exception taken in an exception handler is a *double fault* condition – and causes a fatal Machine Check exception.

All interrupts and exceptions cause an immediate switch into kernel mode. The Memory Management Unit (if present) is not disabled on entry to an interrupt or exception handler, and the process-ID (ASID) register is not altered. Both levels of interrupt are disabled on entry to an exception handler.

Exception Precision

In the ARCompact based processor precise exceptions are said to be synchronous interrupts associated with specific instructions. Imprecise exceptions are asynchronous events that may or may not be associated with a specific instruction.

In the ARCtangent-A5 and ARC 600 processor the exception scheme is imprecise. The Instruction Error and Memory Error exceptions are non recoverable, in that the instruction that caused the error cannot be returned to.

The ARC 700 processor uses a precise exception scheme. Instructions are restartable, they can be abandoned before completion and restarted later. On receipt of an exception an operating system can therefore choose to either:

- Kill the process
- Send a signal to the process
- Intervene to remove the cause of the exception, and restart operation with the instruction that caused the exception

A memory error exception may not be recoverable depending on the machine state that caused the memory error. For example:

- An instruction cache load that causes a bus error, and hence a Machine Check, Instruction Fetch Memory Error, is said to be precise since the address of the instruction is known at the time of the memory error.
- A data cache load that causes a bus error, and hence a Memory Error, is said to be imprecise, since the instruction is not known at the time of the Memory Error.

Exception Vectors and Exception Cause Register

Any exception that occurs has the following associated information

- Vector Name
- Vector Number
- Vector Offset
- Cause Code
- Parameter

Vector Name

The vector name directly corresponds to the vector number.

Vector Numbers

An eight-bit number, directly corresponding to the vector number and vector name being used.

Vector Offset

The Vector Offset is used to determine the position of the appropriate interrupt or exception service routine for a given interrupt or exception. The vector offset is calculated as 8 times the vector number, and is offset from interrupt/exception vector base address.

The vector offsets are summarized in the following table.

Table 25 Exception vectors

Name	Vector offset	Vector Number	Exception Types
Reset	0x000	0x00	Exception
Memory Error	0x008	0x01	Interrupt
Instruction Error	0x010	0x02	Exception
Interrupts	0x018 - 0x078	-	Interrupt
Interrupts	0x080 - 0x0F8	-	Interrupt
EV_MachineCheck	0x100	0x20	Exception
EV_TLBMissI	0x108	0x21	Exception
EV_TLBMissD	0x110	0x22	Exception
EV_TLBProtV	0x118	0x23	Exception
EV_PrivilegeV	0x120	0x24	Exception
EV_Trap	0x128	0x25	Exception
EV_Extension	0x130	0x26	Exception
Reserved	0x138 - 0x1F8	0x27 - 0xFF	Exception

Table 26 on page 87 shows further details of the exception priorities and exception cause parameters.

Cause Codes

Since multiple exceptions share each vector, this eight bit number is used to identify the exact cause of an exception.

Parameters

This eight bit number is used to pass additional information about an exception that cannot be contained in the previous fields

For the TRAP exception, this field contains the zero-extended six-bit immediate value from the TRAP instruction.

For the Privilege Violation, Disabled Extension exception, this field contains the zero-extended five-bit number of the disabled extension group that was accessed.

When an actionpoint is hit, the parameter contains the number of the actionpoint that triggered the exception.

The parameter can also be used for extension instruction purposes.

Exception Cause Register

The Exception Cause register (ECR) is provided to allow an exception handler access to information about the source of the exception condition. The value in the Exception Cause register is made up from the Vector Number, Cause Code and Parameter, as shown in Figure 61 on page 58.

For example, the TRAP exception has the following values:

Vector Name: EV_Trap

Vector Number: 0x25

Vector Offset: 0x128

Cause Code: 0x00

Parameter: *nn*

This would mean that the cause code register value for TRAP is 0x002500*nn*

Exception Types and Priorities

Multiple exceptions can be associated with a single instruction. In the ARC 700 processor, only one exception can be handled at a time. Remaining exceptions will present themselves when the instruction is restarted after the first exception handler has completed. This process will continue until no further exceptions remain.

Interrupts and exceptions will be evaluated with the following priority:

1. Reset
2. Machine Check, Fatal Cache / TLB error
3. Machine Check, Memory Error – Memory error on D$ flush or Kernel data access
4. Privilege Violation, Instruction fetch Actionpoint hit
5. Machine Check, Double Fault – exception detected when STATUS32[AE]=1
6. Machine Check, Instruction Fetch Memory Error
7. Instruction Fetch TLB miss
8. Instruction Fetch TLB Protection violation
9. Instruction Error - Illegal instruction exception
10. Privilege Violation, Instruction or Register access
11. Privilege Violation, Disabled Extension Group
12. Extension Instruction Exception - requested by extension instruction
13. Protection Violation, LD/ST misalignment

14. Data access TLB miss
15. Data access TLB Protection violation
16. TRAP_S or TRAP0 instructions
17. Memory Error - external bus error
18. Level 2 Interrupt
19. Level 1 Interrupt
20. Core register, Aux register or Memory-access Actionpoint hit

Table 26 on page 87 shows further details of the exception priorities and exception cause parameters.

Reset

A Reset is an external reset signal that causes the ARCompact based processor to perform a *hard* Reset. Upon Reset, various internal states of the ARCompact based 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 ARCompact based 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 ARCompact based processor.

The Reset value of vector base register determines Reset vector address

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

Machine Check, Overlapping TLB Entries

Multiple matches for an address lookup in the TLB.

Machine Check, Fatal TLB Error

Any fatal error in the TLB or its memories (such as a parity or ECC error).

Machine Check, Fatal Cache Error

Any fatal error in the cache controllers or their memories (such as a parity or ECC error).

Machine Check, Kernel Data Memory Error

A memory error was received as a result of a kernel-mode data transaction (LD /ST /PUSH_S /POP_S /EX)

Machine Check, Data Cache Flush Memory Error

A memory error was received as a result of a data cache flush.

Privilege Violation, Actionpoint Hit Instruction Fetch

Actionpoint hit, triggered by instruction fetch. The parameter field (nn) gives the number of the actionpoint that triggered the exception.

Machine Check, Double Fault

Exception detected with exception handler outstanding, as indicated by STATUS32[AE] bit set.

Machine Check, Instruction Fetch Memory Error

A memory error was triggered by an instruction fetch. (memory errors triggered by incorrectly speculated accesses are ignored).

Instruction Fetch TLB Miss

An instruction fetch caused a TLB miss.

Instruction Fetch Protection Violation

An instruction fetch was fetched without the execute permission set.

Instruction Error

If an invalid instruction is fetched that the ARCompact based processor cannot execute, then an Instruction Error is caused.

In the ARCtangent-A5 and 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. This exception uses the level 2 interrupt mechanism and the return information is contained in the ILINK2 and STATUS32_L2 registers.

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

Full decodes of all instructions are performed in the ARC 700 processor. Use of unimplemented instructions, condition codes, core registers, auxiliary registers or encodings will trigger the Instruction Error exception.

In the ARC 700 processor this exception uses the exception mechanism and the return information is contained in the ERET, ERSTATUS and ERBTA registers.

Illegal Instruction Sequence

Triggered when an instruction sequence has been attempted that is not permitted.

The Illegal Instruction Sequence type will occur when any jump or branch instruction straddles the loop end position such that:

- the jump or branch instruction is in the last instruction position of the loop and
- the excuted delay slot is outside the the loop

The Illegal Instruction Sequence type also occurs when any of the following instructions are attempted in an executed delay slot of a jump or branch:

- Another jump or branch instruction (Bcc, BLcc, Jcc, JLcc)
- Conditional loop instruction (LPcc)
- Return from interrupt (RTIE)
- Any instruction with long-immediate data as a source operand

Privilege Violation, Kernel Only Access

Kernel-only instruction, core register or auxiliary register has been accessed from user mode.

Privilege Violation, Disabled Extension

Disabled instruction or register has been accessed. The parameter field (nn) gives the group number (0-31) of the disabled extension.

Extension Instruction Exception

Triggered by an extension instruction if it requires that an exception be taken (e.g. floating point extensions would need to generate many different types of exception). The following are supplied by the extension instruction:

mm = subcode
nn = parameter

Protection Violation, Misaligned Data Access

A misaligned data access causes a TLB protection violation.

Data TLB Miss

Data TLB miss caused by LD, ST, PUSH_S, POP_S or EX instruction.

Data TLB Protection Violation

Data TLB protection violation caused by LD, ST, PUSH_S, POP_S or EX instruction. Caused when the attempted access does not match the permission bits for the page.

Trap

nn = parameter supplied by TRAP_S instruction. TRAP0 supplies nn=00

Note that the instruction always commits, and the return address is the next instruction after the TRAP. This is unlike all other exceptions where the faulting instruction is aborted, and the return address is that of the faulting instruction.

Memory Error

A Memory Error exception is a condition that is detected externally to the CPU. Generally the memory subsystem would detect and raise an error. The types of memory errors typically range from non-existent memory regions to parity/EEC errors.

A memory error condition that is flagged by the external memory system has different effects depending on the context.

A level 2 interrupt is generated if a User mode process triggers a Memory Error condition on the processor bus. This memory error condition is maskable through use of the STATUS32[E2] flag.

An exception is generated if either an instruction fetch access or Kernel mode data access triggers a Memory Error condition on the processor bus.

As precise exception handling is not supported, Memory Errors are handled as non-maskable interrupts. The return address stored for a memory error is not guaranteed to be the address of the faulting instruction. It is the address of the next instruction to be executed in program sequence at the point when the memory error, non-maskable interrupt was received.

Successful recovery from a memory error is not always possible. The non-maskable interrupts use the same interrupt return registers as the highest level of maskable interrupts (Level 2). This means a memory error could be detected whilst the machine is handling a Level 2 interrupt. In this circumstance, the return address information for the interrupt handler would be overwritten by data from the non-maskable interrupt.

NOTE The memory error interrupt is not precise, so an error could be triggered by an instruction outside of a Level 2 Interrupt Service Routine (ISR), but be detected after such an ISR was underway

Systems using Level 2 interrupts cannot guarantee recovery from a memory error non-maskable interrupt.

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

Level 2 Interrupt

Only when STATUS32[E2]=1.

Note that Interrupts do not set the exception cause register. Receipt of this interrupt sets the ICAUSE2 register to the number of the last taken interrupt..

Level 1 Interrupt

Only when STATUS32[E1]=1.

Note that Interrupts do not set the exception cause register. Receipt of this interrupt sets the ICAUSE1 register to the number of the last taken interrupt..

Privilege Violation, Actionpoint Hit Memory or Register

Triggered by Memory access, Core or Auxiliary register access. The parameter field (nn) gives the number of the actionpoint that triggered the exception.

Table 26 Exception Priorities and Vectors

Exception	Vector Name	Vector offset	Vector Number	Cause Code	Exception Cause Register
Reset	Reset	0x000	0x00	0x00	0x000000
Overlapping TLB Entries	EV_MachineCheck	0x100	0x20	0x01	0x200100
Fatal TLB Error	EV_MachineCheck	0x100	0x20	0x02	0x200200
Fatal Cache Error	EV_MachineCheck	0x100	0x20	0x03	0x200300
Kernel Data Memory Error	EV_MachineCheck	0x100	0x20	0x04	0x200400
D$ Flush Memory Error	EV_MachineCheck	0x100	0x20	0x05	0x200500
Actionpoint Hit, Instruction Fetch	EV_PrivilegeV	0x120	0x24	0x02	0x2402nn
Double Fault	EV_MachineCheck	0x100	0x20	0x00	0x200000
Instruction Fetch Memory Error	EV_MachineCheck	0x100	0x20	0x06	0x200600
Instruction Fetch TLB Miss	EV_ITLBMiss	0x108	0x21	0x00	0x210000
Instruction Fetch Protection Violation	EV_TLBProtV	0x118	0x23	0x00	0x230000
Illegal Instruction	Instruction Error	0x010	0x02	0x00	0x020000
Illegal Instruction Sequence	Instruction Error	0x010	0x02	0x01	0x020000
Privilege Violation	EV_PrivilegeV	0x120	0x24	0x00	0x240000
Disabled Extension	EV_PrivilegeV	0x120	0x24	0x01	0x2401nn
Extension Instruction Exception	EV_Extension	0x130	0x26	mm	0x26mmnn

Exception	Vector Name	Vector offset	Vector Number	Cause Code	Exception Cause Register
Misaligned data access	EV_ProtV	0x118	0x23	0x04	0x230400
Data TLB Miss, LD	EV_DTLBMiss	0x110	0x22	0x01	0x220100
Data TLB miss, ST	EV_DTLBMiss	0x110	0x22	0x02	0x220200
Data TLB miss, EX	EV_DTLBMiss	0x110	0x22	0x03	0x220300
Data TLB protection violation, LD	EV_ProtV	0x118	0x23	0x01	0x230100
Data TLB protection violation, ST	EV_ProtV	0x118	0x23	0x02	0x230200
Data TLB protection violation, EX	EV_ProtV	0x118	0x23	0x03	0x230300
Trap	EV_Trap	0x128	0x25	0x00	0x2500nn
External Memory Bus Error	Memory Error	0x008	-	-	-
Level 2 Interrupt	Interrupts	0x018 to 0x0F8	-	-	-
Level 1 Interrupt	Interrupts	0x018 to 0x0F8	-	-	-
Actionpoint Hit, Memory or Register	EV_PrivilegeV	0x120	0x24	0x02	0x2402nn

Exception Detection

Exceptions are taken in strict program order. If more than one exception can be attributed to an instruction, the highest priority exception will be taken and all others ignored. Any remaining exception conditions will be handled when the faulting instruction is re-executed.

Interrupts and Exceptions

The processor is designed to allow exceptions to be taken and handled from user mode or kernel mode and from interrupt service routines. An exception taken in an exception handler is a *double fault* condition – and causes a fatal *machine check* exception.

All interrupts and exceptions cause an immediate switch into kernel mode. The Memory Management Unit is not disabled on entry to an interrupt or exception handler, and the process-ID (ASID) register is not altered. Both levels of interrupt are disabled on entry to an exception handler.

Exception Entry

Note that all addresses described below are the logical addresses used by the program itself.

When an exception is detected the following steps are taken:

- The faulting instruction is cancelled
  - No state changes caused by this instruction can be committed
  - All subsequent instructions that have been fetched into the pipeline are also cancelled.
  - Cache behavior is not explicitly defined by the ISA, and is implementation dependent.
  - All state changes associated with extension core registers or condition codes must also be prevented if an instruction is cancelled, in order that the instruction functions correctly when it is re-fetched.
- When a fault is detected on an instruction, the exception return address register (ERET) is loaded with the PC value used to fetch the faulting instruction.
  - If the exception is coerced using a TRAP_S or TRAP0 instruction, the exception return register (ERET) is loaded with the address of the next instruction to be fetched after the TRAP instruction. This value is the architectural PC expected after the TRAP completes – hence pending branches and loops are taken into account.
- The exception return status register (ERSTATUS) is loaded with the contents of STATUS32 used for execution of the faulting instruction.
  - Since there is a single exception detection point immediately before the commit point, then the value used to load ERSTATUS will be the last value committed to STATUS32.
  - If a delayed program-counter update is pending – due to the faulting instruction being in the delay slot of a taken branch/jump, then the delay-slot bit will be true. STATUS32[DE] = 1
- If a delayed program-counter update is pending – indicated by the STATUS32[DE] bit being true, the exception return branch target address register (ERBTA) is loaded with the pending target PC value. This mechanism is not affected by zero-overhead loops.
- The exception cause register (ECR) is loaded with a code to indicate the cause of the exception – see Table 26 on page 87.
- The exception fault address register (EFA) is loaded with the address associated with the fault. For LD/ST operations, this is the target of the operation. For all other faults, the EFA register will be loaded with the address of the faulting instruction.
- The CPU is switched into kernel mode STATUS32[U] = 0
- Interrupts are disabled STATUS32[E1,E2] = 0
- The exception handler underway flag is set. STATUS32[AE] = 1
- The Program Counter will be loaded with the address of the appropriate exception vector. This is determined by the type of exception detected and the value in the interrupt/exception vector table base register.
- The DE bit in the status register is cleared. STATUS32[DE] = 0

No other state is altered –the stack pointer and all other registers remain unchanged.

The exception handlers must be able to save and restore all processor state that they alter during exception handling.

The MMU provides a 32-bit register SCRATCH_DATA0 that can be used by an Operating System to store data.

Saving of the stack pointer means having a fixed location in the unmapped region of the address space that is used to swap the user mode stack pointer with the exception stack pointer. The use of separate exception/interrupt stacks is a feature of many operating systems. It may also be actually necessary if the memory locations used for the user mode stack for the faulting process do not have read/write privileges enabled for kernel mode.

Exception Exit

Once the exception handler has completed its operations, it must restore the correct context for the task that is to continue execution. The RTIE instruction is used to return from exceptions. The JAL.F [ILINKn] instruction cannot be used.

The RTIE instruction determines which operating mode and interrupt state to return to by checking the A2, A1 and AE bits of STATUS32 in order to establish which copy of the status register (ERSTATUS, STATUS32_L1 or STATUS32_L2) should be used to determine the exception return mode. The U bit of the corresponding link register is used for this purpose.

Table 27 Exception and Interrupt Exit Modes

U	AE	A2	A1	Current Mode	RTIE Response	Link Registers Used
0	0	0	0	Kernel	Exception Exit	ERET ERSTATUS ERBTA
0	0	0	1	ISR Level 1	Interrupt Level 1 Exit	ILINK1 STATUS32_L1 ERBTA_L1
0	0	1	0	ISR Level 2	Interrupt Level 2 Exit	ILINK2 STATUS32_L2 ERBTA_L2
0	0	1	1	ISR Level 2	Interrupt Level 2 Exit	ILINK2 STATUS32_L2 ERBTA_L2
0	1	0	0	Exception	Exception Exit	ERET ERSTATUS ERBTA
0	1	0	1	Exception	Exception Exit	ERET ERSTATUS ERBTA
0	1	1	0	Exception	Exception Exit	ERET ERSTATUS ERBTA
0	1	1	1	Exception	Exception Exit	ERET ERSTATUS ERBTA
1	-	-	-	User	Privilege Violation	Kernel

The case when U, AE, A2, and A1 are all set to 0 is used for state changes from kernel mode, for example when scheduling a user mode task.

If the AE bit is set, or AE, A1 and A2 are all zero, the exception-exit sequence is followed. If AE is zero and either A1 or A2 are set true, the interrupt-exit sequence is followed. See description of the RTIE instruction for further details.

The program counter is loaded with the exception return address from the ERET register, the contents of ERSTATUS are copied into STATUS32and the contents of ERBTA are copied into BTA.

If the delay-slot bit STATUS32[DE] is set as a result, an unconditional delayed branch is set up to the address contained in the branch target address register BTA.

Exceptions and Delay Slots

For the ARCompact based processor exceptions are supported for instructions in the delay slots of branches.

Example 15 Exception in a Delay Slot

```
J.D [blink]          ; Branch/Jump Instruction
LD  fp,[sp,24]       ;
…                    ;
MOV r0,0             ; Target of the branch/jump
```

The ARC 700 processor has features specifically for recovery from exceptions caused by instructions found in branch/jump delay slots.

When an exception is detected on a delay slot instruction, the return address stored on exception entry will be the address of the instruction in the delay slot, which allows an exception handler to return to the delay slot instruction of a taken branch, and for subsequent instructions to be executed starting at the branch target address.

This functionality allows branch instructions that can change processor state to also have delay slots, for example BRcc /BBITn /Jcc using auto-update extension core registers, or simply the BLcc instruction.

Many possible hazards are removed in this scheme which would otherwise occur when not returning to a faulting instruction that was previously cancelled, for example the possibility of TLB thrash/deadlock with a fully-associative scheme

Emulation of Extension Instructions

An illegal exception instruction handler whose intent is to emulate the function of an extension instruction must be able to:

- Get the address of the faulting instruction from the ERET register
- Disassemble the instruction sufficiently to determine whether it should be emulated
- Perform the emulation function, and make whatever changes to processor state (real or emulated) that are required
    - Note that any required changes to ZNCV flags would have to be made in the ERSTATUS register to be restored on exception return
- Return to the next instruction *after* the emulated instruction. The return address could be one of the following (in order of priority):
    - ERBTA – exception branch target address if the faulting instruction was in the delay slot of a taken branch
    - LP_START if the faulting instruction was the last instruction in a zero-overhead loop, and it’s not the last loop iteration (ERET+emulated_instruction_size = LP_END, and LP_COUNT>1).
    - ERET + emulated_instruction_size for *norml* linear code execution

NOTE When an extension is present but disabled using the XPU register, the exception vector used is Privilege Violation and not Illegal Instruction.

Emulation of Extension Registers and Condition Codes

A similar scheme, as defined for emulation of extension instructions, can be used to emulate extension registers and condition codes, again using the illegal instruction exception, which is triggered if an instruction references an unmapped extension operand.

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. 16-bit instructions have a reduced range of source and target core registers unless indicated otherwise. See Table 87 on page 173 for an alphabetic list of instructions. The following notation is used for the syntax of operations.

Table 28 Instruction Syntax Convention

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

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 or set the flags, or both.

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

The basecase ALU instructions are summarized in the following table:

Table 29 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
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

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 ARCompact based 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 ARCompact based processor ALU before the subtract operation.

Register-Register (TST, CMP & RCMP)

The General Operations Register-Register format on page 142 is implemented, where the destination field A is ignored, 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 on page 98)*
op	b,limm	*(b=source 1, c=limm=source 2. Redundant format see Conditional Register format on page 98)*
op	limm,c	*(limm=source 1, c=source 2. Redundant format see Conditional Register*

		format on page 98)
op	limm,limm	*(limm=source 1, limm=source 2.. Redundant format see* Conditional Register *format on page 98)*

Register with Unsigned 6-bit Immediate (TST, CMP & RCMP)

The General Operations Register with Unsigned 6-bit Immediate format on page 143 is implemented, where the destination field A is ignored, 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 on page 98.)*
op	limm,u6	*(limm=source 1, u6=source 2. Redundant format, see* Conditional Register with Unsigned 6-bit Immediate *format on page 98.)*

Register with Signed 12-bit Immediate (TST, CMP & RCMP)

The General Operations Register with Signed 12-bit Immediate format on page 143 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 (TST, CMP & RCMP)

The General Operations Conditional Register format on page 143 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 (TST, CMP & RCMP)

The General Operations Conditional Register with Unsigned 6-bit Immediate format on page 143 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 (BTST)

The General Operations Register-Register format on page 142 is implemented, where the destination field A is ignored, and provides the following redundant formats for BTST:

BTST	b,c	*(b=source 1, c=source 2.Redundant format see* Conditional Register *format on page* 100*)*
BTST	b,limm	*(b=source 1, c=limm=source 2. Redundant format see* Conditional Register with Unsigned 6-bit Immediate *format on page* 100*)*
BTST	limm,c	*(limm=source 1, c=source 2. Redundant format see* Conditional Register *format on page* 100*)*
BTST	limm,limm	*(limm=source 1, limm=source 2. Redundant format see* Conditional Register with Unsigned 6-bit Immediate *format on page* 100*)*

Register with Unsigned 6-bit Immediate (BTST)

The General Operations Register with Unsigned 6-bit Immediate format on page 143 is implemented, where the destination field A is ignored, 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 on page* 100*.)*
BTST	limm,u6	*(limm=source 1, u6=source 2. Redundant format, see* Conditional Register with Unsigned 6-bit Immediate *format on page* 100*.)*

Register with Signed 12-bit Immediate (BTST)

The General Operations Register with Signed 12-bit Immediate format on page 143 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 on page 100.)*
BTST	limm,s12	*(limm=source 1, s12=source 2. Redundant format, see Conditional Register with Unsigned 6-bit Immediate format on page 100.)*

Conditional Register (BTST)

The General Operations Conditional Register format on page 143 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 on page 100.)*
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 on page 100.)*

Conditional Register with Unsigned 6-bit Immediate (BTST)

The General Operations Conditional Register with Unsigned 6-bit Immediate format on page 144 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 (BSET, BCLR, BXOR & BMSK)

The General Operations Register-Register format on page 142 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 on page 101)*
op<.f>	a,limm,limm	*(if b=c=limm. Redundant format see Register with Unsigned 6-bit Immediate format on page 101)*
op<.f>	0,b,c	*(if a=0)*
op<.f>	0,limm,c	*(Redundant format, see Conditional Register format on page 101)*
op<.f>	0,b,limm	*(if a=0, c=limm. Redundant format see Register with Unsigned 6-bit Immediate format on page 101)*
op<.f>	0,limm,limm	*(if a=0, b=c=limm. Redundant format see Conditional Register with Unsigned 6-bit Immediate format on page 101)*

Register with Unsigned 6-bit Immediate (BSET, BCLR, BXOR & BMSK)

The General Operations Register with Unsigned 6-bit Immediate format on page 143 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 on page* 101*)*

Register with Signed 12-bit Immediate (BSET, BCLR, BXOR & BMSK)

The General Operations Register with Signed 12-bit Immediate format on page 143 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 on page* 101*)*
op<.f>	0,limm,s12	*(Redundant format see* Conditional Register with Unsigned 6-bit Immediate *format on page* 101*)*

Conditional Register (BSET, BCLR, BXOR & BMSK)

The General Operations Conditional Register format on page 143 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 on page* 101*)*
op<.cc><.f>	0,limm,limm	*(Redundant format see* Conditional Register with Unsigned 6-bit Immediate *format on page* 101*)*

Conditional Register with Unsigned 6-bit Immediate (BSET, BCLR, BXOR & BMSK)

The General Operations Conditional Register with Unsigned 6-bit Immediate format on page 144 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)*

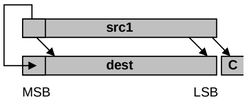
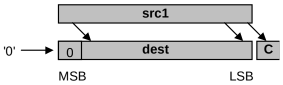
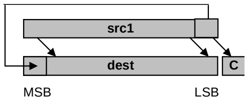
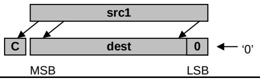
BXOR<.f>	a,b,c	*(a = b xor (1<<c))*
BXOR<.cc><.f>	b,b,c	*(b = b xor (1<<c))*
BXOR<.f>	a,b,u6	*(a = b xor (1<<u6))*
BXOR<.cc><.f>	b,b,u6	*(b = b xor (1<<u6))*

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 32 on page 105.

Table 30 Barrel Shift Operations

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

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)*

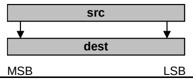
ROR<.f>	a,b,c	*(a = (b<<(31-c)):(b>>c))*
ROR<.f>	a,b,u6	*(a = (b<<(31-u6)):(b>>u6))*
ROR<.f>	b,b,s12	*(b = (b<<(31-s12)):(b>>s12))*
ROR<.cc><.f>	b,b,c	*(b = (b<<(31-c)):(b>>c))*
ROR<.cc><.f>	b,b,u6	*(b = (b<<(31-u6)):(b>>u6))*
ROR<.f>	a,limm,c	*(a = (limm<<(31-c)):(limm>>c))*
ROR<.f>	a,b,limm	*(a = (b<<(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 tables shows the move, extend, negate, rotate and shift operations.

Table 31 Single operand: moves and extends

Instruction	Operation	Description
MOV		Move

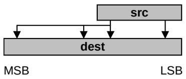
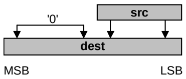
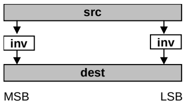
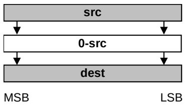
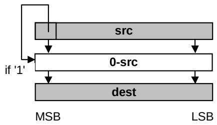
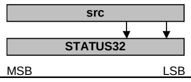
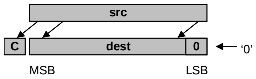
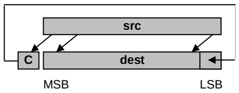
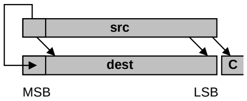
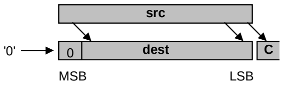
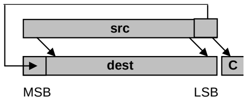
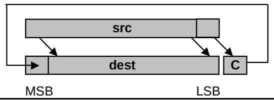
Instruction	Operation	Description
SEX		Sign extend byte or word
EXT		Zero extend byte or word
NOT		Logical NOT
NEG		Negate
ABS		Absolute
FLAG		Set flags

Table 32 Single operand: Rotates and Shifts

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

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 (MOV)

The General Operations Register-Register format on page 142 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 on page 106.)*
MOV<.f>	b,limm	*(b=destination, c=limm=source. Redundant format, see Conditional Register format on page 106.)*
MOV<.f>	0,c	*(b=limm, c=source. Redundant format, see Conditional Register format on page 106.)*
MOV<.f>	0,limm	*(if b=limm, b= c=limm=source. Redundant format, see Conditional Register format on page 106.)*

Register with Unsigned 6-bit Immediate (MOV)

The General Operations Register with Unsigned 6-bit Immediate format on page 143 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 on page 107.)*
MOV<.f>	0,u6	*(b=limm, u6=source. Redundant format, see Conditional Register with Unsigned 6-bit Immediate format on page 107.)*

Register with Signed 12-bit Immediate (MOV)

The General Operations Register with Signed 12-bit Immediate format on page 143 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 (MOV)

The General Operations Conditional Register format on page 143 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 (MOV)

The General Operations Conditional Register with Unsigned 6-bit Immediate format on page 144 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 (MOV)

The Mov/Cmp/Add with High Register, 0x0E, [0x00 - 0x03] format on page 156 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 (MOV)

The Move Immediate, 0x1B format on page 165 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 (FLAG)

The General Operations Register-Register format on page 142 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 on page* 107*.)*
FLAG	b,limm	*(a = ignored, b= ignored, c=limm=source. Redundant format, see* Conditional Register *format on page* 107*.)*

Register with Unsigned 6-bit Immediate (FLAG)

The General Operations Register with Unsigned 6-bit Immediate format on page 143 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 on page* 108*.)*

Register with Signed 12-bit Immediate (FLAG)

The General Operations Register with Signed 12-bit Immediate format on page 143 provides the following syntax for the FLAG instruction:

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

Conditional Register (FLAG)

The General Operations Conditional Register format on page 143 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 (FLAG)

The General Operations Conditional Register with Unsigned 6-bit Immediate format on page 144 provides the following syntax for the FLAG instruction:

FLAG<.cc>	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 as an encoding of the reverse subtract instruction using an unsigned 6-bit immediate value set to 0.

The syntax for negate operations is:

NEG_S	b,c	*(b = 0-c, reduced set of regs)*
NEG<.f>	a,b	*(encoded as RSUB<.f> a,b,0, where 0 is u6)*
NEG<.cc><.f>	b,b	*(encoded as RSUB<.cc><.f> b,b,0, where 0 is u6)*

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.

Table 33 Basecase ZOP instructions

Instruction	Operation	Description
NOP	No operation	Null Instruction
SLEEP	Sleep until interrupt or restart	Sleep
SWI	Raise Instruction Error exception	Software interrupt
BRK	Stop and flush processor pipeline	Breakpoint Instruction
TRAP0	raise an exception of value 0	Software Breakpoint Exception
TRAP_S	raise an exception of value n	User Exception
UNIMP_S	Unimplemented Instruction	Raise Instruction Error Exception
RTIE	Return from interrupt/exception	Return from interrupt/exception
SYNC	Synchronize with memory	Wait for all data-based memory transactions to complete

Zero operand instruction syntax is:

NOP	*(encoded as MOV 0,0)*
NOP_S	*(16-bit instruction form)*
SLEEP	u6
SWI	*(encoded as SWI 0, i.e. "swi" with u6=0)*

BRK_S	*(Breakpoint instruction, 16-bit format)*
BRK	*(Breakpoint instruction, 32-bit format)*
TRAP0	*(encoded as SWI 0, i.e. "swi" with u6=0)*
TRAP_S	u6
UNIMP_S	
RTIE	
SYNC	
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 ARCompact based processor from performing any instructions beyond the breakpoint. Since the breakpoint is a serializing instruction, the pipeline is also flushed upon decode of this instruction.

Sleep Instruction

The sleep mode is entered when the ARCompact based 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 is serializing which means the SLEEP instruction will complete and then flush the pipeline.

Software Interrupt Instruction

The execution of an undefined extension instruction in ARCompact based 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.

While the mnemonic SWI is available, its use is not recommended in the ARC 700 processor, TRAP0 should be used instead which raises a trap exception.

Trap Instruction

The instructions, TRAP_S and TRAP0, raise an exception and call any operating system in kernel mode. Traps can be raised from user or kernel modes.

Return from Interrupt/Exception Instruction

The return from interrupt/exception instruction, RTIE, allows exit from interrupt and exception handlers, and to allow the processor to switch from kernel mode to user mode.

Synchronize Instruction

The synchronize instruction, SYNC, waits until all data-based memory operations (LD, ST, EX, cache fills) have completed.

Branch Instructions

Due to the pipeline in the ARCompact based 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.

Table 34 Delay Slot Execution Modes

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 50 on page 135.

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.

Table 35 Branch on compare/test mnemonics

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 36 Branch on compare pseudo mnemonics, register-register

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)*

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.

Table 37 Branch on compare pseudo mnemonics, register-immediate

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)*

In the ARCtangent-A5 processor there are two delay slots due to the branch occurring a cycle later than other branches. Only one delay slot can be optionally executed by using the ".D" delay slot mode. The second delay slot is always nullified if the branch is taken.

Due to the ARC 600 processor pipeline there are 3 delay slots due to the branch occurring a cycle later than other branches. The first delay slot position can be optionally executed using the ".D" delay slot mode. The second and third delay slots are always nullified if the branch is taken.

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)*

Jump Instructions

Due to the pipeline in the ARCompact based processor, the jump instruction does not take effect immediately, but after a one-cycle delay. 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.

Table 38 Delay Slot Execution Modes

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

Table 39 Basecase Jump 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

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 Instruction

The ARCompact based 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 by the address defined in the LP instruction.

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

Register-Register (LP)

The General Operations Register-Register format on page 142 is not implemented for the LP instruction. Using this format will raise an Instruction Error exception.

Register with Unsigned 6-bit Immediate (LP)

The General Operations Register with Unsigned 6-bit Immediate format on page 143 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 redundant syntax for the LP instruction:

LP	u7	*(a = ignored, b= ignored, u7=source. Redundant format, see* Conditional Register with Unsigned 6-bit Immediate *format on page* 114*.)*

Register with Signed 12-bit Immediate (LP)

The General Operations Register with Signed 12-bit Immediate format on page 143 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 (LP)

The General Operations Conditional Register format on page 143 is not implemented for the LP instruction. Using this format will raise an Instruction Error exception.

Conditional Register with Unsigned 6-bit Immediate (LP)

The General Operations Conditional Register with Unsigned 6-bit Immediate format on page 144 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)*

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.

Table 40 Auxiliary Register Operations

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

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 format on page 142, the General Operations Register with Unsigned 6-bit Immediate format on page 143, and the General Operations Register with Signed 12-bit Immediate format on page 143 to provide the following syntax:

LR	b,[c]	
LR	b,[limm]	
LR	b,[u6]	
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 format on page 142, the General Operations Register with Unsigned 6-bit Immediate format on page 143, and the General Operations Register with Signed 12-bit Immediate format on page 143 to provide the following syntax:

SR	b,[c]	
SR	b,[limm]	*(c=limm)*
SR	b,[u6]	
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 or .AW (register updated pre memory transaction), and .AB (register updated post memory transaction). Addresses are interpreted as byte addresses unless the scaled address mode is used, as indicated by the address suffix .AS. The scaled address mode does not write back the result of the address calculation to the address source register.

NOTE Using the scaled address mode with 8-bit data size (LDB.AS or STB.AS) has undefined behavior and should not be used.

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 system may have extensions or customizations in this area, please see associated documentation.

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

LD<zz><.x><.di>	0,[limm]	*(Prefetch)*
LD<zz><.x><.aa><.di>	0,[b,c]	*(Prefetch)*
LD<zz><.x><.aa><.di>	0,[b,limm]	*(Prefetch)*
LD<zz><.x><.di>	0,[limm,c]	*(Prefetch)*
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)*

Prefetch

The PREFETCH instruction is provided as a synonym for a particular encoding of the LD instruction. The PREFETCH instruction is used to initiate a data cache load without writing to any core register.

The syntax for the PREFETCH instruction is:

PREFETCH<.aa>	[b,s9]	*(= ld<.aa> 0,[b,s9])*
PREFETCH	[limm,s9]	*(Redundant format, use PREFETCH [limm])*
PREFETCH	[limm]	*(= ld 0,[limm])*
PREFETCH<.aa>	[b,c]	*(= ld<.aa> 0,[b,c])*
PREFETCH<.aa>	[b,limm]	*(= ld<.aa> 0,[b,limm])*
PREFETCH	[limm,c]	*(= ld<.aa> 0,[limm,c])*

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 ARCompact based 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.

Atomic Exchange

An atomic exchange operation, EX, is provided as a primitive for multiprocessor synchronization allowing the creation of semaphores in shared memory.

Two forms are provided: an uncached form (using the .DI directive) for synchronization between multiple processors, and a cached form for synchronization between processes on a single-processor system.

The EX instruction exchanges the contents of the specified memory location with the contents of the specified register. This operation is atomic in that the memory system ensures that the memory read and memory write cannot be separated by interrupts or by memory accesses from another processor.

The instruction syntax for the atomic exchange instruction is:

EX<.di>	b,[c]
EX<.di>	b,[limm]
EX<.di>	b,[u6]

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 or set the flags or both.

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 Instructions Library

The optional instructions library consists of a number of components that can be used to add functionality to the ARCtangent-A5 processor. These components are function units, which are interfaced to the ARCtangent-A5 processor through the use of extension instructions or registers.

The optional instructions 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 Normalze and Swap instructions are built in to the ARC 700 processor. The multiply instruction, however, is optional.

Summary of Optional Instructions Library

The library currently consists of the following components:

- 32 bit Multiplier
- Normalize (find-first-bit) instruction
- Swap instruction

Table 41 Dual Operand Optional Instructions for ARCtangent-A5 and ARC 600

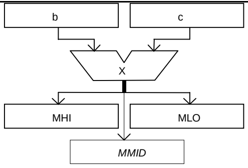
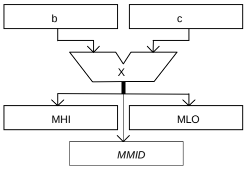
Instruction	Operation	Description
MUL64		Signed 32x32 Multiply
MULU64		Unsigned 32x32 Multiply

Table 42 Dual Operand Optional Instructions for ARC 700

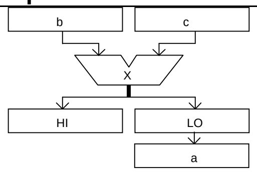
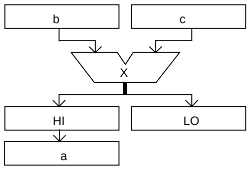
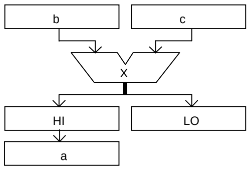
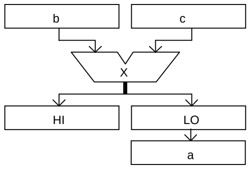
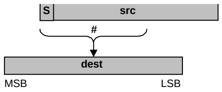
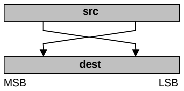
Instruction	Operation	Description
MPY		32 X 32 signed multiply
MPYH		32 X 32 signed multiply
MPYHU		32 X 32 unsigned multiply
MPYU		32 X 32 unsigned multiply

Table 43 Single Operand Optional Instructions

Instruction	Operation	Description
NORM		Normalize (find-first-bit)
SWAP		Exchange upper and lower 16 bits

Multiply 32 X 32, Special Result Registers

The scoreboarded 32x32 multiplier performs signed or unsigned multiply. The full 64-bit result is available to be read from special result registers in 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 ARCompact based instruction, the processor will stall until the multiply has finished. The destination is always ignored for the multiply instruction and thus the syntax for the multiply instructions can optionally supply a "0" as the destination register. Two instructions are provided to perform either a 32x32 signed multiply (MUL64) or a 32x32 unsigned multiply (MULU64).

Register-Register (MUL64 & MULU64)

The General Operations Register-Register format on page 142 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 on page 122)*

MUL64	<0,>b,limm	*(a = limm, b limm, c = source 2. Redundant format see* Conditional Register *format on page* 122*)*
MUL64	<0,>limm,c	*(a = limm, b = source 1, c = limm. Redundant format see* Conditional Register *format on page* 122*)*
MUL64	<0,>limm,limm	*(a = limm, b = limm, c = limm. Redundant format see* Conditional Register *format on page* 122*)*
MULU64	<0,>b,c	*(a = limm, b = source 1, c = source 2. Redundant format see* Conditional Register *format on page* 122*)*
MULU64	<0,>b,limm	*(a = limm, b = limm, c = source 2. Redundant format see* Conditional Register *format on page* 122*)*
MULU64	<0,>limm,c	*(a = limm, b = source 1, c = limm. Redundant format see* Conditional Register *format on page* 122*)*
MULU64	<0,>limm,limm	*(a = limm, b = limm, c = limm. Redundant format see* Conditional Register *format on page* 122*)*

Register with Unsigned 6-bit Immediate (MUL64 & MULU64)

The General Operations Register with Unsigned 6-bit Immediate format on page 143 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 on page* 123*)*
MUL64	<0,>limm,u6	*(a = limm, b = limm, u6 = source 2. Redundant format see* Conditional Register with Unsigned 6-bit Immediate *format on page* 123*)*
MULU64	<0,>b,u6	*(a = limm, b = source 1, u6 = source 2. Redundant format see* Conditional Register with Unsigned 6-bit Immediate *format on page* 123*)*
MUL64	<0,>limm,u6	*(a = limm, b = limm, u6 = source 2. Redundant format see* Conditional Register with Unsigned 6-bit Immediate *format on page* 123*)*

Register with Signed 12-bit Immediate (MUL64 & MULU64)

The General Operations Register with Signed 12-bit Immediate format on page 143 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 (MUL64 & MULU64)

The General Operations Conditional Register format on page 143 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 (MUL64 & MULU64)

The General Operations Conditional Register with Unsigned 6-bit Immediate format on page 144 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 (MUL64 & MULU64)

The unsigned multiply operation does not have a 16-bit instruction equivalent. The General Register Format Instructions, 0x0F, [0x00 - 0x1F] format on page 157 provides the following syntax for the signed multiply

MUL64_S	<0,>b,c

Multiply 32 X 32, Any Result Register

The scoreboarded 32x32 multiplier performs signed or unsigned multiply. The higher or lower 32-bit portion of the full 64-bit result can be written to any core register. The multiply is scoreboarded in such a way that if a multiply is being carried out, and if the result registers is required by another ARCompact based instruction, the processor will stall until the multiply has finished. Four instructions are provided to perform the 32x32 multiply and write either the signed low (MPY), signed high (MPYH), unsigned low (MPYU) or unsigned high (MPYHU) result into a specified core register.

The syntax for the multiply instruction is:

MPYH<.f>	a,b,c
MPYH<.f>	a,b,u6
MPYH<.f>	b,b,s12
MPYH<.cc><.f>	b,b,c
MPYH<.cc><.f>	b,b,u6
MPYH<.f>	a,limm,c
MPYH<.f>	a,b,limm
MPYH<.cc><.f>	b,b,limm
MPYH<.f>	0,b,c
MPYH<.f>	0,b,u6
MPYH<.cc><.f>	0,limm,c

MPYH<.f>	a,b,c
MPYH<.f>	a,b,u6
MPYH<.f>	b,b,s12
MPYH<.cc><.f>	b,b,c
MPYH<.cc><.f>	b,b,u6
MPYH<.f>	a,limm,c
MPYH<.f>	a,b,limm
MPYH<.cc><.f>	b,b,limm
MPYH<.f>	0,b,c
MPYH<.f>	0,b,u6
MPYH<.cc><.f>	0,limm,c

MPYU<.f>	a,b,c
MPYU<.f>	a,b,u6
MPYU<.f>	b,b,s12
MPYU<.cc><.f>	b,b,c
MPYU<.cc><.f>	b,b,u6
MPYU<.f>	a,limm,c
MPYU<.f>	a,b,limm
MPYU<.cc><.f>	b,b,limm
MPYU<.f>	0,b,c
MPYU<.f>	0,b,u6
MPYU<.cc><.f>	0,limm,c

MPYHU<.f>	a,b,c
MPYHU<.f>	a,b,u6
MPYHU<.f>	b,b,s12
MPYHU<.cc><.f>	b,b,c
MPYHU<.cc><.f>	b,b,u6
MPYHU<.f>	a,limm,c
MPYHU<.f>	a,b,limm
MPYHU<.cc><.f>	b,b,limm
MPYHU<.f>	0,b,c

MPYHU<.f>	0,b,u6
MPYHU<.cc><.f>	0,limm,c

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 ARCompact based instruction cycles.

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.

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

Extended Arithmetic Library

The extended arithmetic instruction library consists of a number of components that can be used to add functionality to the ARCtangent-A5 processor. These components are function units, which are interfaced to the ARCtangent-A5 processor through the use of extension instructions or registers.

The extended arithmetic instruction 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 extensions library is built in to the ARC 700 processor

The extended arithmetic instructions are targeted at telephony applications requiring bit-accuracy for speech coders and audio applications requiring extended precision.

Summary of Extended Arithmetic Library Instructions

The following notation is used for the operation of the extended arithmetic instructions.

Table 44 Extended Arithmetic Operation Notation

operand.high	The top 16-bits of the operand.
operand.low	The bottom 16-bits of the operand.
function(*operand*).high	The high part of the result of the function.
accumulator.high	The high part of the accumulator.
rnd_{16} (*operand*)	= round operand to 16-bits
sat_{16} (*operand*)	= saturate operand to 16-bits
sat_{32} (*operand*)	= saturate operand to 32-bits
A*n*	Internal accumulator *n*

Table 45 Extended Arithmetic Dual Operand Instructions

Instruction	Operation	Description
ADDS	$a \leftarrow \text{sat}_{32}(b+c)$	Add and saturate.
SUBS	$a \leftarrow \text{sat}_{32}(b-c)$	Subtract and saturate.

DIVAW	b_temp ← b<<1
	if (b_temp>=c)
	a ← ((b_temp-c)+1)
	else
	a ← b
		Division assist.

ASLS	$a \leftarrow \text{sat}_{32}(b<<c)$	Arithmetic shift left and saturate. Supports negative shift values for right shift.
ASRS	$a \leftarrow \text{sat}_{32}(b>>c)$	Arithmetic shift right and saturate. Supports -ve shift values for left shift.

Instruction	Operation	Description
ADDSDW	a ← sat_{16}(b.high+c.high): sat_{16}(b.low+c.low)	Dual 16-bit add and saturate.
SUBSDW	a ← sat_{16}(b.high-c.high): sat_{16}(b.low-c.low)	Dual16-bit subtract and saturate.

Table 46 Extended Arithmetic Single Operand Instructions

Instruction	Operation	Description
SAT16	b ← sat_{16}(c)	Saturate 32-bit input to 16-bits
RND16	b ← sat_{32}(c+0x00008000)&0xffff0000	Round 32-bit input to 16-bits
ABSSW	b ← sat_{16}(abs(c.low))	Absolute value of 16-bit input
ABSS	b ← sat_{32}(abs(c))	Absolute value of 32-bit input
NEGSW	b ← sat_{16}(neg(c.low))	Negate and saturate 16-bit input
NEGS	b ← sat_{32}(neg(c))	Negate and saturate 32-bit input

Add with Saturation

The ADD instruction is extended to provide saturation logic. A dual-word form is also provided. The syntax for ADDS is:

```
ADDS<.f>                a,b,c
ADDS<.f>                b,b,u6
ADDS<.f>                c,b,s12
ADDS<.cc><.f>           b,b,c
ADDS<.cc><.f>           b,b,u6
ADDS<.f>                a,limm,c
ADDS<.f>                a,b,limm
ADDS<.cc><.f>           b,b,limm
ADDS<.f>                0,b,c
ADDS<.f>                0,b,u6
ADDS<.f>                0,b,limm
ADDS<.cc><.f>           0,limm,c

ADDSDW<.f>              a,b,c
ADDSDW<.f>              b,b,u6
ADDSDW<.f>              c,b,s12
ADDSDW<.cc><.f>         b,b,c
ADDSDW<.cc><.f>         b,b,u6
ADDSDW<.f>              a,limm,c
ADDSDW<.f>              a,b,limm
```

ADDSDW<.cc><.f>	b,b,limm
ADDSDW<.f>	0,b,c
ADDSDW<.f>	0,b,u6
ADDSDW<.f>	0,b,limm
ADDSDW<.cc><.f>	0,limm,c

Subtract with Saturation

The SUB instruction is extended to provide saturation logic. A dual-word form is also provided. The syntax for SUBS is:

SUBS<.f>	a,b,c
SUBS<.f>	b,b,u6
SUBS<.f>	c,b,s12
SUBS<.cc><.f>	b,b,c
SUBS<.cc><.f>	b,b,u6
SUBS<.f>	a,limm,c
SUBS<.f>	a,b,limm
SUBS<.cc><.f>	b,b,limm
SUBS<.f>	0,b,c
SUBS<.f>	0,b,u6
SUBS<.f>	0,b,limm
SUBS<.cc><.f>	0,limm,c

SUBSDW<.f>	a,b,c
SUBSDW<.f>	b,b,u6
SUBSDW<.f>	c,b,s12
SUBSDW<.cc><.f>	b,b,c
SUBSDW<.cc><.f>	b,b,u6
SUBSDW<.f>	a,limm,c
SUBSDW<.f>	a,b,limm
SUBSDW<.cc><.f>	b,b,limm
SUBSDW<.f>	0,b,c
SUBSDW<.f>	0,b,u6
SUBSDW<.f>	0,b,limm
SUBSDW<.cc><.f>	0,limm,c

Negate with Saturation

The negate instruction is extended to provide saturation logic. A single-word form is also provided. The syntax for NEGS is:

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

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

Absolute with Saturation

The absolute instruction returns the absolute value of a number and saturates. A single-word form is also provided. The syntax for ABSS is:

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

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

Round

The round instruction, RND16, rounds to a 16-bit number. The syntax for RND16 is:

RND16<.f>	b,c
RND16<.f>	b,u6
RND16<.f>	b,limm
RND16<.f>	0,c
RND16<.f>	0,u6
RND16<.f>	0,limm

Saturate

The saturate instruction, SAT16, provides the saturated value of a 16-bit number. The syntax for SAT16 is:

SAT16<.f>	b,c
SAT16<.f>	b,u6
SAT16<.f>	b,limm
SAT16<.f>	0,c
SAT16<.f>	0,u6
SAT16<.f>	0,limm

Positive/Negative Barrel Shift with Saturation

Shift instructions operate with both positive and negative shifts (reverse shift) and provide saturation according to ETSI/ITU-T definitions. The syntax for the positive and negative shifts is:

ASLS<.f>	a,b,c
ASLS<.f>	b,b,u6
ASLS<.f>	c,b,s12
ASLS<.cc><.f>	b,b,c
ASLS<.cc><.f>	b,b,u6
ASLS<.f>	a,limm,c
ASLS<.f>	a,b,limm
ASLS<.cc><.f>	b,b,limm
ASLS<.f>	0,b,c
ASLS<.f>	0,b,u6
ASLS<.f>	0,b,limm
ASLS<.cc><.f>	0,limm,c
ASRS<.f>	a,b,c

ASRS<.f>	b,b,u6
ASRS<.f>	c,b,s12
ASRS<.cc><.f>	b,b,c
ASRS<.cc><.f>	b,b,u6
ASRS<.f>	a,limm,c
ASRS<.f>	a,b,limm
ASRS<.cc><.f>	b,b,limm
ASRS<.f>	0,b,c
ASRS<.f>	0,b,u6
ASRS<.f>	0,b,limm
ASRS<.cc><.f>	0,limm,c

Division Assist

DIVAW is a division accelerator used in the division algorithm as described by the ITU and ETSI. Repeated execution of DIVAW fifteen times implements a 16-bit conditional add-subtract division algorithm. The syntax for the DIVAW instruction is:

DIVAW<.f>	a,b,c
DIVAW<.f>	b,b,u6
DIVAW<.f>	c,b,s12
DIVAW<.cc><.f>	b,b,c
DIVAW<.cc><.f>	b,b,u6
DIVAW<.f>	a,limm,c
DIVAW<.f>	a,b,limm
DIVAW<.cc><.f>	b,b,limm
DIVAW<.f>	0,b,c
DIVAW<.f>	0,b,u6
DIVAW<.f>	0,b,limm
DIVAW<.cc><.f>	0,limm,c

This page is intentionally left blank.

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. Instruction Set Summary on page 93 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 Instruction Set Details on page 173.

A complete list of the major opcodes is shown in Table 47 on page 133.

Table 47 Major opcode Map, 32-bit and 16-Bit instructions

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 a,b,c	User 32-bit extension instructions	32-bit
0x09	op <market specific>	ARC market-specific extension instructions	32-bit
0x0A	op <market specific>	ARC market-specific extension instructions	32-bit
0x0B	op <market specific>	ARC market-specific extension instructions	32-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_Sb,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

Encoding Notation

This chapter shows the full encoding details along with the shortened form, represented by a set of characters, used in Instruction Set Details on page 173.The list of syntax conventions is shown in Table 28 on page 93.

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.

The notation used for the encoding is shown in Table 48 on page 134 and Table 49 on page 134.

Table 48 Key for 32-bit Addressing Modes and Encoding Conventions

Encoding Character	Encoding Field	Syntax
I	I[4:0]	instruction major opcode
i	i[n:0]	instruction sub opcode
A	A[5:0]	destination register
b	B[2:0]	lower bits source/destination register
B	B[5:3]	upper bits source/destination register
C	C[5:0]	source/destination register
Q	Q[4:0]	condition code
u	U[*n*:0]	unsigned immediate (number is bit field size)
s	S[*n*:0]	lower bits signed immediate (number is bit field size)
S	S[*m*:*n*+1]	upper bits signed immediate (number is bit field size)
T	S[24:21]	upper bits signed immediate (branch unconditionally far)
P	P[1:0]	operand format
M	M	conditional instruction operand mode
N	N	<.d> delay slot mode
F	F	Flag Setting
R	R	Reserved
D	Di	<.di> direct data cache bypass
A	A	<.aa> address writeback mode
Z	Z	<.zz> data size
X	X	<.x> sign extend

Table 49 Key for 16-bit Addressing Modes and Encoding Conventions

Encoding Character	Encoding Field	Syntax
I	I[4:0]	instruction major opcode
i	i[*n*:0]	instruction sub-opcode
a	a[2:0]	source/destination register (r0-3,r12-15)
b	b[2:0]	source/destination register (r0-3,r12-15)
c	c[2:0]	source/destination register (r0-3,r12-15)
h	h[2:0]	source/destination register high (r0-r63)

Encoding Character	Encoding Field	Syntax
H	h[5:3]	source/destination register high (r0-r63)
u	u[*n*:0]	unsigned immediate (number is bit field size)
s	s[*n*:0]	signed immediate (number is bit field size)

Condition Code Tests

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

Table 50 Condition codes

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

Branch Jump Delay Slot Modes

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

Table 51 Delay Slot Modes

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

Load Store Address Write-back Modes

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

Table 52 Address Write-back Modes

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

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.

Table 53 Direct to Memory Bypass Mode

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

Load Store Data Size Mode

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

Table 54 Load Store Data Size Mode

ZZ Code	ZZ Suffix	Access mode
00		Default, Long word
01	B	Byte
10	W	Word
11		*Reserved* Will raise an Instruction Error exception for the ARC 700 processor. Undefined behavior for the ARCtangent-A5 and ARC 600 processor.

Load Data Extend Mode

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

Table 55 Load Data Extend Mode

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

Use of Reserved Encodings

In a given format, one or more bits of an encoding can be marked as *Reserved*. In some formats, an entire field may be reserved, such as when a register field is present in a given format but is not used in the particular opcode (such as a MOV in format 0x04, which does not use source 1).

The presence of reserved bits has the following effect:

- The processor will ignore reserved bits. It will not generate an exception on an instruction based on the value assigned to reserved bits, the functionality of the instruction will not be affected by them.
- The reserved bits should be set to 0 when encoding instructions. This permits future revisions of the architecture to assign new functionality to encodings that set bits currently reserved.

Use of Illegal Encodings

There are two major categories of illegal encodings:

- Reserved ranges of fields
- Illegal combinations of fields

Reserved Ranges of Fields

A given field can support a range of values, not all of which are used for supported functions. For example, within most major formats there are opcodes that are reserved for future expansion. These are now to be re-defined as *Illegal*.

If such an field is used, an Instruction Error exception will result.

Illegal Combinations of Fields

Fields are normally orthogonal, but certain combinations or values between 2 or more fields create an instruction whose behavior is either nonsense or cannot be realized.

For example, the EX instruction, in format 0x04, exchanges one source (a memory location) with another (a register). However, format 0x04 has sub-format that allow the source register to be a constant. For the EX instruction sub-formats such as these do not make sense.

In such cases, nonsense combinations will raise an Instruction Error exception.

Branch Conditionally, 0x00, [0x0]

The target address is 16-bit aligned to target 16-bit aligned instructions. See Table 50 on page 135 for information on condition code test encoding, and Table 51 on page 135 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
I[4:0]					S[10:1]										0	S[20:11]										N	Q[4:0]				
0	0	0	0	0	s	s	s	s	s	s	s	s	s	s	0	S	S	S	S	S	S	S	S	S	S	N	Q	Q	Q	Q	Q

Values 0x12 to 0x1F in the condition code field, Q, will raise an Instruction Error exception. Condition code tests, 0x10 an 0x11 are used to test the extended arithmetic saturation flag.

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 51 on page 135 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
I[4:0]					S[10:1]										1	S[20:11]										N	R	S[24:21]			
0	0	0	0	0	s	s	s	s	s	s	s	s	s	s	1	S	S	S	S	S	S	S	S	S	S	N	0	T	T	T	T

A value of 1 in the reserved field, R, will raise an Instruction Error exception.

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 51 on page 135 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
I[4:0]					B[2:0]			S[7:1]							1	S8	B[5:3]			C[5:0]						N	0	i[3:0]			
0	0	0	0	1	b	b	b	s	s	s	s	s	s	s	1	S	B	B	B	C	C	C	C	C	C	N	0	i	i	i	i

Values 0x6 to 0xD in the sub-opcode field, i, will raise an Instruction Error exception.

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)*

Table 56 Branch on compare/bit test register-register

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
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

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 51 on page 135 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
I[4:0]					B[2:0]			S[7:1]							1	S8	B[5:3]			U[5:0]						N	1	i[3:0]			
0	0	0	0	1	b	b	b	s	s	s	s	s	s	s	1	S	B	B	B	U	U	U	U	U	U	N	1	i	i	i	i

Values 0x6 to 0xD in the sub-opcode field, i, will raise an Instruction Error exception.

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)*

Table 57 Branch Conditionally/bit test on register-immediate

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
…			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

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

The target address must be 32-bit aligned. See Table 50 on page 135 for information on condition code test encoding, and Table 51 on page 135 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
I[4:0]					S[10:2]									0	0	S[20:11]										N	Q[4:0]				
0	0	0	0	1	s	s	s	s	s	s	s	s	s	0	0	S	S	S	S	S	S	S	S	S	S	N	Q	Q	Q	Q	Q

Values 0x12 to 0x1F in the condition code field, Q, will raise an Instruction Error exception. Condition code tests, 0x10 an 0x11 are used to test the extended arithmetic saturation flag.

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 51 on page 135 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
					S[10:2]									1	0	S[20:11]										N	R	S[24:21]			
0	0	0	0	1	s	s	s	s	s	s	s	s	s	1	0	S	S	S	S	S	S	S	S	S	S	N	0	T	T	T	T

The reserved field, R, is ignored by the processor.

Syntax:

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

Load Register with Offset, 0x02

See Table 52 on page 136, Table 53 on page 136, Table 54 on page 136 and Table 55 on page 137 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
I[4:0]					B[2:0]			S[7:0]								S8	B[5:3]			Di	a	a	Z	Z	X	A[5:0]					
0	0	0	1	0	b	b	b	s	s	s	s	s	s	s	s	S	B	B	B	D	a	a	Z	Z	X	A	A	A	A	A	A

Extension core registers and the program counter (PCL) are not permitted to be the destination of a load instruction. Values 0x20 to 0x3B, 0x3D and 0x3F in the destination register field, A, will raise an Instruction Error exception.

The loop counter register (LP_COUNT) is not permitted to be the destination of a load instruction, A=0x3C, and will raise a Privilege Violation exception.

A value of 0x3 in the data size mode field, ZZ, will raise an Instruction Error exception.

The sign extension field, X, should not be set when the load is of longword data ZZ=0x0. This combination will raise an Instruction Error exception.

Using incrementing addressing modes in combination with a long immediate values as the base register is illegal. Values 0x1 and 0x2 in the addressing mode field, a, and a value of 0x3E in the base register field, B, will raise an Instruction Error exception.

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

Store Register with Offset, 0x03

See Table 52 on page 136, Table 53 on page 136 and Table 54 on page 136 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
I[4:0]					B[2:0]			S[7:0]								S8	B[5:3]			C[5:0]						Di	A	A	Z	Z	R
0	0	0	1	1	b	b	b	s	s	s	s	s	s	s	s	S	B	B	B	C	C	C	C	C	C	D	a	a	Z	Z	0

A value of 0x3 in the data size mode field, ZZ, will raise an Instruction Error exception.

Using incrementing addressing modes in combination with a long immediate values as the base register is illegal. Values 0x1 and 0x2 in the addressing mode field, a, and a value of 0x3E in the base register field, B, will raise an Instruction Error exception.

The reserved field, R, is ignored by the processor.

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]

Operand Format Indicators

There are four operand formats (P[1:0]) in major opcode 0x04 which are used to specify the format of operands that are used by the instructions. The conditional format has a sub operand format indicator M. The operand format indicators are summarized in Table 58 on page 142.

Table 58 Operand Format Indicators

Operand format Name	Operand Format P[1:0]	Sub Operand Format M	Comment
REG_REG	00	N/A	Destination and both sources are registers
REG_U6IMM	01	N/A	Source 2 is a 6-bit unsigned immediate
REG_S12IMM	10	N/A	Source 2 is a 12-bit signed immediate
COND_REG	11	0	Conditional instruction. Destination (if any) is source 1. Source 2 is a register
COND_REG_U6IMM	11	1	Conditional instruction. Destination (if any) is source 1. Source 2 is a 6-bit unsigned immediate

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
I[4:0]					B[2:0]			P[1:0]		i[5:0]						F	B[5:3]			C[5:0]						A[5:0]					
0	0	1	0	0	b	b	b	0	0	i	i	i	i	i	i	F	B	B	B	C	C	C	C	C	C	A	A	A	A	A	A

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	*(Redundant format, see General Operations Conditional Register format on page 143)*
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
I[4:0]					B[2:0]			P[1:0]		i[5:0]						F	B[5:3]			U[5:0]						A[5:0]					
0	0	1	0	0	b	b	b	0	1	i	i	i	i	i	i	F	B	B	B	U	U	U	U	U	U	A	A	A	A	A	A

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
I[4:0]					B[2:0]			P[1:0]		i[5:0]						F	B[5:3]			S[5:0]						S[11:6]					
0	0	1	0	0	b	b	b	1	0	i	i	i	i	i	i	F	B	B	B	s	s	s	s	s	s	S	S	S	S	S	S

A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and will raise an Instruction Error exception.

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
I[4:0]					B[2:0]			P[1:0]		i[5:0]						F	B[5:3]			C[5:0]						M	Q[4:0]				
0	0	1	0	0	b	b	b	1	1	i	i	i	i	i	i	F	B	B	B	C	C	C	C	C	C	0	Q	Q	Q	Q	Q

A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and will raise an Instruction Error exception.

Values 0x12 to 0x1F in the condition code field, Q, will raise an Instruction Error exception. Condition code tests, 0x10 an 0x11 are used to test the extended arithmetic saturation flag.

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
I[4:0]					B[2:0]			P[1:0]		i[5:0]						F	B[5:3]			U[5:0]						M	Q[4:0]				
0	0	1	0	0	b	b	b	1	1	i	i	i	i	i	i	F	B	B	B	U	U	U	U	U	U	1	Q	Q	Q	Q	Q

A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and will raise an Instruction Error exception.

Values 0x12 to 0x1F in the condition code field, Q, will raise an Instruction Error exception. Condition code tests, 0x10 an 0x11 are used to test the extended arithmetic saturation flag.

Syntax:

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

Long Immediate with General Operations

Any 6-bit register field in an instruction can indicate that long immediate data is used. The long immediate indicator (r62) can be used multiple times in an instruction. When a source register is set to r62, an explicit long immediate value will follow the instruction word.

When a destination register is set to r62 there is no destination for the result of the instruction so the result is discarded. Any flag updates will still occur according to the set flags directive (.F or implicit in the instruction).

If the long immediate indicator is used in both a source and destination operand the following long immediate value will be used as the source operand and the result will be discarded as expected.

When an instruction uses long immediate, the first long-word instruction is the instruction that contains the long immediate data indicator (register r62). The second long-word instruction 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

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

Table 59 ALU Instructions

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

Sub-opcode i field (6 bits)	Instruction	Operation	Description
0x07	XOR	a ← b exclusive-or c	logical bitwise exclusive-OR
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 on page 106
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
0x19	SUB3	a ← b - (c << 3)	subtract with left shift by 3
0x1A	MPY	a ← (a X c).low	32 X 32 signed multiply
0x1B	MPYH	a ← (a X c).high	32 X 32 signed multiply
0x1C	MPYHU	a ← (a X c).high	32 X 32 unsigned multiply
0x1D	MPYU	a ← (a X c).low	32 X 32 unsigned multiply
0x1E		Instruction Error	Reserved
0x1F		Instruction Error	Reserved

Special Format Instructions, 0x04, [0x20 - 0x3F]

Table 60 Special Format Instructions

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		Instruction Error	Reserved

Sub-opcode I field (6 bits)	Instruction	Operation	Description
0x25		Instruction Error	Reserved
0x26		Instruction Error	Reserved
0x27		Instruction Error	Reserved

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 on page 115
0x2B	SR	aux.reg[c] ← b	store to auxiliary register. See section Store to Auxiliary Register on page 115
0x2C		Instruction Error	Reserved
0x2D		Instruction Error	Reserved
0x2E		Instruction Error	Reserved
0x2F	SOPs	A field is sub-opcode2	See section:
0x30...0x37	LD	Load register-register	See section Load Register-Register, 0x04, [0x30 - 0x37] on page 147
0x38		Instruction Error	Reserved
...		Instruction Error	Reserved
0x3F		Instruction Error	Reserved

Move and Compare Instructions, 0x04, [0x0A - 0x0D] and 0x04, [0x11]

The move and compare instructions (MOV, TST, CMP, RCMP and BTST) use two operands. The destination field A is ignored for these instructions and instead the B and C fields are used accordingly.

Jump and Jump-and-Link Conditionally, 0x04, [0x20 - 0x23]

The jump (Jcc) and jump-and link (JLcc) instructions are specially encoded in major opcode 0x04 in that the B field is reserved and should be set to 0x0. Any value in the B field is ignored by the processor. The destination register, A field, should also be set to 0x0 when the operand mode, P, is 0x0 or 0x1. In the case where P is 0x0 or 0x1, any value in the A field is ignored.

When using ILINK1 or ILINK2 the flag setting field, F, is always encoded as 1 for these instructions.

If the ILINK1 or ILINK2 registers are used without the flag setting field being set an Instruction Error exception will be raised. If the flag setting field, F, is set without using the ILINK1 or ILINK2 register, an Instruction Error exception will be raised.

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.

Using an immediate value in the destination register field is not allowed for the ARCtangent-A5 or ARC 600 processor.

Using an immediate value in the destination register field causes a prefetch with the ARC 700 processor.

See Table 52 on page 136, Table 53 on page 136 and Table 54 on page 136 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
I[4:0]					B[2:0]			A	A	1	1	0	Z	Z	X	Di	B[5:3]			C[5:0]						A[5:0]					
0	0	1	0	0	b	b	b	a	a	1	1	0	Z	Z	X	D	B	B	B	C	C	C	C	C	C	A	A	A	A	A	A

Extension core registers and the program counter (PCL) are not permitted to be the destination of a load instruction. Values 0x20 to 0x3B, 0x3D and 0x3F in the destination register field, A, will raise an Instruction Error exception.

The loop counter register (LP_COUNT) is not permitted to be the destination of a load instruction, A=0x3C, and will raise a Privilege Violation exception.

A value of 0x3 in the data size mode field, ZZ, will raise an Instruction Error exception.

The sign extension field, X, should not be set when the load is of longword data ZZ=0x0. This combination will raise an Instruction Error exception.

Using incrementing addressing modes in combination with a long immediate values as the base register is illegal. Values 0x1 and 0x2 in the addressing mode field, a, and a value of 0x3E in the base register field, B, will raise an Instruction Error exception.

Syntax:

```
LD<zz><.x><.aa><.di>    a,[b,c]
LD<zz><.x><.aa><.di>    a,[b,limm]
LD<zz><.x><.di>         a,[limm,c]
LD<zz><.x><.aa><.di>    0,[b,c]         (Prefetch, a=limm)
LD<zz><.x><.aa><.di>    0,[b,limm]      (Prefetch, a=limm, c=limm)
LD<zz><.x><.di>         0,[limm,c]      (Prefetch, a=limm, b=limm)
```

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.

Table 61 Single Operand Instructions

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

Sub-opcode2 A field (6 bits)	Instruction	Operation	Description
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	EX	b ← mem[c];
		mem[c] ← b
			Atomic Exchange

0x0D		Instruction Error	Reserved
...		Instruction Error	Reserved
0x3E		Instruction Error	Reserved
0x3F	ZOPs	B field is sub-opcode3	See Zero operand (ZOP) table

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.

Table 62 Zero Operand Instructions

Sub-opcode3 B field (6 bits)	Instruction	Operation	Description
0x00		Instruction Error	Reserved
0x01	SLEEP	Sleep	Sleep
0x02	SWI/TRAP0	Swi	Software interrupt
0x03	SYNC	Synchronize	Wait for all data-based memory transactions to complete
0x04	RTIE	Return	Return from interrupt/exception
0x05	BRK	Breakpoint	Breakpoint instruction
0x06		Instruction Error	Reserved
...		Instruction Error	Reserved
0x3F		Instruction Error	Reserved

Syntax:

```
SLEEP
SLEEP    u6
SLEEP    c
```

SWI

TRAP0

SYNC

RTIE

BRK *(Encoded as REG_U6IMM, but the redundant REG_REG format is also valid. See Table 58 on page 142)*

32-bit Extension Instructions, 0x05 - 0x08

Any instruction opcodes that are not implemented will raise an Instruction Error exception.

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

Table 63 Summary of Extension Instruction Encoding

Major Opcode [31:27]	Sub Opcode1 [21:16] i-field	Sub Opcode2 [5:0] a-field	Sub Opcode3 [14:12]: [26:24] b-field	Instruction Usage
0x05	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
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
0x08	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

Extension ALU Operation, Register-Register

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

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

Figure 84 Extension ALU Operation, register-register

Extension ALU Operation, Register with Unsigned 6-bit Immediate

Using major opcode 0x05 as an example, the syntax of op<.f> a,b,u6 is encoded as shown below.

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

Figure 85 Extension ALU Operation, register with unsigned 6-bit immediate

Extension ALU Operation, Register with Signed 12-bit Immediate

Using major opcode 0x05 as an example, the syntax of op<.f> b,b,s12 is encoded as shown in the following diagram.

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

Figure 86 Extension ALU Operation, register with signed 12-bit immediate

A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and will raise an Instruction Error exception.

Extension ALU Operation, Conditional Register

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

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

Figure 87 Extension ALU Operation, conditional register

A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and will raise an Instruction Error exception.

Values 0x12 to 0x1F in the condition code field, Q, will raise an Instruction Error exception. Condition code tests, 0x10 an 0x11 are used to test the extended arithmetic saturation flag.

Extension ALU Operation, Conditional Register with Unsigned 6-bit Immediate

Using major opcode 0x05 as an example, 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
I[4:0]					B[2:0]			P[1:0]		I[5:0]						F	B[5:3]			U[5:0]						M	Q[4:0]				
0	0	1	0	1	b	b	b	1	1	i	i	i	i	i	i	F	B	B	B	U	U	U	U	U	U	1	Q	Q	Q	Q	Q

Figure 88 Extension ALU Operation, cc register with unsigned 6-bit immediate

A value of 0x2F in the sub-opcode field, i, indicates a single operand instruction which is invalid for this operand mode and will raise an Instruction Error exception.

Values 0x12 to 0x1F in the condition code field, Q, will raise an Instruction Error exception. Condition code tests, 0x10 an 0x11 are used to test the extended arithmetic saturation flag.

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

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

Table 64 Extension ALU Instructions

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 logical 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	ADDS	a ← sat32(b+c)	Add and saturate.
0x07	SUBS	a ← sat32 (b-c)	Subtract and saturate.

0x08	DIVAW	b_temp ← b<<1
		if (b_temp>=c)
		a ← ((b_temp-c)+1)
		else
		a ← b
			Division assist.

0x0A	ASLS	a ← sat32 (b<<c)	Arithmetic shift left and saturate. Supports negative shift values for right shift.
0x0B	ASRS	a ← sat32 (b>>c)	Arithmetic shift right and saturate. Supports -ve shift values for left shift.

Sub-opcode i field (6 bits)	Instruction	Operation	Description
0x28	ADDSDW	a ← sat16(b.high+c.high): sat16(b.low+c.low)	Dual 16-bit add and saturate.
0x29	SUBSDW	a ← sat16(b.high-c.high): sat16(b.low-c.low)	Dual16-bit subtract and saturate.
0x2A		Instruction Error	Reserved
...		Instruction Error	Reserved
0x2E		Instruction Error	Reserved
0x2F	SOPs	A field is sub-opcode2	See Single operand SOP table
0x30		Instruction Error	Reserved
...		Instruction Error	Reserved
0x3F		Instruction Error	Reserved

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.

Table 65 Extension Single Operand Instructions

Sub-opcode2 A field (6 bits)	Instruction	Operation	Description
0x00	SWAP	b ← swap(c)	Swap words
0x01	NORM	b ← norm(c)	Normalize
0x02	SAT16	b ← sat16(c)	Saturate 32-bit input to 16-bits
0x03	RND16	b ← sat32(c+0x00008000)&0xffff0000	Round 32-bit input to 16-bits
0x04	ABSSW	b ← sat16(abs(c.low))	Absolute value of 16-bit input
0x05	ABSS	b ← sat32(abs(c))	Absolute value of 32-bit input
0x06	NEGSW	b ← sat16(neg(c.low))	Negate and saturate 16-bit input
0x07	NEGS	b ← sat32(neg(c))	Negate and saturate 32-bit input
0x08	NORMW	b ← norm(c)	Normalize word
0x09			Reserved
...			Reserved
0x3F	ZOPs	B field is sub-opcode3	See Zero operand (ZOP) table

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.

Table 66 Extension Zero Operand Instructions

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

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 63 on page 149.

Market Specific Extension Instructions, 0x09 - 0x0B

The market-specific extension instructions are special instructions that use the major opcodes 0x09 to 0x0B. The remaining encoding fields of each of these instructions are not detailed here and are to be interpreted by the market-specific extension instructions themselves.

Any instruction opcodes that are not implemented raise an Instruction Error exception.

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

Table 67 Summary of Market-Specific Extension Instruction Encoding

Major Opcode	Instruction Usage
0x09	ARC market-specific extension instructions
0x0A	ARC market-specific extension instructions
0x0B	ARC market-specific extension instructions

Market Specific Extension Instruction, 0x09

At major opcode 0x09, the market-specific instruction 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
I[4:0]					Market specific																										
0	1	0	0	1	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Figure 89 Market-Specific Extension Instruction 0x09 Encoding

Market Specific Extension Instruction, 0x0A

At major opcode 0x0A, the market-specific instruction 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
I[4:0]					Market specific																										
0	1	0	1	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Figure 90 Market-Specific Extension Instruction 0x0A Encoding

Market Specific Extension Instruction, 0x0B

At major opcode 0x0B, the market-specific instruction 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
I[4:0]					Market specific																										
0	1	0	1	1	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Figure 91 Market-Specific Extension Instruction 0x0B Encoding

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. Instruction Set Summary on page 93 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 Instruction Set Details on page 173.

A complete list of the major opcodes is shown in Table 47.on page 133. The list of syntax conventions is shown in Table 28 on page 93.The encoding notation shown in Table 48 on page 134 and Table 49 on page 134.

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

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]			i[1:0]		a[2:0]		
0	1	1	0	0	b	b	b	c	c	c	i	i	a	a	a

Syntax:

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

Table 68 16-Bit, LD / ADD Register-Register

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.)
0x02	LDW_S	a ← mem[b + c].w	Load unsigned word (reg.+reg.)
0x03	ADD_S	a ← b + c	Add

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

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]			i[1:0]		u[2:0]		
0	1	1	0	1	b	b	b	c	c	c	i	i	u	u	u

Syntax:

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

Table 69 16-Bit, ADD/SUB Register-Immediate

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

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
I[4:0]					b[2:0]			h[2:0]			i[1:0]		h[5:3]		
0	1	1	1	0	b	b	b	h	h	h	i	i	H	H	H

For the ARC 700 processor the program counter (PCL) is not permitted to be the destination of an instruction. A value of in 0x03 in the sub opcode field, i, and a value of 0x3F in destination register field, H, will raise an Instruction Error exception.

Syntax:

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

MOV_S h, b

MOV_S 0, b *(h=limm)*

Table 70 16-Bit MOV/CMP/ADD with High Register

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

Long Immediate with Mov/Cmp/Add

The 6-bit register field in the instruction can indicate that long immediate data is used.

When a source register is set to r62, an explicit long immediate value will follow the instruction word.

When a destination register is set to r62 there is no destination for the result of the instruction so the result is discarded.

If the long immediate indicator is used in both a source and destination operand the following long immediate value will be used as the source operand and the result will be discarded as expected.

When an instruction uses long immediate, the first instruction word is the instruction that contains the long immediate data indicator (register r62). The second long-word instruction 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]

General Operations, register-register

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]			i[4:0]				
0	1	1	1	1	b	b	b	c	c	c	i	i	i	i	i

Syntax:

op_S b,b,c

op_S b,c

General Operations, Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					b[2:0]			i[2:0]			0x00				
0	1	1	1	1	b	b	b	i	i	i	0	0	0	0	0

Syntax:

op_S b

op_S b,b

General Operations, No Registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I[4:0]					i[2:0]			0x07			0x00				
0	1	1	1	1	i	i	i	1	1	1	0	0	0	0	0

Syntax:

op_S

General Operations, 0x0F, [0x00 - 0x1F]

Table 71 16-Bit General Operations

Sub-opcode i field (5 bits)	Instruction	Operation	Description
0x00	SOPs	c field is sub-opcode2	See 16-Bit Single Operand Instructions table on page 159.
0x01		Instruction Error	Reserved
0x02	SUB_S	b ← b - c	Subtract
0x03		Instruction Error	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		Instruction Error	Reserved
0x09		Instruction Error	Reserved
0x0A		Instruction Error	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
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

Sub-opcode i field (5 bits)	Instruction	Operation	Description
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		Instruction Error	Reserved
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	TRAP_S	Trap	Raise Exception
0x1F	BRK_S	Break	Break (Encoding is 0x7FFF)

Single Operand, Jumps and Special Format Instructions, 0x0F, [0x00, 0x00 - 0x07]

Syntax:

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

Table 72 16-Bit Single Operand Instructions

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		Instruction Error	Reserved
0x05		Instruction Error	Reserved
0x06	SUB_S.NE	if (flags.Z==0) then b ← b - b	If Z flag is 0, clear register
0x07	ZOP s	b field is sub-opcode3	See 16-Bit Zero Operand Instructions table on page 160

Zero Operand Instructions, 0x0F, [0x00, 0x07, 0x00 - 0x07]

Syntax:

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

Table 73 16-Bit Zero Operand Instructions

Sub-opcode3 b field (3 bits)	Instruction	Operation	Description
0x00	NOP_S	nop	No operation
0x01	UNIMP_S	Instruction Error	Unimplemented Instruction
0x02		Instruction Error	Reserved
0x03		Instruction Error	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

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]				
I	I	I	I	I	b	b	b	c	c	c	u	u	u	u	u

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)*

Table 74 16-Bit Load and Store with Offset

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

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

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

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

Table 75 16-Bit Shift/SUB/Bit Immediate

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

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

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

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	

Table 76 16-Bit Stack Pointer based Instructions

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.
0x04	ADD_S	b ← SP + u7	Add
0x05	ADD_S /SUB_S	sp ← sp +- u7	See Table 77 on page 163
0x06	POP_S	Pop register from stack	See Table 78 on page 163
0x07	PUSH_S	Push register to stack	See Table 79 on page 163

Add/Subtract SP Relative, 0x18, [0x05, 0x00-0x07]

Table 77 16-Bit Add/Subtract SP relative Instructions

Sub-opcode b field (3 bits)	Instruction	Operation	Description
0x00	ADD_S	sp ← sp +- u7	Add immediate to SP
0x01	SUB_S	sp ← sp - u7	Subtract immediate from SP
0x02		Instruction Error	Reserved
…		Instruction Error	Reserved
0x07		Instruction Error	Reserved

POP Register from Stack, 0x18, [0x06, 0x00-0x1F]

Table 78 16-Bit POP register from stack instructions

Sub-opcode u field (5 bits)	Instruction	Operation	Description
0x00		Instruction Error	Reserved

0x01	POP_S b	b ← mem[SP].l
		SP ← SP + 4
			Pop register from stack

0x02		Instruction Error	Reserved
…		Instruction Error	Reserved
0x10		Instruction Error	Reserved

0x11	POP_S blink	blink ← mem[SP].l
		SP ← SP + 4
			Pop blink from stack
			(b=reserved)

0x12		Instruction Error	Reserved
…		Instruction Error	Reserved
0x1F		Instruction Error	Reserved

PUSH Register to Stack, 0x18, [0x07, 0x00-0x1F]

Table 79 16-Bit PUSH register to stack instructions

Sub-opcode u field (5 bits)	Instruction	Operation	Description
0x00		Instruction Error	Reserved

0x01	PUSH_S b	SP ← SP - 4
		mem[SP].l ← b
			Push register to stack

0x02		Instruction Error	Reserved
…		Instruction Error	Reserved
0x10		Instruction Error	Reserved

0x11	PUSH_S blink	SP ← SP - 4
		mem[SP].l ← blink
			Push blink to stack
			(b=reserved)

Sub-opcode u field (5 bits)	Instruction	Operation	Description
0x12		Instruction Error	Reserved
…		Instruction Error	Reserved
0x1F		Instruction Error	Reserved

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

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

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)*

Table 80 16-Bit GP Relative Instructions

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

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
I[4:0]					b[2:0]			u[7:0]							
1	1	0	1	0	b	b	b	u	u	u	u	u	u	u	u

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
I[4:0]					b[2:0]			u[7:0]							
1	1	0	1	1	b	b	b	u	u	u	u	u	u	u	u

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
I[4:0]					b[2:0]			i	u[6:0]						
1	1	1	0	0	b	b	b	i	u	u	u	u	u	u	u

Syntax:

ADD_S b, b, u7

CMP_S b, u7

Table 81 16-Bit ADD/CMP Immediate

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

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
I[4:0]					b[2:0]			i	s[7:1]						
1	1	1	0	1	b	b	b	i	s	s	s	s	s	s	s

Syntax:

BREQ_S b, 0, s8

BRNE_S b, 0, s8

Table 82 16-Bit Branch on Compare

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

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
I[4:0]					i[1:0]		s[9:1]								
1	1	1	1	0	i	i	s	s	s	s	s	s	s	s	s

Syntax:

B_S	s10
BEQ_S	s10
BNE_S	s10

Table 83 16-Bit Branch, Branch Conditionally

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

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
I[4:0]					0x03		i[2:0]			s[6:1]					
1	1	1	1	0	1	1	i	i	i	s	s	s	s	s	s

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

Table 84 16-Bit Branch Conditionally

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

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
I[4:0]					s[12:2]										
1	1	1	1	1	s	s	s	s	s	s	s	s	s	s	s

Syntax:

BL_S s13

This page is intentionally left blank.

Chapter 8 — Condition Codes

Introduction

The ARCompact based processor has an extensive instruction set most of which can be carried out conditionally or set the flags or both. 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.

Flag Setting

For those 32-bit instructions that are able to set the flags, updates will only occur if the set flags directive (.F) is used. For some instructions the only effect is to set the flags and not update any general purpose register. Such instructions include CMP, RCMP and TST.

For 16-bit instructions no flag setting will occur, except for a few instructions where flag setting is implicit e.g. BTST_S, CMP_S and TST_S.

Status Register

The status register contains the status flags. The status register (STATUS32), shown in Figure 45 on page 51, contains the following status flags for the condition codes: zero (Z), negative (N), carry (C) and overflow (V).

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 85 on page 170 shows condition names and the conditions they test.

Table 85 Condition codes

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

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

For the ARCtangent-A5 and ARC 600 processors, 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).

For the ARC 700 processor, if an extension condition code is used that is not implemented then an Instruction Error exception will be raised.

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

Extended Arithmetic Condition Codes

The extended arithmetic library provides additional status flags in the AUX_MACMODE register which are set by the Extended Arithmetic Library instructions on page 126. See Extended Arithmetic Auxiliary Registers on page 62 for further details of the AUX_MACMODE register.

The following extension condition codes are available with the extended arithmetic library and may be used on any conditionally executable instruction to test the saturate bits:

Table 86 Extended Arithmetic Condition Codes

Mnemonic	Condition	Test	Code
SS	Saturate Set	S_1 or S_2	0x10
SC	Saturate Clear	$/S_1$ and $/S_2$	0x11

This page is intentionally left blank.

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 Instruction Set Summary on page 93.

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 list of syntax conventions is shown in Table 28 on page 93.The encoding notation shown in Table 48 on page 134 and Table 49 on page 134.

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 extended arithmetic library contains 13 instructions. The extension options provide an additional 4 instructions of 32-bit formats and 1 instruction in 16-bit format, giving a total of 104 instructions.

The ARC 700 processor additionally supports, 4 multiply instructions (as options) and 7 more basecase instructions. The 2 ARCtangent-A5 and ARC 600 multiply instructions are not supported, giving a total of 113 instructions.

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

Table 87 List of Instructions

32-Bit Instructions		**16-Bit Instructions**	
Instruction	**Operation**	**Instruction**	**Operation**
ABS	Absolute value	ABS_S	Absolute value
ABSS	Absolute and saturate		
ABSSW	Absolute and saturate of word		
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

32-Bit Instructions		16-Bit Instructions	
Instruction	Operation	Instruction	Operation
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
ADDS	Add and saturate		
ADDSDW	Add and saturate dual word		
AND	Logical AND	AND_S	Logical AND
ASL	Arithmetic Shift Left	ASL_S	Arithmetic Shift Left
ASLS	Arithmetic shift left and saturate		
ASR	Arithmetic Shift Right	ASR_S	Arithmetic Shift Right
ASRS	Arithmetic shift right and saturate		
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
BRK	Break (halt) processor	BRK_S	Break (halt) processor
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
DIVAW	Division assist		
EX	Atomic Exchange		
EXT	Unsigned extend	EXT_S	Unsigned extend
FLAG	Write to Status Register		
Jcc	Jump	Jcc_S	Jump
JLcc	Jump and Link	JL_s	Jump and Link
LD	Load from memory	LD_S	Load from memory
LPcc	Loop (zero-overhead loops)		
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 Signed Multiply	MUL64_S	32 x 32 Multiply
MULU64	32 x 32 Unsigned Multiply		
MPY	32 x 32 Signed Multiply (low)		
MPYH	32 x 32 Signed Multiply (high)		
MPYHU	32 x 32 Unsigned Multiply (high)		

32-Bit Instructions		16-Bit Instructions	
Instruction	Operation	Instruction	Operation
MPYU	32 x 32 Unsigned Multiply (low)		
NEG	Negate	NEG_S	Negate
NEGSW	Negate and saturate of word		
NEGS	Negate and saturate		
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
PREFETCH	Prefetch from memory		
RCMP	Reverse Compare		
RLC	Rotate Left through Carry		
RND16	Round to word		
ROR	Rotate Right		
RRC	Rotate Right through Carry		
RSUB	Reverse Subtraction		
RTIE	Return from Interrupt/Exception		
SAT16	Saturate to word		
SBC	Subtract with carry		
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		
SUBS	Subtract and saturate		
SUBSDW	Subtract and saturate dual word		
SWAP	Swap 16 x 16		
SWI	Software Interrupt		
SYNC	Synchronize		
TRAP0	Raise exception with param. 0	TRAP_S	Raise exception
TST	Test	TST_S	Test
XOR	Logical Exclusive-OR	XOR_S	Logical Exclusive-OR
		NOP_S	No Operation
		POP_S	Restore register value from stack
		PUSH_S	Store register value on stack
		UNIMP_S	Unimplemented Instruction

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.

Operation	Operation of the instruction
Format	Instruction format
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

ABS

Absolute

Arithmetic Operation

Operation:

dest ← ABS(src)

Format:

inst dest, 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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

ABSS

Absolute with Saturation

Extended Arithmetic Operation

Operation:

dest ← sat_{32} (abs(src))

Format:

inst dest, src

Format Key:

dest	=	Destination Register
src	=	Source Operand 1

Syntax:

With Result		**Instruction Code**	
ABSS<.f>	b,c	00101bbb00101111FBBBCCCCCC000101	
ABSS<.f>	b,u6	00101bbb01101111FBBBuuuuuu000101	
ABSS<.f>	b,limm	00101bbb00101111FBBB111110000101	L
Without Result			
ABSS<.f>	0,c	0010111000101111F111CCCCCC000101	
ABSS<.f>	0,u6	0010111001101111F111uuuuuu000101	
ABSS<.f>	0,limm	0010111000101111F111111110000101	L

Flag Affected (32-Bit):

Z	●	= Set if input is zero
N	●	= Set if most significant bit of input is set
C		= Unchanged
V	●	= Set if input is 0x8000_0000 otherwise cleared
S	●	= Set if input is 0x8000_000 (‘sticky’ saturation)

Key:

L	= Limm Data

NOTE In contrast with other instructions, the value of the input operand is used to set the flags.

Related Instructions:

SAT16

RND16

ABSSW

NEGSW

Description:

Obtain the absolute value of long word operand and place the result in the destination register. Note that, the absolute value of 0x8000_0000 yields 0x7FFF_FFFF. Both saturation flags S1 and S2 will be set if the result of the instruction saturates. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if src <= 0x7FFF_FFFF                                  /* ABSS */
 sat = 0                                               // Using
 dest = src                                            // unsigned
else                                                   // pseudo
 sat = 0                                               // arithmetic
 dest = 0 - src
if src==0x8000_0000
 sat = 1
 dest = 0x7FFF_FFFF
if F==1 then
 Z_flag = if src==0 then 1 else 0
 N_flag = src[31]
 V_flag = sat
 S_flag = S_flag || sat
```

Assembly Code Example:

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

ABSSW

Absolute Word with Saturation

Extended Arithmetic Operation

Operation:

dest ← sat_{16} (abs(src.low))

Format:

inst dest, src

Format Key:

dest = Destination Register
src = Source Operand 1

Syntax:

With Result		**Instruction Code**	
ABSSW<.f>	b,c	00101bbb00101111FBBBCCCCCC000100	
ABSSW<.f>	b,u6	00101bbb01101111FBBBuuuuuu000100	
ABSSW<.f>	b,limm	00101bbb00101111FBBB111110000100	L
Without Result			
ABSSW<.f>	0,c	0010111000101111F111CCCCCC000100	
ABSSW<.f>	0,u6	0010111001101111F111uuuuuu000100	
ABSSW<.f>	0,limm	0010111000101111F111111110000100	L

Flag Affected (32-Bit):

Z	●	= Set if input is zero
N	●	= Set if most significant bit of input is set
C		= Unchanged
V	●	= Set if input is 0x8000 otherwise cleared
S	●	= Set if input is 0x8000 (‘sticky’ saturation)

Key:

L	= Limm Data

NOTE In contrast with other instructions, the value of the input operand is used to set the flags.

Related Instructions:

SAT16	ABSS
RND16	NEGSW

Description:

Obtain the result is the absolute value of least significant word (LSW) of the source operand. Note that the absolute value of 0xFFFF_8000 yields 0x0000_7FFF. Both saturation flags S1 and S2 will be set if the result of the instruction saturates. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
src16 = src & 0x0000_FFFF                                  /* ABSSW */
if src16 <= 0x7FFF                                         // Using
 sat = 0                                                   // unsigned
 dest = src16                                              // pseudo
else                                                       // arithmetic
 sat = 0
 dest = 0x0000_0000 - src16
if src16==0x8000
 sat = 1
 dest = 0x0000_7FFF
if F==1 then
 Z_flag = if src==0 then 1 else 0
 N_flag = src[31]
 V_flag = sat
 S_flag = S_flag || sat
```

Assembly Code Example:

```
ABSSW r1,r2             ; Take the LSW absolute saturated value of
                        ; r2 and write result into r1
```

ADC

Addition with Carry

Arithmetic Operation

Operation:

if (cc=true) then dest ← src1 + src2 + carry

Format:

inst dest, src1, src2

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

ADD1

ADD2

ADD3

Description:

Add source operand 1 (src1) and source operand 2 (src2) and carry, and place the result in the destination register. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then dest ← src1 + src2

Format:

inst 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:

ADC

ADD1

ADD2

ADD3

Description:

Add source operand 1 (src1) to source operand 2 (src2) and place the result in the destination register. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then dest ← src1 + (src2 << 1)

Format:

inst dest, src1, src2

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
ADC 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. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then dest ← src1 + (src2 << 2)

Format:

inst dest, src1, src2

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

ADC

ADD1

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. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then dest ← src1 + (src2 << 3)

Format:

inst dest, src1, src2

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
ADC	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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

ADDS

Signed Add with Saturation

Extended Arithmetic

Operation:

dest ← sat_{32} (src1 + src2)

Format:

inst dest, src1, src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
ADDS<.f>	a,b,c	00101bbb00000110FBBBCCCCCCAAAAAA	
ADDS<.f>	a,b,u6	00101bbb01000110FBBBuuuuuuAAAAAA	
ADDS<.f>	b,b,s12	00101bbb10000110FBBBssssssSSSSSS	
ADDS<.cc><.f>	b,b,c	00101bbb11000110FBBBCCCCCC0QQQQQ	
ADDS<.cc><.f>	b,b,u6	00101bbb11000110FBBBuuuuuu1QQQQQ	
ADDS<.f>	a,limm,c	0010111000000110F111CCCCCCAAAAAA	L
ADDS<.f>	a,b,limm	00101bbb00000110FBBB111110AAAAAA	L
ADDS<.cc><.f>	b,b,limm	00101bbb11000110FBBB1111100QQQQQ	L
Without Result			
ADDS<.f>	0,b,c	00101bbb00000110FBBBCCCCCC111110	
ADDS<.f>	0,b,u6	00101bbb01000110FBBBuuuuuu111110	
ADDS<.f>	0,b,limm	00101bbb00000110FBBB111110111110	L
ADDS<.cc><.f>	0,limm,c	0010111011000110F111CCCCCC0QQQQQ	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 from the add
V	●	= Set if result saturated, otherwise cleared
S	●	= Set if result saturated (‘sticky’ saturation)

Key:

L	= Limm Data

Related Instructions:

SUBS ADDSDW

Description:

Perform a signed addition of the two source operands. If the result overflows, limit it to the maximum signed value. Both saturation flags S1 and S2 will be set if the result of the instruction saturates. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if cc==true then                                    /* ADDS */
 dest = src1 + src2
 sat = sat32(dest)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = 0
  V_flag = sat
  S_flag = S_flag || sat
```

Assembly Code Example:

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

ADDSDW

Signed Add with Saturation Dual Word

Extended Arithmetic Operation

Operation:

Dest ← sat_{16}(src1.high+src2.high): sat_{16}(src1.low+src2.low)

Format:

inst dest, src1, src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
ADDSDW<.f>	a,b,c	`00101bbb00101000FBBBCCCCCCAAAAAA`	
ADDSDW<.f>	a,b,u6	`00101bbb01101000FBBBuuuuuuAAAAAA`	
ADDSDW<.f>	b,b,s12	`00101bbb10101000FBBBssssssSSSSSS`	
ADDSDW<.cc><.f>	b,b,c	`00101bbb11101000FBBBCCCCCC0QQQQQ`	
ADDSDW<.cc><.f>	b,b,u6	`00101bbb11101000FBBBuuuuuu1QQQQQ`	
ADDSDW<.f>	a,limm,c	`0010111000101000F111CCCCCCAAAAAA`	L
ADDSDW<.f>	a,b,limm	`00101bbb00101000FBBB111110AAAAAA`	L
ADDSDW<.cc><.f>	b,b,limm	`00101bbb11101000FBBB111110QQQQQQ`	L
Without Result	**- only flags will be set**		
ADDSDW<.f>	0,b,c	`00101bbb00101000FBBBCCCCCC111110`	
ADDSDW<.f>	0,b,u6	`00101bbb01101000FBBBuuuuuu111110`	
ADDSDW<.cc><.f>	0,limm,c	`0010111011101000F111CCCCCC0QQQQQ`	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V	●	= Set if result saturated, otherwise cleared
S	●	= Set if result saturated ('sticky' saturation)

Key:

L	= Limm Data

Related Instructions:

SUBSDW SUBS

ADDS

Description:

Perform a signed dual-word addition of the two source operands. If the result overflows, limit it to the maximum signed value. The saturation flags S1 and S2 will be set according to the result of the channel 1 (high 16-bit) and channel 2 (low 16-bit) calculations respectively. Any flag updates will only occur if the set flags suffix (.F) is used.

Assembly Code Example:

```
ADDSDW r1,r2,r3        ;
```

AND

Bitwise AND Operation

Logical Operation

Operation:

if (cc=true) then dest ← src1 AND src2

Format:

inst dest, src1, 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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

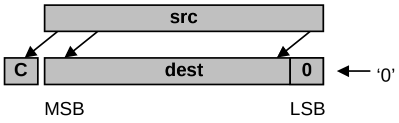
ASL

Arithmetic Shift Left

Logical Operation

Operation:

dest ← src + src



Format:

inst dest, 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	`0010011000101111F111CCCCCC000000`	
ASL<.f>	0,u6	`0010011001101111F111uuuuuu000000`	
ASL<.f>	0,limm	`0010011000101111F111111110000000`	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
ROR
ASL multiple
ROR multiple
LSR
RRC
ASR multiple
LSR multiple

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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

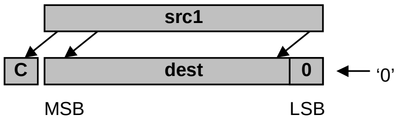
ASL multiple

Multiple Arithmetic Shift Left

Logical Operation

Operation:

if (cc=true) then dest ← arithmetic shift left of src1 by src2



Format:

inst dest, src1, 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
ROR
ASR multiple
ROR multiple

LSR
RRC
LSR multiple

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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

ASLS

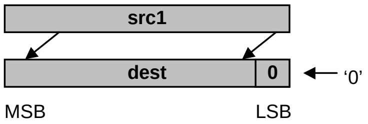
Arithmetic +/- Shift Left with Saturation

Extended Arithmetic Operation

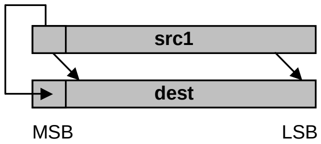
Operation:

dest ← sat_{32} (src1 << src2)

Positive src2: dest ← arithmetic shift left of src1 by src2 with saturation on the result.



Negative src2: dest ← arithmetic shift right of src1 by -src2.



Format:

inst dest, src1, src2

Format Key:

dest = Destination Register
src1 = Source Operand 1
src2 = Source Operand 2

Syntax:

With Result		**Instruction Code**	
ASLS<.f>	a,b,c	00101bbb00001010FBBBCCCCCCAAAAAA	
ASLS<.f>	a,b,u6	00101bbb01001010FBBBuuuuuuAAAAAA	
ASLS<.f>	b,b,s12	00101bbb10001010FBBBssssssSSSSSS	
ASLS<.cc><.f>	b,b,c	00101bbb11001010FBBBCCCCCC0QQQQQ	
ASLS<.cc><.f>	b,b,u6	00101bbb11001010FBBBuuuuuu1QQQQQ	
ASLS<.f>	a,limm,c	0010111000001010F111CCCCCCAAAAAA	L
ASLS<.f>	a,b,limm	00101bbb00001010FBBB111110AAAAAA	L
ASLS<.cc><.f>	b,b,limm	00101bbb11001010FBBB111110QQQQQQ	L
Without Result			
ASLS<.f>	0,b,c	00101bbb00001010FBBBCCCCCC111110	
ASLS<.f>	0,b,u6	00101bbb01001010FBBBuuuuuu111110	
ASLS<.cc><.f>	0,limm,c	0010111011001010F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V	●	= Set if result saturated, otherwise cleared
S	●	= Set if result saturated ('sticky' saturation)

Key:

L = Limm Data

Related Instructions:

ASRS ASL

Description:

a) If src2 is positive, with a value in the range 0<= operand2 <= 31, arithmetically shift src1 left by src2 places. The result is saturated and then placed in the destination register.

When src2 is larger than 31, the result is set to 0x7FFF_FFF and 0x8000_0000 (saturation) for positive non-zero and negative input respectively.

b) If src2 is negative, with a value in the range -31<= operand2 <0, arithmetically shift src1 right by -src2 places (positive right shift) and placed in the destination register.

When src2 is less than -31, src2 is set to –31, ensuring a maximum right shift of 31 places.

Both saturation flags S1 and S2 will be set if the result of the instruction saturates. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if cc==true then                                  /* ASLS */
 if src2 > 0x0000_001F and src2 < 0x7FFF_FFFF    /* Multiple */
  tempdest = src1 << 0x0000_001F                  /* and */
 if src2 > 0x8000_0000 and src2 < 0xFFFFFFE1     /* Saturated */
  tempdest = src1 >> 0x0000_001F                  /* using */
 if src2 >= 0 and src2 <= 0x0000_001F             /* unsigned */
  tempdest = src1 << src2                         /* pseudo code */
 if src2 < 0 and src2 >= 0xFFFFFFE1
  tempdest = src1 >> (0 - src2)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
; 0 <= operand2 <= 31 : Arithmetically shift left operand1
; by operand2 places with saturation:

ASLS r0, 0x00001111, 1     ; Yields r0=0x0000_2222
ASLS r0, 0x00001111, 2     ; Yields r0=0x0000_4444
ASLS r0, 0x00001111, 3     ; Yields r0=0x0000_8888
ASLS r0, 0x10001111, 1     ; Yields r0=0x2000_2222
ASLS r0, 0x10001111, 2     ; Yields r0=0x4000_4444
ASLS r0, 0x10001111, 3     ; Yields r0=0x7FFF_FFFF (saturation)
ASLS.f r0, 0x10001111, 3   ; Yields r0=0x7fff_ffff
                           ; (saturation, V and S flags are set)

ASLS r0, 0x10001111, 31    ; Yields r0=0x7FFF_FFFF (saturation)

; Operand2 > 31 : Result is set to 0x7FFF_FFFF or
; 0x8000_0000 (saturation) for positive (non-zero)
; and negative input respectively.
ASLS r0, 0x00000001, 33    ; Yields r0=0x7FFF_FFFF
                           ; (saturate to largest positive value)
ASLS r0, 0xFFFFFFFF, 33    ; Yields r0=0x8000_0000
                           ; (saturate to largest negative value)

; Supports ASRS with negative shift (operand2):
ASLS r0, r1, -1            ; in effect performs asrs r0, r1, 1
ASLS r0, 0x00001111, -1    ; Yields r0=0x0000_0888
ASLS r0, 0x00001111, -12   ; Yields r0=0x0000_0001
ASLS r0, 0x00001111, -13   ; Yields r0=0x0000_0000

ASLS r0, 0xFFFFEEEF, -1    ; Yields r0=0xFFFF_F777
ASLS r0, 0xFFFFEEEF, -12   ; Yields r0=0xFFFF_FFFF
ASLS r0, 0xFFFFEEEF, -13   ; Yields r0=0xFFFF_FFFF (sign filled)
```

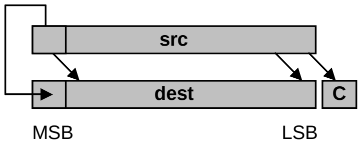
ASR

Arithmetic Shift Right

Logical Operation

Operation:

dest ← src >> 1



Format:

inst dest, 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	LSR
ROR	RRC
ASL multiple	ASR multiple
ROR multiple	LSR multiple

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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

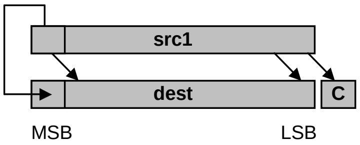
ASR multiple

Multiple Arithmetic Shift Right

Logical Operation

Operation:

if (cc=true) then dest ← arithmetic shift right of src1 by src2



Format:

inst dest, src1, 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

ROR

ASR multiple

ROR multiple

LSR

RRC

LSR multiple

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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

ASRS

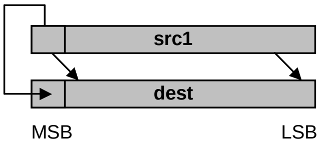
Arithmetic +/- Shift Right with Saturation

Extended Arithmetic Operation

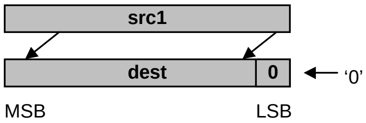
Operation:

dest ← sat_{32} (src1 >> src2)

Positive src2: dest ← arithmetic shift right of src1 by src2



Negative src2: dest ← arithmetic shift left of src1 by -src2 with saturation



Format:

inst dest, src1, src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
ASRS<.f>	a,b,c	00101bbb00001011FBBBCCCCCCAAAAAA	
ASRS<.f>	a,b,u6	00101bbb01001011FBBBuuuuuuAAAAAA	
ASRS<.f>	b,b,s12	00101bbb10001011FBBBssssssSSSSSS	
ASRS<.cc><.f>	b,b,c	00101bbb11001011FBBBCCCCCC0QQQQQ	
ASRS<.cc><.f>	b,b,u6	00101bbb11001011FBBBuuuuuu1QQQQQ	
ASRS<.f>	a,limm,c	0010111000001011F111CCCCCCAAAAAA	L
ASRS<.f>	a,b,limm	00101bbb00001011FBBB111110AAAAAA	L
ASRS<.cc><.f>	b,b,limm	00101bbb11001011FBBB1111100QQQQQ	L
Without Result			
ASRS<.f>	0,b,c	00101bbb00001011FBBBCCCCCC111110	
ASRS<.f>	0,b,u6	00101bbb01001011FBBBuuuuuu111110	
ASRS<.cc><.f>	0,limm,c	0010111011001011F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V	●	= Set if result saturated, otherwise cleared
S	●	= Set if result saturated ('sticky' saturation)

Key:

L	= Limm Data

Related Instructions:

ASLS ASR

Description:

a) If src2 is positive, with a value in the range 0<= src2 <=31, arithmetically shift src1 right by src2 places and put the result in the destination register.

NOTE When src2 is larger than 31, src2 is set to 31, ensuring a maximum right shift of 31 places.

b) If src2 is negative with a value in the range -31<= src2 <= 0, arithmetically shift src1 left by src2 places (positive left shift). Result is saturated and then placed in the destination register.

When src2 is less than -31, the result is set to 0x7FFF_FFF and 0x8000_0000 (saturation) for positive non-zero and negative input respectively.

Both saturation flags S1 and S2 will be set if the result of the instruction saturates. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if cc==true then                                 /* ASRS */
 if src2 > 0x0000_001F and src2 < 0x7FFF_FFFF    /* Multiple */
  tempdest = src1 >> 0x0000_001F                 /* and */
 if src2 > 0x8000_0000 and src2 < 0xFFFFFFE1     /* Saturated */
  tempdest = src1 << 0x0000_001F                 /* using */
 if src2 >= 0 and src2 <= 0x0000_001F            /* unsigned */
  tempdest = src1 >> src2                        /* pseudo code */
 if src2 < 0 and src2 >= 0xFFFFFFE1
  tempdest = src1 << (0 - src2)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
```

Assembly Code Example:

```
; 0 <= operand2 <= 31 : Arithmetically shift right operand1
; by operand2 places:
ASRS r0, 0x00001111, 1     ; Yields r0=0x0000_0888
ASRS r0, 0x00001111, 2     ; Yields r0=0x0000_0444
ASRS r0, 0x00001111, 3     ; Yields r0=0x0000_0222

; Operand2 > 31 : The number of right shifts is limited to
; 31 places.
ASRS r0, 0x7FFFFFFF, 33    ; Yields r0=0x0000_0000
ASRS r0, 0x80000000, 33    ; Yields r0=0xFFFF_FFFF

; Supports ASLS with negative shift (operand2).
; For shifts in the range -31<= operand2 <= 0,
; arithmetically shift left operand1
; by -operand2 places (positive left shift). In the case
; of overflow result is saturated.
ASRS r0, r1, -1            ; in effect performs ASLS r0, r1, 1
ASRS r0, 0x0000_1111, -1   ; Yields     r0=0x0000_2222
ASRS r0, 0x1000_1111, -3   ; Yields     r0=0x7FFF_FFFF
                           ; (saturation)
ASRS.f r0, 0x1000_1111, -3 ; Yields     r0=0x7FFF_FFFF
                           ; (saturation, V and S flags are set)
ASRS r0, 0xFFFF_FF00, -31  ; Yields     r0=0x8000_0000
                           ; (saturation)

; When -operand2 is larger than 31, result is set to
; 0x7FFF_FFFF and 0x8000_0000 (saturation) for positive (non-zero)
; and negative input respectively.
```

BBIT0

Branch on Bit Test Clear

Branch Operation

Operation:

if (src1 AND 2^{src2}) = 0 then cPC ← cPCL+rd

Format:

inst src1, src2, 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 BRcc

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. The status flags are not updated with this instruction.

To take advantage of the ARC 600 branch prediction unit, 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.

For the ARC 600 processor, r63 (PCL) should not be used as a source operand in a branch on compare instruction (BBIT0, BBIT1, or BRcc).

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 Operation

Operation:

if (src1 AND 2^{src2}) = 1 then cPC ← cPCL+rd

Format:

inst src1, src2, 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	BRcc

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. The status flags are not updated with this instruction.

To take advantage of the ARC 600 branch prediction unit, 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.

For the ARC 600 processor, r63 (PCL) should not be used as a source operand in a branch on compare instruction (BBIT0, BBIT1, or BRcc).

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

Branch Operation

Operation:

if (cc=true) then cPC ← (cPCL+rd)

Format:

inst rel_addr

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>:

Code	Q Field	Description	Test	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

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged

Key:

L = Limm Data

V □ = Unchanged

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. The status flags are not updated with this instruction.

CAUTION The Bcc instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D instruction.

The ARC 700 processor will raise an Illegal Instruction Sequence exception if an executed delay slot contains:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt (RTIE)
- Any instruction with long-immediate data as a source operand

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
```

Bcc_S

16-Bit Branch

Branch Operation

Operation:

if (cc=true) then cPC ← (cPCL+rd)

Format:

inst rel_addr

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	`1111000sssssssss`

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)
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	BRcc

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. The status flags are not updated with this instruction.

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

Logical Operation

Operation:

if (cc=true) then dest ← (src1 AND (NOT 2^{src2}))

Format:

inst dest, src1, 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	0010011000010000F111CCCCCCAAAAAA	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	0010011011010000F111CCCCCC0QQQQQ	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). Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then dest ← src1 AND NOT src2

Format:

inst dest, src1, 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	0010011000000110F111CCCCCCAAAAAA	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	0010011011000110F111CCCCCC0QQQQQ	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. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Branch Operation

Operation:

if (cc=true) then (cPC ← cPCL +rd) & (r31 ← nPC or dPC)

Format:

inst rel_addr

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	`00001sssssssss00SSSSSSSSSSNQQQQQ`
Branch Far (Unconditional)		
BL<.d>	s25	`00001sssssssss10SSSSSSSSSSNRtttt`
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>:

Code	Q Field	Description	Test	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

Code	Q Field	Description	Test	Code	Q Field	Description	Test
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 JLcc

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 ARC 700 processor will raise an Illegal Instruction Sequence exception if an executed delay slot contains:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt (RTIE)
- Any instruction with long-immediate data as a source operand

The conditional branch and link instruction has a maximum branch range of +/- 1MByte. The unconditional branch far format has a maximum branch range of +/- 16Mbytes. The target address for any branch and link instruction must be 32-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. The status flags are not updated with this instruction.

NOTE Since the 16-bit encoded instructions the target address is aligned to 32-bit, a special encoding allows for a larger branch displacement. 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
```

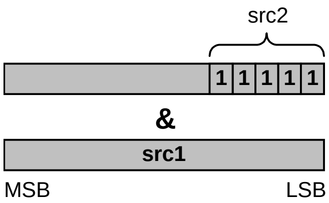
BMSK

Bit Mask

Logical Operation

Operation:

if (cc=true) then dest ← src1 AND $((2^{(src2+1)})-1)$



Format:

inst dest, src1, src2

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). Any flag updates will only occur if the set flags suffix (.F) is used.

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

Branch Operation

Operation:

if (cc=true) then cPC ← (cPCL+rd)

Format:

inst src1, src2, 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
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.

For the ARCtangent-A5 processor 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.

For the ARC 600 processor all 32-bit compare and branch instructions have three delay slots. The behavior of the 1st delay slot can be controlled by specifying the delay slot mode <.d>, however the following delay slots cannot be controlled, and any instruction present in the 2nd or 3rd delay slot is killed if the branch is taken.

To take advantage of the ARC 600 branch prediction unit, 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.

For the ARC 600 processor, r63 (PCL) should not be used as a source operand in a branch on compare instruction (BBIT0, BBIT1, or BRcc).

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.

The status flags are not updated with this 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:

Example 16 ARCtangent-A5 Branch on Compare

```
       ; if r0=2, r1=2
       brne          r0,r1,ok1    ; r0=r1, no branch to "ok1"
       add           r2,r2,1      ; executed
       add           r3,r3,1      ; 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      ; killed
       add           r3,r3,1      ; 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      ; executed
       add           r3,r3,1      ; 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      ; executed
       add           r3,r3,1      ; killed
       add           r4,r4,1      ; not fetched
ok4:
```

Example 17 ARC 600 Branch on Compare

```
       ; if r0=2, r1=2
       brne          r0,r1,ok1    ; r0=r1, no branch to "ok1"
       add           r2,r2,1      ; executed
       add           r3,r3,1      ; 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      ; killed
       add           r3,r3,1      ; killed
       add           r4,r4,1      ; killed
ok2:
       ; if r0=2, r1=2
       brne.d r0,r1,ok3           ; r0=r1, no branch to "ok3"
       add           r2,r2,1      ; executed
       add           r3,r3,1      ; 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      ; executed
       add           r3,r3,1      ; killed
       add           r4,r4,1      ; killed
ok4:
```

BRK

Breakpoint

Kernel/Debug Operation

Operation:

Halt and flush the processor

Format:

inst

Format Key:

inst = Instruction Mnemonic

Syntax:

	Instruction Code
BRK_S	0111111111111111
BRK	00100101011011110000000000111111

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 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 ARCompact based processor from performing any instructions beyond the breakpoint. Since the breakpoint is a serializing instruction, the pipeline is also flushed upon decode of this instruction.

To restart the ARCompact based 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 ARCompact based processor and to reset the initial stages of the pipeline. 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 Debug Register on page 50) in the Debug register when it is decoded at stage one of the pipeline. This allows the debugger to determine what caused the ARCompact based 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 ARCompact based processor.

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

In the ARCtangent-A5 processor, the breakpoint instruction can be placed anywhere in a program, except immediately following a BRcc or BBIT*n* instruction. 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.

In the ARC 600 processor, 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. Therefore the PC value after a break is the address of the next instruction to be executed. In order to continue after a BRK instruction the debugger decrements the PC value by 2 to obtain the re-start address.

If a BRK is put at the last location of a zero overhead loop then the PC value after the break could be the address of first instruction in the loop, so the debugger would not evaluate the correct restart address. The programmer should never insert a BRK as the last instruction in loops.

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 ARCompact based will stall stages one and two of the pipeline while allowing instructions in subsequent stages to proceed to completion.

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 ARCompact based processor is started, the link register will update as normal.

Interrupts are treated in the same manner by the ARCompact based 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.

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.

In the ARC 700 processor, the breakpoint instruction is a kernel only instruction unless enabled by the UB bit in the DEBUG register. Both a 32-bit, BRK, and 16-bit, BRK_S, form are supported. The breakpoint instruction is decoded in stage 1 an allows all preceding instructions to complete. The processor will halt with the program counter pointing at the address of the breakpoint instruction.

The breakpoint instruction can be placed anywhere in a program, including the delay slot of branch and jump instructions, and also immediately following a BRcc, a BBIT1 or a BBIT2 instruction.

NOTE A code sequence where a FLAG 1 is followed by BRK will operate as expected. The FLAG 1 will complete, halt the processor and flush the pipeline, all before the BRK is executed.

Pseudo Code Example:

```
FlushPipe()                          /* BRK_S */
DEBUG[BH] = 1
DEBUG[H] = 1
Halt()
```

Assembly Code Example:

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
```

For the ARC 700 processor, 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
```

BSET

Bit Set

Logical Operation

Operation:

if (cc=true) then dest ← (src1 OR (2^{src2}))

Format:

inst dest, src1, 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	`00100110000001111F111CCCCCCAAAAAA`	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). Any flag updates will only occur if the set flags suffix (.F) is used.

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

Operation:

if (cc=true) then (src1 AND (2src2))

Format:

inst src1, 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. The flag setting field, F, is always encoded as 1 for this instruction.

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)

Logical Operation

Operation:

if (cc=true) then dest ← (src1 XOR (2^{src2}))

Format:

inst dest, src1, 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). Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then src1 – src2

Format:

inst src1, src2

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition Code

Syntax:

		Instruction Code	
CMP	b,s12	00100bbb10001100 1BBBssssssSSSSSS	
CMP<.cc>	b,c	00100bbb110011001BBBCCCCCC0QQQQQ	
CMP<.cc>	b,u6	00100bbb110011001BBBuuuuuu1QQQQQ	
CMP<.cc>	b,limm	00100bbb110011001BBB111110 0QQQQQ	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. The flag setting field, F, is always encoded as 1 for this instruction.

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
```

DIVAW

Division Accelerator

Extended Arithmetic Operation

Operation:

```
if (src1 == 0)
        dest ← 0
else
{       src1_temp ← src1<<1
        if (src1_temp >= src2)
                dest ← ((src1_temp - src2) | 1)
        else
                dest ← src1_temp
}
```

Format:

inst dest, src1, src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
DIVAW	a,b,c	00101bbb000010000BBBCCCCCCAAAAAA	
DIVAW	a,b,u6	00101bbb010010000BBBuuuuuuAAAAAA	
DIVAW	b,b,s12	00101bbb100010000BBBssssssSSSSSS	
DIVAW<.cc>	b,b,c	00101bbb110010000BBBCCCCCC0QQQQQ	
DIVAW<.cc>	b,b,u6	00101bbb110010000BBBuuuuuu1QQQQQ	
DIVAW	a,limm,c	00101110000010000111CCCCCCAAAAAA	L
DIVAW	a,b,limm	00101bbb000010000BBB111110AAAAAA	L
DIVAW<.cc>	b,b,limm	00101bbb110010000BBB111110QQQQQQ	L
Without Result			
DIVAW	0,b,c	00101bbb000010000BBBCCCCCC111110	
DIVAW	0,b,u6	00101bbb010010000BBBuuuuuu111110	
DIVAW<.cc>	0,limm,c	00101110110010000111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged
S	□	= Unchanged

Key:

L = Limm Data

Description:

DIVAW is a division accelerator used in the division algorithm as described by the ITU and ETSI. DIVAW accelerates division by generating a fractional result from a division of the integer operand 1 (numerator) by the integer operand 2 (denominator).

The integer numerator format is shown in Figure 92, the integer denominator format in Figure 93, and the DIVAW result format in Figure 94.

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
16-bit data	zero

Figure 92 DIVAW 16-bit input numerator data format

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
16-bit data	zero

Figure 93 DIVAW 16-bit input denominator data format

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
16-bit remainder	16-bit result (quotient)

Figure 94 DIVAW 16-bit output data format

The status flags are not updated with this instruction therefore the flag setting field, F, is encoded as 0.

The particular code that DIVAW accelerates is shown in the C Description. Repeated execution of DIVAW fifteen times implements a 16-bit conditional add-subtract division algorithm as shown in Assembly Code Example.

Notice that for the set of 15 executions of the DIVAW instruction in the Assembly Code Example:

- The result is positive.
- Both numerator and denominator must be positive and the denominator must have a non-zero value that is greater or equal to numerator.
- If NUMERATOR = DENOMINATOR then the result of the division algorithm is 0x00007FFF (assuming non-zero numerator and denominator).
- If NUMERATOR is zero, regardless of value of DENOMINATOR, the returned result is zero.
- The 16-bit result (quotient) is in the low word of the destination register.
- The 16-bit remainder is in the high word of the destination register.

C Description:

```
Word16 div_s(Word16 num, Word16 denom)
{
   Word16 var_out =0;
   Word16 iteration;
   Word32 Lm;
   Word32 L_denom;

   Lm = (num)<<15;
   L_denom = (denom)<<15;

/* DIVAW can be iterated to perform this section of code */
   for(iteration=0;iteration<15;iteration++)
   {
      Lm << 1;
      if (Lm >= L_denom)
      {
          Lm = L_sub(Lm,L_denom);/* 32-bit subtract*/
          Lm = L_add(Lm,1);
      }
   }   ; remainder in MSW of Lm quotient in LSW of Lm

   var_out = (short) Lm;
   return(var_out);
}
```

Pseudo Code Example:

```
if (src1 == 0)                                    /* DIVAW */
  dest = 0
else
{
  src1_temp =  src1 << 1
  if (src1_temp >= src2)
    dest = ((src1_temp - src2) | 0x0000_0001)
  else
    dest = src1_temp
}
```

Assembly Code Example:

```
; Input: Data is in the LSW of r0 (Lm) and r1 (L_denom)

ASL r0, r0, 15
ASL r1, r1, 15

; Division:
.rep 15
DIVAW r0, r0, r1
.endr

; Remainder in MSW of r0
; Quotient in LSW of r0

AND %r0, %r0, 0x0000_7fff ;mask to leave quotient in LSW
```

EX

Atomic Exchange

Memory Operation

Operation:

dest ← Result of memory load from address @ src

memory @ src ← dest

Format:

inst dest, src

Format Key:

src	=	Source Operand
dest	=	Destination

Syntax:

		Instruction Code	
EX<.di>	b,[c]	`00100bbb00101111DBBBCCCCCC001100`	
EX<.di>	b,[u6]	`00100bbb01101111DBBBuuuuuu001100`	
EX<.di>	b,[limm]	`00100bbb00101111DBBB111110001100`	L

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

LD	ST
SYNC	

Description:

An atomic exchange operation, EX, is provided as a primitive for multiprocessor synchronization allowing the creation of semaphores in shared memory.

Two forms are provided: an uncached form (using the .DI directive) for synchronization between multiple processors, and a cached form for synchronization between processes on a single-processor system.

The EX instruction exchanges the contents of the specified memory location with the contents of the specified register. This operation is atomic in that the memory system ensures that the memory read and memory write cannot be separated by interrupts or by memory accesses from another processor.

The status flags are not updated with this instruction.

An immediate value is not permitted to be the destination of the exchange instruction. Using the long immediate indicator in the destination field, B=0x3E, will raise a Instruction Error exception.

NOTE When used in translated memory, both the read and write permissions must be set in order for EX to operate without causing a protection violation exception.

Pseudo Code Example:

```
temp = dest                                       /* EX */
dest = Memory(src)
Memory(src) = temp
```

Assembly Code Example:

In this example the processor attempts to get access to a shared resource by testing a semaphore against values 0 and 1.

- If the returned value is a 0 then the resource was free and this device is now the owner.
- If the returned value is a 1, the resource is busy and the processor must wait till a 0 is returned.

The value 1 is always written to SEMPHORE_ADDR so all processes trying to own the semaphore should all write the same value.

The value at SEMPHORE_ADDR should not be used for a determination of the current owner of the semaphore.

Example 18 To obtain a semaphore using EX

```
wait_for_resource:
  MOV R2, 0x00000001         ; indicates semaphore is owned

wfr1:
  EX   R2,                   ; exchange r2 and semaphore
[SEMAPHORE_ADDR]             ; see if we own the semaphore
  CMP_S R2, 0                ; wait for resource to free
  BNE wfr1
```

Example 19 To Release Semaphore using ST

```
release_resource:
  MOV R2, 0x00000000         ; indicates semaphore is free
  ST  R2, [SEMAPHORE_ADDR]   ; release semaphore
```

EXTB

Zero Extend Byte

Arithmetic Operation

Operation:

dest ← zero extend from byte (src)

Format:

inst dest, 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. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

dest ← zero extend from word (src)

Format:

inst dest, 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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

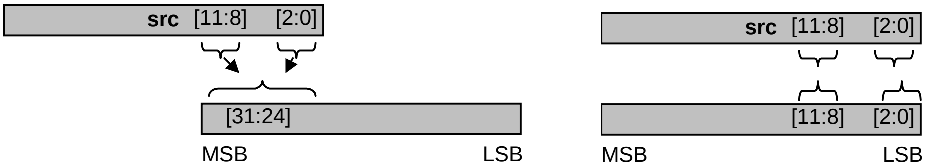
FLAG

Set Flags

Control Operation

Operation:

if (cc=true) then flags ← src



STATUS Register
Auxiliary (0x00)

STATUS32 Register
Auxiliary (0x0A)

Format:

inst src

Format Key:

src = Source Operand

Syntax:

		Instruction Code	
FLAG<.cc>	c	00100rrr111010010RRRCCCCCC0QQQQQ	
FLAG<.cc>	u6	00100rrr111010010RRRuuuuuu1QQQQQ	
FLAG<.cc>	limm	00100rrr111010010RRR1111100QQQQQ	L
FLAG	s12	00100rrr101010010RRRssssssSSSSSS	

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)
L		= Unchanged
U		= Unchanged
DE		= Unchanged
AE		= Unchanged
A2		= Unchanged
A1		= Unchanged

Key:

L = Limm Data

Related Instructions:

SLEEP BRK

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.

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).

If the H flag is set (halt processor flag), all other flag states are ignored and are not updated.

In the ARC 700 processor, the FLAG instruction is serializing – ensuring that no further instructions can be completed before any flag updates take effect.

The halt flag, H, and interrupt enable flags, E1 and E2, can only be set in Kernel mode.

Bits L, U, DE, AE, A2, A1 in the STATUS32 register may not be set with the FLAG instruction. These are updated by the processor changing state or by the raise-exception instruction, TRAP, and the return from interrupt/exception instructions, RTIE, J.F [ILINK1] and J.F [ILINK2].

Both the (obsolete) Status Register (auxiliary address 0x00) and STATUS32 register (auxiliary address 0x0A) are updated automatically upon using the FLAG instruction. The flag setting field, F, is always encoded as 0 for this instruction.

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

Jump Operation

Operation:

if (cc=true) then cPC ← src **

Format:

inst 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>:

Code	Q Field	Description	Test	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

Code	Q Field	Description	Test	Code	Q Field	Description	Test
							/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 Bcc

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 as the absolute address 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.

The ARC 700 processor will raise an Illegal Instruction Sequence exception if an executed delay slot contains:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt (RTIE)
- Any instruction with long-immediate data as a source operand

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.

When using ILINK1 or ILINK2 the flag setting field, F, is always encoded as 1 for this instruction. The reserved field, R, is ignored by the processor but should be set to 0.

If the ILINK1 or ILINK2 registers are used without the flag setting field being set an Instruction Error exception will be raised. If the flag setting field, F, is set without using the ILINK1 or ILINK2 register, an Instruction Error exception will be raised.

For the ARC 700 processor it is recommended that the RTIE instruction is used to return from an interrupt service routine.

In the ARC 700 processor the appropriate BTA link register is also loaded into BTA when jump-based interrupt return is executed.

The operation of J.F [ILINK1] or J_S.F [ILINK1] is thus:

- PC ← ILINK1
- STATUS32 ← STATUS32_L1
- BTA ← BTA_L1

The operation of J.F [ILINK2] or J_S.F [ILINK2] is now as follows:

- PC ← ILINK2
- STATUS32 ← STATUS32_L2
- BTA ← BTA_L2

As with RTIE, if the STATUS32[DE] bit becomes set as a result of the J_S.F [ILINKn] or Jcc.F [ILINKn] instruction, the processor will be put back into a state where a branch with a delay slot is pending. The target of the branch will be contained in the BTA register. The value in BTA will have been restored from the appropriate Interrupt Return BTA register (BTA_L1 or BTA_L2).

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
  BTA = BTA_L1 ;ARC 700 only
 if F==1 and src==ILINK2 then
  STATUS32 = STATUS32_L2
  BTA = BTA_L2 ;ARC 700 only
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
```

JLcc

Jump and Link Conditionally

Jump Operation

Operation:

if (cc=true) then (cPC ← src) & (BLINK ← nPC)

Format:

inst src

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>:

Code	Q Field	Description	Test	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	(N and V) or (/N and /V)

Code	Q Field	Description	Test	Code	Q Field	Description	Test
						(signed)	
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 BLcc

Description:

If the specified condition is met (cc=true), then the program execution is resumed from the new program counter address that is specified as the absolute address 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.

The ARC 700 processor will raise an Illegal Instruction Sequence exception if an executed delay slot contains:

- Another jump or branch instruction
- Conditional loop instruction (LPcc)
- Return from interrupt (RTIE)
- Any instruction with long-immediate data as a source operand

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
```

LD

Delayed Load from Memory

Memory Operation

Operation:

dest ← Result of Memory Load address @ (src1+src2)

Format:

inst dest, 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]	00010110000000000111DRRZZXAAAAAA	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<zz><.x><.aa><.di>	0,[b,s9]	00010bbbssssssssSBBBDaaZZX111110	
LD<zz><.x><.di>	0,[limm]	00010110000000000111DRRZZX111110	L
LD<zz><.x><.aa><.di>	0,[b,c]	00100bbbaa110ZZXDBBBCCCCCC111110	
LD<zz><.x><.aa><.di>	0,[b,limm]	00100bbbaa110ZZXDBBB111110111110	L
LD<zz><.x><.di>	0,[limm,c]	00100110RR110ZZXD111CCCCCC111110	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 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 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).

CAUTION The addition of src1 to src2 is performed with a simple 32-bit adder which is independent of the ALU. No exception occurs if a carry or overflow occurs. The resultant calculated address may overlap into unexpected regions depending of the values of src1 and src2.

The status flags are not updated with this instruction.

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 for the ARCtangent-A5 and ARC 600 processors and should not be used.

Using the sign extend suffix on the LD instruction with a 32-bit data size will raise an Instruction Error exception on the ARC 700 processor.

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).

For the ARC 600 processor loads to a null register using the long-immediate data indicator should be avoided.

For the ARC 700 processor loads to a null register using the long-immediate data performs a pre-fetch operation

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

Branch Operation

Operation:

if (cc=false) then cPC ← (cPCL+rd) else (LP_END ← cPCL+rd) & (LP_START ← nPC)

Format:

inst rel_addr

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>:

Code	Q Field	Description	Test	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

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.

CAUTION The LPcc instruction should not be in the executed delay slot of branch and jump instructions, and therefore the LPcc instruction cannot immediately follow a Bcc.D, BLcc.D, Jcc.D, JLcc.D, BRcc.D or BBITn.D 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, that before executing the code that LP_END is initialized to a safe value (i.e. 0) to prevent accidental LP triggering. Similar caution is required if using any form of MMU or memory mapping.

The LP instruction is encoded to use immediate values (syntax u7 or syntax s12). Encoding the operand mode (bits 23:22) to be 0x0 or 0x1 is not recommended. Additionally using operand mode 0x3 with sub-operand mode 0x0 is not recommended. The reserved field, R, is ignored by the processor.

The LP instruction may be used to set up a loop with a maximum set by the limit of the branch offset available in the LP instruction used.

- Conditional branch – 6 bits of unsigned offset gives +128 bytes
- Unconditional branch – 12 bits of signed offset gives +4094/-4096 bytes

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.

For the ARCompact based processor, 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:

- When a branch instruction appears in the last instruction fetch of the loop.
- When the delay slot of a branch appears in the last instruction fetch of a loop (i.e. a branch with a delay slot is the penultimate instruction fetch of the loop).

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.

The use of zero delay loops is illustrated in the following example.

Example 20 Example Loop Code

```
             MOV    LP_COUNT,2    ; do loop 2 times (flags not set)
             ...                  ; Some intermediate instructions
             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
```

Direct writes to the LP_START and LP_END registers should be used to set up larger loops, if required.

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.

ARCtangent-A5 Loop Operation

For the ARCtangent-A5 processor, the operation of the loop mechanism is such that NEXT_PC is constantly compared with the value LP_END. If the comparison is true, then LP_COUNT is tested. If LP_COUNT is not equal to 1, then the PC is loaded with the contents of LP_START, and LP_COUNT is decremented. If, however, LP_COUNT is 1, then the PC is allowed to increment normally and LP_COUNT is decremented. This is illustrated in Figure 95 on page 249.

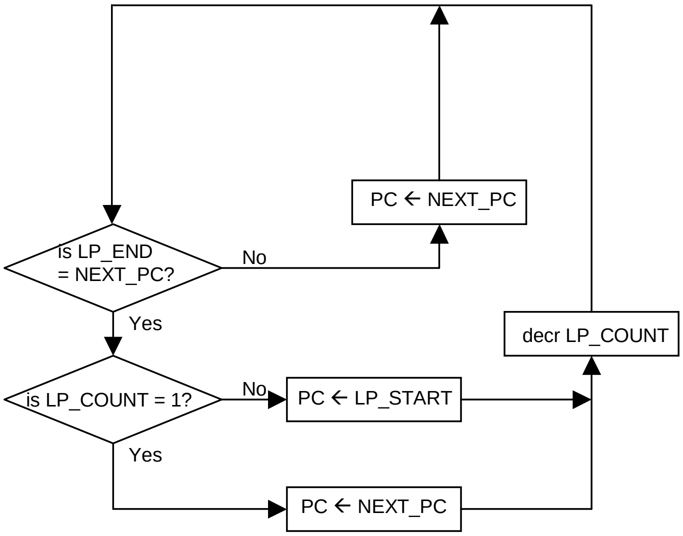


Figure 95 Loop Detection and Update Mechanism, ARCtangent-A5

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. For the ARCtangent-A5 processor, unpredictable behavior will result when LP_START or LP_END are set to point to any other locations.

For the ARCtangent-A5 processor, the LP instruction must not be used to set up loops with a single instruction word. The LP instruction can only set up loops containing at least two instruction words.

This means that the LP instruction can be used to set up a loop containing a single instruction that references long immediate data – since it has in fact two instruction words.

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. Example 21 on page 250 shows this. 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.

Example 21 Setting up an ARCtangent-A5 Single Instruction Loop

```
             MOV    LP_COUNT,5     ; no. of times to do loop
             MOV    r0,dooploop    ; load START loop address
             MOV    r1,dooploopend ; 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
dooploop:    OR     r21,r22,r23   ; single instruction in loop
dooploopend: ADD r19,r19,r20      ; first instruction after loop
```

There are also rules about where SLEEP and BRK instructions may be placed within zero-overhead loops. The programmer should never insert a BRK or a SLEEP as the last instruction in a zero overhead loop. To summarize the effect that the loop mechanism has on these special cases see the following tables, according to the the 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.

The following table covers loop setup and use of long immediate data for the ARCtangent-A5 processor.

Table 88 Loop setup and long immediate data, ARCtangent-A5

	Loop Set Up LP loop_end	**Writing** LP_COUNT	**Reading** LP_COUNT	**Writing** LP_END, LP_START	**Reading** LP_END, LP_START	**Long Imm.** op limm
Loop_st:						
Ins1	...	...	...	...	...	...
Ins2	...	...	...	...	...	...
Ins3	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
Insn-4	...	…	...	...	...	...
Insn-3	...	?^{1}	...	...	...	...
Insn-2	…	?^{1}	...	x	...	...
Insn-1	x	?^{1}	…	x	...	...
Insn	x	?^{1}	?^{2}	x	...	n/a
Loop_end:						
Outins1						
Outins2						

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.
x	An instruction of this type may not be executed in this instruction slot.
n/a	Instructions using long immediate data take two slots. Hence the instruction itself

	cannot be present in the last instruction slot.

The following tables cover use of branch and jump instructions for the ARCtangent-A5 processor:

Table 89 Branch and Jumps in loops, flow(1), ARCtangent-A5

	Bcc Jcc [Rn]	BRcc BBITn	Bcc.d Jcc.d J_S.d BLcc JLcc	BLcc.d JLcc.d JL_S.d	BRcc.d BBITn.d
Loop_st:					
Ins1	...	...	...	...	...
Ins2	...	...	...	...	...
Ins3	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
Insn-4	...	...	...	...	...
Insn-3	...	...	...	...	...
Insn-2	...	!	...	...	!
Insn-1	!	!	!	x	!
Insn	!	!	x	x	x
Loop_end:					
Outins1					
Outins2					
Key:					
!	Loop count register value unpredictable when branch taken to exit early from the loop.				
x	An instruction of this type may not be executed in this instruction slot.				

Table 90 Branch and Jumps in loops, flow(2), ARCtangent-A5

	Jcc limm	JLcc limm	LP other_loop	SLEEP BRK
Loop_st:				
Ins1	...	...	...	...
Ins2	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Insn-2	...	...	...	...
Insn-1	!	x	x	...
Insn	n/a	n/a	x	x
Loop_end:				
Outins1				
Outins2				
Key:				
!	Loop count register value unpredictable when branch taken to exit early from the loop.			
x	An instruction of this type may not be executed in this instruction slot.			
n/a	Instructions using long immediate data take two slots. Hence the instruction itself cannot be present in the last instruction slot.			

ARC 600 Loop Operation

The ARC 600 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 96 on page 252.

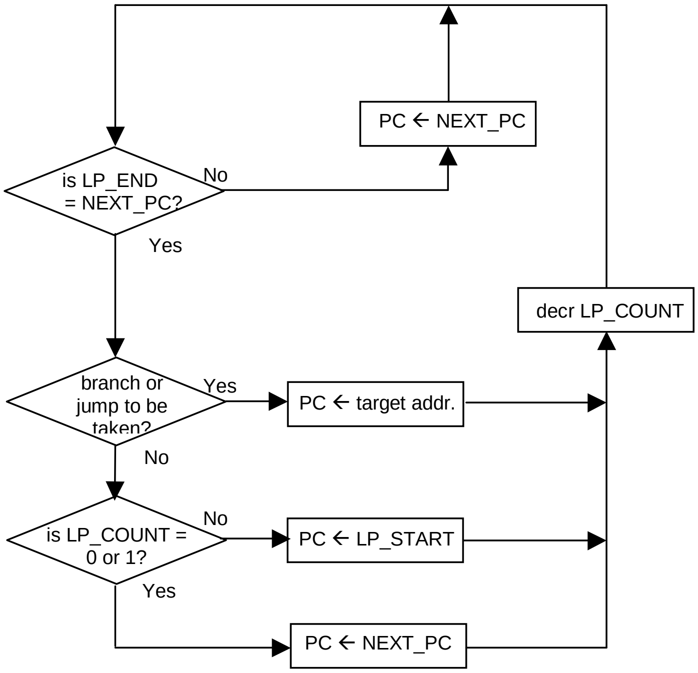


Figure 96 Loop Detection and Update Mechanism, ARC 600

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. For the ARC 600 processor, unpredictable behavior will result when LP_START or LP_END are set to point to any other locations.

For the ARC 600 processor, the LP instruction must not be used to set up loops with a single instruction word. The LP instruction can only set up loops containing at least two instruction words. This means that the LP instruction can be used to set up a loop containing a single instruction that references long immediate data – since it has in fact two instruction words.

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. Example 22 on page 253 shows this. 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.

Example 22 Setting up an ARC 600 Single Instruction Loop

```
             MOV    LP_COUNT,5     ; no. of times to do loop
             MOV    r0,dooploop    ; load START loop address
             MOV    r1,dooploopend ; 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
dooploop:    OR     r21,r22,r23  ; single instruction in loop
dooploopend: ADD r19,r19,r20     ; first instruction after loop
```

There are also rules about where SLEEP and BRK instructions may be placed within zero-overhead loops. The programmer should never insert a BRK or a SLEEP as the last instruction in a zero overhead loop.

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.

The following table covers loop setup and use of long immediate data for the ARC 600 processor.

Table 91 Loop setup and long immediate data, ARC 600

	Loop Set Up LP loop_end	**Writing** LP_COUNT	**Reading** LP_COUNT	**Writing** LP_END, LP_START	**Reading** LP_END, LP_START	**Long Imm.** op limm
Loop_st:						
Ins1	...	...	...	...	...	...
Ins2	...	...	...	...	...	...
Ins3	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
Insn-4	...	...	...	...	...	...
Insn-3	...	?^{1}	...	...	...	...
Insn-2	...	?^{1}	...	x	...	...
Insn-1	x	?^{1}	…	x	...	...
Insn	x	?^{1}	?^{2}	x	...	n/a
Loop_end:						
Outins1						
Outins2						
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.					
x	An instruction of this type may not be executed in this instruction slot.					
n/a	Instructions using long immediate data take two slots. Hence the instruction itself cannot be present in the last instruction slot.					

The following tables cover use of branch and jump instructions for the ARC 600 processor:

Table 92 Branch and Jumps in loops, flow(1), ARC 600

	Bcc Jcc [Rn]	BRcc BBITn	BRcc limm BBITn limm	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	...	...	x	...	x	x	x	x
Insn	x^2	x	x	x	x	x	x	x
Loop_end:								
Outins1								
Outins2								

Key:	
x	An instruction of this type may not be executed in this instruction slot.
x^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.

Table 93 Branch and Jumps in loops, flow(2), ARC 600

	Jcc limm	JLcc limm	LP other_loop	SLEEP BRK SWI
Loop_st:				
Ins1	...	...	...	...
Ins2	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Insn-2	...	...	...	...
Insn-1	x	x	x	...
Insn	x	x	x	x
Loop_end:				
Outins1				
Outins2				

Key:	
x	An instruction of this type may not be executed in this instruction slot.

ARC 700 Loop Operation

For the ARC 700 processor, the loop mechanism is active when the loop-inhibit bit STATUS32[L] is set to zero. This bit is set to disable the loop mechanism on an interrupt or an exception (including TRAP instructions). Loops are enabled (STATUS32[L]=0) after Reset. The loop-inhibit bit is cleared (loops allowed) whenever the processor commits a taken conditional LP instruction or an unconditional LP instruction. From kernel mode, the value of the bit can also be set/restored using the RTIE instruction.

When the loop mechanism is disabled (STATUS32[L]=1), loop-end conditions are ignored - no change of program flow is taken, loop count is not decremented. The STATUS32[L] register does not affect reads and writes to the loop control registers.

The machine checks for a loop-end condition when calculating the next program counter address, before each instruction is completed.

A loop-end condition is detected when:

- The instruction to be completed is not a *taken* branch or jump - note this includes a LPcc which evaluates false.
  - In the case of a *taken* branch or jump, the loop-end condition is bypassed, and the next instruction (NEXT_PC) comes from the branch/jump target.
- STATUS32[DE] is 0 and BTA[0] = - the instruction is not in the delay slot instruction of a branch.
  - In the case when STATUS32[DE] = 1 and BTA[0] = 1, the instruction pointed to by PC is the delay slot instruction of a branch, therefore the next instruction (NEXT_PC) comes from the address in the Branch Target Address (BTA) register.
  - In the case when STATUS32[DE] = 1 and BTA[0] = 0, the preceeding branch was not-taken, therefore the current instruction is still considered as end-of-loop.
- STATUS32[L] is 0
  - This bit is set to 1 to disable loop-end detection.
- The instruction to be completed is the last in a loop
  - Current PC + current instruction_size = LP_END
- LP_COUNT is not equal to 1
  - In the case when LP_COUNT=1, LP_COUNT is decremented and execution continues from the instruction pointed to by LP_END.

When a loop-end condition is detected, the machine jumps to the address in LP_START, and LP_COUNT is decremented.

If LP_COUNT is 1, then the machine will continue execution from the instruction pointed to by LP_END; LP_COUNT is also decremented. This is illustrated in the following diagram.

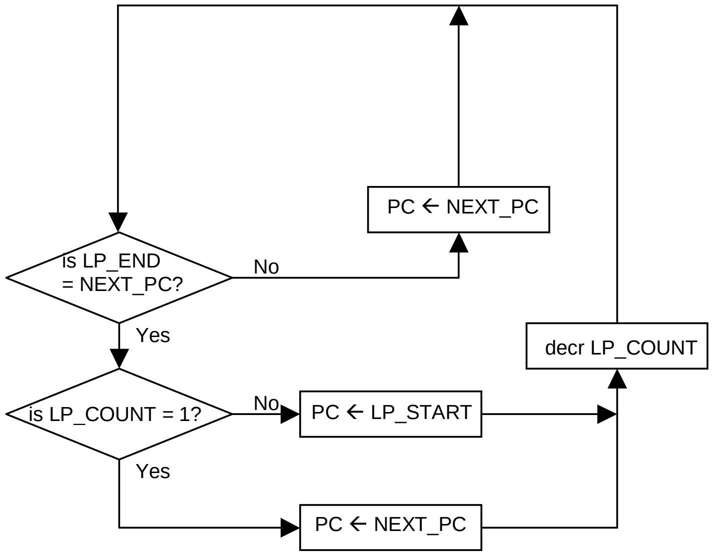


Figure 97 Loop Detection and Update Mechanism, ARC 700

The ARC 700 processor allows the LP instruction to be used to set up a loop with a minimum of one instruction

If a LP_START value is provided which does not match the start of an instruction, and the loop-end condition is reached, the result will the same as if a branch or jump had been made to the faulty address.

If a LP_END value is provided which does not match the start of an instruction, the loop-end condition will never be detected.

The update to the LP_START and LP_COUNT registers will take effect immediately after the LP instruction has committed. Note that any change of program flow required (i.e. jump to LP_START) will be completed before LP_START and LP_END are updated.

As a result, executing a LP instruction from the last instruction in the loop will take effect from the next loop iteration. Executing LP from any other position in the loop will take effect in the current loop iteration.

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.

The following table covers loop setup and use of long immediate data for the ARC 700 processor.

Table 94 Loop setup and long immediate data, ARC 700

	Loop Set Up
	`LP`
	`loop_end`
		Writing
		`LP_COUNT`
			Reading
			`LP_COUNT`
				Writing
				`LP_END,`
				`LP_START`
					Reading
					`LP_END,`
					`LP_START`
						Long Imm.
						`op limm`

`Loop_st:`						
`Ins1`	...	...	...	...	...	...
`Ins2`	...	...	...	...	...	...
`Ins3`	...	...	...	...	...	...
`...`	...	...	...	...	...	...
`...`	...	...	...	...	...	...
`Insn-4`	...	...	...	...	...	...
`Insn-3`	...	...	...	...	...	...
`Insn-2`	...	...	...	...	...	...
`Insn-1`	...	...	...	...	...	...
`Insn`	n	...	n	n	...	n/a
`Loop_end:`						
`Outins1`						
`Outins2`						
Key:						
`n`	Updates to loop registers take affect after loop end condition has been evaluated, i.e. in the next loop iteration					
`n/a`	Instructions using long immediate data take two slots. Hence the instruction itself cannot be present in the last instruction slot.					

The following tables cover use of branch and jump instructions for the ARC 700 processor:

Table 95 Branch and Jumps in loops, flow(1), ARC 700

	`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`	...	...	...	...	o	...
`Insn`	...	...	o	x	x	x
`Loop_end:`						
`Outins1`						
`Outins2`						
Key:						
`x`	An instruction of this type may not be executed in this instruction slot. An Illegal Instruction Sequence exception is taken if the instruction is attempted.					
`o`	Return address will be outside the loop					

Table 96 Branch and Jumps in loops, flow(2), ARC 700

	Jcc limm	JLcc limm	LP other_loop	SLEEP BRK
Loop_st:				
Ins1	...	...	...	...
Ins2	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Insn-2	...	...	...	...
Insn-1	...	...	...	...
Insn	...	o	n	...
Loop_end:				
Outins1				
Outins2				
Key:				
n	Updates to loop registers take affect after loop end condition has been evaluated, i.e. in the next loop iteration			
o	Return address will be outside the loop			

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
```

The use of zero delay loops is illustrated below.

```
             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
```

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
```

LR

Load from Auxiliary Register

Control Operation

Operation:

dest ← aux_reg(src)

Format:

inst dest, 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,[u6]	00100bbb01101010 0BBBuuuuuu000000	
LR	b,[s12]	00100bbb10101010 0BBBssssssSSSSSS	

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

SR LD

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).

The status flags are not updated with this instruction therefore the flag setting field, F, is encoded as 0. The reserved field, R, is ignored by the processor, but should be set to 0.

The LR instruction cannot be conditional therefore encoding the operand mode (bits 23:22) to be 0x3 will raise an Instruction Error exception in the ARC 700 processor.

For the ARCtangent-A5 and ARC 600 processors, the behavior is undefined if an LR instruction is encoded using the operand mode of 0x3.

Pseudo Code Example:

```
dest = Aux_reg(src)                               /* LR */
```

Assembly Code Example:

```
LR r1,[r2]          ; Load contents of Aux. register pointed
                    ; to by r2 into r1
```

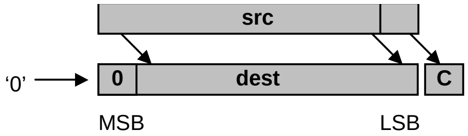
LSR

Logical Shift Right

Logical Operation

Operation:

dest ← LSR by 1 (src)



Format:

inst dest, 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
ASL multiple	ASR multiple
ROR multiple	LSR multiple

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.

Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

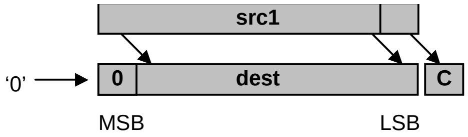
LSR multiple

Multiple Logical Shift Right

Logical Operation

Operation:

if (cc=true) then dest ← logical shift right of src1 by src2



Format:

inst dest, src1, 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
ASL multiple	ASR multiple
ROR multiple	

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.

Any flag updates will only occur if the set flags suffix (.F) is used.

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

Operation:

if (cc=true) then dest ← MAX(src1, src2)

Format:

inst dest, 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	00100110000001000F111CCCCCCAAAAAA	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 CMP

Description:

Return the maximum of the two signed source operands (src1 and src2) and place the result in the destination register (dest). Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then dest ← MIN(src1, src2)

Format:

inst dest, 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	0010011000001001F111CCCCCCAAAAAA	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	0010011011001001F111CCCCCC0QQQQQ	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 CMP

Description:

Return the minimum of the two signed source operands (src1 and src2) and place the result in the destination register (dest). Any flag updates will only occur if the set flags suffix (.F) is used.

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

Operation:

if (cc=true) then dest ← src

Format:

inst 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	hob	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:

EXTB SWAP

EXTW SEXB

Description:

The contents of the source operand (src) are moved to the destination register (dest). Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

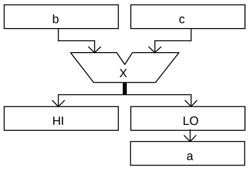
MPY

32 x 32 Signed Multiply Low

Extension Option

Operation:

dest ← (src1 X src2).low



Format:

inst dest, src1, src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
MPY<.f>	a,b,c	00100bbb00011010FBBBCCCCCCAAAAAA	
MPY<.f>	a,b,u6	00100bbb01011010FBBBuuuuuuAAAAAA	
MPY<.f>	b,b,s12	00100bbb10011010FBBBssssssSSSSSS	
MPY<.cc><.f>	b,b,c	00100bbb11011010FBBBCCCCCC0QQQQQ	
MPY<.cc><.f>	b,b,u6	00100bbb11011010FBBBuuuuuu1QQQQQ	
MPY<.f>	a,limm,c	00100110000110 10F111CCCCCCAAAAAA	L
MPY<.f>	a,b,limm	00100bbb00011010FBBB111110AAAAAA	L
MPY<.cc><.f>	b,b,limm	00100bbb11011010FBBB1111100QQQQQ	L
Without Result			
MPY<.f>	0,b,c	00100bbb00011010FBBBCCCCCC111110	
MPY<.f>	0,b,u6	00100bbb01011010FBBBuuuuuu111110	
MPY<.cc><.f>	0,limm,c	00100110110110 10F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	•	= Set when the destination register is zero.
N	•	= Set when the sign bit of the 64-bit result is set
C		= Unchanged
V	•	= Set when the signed result cannot be wholly contained within the lower part of the 64-bit result. In other words, when bits 62:31 do not equal bit 64, the sign bit.

Key:

L = Limm Data

Related Instructions:

MPYH	MPYU
MPYHU	DIVAW

Description:

Perform a signed 32-bit by 32-bit multiply of operand1 and operand2 then place the least significant 32 bits of the 64-bit result in the destination register. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if cc==true then                                              /* MPY */
 dest = (src1 * src2) & 0x0000_0000_FFFF_FFFF
```

Assembly Code Example:

```
MPY r1,r2,r3            ; Multiply r2 by r3
                        ; and put low part of the result in r1
```

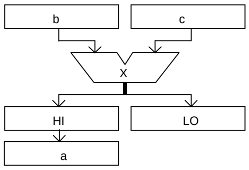
MPYH

32 x 32 Signed Multiply High

Extension Option

Operation:

dest ← (src1 X src2).high



Format:

inst dest, src1, src2

Format Key:

dest = Destination Register
src1 = Source Operand 1
src2 = Source Operand 2

Syntax:

With Result		**Instruction Code**	
MPYH<.f>	a,b,c	00100bbb00011011FBBBCCCCCCAAAAAA	
MPYH<.f>	a,b,u6	00100bbb01011011FBBBuuuuuuAAAAAA	
MPYH<.f>	b,b,s12	00100bbb10011011FBBBssssssSSSSSS	
MPYH<.cc><.f>	b,b,c	00100bbb11011011FBBBCCCCCC0QQQQQ	
MPYH<.cc><.f>	b,b,u6	00100bbb11011011FBBBuuuuuu1QQQQQ	
MPYH<.f>	a,limm,c	00100110000110 11F111CCCCCCAAAAAA	L
MPYH<.f>	a,b,limm	00100bbb00011010FBBB111110AAAAAA	L
MPYH<.cc><.f>	b,b,limm	00100bbb11011011FBBB1111100QQQQQ	L
Without Result			
MPYH<.f>	0,b,c	00100bbb00011011FBBBCCCCCC111110	
MPYH<.f>	0,b,u6	00100bbb01011011FBBBuuuuuu111110	
MPYH<.cc><.f>	0,limm,c	00100110110 11011F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z • = Set when the destination register is zero.
N • = Set when the sign bit of the 64-bit result is set
C = Unchanged
V • = Always cleared.

Key:

L = Limm Data

Related Instructions:

MPY	MPYU
MPYHU	DIVAW

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 the destination register. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if cc==true then                      /* MPYH */
 dest = (src1 * src2) >> 32
```

Assembly Code Example:

```
MPYH r1,r2,r3           ; Multiply r2 by r3 and put high part of the result in r1
```

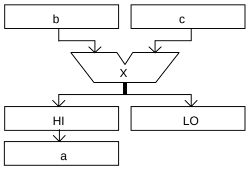
MPYHU

32 x 32 Unsigned Multiply High

Extension Option

Operation:

dest ← (src1 X src2).high



Format:

inst dest, src1, src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
MPYHU<.f>	a,b,c	00100bbb00011100FBBBCCCCCCAAAAAA	
MPYHU<.f>	a,b,u6	00100bbb01011100FBBBuuuuuuAAAAAA	
MPYHU<.f>	b,b,s12	00100bbb10011100FBBBssssssSSSSSS	
MPYHU<.cc><.f>	b,b,c	00100bbb11011100FBBBCCCCCC0QQQQQ	
MPYHU<.cc><.f>	b,b,u6	00100bbb11011100FBBBuuuuuu1QQQQQ	
MPYHU<.f>	a,limm,c	0010011000011100F111CCCCCCAAAAAA	L
MPYHU<.f>	a,b,limm	00100bbb00011100FBBB111110AAAAAA	L
MPYHU<.cc><.f>	b,b,limm	00100bbb11011100FBBB1111100QQQQQ	L
Without Result			
MPYHU<.f>	0,b,c	00100bbb00011100FBBBCCCCCC111110	
MPYHU<.f>	0,b,u6	00100bbb01011100FBBBuuuuuu111110	
MPYHU<.cc><.f>	0,limm,c	0010011011011100F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	•	= Set when the destination register is zero.
N	•	= Always cleared.
C		= Unchanged
V	•	= Always cleared.

Key:

L = Limm Data

Related Instructions:

MPY	MPYU
MPYH	DIVAW

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 the destination register. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if cc==true then                    /* MPYHU */
 dest = (src1 * src2) >> 32
```

Assembly Code Example:

```
MPYHU r1,r2,r3        ; Multiply r2 by r3
                      ; and put high part of the result in r1
```

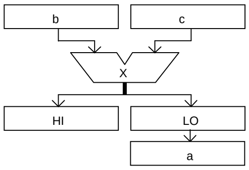
MPYU

32 x 32 Unsigned Multiply Low

Extension Option

Operation:

dest ← (src1 X src2).low



Format:

inst dest, src1, src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
MPYU<.f>	a,b,c	00100bbb00011101FBBBCCCCCCAAAAAA	
MPYU<.f>	a,b,u6	00100bbb01011101FBBBuuuuuuAAAAAA	
MPYU<.f>	b,b,s12	00100bbb10011101FBBBssssssSSSSSS	
MPYU<.cc><.f>	b,b,c	00100bbb11011101FBBBCCCCCC0QQQQQ	
MPYU<.cc><.f>	b,b,u6	00100bbb11011101FBBBuuuuuu1QQQQQ	
MPYU<.f>	a,limm,c	0010011000011101F111CCCCCCAAAAAA	L
MPYU<.f>	a,b,limm	00100bbb00011101FBBB111110AAAAAA	L
MPYU<.cc><.f>	b,b,limm	00100bbb11011101FBBB1111100QQQQQ	L
Without Result			
MPYU<.f>	0,b,c	00100bbb00011101FBBBCCCCCC111110	
MPYU<.f>	0,b,u6	00100bbb01011101FBBBuuuuuu111110	
MPYU<.cc><.f>	0,limm,c	0010011011011101F111CCCCCC0QQQQQ	L

Flag Affected (32-Bit):

Z	•	= Set when the destination register is zero.
N	•	= Always cleared
C		= Unchanged
V	•	= Set when the high part of the 64-bit result is non-zero

Key:

L = Limm Data

Related Instructions:

MPY	MPYH
MPYHU	DIVAW

Description:

Perform an unsigned 32-bit by 32-bit multiply of operand1 and operand2 then place the least significant 32 bits of the 64-bit result in the destination register. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if cc==true then                                  /* MPYU */
 dest = (src1 * src2) & 0x0000_0000_FFFF_FFFF
```

Assembly Code Example:

```
MPYU r1,r2,r3          ; Multiply r2 by r3 and put low part of the result in r1
```

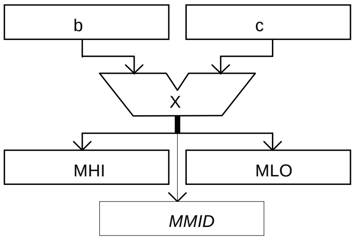
MUL64

32 x 32 Signed Multiply

Extension Option

Operation:

MLO ← low part of (src1 * src2)
MHI ← high part of (src1 * src2)
MMID ← middle part of (src1 * src2)



Format:

inst dest, 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 DIVAW

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.

If an instruction condition placed on a MUL64 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.

The extension auxiliary register MULHI is used to restore the high part of multiply result register if the multiply has been used, for example, by an interrupt service routine. The lower part of the multiply result register can be restored by multiplying the desired value by 1.

The status flags are not updated with this instruction therefore the flag setting field, F, should be encoded as 0.

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
```

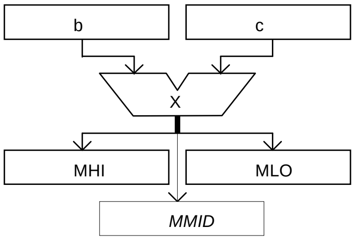
MULU64

32 x 32 Unsigned Multiply

Extension Option

Operation:

MLO ← low part of (src1 * src2)
MHI ← high part of (src1 * src2)
MMID ← middle part of (src1 * src2)



Format:

inst dest, 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 DIVAW

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.

If an instruction condition placed on a 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.

The extension auxiliary register MULHI is used to restore the high part of multiply result register if the multiply has been used, for example, by an interrupt service routine. The lower part of the multiply result register can be restored by multiplying the desired value by 1.

The status flags are not updated with this instruction therefore the flag setting field, F, should be encoded as 0.

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

Negate

Arithmetic Operation

Operation:

dest ← 0 - src

Format:

inst dest, src

Format Key:

src	=	Source Operand
dest	=	Destination

Syntax:

		Instruction Code
NEG<.f>	a,b	`00100bbb01001110FBBB000000AAAAAA`
NEG<.cc><.f>	b,b	`00100bbb11001110FBBB0000001QQQQQ`
NEG_S	b,c	`01111bbbccc10011`

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:

ABS RSUB

Description:

The negate instruction subtracts the source operand (src) from zero and places the result into the destination register (dest). Any flag updates will only occur if the set flags suffix (.F) is used.

NOTE The 32-bit instruction format is an encoding of the reverse subtract instruction using an unsigned 6-bit immediate value set to 0.

Pseudo Code Example:

```
dest = 0 - src                              /* NEG */
```

Assembly Code Example:

```
NEG r1,r2              ; Negate r2 and write result
                       ; into r1
```

NEGS

Negate with Saturation

Extended Arithmetic Operation

Operation:

dest ← sat_{32}(0-src)

Format:

inst dest, src

Format Key:

dest	=	Destination Register
src	=	Source Operand 1

Syntax:

With Result		**Instruction Code**	
NEGS<.f>	b,c	`00101bbb00101111FBBBCCCCCC000111`	
NEGS<.f>	b,u6	`00101bbb01101111FBBBuuuuuu000111`	
NEGS<.f>	b,limm	`00101bbb00101111FBBB111110000111`	L
Without Result			
NEGS<.f>	0,c	`0010111000101111F111CCCCCC000111`	
NEGS<.f>	0,u6	`0010111001101111F111uuuuuu000111`	
NEGS<.f>	0,limm	`0010111000101111F111111110000111`	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V	●	= Set if input is 0x8000_0000 otherwise cleared
S	●	= Set if input is 0x8000_0000 (‘sticky’ saturation)

Key:

L	= Limm Data

Related Instructions:

SAT16

RND16

NEGSW

ABSS

Description:

Negate the 32-bit operand with saturation and place the result in the destination register. Note that, NEGS 0x8000_0000 yields 0x7FFF_FFFF. Both saturation flags S1 and S2 will be set if the result of the instruction saturates. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if src==0x8000_0000                                 /* NEGS */
 sat = 1                                            // Using
 dest = 0x7FFF_FFFF                                 // unsigned
else                                                // pseudo
 sat = 0                                            // arithmetic
 dest = 0 - src
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 V_flag = sat
 S_flag = S_flag || sat
```

Assembly Code Example:

```
NEGS r1,r2              ; Negate and saturate the value of
                        ; r2 and write result into r1
```

NEGSW

Negate Word with Saturation

Extended Arithmetic

Operation:

dest ← sat_{16}(0-src.low)

Format:

inst dest, src

Format Key:

dest = Destination Register
src = Source Operand 1

Syntax:

With Result		**Instruction Code**	
NEGSW<.f>	b,c	00101bbb00101111FBBBCCCCCC000110	
NEGSW<.f>	b,u6	00101bbb01101111FBBBuuuuuu000110	
NEGSW<.f>	b,limm	00101bbb00101111FBBB111110000110	L
Without Result			
NEGSW<.f>	0,c	0010111000101111F111CCCCCC000110	
NEGSW<.f>	0,u6	0010111001101111F111uuuuuu000110	
NEGSW<.f>	0,limm	0010111000101111F111111110000110	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V	●	= Set if input is 0x8000 otherwise cleared
S	●	= Set if input is 0x8000 ('sticky' saturation)

Key:

L	= Limm Data

Related Instructions:

SAT16
RND16
NEGS
ABSSW

Description:

Obtain the negated value of the least significant word (LSW) of 32-bit operand with saturation. Place the result in the LSW of the destination register with MSW being sign extended. Note that, negate of 0xFFFF_8000 yields 0x0000_7FFF. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
src16 = src & 0x0000_FFFF                          /* NEGSW */
if src16 <= 0x7FFF                                 // Using
 sat = 0                                           // unsigned
 dest = 0 - src16                                  // pseudo
else                                               // arithmetic
 sat = 0
 dest = 0x0000_0000 - src16
if src16==0x8000
 sat = 1
 dest = 0x0000_7FFF
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 V_flag = sat
 S_flag = S_flag || sat
```

Assembly Code Example:

```
NEGSW r1,r2            ; Negate the LSW value of r2 and write result into r1
```

NOP

No Operation

Control Operation

Operation:

No Operation

Format:

inst

Format Key:

inst = Instruction

Syntax:

	Instruction Code
NOP_S	0111100011100000
NOP	00100110010010100111000000000000

Flag Affected (32-Bit):

Z □ = Unchanged
N □ = Unchanged
C □ = Unchanged
V □ = Unchanged

Key:

L = Limm Data

Related Instructions:

UNIMP_S

Description:

No operation. The state of the processor is not changed by this instruction. The 32-bit NOP is an encoding of the MOV instruction (syntax MOV 0,u6) using the General Operations Register with Unsigned 6-bit Immediate format on page 143.

Pseudo Code Example:

```
/* NOP_S */
```

Assembly Code Example:

```
NOP_S                   ; No operation
```

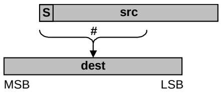
NORM

Normalize

Extension Option

Operation:

dest ← normalization integer of src



Format:

inst dest, 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:

EXTB SEXB
NORMW

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". Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

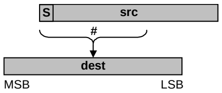
NORMW

Normalize Word

Extension Option

Operation:

dest ← normalization integer of src



Format:

inst dest, 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:

EXTW SEXW
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". Any flag updates will only occur if the set flags suffix (.F) is used. 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
```

NOT

Logical Bitwise NOT

Logical Operation

Operation:

dest ← NOT(src)

Format:

inst dest, 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 NEG

Description:

Logical bitwise NOT (inversion) of the source operand (src) with the result placed into the destination register (dest). Any flag updates will only occur if the set flags suffix (.F) is used.

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               ; Logical bitwise NOT r2 and
                        ; write result into r1
```

OR

Logical Bitwise OR

Logical Operation

Operation:

if (cc=true) then dest ← (src1 OR src2)

Format:

inst dest, src1, 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	0010011000000101F111CCCCCCAAAAAA	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). Any flag updates will only occur if the set flags suffix (.F) is used.

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

Operation:

dest ←Result of Memory Load from Address [sp] then sp ← sp+4

Format:

inst dest

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 LD

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). The status flags are not updated with this instruction.

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
```

PREFETCH

Prefetch from Memory

Memory Operation

Operation:

prefetch @ (src1+src2)

Format:

inst src1, src2

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2 (Offset)

Syntax:

		Instruction Code	
PREFETCH<.aa>	[b,s9]	`00010bbbssssssssSBBB0aa000111110`	
PREFETCH	[limm]	`000101100000000001110RR000111110`	L
PREFETCH<.aa>	[b,c]	`00100bbbaa1100000BBBCCCCCC111110`	
PREFETCH<.aa>	[b,limm]	`00100bbbaa1100000BBB111110111110`	L
PREFETCH	[limm,c]	`00100110RR1100000111CCCCCC111110`	L

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

NOTE 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

Related Instructions:

LD

ST

POP_S

Description:

The PREFETCH instruction is provided as a synonym for a particular encoding of the LD instruction.

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 loaded into the data cache. The returning load is not written to any core register.

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 set to long-word. The status flags are not updated with this instruction.

Pseudo Code Example:

```
if AA==0 then address = src1 + src2      /* PREFETCH */
if AA==1 then address = src1 + src2
if AA==2 then address = src1
if AA==3 then
 address = src1 + (src2 << 2)
if AA==1 or AA==2 then
 src1 = src1 + src2
DEBUG[LD] = 1

if NoFurtherLoadsPending() then          /* On Returning Load */
 DEBUG[LD] = 0
```

Assembly Code Example:

```
PREFETCH [r1,4]       ; Prefetch long word from memory
                      ; address r1+4
```

PUSH_S

Push onto Stack

Memory Operation

Operation:

sp ← sp-4 then Memory Write Address [sp] ← src

Format:

inst 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). The status flags are not updated with this instruction.

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
```

RCMP

Reverse Comparison

Arithmetic Operation

Operation:

if (cc=true) then src2 – src1

Format:

inst src1, src2

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2
cc	=	Condition Code

Syntax:

		Instruction Code	
RCMP	b,s12	00100bbb10011011BBBssssssSSSSSS	
RCMP<.cc>	b,c	00100bbb11001101 1BBBCCCCCC0QQQQQ	
RCMP<.cc>	b,u6	00100bbb110011011BBBuuuuuu1QQQQQ	
RCMP<.cc>	b,limm	00100bbb110011011BBB1111100QQQQQ	L
RCMP<.cc>	limm,c	00100110110011011111CCCCCC0QQQQQ	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. The flag setting field, F, is always encoded as 1 for this instruction.

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
```

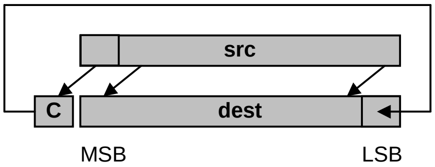
RLC

Rotate Left Through Carry

Logical Operation

Operation:

dest ← RLC by 1 (src)



Format:

inst dest, 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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

RND16

Two’s complement Rounding

Extended Arithmetic Operation

Operation:

dest ← (sat_{32}(src+0x00008000) & 0xFFFF0000)>>16

Format:

inst dest, src

Format Key:

dest	=	Destination Register
src	=	Source Operand 1

Syntax:

With Result		**Instruction Code**	
RND16<.f>	b,c	00101bbb00101111FBBBCCCCCC000011	
RND16<.f>	b,u6	00101bbb01101111FBBBuuuuuu000011	
RND16<.f>	b,limm	00101bbb00101111FBBB111110000011	L
Without Result			
RND16<.f>	0,c	0010111000101111F111CCCCCC000011	
RND16<.f>	0,u6	0010111001101111F111uuuuuu000011	
RND16<.f>	0,limm	0010111000101111F111111110000011	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V	●	= Set if result saturated, otherwise cleared
S	●	= Set if result saturated (‘sticky’ saturation)

Key:

L	= Limm Data

Related Instructions:

ABSSW	ABSS
SAT16	NEGSW

Description:

Round the 32-bit source operand into its most significant word (MSW) using two’s compliment rounding with saturation. Place the result in the LSW of the destination register with the MSW of the result being sign extended. Any flag updates will only occur if the set flags suffix (.F) is used.

Two’s complement rounding is equivalent to adding 0x0000_8000 to the 32-bit input, and truncating the result to its MSW.

Pseudo Code Example:

```
if src >= 0x7FFF_8000 and src <= 0x7FFF_FFFF      /* RND16 */
 dest = 0x7FFF                                    // Using
 sat = 1                                          // unsigned
else                                              // pseudo
 dest = (src + 0x0000_8000) >> 16                 // arithmetic
 sat = 0
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 V_flag = sat
 S_flag = S_flag || sat
```

Assembly Code Example:

```
RND16 r1,r2            ; write the rounded result of r2 into r1
```

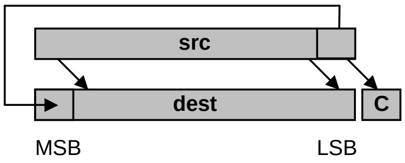
ROR

Rotate Right

Logical Operation

Operation:

dest ← ROR by 1 (src)



Format:

inst dest, 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	0010011000101111F111CCCCCC000011	
ROR<.f>	0,u6	0010011001101111F111uuuuuu000011	
ROR<.f>	0,limm	0010011000101111F111111110000011	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

ROR multiple

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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

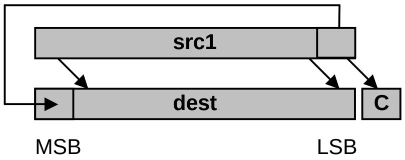
ROR multiple

Multiple Rotate Right

Logical Operation

Operation:

if (cc=true) then dest ← rotate right of src1 by src2



Format:

inst dest, src1, 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	00101110000000011F111CCCCCCAAAAAA	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	00101110110000011F111CCCCCC0QQQQQ	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
RLC	RRC
ASL multiple	ASR multiple
LSR multiple	

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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

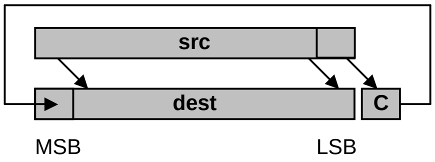
RRC

Rotate Right through Carry

Logical Operation

Operation:

dest ← RRC by 1 (src)



Format:

inst dest, 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. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then dest ← src2 – src1

Format:

inst dest, src1, src2

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	00100110000011110F111CCCCCCAAAAAA	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	00100110110011110F111CCCCCC0QQQQQ	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

SUBS

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’. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

RTIE

Return from Interrupt/Exception

Kernel Operation

Operation:

Return from interrupt or exception.

Format:

inst

Format Key:

inst = Instruction

Syntax:

	Instruction Code
RTIE	00100100011011110000000000111111

Flag Affected (32-Bit):

Z	●	= Set according to status register update
N	●	= Set according to status register update
C	●	= Set according to status register update
V	●	= Set according to status register update
E1	●	= Set according to status register update
E2	●	= Set according to status register update
U	●	= 1
AE	●	= 0

Key:

L = Limm Data

Related Instructions:

TRAP_S	SWI/TRAP0
J.F [ILINK1]	J.F [ILINK2]

Description:

The return from interrupt/exception instruction, RTIE, allows exit from interrupt and exception handlers, and to allow the processor to switch from kernel mode to user mode.

The RTIE instruction is available only in kernel mode. Attempted use when in user mode causes a Privilege Violation exception.

The RTIE instruction can be used by interrupt and exception handlers as a single instruction for exit. The RTIE instruction updates the program counter and status registers depending on whether a high or low interrupt, or an exception is being serviced according to the following:

- High level interrupt return registers – ILINK2, STATUS32_L2
- Low level interrupt return registers – ILINK1, STATUS32_L1
- Exception return registers – ERET, ERSTATUS

Bits in the STATUS32 register are provided to allow the RTIE instruction to determine from where to reload the pre-interrupt/exception machine state.

Since interrupts and exceptions are permitted between a branch/jump and an executed delay slot instruction, special branch target address registers are used for interrupt and exception handler returns.

If the STATUS32[DE] bit becomes set as a result of the RTIE instruction, the processor will be put back into a state where a branch with a delay slot is pending. The target of the branch will be contained in the BTA register. The value in BTA will have been restored from the appropriate Interrupt or Exception Return BTA register (ERBTA, BTA_L1 or BTA_L2).

When returning from an interrupt, the Branch Target Address register (BTA) is loaded from the appropriate high- or low-level Interrupt Return Branch Target Address register (BTA_L1 or BTA_L2).

When returning from an exception, the Branch Target Address register (BTA) is loaded from the Exception Return Branch Target Address (ERBTA) register.

NOTE Exit of an interrupt handler is also supported through the use of Jcc.F [ILINKn] and J_S.F [ILINKn]. Using these instructions will cause the appropriate Interrupt Return Link Register (BTA_L1 or BTA_L2) to be copied to BTA.

Pseudo Code Example

```
if STATUS[AE] == 1 then                                  /* RTIE */
  PC = ERET
  STATUS32 = ERSTATUS
  BTA   = ERBTA
else if STATUS[A2] == 1 then
  PC = ILINK2
  STATUS32 = STATUS32_L2
  BTA = BTA_L2
else if STATUS[A1] == 1 then
  PC = ILINK1
  STATUS32 = STATUS32_L1
  BTA = BTA_L1
else
  PC = ERET
  STATUS32 = ERSTATUS
  BTA = Verbatim STATUS[AE]
```

Assembly Code Example:

```
RTIE                         ; Return from interrupt/exception
```

SAT16

Saturation

Extended Arithmetic Operation

Operation:

dest ← $\text{sat}_{16}(\text{src})$

Format:

inst dest, src

Format Key:

dest = Destination Register
src = Source Operand 1

Syntax:

With Result		**Instruction Code**	
SAT16<.f>	b,c	00101bbb00101111FBBBCCCCCC000010	
SAT16<.f>	b,u6	00101bbb01101111FBBBuuuuuu000010	
SAT16<.f>	b,limm	00101bbb00101111FBBB111110000010	L
Without Result			
SAT16<.f>	0,c	0010111000101111F111CCCCCC000010	
SAT16<.f>	0,u6	0010111001101111F111uuuuuu000010	
SAT16<.f>	0,limm	0010111000101111F111111110000010	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V	●	= Set if result saturated, otherwise cleared
S	●	= Set if result saturated ('sticky' saturation)

Key:

L = Limm Data

Related Instructions:

ABSSW ABSS
RND16 NEGSW

Description:

Limit the 32-bit signed input operand to the range of a 16 bit signed word. The result of this operation has a signed value in the range 0xFFFF_8000 (negative value) up to 0x0000_7FFFF (positive value). Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if src >= 0xFFFF_8000 and src <= 0x0000_7FFF      /* SAT16 */
 dest = src                                       // Using
 sat = 0                                          // unsigned
if src < 0xFFFF_8000                              // pseudo
 dest = 0xffff_8000                               // arithmetic
 sat = 1
if src > 0x0000_7FFF
 dest = 0x0000_7FFF
 sat =1
if F==1 then
 Z_flag = if dest==0 then 1 else 0
 N_flag = dest[31]
 V_flag = sat
 S_flag = S_flag || sat
```

Assembly Code Example:

```
SAT16 r1,r2            ; Take the 16 bit saturated value of
                       ; r2 and write result into r1
```

SBC

Subtract with Carry

Arithmetic Operation

Operation:

if (cc=true) then dest ← (src1 – src2) - C

Format:

inst dest, src1, src2

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	RSUB
SUB1	SUB3
SUB2	SUBS

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'. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

dest ← SEXB(src)

Format:

inst dest, 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 EXTB

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). Any flag updates will only occur if the set flags suffix (.F) is used.

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

Operation:

dest ← SEXW(src)

Format:

inst dest, 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 EXTW

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). Any flag updates will only occur if the set flags suffix (.F) is used.

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

Kernel/Control Operation

Operation:

Enter Processor Sleep Mode

Format:

inst

Format Key:

inst	=	Instruction

Syntax:

		Instruction Code
SLEEP	<u6>	`00100001011011110000uuuuuu111111`
SLEEP	c	`00100001001011110000CCCCCC111111`

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged
ZZ	●	= 1

Key:

L = Limm Data

Related Instructions:

BRK

Description:

The sleep mode is entered when the ARCompact based 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 is a single operand instruction without flags. A SLEEP instruction without a source operand is encoded as SLEEP 0.

When a SLEEP instruction is detected at the decode stage of the pipeline, the instruction fetch stage is stalled and the pipeline is flushed ensuring that all instructions remaining in the pipeline are executed until the pipeline is empty. The the sleep mode flag, ZZ, found in the DEBUG register is then set.

The SLEEP instruction is serializing meaning the SLEEP instruction will complete and then flush 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.

For the ARC 600 processor the SLEEP instruction should not be placed in the last instruction position of a zero overhead loop.

For the ARCtangent-A5 and ARC 600 processor, 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 that interrupts remain disabled until FLAG has completed its update of the flag registers in stage 4 of the ARCompact based pipeline. Hence, if SLEEP follows into the pipeline immediately behind FLAG, then no interrupt can be taken between the FLAG and SLEEP.

NOTE For the ARCtangent-A5 and ARC 600 processor, 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.

CAUTION In some circumstances, for the ARC 600 processor with certain memory systems it may not be possible to guarantee that the FLAG/SLEEP instruction pair is atomic. For example if the memory system wait states are 2 or greater and the instruction cache is disabled or not capable of line locking (direct mapped) then even aligning the FLAG/SLEEP pair to a cache line will not necessarily ensure atomic operation.

It is possible for the instruction fetch to stall after the FLAG is passed to the pipeline in these circumstances which means an interrupt could occur between the FLAG instruction and the SLEEP instruction.

For the ARC 700 processor the bottom 2 bits of the source field, u6 or c, are used as the enable flags value, the remaining 4 bits are ignored. The SLEEP instruction will set interrupt enables according to the following values of the source operand:

Instruction	Operand	Effect on interrupt enables (E1/E2)
SLEEP	0x0	-
SLEEP	0x1	e1 = 1, e2 = 0
SLEEP	0x2	e1 = 0, e2 = 1
SLEEP	0x3	e1 = 1, e2 = 1

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. Every single-step operation is a restart and the ARCompact based processor wakes up at the next single-step. Consequently, the SLEEP instruction behaves exactly like a NOP propagating through the pipeline.

Pseudo Code Example:

```
FlushPipe()                                  /* SLEEP */
DEBUG[ZZ] = 1
WaitForInterrupt()
DEBUG[ZZ] = 0
ServiceInterrupt()
```

Assembly Code Example:

The SLEEP instruction can be put anywhere in the code, as in the following example.

For the ARC 600 processor the SLEEP instruction should not be placed in the last instruction position of a zero overhead loop.

For the ARCtangent-A5 and ARC 600 processor, the SLEEP instruction cannot immediately follow a BRcc or BBITn instruction.

Example 23 Sleep placement in code

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

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

Example 24 Sleep placement after Branch

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

NOTE In this example, the ARCompact based processor goes to sleep after the branch instruction has been executed. When the ARCompact based processor is sleeping, the PC points to the “`add`” instruction after the label "`after_sleep`". When an interrupt occurs, the ARCompact based 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:

Example 25 Sleep placement after Branch with killed delay slot

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

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

Example 26 Enable Interrupts and Sleep, ARCtangent-A5 and ARC 600

```
.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
```

For the ARC 700 processor the following code will ensure successful use of the SLEEP instruction for RTOS type applications.

Example 27 Enable Interrupts and Sleep, ARC 700

```
.equ e1, 0x1
.equ e2, 0x2
.equ e1e2, 0x3

SLEEP
SLEEP e1      ; e1 = 1, e2 = 0
SLEEP e2      ; e1 = 0, e2 = 1
SLEEP e1e2    ; e1 = 1, e2 = 1
```

SR

Store to Auxiliary Register

Control Operation

Operation:

aux_reg(src2) ← src1

Format:

inst src1, src2

Format Key:

src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

		Instruction Code	
SR	b,[c]	00100bbb001010110BBBCCCCCCRRRRRR	
SR	b,[limm]	00100bbb001010110BBB111110RRRRRR	L
SR	b,[u6]	00100bbb011010110BBBuuuuuu000000	
SR	b,[s12]	00100bbb101010110BBBssssssSSSSSS	
SR	limm,[c]	00100110001010110111CCCCCCRRRRRR	L
SR	limm,[u6]	00100110011010110111uuuuuu000000	□
SR	limm,[s12]	00100110101010110111ssssssSSSSSS	L

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

LR ST

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).

The status flags are not updated with this instruction therefore the flag setting field, F, should be encoded as 0.

The reserved field, R, is ignored by the processor, but should be set to 0.

The SR instruction cannot be conditional therefore encoding the operand mode (bits 23:22) to be 0x3 will raise an Instruction Error exception.

For the ARCtangent-A5 and ARC 600 processors, the behavior is undefined if an SR instruction is encoded using the operand mode of 0x3.

Pseudo Code Example:

```
Aux_reg(src2) = src1                           /* SR */
```

Assembly Code Example:

```
SR r1,[r2]             ; Store contents of r1 into
                       ; Aux. register pointed to by r2
```

ST

Store to Memory

Memory Operation

Operation:

Memory Store Address @ (src2+src3) ← src1

Format:

inst src1, src2, src3

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

Instruction	Format	Operation	Description
			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
PUSH_S
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 status flags are not updated with this instruction.

CAUTION The addition of src2 to src3 is performed with a simple 32-bit adder which is independent of the ALU. No exception occurs if a carry or overflow occurs. The resultant calculated address may overlap into unexpected regions depending of the values of src2 and src3.

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
```

SUB

Subtract

Arithmetic Operation

Operation:

if (cc=true) then dest ← src1 – src2

Format:

inst 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	0010011000000010F111CCCCCCAAAAAA	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	0010011011000010F111CCCCCC0QQQQQ	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'. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then dest ← src1 – (src2 << 1)

Format:

inst dest, src1, src2

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	SUB2	SBC
SUB	SUB3	

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'. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then dest ← src1 – (src2 << 2)

Format:

inst dest, src1, src2

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	`00100110000011000F111CCCCCCAAAAAA`	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	`00100110110110000F111CCCCCC0QQQQQ`	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	SBC
SUB1	SUB3	

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'. Any flag updates will only occur if the set flags suffix (.F) is used.

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

Arithmetic Operation

Operation:

if (cc=true) then dest ← src1 – (src2 << 3)

Format:

inst dest, src1, src2

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	SUB2	SBC
SUB1	SUB	

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'. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

SUBS

Signed Subtraction with Saturation

Extended Arithmetic Operation

Operation:

dest ← sat_{32} (src1 - src2)

Format:

inst dest, src1, src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
SUBS<.f>	a,b,c	00101bbb00000111FBBBCCCCCCAAAAAA	
SUBS<.f>	a,b,u6	00101bbb01000111FBBBuuuuuuAAAAAA	
SUBS<.f>	b,b,s12	00101bbb10000111FBBBssssssSSSSSS	
SUBS<.cc><.f>	b,b,c	00101bbb11000111FBBBCCCCCC0QQQQQ	
SUBS<.cc><.f>	b,b,u6	00101bbb11000111FBBBuuuuuu1QQQQQ	
SUBS<.f>	a,limm,c	0010111000000111F111CCCCCCAAAAAA	L
SUBS<.f>	a,b,limm	00101bbb00000111FBBB111110AAAAAA	L
SUBS<.cc><.f>	b,b,limm	00101bbb11000111FBBB111110QQQQQQ	L
Without Result			
SUBS<.f>	0,b,c	00101bbb00000111FBBBCCCCCC111110	
SUBS<.f>	0,b,u6	00101bbb01000111FBBBuuuuuu111110	
SUBS<.f>	0,b,limm	00101bbb00000111FBBB111110111110	L
SUBS<.cc><.f>	0,limm,c	0010111011000111F111CCCCCC0QQQQQ	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 by the subtract
V	●	= Set if result saturated, otherwise cleared
S	●	= Set if result saturated ('sticky' saturation)

Key:

L	= Limm Data

Related Instructions:

ADDS SUB

Description:

Perform a signed subtraction of the two source operands. If the result overflows, limit it to the maximum signed value. Both saturation flags S1 and S2 will be set if the result of the instruction saturates. Any flag updates will only occur if the set flags suffix (.F) is used.

Pseudo Code Example:

```
if cc==true then                                         /* SUBS */
 dest = src1 - src2
 sat = sat32(dest)
 if F==1 then
  Z_flag = if dest==0 then 1 else 0
  N_flag = dest[31]
  C_flag = 0
  V_flag = sat
  S_flag = S_flag || sat
```

Assembly Code Example:

```
SUBS r1,r2,r3          ; Subtract contents of r3 from r2
                       ; and write result into r1
```

SUBSDW

Signed Subtract with Saturation Dual Word

Extended Arithmetic Operation

Operation:

dest ← sat_{16}(src1.high-src2.high) : sat_{16}(src1.low-src2.low)

Format:

inst dest, src1, src2

Format Key:

dest	=	Destination Register
src1	=	Source Operand 1
src2	=	Source Operand 2

Syntax:

With Result		**Instruction Code**	
SUBSDW<.f>	a,b,c	`00101bbb00101001FBBBCCCCCCAAAAAA`	
SUBSDW<.f>	a,b,u6	`00101bbb01101001FBBBuuuuuuAAAAAA`	
SUBSDW<.f>	b,b,s12	`00101bbb10101001FBBBssssssSSSSSS`	
SUBSDW<.cc><.f>	b,b,c	`00101bbb11101001FBBBCCCCCC0QQQQQ`	
SUBSDW<.cc><.f>	b,b,u6	`00101bbb11101001FBBBuuuuuu1QQQQQ`	
SUBSDW<.f>	a,limm,c	`00101110001010001F111CCCCCCAAAAAA`	L
SUBSDW<.f>	a,b,limm	`00101bbb00101001FBBB111110AAAAAA`	L
SUBSDW<.cc><.f>	b,b,limm	`00101bbb11101001FBBB1111100QQQQQ`	L
Without Result	**- only flags will be set**		
SUBSDW<.f>	0,b,c	`00101bbb00101001FBBBCCCCCC111110`	
SUBSDW<.f>	0,b,u6	`00101bbb01101001FBBBuuuuuu111110`	
SUBSDW<.cc><.f>	0,limm,c	`00101110111010 01F111CCCCCC0QQQQQ`	L

Flag Affected (32-Bit):

Z	●	= Set if result is zero
N	●	= Set if most significant bit of result is set
C		= Unchanged
V	●	= Set if result saturated, otherwise cleared
S	●	= Set if result saturated ('sticky' saturation)

Key:

L = Limm Data

Related Instructions:

ADDSDW SUB
ADDS SUBS

Description:

Perform a signed dual-word subtraction of the two source operands. If the result overflows, limit it to the maximum signed value. The saturation flags S1 and S2 will be set according to the result of the channel 1 (high 16-bit) and channel 2 (low 16-bit) calculations respectively. Any flag updates will only occur if the set flags suffix (.F) is used.

Assembly Code Example:

```
SUBSDW r1,r2,r3        ;
```

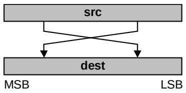
SWAP

Swap words

Extension Option

Operation:

dest ← word swap of src



Format:

inst dest, 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:

NORM MOV

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. Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

SWI/TRAP0

Software Interrupt or Software Breakpoint

Control Operation

Operation:

Trigger Instruction Error Level Interrupt

Format:

inst

Format Key:

inst = Instruction

Syntax:

	Instruction Code
SWI	`00100010011011110000000000111111`
TRAP0	`00100010011011110000000000111111`

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged
E1	●	= 0
E2	●	= 0
U	●	= 0
AE	●	= 1

Key:

L	= Limm Data

Related Instructions:

TRAP_S

Description:

The software interrupt instruction is decoded in stage two of the pipeline and if executed, then it immediately raises the Instruction Error interrupt. The Instruction Error interrupt will be serviced using the normal interrupt system. ILINK2 is used as the return address in the service routine.

Once an Instruction Error interrupt 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.

The TRAP0 instruction raises an exception and calls any operating system in kernel mode. Traps can be raised from user or kernel modes. A value of 0 is loaded into the exception cause register (ECR) as the cause parameter along with the cause code for a trap and the trap vector number.

The source value of 0 is used for software breakpoints. TRAP_S 0 provides a 15-bit encoding of the TRAP0 instruction.

The Exception Fault Address register (EFA) is set to point to the address of the trap instruction. The Exception Return Address register (ERET) is set to the address of the instruction immediately following the trap instruction.

When the exception handler has completed, program execution will resume at the instruction immediately following the trap instruction.

When inserting a software breakpoint, the instruction at the appropriate address is replaced by a trap instruction of the same size TRAP_S 0 for 16-bit instructions and TRAP0 for 32-bit instructions.

While the mnemonic SWI is available, its use is not recommended in the ARC 700 processor, TRAP0 should be used instead.

Pseudo Code Example:

```
ILINK2 = nPc                                         /* SWI */
STATUS32_L2 = STATUS32
STATUS32[E2] = 0
STATUS32[E1] = 0
PC = INT_VECTOR_BASE + 0x10
ERET = NEXTPC                                        /* TRAP0 */
ERSTATUS = STATUS32
if STATUS32[DE] == 1 then
  ERBTA = pending PC
ECR = 0x00 : 0x25 : 0x00 : 0x00
EFA = PC
STATUS32[U] = 0
STATUS32[E2] = 0
STATUS32[E1] = 0
STATUS32[AE] = 1
PC = INT_VECTOR_BASE + 0x128
```

Assembly Code Example:

```
SWI                            ; Software interrupt
TRAP0                          ; Software Breakpoint
```

SYNC

Synchronize

Control Operation

Operation:

Wait for all data memory transactions to complete

Format:

inst

Format Key:

inst = Instruction

Syntax:

	Instruction Code
SYNC	`00100011011011110000000000111111`

Flag Affected (32-Bit):

Flag		
Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

LD	ST

Description:

The synchronize instruction, SYNC, waits until all data based memory operations (LD, ST, EX, cache fills) have completed. The status flags are not updated with this instruction therefore the flag setting field, F, should be encoded as 0.

In order to provide the instruction sync function, the instruction serializes on completion, meaning that the contents of the pipeline are discarded, and fetching restarted from the stored program counter value.

For data synchronization, the purpose of the SYNC instruction is to ensure that all memory operations started by the processor have finished before any new operations (of any kind) can begin. This includes all of the following memory operations:

- All outstanding LD, ST and EX instructions
- All data cache operations
  - line fills and flushes
- All instruction cache fill operations

NOTE The SYNC instruction does not wait on memory operations started by other processors.

The SYNC instruction can also be used to ensure that the interrupt request of a memory mapped peripheral has been cleared down before an interrupt handler exits.

Example 28 Using SYNC to clear down an interrupt request

- A peripheral generates interrupt to the processor by setting a signal true.
- The control registers for the peripheral are memory mapped
- The processor's interrupt unit is set to ‘level sensitive’ for this interrupt.

- The interrupt handler must clear the interrupt request signal before exiting
    - The SYNC instruction is used to ensure that the store to change the peripheral status happens before the interrupt exit

If the SYNC was not used, the peripheral may still be asserting the interrupt-request signal after the interrupt exit – hence a bogus interrupt would be generated.

Pseudo Code Example:

```
do                                                  /* SYNC */
  null
until not (load_pending or store_pending or
            dcache_fill or dcache_flush or
            icache_fill)
```

Assembly Code Example:

```
SYNC                       | ; Synchronize
```

TRAP_S

Trap

Control Operation

Operation:
Raise an exception

Format:
inst src

Format Key:
inst = Instruction

Syntax:

	Instruction Code
TRAP_S u6	`01111uuuuuu11110`

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged
E1	●	= 0
E2	●	= 0
U	●	= 0
AE	●	= 1

Key:

L = Limm Data

Related Instructions:
SWI/TRAP0

Description:
The TRAP_S instruction raises an exception and calls any operating system in kernel mode. Traps can be raised from user or kernel modes. The source operand is loaded into the exception cause register (ECR) as the cause parameter along with the cause code for a trap and the trap vector number.

The source value can be used to signal a type of command to any operating system that is running on the processor. Source values 1 to 63 should be used of operating system calls and a source value of 0 for software breakpoints. TRAP_S 0 provides a 15-bit encoding of the TRAP0 instruction.

The Exception Fault Address register (EFA) is set to point to the address of the trap instruction. The Exception Return Address register (ERET) is set to the address of the instruction immediately following the trap instruction.

When the exception handler has completed, program execution will resume at the instruction immediately following the trap instruction.

When inserting a software breakpoint, the instruction at the appropriate address is replaced by a trap instruction of the same size TRAP_S 0 for 16-bit instructions and TRAP0 for 32-bit instructions.

Pseudo Code Example:

```
ERET = NEXTPC                                        /* TRAP_S */
ERSTATUS = STATUS32
if STATUS32[DE] == 1 then
  ERBTA = pending PC
ECR = 0x00 : 0x25 : 0x00 : src
EFA = PC
STATUS32[U] = 0
STATUS32[E2] = 0
STATUS32[E1] = 0
```

```
STATUS32[AE] = 1
PC = INT_VECTOR_BASE + 0x128
```

Assembly Code Example:

```
TRAP_S 0                    ; Trap
```

TST

Test

Logical Operation

Operation:

if (cc=true) then src1 AND src2

Format:

inst src1, 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. The flag setting field, F, is always encoded as 1 for this instruction.

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
```

UNIMP_S

Unimplemented Instruction

Control Operation

Operation:

InstError

Format:

inst

Format Key:

inst	=	Instruction
InstError	=	Raise Instruction Error Exception

Syntax:

	Instruction Code
UNIMP_S	`0111100111100000`

Flag Affected (32-Bit):

Z	□	= Unchanged
N	□	= Unchanged
C	□	= Unchanged
V	□	= Unchanged

Key:

L = Limm Data

Related Instructions:

NOP

Description:

An Instruction Error exception will be generated. Used by debugging tools to fill unused memory regions. The status flags are not updated with this instruction.

Pseudo Code Example:

```
InstError = 1;                    /* UNIMP_S */
```

Assembly Code Example:

```
UNIMP_S              ; Unimplemented Instruction
```

XOR

Logical Bitwise Exclusive OR

Logical Operation

Operation:

if (cc=true) then dest ← src1 XOR src2

Format:

inst dest, src1, 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	00100110000001111F111CCCCCCAAAAAA	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). Any flag updates will only occur if the set flags suffix (.F) is used.

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
```

This page is intentionally left blank.

Chapter 10 — The Host

The Host Interface

The ARCompact based 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 ARCompact based processor appear to the host is host interface dependent but an outline of the techniques to control ARCompact based 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 system may have extensions or customizations in this area, please see associated documentation.

It is expected that the registers and the program memory of ARCompact based processor will appear as a memory mapped section to the host. Figure 98 on page 337 shows an example host system using contiguous part of host memory. Figure 99 on page 337 shows an example host system using a section of memory and a section of I/O space.

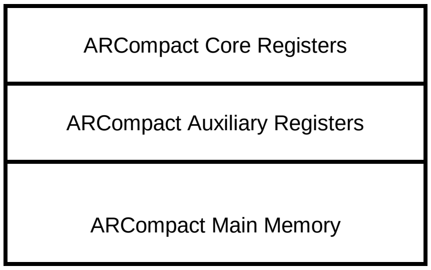


Figure 98 Example Host Memory Maps, Contiguous Host Memory

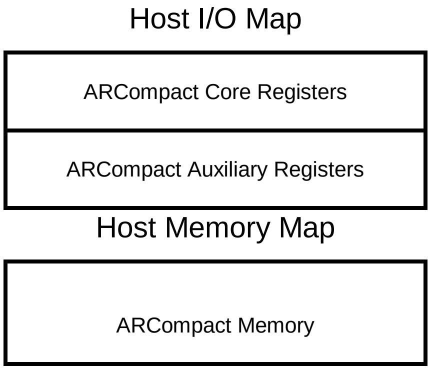


Figure 99 Example Host Memory Maps, Host Memory and Host IO

Once a Reset has occurred, the ARCompact based 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 ARCompact based processor , depending on whether the ARCompact based processor is running or halted as shown in Table 97 on page 338.

Table 97 Host Accesses to the ARCompact based processor

	Running	Halted
Memory	Read/Write	Read/Write
Auxiliary Registers	Mainly No access	Read/Write
Core Registers	No access	Read/Write

Halting

The ARCompact based processor can halt itself with the FLAG instruction or it can be halted by the host. The host halts the ARCompact based processor by setting the H bit in the status register (STATUS32), or by setting the FH bit in the DEBUG register. See Figure 43 on page 50 and Figure 45 on page 51.

Note that when the ARCompact based processor is running that only the H bit will change if the host writes to STATUS32 register. However, if ARCompact based processor had halted itself, the *whole* of the STATUS32 register will be updated when the host writes to the STATUS32 register.

The consequence of this is that the host may assume that the ARCompact based processor is running by previously reading the STATUS32 register. By the time that the host forces a halt, the ARCompact based 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 ARCompact based processor to halt without overwriting the other status flags the additional FH bit in the DEBUG register is provided. See Figure 43 on page 50. The host can test whether the ARCompact based processor has halted by checking the state of the H bit in the STATUS32 register. Additionally, the SH bit in the debug register is available to test whether the halt was caused by the host, the ARCompact based 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 ARCompact based processor.

Starting

The host starts the ARCompact based 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 ARCompact based processor, by sending NOP instructions through, so that any pending instructions that are about to modify any registers in the ARCompact based processor are allowed to complete.

If the ARCompact based 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 ARCompact based 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 ARCompact based 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:

- Stop the ARCompact based processor
- Download a NOP instruction into memory.
- Invalidate instruction cache to ensure that the NOP is loaded from memory
- Point the PC register to the downloaded NOP
- Single step until the values in the program counter or loop count register change.
- Point the PC register to the downloaded NOP
- Single step until the values in the program counter or loop count register change.
- Point the PC register to the downloaded NOP
- Single step until the values in the program counter or loop count register change.

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 Instruction Stepping

The Single Instruction Step function is controlled by a bit in the DEBUG register. This bit can be set by the debugger to enable Instruction Stepping. The Instruction Step (IS), is write-only by the host and keeps it value for one cycle (see Table 98 on page 340).

Table 98 Single Step Flags in Debug Register

Field	Description	Access Type
IS	Instruction Step:- Instruction Step enable	Write only from the host

The Single Instruction Step function enables the processor for completion of a whole instruction.

For the ARC 600 core the Single Instruction Step function is enabled by setting both the SS and IS bits in the debug register when the processor is halted.

For the ARC 700 core the Single Instruction Step function is enabled by setting the IS bit in the debug register when the processor is halted. The SS bit is ignored.

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 killed 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 Instruction Step Mode

The SLEEP instruction is treated like 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 like a NOP propagating through the pipeline.

See SLEEP on page 309 for further details.

BRK Instruction in Single Instruction Step Mode

The BRK instruction behaves exactly as when the processor is not in the Single Step Mode. The BRK instruction is detected in the initial stages of the pipeline and kept there forever until removed by the host.

Software Breakpoints

The BRK instruction can also be used to insert a software breakpoint. BRK will halt the ARCompact based 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 ARCompact based processor code memory, it can also replace a ARCompact based 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.

Core Registers

The core registers of ARCompact based processor are available to be read and written by the host. These registers should be accessed by the host once the ARCompact based processor has halted.

Auxiliary Register Set

Some auxiliary registers, unlike the core registers, may be accessed while the ARCompact based processor is running. These dual access registers in the base case are:

STATUS32

The host can read the status register (STATUS32) when the ARCompact based processor is running. The main purpose is to see if the processor has halted. See Figure 45 on page 51.

PC

Reading the PC is useful for code profiling. See Figure 44 on page 51.

SEMAPHORE

The semaphore register (SEMAPHORE) is used for inter-processor and host to ARCompact based processor communications. Protocols for using shared memory and provision of mutual exclusion can be accomplished with this register. See Figure 39 on page 48.

IDENTITY

The host can determine the version of ARCompact based processor by reading the identity register (IDENTITY). See Figure 41 on page 49. Information on extensions added to the ARCompact based 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 ARCompact based processor, the host needs to set the FH bit of the debug register (DEBUG). The host can determine how the ARCompact based was halted and if there are any pending loads. See Figure 43 on page 50.

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 ARCompact based memory space, then the instruction cache should be invalidated.