



ARCtangent™-A4

Programmer's Reference

ARCTangent™-A4 Programmer's Reference

ARC™ International

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

Confidential and proprietary information

© 2000-2002 ARC International (unpublished). All rights reserved.

Notice

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

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

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

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

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

5050-001 August-2002

Contents

Chapter 1 — Preface	1
Key Features	1
Chapter 2 — Architectural Description	5
Introduction	5
Programmer's Model	5
Core register set	6
Auxiliary register set	7
The Host	7
Extensions	9
Extension core registers	9
Extension auxiliary registers	9
Extension instruction set	10
Extension condition codes	10
System Customization	11
Memory controller	11
Load store unit	11
Interrupt unit	11
Debugging Features	12
Power Management	12
Chapter 3 — Data Organization and Addressing	15
Introduction	15
Operand Size	15
Data Organization	16
Registers	16
Immediate data	16
Memory	16
Addressing Modes	16
Memory Addressing	19
Instruction Format	19
Register	20

Short immediate	20
Long immediate	20
Branch	20
Register Notation	20
Chapter 4 — Interrupts	23
Introduction	23
ILINK Registers	23
Interrupt Vectors	23
Interrupt Enables	24
Returning from Interrupts	25
Reset	26
Memory Error	26
Instruction Error	26
Interrupt Times	27
Alternate Interrupt Unit	27
Chapter 5 — Instruction Set Summary	29
Introduction	29
Arithmetic and Logical Operations	29
Null Instruction	30
Single Operand Instructions	30
Jump, Branch and Loop Operations	33
Zero Overhead Loop Mechanism	35
LP_COUNT must not be loaded directly from memory	37
Single instruction loops	37
Loop count register	38
Branch and jumps in loops	39
Instructions with long immediate data: correct coding	40
Instructions with long immediate data: incorrect coding	40
Valid instruction regions in loops	41
Breakpoint Instruction	43
BRK instruction in delay slot	44
Sleep Instruction	44
SLEEP instruction in delay slot	45
SLEEP instruction in delay slot of Jump	46
SLEEP instruction in single step mode	46
Software Interrupt Instruction	46
SWI instruction format	46
Load and Store Operations	46
Auxiliary Register Operations	48

Extension Instructions	49
Optional Extensions Library	49
Multiply 32 X 32	50
Barrel shift/rotate block	51
Normalize instruction	51
SWAP instruction	52
MIN/MAX instructions	53
Chapter 6 — Condition Codes	55
Introduction	55
Condition Code Register	55
Condition Code Register Notation	55
Condition Code Test	56
Chapter 7 — Register Set Details	59
Core Register Set	60
Link registers	61
Loop count register	61
Immediate data indicators	61
Extension core registers	62
Multiply result registers	62
Auxiliary Register Set	62
Status register	63
Semaphore register	63
Loop control registers	65
Identity register	65
Debug register	65
Extension auxiliary registers	67
Optional extensions auxiliary registers	67
Multiply restore register	67
Chapter 8 — Instruction Set Details	69
Introduction	69
Instruction Map	69
Addressing Modes	71
Dual operand instructions	71
Single operand instructions	72
Branch type Instructions	72
Jump Instruction	72
Load Instruction	73
Store instruction	73
Load from auxiliary register instruction	74

Store to auxiliary register instruction	74
Instruction Encoding	75
Register	76
Short immediate	77
Single operand	77
Branch	77
Instruction Set Details	79
ADC	80
ADD	81
AND	82
ASL/LSL	83
ASL multiple	84
ASR	85
ASR multiple	86
BIC	87
Bcc	88
BLcc	89
BRK	91
EXT	92
FLAG	93
Jcc	95
JLcc	97
LD	99
LPcc	101
LR	102
LSL	103
LSR	104
LSR multiple	105
MAX	106
MIN	107
MOV	108
MUL64	109
MULU64	111
NOP	113
NORM	114
OR	116
RLC	117
ROL	118
ROR	119
ROR multiple	120
RRC	121
SBC	122
SEX	123

SLEEP	124
SR	125
ST	126
SUB	128
SWAP	129
SWI	130
XOR	131
Chapter 9 — The Host	133
Halting	135
Starting	135
Pipecleaning	136
Single Stepping	137
Single cycle step	137
Single instruction step	138
SLEEP instruction in single step mode	138
BRK instruction in single step mode	138
Software Breakpoints	139
ARCTangent-A4 Core Registers	139
ARCTangent-A4 Auxiliary Registers	139
STATUS	139
SEMAPHORE	139
IDENTITY	140
DEBUG	140
ARCTangent-A4 Memory	140
Chapter 10 — Pipeline and Timings	141
Introduction	141
Stage 1. Instruction fetch	141
Stage 2. Operand fetch	141
Stage 3. ALU	142
Stage 4. Write back	142
Pipeline-Cycle Diagram	142
Arithmetic and Logic Function Timings	143
Immediate Data Timing	144
Short immediate	144
Long immediate	144
Destination immediate	145
Conditional Instruction Timing	146
Extension Instruction Timings	147
Single cycle extension instructions	147
Multi cycle extension instructions	147

Multiply timings	148
Barrel shift timings	150
Jump and Branch Timings	151
Jump instruction	151
Jump and nullify delay slot instruction	152
Jump and execute delay slot instruction	152
Jump with immediate address	153
Jump setting flags	154
Conditional jump	154
Jump and link	156
Branch	158
Conditional branch	159
Software breakpoints	160
Software breakpoint return address calculation	160
Branch and link	161
Loop Timings	162
Loop set up	162
Conditional loop	164
Loop execution	165
Single instruction loops	166
Reading loop count register	168
Writing loop count register	169
Branch and jumps in loops	171
Software breakpoints in loops	172
Instructions with long immediate data	172
Flag Instruction Timings	173
Breakpoint	173
Sleep Mode	174
Load and Store Timings	176
Load	176
Store	178
Auxiliary Register Access	179
Load from register (LR)	179
Store to register (SR)	180
Interrupt Timings	181
Interrupt on arithmetic instruction	182
Software interrupt	183
Interrupt on jump, branch or loop set up	184
Interrupt on loop execution	185
Interrupt on load	186
Interrupt on store	186
Interrupt on auxiliary register access	186

Single Instruction Step	186
Single instruction step on single word instructions	186
Single instruction step on instruction with long immediate data	187
Index	189

List of Figures

Figure 1 Data flows in the AR Ctangent-A4 architecture	6
Figure 2 Example Host Memory Maps	8
Figure 3 Power Management Block Diagram	13
Figure 4 Status Register	25
Figure 5 PC Update and Loop Detection Mechanism for Loops	36
Figure 6 Valid Instruction Regions in Loops	42
Figure 7 32x32 Multiply	50
Figure 8 Barrel Shift Operations	51
Figure 9 Norm Instruction	52
Figure 10 SWAP Instruction	52
Figure 11 Status Register	55
Figure 12 Core Register Map	59
Figure 13 Auxiliary Register Set	60
Figure 14 Status Register	63
Figure 15 Semaphore Register	63
Figure 16 Loop Start Register	65
Figure 17 Loop End Register	65
Figure 18 Identity Register	65
Figure 19 Debug Register	65
Figure 20 Multiply Restore Register	67
Figure 21 Example Host Memory Maps	134
Figure 22 AR Ctangent-A4 Pipeline	141
Figure 23 Pipeline-Cycle Diagram	142

List of Tables

Table 1 Core Register Map	6
Table 2 Auxiliary Register Map	7
Table 3 Data Addressing Modes	18
Table 4 Key for Addressing Modes and Conventions	18
Table 5 Interrupt Summary	24
Table 6 Arithmetic and Logical Instructions	30
Table 7 Single Operand Instructions: Move and Extend	31
Table 8 Single Operand Instructions: Rotates and Shifts	32
Table 9 Single Operand Instructions: Flags and Halts	32
Table 10 Jump, Branch and Loop Instructions	33
Table 11 Load and Store Instructions	47
Table 12 Auxiliary Register Operations	48
Table 13 Condition Codes	57
Table 14 Core Register Map	61
Table 15 Multiply Result Registers	62
Table 16 Auxiliary Register Set	63
Table 17 Basecase Instruction Map	71
Table 18 Host Accesses to the ARCtangent-A4 processor	134
Table 19 Single Step Flags in Debug Register	137

Chapter 1 — Preface

This manual is aimed at programmers of the ARCTangent™-A4 processor and serves as a reference to the instruction set of the basecase ARCTangent-A4 (version 8) core.

Programmer's reference for additional extensions or customizations that have been implemented in the target ARCTangent-A4 system are contained in other manuals.

Key Features

Data Paths

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

Registers

- 32 General Purpose Core Registers
- Auxiliary Register Set

Load/Store Unit

- Delayed Load mechanism with Register Scoreboard
- Buffered Store
- Address Register Write-Back

Program Flow

- 4 Stage Pipeline
- Single Cycle Instructions

- All ALU Instructions are Conditional
- Single Cycle Immediate Data
- Jumps and Branches with Single Instruction Delay Slot
- Delay Slot Execution Modes
- Zero Overhead Loops

Interrupts and Exceptions

- Levels of Exception
- Non-Maskable Exceptions
- Maskable External Interrupts in basecase ARCtangent-A4 processor

Extensions

- 16 Extension Dual Operand Instruction Codes
- 55 Extension Single Operand Instruction Codes
- 28 Extension Core Registers
- 32 Bit addressable Auxiliary Register Set
- 16 Extension Condition Codes
- Build Configuration Registers

System Customizations

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

Host Interface Debug Features

- Start, stop and single step the ARCtangent-A4 processor via special registers
- Check and change the values in the register set and ARCtangent-A4 memory
- Communicate via the semaphore register and shared memory
- Perform code profiling by reading the status register

- Breakpoint Instruction

Power Management

- Sleep Mode
- Clock Gating Option

Chapter 2 — Architectural Description

Introduction

The ARCTangent-A4 is a 4-stage pipeline processor incorporating full 32-bit instruction, data and addressing. In line with RISC (reduced instruction set computer) based architectures, ARCTangent-A4 has an orthogonal instruction set with all addressing modes implemented on all arithmetic and logical instructions.

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

This document describes the minimum basecase version of ARCTangent-A4 with which all future designs incorporating ARCTangent-A4 must adhere to.

NOTE The implemented ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Programmer's Model

The programmer's model is common to all implementations of ARCTangent-A4 processor to allow upward compatibility of code.

Logically, ARCTangent-A4 processor is based around a 3 (or 4)-port core register file with many of the instructions being dual operand and 1 destination register. Other registers are contained in the auxiliary register set and are accessed with the LOAD-REGISTER/STORE-REGISTER commands or other special commands.

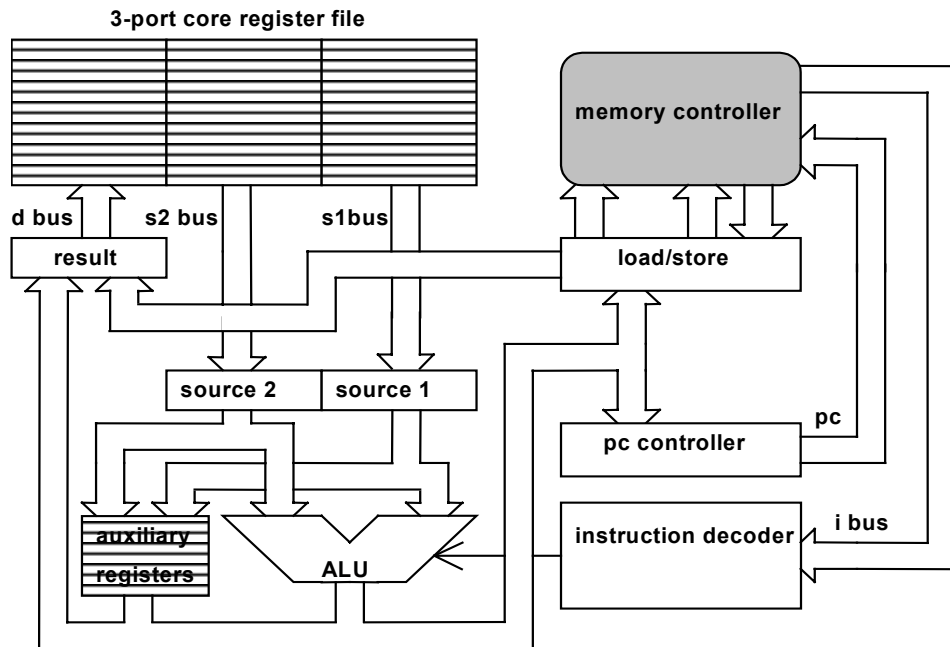


Figure 1 Data flows in the ARctangent-A4 architecture

Core register set

Number	Core register name	Function
0-28	r0-r28	General Purpose Registers
29	ILINK1 or r29	Maskable interrupt link register
30	ILINK2 or r30	Maskable interrupt link register
31	BLINK or r31	Branch link register
32-59	r32-r59	Register space reserved for extensions
60	LP_COUNT	Loop count register (24 Bits)
61	-	Short immediate data indicator setting flags
62	-	Long immediate data indicator
63	-	Short immediate data indicator not setting flags

Table 1 Core Register Map

The core register set in ARctangent-A4 processor is shown in Table 1. Other predefined registers are in the auxiliary register set and they are shown in Table

2. The general purpose registers (r0-r28) can be used for any purpose by the programmer.

Auxiliary register set

Number	Auxiliary register name	Description
0x0	STATUS	Status register
0x1	SEMAPHORE	Inter-process/Host semaphore register
0x2	LP_START	Loop start address (24 bits)
0x3	LP_END	Loop end address (24 bits)
0x4	IDENTITY	ARCTangent-A4 Identification register
0x5	DEBUG	Debug register
0x60 - 0x7F	RESERVED	Build Configuration Registers

Table 2 Auxiliary Register Map

The auxiliary register set contains special status and control registers. Auxiliary registers occupy a special address space that is accessed using special load and store instructions, or other special commands. The basecase ARCTangent-A4 processor uses 6 status and control registers, and reserves the additional registers 0x60 to 0x7F, leaving the rest of the 2³² registers for extension purposes.

The Host

The ARCTangent-A4 processor was developed with an integrated host interface to support communications with a host system. The ARCTangent-A4 processor can be started, stopped and communicated by the host system using special registers. Further information is contained in later sections of this manual.

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

NOTE The implemented ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

It is expected that the registers and the program memory of ARCTangent-A4 processor will appear as a memory mapped section to the host. For example, Figure 2 shows two examples: a) a contiguous part of host memory and b) a section of memory and a section of I/O space.

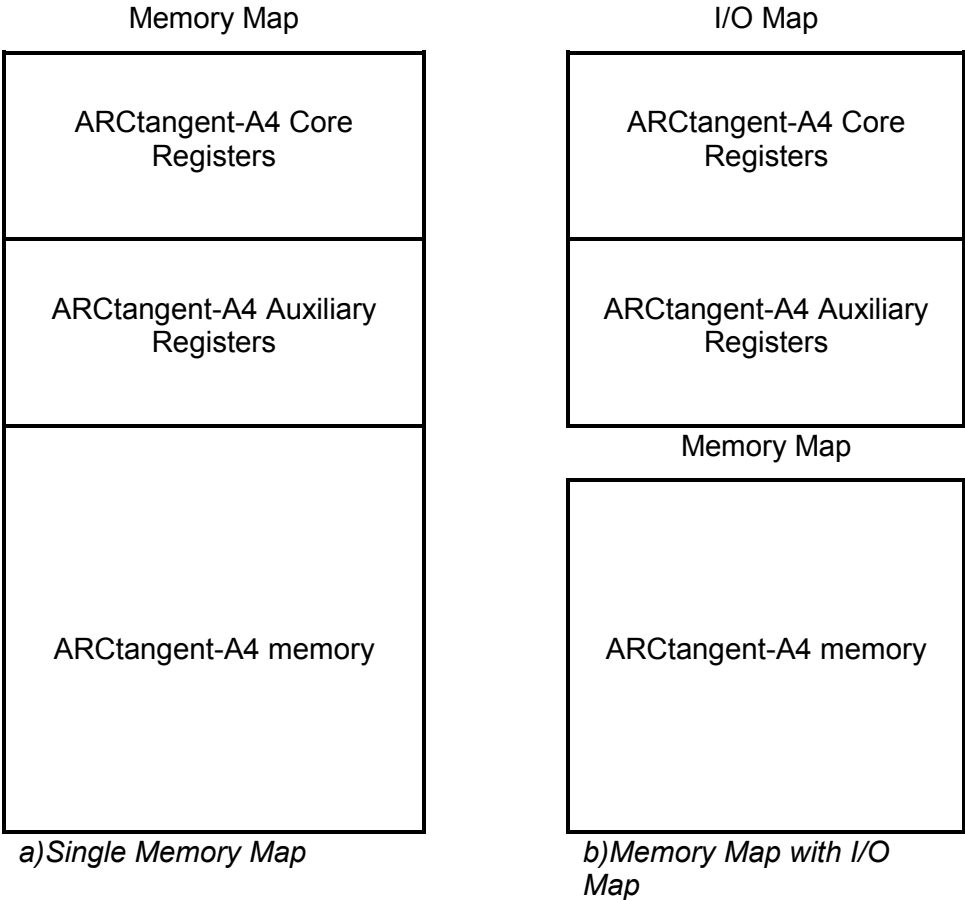


Figure 2 Example Host Memory Maps

Extensions

The ARCTangent-A4 processor is designed to be extendible according to the requirements of the system in which it is used. These extensions include more core and auxiliary registers, new instructions, and additional condition code tests. This section is intended to inform the programmer of the ARCTangent-A4 processor where these extensions occur and how they affect the programmer's view of the ARCTangent-A4 processor.

NOTE The implemented ARCTangent-A4 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 positions. The first 29 are general purpose basecase registers, the next 3 are the link registers and the last 4 are the loop count register and immediate data indicators. This leaves positions 32 to 59 for extension purposes. The extension registers are referred to as r32, r33,..etc.....,r59. The core register map is shown in [Table 1](#).

NOTE The implemented ARCTangent-A4 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 memory address space excepting those positions defined in the basecase for auxiliary registers. They are referred to using the load from register (LR) and store to register (SR) instructions or special extension instructions. The reserved auxiliary register addresses are shown in [Table 2](#).

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

The auxiliary register address region 0x7F up to 0x80, is reserved for the Build Configuration Registers (BCR) that can be used by embedded software or host debug software to detect the configuration of the ARCTangent-A4 hardware. The Build Configuration Registers contain the version of each ARCTangent-A4 extension, as well as configuration information that is build specific. The

registers are available for ARCTangent-A4 basecase version 8 processor onwards and will always remain backwardly compatible.

NOTE The Build Configuration Registers are fully described associated documentation.

The implemented ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Extension instruction set

Instructions are encoded onto the instruction word using a 5 bit binary number. This gives 32 separate instructions. The first 16 instructions are defined in the basecase ARCTangent-A4 processor. The remaining 16 instructions are available for extension. The basecase and extension instruction codes are given in Table 17.

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

Extension condition codes

The condition code test on an instruction is encoded using a 5 bit binary number. This gives 32 different possible conditions that can be tested. The first 16 codes (00-0F) are those condition codes defined in the basecase version of ARCTangent-A4 processor which use only the internal condition flags from the status register (Z, N, C, V), see Table 13 Condition Codes.

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

NOTE The implemented ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

System Customization

As well as the extensions mentioned in the previous section, ARCTangent-A4 processor can be additionally customized to match memory, cache, and interrupt requirements. This is achieved by using a separate memory controller, load/store unit and interrupt unit.

Memory controller

This unit is defined according to the memory system with which the ARCTangent-A4 processor is being used. Instruction-cache, data-cache, DRAM control, instruction versus data arbitration and other memory specific logic will be defined in the memory controller.

NOTE The implemented ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Load store unit

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

NOTE The implemented ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Interrupt unit

The interrupt unit contains the exception and interrupt vector positions, the logic to tell the ARCTangent-A4 which of the 3 levels of interrupt has occurred, and the arbitration between the interrupts and exceptions. The interrupt unit can be modified to alter the priority of interrupts, the vector positions and the number of interrupts.

NOTE The implemented ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Debugging Features

It is possible for the ARCTangent-A4 to be controlled from a host processor using special debugging features. The host can:

- start and stop the ARCTangent-A4 processor via the status and debug register
- single step the ARCTangent-A4 processor via the debug register
- check and change the values in the register set and ARCTangent-A4 memory
- communicate with the ARCTangent-A4 processor via the semaphore register and shared memory
- perform code profiling by reading the status register
- enable software breakpoints by using Bcc instruction
- enable software breakpoints by using BRK instruction

With these abilities it is possible for the host to provide software breakpoints, single stepping and program tracing of the ARCTangent-A4 processor.

It is possible for the ARCTangent-A4 processor to halt itself with the FLAG instruction. The self halt bit (SH) in the debug register is set if the ARCTangent-A4 processor halts itself.

Power Management

ARCTangent-A4 basecase version 8 processor and above have special power management features. The SLEEP instruction halts the ARCTangent-A4 processor and halts the pipeline until an interrupt or a restart occurs. Sleep mode stalls the core pipeline and disables any on-chip RAM.

Optional clock gating is provided which will switch off all non-essential clocks when the ARCTangent-A4 processor is halted or the ARCTangent-A4 processor is in sleep mode. This means the internal ARCTangent-A4 control unit is not active and major blocks are disabled. The host interface, interrupt unit and memory interfaces are always left enabled to allow host accesses and "wake" feature. The following diagram shows a summary of the clock gating and sleep circuitry.

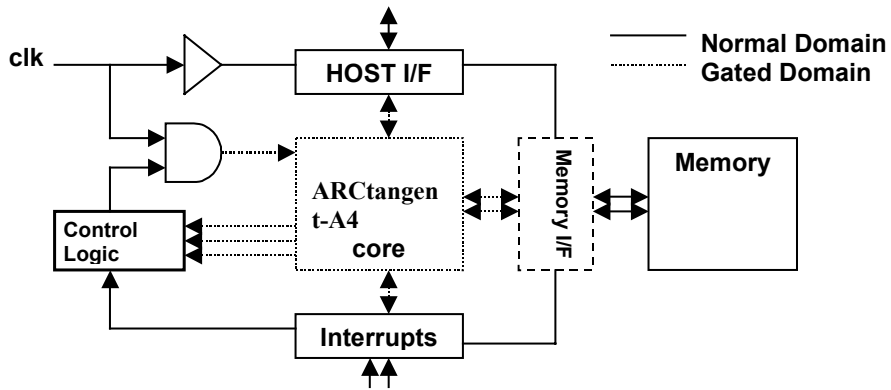


Figure 3 Power Management Block Diagram

Chapter 3 — Data Organization and Addressing

Introduction

This chapter describes the data organization and addressing of the ARCTangent-A4 processor.

Operand Size

The ARCTangent-A4 is a 32-bit word architecture and as such most operations are with 32-bit data. However, there are a few exceptions.

The basic data types are:

- Long word (32-bit) for register-register operation, immediate data and load/store
- Word (16-bit) for load/store operations only
- Short Immediate (9-bit) for short immediate data only
- Byte (8 bit) for load/store operations only

and addressing:

- absolute (32-bit) for load/store and jumps
- relative (20-bit) for branch and loop

Data Organization

Registers

The core registers and auxiliary registers are 32-bit (long word) wide.

Immediate data

The immediate data as an operand can be 32-bit (long immediate), or 9-bit sign extended to 32-bit (short immediate).

Memory

The memory operations (load and store) can have data of 32 bit (long word), 16 bit (word) or 8 bit(byte) wide. Byte operations use the low order 8 bits and may extend the sign of the byte across the rest of the long word depending on the load/store instruction. The same applies to the word operations with the word occupying the low order 16 bits. Data memory is accessed using byte addresses, which means long word or word accesses can be supplied with non-aligned addresses. The following should be supported as a minimum:

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

There is no "unaligned access exception" available in the ARCtangent-A4 processor. The basecase ARCtangent-A4 processor is "Endian free", in that the endianness of the implemented ARCtangent-A4 system is dependant entirely on the memory system.

Addressing Modes

The addressing modes that the instructions use are encoded within the register fields of the instruction word. There are basically only 3 addressing modes: register-register, register-immediate and immediate-immediate. However, as a consequence of the action performed by the different instruction groups, these can be expanded as shown in Table 3 Data Addressing Modes.

Mode	Syntax	Operation
Register, register	op a,b,c	$a \leftarrow b \text{ op } c$
Register, immediate	op a,b,imm	$a \leftarrow b \text{ op } \text{imm}$
immediate, register	op a,imm,c	$a \leftarrow \text{imm op } c$
immediate, immediate	op a,imm,imm	$a \leftarrow \text{imm op } \text{imm}$
test	op 0,b,c	no result but b op c can set flags
test with immediate	op 0,b,imm	b op imm can set flags
	op 0,imm,c	imm op c can set flags
single operand (register)	single_op a,b	$a \leftarrow \text{single_op } b$
single operand immediate	single_op a,imm	$a \leftarrow \text{single_op } \text{imm}$
single operand test (register)	single_op 0,b	Single_op b can set flags
single operand test immediate	single_op 0,imm	Single_op imm can set flags
flag with register	flag b	Flags $\leftarrow b$
flag with immediate	flag imm	Flags $\leftarrow \text{imm}$
load	ld a,[b,c]	$a \leftarrow \text{data at address } [b+c]$
load with immediate offset	ld a,[b,imm]	$a \leftarrow \text{data at address } [b + \text{imm}]$
	ld a,[imm,c]	$a \leftarrow \text{data at address } [\text{imm} + c]$
load from immediate address	ld a,[imm]	$a \leftarrow \text{data at address } [\text{imm}]$
load from auxiliary register	lr a,[b]	$a \leftarrow \text{data in reg. at address } [b]$
	lr a,[imm]	$a \leftarrow \text{data in reg. at address } [\text{imm}]$
store	st c,[b]	Data at address [b] $\leftarrow c$
store with immediate offset	st c,[b,shimm]	Data at address [b + shimm] $\leftarrow c$
store to immediate address	st c,[imm]	Data at address [imm] $\leftarrow c$
store 0	st 0,[b]	Data at address [b] $\leftarrow 0$

Mode	Syntax	Operation
store shimm with immediate offset	st shimm,[b,shimm]	Data at address [b + shimm] ← shimm (shimms MUST match)
store limm with immediate offset	st limm,[b,shimm]	Data at address [b + shimm] ← limm
store to auxiliary register	sr c,[b] sr c,[imm]	Data in reg. at address [b] ← c Data in reg. at address [imm] ← c

Table 3 Data Addressing Modes

Key for addressing modes and conventions

←	replaced by		
a	result register	b	operand register 1
c	operand register 2	C	carry flag in status register
op	instruction	single_op	single operand instr
ld	load instruction	st	store instruction
lr	load from auxiliary register instruction	sr	store to auxiliary register instruction
flag	flag instruction	imm	immediate data (long or short)
limm	long immediate (32 bit constant)	shimm	short immediate (signed 9 bit constant)
addr	absolute address	rel_addr	relative address
<cc>	optional condition code	<f>	optional set flags field
<zz>	optional size field	<di>	optional data cache bypass field
<a>	optional address write-back field	<x>	optional sign extend field
<dd>	optional delay slot execution mode	.	separator if other fields are used

Table 4 Key for Addressing Modes and Conventions

Memory Addressing

Branch and jump instructions that refer to memory (i.e. J, JL, B, BL, LP) contain an address. This address is referred to in the form $[n:2]$, where n is the most significant bit of the word. It is used as a long-word offset or address, but the numbering has retained the convention for byte addressing.

As an example to refer to the address 4 long words forward in a branch instruction would be 16 bytes ahead but only bits 21:2 are encoded. However, the syntax in assembly language would still be in bytes. Therefore, to branch 4 long words forward, the syntax would be `bra 16`, although it is unlikely that a programmer would specify a branch's relative address in such a way.

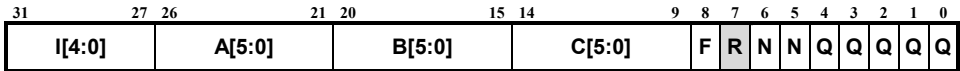
With the load and store commands (LD and ST), the address calculated by the instruction is passed as a 32-bit word to the memory controller, and used as a byte address.

An interrupt may be caused by the memory controller if the size of the operation and the address are incompatible, e.g. if the memory controller cannot fetch long-words from byte boundaries. This will be dependent on the memory controller being used with the ARCtangent-A4 processor and is not part of the basecase ARCtangent-A4 processor.

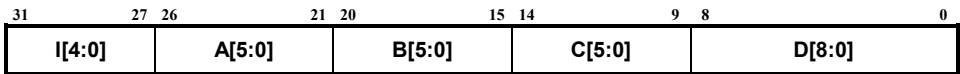
Instruction Format

Instructions are one long word in length and may have a long word immediate value following. There are three basic instruction layouts. The instruction is encoded on the I field. The result of the instruction is sent to the register defined by the A field. The two register source addresses are encoded on the B and C fields. If the result of the instruction needs to set the flags then the F bit is set. The condition that causes the instruction to be executed is encoded on the condition code field Q. The reserved bits R are undefined and should be set to 0. The L field in the branch type instruction specifies the signed relative jump address and the N field is used in jumps and branches to nullify or execute the next instruction. See also Chapter 5 — Instruction Set Summary and Chapter 8 — Instruction Set Details for further details.

Register

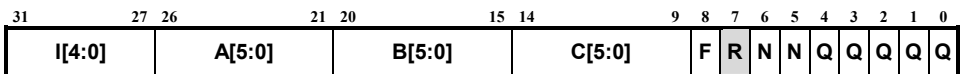


Short immediate

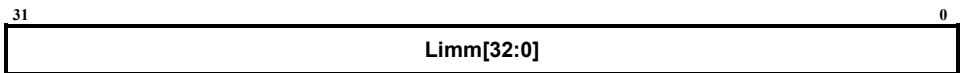


Long immediate

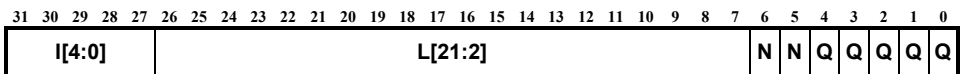
First long-word, the instruction



Second long-word, the data



Branch



Register Notation

The core registers are identified as follows:

<code>rn</code>	general purpose register number <i>n</i>
<code>ILINK1</code>	maskable interrupt link register 1
<code>ILINK2</code>	maskable interrupt link register 2
<code>BLINK</code>	branch link register
<code>LP_COUNT</code>	loop count register (24 bits)

Example syntax:

```

AND    r1,r2,r3           ;r1 ← r2 AND r3
AND    ILINK2,r21,r21      ;ILINK2 ← r21
    
```

The auxiliary registers are identified as:

STATUS	status register
SEMAPHORE	semaphore register
LP_START	loop start address (24 bits)
LP_END	loop end address (24 bits)
IDENTITY	ARCTangent-A4 identification register
DEBUG	ARCTangent-A4 debug register

Example syntax:

```
SR    r5, [SEMAPHORE]    ; [SEMAPHORE] ← r5
LR    r4, [LP_START]     ; r4 ← [LP_START]
```


Chapter 4 — Interrupts

Introduction

The ARCTangent-A4 interrupt mechanism is such that 3 levels of interrupts are provided.

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

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

NOTE The implemented ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

ILINK Registers

When an interrupt occurs, the link register, where appropriate, is loaded with the status register containing the next PC and the current flags; the PC is then loaded with the relevant address for servicing the interrupt.

Link register ILINK2 is associated with the level 2 set of interrupts and the two exceptions: memory error and instruction error. ILINK1 is associated with the level 1 set of interrupts.

Interrupt Vectors

In the basecase ARCTangent-A4 processor, there are three exceptions and each exception has it's own vector position, an alternate interrupt unit may be implemented, see section Alternate Interrupt Unit.

The ARCTangent-A4 processor does not implement interrupt vectors as such, but rather a table of jumps. When an interrupt occurs the ARCTangent-A4 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 ARCTangent-A4 system and is often a set of contiguous jump vectors.

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

Vector	Name	Link register	Byte Offset
0	reset	-	0x00
1	memory exception	ILINK2	0x08
2	instruction error	ILINK2	0x10
3 – n	irq3-irqn	ILINKm	0x18 – 0x08*n

Table 5 Interrupt Summary

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

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

NOTE The implemented ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Interrupt Enables

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

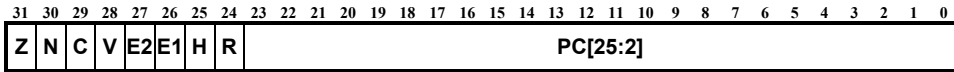


Figure 4 Status Register

Example:

```
.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
```

Returning from Interrupts

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

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

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

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

```
ivect4:      JAL    iservice4      ;vector 4
ivect5:      JAL.F  [r29]          ;vector 5 (jump to ilink1)
              NOP                  ;instruction padding
ivect6:      JAL    iservice6      ;vector 6
```

Reset

A reset is an asynchronous, external reset signal that causes the ARCtangent-A4 processor to perform a “hard” reset. Upon reset, various internal states of the ARCtangent-A4 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 ARCtangent-A4 processor 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 ARCtangent-A4 processor.

NOTE The implemented ARCtangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Memory Error

A memory error can be caused by an instruction fetch from, a load from or a store to an invalid part of memory. In the basecase ARCtangent-A4 processor, this exception is non-recoverable in that the instruction that caused the error cannot be returned to.

NOTE The implemented ARCtangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Instruction Error

If an invalid instruction is fetched that the ARCtangent-A4 processor cannot execute, then an instruction error is caused. In the basecase ARCtangent-A4 processor, this exception is non-recoverable in that the instruction that caused the error cannot be returned to. The standard instruction field (I[4:0]) is used to decode whether the instruction is valid. This means that a non-implemented single-operand instruction will not generate an instruction error when executed.

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

Interrupt Times

Interrupts are held off for one cycle when an 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.

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

Alternate Interrupt Unit

It should be assumed that the ARCTangent-A4 processor adheres to the interrupt mechanism according to this chapter. It is possible, however, that an alternate interrupt unit may be provided on a particular system. The interrupt unit contains the exception and interrupt vector positions, the logic to tell the ARCTangent-A4 processor which of the 3 levels of interrupt has occurred, and the arbitration between the interrupts and exceptions.

The interrupt unit can be modified to alter the priority of interrupts, the vector positions and the number of interrupts. The 3 levels of interrupt which are set with the status register and the return mechanism through link registers can not be altered. Further masking bits and extra link registers can be provided by the

use of extensions in the auxiliary and core register set. How this would be done is entirely system dependent.

NOTE The implemented ARCtangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Chapter 5 — Instruction Set Summary

Introduction

This chapter contains an overview of the types of instructions in the ARCTangent-A4 processor. The types of instruction in the ARCTangent-A4 processor are:

- Arithmetic and Logical ADD, AND, OR...etc.
- Single Operand FLAG, MOV, LSL...etc.
- Jump, Branch and Loop J,B, LP...etc.
- Load and Store LD, ST...etc.
- Control BRK, SLEEP, SWI...etc

NOTE The implemented ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

For the operations of the instructions the notation shown in Table 4 is used.

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 operations (e.g.: SUB, BIC) All arithmetic and logical instructions can be conditional and/or set the flags. However, instructions using the short immediate addressing mode can *not* be conditional.

Instruction	Operation	Description
ADD	$a \leftarrow b + c$	add
ADC	$a \leftarrow b + c + C$	add with carry

Instruction	Operation	Description
SUB	$a \leftarrow b - c$	subtract
SBC	$a \leftarrow (b - c) - C$	subtract with carry
AND	$a \leftarrow b \text{ and } c$	logical bitwise AND
OR	$a \leftarrow b \text{ or } c$	logical bitwise OR
BIC	$a \leftarrow b \text{ and not } c$	logical bitwise AND with invert
XOR	$a \leftarrow b \text{ exclusive-or } c$	logical bitwise exclusive-OR

Table 6 Arithmetic and Logical Instructions

The syntax for arithmetic and logical instructions is:

$op<.cc><.f> \quad a,b,c$

Examples:

```

AND      r1,r2,r3    ; r1 replaced by r2 AND r3
AND.NZ   r1,r2,r3    ; if zero flag not set then r1 is replaced by
                        ; r2 AND r3
AND.F    r1,r2,r3    ; r1 is replaced by r2 AND r3 and appropriate
                        ; flags set
AND.NZ.F r1,r2,r3    ; if zero flag not set then r1 is replaced by
                        ; r2 AND r3 and the appropriate flags set
    
```

Null Instruction

Many instructions can be encoded in such a way that no operation is performed. This is very useful if a NOP instruction is required. To encode a NOP, it is just a matter of having short immediate data in all register fields and not setting flags. For example, the encoding of NOP is actually equivalent to:

```
XOR 0x1FF,0x1FF,0x1FF
```

Single Operand Instructions

Some instructions require just a single operand. These include flag and rotate instructions. These instructions are of the form $a \leftarrow op \ b$ where the destination (a) is replaced by the operation (op) on the operand source (b). Single operand instructions can be conditional and/or set the flags. However, instructions using the short immediate addressing mode can *not* be conditional.

The following table shows the move and extend functions.

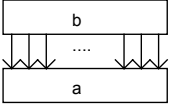
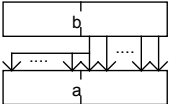
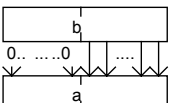
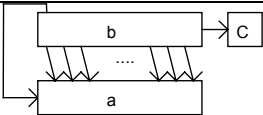
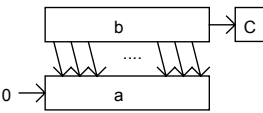
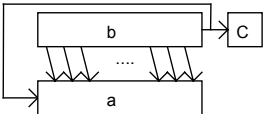
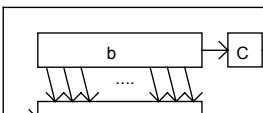
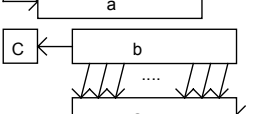
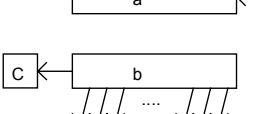
Instruction	Operation	Description
MOV		move source to register (included for instruction set symmetry, basically it is an encoding of the AND instruction: AND a,b,b)
SEX		sign extend, byte or word
EXT		zero extend, byte or word

Table 7 Single Operand Instructions: Move and Extend

The following table shows rotates and shifts.

Instruction	Operation	Description
ASR		arithmetic shift right
LSR		logical shift right
ROR		rotate right
RRC		rotate right through carry
ASL		arithmetic shift left (included for instruction set symmetry, it's an encoding of ADD instruction: ADD a,b,b)
LSL		logical shift left (same as ASL)

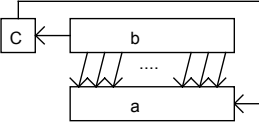
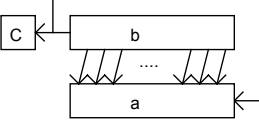
Instruction	Operation	Description
RLC		rotate left through carry (included for instruction set symmetry, it is basically an encoding of the ADC instruction: ADC.F a,b,b)
ROL		rotate left (included for instruction set symmetry, it takes 2 cycles to execute: ADD.F a,b,b then ADC a,a,0)

Table 8 Single Operand Instructions: Rotates and Shifts

The following table shows some special single operand instructions that affect registers other than core registers.

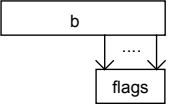



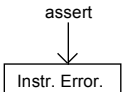
Instruction	Operation	Description
FLAG		move low bits of source into flags in the STATUS register
FLAG 1		Halt - set H bit in STATUS register. (decoded at stage 3, halts the ARCtangent-A4, processor leaving other bits unchanged)
BRK		Break - set H bit in STATUS register (decoded at stage one, flushes the earlier instructions from the pipe and halts the ARCtangent-A4 processor, other bits are unchanged)
SLEEP		Sleep - set ZZ bit in DEBUG register (decoded at stage two, flushes the earlier instructions from the pipe, puts the processor to sleep mode and stalls the ARCtangent-A4 processor, other bits are unchanged)
SWI		Software Interrupt – generate an instruction error exception (decoded at stage 2, execution jumps to the interrupt vector for instruction error, ILINK2 is used as the link register)

Table 9 Single Operand Instructions: Flags and Halts

The syntax for single operand instructions is:

op<.cc><.f> a, b

Examples:

```
ROR      r1,r2      ; rotate right r2 by one and put result in r1
FLAG     r1         ; move low bits of r1 into flags register
ROR.NZ   r1,r2      ; if zero flag not set then r1 is replaced by
                    ; r2 rotated right by one
ROR.F    r1,r2      ; r1 is replaced by r2 rotated right by one
                    ; and appropriate flags set
ROR.NZ.F  r1,r2      ; if zero flag not set then r1 is replaced by
                    ; ROR r2 and the appropriate flags set
FLAG.NZ   r1         ; if zero flag not set then update flags with
                    ; r1
```

Jump, Branch and Loop Operations

Although most instructions can be conditional, additional program control is provided with jump (J, JL), branch (B, BL) and loop (LP) instructions. 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.

This means that if an instruction setting the flags is immediately followed by a jump, then a single cycle stall will be incurred before executing the jump instruction (Even if the jump is unconditional). In this case, performance can be increased by inserting a useful non-flag setting instruction between the flag setting instruction and the jump.

Instruction	Operation (if cc true)	Description
Jcc	pc ← addr	Jump
JLcc	blink ← pc	Jump and link
	pc ← addr	(ARCOVER 0x06 and higher)
Bcc	pc ← reladdr + pc	Branch
BLcc	blink ← pc	branch and link
	pc ← reladdr + pc	
LPcc	lp_end ← addr	Set up Zero-overhead loop
	lp_start ← pc	

Table 10 Jump, Branch and Loop Instructions

Due to the pipeline in the ARctangent-A4 processor, the jump instruction does not take effect immediately, but after a one cycle delay. The execution of the immediately following instruction after a jump, branch or loop can be controlled. This instruction is said to be in the *delay slot*. The branch and link instruction (BL) and the jump and link instruction (JL for the ARctangent-A4 basecase processor version 6 and higher) also save the whole of the status register to the link register. This status register is taken either from the first instruction following the branch (current PC) or the instruction after that (next PC) according to the delay slot execution mode. The modes for specifying the execution of the delay slot instruction are:

Mode	Operation	Link Register
ND	No Delay slot instruction (default) Only execute next instruction when <i>not</i> jumping	Link to current PC
D	Delay slot instruction Always execute next instruction	Link to next PC
JD	Jump Delay slot instruction Only execute next instruction when jumping	Link to next PC

Branch type instructions use 20-bit relative addressing. The syntax of the branch type instruction is:

op<cc><.dd> reladdr

Examples:

```

LP      end_of_loop      ; set up loop registers
LPNZ    end_of_loop      ; if not zero set up loop regs
                    ; otherwise jump to end_of_loop
BL      subroutine1      ; Branch and link to subroutine1
                    ; saving status reg to BLINK
BL.D    subroutine1      ; and always execute next instruction
BNE     nother_bit       ; if zero flag not set then branch
                    ; to nother_bit
BNE.JD  label            ; if zero flag not set then execute
                    ; next instruction and branch to
                    ; label else skip next instruction

```

The jump instruction uses 32-bit absolute addressing. To enable the correct flag state when returning from interrupts the jump instruction also has a flag set field.

NOTE If the jump instruction is used with long immediate data, then the delay slot execution mechanism does not apply, but should default to .JD for JLcc.

For ease of programming, an alternative syntax is allowed when setting flags with the jump instruction.

The syntax of the jump instruction is:

op<cc><.dd><.f> [addr]

or op<cc><.dd>.f [addr],flag_value

Examples:

```
JAL    [r1]        ; jump to address in register r1
JAL    start       ; jump to start
JNZ    specbit     ; If zero flag clear jump to specbit
JAL.ND.F [r29]     ; return from interrupt and restore flags too
JAL.F  end,64      ; jump always to end and set Z bit
JL     subroutine; jump and link always to subroutine
        ; saving status reg to BLINK
JLAL   sub1        ; jump and link always to sub1
        ; saving status reg to BLINK
```

Zero Overhead Loop Mechanism

The ARctangent-A4 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.

NOTE The LP_START, LP_END and LP_COUNT registers are only 24 bit registers, with the top 8 bits reading as zeros. The maximum number of loop iterations is 16,777,216 (if LP_COUNT = 0 on entry). 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 into. If the condition code test for the LP instruction returns *false*, then a branch occurs to the address specified in the LP instruction. 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.

NOTE 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 (0xFFFFFC) 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.

When there is *not* a pipeline stall, an interrupt, a branch or a jump then the loop mechanism comes into operation.

The operation of the loop mechanism is such that PC+1 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 increment normally and LP_COUNT is decremented. This is illustrated in Figure 5.

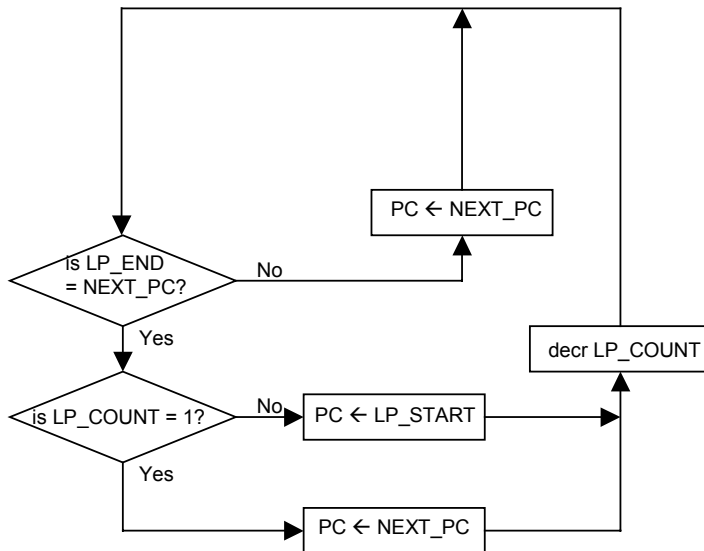


Figure 5 PC Update and Loop Detection Mechanism for Loops

The use of zero delay loops is illustrated in the following code sample:

```

        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

```

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

LP_COUNT must not be loaded directly from memory

In the current microarchitecture of the ARctangent-A4 processor there is no shortcut path to the LP_COUNT register. This path is used by returning loads, to boost performance.

As a consequence of not having the shortcut available, the LP_COUNT register should **not** be used as the destination of a load instruction. Attempting to do so may cause an incorrect value to be loaded into LP_COUNT.

The following is an example of code that may not function correctly:

```
LD    LP_COUNT,[r0]    ; caution!! LP_COUNT loaded from memory!
```

Instead, an intermediary register should be used, as follows:

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

This second example loads a value into a register (a process that does have a shortcut path and which, therefore, will function correctly). The register value is loaded into the LP_COUNT register, a process that does not require shortcutting and which will function correctly.

Single instruction loops

Single instruction loops cannot be set up with the LP instruction. The LP instruction can set up loops with 2 or more instructions in them. However, it is possible to set up a single instruction loop with the use of the LR and SR instructions. If a single instruction loop is attempted to be set up with the LP instruction, as in the following example, then the instruction in the loop (OR) will be executed once and then the code following the loop (ADD) will be executed as normal. The LP_START and LP_END registers *will* be updated by

the time the instruction after the attempted loop (ADD) is being fetched, which is, however, too late for the loop mechanism.

```

      LP      loop_end      ; this will execute only once
loop_in: OR    r21,r22,r23  ; single instruction in loop
loop_end:      ADD    r19,r19,r20  ; first instruction after loop

```

If the user wishes to have single instruction loops, then code like that in the following code example can be used. Notice, there has to be a delay to allow the loop start and loop end registers to be updated with the SR instruction. The code basically updates the registers in the loop mechanism that would normally be updated by the LP instruction.

```

      MOV     LP_COUNT,5      ; no. of times to do loop
      MOV     r0,dooploop>>2 ; convert to long-word size
      ADD     r1,r0,1         ; add 1 to dooploop 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
      ADD     r19,r19,r20     ; first instruction after loop

```

Loop count register

The loop count register, unlike other core registers, has short cutting disabled (See Chapter 10 — Pipeline and Timings). This means that there must be at least 2 instructions (actually 2 cycles) between an instruction writing LP_COUNT and one reading LP_COUNT.

```

      MOV     LP_COUNT,r0     ; update loop count register
      MOV     r1,LP_COUNT     ; old value of LP_COUNT
      MOV     r1,LP_COUNT     ; old value of LP_COUNT
      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 3 instructions (actually 3 cycles) between it and the last instruction in the loop. In the following example, the MOV instruction will override the loop mechanism (which would decrement LP_COUNT) and the loop will be executed one more time than expected. The MOV instruction must be followed by a NOP for correct execution. The following code sample shows an invalid count loop setup.

```

      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

```

The following code sample shows a valid count loop setup

```

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

```

When reading from the loop count register (LP_COUNT) the user must be aware that the value returned is that value of the counter that applies to the next instruction to be executed. If the last instruction in a loop reads LP_COUNT, then the value returned would be that value after the loop mechanism has updated it. The following code example shows a 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

```

Branch and jumps in loops

Jumps or branches without linking will work correctly in any position in the loop. There are, however, some side effects for delay slots and link registers when a branch or jump is the last instruction in a loop:

Firstly, it is possible that the branch or jump instruction is contained in the very last long-word position in the loop. This means that the instruction in the delay slot (See Chapter 5 — Instruction Set Summary and Chapter 10 — Pipeline and Timings) would be either the first instruction *after* the loop or the first instruction *in* the loop (pointed to by loop start register) depending on the result of the loop mechanism. The instruction in the delay slot will be that which would be executed if the branch or jump was replaced by a NOP.

If a branch-and-link or jump-and-link instruction is used in the one before last long-word position in a loop, then the return address stored in the link register (BLINK) may contain the wrong value. The following instructions will store the address of the first instruction *after* the loop, and therefore should not be used in the second to last position:

```

BLcc.D      address
BLcc.JD     address
JLcc.D      [Rn]

```

JLcc.JD	[Rn]
JLcc	address

If the ND delay slot execution mode is used for branch-and-link or jump-and-link instruction in the one before last long-word position in a loop, then the return address is stored correctly in the link register.

The loop count does not decrement if the instruction fetched was subsequently killed as the result of a branch/jump operation. For these reasons it is recommended that subroutine calls should not be used within the loop mechanism.

Instructions with long immediate data: correct coding

Instructions with long immediate data will work correctly with the zero overhead loop mechanism as long as the LP instruction is used. Even if the instruction containing the long immediate data is seen as the last instruction in the loop. Here, we are setting up the loop with an instruction that uses long immediate data. The `loop_end` label points to the first instruction after the loop.

```
        MOV    LP_COUNT,r0    ; do loop r0 times (flags not set)
        LP     loop_end
loop_in: ...
        ...
        XOR     r1,r2,r3
        OR      r21,r22,2048  ; last instruction in loop
loop_end:
        ADD     r19,r19,r20    ; first instruction after loop
```

Instructions with long immediate data: incorrect coding

It is difficult, but nonetheless possible, that an instruction that uses long immediate data could fall across the very last long-word position in the loop. This means that the long immediate data would be either be taken from the first location *after* the loop or the first location *in* the loop (pointed to by loop start register) depending on the result of the loop mechanism. It is unlikely that this would occur with sensible coding, but the following example shows how it could be done. Here, we are setting up the loop mechanism by writing the loop registers directly. The only register write shown here is the writing of `LP_END`.

```

MOV    r1,limmloop>>2    ; convert to long-word size
ADD    r1,r1,1            ; add 1 to limmloop address
SR     r1,[LP_END]        ; set up loop end register
NOP
...
...
NOP
limmloop: OR    r21,r22,2048 ; instruction across loop end
        ADD    r19,r19,r20  ;

```

The OR instruction in the above example has a long immediate value of 2048 which crosses the loop end address. This is accomplished by getting the address of the OR instruction and adding 1 to the address to force the LP_END register to point to the position just after the OR instruction, but before the long immediate value.

Valid instruction regions in loops

To summarise the effect that the loop mechanism has on these special cases see Figure 6. As an example, if an instruction that reads LP_COUNT is in position `insn` (like `MOV r1,LP_COUNT`), then the value that the instruction reads will be that value after the loop mechanism updated it.

For further details see [Chapter 7 — Register Set Details](#).

The Loop	Loop Set Up	Writing	Reading	Delay Slots	Immediates
	LP loop_end	LP_COUNT	LP_COUNT	Bcc or Jcc	limm instr
Loop_start: Ins1	works normally	update before loop mechanism	value before loop mechanism	Works normally	works normally
Ins2
Ins3
...
...
Insn-4	...	update before loop mechanism
Insn-3	...	overwrite loop mechanism
Insn-2	works normally	update after loop mechanism	...	Works normally	...
Insn-1	loop end not set up in time	update after loop mechanism	value before loop mechanism	wrong return address may be stored in BLINK LP_COUNT decrements according to delay slot mode	works normally
Insn	loop end not set up in time	update after loop mechanism	value after loop mechanism	loop_count decrements delay slot = ins1 or outs1	imm data = ins1 or outs1
Loop_end: Outins1 Outins2					

Figure 6 Valid Instruction Regions in Loops

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 ARCTangent-A4 processor from performing any instructions beyond the breakpoint. The pipeline is also flushed upon decode of this instruction. To restart the ARCTangent-A4 processor at the correct instruction the old instruction is rewritten into main memory. It is immediately followed by an invalidate instruction cache line command (if an instruction cache has been implemented) to ensure that the correct instruction is loaded into the cache before being executed by the ARCTangent-A4 processor. The program counter must also be rewritten in order to generate a new instruction fetch, which reloads the instruction. Most of the work is performed by the debugger with regards to insertion, removal of instructions with the breakpoint instruction.

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

NOTE The breakpoint instruction sets the BH bit (refer to section Programmer's Model) in the Debug register when it is decoded at stage one of the pipeline. This allows the debugger to determine what caused the ARCTangent-A4 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 ARCTangent-A4 processor.

A breakpoint instruction may be inserted into any position:

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

The above code shows assembly code with BRK instruction

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 ARCTangent-A4 instruction.

The breakpoint instruction can be placed anywhere in a program. 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.

BRK instruction in delay slot

Due to stage 2 to stage 1 dependencies, the breakpoint instruction behaves differently when it is placed in the delay slots of Branch, and Jump instructions. In these cases, the ARCTangent-A4 processor will stall stages one and two of the pipeline while allowing instructions in subsequent stages (three and four) to proceed to completion.

The following example shows BRK in a delay slot of 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                                ;<---- caution break inserted
                                ; into delay slot here
OR     r5, r4, 0x10
```

The link register is not updated for Branch and Link, BL, (or Jump and Link, JL) instruction when the BRK instruction is placed in the delay slot. When the ARCTangent-A4 processor is started, the link register will update as normal. Interrupts are treated in the same manner by the ARCTangent-A4 processor as Branch, and Jump instructions when a BRK instruction is detected. Therefore, an interrupt that reaches stage two of the pipeline when a BRK instruction is in stage one will keep it in stage two, and flush the remaining stages of the pipeline. It is also important to note that an interrupt that occurs in the same cycle as a breakpoint is held off as the breakpoint is of a higher priority. An interrupt at stage three is allowed to complete when a breakpoint instruction is in stage one.

Sleep Instruction

The sleep mode is entered when the ARCTangent-A4 processor encounters the SLEEP instruction. It stays in sleep mode until an interrupt or restart occurs. Power consumption is reduced during sleep mode since the pipeline ceases to change state, and the RAMs are disabled. More power reduction is achieved when clock gating option is used, whereby all non-essential clocks are switched off.

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

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

...

The SLEEP instruction is a single operand instruction without flags or operands. The SLEEP instruction is decoded in pipeline stage 2. If a SLEEP instruction is detected, then the sleep mode flag (ZZ) is immediately set and the pipeline stage 1 is stalled. A flushing mechanism assures that all earlier instructions are executed until the pipeline is empty. The SLEEP instruction itself leaves the pipeline during the flushing. When in sleep mode, the sleep mode flag (ZZ) is set and the pipeline is stalled, but not halted. The host interface operates as normal allowing access to the DEBUG and the STATUS registers and it can halt the processor. The host cannot clear the sleep mode flag, but it can wake the ARCTangent-A4 processor by halting then restarting ARCTangent-A4 processor. The program counter PC points to the next instruction in sequence after the sleep instruction.

The ARCTangent-A4 processor will wake from sleep mode on an interrupt or when the ARCTangent-A4 is restarted. If an interrupt wakes up the ARCTangent-A4 processor, the ZZ flag is cleared and the instruction in pipeline stage 1 is killed. The interrupt routine is serviced and execution resumes at the instruction in sequence after the SLEEP instruction. When the ARCTangent-A4 processor is started after having been halted, the ZZ flag is cleared.

SLEEP instruction in delay slot

A SLEEP instruction can be put in a delay slot as in the following code example:

```
BAL.D  after_sleep
SLEEP

after_sleep:
ADD    r1, r1, 0x2
```

In this example, the ARCTangent-A4 processor goes to sleep after the branch instruction has been executed. When the ARCTangent-A4 processor is sleeping, the PC points to the “add” instruction after the label “after_sleep”. When an interrupt occurs, the ARCTangent-A4 processor wakes up, executes the interrupt service routine and continues with the “add” instruction. If the delay slot is killed, as in the following code example, the SLEEP instruction in the delay slot is never executed:

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

SLEEP instruction in delay slot of Jump

The SLEEP instruction is normally used in RTOS type applications by using a J.F with SLEEP in the delay slot. This allows the interrupts to be re-enabled at the same time as SLEEP is entered, i.e an atomic operation.

SLEEP instruction in single step mode

The SLEEP instruction acts as a NOP during single step mode, because every single-step is a restart and the ARCTangent-A4 processor wakes up at the next single-step. Consequently, the SLEEP instruction behaves exactly like a NOP propagating through the pipeline.

Software Interrupt Instruction

The execution of an undefined extension instruction in ARCTangent-A4 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.

SWI instruction format

The SWI instruction is a single operand instruction in the same class as the SLEEP and BREAK instructions and takes no operands or flags.

Load and Store Operations

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. This is accomplished with the address write-back suffix: .A

Loads are passed to the memory controller once the address has been calculated, and the register which is the destination of the load is tagged to indicate that 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 which 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 which 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.

Byte and word loads can be sign extended to 32-bits, or simply loaded into the appropriate register with unused bits set to zero. This is accomplished with the sign extend suffix: `.X`

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. Note that if the offset is not required during a store, the value encoded will be set to 0.

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 ARCtangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Instruction	Operation	Description
LD	$a \leftarrow [b+c]$	load
ST	$[b + \text{shimm}] \leftarrow c$	store

Table 11 Load and Store Instructions

The syntax of the load instruction is:

`op<zz><.x><.a><.di>` `a, [b, c]`

Examples:

```
LD      r1,[r2,r3] ; r1 replaced by data at address
                ; r2+r3
LD      r4,[r2,10] ; r4 replaced by data at address
                ; r2 plus offset 10
LD.A.DI r4,[r2,10] ; r4 replaced by data at address
                ; r2 plus offset 10 and write-back
                ; result of address calculation to
                ; r2, bypassing data cache
LDW.X   r1,[r2,r3] ; r1 replaced by sign extended word
                ; from address at r2+r3
```

The syntax of the store instruction is:

```
op<zz><.a><.di>      c,[b,offset]
```

Examples:

```
ST      r1,[r2]      ; data at address r2 replaced by r1
STB     r1,[r2]      ; bottom 8 bits of r1 put into
                ; address r2. Offset = 0
ST.A    r1,[r2,14]   ; with write-back
```

Auxiliary Register Operations

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

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

Instruction	Operation	Description
LR	$a \leftarrow \text{aux. reg } [b]$	load from auxiliary register
SR	$\text{aux. reg.}[b] \leftarrow c$	store to auxiliary register

Table 12 Auxiliary Register Operations

The syntax of the load from auxiliary register instruction is:

```
op      a,[b]
```

Examples:

```

LR      r1,[r2]      ; r1 replaced by data from auxiliary
                ; register pointed to by r2
LR      r4,[10]      ; r4 replaced by data from auxiliary
                ; register number 10

```

The syntax of the store to auxiliary register instruction is:

```
op      c,[b]
```

Examples:

```

SR      r1,[r2]      ; data in auxiliary register pointed
                ; to by r2, replaced by data in r1
SR      r1,[14]      ; data in auxiliary register number
                ; 14 replaced by data in r1

```

Extension Instructions

These operations are of the form **a ← b op c** (or **a ← op b** for single operand instructions) 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 operations (e.g.: SUB, BIC). All arithmetic and logical instructions can be conditional and/or set the flags. However, instructions using the short immediate addressing mode can *not* be conditional.

The syntax for extension instructions is:

```
op<.cc><.f>    a,b,c
```

The syntax for extension single operand instructions is:

```
op<.cc><.f>    a,b
```

Optional Extensions Library

The extensions library consists of a number of components that can be used to add functionality to an ARctangent-A4 processor. These components are function units, which are interfaced to the ARctangent-A4 processor through the use of extension instructions or registers.

The library currently consists of the following components:

- 32-bit Multiplier, small (10 cycle) implementation
- 32-bit Multiplier, fast (4 cycle) implementation

- 32-bit Barrel shift/rotate block (single cycle)
- 32-bit Barrel shift/rotate block (multi cycle)
- Normalise (find-first-bit) instruction
- Swap instruction
- MIN/MAX instructions

Multiply 32 X 32

Two versions of the scoreboarded 32x32 multiplier function are available, 'fast' and 'small', taking four and ten cycles respectively. The full 64-bit result is available to be read from the core register set. The middle 32 bits of the 64-bit result are also available. The multiply is scoreboarded in such a way that if a multiply is being carried out, and if one of the result registers is required by another ARctangent-A4 instruction, the processor stalls until the multiply has finished.

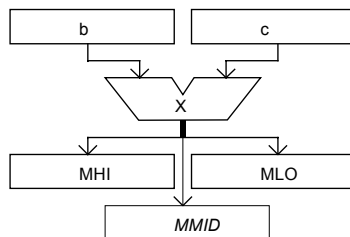


Figure 7 32x32 Multiply

The syntax of the multiply instruction is:

`op<.cc> <0>,b,c`

Example:

`MUL64 r1,r2 ;multiply r1 by r2`

Quick exit for conditional multiplies

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

Reading and pre-loading the 32X32 multiply results

The results are accessed via the read-only extension core registers MLO, MMID and MHI. The extension auxiliary register MULHI is used to restore the multiply

result register if the multiply has been used, for example, by an interrupt service routine. See Multiply restore register.

Barrel shift/rotate block

This block 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.

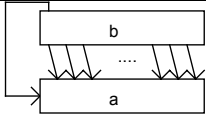
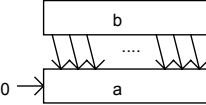
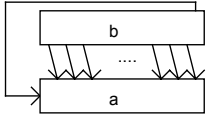
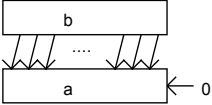
Instruction	Operation	Description
ASR		arithmetic shift right, sign filled
LSR		logical shift right, zero filled
ROR		rotate right
ASL		arithmetic shift left, zero filled

Figure 8 Barrel Shift Operations

The syntax for the barrel shift operations is:

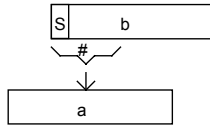
`op<.cc><.ff> a,b,c`

Example:

`ASR r2,r2,6 ;arithmetic shift right r2 by 5 places`

Normalize instruction

The NORM instruction gives the normalisation integer for the signed value in the operand. The normalisation integer is the amount by which the operand should be shifted left to normalise it as a 32-bit signed integer. To find the normalisation integer of a 32-bit register by using software without a NORM instruction, requires many ARctangent-A4 instruction cycles.

**Figure 9 Norm Instruction**

Uses for the NORM instruction include:

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

The syntax for the normalize instruction is:

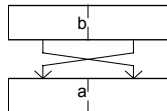
op<.cc><.f> a,b

Example:

```
NORM r1,r2 ; find normalization integer for r2
           ; and write into r1
```

SWAP instruction

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

**Figure 10 SWAP Instruction**

The syntax for the swap instruction is:

op<.cc><.f> a,b

Example:

```
SWAP r1,r2 ; swap upper and lower 16 bits of r2
           ; and write into r1
```


MIN/MAX instructions

These instructions are useful in applications where sorting takes place. Two signed 32-bit words are compared, and either the larger or smaller of the two is returned, depending on which instruction is being used.

The syntax for the min/mas instructions is:

`op<.cc><.f> a,b,c`

Example:

`MIN r1,r2,r3 ; write minimum of r2 and r3 into r1`

Chapter 6 — Condition Codes

Introduction

The ARctangent-A4 processor has an extensive instruction set most of which can be carried out conditionally and/or set the flags. Those instructions using short immediate data can not have a condition code test.

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

Condition Code Register

The condition code register is part of the status register.

The status register (STATUS), shown in Figure 11, contains the condition codes: zero (Z), negative (N), carry (C) and overflow (V); the interrupt mask bits (E[2:1]); the halt bit (H); and the program counter (PC[25:2]).

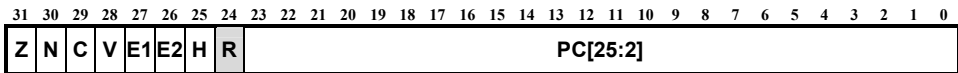


Figure 11 Status Register

Condition Code Register Notation

In the instruction set details in the next chapter the following notation is used:

Condition codes:



where:

Z (zero)	Set if the result equals zero. Otherwise bit cleared
N (negative)	Set if most significant bit of result is set. Else cleared
C (carry)	Set if a carry is generated after the result of an arithmetic operation. Otherwise bit cleared.
V (overflow)	Set if there was an overflow generated from an arithmetic operation. Otherwise bit cleared.

The convention used in the next chapter for the effect of an operation on the condition codes is:

*	Set according to the result of the operation		
.	Not affected	?	Bit undefined after the operation
0	Bit cleared	1	Bit set

Condition Code Test

Table 13 Condition Codes shows condition names and the conditions they test.

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

Mnemonic	Condition	Test	Code
LT	Less than (signed)	(N and /V) or (/N and V)	0x0B
LE	Less than or equal to (signed)	Z or (N and /V) or (/N and V)	0x0C
HI	Higher than (unsigned)	/C and /Z	0x0D
LS	Lower than or same (unsigned)	C or Z	0x0E
PNZ	Positive non-zero	/N and /Z	0x0F

Table 13 Condition Codes

NOTE PNZ does not have an inverse condition.

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

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

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

NOTE The implemented AR Ctangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Chapter 7 — Register Set Details

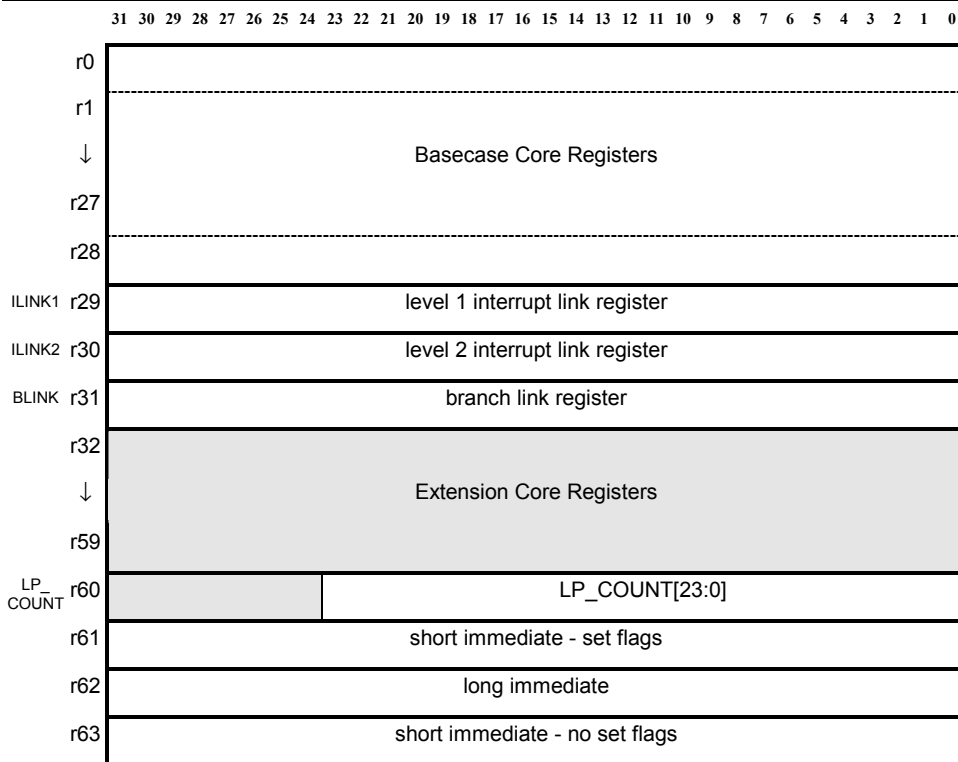


Figure 12 Core Register Map

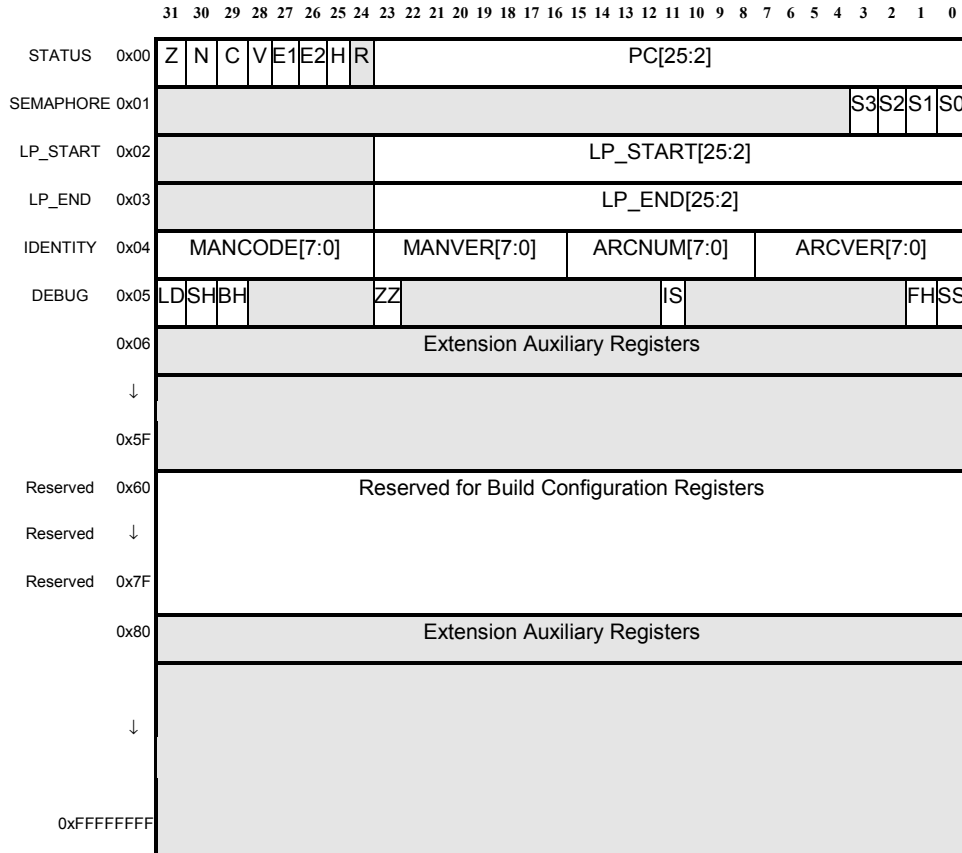


Figure 13 Auxiliary Register Set

Core Register Set

The core register set in the ARctangent-A4 processor is shown in Figure 12 and Table 14. Other predefined registers are in the auxiliary register set and they are shown in Figure 13 and Table 16.

Number	Core register name	Function
0-28	r0-r28	General Purpose Registers
29	ILINK1 or r29	Maskable interrupt link register
30	ILINK2 or r30	Maskable interrupt link register

Number	Core register name	Function
31	BLINK or r31	Branch link register
32-59	r32-r59	Register space reserved for extensions
60	LP_COUNT	Loop count register (24 Bits)
61	-	Short immediate data indicator setting flags
62	-	Long immediate data indicator
63	-	Short immediate data indicator not setting flags

Table 14 Core Register Map

The general purpose registers (r0-r28) can be used for any purpose by the programmer.

Link registers

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

In the basecase ARCTangent-A4 processor prior to version 7, the branch-and-link and jump-and-link instructions write to the BLINK register in a way that bypasses the LD scoreboard mechanism. Basecase ARCTangent-A4 processor version 7 remedies this problem by enabling additional scoreboarding on the link registers.

Loop count register

The loop count register (LP_COUNT) is used for zero delay loops. Because LP_COUNT is decremented if the program counter equals the loop end address and also LP_COUNT does not have next cycle bypass like the other core registers, it is not recommended that LP_COUNT be used as a general purpose register, see later in this documentation for details. Note that LP_COUNT is only 24 bits wide.

Immediate data indicators

Register positions 63 to 61 are reserved for encoding immediate data addressing modes onto instruction words. They are reserved for that purpose and are not available to the programmer as general purpose registers.

Extension core registers

The register set is extendible in register positions 32-59 (r32-r59). Results of accessing the extension register region are undefined in the basecase version of the ARctangent-A4 processor. If a core register is read that is not implemented, then an unknown value is returned. No exception is generated. Writes to non implemented core registers are ignored.

NOTE The implemented ARctangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Multiply result registers

Table 15 shows the defined extension core registers for the optional multiply.

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

Table 15 Multiply Result Registers

Auxiliary Register Set

The auxiliary register set contains special status and control registers. Auxiliary registers occupy a special address space that is accessed using special load and store instructions, or other special commands. The basecase ARctangent-A4 processor uses 6 status and control registers, and reserves the additional registers 0x60 to 0x7F, leaving the rest of the 2^{32} registers for extension purposes. If an auxiliary register is read that is not implemented, then the IDENTITY register contents is returned. No exception is generated. Writes to non implemented auxiliary registers are ignored.

NOTE The implemented ARctangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Number	Auxiliary register name	Description
0x0	STATUS	Status register
0x1	SEMAPHORE	Inter-process/Host semaphore register

0x2	LP_START	Loop start address (24 bits)
0x3	LP_END	Loop end address (24 bits)
0x4	IDENTITY	ARCTangent-A4 Identification register
0x5	DEBUG	Debug register
0x60 - 0x7F	RESERVED	Build Configuration Registers

Table 16 Auxiliary Register Set

Status register

The STATUS register contains the PC, the condition flags and interrupt mask bits. LP_START and LP_END are the other registers used by the zero delay loop mechanism. The SEMAPHORE register is used to control inter-process communication. IDENTITY is used by the host and ARCTangent-A4 processor to determine the version number of the processor and other implementation specific information. DEBUG is used by the host to test and control the ARCTangent-A4 processor during debug situations.

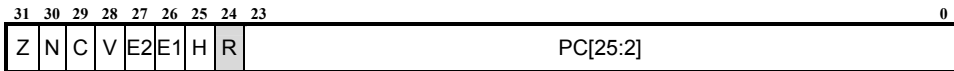


Figure 14 Status Register

The status register (STATUS), shown in Figure 14, contains the condition codes: zero (Z), negative (N), carry (C) and overflow (V); the interrupt mask bits (E[2:1]); the halt bit (H); and the program counter (PC[25:2]). When STATUS is read with a LR instruction, it will return the address of the instruction following the LR and the current condition flags. STATUS cannot be written with SR instruction. The FLAG and Jcc instructions are used to affect the status register.

Semaphore register

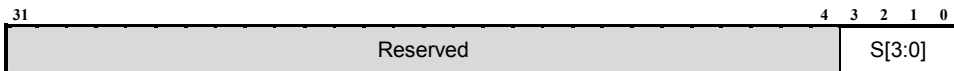


Figure 15 Semaphore Register

The SEMAPHORE register, Figure 15, is used to control inter-process or ARCTangent-A4 processor to host communication. The basecase ARCTangent-A4 processor has at least 4 semaphore bits (S[3:0]). The remaining bits of the

semaphore register are reserved for future versions of the ARCTangent-A4 processor.

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 ARCTangent-A4 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 ARCTangent-A4 processor and the host. In other words, if the host claims a particular semaphore bit, the ARCTangent-A4 processor will not be able to claim that same semaphore bit until the host has released it. Conversely, if the ARCTangent-A4 processor claims a particular semaphore bit, the host will not be able to claim that same semaphore bit until the ARCTangent-A4 processor has released it.

The semaphore bits are cleared to 0 after a reset, which is the state where neither the ARCTangent-A4 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. This could be accomplished by keeping a local copy, or reading the semaphore register, and ORing that value with the bit to be claimed before writing back to the semaphore register.

Example:

```
.equ SEMBIT0,1      ; constant to indicate semaphore bit 0
.equ SEMBIT1,2      ; constant to indicate semaphore bit 1
.equ SEMBIT2,4      ; constant to indicate semaphore bit 2
.equ SEMBIT3,8      ; constant to indicate semaphore bit 3
LR    r2,[SEMAPHORE] ; r2 <= semaphore pattern already attained
OR    r2,r2,SEMBIT1  ; r2 <= semaphore pattern attained and wanted
SR    r2,[SEMAPHORE] ; attempt to get the semaphore bit
LR    r2,[SEMAPHORE] ; read back semaphore register
AND.F 0,r2,SEMBIT1  ; test for the semaphore bit being set
                        ; EQ means semaphore not attained
                        ; NE means semaphore attained
```

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

Loop control registers

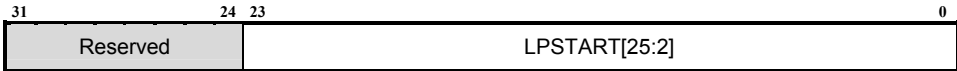


Figure 16 Loop Start Register

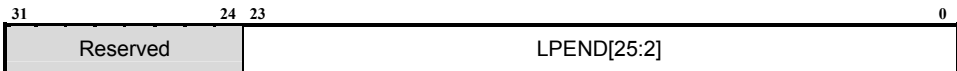


Figure 17 Loop End Register

The loop start (LP_START) and loop end (LP_END) registers contain the addresses for the zero delay loop mechanism. Figure 16 and Figure 17 show the format of these registers. The loop start and loop end registers can be set up with the special loop instruction (LP) or can be manipulated with the auxiliary register access instructions (LR and SR).

Identity register

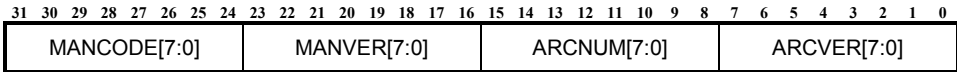


Figure 18 Identity Register

Figure 18 shows the identity register (IDENTITY). It contains the manufacturer code (MANCODE[7:0]), manufacturer version number (MANVER[7:0]), the additional identity number (ARCNUM[7:0]) and the ARCTangent-A4 basecase processor version number (ARCVER[7:0]).

Debug register

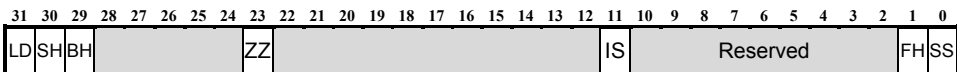


Figure 19 Debug Register

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

LD can be read at any time by either the host or the ARCTangent-A4 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 ARCTangent-A4 processor.

SH indicates that the ARCTangent-A4 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 ARCTangent-A4 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 ARCTangent-A4 processor. BH is only available for ARCTangent-A4 basecase processor version 7 or higher.

ZZ bit indicates that the ARCTangent-A4 processor is in "sleep" mode. The ARCTangent-A4 processor enters sleep mode following a SLEEP instruction. ZZ is cleared whenever the processor "wakes" from sleep mode. ZZ is only available for ARCTangent-A4 basecase processor version 7 or higher.

The force halt bit (FH) is only available for the ARCTangent-A4 basecase processor version (ARCV_{ER} in IDENTITY register) of 5 or higher. FH is a foolproof method of stopping the processor externally by the host. The host setting this bit does not have any side effects when the ARCTangent-A4 processor is halted already. FH is not a mirror of the STATUS register H bit:- clearing FH will *not* start the ARCTangent-A4 processor. FH always returns 0 when it is read. See also [Halting](#).

Single stepping is provided through the use of IS and SS. Single instruction step (IS) is used in combination with SS. When IS and SS are both set by the host the ARCTangent-A4 processor will execute one full instruction. IS is only available for ARCTangent-A4 basecase processor version 7 or higher.

SS is a write only bit that when set by the host will cause the ARCTangent-A4 processor to single cycle step. The single cycle step function enables the processor for one cycle. It should be noted that this does not necessarily correspond to one instruction being executed, since stall conditions may be present. In order to execute a single instruction, the remote system must repeatedly single-step the ARCTangent-A4 processor until the values change in either the program counter or loop count register, or use SS in combination with IS.

Extension auxiliary registers

The auxiliary register set is extendible up to the full 2^{32} register space. Results of accessing the extension auxiliary register region are undefined in the basecase version of the ARCtangent-A4 processor. If an auxiliary register is read that is not implemented, then an unknown value is returned. No exception is generated. Writes to non implemented auxiliary registers are ignored.

NOTE The implemented ARctangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Optional extensions auxiliary registers

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

Number	Name	r/w	Description
0x12	MULHI	w	High part of multiply to restore multiply state
0x7B	MULTIPLY_BUILD	r	Build configuration for: multiply
0x7C	SWAP_BUILD	r	Build configuration for: swap
0x7D	NORM_BUILD	r	Build configuration for: normalize
0x7E	MINMAX_BUILD	r	Build configuration for: min/max
0x7F	BARREL_BUILD	r	Build configuration for: barrel shift

Multiply restore register

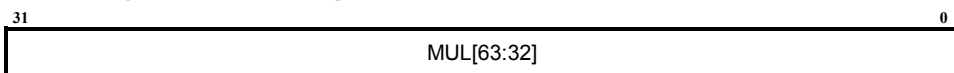


Figure 20 Multiply Restore Register

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. This is performed by reading one of the scoreboarded multiplier result registers.

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

Example

To read the upper and lower parts of the multiply results

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

To restore the multiply results

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


Chapter 8 — Instruction Set Details

Introduction

This chapter contains the detailed information about all the instructions available in the basecase version of the ARCTangent-A4 processor. They are arranged in alphabetical order.

Instruction Map

There are 32 different instruction codes, only the first 16 of which are used for the basecase ARCTangent-A4 processor according to the following table:

Basecase instruction set		
Code	Instruction and/or Type	Notes
0x00	LD <i>register + register</i>	Delayed load (core registers only)
0x01	LD <i>register + offset</i> , LR	Delayed load or load from aux. register
0x02	ST <i>register + offset</i> , SR	Buffered store or store to aux. register
0x03	<i>single operand instructions</i>	FLAG, single shifts, sign extend
0x04	Bcc	Branch conditionally
0x05	BLcc	Branch and link conditionally
0x06	LPcc	Loop set up or jump conditionally
0x07	Jcc, JLcc	Jump (and link) conditionally
0x08	ADD	Addition
0x09	ADC	Addition with carry
0x0A	SUB	Subtract
0x0B	SBC	Subtract with carry
0x0C	AND	Logical bitwise AND
0x0D	OR	Logical bitwise OR

Basecase instruction set		
Code	Instruction and/or Type	Notes
0x0E	BIC	Logical bitwise AND with invert
0x0F	XOR	Logical bitwise exclusive-OR
0x10	ASL	Optional multiple arithmetic shift left
0x11	LSR	Optional multiple logical shift right
0x12	ASR	Optional multiple arithmetic shift right
0x13	ROR	Optional multiple rotate right
0x14	MUL64	Optional 32 X 32 signed multiply
0x15	MULU64	Optional 32 X 32 unsigned multiply
0x1E	MAX	Optional maximum of two signed integers
0x1F	MIN	Optional minimum of two signed integers

NOTE The implemented ARCtangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Specially Encoded Instructions		
Code	Instruction	Notes
01,[10]	LR	Load from Aux. Register
02,[10]	SR	Store to Aux. Register
03,[00]	FLAG	Set the flags
03,[3F,0]	BRK	Break Point
03,[3F,1]	SLEEP	Sleep
03,[3F,2]	SWI	Software Interrupt
03,[01]	ASR	Arithmetic Shift Right by one
03,[02]	LSR	Logical Shift Right by one
03,[03]	ROR	Rotate Right by one
03,[04]	RRC	Rotate Right through Carry by one
03,[05]	SEXB	Sign Extend byte to long word
03,[06]	SEXW	Sign Extend word to long word
03,[07]	EXTB	Zero Extend byte to long word
03,[08]	EXTW	Zero Extend word to long word
03,[9]	SWAP	Optional swap words

03,[A]	NORM	Optional normalise Integer
--------	------	----------------------------

Table 17 Basecase Instruction Map

Addressing Modes

The addressing mode of the instruction are encoded on the instruction. There are basically only 3 addressing modes: register-register, register-immediate and immediate-immediate. However, as a consequence of the action performed by the different instruction groups, these can be expanded as shown in Table 3 Data Addressing Modes. The operating modes use the key in Table 4.

Dual operand instructions

If we take the ADD instruction as an example:

```

ADD      r1,r2,r3      ; register register
ADD.NZ   r1,r2,r3      ; conditional
ADD.F    r1,r2,r3      ; setting flags
ADD.NZ.F  r1,r2,r3      ; conditional and conditionally set flags
ADD      r1,r2,34       ; register immediate
ADD      r1,34,r2       ; immediate register
ADD      r1,255,255     ; immediate immediate (shimms MUST match)
ADD.F    0,r1,r2        ; test
ADD.F    0,r1,34        ; test with immediate
ADD.F    0,34,r1        ; test with immediate
ADD      0,0,0          ; null instruction, NOP

```

The complete operating modes for ADD are:

ADD	with result	ADD	without result
ADD<.cc><.f>	a,b,c	ADD<.cc><.f>	0,b,c
ADD<.f>	a,b,shimm	ADD<.f>	0,b,shimm
ADD<.f>	a,shimm,c	ADD<.f>	0,shimm,c
ADD<.f>	a,shimm,shimm	ADD<.f>	0,shimm,shimm
ADD<.cc><.f>	a,b,limm	ADD<.cc><.f>	0,b,limm
ADD<.cc><.f>	a,limm,c	ADD<.cc><.f>	0,limm,c
		ADD	0,shimm,shimm ;nop

Single operand instructions

Taking ROR as a single operand example:

```

ROR      r1,r2      ; register
ROR.NZ   r1,r2      ; conditional
ROR.F     r1,r2      ; setting flags
ROR.NZ.F  r1,r2      ; conditional and conditionally set flags
ROR      r1,22       ; immediate
ROR.F     0,r2       ; test

```

The complete operating modes for ROR are

ROR	with result	ROR	without result
ROR<.cc><.f>	a,b	ROR<.cc><.f>	0,b,c
ROR<.f>	a,shimm	ROR<.f>	0,shimm
ROR<.cc><.f>	a,limm	ROR<.cc><.f>	0,limm
		ROR	0,shimm ;nop

Branch type Instructions

Branch instructions:

```

B      pos          ; relative branch to pos
BNE    pos          ; conditional branch
B.D     pos         ; branch and execute next instruction
BNE.D   pos         ; conditional and always execute next

```

The operating mode for Bcc is:

Bcc
B<cc><.dd> rel_addr

Jump Instruction

Example

```

JAL      [r1]        ; jump to address in register r1
JAL      2000        ; jump to address 2000
JNZ      [r1]        ; conditional jump
JNZ      2000        ; conditional jump
JAL.D     [r1]        ; jump and execute next instruction
JNZ.D     [r1]        ; conditional jump, always execute next
JNZ.JD    [r1]        ; conditional jump, execute next only
           ; if jump taken
JAL.F     [r1]        ; jump to address and update flags
JNZ.D.F   [r1]        ; conditional, execute next, update flags
JNZ.F     2000,64     ; conditional jump to 2000 and set the Z flag

```

The operating modes for J are:

Jcc		
J<cc>.<dd><.f>	[b]	
J<cc>.<dd><.f>	limm	

Load Instruction

Example

```
LD      r1,[r2,r3] ; r1 replaced with data at r2+r3
LD      r1,[r2,20] ; r1 replaced with data at r2+20
LDB     r1,[r2,r3] ; load byte from r2+r3
LD.A    r4,[r2,10] ; r4 replaced by data at address
                  ; r2 plus offset 10 and writeback
                  ; address calculation to r2
LDW.X   r1,[r2,r3] ; r1 replaced by sign extended word
                  ; from address at r2+r3
LDW.X.A r1,[r2,r3] ; word, sign extended with writeback
                  ; from address at r2+r3
LD      r1,[900]   ; load from address 900
```

The operating modes for LD are:

LD		
LD<zz><.x><.a><.di>	a,[b,c]	
LD<zz><.x><.a><.di>	a,[b,shimm]	
LD<zz><.x>.<di>	a,[imm]	

Store instruction

Example:

```
ST      r1,[r2]           ; data at address r2 replaced by r1
ST      r1,[r2,14]        ; store with offset
STB     r1,[r2]           ; store bottom 8 bits of r1 to
                          ; address r2
ST.A    r1,[r2,14]        ; with writeback r2+14 to r2
STW.A   r1,[r2,2]         ; store bottom 16 bits of r1 to
                          ; address r2+2 and writeback
                          ; r2+2 to r2
ST      r1,[900]          ; store r1 to address 900
STB     0,[r2]            ; store byte 0 to address r2
ST      -8,[r2,-8]        ; store -8 to address r2-8
STW     80,[750]          ; store word 80 to address 750
ST      12345678,[r2+8]   ; store 12345678 to address r2+8
```

The operating modes for ST are:

ST	
ST<zz><.a><.di>	c,[b]
ST<zz><.a><.di>	c,[b,shimm]
ST<zz><.di>	c,[imm]
ST<zz><.a><.di>	0,[b]
ST<zz><.a><.di>	shimm,[b,shimm] (shimms MUST match)
ST<zz><.di>	shimm,[limm]
ST<zz><.a><.di>	limm,[b,shimm]

Load from auxiliary register instruction

Example:

```
LR    r1,[r2]    ; r1 replaced with data from aux reg pointed
                  ; to by r2
LR    r1,[20]    ; r1 replaced with data from aux reg 20
```

The operating modes for LR are:

LR	
LR	a,[b]
LR	a,[imm]

Store to auxiliary register instruction

Example:

```
SR    r1,[r2]    ; data in aux reg pointed to by r2
                  ; replaced by data in r1
SR    r1,[14]    ; data in aux reg 14 replaced by
                  ; data in r1
```

The operating modes for SR are:

SR	
SR	c,[b]
SR	c,[imm]

Instruction Encoding

The instructions are encoded according to the type of instruction.

The general encoding outlines are shown below. Some fields have additional encoding on them and are covered in detail for each instruction.

Those instructions that test the condition codes use the encoding shown in the following table.

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

Condition codes 0x10 to 0x1F are reserved for extension purposes.

NOTE The implemented ARctangent-A4 system may have extensions or customizations in this area, please see associated documentation.

Immediate data is indicated on the instruction according to the following table:

Short immediate setting flags	61/0x3D
Short immediate not setting flags	63/0x3F
Long immediate	62/0x3E

The immediate data indicator is encoded on the B or C register address field according to the ordering of operands or the type of instruction.

The result of an operation is discarded if an immediate data indicator is encoded in the destination field. If a long immediate data indicator is encoded in the destination field, *only* then, long immediate data is NOT fetched. However, if one of the source register fields, B or C, contains a long immediate data indicator as well as the destination field then long immediate data IS fetched as normal.

Register

This is the general form used for register-register and register-long-immediate addressing.

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]					A[5:0]					B[5:0]					C[5:0]					F		R	N	N	Q	Q	Q	Q	Q	Q	

I[4:0]	Instruction opcode	A[5:0]	Destination register address
B[5:0]	Operand 1 address	C[5:0]	Operand 2 address
F	Flags set field	N	Jump/Call nullify instruction mode
Q	Condition code test field	R	Reserved should be set to 0.

To encode AND r1,r2,r3

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

The flag field is clear and there is no condition test.

To encode AND r1,r2,0x13A with long immediate data.

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

The flag field is clear and there is no condition test.

The special register number 62 is used to indicate that long immediate data is used for the second operand field. The instruction word is then followed by the extra word [0000 0000 0000 0000 0000 0001 0011 1010] which is 0x13A.

Short immediate

This is the form used for register with short immediate. Note that the short immediate data is always sign extended to 32 bits before use.

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]					A[5:0]					B[5:0]					C[5:0]					D[8:0]											

I[4:0] Instruction opcode A[5:0] Destination register address

B[5:0] Operand 1 address C[5:0] Operand 2 address

D[8:0] Short immediate data

To encode AND r1, r2, 0x03A with short immediate data.

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

The code in field C[5:0] (instruction bits[14:9]) is 63 for short immediate data without setting flags. If the instruction needed to set flags, then code 61 would be used.

The result of the operation is discarded if the short immediate code is included in the destination field A[5:0].

Single operand

Single operand instructions use the same format as “register” and “short immediate” encoding styles, except that the I-field contains 0x03 and the C-field is used to encode the particular single operand instruction code.

Branch

This form is used for the branch type instructions.

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]					L[21:0]																			N	N	Q	Q	Q	Q	Q	

I[4:0] Instruction opcode L[21:2] Relative address

Q Condition code test field N Jump/Call nullify instruction mode

The Jump/Call nullify instruction modes are shown below.

Mnemonic	Operation	Code
ND	No Delayed instruction slot. Default mode. Only execute next instruction when <i>not</i> jumping	0
D	Delayed instruction slot Always execute next instruction	1
JD	Jump Delayed instruction slot Only execute next instruction when jumping	2
-	Reserved	3

The branch target address is calculated by adding the offset within the instruction to the address of the instruction fetched after the branch. Hence if the relative address was 0 then the target of the branch would be the instruction immediately following the branch. (i.e. the instruction in the delay slot.)

Care should be taken when the instruction in the delay slot is not the immediately following instruction address, in which case the relative address could be encoded incorrectly. With bad coding, this can occur if the branch is the very last instruction in a zero overhead loop or if the branch is itself is executed in the delay slot of another branch, jump or loop.

To encode BRA 8000, jumping 8000 bytes (2000 long words) forward.

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

The condition code for "branch always" instruction is 0. The instruction following this branch would not be executed since 0 is contained in the call/jump nullify field.

Instruction Set Details

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 the instruction opcode.

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

Operation: The operation of the instruction using the following key:

dest:	The target register
operand1:	The first source operand
operand2:	The second source operand
flags:	The flag bits in the status register
PC:	The program counter
rel_addr:	Relative branch address

Syntax: The syntax of the instruction and supported constructs using the key in Table 4.

Example: A simple coding example

Description: Full description of the instruction

Status Flags: The status flags that are affected

Instruction format: Layout of the instruction encoding

Instruction fields: Description of the fields used in the instruction format

ADC

Addition with Carry
Arithmetic Operation

ADC

Operation:

$$\text{dest} \leftarrow \text{operand1} + \text{operand2} + \text{carry}$$

Syntax:

with result

ADC<.cc><.f> a,b,c
ADC<.f> a,b,shimm
ADC<.f> a,shimm,c
ADC<.f> a,shimm,shimm
ADC<.cc><.f> a,b,limm
ADC<.cc><.f> a,limm,c

without result

ADC<.cc><.f> 0,b,c
ADC<.f> 0,b,shimm
ADC<.f> 0,shimm,c
ADC<.f> 0,shimm,shimm
ADC<.cc><.f> 0,b,limm
ADC<.cc><.f> 0,limm,c

Example:

ADC r1, r2, r3

Description:

Add operand1 to operand2 and carry, and place the result in the destination register.

Status flags:

Z	N	C	V
*	*	*	*

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 an overflow is generated

Instruction 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
0	1	0	0	1	A[5:0]					B[5:0]					C[5:0]					F	Res.	Q	Q	Q	Q	Q	Q				

OR

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

Instruction fields:

A[5:0] Destination register address. B[5:0] Operand 1 address
C[5:0] Operand 2 address D[8:0] Immediate data field
Q Condition code field Res Reserved. Should be set to 0.
F Set flags on result if set to 1

ADD

Addition Arithmetic Operation

ADD

Operation:

$\text{dest} \leftarrow \text{operand1} + \text{operand2}$

Syntax:

with result

ADD<.cc><.f> a,b,c
ADD<.f> a,b,shimm
ADD<.f> a,shimm,c
ADD<.f> a,shimm,shimm

ADD<.cc><.f> a,b,limm
ADD<.cc><.f> a,limm,c

Without result

ADD<.cc><.f> 0,b,c
ADD<.f> 0,b,shimm
ADD<.f> 0,shimm,c
ADD<.f> 0,shimm,shimm
(shimms MUST match)
ADD<.cc><.f> 0,b,limm
ADD<.cc><.f> 0,limm,c

Example:

ADD r1, r2, r3

Description:

Add operand1 to operand2 and place the result in the destination register.

Status flags:

Z	N	C	V
*	*	*	*

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 an overflow is generated

Instruction 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
0	1	0	0	0	A[5:0]					B[5:0]					C[5:0]					F	Res.	Q	Q	Q	Q	Q	Q	Q	Q	Q	Q

OR

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

Instruction fields:

A[5:0] Destination register address. B[5:0] Operand 1 address
C[5:0] Operand 2 address D[8:0] Immediate data field
Q Condition code field Res Reserved. Should be set to 0.
F Set flags on result if set to 1

AND

Logical Bitwise AND Logical Operation

AND

Operation:

$\text{dest} \leftarrow \text{operand1 AND operand2}$

Syntax:

with result

AND<.cc><.f> a,b,c
AND<.f> a,b,shimm
AND<.f> a,shimm,c
AND<.f> a,shimm,shimm

AND<.cc><.f> a,b,limm
AND<.cc><.f> a,limm,c

without result

AND<.cc><.f> 0,b,c
AND<.f> 0,b,shimm
AND<.f> 0,shimm,c
AND<.f> 0,shimm,shimm
(shimms MUST match)
AND<.cc><.f> 0,b,limm
AND<.cc><.f> 0,limm,c

Example:

AND r1, r2, r3

Description:

Logical bitwise AND of operand1 with operand2 and place the result in the destination register.

Status flags:

Z	N	C	V
*	*	.	.

Z Set if result is zero N Set if most significant bit of result is set
C Unchanged V Unchanged

Instruction 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
0	1	1	0	0	A[5:0]					B[5:0]					C[5:0]					F	Res.	Q	Q	Q	Q	Q	Q	Q	Q	Q	Q

OR

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

Instruction fields:

A[5:0] Destination register address. B[5:0] Operand 1 address
C[5:0] Operand 2 address D[8:0] Immediate data field
Q Condition code field Res Reserved. Should be set to 0.
F Set flags on result if set to 1

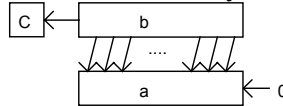
ASL/LSL

Arithmetic Shift Left Logical Operation

ASL/LSL

Operation:

dest \leftarrow arithmetic shift left by one of operand



Syntax:

with result

ASL<.cc><.f> a,b
 ASL<.f> a,shimm
 ASL<.cc><.f> a,limm

without result

ASL<.cc><.f> 0,b
 ASL<.f> 0,shimm
 ASL<.cc><.f> 0,limm

Example:

ASL r1, r2

Description:

Arithmetically shift operand left by one place and place the result in the destination register. When interpreting as an arithmetic shift, the overflow flag will be set if the sign bit changes after the shift. When interpreting as a logical shift, the overflow flag can be ignored. ASL is included for instruction set symmetry. It is basically the ADD instruction. (ADD a,b,b etc)

Status flags:

Z	N	C	V
*	*	*	*

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

Instruction 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
0	1	1	0	0	A[5:0]					B[5:0]					B[5:0]					F	Res.	Q	Q	Q	Q	Q	Q				

OR

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

Instruction fields:

A[5:0] Destination register address. B[5:0] Operand 1 address - in both fields
 D[8:0] Immediate data field Q Condition code field
 Res Reserved. Should be set to 0. F Set flags on result if set to 1

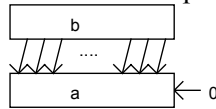
ASL multiple

Multiple Arithmetic shift left
Extension Option

ASL multiple

Operation:

dest \leftarrow arithmetic shift left of operand1 by operand2



Syntax:

with result

ASL<.cc><.f> a,b,c
ASL<.cc><.f> a,b,limm
ASL<.f> a,b,shimm
ASL<.cc><.f> a,limm,c
ASL<.f> a,shimm,c

without result

ASL<.cc><.f> 0,b,c
ASL<.cc><.f> 0,b,limm
ASL<.f> 0,b,shimm
ASL<.cc><.f> 0,limm,c
ASL<.f> 0,shimm,c

Example:

ASL r1,r2,r3

Description:

Arithmetically, shift left operand1 by operand2 places and place the result in the destination register.

Status flags:

Z	N	C	V
*	*	.	.

Z Set if result is zero

N

Set if most significant bit of result is set

C Unchanged

V

Unchanged

Instruction 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
1	0	0	0	0																			F	R	R	R	Q	Q	Q	Q	

OR

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
1	0	0	0	0																											

Instruction fields:

A[5:0] Destination register address
B[5:0] Operand 1 address
C[5:0] Operand 2 address
D[8:0] Immediate data field

Q Condition code field
R Reserved: set to 0
F Set flags on result if 1

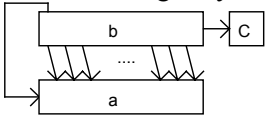
ASR

Arithmetic Shift Right Logical Operation

ASR

Operation:

$\text{dest} \leftarrow \text{arithmetic shift right by one of operand}$



Syntax:

with result

ASR<.cc><.f>

a,b

ASR<.f>

a,shimm

ASR<.cc><.f>

a,limm

Without result

ASR<.cc><.f>

0,b

ASR<.f>

0,shimm

ASR<.cc><.f>

0,limm

Example:

ASR r1,r2

Description:

Arithmetically shift operand right by one place and place the result in the destination register. The sign of the operand is retained after the shift.

Status flags:

Z N C V

*	*	*	.
---	---	---	---

Z Set if result is zero

N

Set if most significant bit of result is set

C Set if carry is generated

V

Unchanged

Instruction 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
0	0	0	1	1	A[5:0]					B[5:0]					0	0	0	0	0	1	F	Res.	Q	Q	Q	Q	Q	Q	Q	Q	

OR

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

Instruction fields:

A[5:0] Destination register address.

B[5:0] Operand address

D[8:0] Immediate data field

Q Condition code field

Res Reserved. Should be set to 0.

F Set flags on result if set to 1

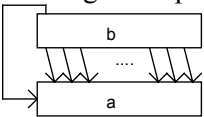
ASR multiple

Arithmetic shift right
Extension Option

ASR multiple

Operation:

dest ← arithmetic shift right of operand1 by operand2



Syntax:

with result

ASR<.cc><.f> a,b,c
ASR<.cc><.f> a,b,limm
ASR<.f> a,b,shimm
ASR<.cc><.f> a,limm,c
ASR<.f> a,shimm,c

without result

ASR<.cc><.f> 0,b,c
ASR<.cc><.f> 0,b,limm
ASR<.f> 0,b,shimm
ASR<.cc><.f> 0,limm,c
ASR<.f> 0,shimm,c

Example:

ASR r1,r2,r3

Description:

Arithmetically, shift right operand1 by operand2 places and place the result in the destination register. The destination is sign filled.

Status flags:

Z	N	C	V
*	*	.	.

Z Set if result is zero N Set if most significant bit of result is set
C Unchanged V Unchanged

Instruction 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
1	0	0	1	0	A[5:0]					B[5:0]					C[5:0]					F	R	R	R	Q	Q	Q	Q	Q			

OR

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
1	0	0	1	0	A[5:0]					B[5:0]					C[5:0]					D[8:0]											

Instruction fields:

A[5:0] Destination register address Q Condition code field
B[5:0] Operand 1 address R Reserved: set to 0
C[5:0] Operand 2 address F Set flags on result if 1
D[8:0] Immediate data field

BIC

Logical bitwise AND with invert

BIC

Logical Operation

Operation:

dest ← operand1 AND NOT operand2

Syntax:

with result		without result	
BIC<.cc><.f>	a,b,c	BIC<.cc><.f>	0,b,c
BIC<.f>	a,b,shimm	BIC<.f>	0,b,shimm
BIC<.f>	a,shimm,c	BIC<.f>	0,shimm,c
BIC<.cc><.f>	a,b,limm	BIC<.cc><.f>	0,b,limm
BIC<.cc><.f>	a,limm,c	BIC<.cc><.f>	0,limm,c

Example:

BIC r1, r2, r3

Description:

Logical bitwise AND of operand1 with the inverse of operand2 and place the result in the destination register.

Status flags:

Z	N	C	V
*	*	.	.

Z	Set if result is zero	N	Set if most significant bit of result is set
C	Unchanged	V	Unchanged

Instruction 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
0	1	1	1	0	A[5:0]					B[5:0]					C[5:0]					F	Res.	Q	Q	Q	Q	Q					

OR

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

Instruction fields:

A[5:0]	Destination register address.	B[5:0]	Operand 1 address
C[5:0]	Operand 2 address	D[8:0]	Immediate data field
Q	Condition code field	Res	Reserved. Should be set to 0.
F	Set flags on result if set to 1		

Bcc

Branch Conditionally Branch Operation

Bcc

Operation:

If condition true then $PC \leftarrow PC + \text{rel_addr}$

Syntax:

B<cc><.dd> rel_addr

Example:

BNE.ND new_code

Description:

If the specified condition is met then program execution is resumed at location $PC + \text{relative displacement (rel_addr)}$, where PC is the address of the instruction in the delay slot. The displacement is a 20 bit signed long word offset. The instruction following the branch is executed according to the nullify instruction mode shown in the following table:

ND	Only execute next instruction when <i>not</i> jumping (<i>Default</i>)		00
D	Always execute next instruction		01
JD	Only execute next instruction when jumping		10

The condition codes that can be used in the condition code field are:

AL, RA	00000	MI, N	00100	VC, NV	01000	LE	01100
EQ, Z	00001	CS, C, LO	00101	GT	01001	HI	01101
NE, NZ	00010	CC, NC, HS	00110	GE	01010	LS	01110
PL, P	00011	VS, V	00111	LT	01011	PNZ	01111

NOTE Condition codes 10000 to 11111 are reserved for extensions.

Status flags:

Not affected.

Instruction 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
0	0	1	0	0	L[21:2]																		N	N	Q	Q	Q	Q	Q	Q	

Instruction fields:

L[21:2] Relative address long word displacement

N Nullify instruction mode

Q Condition code field

BLcc**Branch and Link Conditionally**
Branch Operation**BLcc****Operation:**

If condition true then $PC \leftarrow PC + \text{rel_addr}$.
Return address and flags are written to link register (BLINK)

Syntax:

BL<cc><.dd> rel_addr

Example:

BLNE .ND new_code

Description:

If the specified condition is met, then program execution is resumed at location $PC + \text{relative displacement (rel_addr)}$, where PC is the address of the instruction in the delay slot. The displacement is a 20 bit signed long word offset. The instruction following the branch is executed according to the nullify instruction mode according to the following table:

ND	Only execute next instruction when <i>not</i> jumping (<i>Default</i>)	00
D	Always execute next instruction	01
JD	Only execute next instruction when jumping	10

The return address is stored in the link register BLINK. This address is the whole of the status register and is taken either from the first instruction following the branch (current PC) or the instruction after that (next PC) according to the delay slot execution mode.

The flags stored are those set by the instruction immediately preceding the branch.

Return from the subroutine is accomplished with the jump instruction Jcc.

The condition codes that can be used in the condition code field are:

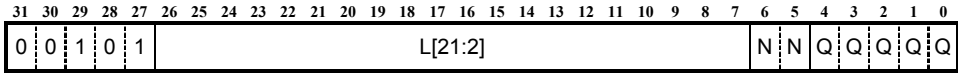
AL, RA	00000	MI, N	00100	VC, NV	01000	LE	01100
EQ, Z	00001	CS, C, LO	00101	GT	01001	HI	01101
NE, NZ	00010	CC, NC, HS	00110	GE	01010	LS	01110
PL, P	00011	VS, V	00111	LT	01011	PNZ	01111

NOTE Condition codes 10000 to 11111 are reserved for extensions.

Status flags:

Not affected.

Instruction format:



Instruction fields:

- L[21:2] Relative address long word displacement
N Nullify instruction mode
Q Condition code field

BRK

Breakpoint Debug Operation

BRK

Operation:

Halt and flush the ARCTangent-A4 processor

Syntax:

BRK

Example:

BRK

Description:

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

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

Interrupts are treated in the same manner by the processor as Branch, and Jump instructions when a BRK instruction is detected. Therefore, an interrupt that reaches stage two of the pipeline when a BRK 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.

Status flags:

Not affected.

Instruction 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
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	

EXT

Zero Extend Arithmetic Operation

EXT

Operation:

$\text{dest} \leftarrow \text{operand zero extended from byte or word}$

Syntax:

with result

EXT<zz><.cc><.f> a,b
EXT<zz><.f> a,shimm
EXT<zz><.cc><.f> a,limm

without result

EXT<zz><.cc><.f> 0,b
EXT<zz><.f> 0,shimm
EXT<zz><.cc><.f> 0,limm

Example:

EXTW r1,r2

Description:

Zero extend operand to most significant bit in long word from byte or word according to size field <zz> and place the result in the destination register. Valid values for <zz> are:

W zero extend from word

B zero extend from byte

Status flags:

Z	N	C	V
*	0	.	.

Z Set if result is zero

N Set if most significant bit of result is set

C Unchanged

V Unchanged

Instruction 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
0	0	0	1	1	A[5:0]					B[5:0]					H[5:0]					F	Res.	Q	Q	Q	Q	Q	Q				

OR

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

Instruction fields:

A[5:0] Destination register address.

B[5:0] Operand 1 address

H[5:0] Operand 2 address

D[8:0] Immediate data field

Q Condition code field

Res Reserved. Should be set to 0.

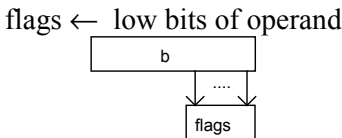
F Set flags on result if set to 1

FLAG

Set Flags Control Operation

FLAG

Operation:



Syntax:

FLAG<.cc> b
 FLAG shimm
 FLAG<.cc> limm

Example:

FLAG r2

Description:

Move the low bits of the operand into the flags register.

Z, N, C, V are replaced by bits [6:3] respectively. The interrupt enables are replaced by bits 2 and 1. The H bit is the processor halt bit and should be set to halt the ARCtangent-A4 processor.

If the H bit is set then the other flag bits are unchanged.

For proper operation, the set flags field should be set to “not set flags”, i.e. bit 8 should be clear, or r63 used for the short-immediate indicator.

Status Flags:

Z	N	C	V	E2	E1	H
*	*	*	*	*	*	*

Z	Set according to bit 6 of operand	N	Set according to bit 5 of operand
C	Set according to bit 4 of operand	V	Set according to bit 3 of operand
E2	Set according to bit 2 of operand	E1	Set according to bit 1 of operand
H	Set according to bit 0 of operand		

Instruction format:

The destination field must contain an immediate operand indicator.

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

OR

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

Instruction fields:

B[5:0]	Operand address.	D[8:0]	Immediate data field
Q	Condition code field	R	Reserved. Should be set to 0.

Jcc

Jump Conditionally

Jcc

Jump Operation

Operation:

If condition true then $PC \leftarrow \text{operand1}$



Syntax:

$J\langle cc \rangle \langle .dd \rangle \langle .f \rangle$ [b]
 $J\langle cc \rangle \langle .JD \rangle \langle .f \rangle$ addr ; *limm* = *addr* (top 7 bits of *addr* will update the flags if flag field is set)
 $J\langle cc \rangle \langle .JD \rangle .f$ addr, flags ; *limm* contains both *flags* (define values as per FLAG) and *addr*

Example:

JNZ.ND [r1]

Description:

If the specified condition is met, then program execution is resumed at location contained in operand 1. If the flag field is set, then operand 1 replaces the whole of the status register (except the halt bit), otherwise if the flag field is clear then only the PC is replaced (the alternative syntax for updating flags is supplied for ease of programming). The operand value updates the status register according to:

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	PC[25:2]																											

If operand 1 is an explicit address (long immediate data), then for this instruction the nullify instruction mode is ignored. Otherwise if operand 1 is a register, the instruction following the jump is executed according to the nullify instruction mode:

ND	Only execute next instruction when <i>not</i> jumping(<i>Default</i>)	00
D	Always execute next instruction	01
JD	Only execute next instruction when jumping	10

The condition codes that can be used in the condition code field are:

AL, RA	00000	MI, N	00100	VC, NV	01000	LE	01100
EQ, Z	00001	CS, C, LO	00101	GT	01001	HI	01101
NE, NZ	00010	CC, NC, HS	00110	GE	01010	LS	01110
PL, P	00011	VS, V	00111	LT	01011	PNZ	01111

NOTE Condition codes 10000 to 11111 are reserved for extensions.

Status flags:

Are changed if flag field is set.

Z	N	C	V	E2	E1	H
*	*	*	*	*	*	.

Z Set according to bit 31 of operand

N Set according to bit 30 of operand

C Set according to bit 29 of operand

V Set according to bit 28 of operand

E2 Set according to bit 27 of operand

E1 Set according to bit 26 of operand

H Unchanged

Instruction 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
0	0	1	1	1	0	0	0	0	0	0	B[5:0]				0	0	0	0	0	0	0	F	R	N	N	Q	Q	Q	Q	Q	

Instruction fields:

B[5:0] Operand address.

Q Condition code field

N Nullify instruction mode

F Set fags if set to 1

R Reserved. Should be set to 0.

JLcc

Jump and Link Conditionally Jump Operation

JLcc

Operation:

If condition true then $PC \leftarrow \text{operand1}$.
Return address and flags are written to link register (BLINK).



Syntax:

JL<cc><.dd><.f> [b]
 JL<cc><.JD><.f> Addr ; *limm* = *addr* (top 7 bits of *addr* will update the flags if flag field is set)
 JL<cc><.JD>.f addr, flags ; *limm* contains both *flags* (define values as per FLAG) and *addr*

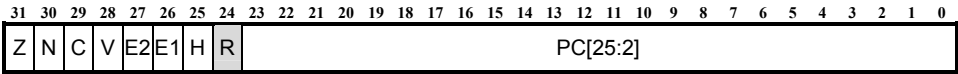
Example:

JLNZ.ND [r1]

Description:

NOTE This instruction is only available for ARCTangent-A4 Basecase processor version 6 and higher.

If the specified condition is met, then program execution is resumed at location contained in operand 1. If the flag field is set then operand 1 replaces the whole of the status register (except the halt bit), otherwise if the flag field is clear, then only the PC is replaced (the alternative syntax for updating flags is supplied for ease of programming). The operand value updates the status register according to the definition:



If operand 1 is an explicit address (long immediate data), then for this instruction the .JD nullify instruction mode *must* be used. If .D or .ND is used, then the link register BLINK will contain the incorrect return address, whereupon, the ARCTangent-A4 processor will attempt to execute the long immediate data on return from the subroutine. When operand 1 is a register, however, the instruction following the jump is executed according to the nullify instruction mode:

ND	Only execute next instruction when <i>not</i> jumping (Default for [b], disallowed for <i>addr</i>)	00
----	--	----

D	Always execute next instruction (<i>Disallowed for addr</i>)	01
JD	Only execute next instruction when jumping (<i>Default for addr</i>)	10

The return address is stored in the link register BLINK. This address is the whole of the status register and is taken either from the first instruction following the jump (current PC) or the instruction after that (next PC) according to the delay slot execution mode. The flags stored are those set by the instruction immediately preceding the jump. Return from the subroutine is accomplished with the jump instruction Jcc.

The condition codes that can be used in the condition code field are:

AL, RA	00000	MI, N	00100	VC, NV	01000	LE	01100
EQ, Z	00001	CS, C, LO	00101	GT	01001	HI	01101
NE, NZ	00010	CC, NC, HS	00110	GE	01010	LS	01110
PL, P	00011	VS, V	00111	LT	01011	PNZ	01111

NOTE Condition codes 10000 to 11111 are reserved for extensions.

Status flags:

Are changed if flag field is set.

Z	N	C	V	E2	E1	H
*	*	*	*	*	*	.

Z Set according to bit 31 of operand
 N Set according to bit 30 of operand
 C Set according to bit 29 of operand
 V Set according to bit 28 of operand

E2 Set according to bit 27 of operand
 E1 Set according to bit 26 of operand
 H Unchanged

Instruction format:

JLcc is encoded as Jcc, except bit 9 is set to '1'

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

Instruction fields:

B[5:0] Operand address. F Set fags if set to 1
 Q Condition code field R Reserved. Should be set to 0.
 N Nullify instruction mode

LD

Delayed load from memory
Memory Operation

LD

Operation:

$\text{dest} \leftarrow \text{contents of address } [\text{operand 1} + \text{operand 2}]$

Syntax:

LD<zz><.x><.a><.di> a,[b]
 LD<zz><.x><.a><.di> a,[b,c]
 LD<zz><.x><.a><.di> a,[b,shimm]
 LD<zz><.x><.a><.di> a,[b,limm]
 LD<zz><.x><.di> a,[limm,c]
 LD<zz><.x><.di> a,[shimm,shimm]
 (shimms MUST match)
 LD<zz><.x><.di> a,[limm]

Example:

LD r1, [r2, r3]

Description:

Add operand1 with operand2, get the data from the calculated address and place it in the destination register. The data size of the load is set according to the size field <zz>. The following table shows the sizes available:

- no field in syntax	Long word	00
W	Word	10
B	Byte	01

When data is loaded, if the size is not a long word the most significant bit of the data can be sign extended to the most significant bit of the long word, with the .X suffix. The result of the address computation can be written back to the first register operand in the address field. This write back occurs when the address write back field, .A, is set. If a data-cache is available in the memory controller the load instruction can bypass the use of that cache when the direct from memory field, .DI, is set.

Note that the destination of a load should not be an immediate data indicator. The operation of the load/store unit may be degraded if this occurs.

When the target of a LD.A instruction is the same register as the one used for address write-back (.A), the returning load will overwrite the value from the address write-back.

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.

Not affected.

Load using generic opcode form:

OR

Instruction fields:

A[5:0]	Destination register address.	Di	Direct to memory (cache bypass) enable
B[5:0]	Operand 1 address	A	Address write-back enable
C[5:0]	Operand 2 address	Z	Size field
D[8:0]	Immediate data offset	X	Sign extend field
R	Reserved. Should be set to 0		

LPcc

Loop Set Up Branch Operation

LPcc**Operation:**

If condition *false* then $PC \leftarrow PC + \text{rel_addr}$.

If condition *true* then $LP_END \leftarrow PC + \text{rel_addr}$ and $LP_START \leftarrow \text{next PC}$.

Syntax:

LP<cc><.dd> rel_addr

Example:

LPNE.ND end_loop1

Description:

If the specified condition is *not* met, then program execution is resumed at location $PC + \text{relative displacement (rel_addr)}$, where PC is the address of the instruction in the delay slot. The displacement is a 20 bit signed long word offset. If the condition is met, then the zero overhead loop registers are set up. The instruction following the loop set up is executed according to the nullify instruction mode according to the following table:

ND	Only execute next instruction when <i>not</i> jumping(<i>Default</i>)	00
D	Always execute next instruction	01
JD	Only execute next instruction when jumping	10

The condition codes that can be used in the condition code field are:

AL, RA	00000	MI, N	00100	VC, NV	01000	LE	01100
EQ, Z	00001	CS, C, LO	00101	GT	01001	HI	01101
NE, NZ	00010	CC, NC, HS	00110	GE	01010	LS	01110
PL, P	00011	VS, V	00111	LT	01011	PNZ	01111

NOTE Condition codes 10000 to 11111 are reserved for extensions.

Status flags:

Not affected.

Instruction 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
0	0	1	1	0	L[21:2]																		N	N	Q	Q	Q	Q	Q	Q	

Instruction fields:

L[21:2] Relative address long word displacement

N Nullify instruction mode

Q Condition code field

LR

Load from auxiliary register Control Operation

LR

Operation:

dest \leftarrow contents of auxiliary register number [operand 1]

Syntax:

LR a,[b]
LR a,[shimm]
LR a,[limm]

Example:

LR r1,[4]

Description:

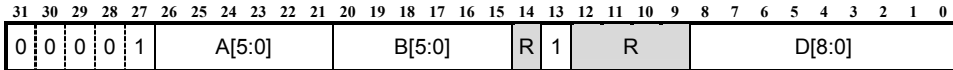
Get the data from the auxiliary register whose number is obtained from operand 1 and place it in the destination register.

Status flags:

Not affected

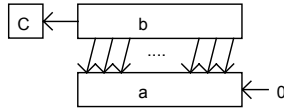
Instruction format:

This is an encoding on the LD instruction.



Instruction fields:

A[5:0] Destination register address. B[5:0] Operand 1 address
D[8:0] Immediate data field R Reserved. Should be set to 0.

LSL**Logical Shift Left
Logical Operation****LSL****Operation:** $\text{dest} \leftarrow \text{logical shift left by one of operand}$ 

See ASL.

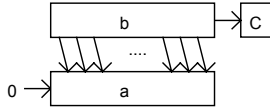
LSR

Logical Shift Right Logical Operation

LSR

Operation:

$\text{dest} \leftarrow \text{logical shift right by one of operand}$



Syntax:

with result

LSR<.cc><.f> a,b
LSR<.f> a,shimm
LSR<.cc><.f> a,limm

without result

LSR<.cc><.f> 0,b
LSR<.f> 0,shimm
LSR<.cc><.f> 0,limm

Example:

LSR r1,r2

Description:

Logically shift operand right by one place and place the result in the destination register.

The most significant bit of the result is replaced with 0.

Status flags:

Z	N	C	V
*	*	*	.

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

Instruction 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
0	0	0	1	1	A[5:0]					B[5:0]					0	0	0	0	1	0	F	Res.	Q	Q	Q	Q	Q	Q	Q	Q	

OR

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

Instruction fields:

A[5:0] Destination register address. B[5:0] Operand address
D[8:0] Immediate data field Q Condition code field
Res Reserved. Should be set to 0. F Set flags on result if set to 1

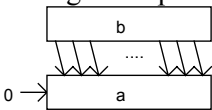
LSR multiple

Logical shift right
Extension Option

LSR mutliple

Operation:

dest ← logical shift right of operand1 by operand2



Syntax:

with result

ASR<.cc><.f> a,b,c
ASR<.cc><.f> a,b,limm
ASR<.f> a,b,shimm
ASR<.cc><.f> a,limm,c
ASR<.f> a,shimm,c

without result

ASR<.cc><.f> 0,b,c
ASR<.cc><.f> 0,b,limm
ASR<.f> 0,b,shimm
ASR<.cc><.f> 0,limm,c
ASR<.f> 0,shimm,c

Example:

LSR r1,r2,r3

Description:

Logical shift right operand1 by operand2 places and place the result in the destination register.

Status flags:

Z	N	C	V
*	*	.	.

Z Set if result is zero

N Set if most significant bit of result is set

C Unchanged

V Unchanged

Instruction 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
1	0	0	0	1	A[5:0]					B[5:0]					C[5:0]					F	R	R	R	Q	Q	Q	Q	Q	Q	Q	Q

OR

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
1	0	0	0	1	A[5:0]					B[5:0]					C[5:0]					D[8:0]											

Instruction fields:

A[5:0] Destination register address

Q Condition code field

B[5:0] Operand 1 address

R Reserved: set to 0

C[5:0] Operand 2 address

F Set flags on result if 1

D[8:0] Immediate data field

MAX

Return Maximum Extension Option

MAX**Operation:**

$$\text{dest} \leftarrow \text{MAX}(\text{operand1}, \text{operand2})$$
Syntax:**with result**

MAX<.cc><.f> a,b,c
 MAX<.f> a,b,shimm
 MAX<.f> a,shimm,c
 MAX<.cc><.f> a,b,limm
 MAX<.cc><.f> a,limm,c

without result

MAX<.cc><.f> 0,b,c
 MAX<.f> 0,b,shimm
 MAX<.f> 0,shimm,c
 MAX<.cc><.f> 0,b,limm
 MAX<.cc><.f> 0,limm,c

Example:

$$\text{MAX } r1, r2, r3$$
Description:

Return the maximum of the two operands and place the result in the destination register. Note, both of the compared numbers are signed.

Status flags:

Z	N	C	V
*	*	*	*

Z Set if both source operands are equal (equivalent to a SUB instruction)
 N Set as the MSB of subtraction result (equivalent to a SUB instruction)
 C Set if the second source operand is selected (src2 >=src1)
 V Set if the subtraction overflows (equivalent to a SUB instruction)

Instruction 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
1	1	1	1	0	A[5:0]					B[5:0]					C[5:0]					F	R	R	R	Q	Q	Q	Q	Q	Q	Q	Q

OR

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
1	1	1	1	0	A[5:0]					B[5:0]					C[5:0]					D[8:0]											

Instruction fields:

A[5:0]	Destination register address	Q	Condition code field
B[5:0]	Operand 1 address	R	Reserved: set to 0
C[5:0]	Operand 2 address	F	Set flags on result if 1
D[8:0]	Immediate data field		

MIN

Return minimum value
Extension Option

MIN

Operation:

dest ← MIN (operand1, operand2)

Syntax:

with result

MIN<.cc><.f> a,b,c
MIN<.f> a,b,shimm
MIN<.f> a,shimm,c
MIN<.cc><.f> a,b,limm
MIN<.cc><.f> a,limm,c

without result

MIN<.cc><.f> 0,b,c
MIN<.f> 0,b,shimm
MIN<.f> 0,shimm,c
MIN<.cc><.f> 0,b,limm
MIN<.cc><.f> 0,limm,c

Example:

MIN r1,r2,r3

Description:

Return the minimum of the two operands and place the result in the destination register. Note, both of the compared numbers are signed.

Condition codes:

Z	N	C	V
*	*	*	*

Z Set if both source operands are equal (equivalent to a SUB instruction)
N Set as the MSB of subtraction result (equivalent to a SUB instruction)
C Set if the second source operand is selected (src2 <=src1)
V Set if the subtraction overflows (equivalent to a SUB instruction)

Instruction 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
1	1	1	1	1	A[5:0]					B[5:0]					C[5:0]					F	R	R	R	Q	Q	Q	Q	Q	Q	Q	Q

OR

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
1	1	1	1	1	A[5:0]					B[5:0]					C[5:0]					D[8:0]											

Instruction fields:

A[5:0] Destination register address
B[5:0] Operand 1 address
C[5:0] Operand 2 address
D[8:0] Immediate data field
Q Condition code field
R Reserved: set to 0
F Set flags on result if 1

MOV

Move contents
Arithmetic Operation

MOV

Operation:

$\text{dest} \leftarrow \text{operand}$

Syntax:

with result

MOV<.cc><>.f> a,b

MOV<.f> a,shimm

MOV<.cc><>.f> a,limm

without result

MOV<.cc><>.f> 0,b

MOV<.f> 0,shimm

MOV<.cc><>.f> 0,limm

Example:

MOV r1,r2

Description:

The contents of the operand are moved to the destination register

Status flags:

Z N C V

*	*	.	.
---	---	---	---

Z Set if result is zero

N

Set if most significant bit of result is set

C Unchanged

V

Unchanged

Instruction format:

MOV is included for instruction set symmetry. It is basically the AND instruction.

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

OR

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

Instruction fields:

A[5:0] Destination register address.

B[5:0] Operand 1 address

D[8:0] Immediate data field

Q Condition code field

Res Reserved. Should be set to 0.

F Set flags on result if set to 1

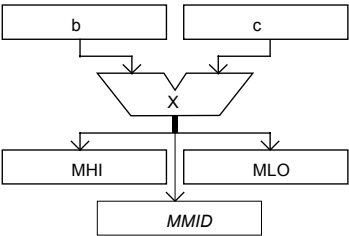
MUL64

32 x 32 Multiply Extension Option

MUL64

Operation:

MLO ← low part of (operand 1 X operand 2)
 MHI ← high part of (operand 1 X operand 2)
 MMID ← middle part of (operand 1 X operand 2)



Syntax:

MUL64<.cc> <0,>b,c
 MUL64 <0,>b,shimm
 MUL64 <0,>shimm,c
 MUL64<.cc> <0,>b,limm
 MUL64<.cc> <0,>limm,c

Example:

MUL64 r2,r3

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.

Status flags:

Not affected.

Instruction 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
1	0	1	0	0	1	1	1	1	1	1	B[5:0]					C[5:0]					0	R	R	R	Q	Q	Q	Q	Q		

OR

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
1	0	1	0	0	1	1	1	1	1	1	B[5:0]						C[5:0]						D[8:0]								

Instruction fields:

B[5:0]	Operand 1 address	Q	Condition code field
C[5:0]	Operand 2 address	R	Reserved: set to 0
D[8:0]	Immediate data field		

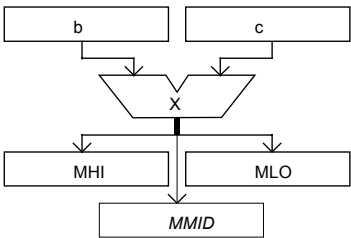
MULU64

32 x 32 Unsigned Multiply Extension Option

MULU64

Operation:

MLO ← low part of (operand 1 X operand 2)
 MHI ← high part of (operand 1 X operand 2)
 MMID ← middle part of (operand 1 X operand 2)



Syntax:

MULU64<.cc> <0,>b,c
 MULU64 <0,>b,shimm
 MULU64 <0,>shimm,c
 MULU64<.cc> <0,>b,limm
 MULU64<.cc> <0,>limm,c

Example:

MULU64 r2,r3

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.

Status flags:

Not affected.

Instruction 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
1	0	1	0	1	1	1	1	1	1	1	B[5:0]					C[5:0]					0	R	R	R	Q	Q	Q	Q			

OR

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			
1	0	1	0	1	1	1	1	1	1	1	B[5:0]											C[5:0]					D[8:0]							

Instruction fields:

B[5:0]	Operand 1 address	Q	Condition code field
C[5:0]	Operand 2 address	R	Reserved: set to 0
D[8:0]	Immediate data field		

NOP

No Operation
Control Operation

NOP

Operation:
No Operation

Syntax:
NOP

Example:
NOP

Description:
No operation. The state of the processor is not changed. NOP is included for instruction set symmetry. It is basically the XOR instruction:

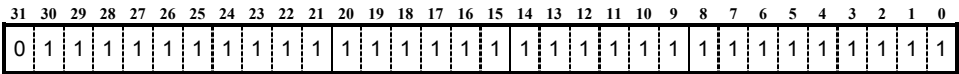
```
XOR 0x1FF, 0x1FF, 0x1FF.
```

Status flags:

Z	N	C	V
.	.	.	.

Z Unchanged N Unchanged
C Unchanged V Unchanged

Instruction format:



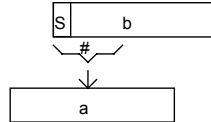
NORM

Normalize Integer Extension Option

NORM

Operation:

dest \leftarrow normalize value of operand



Syntax:

with result

NORM<.cc><.f> a,b
 NORM<.f> a,shimm
 NORM<.cc><.f> a,limm

without result

NORM<.cc><.f> 0,b
 NORM<.f> 0,shimm
 NORM<.cc><.f> 0,limm

Example:

NORM r1,r2

Description:

Gives the normalization integer for the signed value in the operand. The normalisation integer is the amount by which the operand should be shifted left to normalise it as a 32-bit signed integer. This function is sometimes referred to as “find first bit”. Examples of returned values are shown in the table below:

Operand Value	Returned Value	Notes
0x00000000	0x0000001F	
0x1FFFFFFF	0x00000002	
0x3FFFFFFF	0x00000001	
0x7FFFFFFF	0x00000000	
0x80000000	0x00000000	<i>This result is not particularly useful since the operand is the most negative value.</i>
0xC0000000	0x00000001	
0xE0000000	0x00000002	
0xFFFFFFFF	0x0000001F	

Status flags:

Z	N	C	V
*	*	.	.

Z Set if result is zero N Set if most significant bit of result is set
C Unchanged V Unchanged

Instruction 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
0	0	0	1	1	A[5:0]						B[5:0]						0	0	1	0	1	0	F	R	R	R	Q	Q	Q	Q	

OR

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

Instruction fields:

A[5:0] Destination register address Q Condition code field
B[5:0] Operand 1 address R Reserved: set to 0
D[8:0] Immediate data field F Set flags on result if 1

OR

Logical Bitwise OR Logical Operation

OR

Operation:

$\text{dest} \leftarrow \text{operand1 OR operand2}$

Syntax:

with result

OR<cc><.f> a,b,c
OR<.f> a,b,shimm
OR<.f> a,shimm,c
OR<.f> a,shimm,shimm

OR<cc><.f> a,b,limm
OR<cc><.f> a,limm,c

without result

OR<cc><.f> 0,b,c
OR<.f> 0,b,shimm
OR<.f> 0,shimm,c
OR<.f> 0,shimm,shimm
 (shimms MUST match)
OR<cc><.f> 0,b,limm
OR<cc><.f> 0,limm,c

Example:

OR r1, r2, r3

Description:

Logical bitwise OR of operand1 with operand2 and place the result in the destination register.

Status flags:

Z	N	C	V
*	*	.	.

Z Set if result is zero N Set if most significant bit of result is set
C Unchanged V Unchanged

Instruction 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
0	1	1	0	1	A[5:0]						B[5:0]					C[5:0]					F	Res.	Q	Q	Q	Q	Q				

OR

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

Instruction fields:

A[5:0] Destination register address. B[5:0] Operand 1 address
C[5:0] Operand 2 address D[8:0] Immediate data field
Q Condition code field F Set flags on result if set to 1
Res Reserved. Should be set to 0.

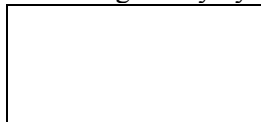
RLC

Rotate Left Through Carry Logical Operation

RLC

Operation:

dest ← rotate left through carry by one of operand



Syntax:

with result

RLC<.cc><.f> a,b
RLC<.f> a,shimm
RLC<.cc><.f> a,limm

without result

RLC<.cc><.f> 0,b
RLC<.f> 0,shimm
RLC<.cc><.f> 0,limm

Example:

RLC r1, r2

Description:

Rotate operand left by one place and place the result in the destination register.

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. RLC is included for instruction set symmetry. It is basically the ADC instruction.

Status flags:

Z N C V

*	*	*	.
---	---	---	---

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

Instruction 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
0	1	0	0	1	A[5:0]					B[5:0]					B[5:0]					F	Res.	Q	Q	Q	Q	Q	Q	Q	Q	Q	Q

OR

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

Instruction fields:

A[5:0] Destination register address. B[5:0] Operand 1 address - in both fields
D[8:0] Immediate data field Q Condition code field
F Set flags on result if set to 1 Res Reserved. Should be set to 0.

ROL

Rotate Left
Not implemented

ROL

Operation:

$\text{dest} \leftarrow \text{rotate left by one of operand}$



The instruction is listed for instruction set symmetry.

To carry out this instruction in the basecase version of ARCTangent-A4 processor, it is recommended that the following 2 instructions are used.

ADD.f a,b,b

ADC<.f> a,a,0

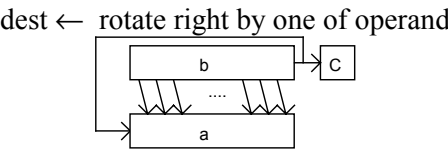
The flags are set by the first instruction, hence ROL cannot be used without affecting the flags.

ROR

Rotate Right
Logical Operation

ROR

Operation:



Syntax:

with result		Without result	
ROR<.cc><.f>	a,b	ROR<.cc><.f>	0,b
ROR<.f>	a,shimm	ROR<.f>	0,shimm
ROR<.cc><.f>	a,limm	ROR<.cc><.f>	0,limm

Example:

ROR r1,r2

Description:

Rotate operand right by one place and place the result in the destination register. The least significant bit of the source is also copied to carry flag.

Status flags:

Z	N	C	V
*	*	*	.

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

Instruction 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
0	0	0	1	1	A[5:0]					B[5:0]					0	0	0	0	1	1	F	Res.		Q	Q	Q	Q	Q	Q		

OR

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

Instruction fields:

A[5:0]	Destination register address.	B[5:0]	Operand address
D[8:0]	Immediate data field	Q	Condition code field
Res	Reserved. Should be set to 0.	F	Set flags on result if set to 1

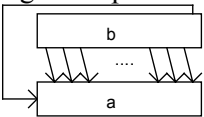
ROR multiple

Rotate right
Extension Option

ROR multiple

Operation:

dest ← rotate right of operand1 by operand2



Syntax:

with result

ROR<.cc><.f> a,b,c
ROR<.cc><.f> a,b,limm
ROR<.f> a,b,shimm
ROR<.cc><.f> a,limm,c
ROR<.f> a,shimm,c

without result

ROR<.cc><.f> 0,b,c
ROR<.cc><.f> 0,b,limm
ROR<.f> 0,b,shimm
ROR<.cc><.f> 0,limm,c
ROR<.f> 0,shimm,c

Example:

ROR r1,r2,r3

Description:

Rotate right operand1 by operand2 places and place the result in the destination register.

Condition codes:

Z	N	C	V
*	*	.	.

Z Set if result is zero

N

Set if most significant bit of result is set

C Unchanged

V

Unchanged

Instruction 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
1	0	0	1	1	A[5:0]					B[5:0]					C[5:0]					F	R	R	R	Q	Q	Q	Q	Q	Q	Q	

OR

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
1	0	0	1	1	A[5:0]					B[5:0]					C[5:0]					D[8:0]											

Instruction fields:

A[5:0] Destination register address

Q Condition code field

B[5:0] Operand 1 address

R Reserved: set to 0

C[5:0] Operand 2 address

F Set flags on result if 1

D[8:0] Immediate data field

RRC

Rotate Right through Carry Logical Operation

RRC

Operation:

dest ← rotate right through carry by one of operand



Syntax:

with result

RRC<.cc><.f> a,b
RRC<.f> a,shimm
RRC<.cc><.f> a,limm

Without result

RRC<.cc><.f> 0,b
RRC<.f> 0,shimm
RRC<.cc><.f> 0,limm

Example:

RRC r1, r2

Description:

Rotate operand right by one place and place the result in the destination register.

The carry flag is shifted into the most significant bit of the result, and the least significant bit of the source is placed in the carry flag.

Status flags:

Z	N	C	V
*	*	*	.

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

Instruction 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
0	0	0	1	1	A[5:0]						B[5:0]						0	0	0	1	0	0	F	Res.	Q	Q	Q	Q	Q		

OR

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

Instruction fields:

A[5:0] Destination register address. B[5:0] Operand address
D[8:0] Immediate data field Q Condition code field
Res Reserved. Should be set to 0. F Set flags on result if set to 1

SBC

Subtract with Carry Arithmetic Operation

SBC

Operation:

$$\text{dest} \leftarrow \text{operand1} - \text{operand2} - \text{/carry}$$

Syntax:

with result

SBC<.cc><.f> a,b,c
SBC<.f> a,b,shimm
SBC<.f> a,shimm,c
SBC<.f> a,shimm,shimm
SBC<.cc><.f> a,b,limm
SBC<.cc><.f> a,limm,c

without result

SBC<.cc><.f> 0,b,c
SBC<.f> 0,b,shimm
SBC<.f> 0,shimm,c
SBC<.f> 0,shimm,shimm
SBC<.cc><.f> 0,b,limm
SBC<.cc><.f> 0,limm,c

Example:

SBC r1, r2, r3

Description:

Subtract operand2 from operand1 with carry, and place the result in the destination register. Operand2 is subtracted from operand1 and if carry has previously been set, the result is decremented by one.

The carry flag is interpreted as a “borrow” for the subtract instruction.

Status flags:

Z	N	C	V
*	*	*	*

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

Instruction 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
0	1	0	1	1	A[5:0]					B[5:0]					C[5:0]					F	Res.	Q	Q	Q	Q	Q					

OR

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

Instruction fields:

A[5:0] Destination register address. B[5:0] Operand 1 address
C[5:0] Operand 2 address D[8:0] Immediate data field
Q Condition code field R Reserved. Should be set to 0.
F Set flags on result if set to 1

SEX

Sign Extend Arithmetic Operation

SEX

Operation:

$$\text{dest} \leftarrow \text{operand sign extended from byte or word}$$

Syntax:

with result

SEX<zz><.cc><.f> a,b
 SEX<zz><.f> a,shimm
 SEX<zz><.cc><.f> a,limm

without result

SEX<zz><.cc><.f> 0,b
 SEX<zz><.f> 0,shimm
 SEX<zz><.cc><.f> 0,limm
 SEX 0,shimm ;nop

Example:

SEXW r1, r2

Description:

Sign extend operand to most significant bit in long word from byte or word according to size field <zz> and place the result in the destination register. Valid values for <zz> are:

W sign extend from word

B sign extend from byte

Status flags:

Z	N	C	V
*	*	.	.

Z Set if result is zero

N Set if most significant bit of result is set

C Unchanged

V Unchanged

Instruction 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
0	0	0	1	1	A[5:0]					B[5:0]					H[5:0]					F	Res.	Q	Q	Q	Q	Q					

OR

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

Instruction fields:

A[5:0] Destination register address.

B[5:0] Operand address

H[5:0] Extend size. 05=byte, 06=word.

D[8:0] Immediate data field

Q Condition code field R

Reserved. Should be set to 0.

F Set flags on result if set to 1

SLEEP

Enter Sleep Mode
Control Operation

SLEEP

Operation:

Enter Processor Sleep Mode

Syntax:

SLEEP

Example:

SLEEP

Description:

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

When in sleep mode, the sleep mode flag (ZZ) is set and the pipeline is stalled, but not halted. The host interface operates as normal allowing access to the DEBUG and the STATUS registers and it can halt the processor. The host cannot clear the sleep mode flag, but it can wake the ARCTangent-A4 processor by halting then restarting it. The program counter PC points to the next instruction in sequence after the sleep instruction.

The ARCTangent-A4 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.

Status flags:

Not affected.

Instruction 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
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	

SR

Store to auxiliary register Control Operation

SR

Operation:

auxiliary register number[operand 2] ← operand 1

Syntax:

SR c,[b]
 SR c,[limm]
 SR c,[shimm]
 SR limm,[shimm]
 SR shimm,[limm]
 SR shimm,[b]
 SR limm,[b]

NOTE The operand syntax matches LR.

Example:

SR r1, [12]

Description:

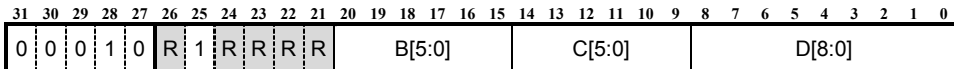
Store operand 1 to the auxiliary register whose number is obtained from operand 2.

Status flags:

Not affected

Instruction format:

This is an encoding on the ST instruction.



Instruction fields:

B[5:0] Operand 2 register address C[5:0] Operand 1 register address
 D[8:0] Immediate data field R Reserved. Should be set to 0.

ST

Store to memory Memory Operation

ST

Operation:

[operand 2 + offset] ← operand 1

Syntax:

ST<zz><.a><.di>	c,[b]	
ST<zz><.di>	c,[limm]	
ST<zz><.a><.di>	c,[b,shimm]	
ST<zz><.di>	c,[shimm,shimm]	<i>shimms MUST match</i>
ST<zz><.a><.di>	0,[b]	
ST<zz><.di>	shimm,[limm]	<i>actually: shimm,[limm,shimm]</i>
ST<zz><.a><.di>	shimm,[b,shimm]	<i>shimms MUST match</i>
ST<zz><.di>	limm,[shimm,shimm]	<i>shimms MUST match</i>
ST<zz><.a><.di>	limm,[b,shimm]	

NOTE The operand syntax matches LD.

Example:

ST.A r1,[r2,10]

Description:

Store operand 1 to the address calculated by adding operand 2 with offset.

NOTE If the offset is not required, the value encoded for the immediate offset will be set to 0.

The data size of the load is set according to the size field <zz>. The following table shows the sizes available.

- no field in syntax	Long word	00
W	Word	10
B	Byte	01

The result of the address computation can be written back to the first register operand in the address field. This write back occurs when the address write back field, .A, is set.

If a data-cache is available in the memory controller the store instruction can bypass the use of that cache when the direct to memory field, .DI, is set.

NOTE Note that 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.

Status flags:

Not affected.

Instruction 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
0	0	0	1	0	Di	0	A	Z	Z	R	B[5:0]					C[5:0]					D[8:0]										

Instruction fields:

B[5:0] Operand 2 register address C[5:0] Operand 1 register address
 D[8:0] Immediate data offset Di Direct to memory (cache bypass) enable
 A Address write-back enable R Reserved. Should be set to 0
 Z Size field

Encoding examples:

ST	r5, [r7, 50]	; ST c, [b, shimm]
I[4:0]	B[5:0]	C[5:0] D[8:0]
2=ST	7	5 50

ST	50, [12345678]	; ST shimm, [limm]
I[4:0]	B[5:0]	C[5:0] D[8:0] LIMM
2=ST	62=limm	63=shimm 50 12345678-50

ST	50, [r7, 50]	; ST shimm, [b, shimm]
I[4:0]	B[5:0]	C[5:0] D[8:0]
2=ST	7	63=shimm 50

ST	r3, [12345678]	; ST c, [limm]
I[4:0]	B[5:0]	C[5:0] D[8:0] LIMM
2=ST	62=limm	3 0 12345678

ST	12345678, [r20, 8]	; ST limm, [b, shimm]
I[4:0]	B[5:0]	C[5:0] D[8:0] LIMM
2=ST	20	62=limm 8 12345678

ST	50, [50, 50]	; ST shimm, [shimm, shimm]
I[4:0]	B[5:0]	C[5:0] D[8:0]
2=ST	62=shimm	62=shimm 50

SUB

Subtract Arithmetic Operation

SUB

Operation:

$\text{dest} \leftarrow \text{operand1} - \text{operand2}$

Syntax:

with result

SUB<.cc><.f> a,b,c
SUB<.f> a,b,shimm
SUB<.f> a,shimm,c
SUB<.f> a,shimm,shimm
SUB<.cc><.f> a,b,limm
SUB<.cc><.f> a,limm,c

without result

SUB<.cc><.f> 0,b,c
SUB<.f> 0,b,shimm
SUB<.f> 0,shimm,c
SUB<.f> 0,shimm,shimm
SUB<.cc><.f> 0,b,limm
SUB<.cc><.f> 0,limm,c

Example:

```
SUB r1,r2,r3
SUB.F 0,r3,200      ; compare r3 with 200 and set flags
SUB.LT r2,r2,r2      ; same effect as MOV.LT r2,0 but no
                     ; limm 0 data needed
```

Description:

Subtract operand2 from operand1 and place the result in the destination register.

The carry flag if set by the subtract instruction is interpreted as a “borrow”.

Status flags:

Z	N	C	V
*	*	*	*

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

Instruction 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
0	1	0	1	0	A[5:0]					B[5:0]					C[5:0]					F	Res.	Q	Q	Q	Q	Q					

OR

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

Instruction fields:

A[5:0] Destination register address. B[5:0] Operand 1 address
C[5:0] Operand 2 address D[8:0] Immediate data field
Q Condition code field Res Reserved
F Set flags on result if set to 1

SWAP

Swap words

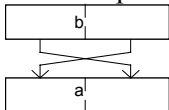
0x03 / 0x09

Extension: Swap instruction

SWAP

Operation:

dest ← word swap of operand



Syntax:

with result

SWAP<.cc><.f> a,b

SWAP<.f> a,shimm

SWAP<.cc><.f> a,limm

without result

SWAP<.cc><.f> 0,b

SWAP<.f> 0,shimm

SWAP<.cc><.f> 0,limm

Example:

SWAP r1,r2

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.

Condition codes:

Z	N	C	V
*	*	.	.

Z Set if result is zero

N Set if most significant bit of result is set

C Unchanged

V Unchanged

Instruction 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
0	0	0	0	1	1	A[5:0]						B[5:0]						0	0	1	0	0	1	F	R	R	R	Q	Q	Q	Q

OR

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

Instruction fields:

A[5:0] Destination register address

Q Condition code field

B[5:0] Operand 1 address

R Reserved: set to 0

D[8:0] Immediate data field

F Set flags on result if 1

SWI

Software Interrupt Control Operation

SWI

Operation:

$\text{instruction_error} \leftarrow '1'$

Syntax:

SWI

Example:

SWI

Description:

The software interrupt (SWI) instruction can be placed anywhere in the program, even in the delay slot of a branch instruction. The software interrupt instruction is decoded in stage two of the pipeline and if executed, then it immediately raises the instruction error exception. The instruction error exception will be serviced using the normal interrupt system. ILINK2 is used at the return address in the service routine.

Once an instruction error exception is taken, then the medium and low priority interrupts are masked off so that ILINK2 register can not be updated again as a result of an interrupt thus preserving the return address of the instruction error exception.

NOTE Only the reset and memory error exceptions have higher priorities than the instruction error exception.

Status flags:

Not affected.

Instruction 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
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1	0

XOR

Logical Bitwise Exclusive OR Logical Operation

XOR

Operation:

$$\text{dest} \leftarrow \text{operand1 XOR operand2}$$

Syntax:

with result

XOR<.cc><.f> a,b,c
 XOR<.f> a,b,shimm
 XOR<.f> a,shimm,c
 XOR<.cc><.f> a,b,limm
 XOR<.cc><.f> a,limm,c

without result

XOR<.cc><.f> 0,b,c
 XOR<.f> 0,b,shimm
 XOR<.f> 0,shimm,c
 XOR<.cc><.f> 0,b,limm
 XOR<.cc><.f> 0,limm,c

Example:

XOR r1,r2,r3

Description:

Logical bitwise Exclusive-OR of operand1 with operand2 and place the result in the destination register.

Status flags:

Z	N	C	V
*	*	.	.

Z Set if result is zero

N Set if most significant bit of result is set

C Unchanged

V Unchanged

Instruction 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
0	1	1	1	1	A[5:0]					B[5:0]					C[5:0]					F	Res.	Q	Q	Q	Q	Q					

OR

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

Instruction fields:

A[5:0] Destination register address.

B[5:0] Operand 1 address

C[5:0] Operand 2 address

D[8:0] Immediate data field

Q Condition code field

Res Reserved. Should be set to 0

F Set flags on result if set to 1

Chapter 9 — The Host

The ARCTangent-A4 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 ARCTangent-A4 processor appear to the host is host interface dependent. An outline of processor control techniques 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 ARCTangent-A4 system may have extensions or customizations in this area, please see associated documentation.

It is expected that the registers and the program memory of the ARCTangent-A4 processor will appear as a memory mapped section to the host. For example, Figure 21 shows two examples: a) a contiguous part of host memory and b) a section of memory and a section of I/O space.

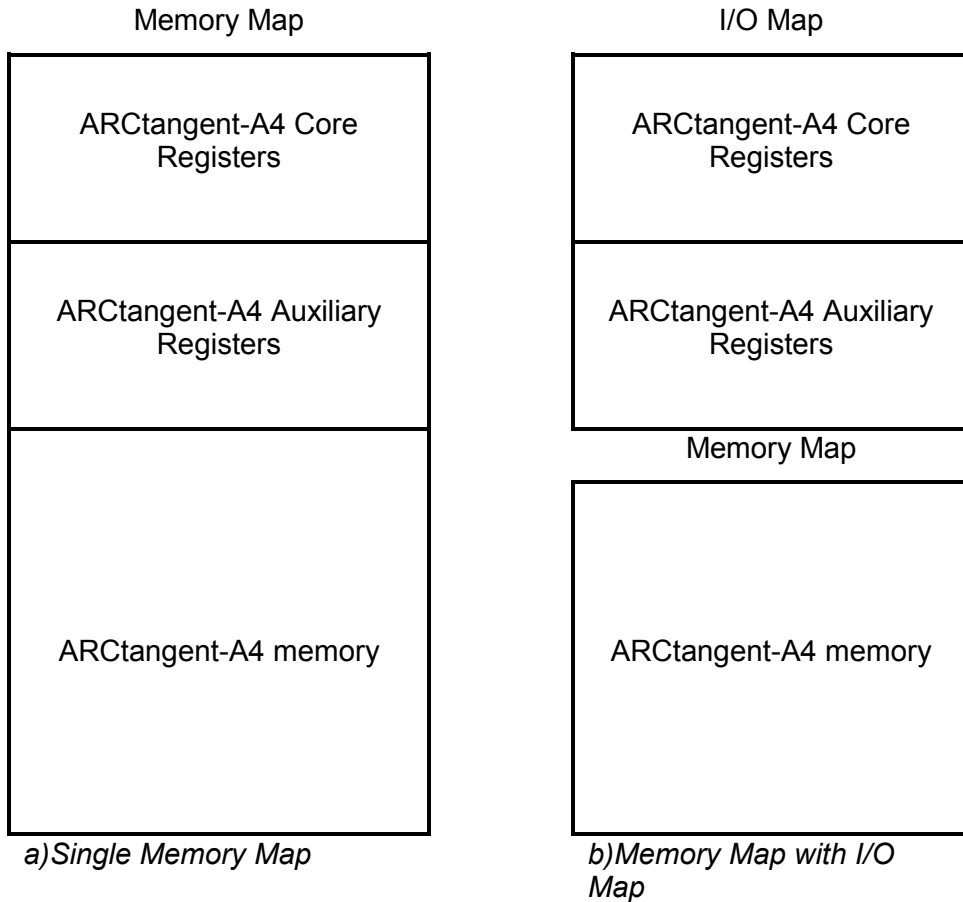


Figure 21 Example Host Memory Maps

Once a reset has occurred, the ARctangent-A4 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 processor, depending on whether the ARctangent-A4 processor is running or halted as shown in Table 18.

	Running	Halted
Memory	Read/Write	Read/Write
Auxiliary Registers	Mainly No access	Read/Write
Core Registers	No access	Read/Write

Table 18 Host Accesses to the ARctangent-A4 processor

Halting

The ARCTangent-A4 processor can halt itself with the FLAG instruction or it can be halted by the host. The host halts the ARCTangent-A4 processor by setting the H bit in the STATUS register, or for basecase version numbers greater than 5 by setting the FH bit in the DEBUG register. See Figure 14 and Figure 19.

NOTE Note that when the ARCTangent-A4 processor is running that only the H bit will change if the host writes to the STATUS register. However, if the ARCTangent-A4 processor had halted itself, the whole of the STATUS register will be updated when the host writes to the STATUS register.

The consequence of this is that the host may assume that the ARCTangent-A4 processor is running by previously reading the STATUS register. By the time that the host forces a halt, the ARCTangent-A4 processor may have halted itself. Therefore, the write of a “halt” number to the STATUS register, say 0x02000000, would overwrite any program counter information that the host required.

In order to force the ARCTangent-A4 processor to halt without overwriting the program counter, Basecase versions greater than 5 have the additional FH bit in the DEBUG register. See Figure 19. The host can test whether the ARCTangent-A4 processor has halted by checking the state of the H bit in the STATUS register. Additionally, the SH bit in the debug register is available to test whether the halt was caused by the host, the ARCTangent-A4 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 processor.

Starting

The host starts the ARCTangent-A4 processor by clearing the H bit in the STATUS register. It is advisable that the host clears any instructions in the pipeline before modifying any registers and re-starting the ARCTangent-A4 processor, by sending NOP instructions through, so that any pending instructions that are about to modify any registers in the processor are allowed to complete. If the ARCTangent-A4 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 PC/Status register
- any extension logic should be reset to its default state

If the ARCTangent-A4 processor has been running and is to be restarted to CONTINUE where it left off, then the procedure is as follows:

- host reads the PC in the STATUS Register
- host writes back to the STATUS register with *the same* PC value as was just read, but clearing the H bit
- the ARCTangent-A4 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 STATUS register has been written and thus the execution starts up again as if there has been no interruption).

Pipecleaning

If the processor is halted whilst it is executing a program, it is possible that the later stages of the pipeline may contain valid instructions. Before re-starting the processor at a new address, these instructions must be cleared to prevent unwanted register writes or jumps from taking place.

If the processor is to be restarted from the point at which it was stopped, then the instructions in the pipeline are to be executed, hence pipecleaning should not be performed.

Pipecleaning is not necessary at times when the pipeline is known to be clean - e.g. immediately after a reset, or if the processor has been stopped by a FLAG instruction followed by three NOPs.

Pipecleaning is achieved as follows:

1. Stop the ARCTangent-A4 processor
2. Download a ‘NOP’ instruction into memory.
3. Invalidate instruction cache to ensure that the NOP is loaded from memory
4. Point the PC/Status register to the downloaded NOP

5. Single step until the values in the program counter or loop count register change.
6. Point the PC/Status register to the downloaded NOP
7. Single step until the values in the program counter or loop count register change.
8. Point the PC/Status register to the downloaded NOP
9. 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 Stepping

The Single Step function is controlled by two bits in the DEBUG register. These bits can be set by the debugger to enable the Single Cycle Stepping or Single Instruction Stepping. The two bits, Single Step (SS) and Instruction Step (IS), are write-only by the host and keep their values for one cycle (see Table 19).

Field	Description	Access Type
SS	Single Step:- Cycle Step enable	Write only from the host
IS	Instruction Step:- Instruction Step enable	Write only from the host

Table 19 Single Step Flags in Debug Register

Single cycle step

The Single Cycle Step function enables the processor for one cycle only. Normally, an instruction is completed in four cycles: fetch, register read, execute and register writeback. In order to complete an instruction, the debugger must repeatedly single cycle step the processor until the program counter value is

updated. Single Cycle Stepping is the only stepping function supported in ARCTangent-A4 basecase processor prior to version 7.

The Single Cycle Step function is enabled by setting the (SS) bit and clearing the (IS) bit in the DEBUG register when the ARCTangent-A4 processor is halted. On the next clock cycle the processor will be enabled for one clock cycle. The single step lasts only for one clock cycle after which the processor is halted.

Single instruction step

The Single Instruction Step function enables the processor for completion of a whole instruction. Single Instruction Stepping is supported on basecase ARCTangent-A4 processor version 7 or above. The Single Instruction Step function is enabled by setting both the (SS) and (IS) bits in the debug register when the ARCTangent-A4 processor is halted.

On the next clock cycle the processor is kept enabled for as many cycle as required to complete the instruction. Therefore, any stalls due to register conflicts or delayed loads are accounted for when waiting for an instruction to complete. All earlier instructions in the pipeline are flushed, the instruction that the program counter is pointing to is completed, the next instruction is fetched and the program counter is incremented.

NOTE 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.

SLEEP instruction in single step mode

The SLEEP instruction is treated as a NOP instruction when the processor is in Single Step Mode. This is because every single step acts as a restart or a wake up call. Consequently, the SLEEP instruction behaves exactly like a NOP propagating through the pipeline.

BRK instruction in single step mode

The BRK instruction behaves exactly as when the processor is not in the Single Step Mode. The BRK instruction is detected and kept in stage one forever until removed by the host.

Software Breakpoints

As long as the host has access to the ARCTangent-A4 code memory, it can replace any ARCTangent-A4 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 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. (See Pipeline Cycle Diagrams for: Loops and Branches).

For ARCTangent-A4 basecase processor versions 7 and higher, the BRK instruction can be used to insert a software breakpoint. BRK will halt the ARCTangent-A4 processor and flush all previous instructions through the pipe. The host can read the STATUS register to determine where the breakpoint occurred.

ARCTangent-A4 Core Registers

The ARCTangent-A4 core registers are available to be read and written by the host. These registers should be accessed by the host once the ARCTangent-A4 processor has halted.

ARCTangent-A4 Auxiliary Registers

Some auxiliary registers, unlike the core registers, may be accessed while the ARCTangent-A4 processor is running. These dual access registers in the basecase processor are:

STATUS

The host can read the status register when the ARCTangent-A4 processor is running. This is useful for code profiling. See Figure 14.

SEMAPHORE

The semaphore register is used for inter-processor and host-ARCTangent-A4 communications. Protocols for using shared memory and provision of mutual exclusion can be accomplished with this register. See Figure 15.

IDENTITY

The host can determine the version of ARCTangent-A4 processor by reading the identity register. See Figure 18. Information on extensions added to the ARCTangent-A4 processor can be determined through build configuration registers. For more information on build configuration registers please refer to the 'ARCTangent-A4 Development Kit for ARCTangent-A4 Release Notes'.

DEBUG

In order to halt ARCTangent-A4 processor, the host needs to set the FH bit of the debug register. The host can determine how the ARCTangent-A4 processor was halted and if there are any pending loads. See Figure 19.

ARCTangent-A4 Memory

The program memory can 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 ARCTangent-A4 memory space, then the instruction cache should be invalidated.

Stage 3. ALU

Any arithmetic or logic functions are carried out on the operands supplied by stage 2.

Stage 4. Write back

Results from stage 3 or data from loads are written back to the *core registers*.

Pipeline-Cycle Diagram

In the explanation of the passage of instructions through the pipeline stages the diagram in Figure 23 is used.

	t	t+1	t+2	t+3	t+4	t+5
Event 1	stage 1	stage 2	stage 3	stage 4		
Event 2		stage 1	stage 2	stage 3	stage 4	
Event 3			stage 1	stage 2	stage 3	stage 4

Figure 23 Pipeline-Cycle Diagram

Time progresses from left to right and events progress down the page.

In order to read the diagram, take as an example the second cycle at time $t+1$.

Here, *Event 1* has reached *stage 2* in the pipeline and *Event 2* has reached *stage 1* of the pipeline.

If we have the following code:

```
AND    r1, r2, r3
OR     r5, r6, r4
BIC    r8, r9, r10
SUB    r14, r12, r13
```

We can show the events in the pipeline with the following diagram:

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND	ifetch	r2, r3	AND	r1			
OR		ifetch	r6, r4	OR	r5		
BIC			ifetch	r9, r10	BIC	r8	
SUB				ifetch	r12, r13	SUB	r14

At cycle $t+3$

The write back stage (stage 4) is updating $r1$

The ALU stage (stage 3) is performing an OR.

The operand fetch (stage 2) fetching the operands $r9$ and $r10$ for BIC.

The instruction fetch of SUB is occurring at stage 1.

Arithmetic and Logic Function Timings

The stages perform the following operations during an Arithmetic or Logic instruction.

Stage 1

Instruction fetch and start decode

Stage 2

Fetch 2 operands from registers

Stage 3

Do Arithmetic or Logic function

Stage 4

Write result to register

When arithmetic and logic functions are executed sequentially, there are sometimes dependencies of registers between the instructions. Take the following code:

```
AND    r1, r2, r3
OR     r5, r1, r4
BIC    r8, r9, r10
SUB    r14, r12, r13
```

The second instruction OR uses r1 as an operand. Notice that r1 is updated from the previous instruction (AND).

We can see the effect of this dependency in the following pipeline-cycle diagram:

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND	ifetch	r2, r3	AND	r1			
OR		ifetch	r1, r4	OR	r5		
BIC			ifetch	r9, r10	BIC	r8	
SUB				ifetch	r12,r13	SUB	r14

Since r1 is not updated until time t+3 and OR needs r1 at t+2 it would appear that the OR operand fetch would have to be delayed for one cycle.

However, since there is an ALU SHORT CUT (see Figure 22), r1 is ready to be used as an operand by the OR instruction at t+2.

Immediate Data Timing

When immediate data is used, the data is available at different times depending on the size of that data.

Short immediate

The short immediate data of an instruction is available at the operand fetch stage and is taken from the low 9 bits of the instruction. The instruction takes the same time to cycle through the pipeline.

Long immediate

The long immediate data is taken from the word in the instruction fetch stage while the instruction is in the operand fetch stage. The stages that the long immediate data would pass through, if it were an instruction, are disabled.

This means that a long immediate instruction takes one cycle longer and the next instruction is a cycle later.

The stages perform the following operations during an Arithmetic or Logic instruction with immediate data.

Stage 1

Instruction fetch and start decode

Stage 2

Fetch 1 operand from registers and the other from the value currently in stage 1

Disable instruction word in stage 1.

Stage 3

Do Arithmetic or Logic function

Stage 4

Write result to register

For example:

```
AND    r1, r2, 2000
OR     r5, r1, r4
SUB    r14, r12, r13
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND	ifetch	r2	AND	r1			
limm		2000	disabled	disabled	disabled		
OR			ifetch	r3, r4	OR	r5	
SUB				ifetch	r12, r13	SUB	r14

Destination immediate

If the destination for the result of an instruction is marked as being immediate, then the write-back at stage 4 is disabled.

For example:

```
AND    0, r2, r3
OR     r5, r1, r4
BIC    r8, r9, r10
SUB    r14, r12, r13
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND	ifetch	r2, r3	AND	disabled			
OR		ifetch	r1, r4	OR	r5		
BIC			ifetch	r9, r10	BIC	r8	
SUB				ifetch	r12, r13	SUB	r14

Conditional Instruction Timing

Condition code tests for branch, loop and jump instructions occur one stage earlier in the pipeline, rather than in conditional arithmetic and logic instructions, and are covered in section 7.6.

The condition code tests for arithmetic and logic instructions are carried out at the beginning of stage 3. Condition codes are updated at the end of stage 3 ready for the next instruction.

If the test returns a false value, then the following parts of the pipeline are affected for that instruction:

- write-back to the core register set at stage 4 is disabled
- update of the flags at stage 3 is disabled
- the ALU SHORTCUT is disabled

The stages during a conditional Arithmetic or Logic instructions:

Stage 1

Instruction fetch and start decode

Stage 2

Fetch 2 operands from registers

Stage 3

Do Arithmetic or Logic function.

If condition *true* then update flags and enable ALU SHORTCUT

If condition *false* then do not update flags and disable ALU SHORTCUT

Stage 4

If condition *true* then write result to register

Take the following code, where the result of the AND is zero:

```
AND.F  r1,r2,r3
OR.NE.F    r5,r6,r4
BIC     r8,r9,r10
SUB     r14,r12,r13
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND.F	ifetch	r2, r3	<i>flag update</i>	r1			
OR.NE.F		ifetch	r6, r4	<i>condition code test</i>	<i>Disable wrt-back</i>		
BIC			ifetch	r9, r10	BIC	r8	
SUB				ifetch	r12,r13	SUB	r14

Extension Instruction Timings

Single cycle extension instructions

Single cycle extension instructions follow the same characteristics as arithmetic and logic functions, immediate data and conditional instruction timing. The following extension options have single cycle extension instruction characteristics:

- 32-bit Barrel shift/rotate block (single cycle)
- Normalise (find-first-bit) instruction
- Swap instruction
- MIN/MAX instructions

Multi cycle extension instructions

Multi cycle extension instructions will stall the pipeline if basecase core registers are being written to. If extension core or auxiliary registers are defined as specific destination registers, then the pipeline will only stall if the extension register is being accessed and the extension instruction has built-in scoreboarding. The following extension options have multi cycle extension instruction characteristics:

- 32-bit Barrel shift/rotate block (multi cycle)
- 32-bit Multiplier, small (10 cycle) implementation
- 32-bit Multiplier, fast (4 cycle) implementation

Multiply timings

The stages perform the following operations for the multiply instruction:

Stage 1

Instruction fetch and start decode

Stage 2

Fetch operands.

Update multiply scoreboard unit with result registers (MLO, MMID, MHI) marked as invalid.

Stage 3

Perform multiply in four (or ten for small implementation) cycles.

Allow shortcutting of multiply result if required.

Stage 4

No action.

On completion of multiply

Update multiply result registers and update multiply scoreboard unit marking result registers as valid.

When the multiply registers are waiting to be updated by a multiply and any of those registers are one of the operands of an instruction in the pipeline at stage 2 then the pipeline is halted until the multiply completes.

A special scoreboard unit is used to retain the information on which registers are waiting to be written. The scoreboard unit is updated at stage 2 when the multiply is executed, and updated at stage 4 when the result registers have been written to.

The pipeline is allowed to proceed if the instruction following the multiply does not use the multiply result registers (this is checked by the instruction in stage 2). Once an instruction does need that register then the pipeline is halted and waits for the load to complete.

The result will not be ready for four (or ten for the small implementation) cycles after the multiply instruction has been issued. Note, that this time is not affected by other pipeline stalls in the system - once it is issued, the multiply will be ready after four (or ten for the small implementation) cycles under all conditions.

When the result of the multiply is ready the multiply result registers are updated with no affect on the pipeline.

In this example, the multiply takes four cycles to get to the short cut value.

main:

```
MUL64 0,r2,r3
AND    r4,r5,r6
OR     r7,r8,r9
SUB    r10,r11,r12
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
MUL64	ifetch	r2,r3	MUL64	<i>killed</i>			<i>wrt-back</i>
AND		ifetch	r5,r6	AND	r4		
OR			ifetch	r8,r9	OR	r7	
SUB				ifetch	r11,r12	SUB	r10

score board	mark as pending	check	check	check	check	result now ready
result registers	<i>not set</i>	<i>not set</i>	<i>not set</i>	<i>not set</i>	<i>not set</i>	mult. result

If the AND used one of the multiply result register then the AND would stall. For example, with a dependency on MLO:

main:

```
MUL64 0,r2,r3
AND    r4,mlo,r6
OR     r7,r8,r9
SUB    r10,r11,r12
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
MUL64	ifetch	r2,r3	MUL64	<i>killed</i>		<i>short-cut</i>	<i>wrt-back</i>
AND		ifetch	mlo,r6	<i>stalled</i>	<i>stalled</i>	<i>stalled</i>	AND
OR			ifetch	<i>stalled</i>	<i>stalled</i>	<i>stalled</i>	r8,r9
SUB							ifetch

score board	mark as pending	check	check	check	check	result now ready
result registers	<i>not set</i>	<i>not set</i>	<i>not set</i>	<i>not set</i>	<i>not set</i>	mult. result

Barrel shift timings

The fast barrel shift has the same timing characteristics as arithmetic and logic functions, immediate data and conditional instruction timing. The small barrel shift, however, will stall the pipeline until the shift is complete. The number of cycles required to complete a barrel shift operation depend on the number of bit shifts in that operation, which in this implementation, processes 4 bit shifts per cycle. Thus, the number of cycles will vary from one (0 to 4 bits) to eight (29 to 32 bits) cycles.

The stages perform the following operations during a barrel shift instruction.

Stage 1

Instruction fetch and start decode

Stage 2

Fetch 2 operands from registers

Stage 3

Do shift function, stalling pipeline if required.

Stage 4

Write result to register

The pipeline will stall depending on the size of the shift. In the following example a shift of 3 will take one cycle and use the short-cutting mechanism to get the result register in time:

```
ASL    r1, r2, 0x3
OR     r5, r1, r4
BIC    r8, r9, r10
SUB    r14, r12, r13
```

The second instruction OR uses r1 as an operand. Notice that r1 is updated from the barrel shift instruction (ASL).

	t	t+1	t+2	t+3	t+4	t+5	t+6
ASL	ifetch	r2, 0x3	ASL	r1			
OR		ifetch	r1, r4	OR	r5		
BIC			ifetch	r9, r10	BIC	r8	
SUB				ifetch	r12, r13	SUB	r14

In the following example a shift of 5 will take two cycles, causing a stall of one cycle using the short-cutting mechanism to get the result:

```
ASL    r1, r2, 0x5
OR     r5, r1, r4
BIC    r8, r9, r10
SUB    r14, r12, r13
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
ASL	ifetch	r2, 0x5	ASL	stalled	r1		
OR		ifetch	r1, r4	stalled	OR	r5	
BIC			ifetch	stalled	r9, r10	BIC	r8
SUB					ifetch	r12, r13	SUB

Jump and Branch Timings

Jump instruction

The jump instruction performs the following action in the pipeline:

Stage 1

Instruction fetch and start decode

Stage 2

Fetch operand.

Test condition code.

If condition *true* update PC with operand, and if flag bit set then update flags. If condition *false* allow PC to update normally.

Execute instruction in delay slot according to the nullify instruction mode.

Stage 3

No action

Stage 4

No action

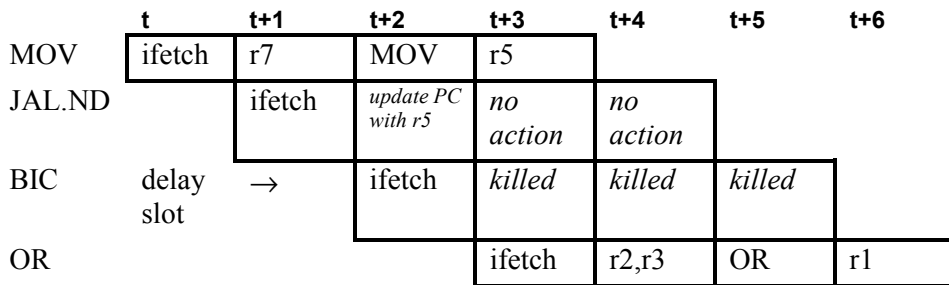
Jump and nullify delay slot instruction

If a jump is not conditional, then the jump is always taken and the instruction immediately following the jump is executed according to the nullify instruction mode.

A single cycle stall will occur if a jump is immediately preceded by an instruction that sets the flags. Assuming that r7 contains the address of the code that starts at label jaddr take the following code:

```
main:
    MOV    r5,r7
    JAL.ND [r5]
    BIC    r8,r9,r10
    SUB    r14,r12,r13
    ...
jaddr:
    OR     r1,r2,r3
```

The current PC is also included in the following diagram:



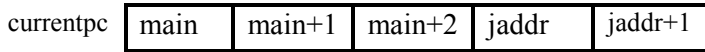
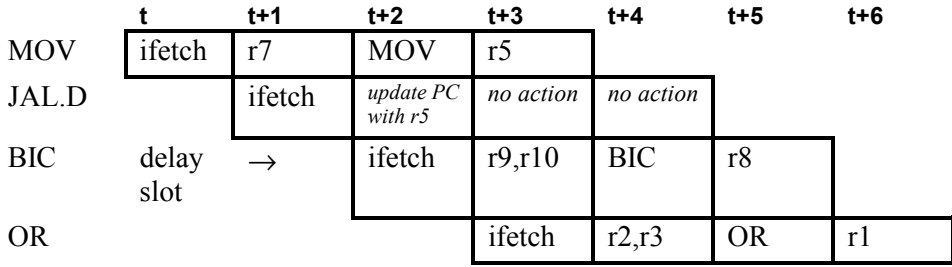
currentpc	main	main+1	main+2	jaddr	jaddr+1
-----------	------	--------	--------	-------	---------

Jump and execute delay slot instruction

When the delay slot execution flag is set then the above code would become:

```
main:
    MOV    r5,r7
    JAL.D  [r5]
    BIC    r8,r9,r10
    SUB    r14,r12,r13
    ...
jaddr:
    OR     r1,r2,r3
```

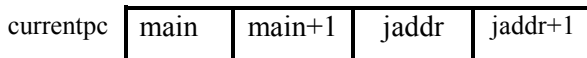
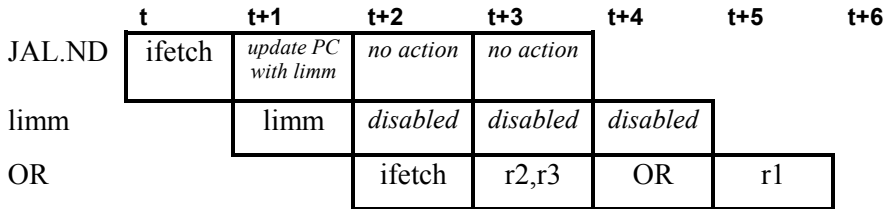
The affect of this code through the pipeline is shown in the following diagram:



Jump with immediate address

When a jump occurs with a long immediate address it takes an extra cycle to execute. The delay slot execution mechanism does not apply since the long immediate data is contained in the delay slot. A single cycle stall will occur if a jump is immediately preceded by an instruction that sets the flags.

```
main:
    JAL    jaddr
    BIC    r8,r9,r10
    SUB    r14,r12,r13
    ...
jaddr:
    OR     r1,r2,r3
```



Jump setting flags

If the flags update field is used by the jump instruction then the flags, except the H bit, will be updated in stage 3. A single cycle stall will occur if a jump is immediately preceded by an instruction that sets the flags.

main:

```
MOV.F  r5,r7
JAL.D.F [r5] ; pipeline stall due to flags set by MOV
BIC.F  r8,r9,r10
SUB    r14,r12,r13
...
```

jaddr:

```
OR     r1,r2,r3
```

NOTE In this case that because the BIC instruction is in the delay slot, the flags are changed after the jump by BIC. If the delay slot instruction was nullified then the flags would only be changed by the jump instruction.

	t	t+1	t+2	t+3	t+4	t+5	t+6
MOV.F	ifetch	r7	MOV	r5			
JAL.D.F		ifetch	stall (wait for flags)	update PC with r5	update flags	no action	
BIC.F	delay slot	→		ifetch	r9,r10	BIC	r8
OR					ifetch	r2,r3	OR

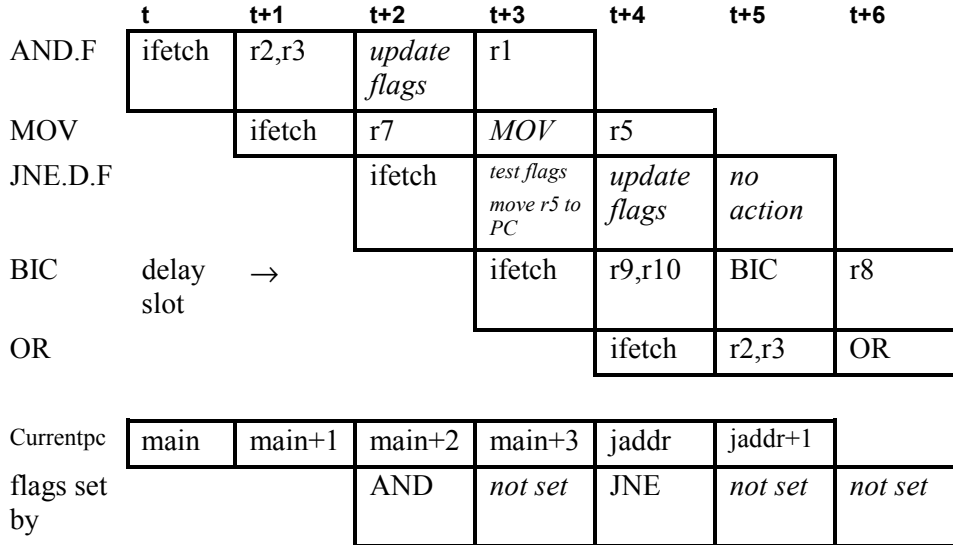
current pc	main	main+1	main+2	main+2	jaddr	jaddr+1	
flags set by			MOV	not set	JAL	BIC	not set

Conditional jump

Condition code tests for branch, loop and jump instructions happen at stage 2 in the pipeline, rather than at stage 3 for conditional arithmetic and logic instructions. As a result, a single cycle stall will occur if a jump is immediately preceded by an instruction that sets the flags.

In the following example, the flags are set by two instructions and it can be seen in the pipeline-cycle diagram where the effects of the flags occur.

```
main:
    AND.F   r1,r2,r3
    MOV     r5,r7
    JNE.D.F [r5]
    BIC     r8,r9,r10
    SUB     r14,r12,r13
    ...
jaddr:
    OR      r1,r2,r3
```

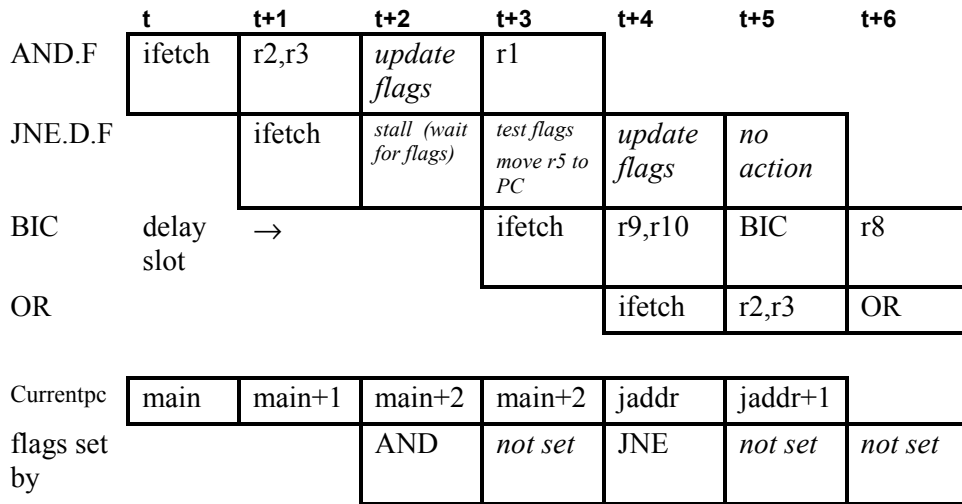


The jump instruction tests the flags that have been updated by the AND instruction.

NOTE If the BIC instruction was conditional then it would be executed according to the flags set by the jump instruction.

In the following example, the flag setting instruction is immediately followed by the jump instruction and it can be seen in the pipeline-cycle diagram where the effects of the flags occur and where the pipeline stall occurs.

```
main:
    AND.F   r1,r2,r3
    JNE.D.F [r5] ; pipeline stall due to flags set by
AND
    BIC     r8,r9,r10
    SUB     r14,r12,r13
    ...
jaddr:
    OR      r1,r2,r3
```



Jump and link

The jump and link instruction is very similar to the jump instruction except that the branch link register (BLINK) is used to allow returns from subroutines. Unlike the non linking jump, stage 3 and stage 4 are enabled to allow the link register to be updated with the status register value. The whole of the status register is saved and is taken either from the first instruction following the branch (current PC) or the instruction after that (next PC) according to the delay slot execution mode. If the destination address is an explicit address (long immediate data) then for this instruction the .JD nullify instruction mode *must* be used. If .D or .ND is used, incorrectly, then the link register BLINK will contain the wrong return address.

The flags stored are those set by the instruction immediately preceding the jump. A single cycle stall will occur if a jump and link is immediately preceded by an instruction that sets the flags.

Stage 1

Instruction fetch and start decode

Stage 2

Test condition code.

If condition *true* update PC with operand, and if flag bit set then update flags.

If condition *false* allow PC to update normally.

Execute instruction in delay slot according to the nullify instruction mode.

Stage 3

If condition *true* then pass PC to stage 4

Stage 4

If condition *true* then write return address to LINK register.

```
main:
AND.F  r1,r2,r3
      MOV    r5,r7
      JLNE.D jaddr
      BIC    r8,r9,r10
      SUB    r14,r12,r13
      ...
jaddr: OR     r1,r2,r3
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND.F	ifetch	r2,r3	<i>update flags</i>	r1			
MOV		ifetch	r7	<i>MOV</i>	r5		
JLNE.D			ifetch	<i>test flags update PC</i>	<i>pass next_pc through</i>	<i>write back BLINK</i>	
BIC	delay slot →			ifetch	r9,r10	BIC	r8
OR					ifetch	r2,r3	OR

currentp c	main	main+1	main+2	main+3	jaddr	jaddr+1	
next_pc	main+1	main+2	main+3	main+4	jaddr+1	jaddr+2	
BLINK			<i>not set</i>	<i>not set</i>	<i>not set</i>	<i>updated</i>	main+4
flags set by			AND	<i>not set</i>	<i>not set</i>	<i>not set</i>	<i>not set</i>

Branch

When a branch is taken, like the jump instruction, the instruction in the delay slot is executed according to the nullify instruction mode. The relative address is calculated and the PC updated in stage 2. A single cycle stall will occur if a branch is immediately preceded by an instruction that sets the flags.

Calculation of the relative address

The branch target address is calculated by adding the offset within the instruction to the address of branch instruction. The target address is calculated thus:

$$\text{new program counter} = \text{branch PC address} + 24\text{-bit offset} + 1$$

Hence, if the relative address was 0, then the target of the branch would be that instruction in the delay slot.

Stage 1

Instruction fetch and start decode

Stage 2

Test condition code. If condition *true* update PC with calculated address.

If condition *false* allow PC to update normally.

Execute instruction in delay slot according to the nullify instruction mode.

Stage 3

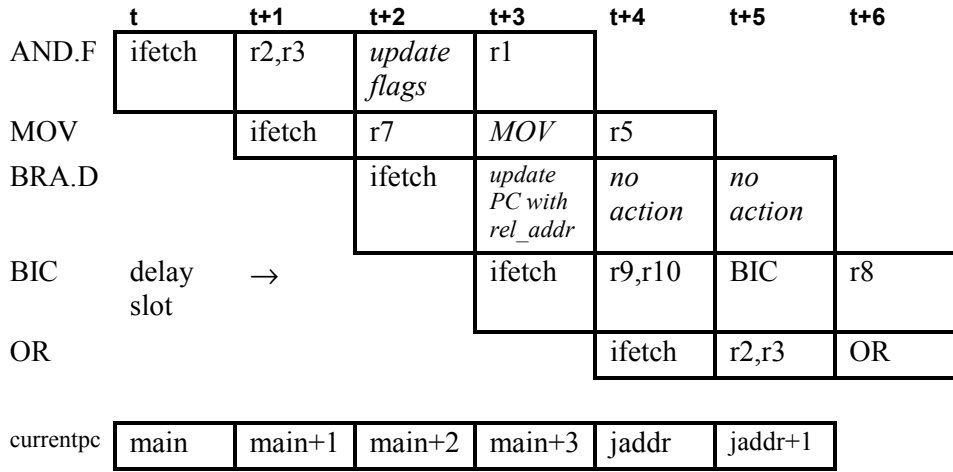
No action

Stage 4

No action

In this example the delay slot instruction is executed.

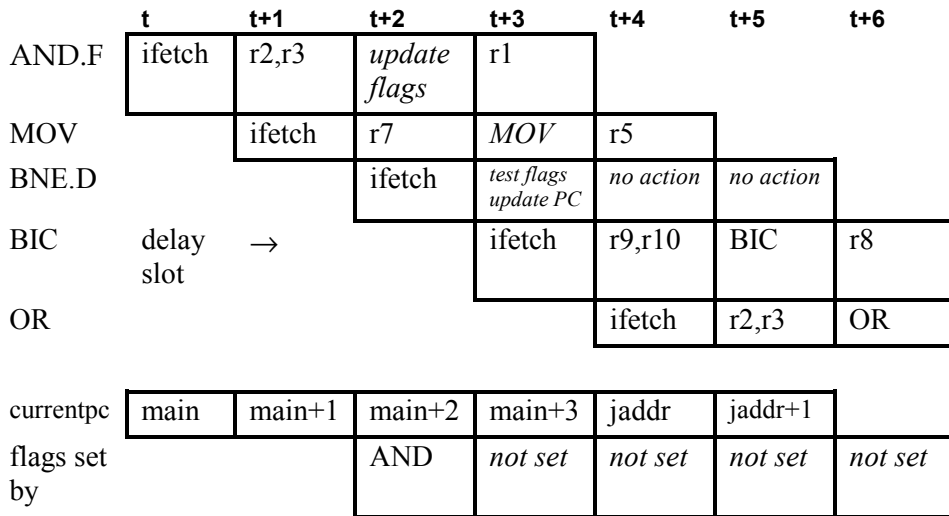
```
main:
    AND.F  r1,r2,r3
    MOV    r5,r7
    BRA.D  jaddr
    BIC    r8,r9,r10
    SUB    r14,r12,r13
    ...
jaddr:
    OR     r1,r2,r3
```



Conditional branch

The condition codes are tested at stage 2, as in the jump instruction. A single cycle stall will occur if a conditional branch is immediately preceded by an instruction that sets the flags.

```
main:
    AND.F  r1,r2,r3
    MOV    r5,r7
    BNE.D  jaddr
    BIC    r8,r9,r10
    SUB    r14,r12,r13
    ...
jaddr:
    OR     r1,r2,r3
```



Software breakpoints

A software breakpoint is implemented by the use of the branch instruction, Bcc. The action of a software breakpoint is to branch to the breakpoint code whereupon the appropriate action will be taken according to the debugging session, for example, write a value to a register and halt the ARCTangent-A4 processor.

Software breakpoint return address calculation

Software breakpoints can be placed anywhere in ARCTangent-A4 code, except in executed delay slots of branches.

For example:

```
BNE.D address
NOP          ; ←break point may not be placed here
BCS.ND address
NOP          ; ←break point may be placed here
```

Once the breakpoint is hit and the breakpoint code is executed, there is a problem on how to restart the code after the breakpoint. The next instruction to have been fetched will be the target of the branch *not* the instruction that was replaced by the breakpoint.

In this case, for debugging purposes, the breakpoint should replace the branch in the ARCTangent-A4 code rather than the instruction in the delay slot.

Breakpoints may be set on instructions following branches, these do not get executed as delay slots. In other words, it is okay to place breakpoints in the instruction slot following a branch, jump or loop instruction that uses the ND delay slot canceling mode.

Branch and link

The branch and link instruction is very similar to the branch instruction except that the branch link register (BLINK) is used to allow returns from subroutines. Unlike the non linking branch, stage 3 and stage 4 are enabled to allow the link register to be updated with the status register value. The whole of the status register is saved and is taken either from the first instruction following the branch (current PC) or the instruction after that (next PC) according to the delay slot execution mode.

The flags stored are those set by the instruction immediately preceding the branch. A single cycle stall will occur if a branch and link is immediately preceded by an instruction that sets the flags.

Stage 1

Instruction fetch and start decode

Stage 2

Test condition code.

If condition *true* update PC with calculated address.

If condition *false* allow PC to update normally.

Execute instruction in delay slot according to the nullify instruction mode.

Stage 3

If condition *true* then pass PC to stage 4

Stage 4

If condition *true* then write return address to LINK register.

```
main:
    AND.F  r1,r2,r3
    MOV    r5,r7
    BLNE.D jaddr
    BIC     r8,r9,r10
    SUB     r14,r12,r13
    ...
jaddr:
    OR      r1,r2,r3
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND.F	ifetch	r2,r3	<i>update flags</i>	r1			
MOV		ifetch	r7	<i>MOV</i>	r5		
BLNE.D			ifetch	<i>test flags update PC</i>	<i>pass next_pc through</i>	<i>write back BLINK</i>	
BIC	delay slot →			ifetch	r9,r10	BIC	r8
OR					ifetch	r2,r3	OR

currentpc	main	main+1	main+2	main+3	jaddr	jaddr+1	
next_pc	main+1	main+2	main+3	main+4	jaddr+1	jaddr+2	
BLINK			<i>not set</i>	<i>not set</i>	<i>not set</i>	updated	main+4
flags set by			AND	<i>not set</i>	<i>not set</i>	<i>not set</i>	<i>not set</i>

Loop Timings

Loop set up

The loop instruction sets up the loop start (LP_START) and loop end (LP_END) registers. LP_START register is updated with CURRENT PC and LP_END updated with the relative address (REL_ADDR) at stage 2.

A single cycle stall will occur if a loop is immediately preceded by an instruction that sets the flags.

Stage 1

Instruction fetch and start decode

Stage 2

Fetch address from instruction.

Test condition code.

If condition *true* allow PC to update normally and update LP_END with address and update LP_START.

If condition *false* update PC with address.

Execute instruction in delay slot according to the nullify instruction mode.

Stage 3

No action

Stage 4

No action

```
main:
    AND.F  r1,r2,r3
    MOV    r5,r7
    LP     loop1
    BIC    r8,r9,r10
    SUB    r14,r12,r13
loop1:
    OR     r1,r2,r3
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND.F	ifetch	r2,r3	update flags	r1			
MOV		ifetch	r7	MOV	r5		
LP			ifetch	update loop registers	no action	no action	
BIC	delay slot	→		ifetch	r9,r10	BIC	r8
SUB					ifetch	r12,r13	SUB

currentpc	main	main+1	main+2	main+3	main+4			
next_pc	main+1	main+2	main+3	main+4				
LP_START			not set	updated	main+3	main+3	main+3	
LP_END			not set	updated	loop1	loop1	loop1	
flags set by			AND	MOV	not set	not set	not set	

Conditional loop

The conditional LP instruction is similar to the branch instruction. If the condition code test for the LP instruction returns *false*, then a branch occurs to the address specified in the LP instruction. 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 condition codes are tested in stage 2, like branch, and there is the same delay slot nullify instruction mode. A single cycle stall will occur if a conditional loop is immediately preceded by an instruction that sets the flags.

For example, the loop is executed:

```
main:
    AND.F r1,r2,r3    ; clears zero flag
    MOV    r5,r7
    LPNE.D loop1
    BIC    r8,r9,r10
    SUB    r14,r12,r13
loop1:
    OR     r1,r2,r3
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND.F	ifetch	r2,r3	update flags	r1			
MOV		ifetch	r7	MOV	r5		
LPNE.D			ifetch	test flags update registers	no action	no action	
BIC	delay slot →			ifetch	r9,r10	BIC	r8
SUB					ifetch	r12,r13	SUB

currentpc	main	main+1	main+2	main+3	main+4		
next_pc	main+1	main+2	main+3	main+4			
LP_START			not set	updated	main+3	main+3	main+3
LP_END			not set	updated	loop1	loop1	loop1
flags set by			AND	MOV	not set	not set	not set

If the condition code result is false, then a jump to the relative address is taken and the instruction in the delay slot executed according to the nullify instruction mode, shown in the following:

```
main:
    AND.F  r1,r2,r3      ; sets zero flag
    MOV    r5,r7
    LPNE.D loop1
    BIC    r8,r9,r10
    SUB    r14,r12,r13
loop1:
    OR     r1,r2,r3
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND.F	ifetch	r2,r3	update flags	r1			
MOV		ifetch	r7	MOV	r5		
LPNE.D			ifetch	test flags update PC	no action	no action	
BIC	delay slot →			ifetch	r9,r10	BIC	r8
OR					ifetch	r2,r3	OR

currentpc	main	main+1	main+2	loop1	loop1+1		
next_pc	main+1	main+2	main+3	main+4	loop1+2		
LP_START			not set	not set	not set	not set	not set
LP_END			not set	not set	not set	not set	not set
flags set by			AND	MOV	not set	not set	not set

Loop execution

The operation of the loop is such that the PC+1 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 increment normally and LP_COUNT is decremented.

```
main:
    AND.F  r1,r2,r3
    MOV    r5,r7
    LP     loop1
    BIC    r8,r9,r10
    SUB    r14,r12,r13
loop1:
    OR     r1,r2,r3
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
BIC	ifetch	r9,r10	BIC	r8			
SUB		ifetch	r12,r13	SUB	r14		
BIC			ifetch	r9,r10	BIC	r8	
SUB				ifetch	r12,r13	SUB	r14
OR					ifetch	r2,r3	OR

currentpc	main+3	main+4	main+3	main+4	loop1
PC+1	main+4	loop1	main+4	loop1	loop1+1
LP_COUNT	2	2→1	1	1→0	0
LP_START	main+3	main+3	main+3	main+3	main+3
LP_END	loop1	loop1	loop1	loop1	loop1

Single instruction loops

Single instruction loops cannot be set up with the LP instruction. The LP instruction can set up loops with 2 or more instructions in them. However, it is possible to set up a single instruction loop with the use of the LR and SR instructions.

If a single instruction loop is attempted to be set up with the LP instruction, then the instruction in the loop (OR) will be executed once and then the code following the loop (ADD) will be executed as normal. The LP_START and LP_END registers *will* be updated by the time the instruction after the attempted loop (ADD) is being fetched, which is, however, too late for the loop mechanism.

```
main:
    LP     loop_end      ; this will execute only once
loop_in: OR     r21,r22,r23 ; single instruction in loop
loop_end:
    ADD    r19,r19,r20    ; first instruction after loop
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
LP	ifetch	update loop registers	no action	no action			
OR		ifetch	r22,r23	OR	r21		
ADD			ifetch	r19,r20	ADD	r19	

currentpc	main	main+1	main+2	main+3	main+4
LP_START	previous	previous	loop_in	loop_in	loop_in
LP_END	previous	previous	loop_end	loop_end	loop_end

If the user wishes to have single instruction loops, then the following code can be used. Notice, there has to be a delay to allow the loop start and loop end registers to be updated with the SR instruction.

```

MOV    LP_COUNT,5      ; no. of times to do loop
MOV    r0,dooploop>>2 ; convert to long-word size
ADD    r1,r0,1         ; add 1 to dooploop address
main:  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
dooploop: OR    r21,r22,r23 ; single instruction in loop
        ADD    r19,r19,r20 ; first instruction after loop

```

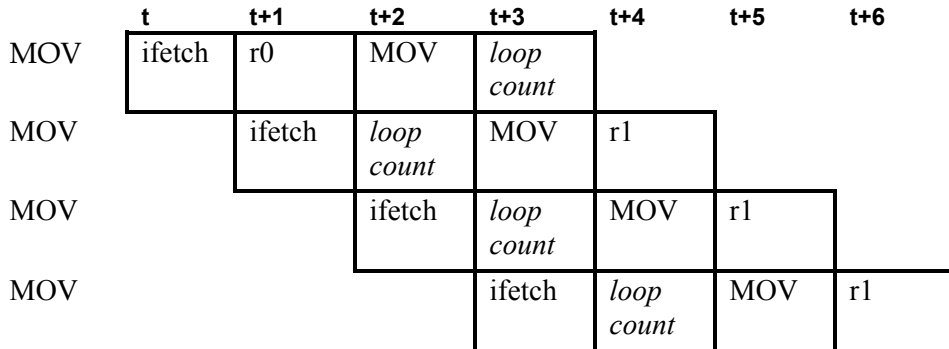
	t	t+1	t+2	t+3	t+4	t+5	t+6
SR	ifetch	r0	update loop start	no action			
SR		ifetch	r1	update loop end	no action		
NOP			ifetch	NOP	NOP	NOP	
NOP				ifetch	NOP	NOP	NOP
OR					ifetch	r22,r23	OR
OR						ifetch	r22,r23

currentpc	main	main+1	main+2	main+3	main+4	dooploop	dooploop
PC+1	main+1	main+2	main+3	main+4	dooploop+1	dooploop+1	dooploop+1
LP_START	previous	previous	previous	dooploop	dooploop	dooploop	dooploop
LP_END	previous	previous	previous	previous	dooploop+1	dooploop+1	dooploop+1

Reading loop count register

The loop count register, unlike other core registers, has short-cutting disabled. This means that there must be at least 2 instructions (actually 2 cycles) between an instruction writing LP_COUNT and one reading LP_COUNT.

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



LP_COUNT	previous	previous	previous	update	new value
----------	----------	----------	----------	--------	--------------

When reading from the loop count register (LP_COUNT) the user must be aware that the value returned is that value of the counter that applies to the next instruction to be executed. This means that if the last instruction in a loop reads LP_COUNT then the value returned would be that value after the loop mechanism has updated it.

```
    . . .
    AND.F    0,0,LP_COUNT    ; loop count for this iteration
    AND.F    0,0,LP_COUNT    ; loop count for next iteration
loop_end:
    ADD      r19,r19,r20    ; first instruction after loop
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
AND.F	ifetch	<i>loop count</i>	AND	<i>no action</i>			
AND.F		ifetch	<i>loop count</i>	AND	<i>no action</i>		

currentpc	loop_end -2	loop_end -1	loop_in
PC+1	loop_end -1	loop_end	loop_in +1
LP_END	loop_end	loop_end	loop_end
LP_COUNT	previous	previous	new value

Writing loop count register

In order for the loop mechanism to work properly, the loop count register must be set up with at least 3 instructions (actually 3 cycles) between it and the last instruction in the loop. In the following example, the MOV instruction will override the loop mechanism and the loop will be executed one more time than expected. The MOV instruction must be followed by a NOP for correct execution.

```
main:      MOV    LP_COUNT,r0      ; do loop r0 times (flags not set)
          LPZ    loop_end        ; if zero flag set jump to loop_end
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
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
MOV	ifetch	r0	MOV	loop count			
LPZ		ifetch	update loop registers	no action	no action		
OR			ifetch	r22,r23	OR	r21	
AND				ifetch	r21,23	AND	killed
OR					ifetch	r22,r23	OR

currentpc	main	main+1	main+2	main+3	loop_in	loop_end -1	loop_in
PC+1	main+1	main+2	main+3	loop_end	loop_end -1	loop_end	loop_end -1
LP_START	previous	previous	previous	loop_in	loop_in	loop_in	loop_in
LP_END	previous	previous	previous	loop_end	loop_end	loop_end	loop_end
LP_COUNT	previous	previous	previous	override	r0	r0	r0-1

The loop count register is set up correctly in the following:

```

MOV    LP_COUNT,r0    ; do loop r0 times (flags not set)
NOP                    ; allow time for loop count set up
LPZ    loop_end        ; if zero flag set jump to
loop_end
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

```

	t	t+1	t+2	t+3	t+4	t+5	t+6
MOV	ifetch	r0	MOV	loop count			
NOP		ifetch	NOP	NOP	NOP		
LPZ			ifetch	update loop registers	no action	no action	
OR				ifetch	r22,r23	OR	r21
AND					ifetch	r21,23	AND
OR						ifetch	r19,r20

currentpc	main	main+1	main+2	main+3	main+4	loop_in	loop_end -1
PC+1	main+1	main+2	main+3	main+4	loop_end	loop_end -1	loop_end
LP_START	previous	previous	previous	previous	loop_in	loop_in	loop_in
LP_END	previous	previous	previous	previous	loop_end	loop_end	loop_end
LP_COUNT	previous	previous	previous	update	r0	r0-1	r0-1

Branch and jumps in loops

Jumps or branches without linking will work correctly in any position in the loop. There are, however, some side effects when a branch or jump is the last instruction in a loop:

Firstly, it is possible that the branch or jump instruction is contained in the very last long-word position in the loop. This means that the instruction in the delay slot would be either the first instruction *after* the loop or the first instruction *in* the loop (pointed to by loop start register) depending on the result of the loop mechanism. The instruction in the delay slot will be that which would be executed if the branch or jump was replaced by a NOP.

If a branch-and-link or jump-and-link instruction is used in the one before last long-word position in a loop then the return address stored in the link register (BLINK) may contain the wrong value. The following instructions will store the address of the first instruction *after* the loop, and therefore should not be used in the second to last position:

BLcc.D address

BLcc.JD address

JLcc.D [Rn]

JLcc.JD[Rn]

JLcc address

If the ND delay slot execution mode is used for branch-and-link or jump-and-link instruction in the one before last long-word position in a loop then the return address is stored correctly in the link register. The loop count does not decrement if the instruction fetched was subsequently killed as the result of a branch/jump operation. For these reasons, it is recommended that subroutine calls should not be used within the loop mechanism.

Software breakpoints in loops

A software breakpoint is implemented by the use of the branch instruction, Bcc. The action of a software breakpoint is to branch to the breakpoint code whereupon the appropriate action will be taken according to the debugging session, for example, write a value to a register and halt the ARCTangent-A4 processor. The loop count does not decrement if the instruction fetched was subsequently killed as the result of a branch/jump operation. Therefore, since the software breakpoint is BRA.ND by default, then the loop counter will not decrement on exit from the loop. On return to the loop the second fetch of the last instruction in the loop will cause the loop counter to decrement as normal.

Instructions with long immediate data

It is difficult, but nonetheless possible, that an instruction that uses long immediate data is contained in the very last long-word position in the loop. This means that the long immediate data would be either be taken from the first location *after* the loop or the first location *in* the loop (pointed to by loop start register) depending on the result of the loop mechanism. It is unlikely that this would occur with sensible coding but the following example shows how you could do it.

```

MOV      r1,limmloop>>2      ; convert to long-word size
ADD      r1,r1,1              ; add 1 to limmloop address
SR       r1,[LP_END]          ; set up loop end register
NOP                               ; allow time to update reg
NOP
limmloop: OR  r21,r22,2048      ; instruction across loop end
ADD      r19,r19,r20           ;

```


Flag Instruction Timings

The flag instruction has very similar timing as arithmetic and logic instruction. However, since the flag instruction can halt the ARctangent-A4 processor, the pipeline is halted with the following instruction in stage 3. If only the H bit is set then the other flags are unchanged. See example below:

```
main:
    FLAG    1          ; halt the ARctangent-A4
    OR      r21,r22,r23 ;
    AND     r1,r2,r3    ;
    XOR     r5,r6,r7    ;
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
FLAG	ifetch	1	FLAG	<i>no action</i>	<i>no action</i>	<i>no action</i>	
OR		ifetch	r22,r23	OR	OR	OR	
AND			ifetch	r2,r3	r2,r3	r2,r3	
XOR				ifetch	ifetch	ifetch	

currentpc	main	main+1	main+2	main+3	main+3	main+3
next_pc	main+1	main+2	main+3	main+4	main+4	main+4
FLAGS	previous	previous	previous	previous	previous	previous
H	0	0	0	1	1	1

Breakpoint

The breakpoint instruction is decoded in stage one of the ARctangent-A4 pipeline, and the remaining stages are allowed to complete. Effectively flushing the pipeline.

The BRK instruction stops any further instructions entering the pipeline. To resume execution the host will read the program counter (frozen at t+1, below), re-write current (BRK) memory location with the required instruction, invalidate the cache (if implemented) and then restart at that memory location.

Stage 1

Decode the BRK instruction. Set the BH bit.

Stage 2

No action

Stage 3

Update the H bit, pipeline halted

Stage 4

No action.

main:

```
ADD    r0,r1,r2      ;
BRK    ;              ;
SUB    r3,r4,r5      ;
```

	t	t+1	t+2	t+3	t+4	t+5	t+6
ADD	ifetch	r1,r2	ADD	r0			
BRK		ifetch	no action	BRK	halted	halted	
			Stalled	no action	no action	no action	
				Stalled	no action	no action	
SUB	Not	fetched					

currentpc	main	main+1	main+1	main+1	main+1	main+1
next_pc	main+1	main+2	main+2	main+2	main+2	main+2
FLAGS	previous	previous	previous	previous	previous	previous
H	0	0	0	0	1	1
BH	0	0	1	1	1	1

Sleep Mode

The SLEEP instruction is decoded at stage 2 of the ARctangent-A4 pipeline. When SLEEP reaches stage 2 the earlier instructions and the SLEEP instruction itself are flushed from the pipe and the processor is then put into sleep mode.

The instruction following the SLEEP enters stage 1 and stays there, until the ARctangent-A4 processor is "woken up" from sleep mode.

Stage 1

Instruction fetch and start decode.

Stage 2

Full decode of sleep instruction. Flush pipeline. Update ZZ bit

Stage 3

No action.

Stage 4

No action

main:

```

ADD    r0,r1,r2      ;
SLEEP                      ;
SUB    r3,r4,r5      ;

```

	t	t+1	t+2	t+3	t+4	t+5	t+6
ADD	ifetch	r1,r2	ADD	r0			
SLEEP		ifetch	SLEEP	no action	no action	no action	
SUB			ifetch	no action	no action	no action	
...				no action	no action	no action	

currentpc	main	main+1	main+2	main+2	main+2	main+2
next_pc	main+1	main+2	main+3	main+3	main+3	main+3
FLAGS	previous	previous	previous	previous	previous	previous
H	0	0	0	0	0	0
ZZ	0	0	0	1	1	1

On interrupt wake up the interrupt mechanism comes into play. The instruction following SLEEP is replaced with by a call to the interrupt service routine. The address of the instruction is copied into the appropriate ILINK register. See interrupt timings for further details.

On host wake up the processor is simply restarted by re-writing the PC with the address of the instruction following the SLEEP with the H bit cleared.

On single-instruction-step the ARCTangent-A4 processor "wakes" from sleep mode. See single instruction step timings for further details.

Load and Store Timings

Loads and stores use the ALU in stage 3 to calculate the address with which the access is to occur.

Load

The stages perform the following in a load instruction:

Stage 1

Instruction fetch and start decode

Stage 2

Fetch operands.

Update scoreboard unit with destination address marked as invalid.

Stage 3

Add operands to form address.

Request load from memory controller.

Stage 4

If address write-back *enabled* then write-back address calculation to first operand register.

If address write-back *disabled* then allow pipeline to continue because data is unlikely to be ready.

also

Stage 4

Re-enabled when data ready from memory controller, pipeline held for one cycle.

Update scoreboard unit marking register as valid.

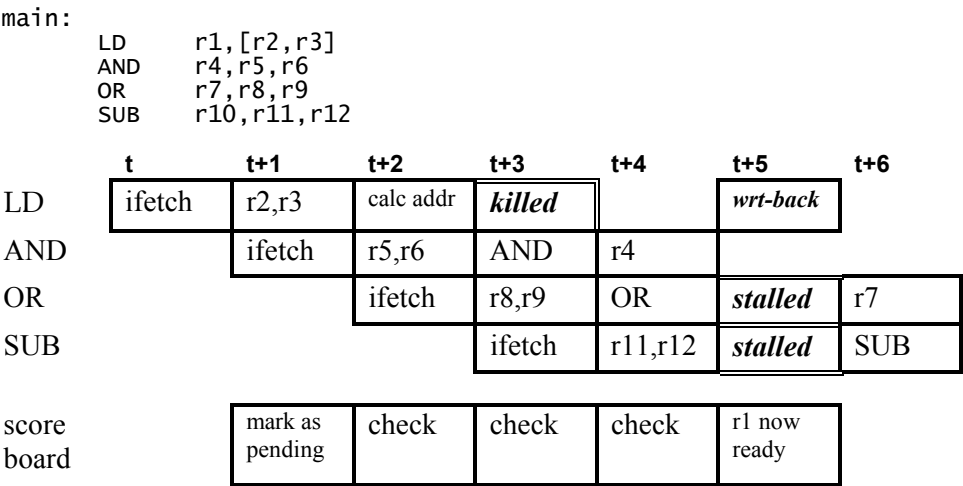
When a register is waiting to be updated by a previous load and that register is one of the operands or results of an instruction in the pipeline at stage 2 then the pipeline is halted until that register is updated.

A scoreboard unit is used to retain the information on which registers are waiting to be written. The scoreboard unit is updated at stage 2 when the destination register address is known, and updated at stage 4 when the register has been written to.

The load is sometimes called a *delayed load* because the data from the load is not guaranteed to be returned by the time the load instruction has reached stage 4 in the pipeline. The pipeline is allowed to proceed if the instruction following the load does not use the destination register of the load (this is checked by the instruction in stage 2). Once an instruction does need that register then the pipeline is halted and waits for the load to complete.

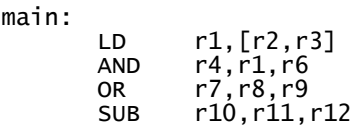
NOTE When the target of a LD.A instruction is the same register as the one used for address write-back (.A), the returning load will overwrite the value from the address write-back.

When the data for the delayed load is ready, the pipeline is stalled because the load uses the write-back in stage 4 to update the register. In this example, the load is delayed by two cycles. The OR instruction is stalled in stage 3 and the SUB stalled in stage 2 until the register write-back is complete.



If the AND used a register that was dependent on the result of the load then the AND would stall.

For example, with a dependency on R1:



	t	t+1	t+2	t+3	t+4	t+5	t+6
LD	ifetch	r2,r3	calc addr	<i>killed</i>		wrt-back	
AND		ifetch	r1,r6	<i>stalled</i>	<i>stalled</i>	alu op	wrt-back
OR			ifetch	<i>stalled</i>	<i>stalled</i>	r8,r9	
SUB						ifetch	r11,r12
score board		mark as pending	check, causes stall	-	-	r1 now ready	

Store

The store instruction takes a single cycle to complete. The data to be stored is ready at stage 2 and the address to which the store is to occur is ready at stage 3.

Stage 1

Instruction fetch and start decode.

Stage 2

Fetch 2 address operands and data operand.

Latch data operand for memory controller.

Stage 3

Add address operand to form address.

Request store to memory controller.

Stage 4

No action

main:

```

ST      r1,[r2,333]
AND     r4,r5,r6
OR      r7,r8,r9
SUB     r10,r11,r12

```

	t	t+1	t+2	t+3	t+4	t+5	t+6
ST	ifetch	r2,r3, shimm	calc addr	<i>killed</i>			
AND		ifetch	r5,r6	AND	r4		
OR			ifetch	r8,r9	OR	r7	
SUB				ifetch	r11,r12	SUB	r10

Auxiliary Register Access

Accesses to the auxiliary registers 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.

Load from register (LR)

The stages perform the following in a LR instruction:

Stage 1

Instruction fetch and start decode.

Stage 2

Fetch address from operand 1.

Stage 3

Perform load from auxiliary register at address

Stage 4

Write-back the result of the load to the destination register.

main:

```

LR      r1, [r2]
AND     r4, r5, r6
OR      r7, r8, r9
SUB     r10, r11, r12
    
```

	t	t+1	t+2	t+3	t+4	t+5
LR	ifetch	r2	LR	wb to r1		
AND		ifetch	r5,r6	AND	r4	
OR			ifetch	r8,r9	OR	r7
SUB				ifetch	r11,r12	SUB

Store to register (SR)

The stages perform the following in a SR instruction:

Stage 1

Instruction fetch and start decode.

Stage 2

Fetch address from operand 1 and data from operand 2.

Stage 3

Perform store of data to auxiliary register at address extracted from operand 1.

Stage 4

No action.

main:

```
SR    r1,[r2]
AND   r4,r5,r6
OR    r7,r8,r9
SUB   r10,r11,r12
```

	t	t+1	t+2	t+3	t+4	t+5
SR	ifetch	r2, r1	SR	<i>killed</i>		
AND		ifetch	r5,r6	AND	r4	
OR			ifetch	r8,r9	OR	r7
SUB				ifetch	r11,r12	SUB

Interrupt Timings

Interrupts occur in a similar way to the branch and link instruction. However, the value that is latched into the link register is the CURRENT PC rather than NEXT_PC.

When an interrupt occurs, the instruction in instruction fetch at stage 1 is replaced by a call to the interrupt service routine.

NOTE Interrupts are not allowed to interrupt anything in a delay slot or a fetch of long immediate data.

Stage 1

Current instruction in ifetch is replaced by a branch like instruction.

The CURRENT PC is *not* updated to NEXT_PC.

Stage 2

CURRENT PC is routed to the data for next stage.

CURRENT PC is updated to the interrupt vector.

Stage 3

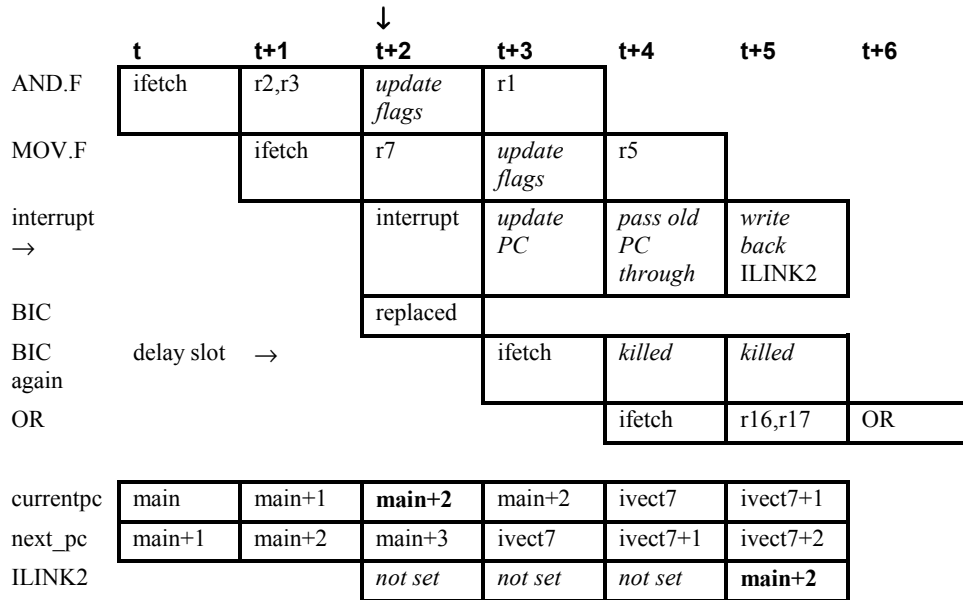
The data from stage 2 is passed to stage 4

Stage 4

The data is written to the ILINK register (the PC from stage 1)

Interrupt on arithmetic instruction

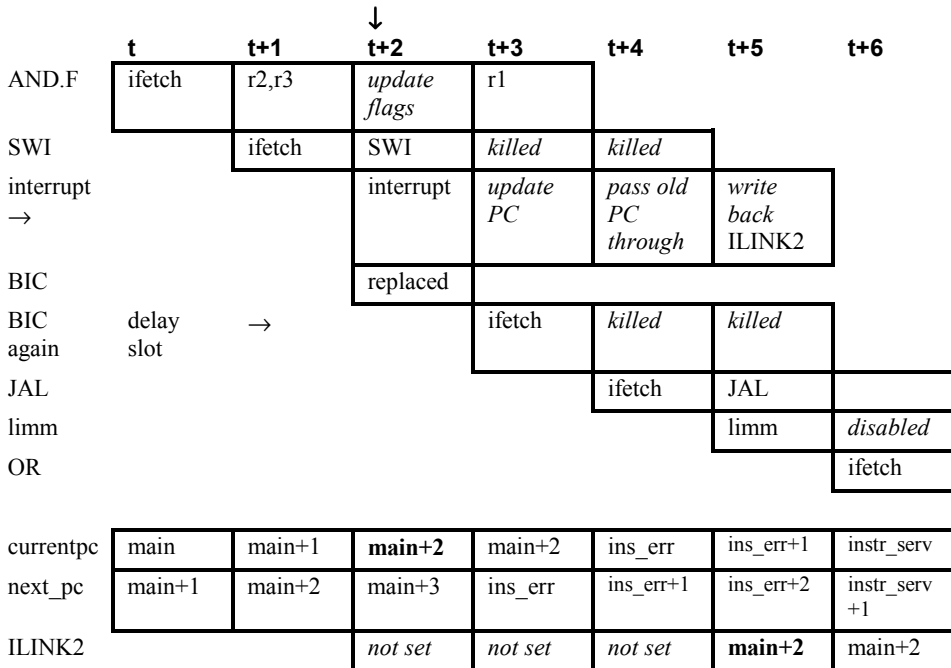
```
main:
    AND.F  r1,r2,r3
    MOV.F  r5,r7
    BIC    r8,r9,r10 ;<---- level 2 Interrupt to ivect7
    SUB    r14,r12,r13
    ...
ivect6:
    JAL    service6
ivect7:
    OR     r15,r16,r17
```



Software interrupt

The software interrupt instruction is decoded in stage two of the pipeline and if executed, then it immediately raises the instruction error exception. In this example a program execution resumes at the instruction error vector, which contains a jump to the instruction error service routine.

```
main:
    AND.F  r1,r2,r3
    SWI
    BIC     r8,r9,r10 ;----- instruction error exception
    SUB     r14,r12,r13
ins_err:
    JAL     instr_serv
instr_serv:
    OR      r15,r16,r17
```



Interrupt on jump, branch or loop set up

Because interrupts are locked out during a delay slot execution, jump, branch, loop set-up or long immediate data fetch, the instruction will either be killed because it is in stage 1 or allowed to continue because there is a delay slot in use.

main:

```
MOV    r5,r7 ;r7 = jaddr
JAL.D  [r5]   ;<---- level 2 Interrupt to ivect7
SUB    r14,r12,r13
...
```

jaddr:

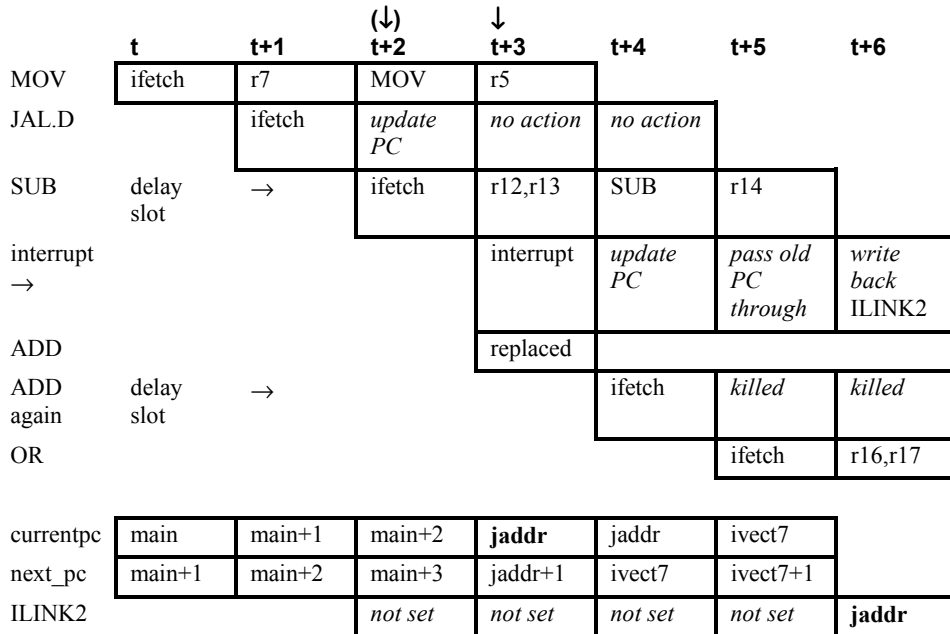
```
ADD    r20,r21,r22
...
```

ivect6:

```
JAL    service6
```

ivect7:

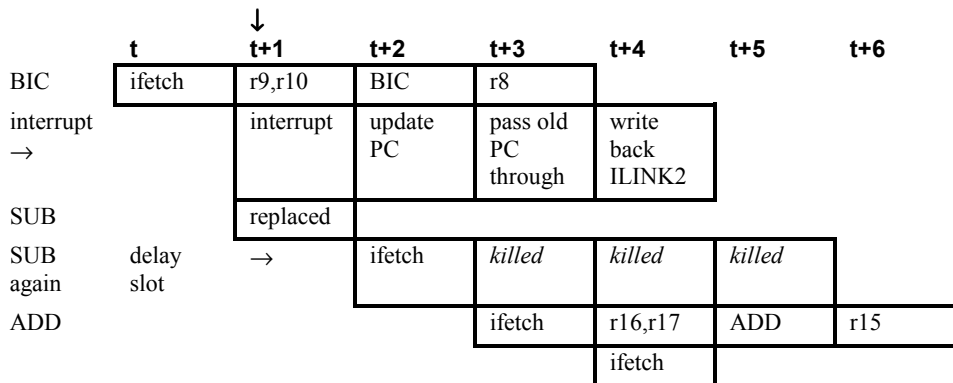
```
OR     r15,r16,r17
```



Interrupt on loop execution

At the end of a loop the NEXT_PC is compared with the LOOP_END value. If this comparison is true then, if LP_COUNT is not 1, then CURRENT PC becomes LP_START. When an interrupt occurs during this comparison-update stage the link register (ILINK2) becomes CURRENT PC. In order to stop LP_COUNT from double decrementing, the loop count decrement mechanism must be disabled during interrupt for 2 cycles.

```
main:
    AND.F    r1,r2,r3
    MOV      r5,r7
    LP       loop1
    BIC      r8,r9,r10
    SUB      r14,r12,r13;<----- level 2 interrupt to ivect7
loop1:
    OR       r1,r2,r3
    ...
ivect6:
    JAL      service6
ivect7:
    ADD      r15,r16,r17
```



currentpc	main+3	main+4	main+4	ivect7
next_pc	main+4	loop1	ivect7	ivect7+1
LP_COUNT	2	<i>disabled</i>	<i>disabled</i>	2
LP_START	main+3	main+3	main+3	main+3
LP_END	loop1	loop1	loop1	loop1
ILINK2				main+4

Interrupt on load

The load instruction is treated in the same way as an arithmetic or logic instruction. However, when the result of a delayed load is ready to be written back to the core register set the load mechanism will stall whatever is about to use the write-back at stage 4. See 0 Load and Store Timings. If the interrupt is about to write to the link register, then it too will be stalled by the write-back from a delayed load.

Interrupt on store

The store instruction is treated in the same way as an arithmetic or logic instruction.

Interrupt on auxiliary register access

The auxiliary register access instructions LR and SR will be interrupted in the same way as arithmetic or logic instructions.

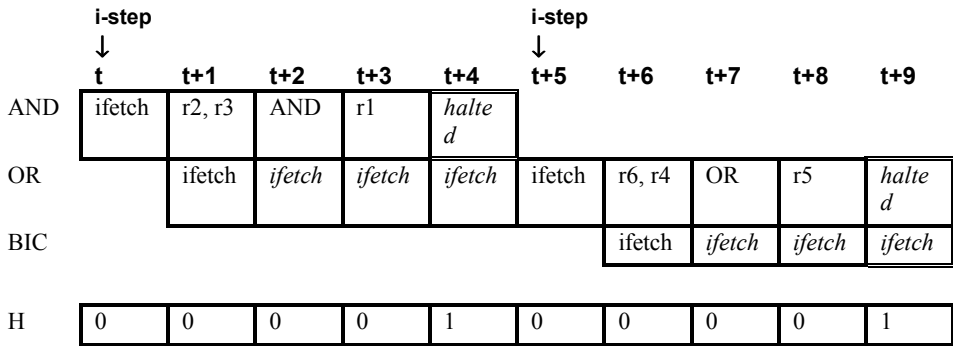
Single Instruction Step

Single cycle step simply enables the pipeline for one clock. Single instruction step, however, enables the ARCTangent-A4 pipeline until the instruction that was being fetched in stage 1 completes.

Single instruction step on single word instructions

The instruction is enabled through the 4 stages of the ARCTangent-A4 pipeline. The following instruction is fetched but held in stage 1.

```
main:
    AND    r1,r2,0x102000
    OR     r5,r6,r4
    BIC    r8,r9,r10
    SUB    r14,r12,r13
```



Single instruction step on instruction with long immediate data

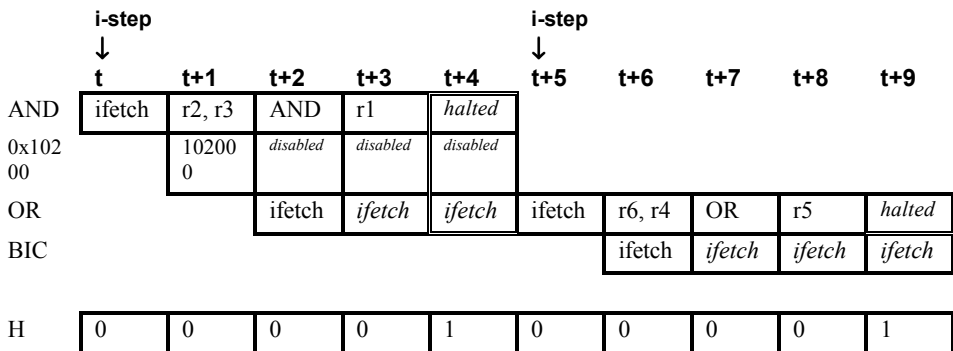
The instruction is enabled through the 4 stages of the ARctangent-A4 pipeline. The following immediate data is fetched and allowed through the pipeline. The following instruction is fetched but held in stage 1.

main:

```

AND    r1, r2, 0x102000
OR      r5, r6, r4
BIC     r8, r9, r10
SUB     r14, r12, r13

```



A

- A field, 19
- ADC, 80
- ADD, 81
- addressing mode, 16, 71
- alternate interrupt unit, 11, 27
- alternate load store unit, 11
- AND, 82
- ARC basecase version number, 65
- arithmetic operations, 29, 49
- ASL multiple, 84
- ASL/LSL, 83
- ASR, 85
- ASR multiple, 86
- auxiliary register set, 5, 139
- auxiliary registers, 48

B

- B field, 19
- barrel shift instructions, 51, 147, 150
- Bcc, 88
- BH bit, 66
- BIC, 87
- BLcc, 89
- branch address calculation, 158
- branch and jump in loops, 39, 171
- branch type instruction, 72, 77
- branches, 33
- breakpoint instruction, 43, 66, 139, 173
- BRK, 91
- byte, 16

C

- C field, 19
- code profiling, 139
- condition code field, 19

condition code register, 55
condition codes, 75

D

data organisation, 15
data-cache, 11, 47, 99, 126
debug register, 65
delay slot, 34
delayed load, 11, 47, 177
direct memory mode, 47, 99, 100, 127
dual access registers, 139
dual operand instruction, 71

E

encoding immediate data, 61
encoding instructions, 75
endianness, 16
EXT, 92
extensions, 9

- auxiliary registers, 9
- condition codes, 10
- core register, 9
- instruction set, 10

extensions, 5
extensions library, 49

F

F bit, 19
FH bit, 135
FLAG, 93
flag instruction, 30
force halt, 65

H

H bit, 135
halting ARC, 12, 93, 135, 154, 173
host interface, 133

I

I field, 19
identity register, 65, 140
immediate data indicator, 76

- instruction encoding, 75
- instruction error, 26
- instruction format, 19
- instruction layout, 19
- instruction map, 10
- instruction set summary, 29
- instruction-cache, 11
- interrupt unit, 11, 27
- interrupt vectors, 24
- interrupts, 181
- IS bit, 66, 137

J

- Jcc, 95
- JLcc, 97
- jump instruction, 72
- jumps, 33

L

- L field, 19
- LD, 99
- LD bit, 66
- link register, 23, 61
- load alignment, 16
- load and store, 46
- load instruction, 73, 74
- load pending, 65, 135
- load register, 48
- load store unit, 11
- logical operations, 29, 49
- long immediate, 16
- long immediate data and loops, 40, 172
- long word, 16
- loop construct, 35
- loop count register, 38, 61, 168
- loop end register, 63
- loop start register, 63
- loops, 33
- LP instruction, 35
- LP_COUNT, 35
- LP_END, 35, 65
- LP_START, 35, 65
- LPcc, 101
- LR, 102
- LSL, 103
- LSR, 104

LSR multiple, 105

M

manufacturer code, 65
manufacturer version number, 65
MAX, 106
memory alignment, 16
memory controller, 11, 19, 47, 141
memory endianness, 16
memory error, 26
MIN, 107
MIN/MAX instructions, 53, 147
MOV, 108
MUL64, 109
multi cycle extension instructions, 147
multiply instruction, 50, 148
multiply scoreboard unit, 148
MULU64, 111

N

N field, 19
NOP, 113
NORM, 114
normalize instruction, 51, 147
null instruction, 30

O

operand size, 15
OR, 116
orthogonal, 5

P

pipecleaning, 136
pipeline, 47, 141
pipeline cycle diagram, 142
pipeline stall, 47, 148, 177
power management features, 12
program counter, 55, 63

Q

Q field, 19

R

- register extensions, 62, 67
- register set, 5
- reset, 26
- RISC, 5
- RLC, 117
- ROL, 118
- ROR, 119
- ROR multiple, 120
- rotate instructions, 30
- RRC, 121

S

- SBC, 122
- scoreboard unit, 11, 47, 176
- self halt, 65
- semaphore register, 63, 139
- SEX, 123
- SH bit, 66
- short immediate, 16
- short immediate addressing, 29, 49, 77
- single cycle extension instructions, 147
- single instruction loops, 37, 166
- single instruction step, 66, 138, 186
- single operand instructions, 30, 72
- single step, 65, 137
- SLEEP, 124
- sleep instruction, 44, 66, 174
- software breakpoints, 12, 139, 160, 172
- software interrupt, 46, 130
- SR, 125
- SS bit, 66
- ST, 126
- starting ARC, 135
- status register, 55, 63
- store alignment, 16
- store instruction, 73, 74
- store register, 48
- SUB, 128
- SWAP, 129
- swap instruction, 52, 147
- SWI, 46, 130

T

timings

- arithmetic and logic functions, 143
- barrel shift, 150
- branch, 158
- branch and link, 156, 161
- conditional branch, 159
- conditional jump, 154
- conditional loop instruction, 164
- interrupt, 181
- jump and execute delay slot, 152
- jump and nullify delay slot, 152
- jump with immediate address, 153
- load, 176
- loop execution, 165
- loop instruction, 162
- multiply, 148
- store, 178
 - with immediate data, 144
- transfer of data, 46

W

- word, 16

X

- XOR, 130, 131

Z

- zero delay loops, 35, 61
- ZZ bit, 66