

# **Nios Embedded Processor**

# **32-Bit Programmer's Reference Manual**



101 Innovation Drive San Jose, CA 95134 (408) 544-7000 http://www.altera.com

MNL-NIOS32PROG-3.1

Document Version: 3.1 Document Date: January 2003

Copyright © 2003 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor

products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



# **About this Manual**



This manual provides comprehensive information about the Altera<sup>®</sup> Nios<sup>®</sup> 32-bit CPU.

The terms Nios processor or Nios embedded processor are used when referring to the Altera soft core microprocessor in a general or abstract context. The term Nios CPU is used when referring to the specific block of logic, in whole or part, that implements the Nios processor architecture.

Table 1 shows this manual's revision history.

Table 1. Revisio	n History
Date	Description
January 2003	Updated PDF and printed manual for Nios CPU v3.0. Includes changes for instruction cache, data cache, and the Nios on- chip instrumentation (OCI) debug core.
April 2002	Updated PDF - version 2.1
January 2002	Printed manual and PDF- version 2.0
July 2001	Updated PDF
March 2001	Printed manual and PDF- version 1.1

# How to Find Information

- The Adobe Acrobat Find feature allows you to search the contents of a PDF file. Click the binoculars toolbar icon to open the Find dialog box.
- Bookmarks serve as an additional table of contents.
- Thumbnail icons, which provide miniature previews of each page, provide a link to the pages.
- Numerous links, shown in green text, allow you to jump to related information.

# How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at http://www.altera.com.

For technical support on this product, go to http://www.altera.com/mysupport. For additional information about Altera products, consult the sources shown in Table 2.

Table 2. How to Contact Altera												
Information Type	USA & Canada	All Other Locations										
Product literature	http://www.altera.com	http://www.altera.com										
Altera literature services	lit_req@altera.com (1)	lit_req@altera.com (1)										
Non-technical customer service	(800) 767-3753	(408) 544-7000 (7:30 a.m. to 5:30 p.m. Pacific Time)										
Technical support	(800) 800-EPLD (3753) (7:30 a.m. to 5:30 p.m. Pacific Time)	(408) 544-7000 <i>(1)</i> (7:30 a.m. to 5:30 p.m. Pacific Time)										
	http://www.altera.com/mysupport/	http://www.altera.com/mysupport/										
FTP site	ftp.altera.com	ftp.altera.com										

#### Note:

(1) You can also contact your local Altera sales office or sales representative.

# Typographic Conventions

The *Nios 32-Bit Programmer's Reference Manual* uses the typographic conventions shown in Table 3.

Table 3. Conventions											
Visual Cue	Meaning										
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: <b>Save As</b> dialog box.										
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: f <sub>MAX</sub> , \QuartusII directory, d: drive, chiptrip.gdf file.										
Bold italic type	Book titles are shown in bold italic type with initial capital letters. Example: <b>1999 Device Data Book</b> .										
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75</i> ( <i>High-Speed Board Design</i> ).										
Italic type	Internal timing parameters and variables are shown in italic type. Examples: $t_{PIA}$ , $n + 1$ . Variable names are enclosed in angle brackets (<>) and shown in italic type. Example: <i><file name=""></file></i> , <i><project name=""></project></i> . <b>pof</b> file.										
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.										
"Subheading Title"	References to sections within a document and titles of Quartus II Help topics are shown in quotation marks. Example: "Configuring a FLEX 10K or FLEX 8000 Device with the BitBlaster <sup>TM</sup> Download Cable."										
Courier type	Signal and port names are shown in lowercase Courier type. Examples: data1, tdi, input. Active-low signals are denoted by suffix n, e.g., resetn.										
	Anything that must be typed exactly as it appears is shown in Courier type. For example: c:\quartusII\qdesigns\tutorial\chiptrip.gdf. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword SUBDESIGN), as well as logic function names (e.g., TRI) are shown in Courier.										
1., 2., 3., and a., b., c.,	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.										
	Bullets are used in a list of items when the sequence of the items is not important.										
$\checkmark$	The checkmark indicates a procedure that consists of one step only.										
Ĩ	The hand points to information that requires special attention.										
4	The angled arrow indicates you should press the Enter key.										
••••	The feet direct you to more information on a particular topic.										





# Contents

About this Manual	iii
How to Find Information	iii
How to Contact Altera	iv
Typographic Conventions	v
Overview	
Introduction	
Audience	
Nios CPU Overview	
Instruction Set	
Register Overview	
General-Purpose Registers	
K Register	
%r0 (%g0) Register	16
Program Counter	
Control Registers	
STATUS (%ctl0)	
ISTATUS (%ctl1)	
WVALID (%ctl2)	
ICACHE (%ctl5)	20
CPU_ID (%ctl6)	20
DCACHE (%ctl7)	20
CLR_IE (%ctl8)	21
SET_IE (%ctl9)	21
Memory Access Overview	21
Reading from Memory (or Peripherals)	22
Writing to Memory (or Peripherals)	23
Cache Memory	24
Initializing Cache Memory	
Bypassing the Data Cache when Reading Peripherals	27
Addressing Modes	27
5/16-bit Immediate Value	27
Full Width Register-Indirect	
Partial Width Register-Indirect	
Full Width Register-Indirect with Offset	
Partial Width Register-Indirect with Offset	
Program-Flow Control	
Relative-Branch Instructions	
Absolute-Jump Instructions	

Trap Instructions	
Conditional Instructions	
Exceptions	
Exception Handling Overview	
Exception Vector Table	34
External Hardware Interrupt Sources	
Internal Exception Sources	35
Register Window Underflow	35
Register Window Overflow	
Direct Software Exceptions (TRAP Instructions)	
Exception Processing Sequence	
Register Window Usage	
Status Preservation: ISTATUS Register	
Return Address	39
Simple & Complex Exception Handlers	40
Simple Exception Handlers	40
Complex Exception Handlers	40
Pipeline Implementation	41
Direct CWP Manipulation	41
Branch Delay Slots	
32-Bit Instruction Set	43
ABS	
ADD	53
ADDI	54
AND	55
ANDN	56
ASR	57
ASRI	58
BGEN	59
BR	60
BSR	61
CALL	62
СМР	63
CMPI	64
EXT16D	
EXT16S	66
EXT8D	
EXT8S	68
FILL16	69
FILL8	70
IF0	
IF1	72
IFRNZ	
IFRZ	74
IFS	

JMP	
LD	
LDP	
LDS	
LRET	
LSL	
LSLI	
LSR	
LSRI	
MOV	
MOVHI	
MOVI	
MSTEP	
MUL	
NEG	
NOP	
NOT	
OR	
PFX	
PFXIO	
RDCTL	
RESTORE	
RET	
RLC	
RRC	
SAVE	
SEXT16	
SEXT8	
SKP0	
SKP1	
SKPRNZ	
SKPRZ	
SKPS	
ST	
ST16D	110
ST16S	
ST8D	112
ST8S	113
STP	114
STS	115
STS16S	116
ST58S	
SUB	
SUBI	
SWAP	
ΤΡΔΡ	
111/111	141

TRET	
USR0	
USRx [x = 1.2.3.or 4]	
WRCTL	
XOR	
Index	



# **List of Tables**

Table 1.	Revision History	iii
Table 2.	How to Contact Altera	. iv
Table 3.	Conventions	v
Table 4.	Nios CPU Architecture	. 13
Table 5.	Register Groups	. 14
Table 6.	Programmer's Model	. 15
Table 7.	Condition Code Flags	. 19
Table 8.	Typical 32-bit Nios CPU Program/Data Memory at Address 0x0100	. 21
Table 9.	N-bit-wide Peripheral at Address 0x0100 (32-bit Nios CPU)	. 21
Table 10.	Instructions Using 5/16-bit Immediate Values	. 28
Table 11.	Instructions Using Register-Indirect Addressing	. 29
Table 12.	Instructions Using Partial Width Register-Indirect Addressing	. 30
Table 13.	Instructions Using Full Width Register-Indirect with Offset Addressing	. 30
Table 14.	Instructions Using Partial Width Register-Indirect with Offset Addressing	. 31
Table 15.	BR Branch Delay Slot Example	. 42
Table 16.	Notation Details	. 44
Table 17.	Instruction Format	. 45
Table 18.	32-bit Opcode Table	. 47
Table 19.	GNU Compiler/Assembler Pseudo-Instructions	. 50
Table 20.	Nios Operators	. 50
Table 21.	Smallest Nios Register File	. 51





# Overview

# Introduction

This document describes the 32-bit variant of the Nios embedded processor. The Nios embedded processor is a soft core CPU optimized for Altera programmable logic devices and system-on-a-programmable chip (SOPC) integration. It is a configurable, general-purpose RISC processor that can be combined with user logic and programmed into an Altera programmable logic device (PLD). The Nios CPU can be configured for a wide range of applications. A 32-bit Nios CPU core with external flash program storage and large external main memory is a powerful 32-bit embedded processor system.

### Audience

This reference manual is for software and hardware engineers creating Nios processor-based systems. This manual assumes you are familiar with electronics, microprocessors, and assembly language programming. To become familiar with the conventions used with the Nios CPU, see Table 13 on page 44.

# Nios CPU Overview

The Nios CPU is a pipelined, single-issue RISC processor in which most instructions run in a single clock cycle. The Nios instruction set is targeted for compiled embedded applications. The 32-bit Nios CPU has a word size of 32 bits. In the context of the Nios processor, byte refers to an 8-bit quantity, half-word refers to a 16-bit quantity, and word refers to a 32-bit quantity. The Nios family of soft core processors includes 32-bit and 16-bit architecture variants as shown in Table 1.

Nios CPU Details	32-bit Nios CPU	16-bit Nios CPU
Data bus size (bits)	32	16
ALU width (bits)	32	16
Internal register width (bits)	32	16
Address bus size (bits)	32	16
Instruction size (bits)	16	16

#### Table 1. Nios CPU Architecture

Nios development kits ship with the GNUPro compiler and debugger from Cygnus, an industry-standard open-source C/C++ compiler, linker and debugger toolkit. The GNUPro toolkit includes a C/C++ compiler, macro- assembler, linker, debugger, binary utilities, and libraries.

# Instruction Set

**ON Set** The Nios instruction set is tailored to support programs compiled from C and C++. It includes a standard set of arithmetic and logical operations, and instruction support for bit operations, byte extraction, data movement, control flow modification, and conditionally executed instructions, which can be useful in eliminating short conditional branches.

Register Overview This section describes the organization of the Nios CPU general-purpose registers and control registers. The Nios CPU architecture has a large general-purpose register file, several machine-control registers, a program counter, and the K register used for instruction prefixing.

### **General-Purpose Registers**

The general-purpose registers are 32 bits wide in the 32-bit Nios CPU. The register file size is configurable and contains a total of either 128, 256, or 512 registers. The software can access the registers using a 32-register-long sliding window that moves with a 16-register granularity. This sliding window can traverse the entire register file and provides access to a subset of the register file.

The register window is divided into four even sections as shown in Table 2. The lowest eight registers (%r0-%r7) are *Global* registers, also known as %g0-%g7. These *Global* registers do not change with the movement or position of the window, but remain accessible as (%g0-%g7). The top 24 registers (%r8-%r31) in the register file are accessible through the current window.

Table 2. Register Groups			
<i>In</i> registers	%r24-%r31	or	%i0-%i7
Local registers	%r16-%r23	or	%L0-%L7
<i>Out</i> registers	%r8-%r15	or	%00-%07
<i>Global</i> registers	%r0-%r7	or	%g0-%g7

The top eight registers (%i0-%i7) are known as *In* registers, the next eight (%L0-%L7) as *Local* registers, and the other eight (%o0-%o7) are known as *Out* registers. When a register window moves down 16-registers (as it does for a SAVE instruction), the *Out* registers become the *In* registers of the new window position. Also, the *Local* and *In* registers of the last window position become inaccessible. See Table 3 on page 15 for more detailed information.

Ta	abl	e 3. Pi	rol	gramı	ner	's I	Mod	del																														
			Ī	31 30	29	28	27	26	25	24	23	22	21	20	) 19	18	17	16	15	5 1	14	13	12	11	1	10	9	8	7		6	5	T	4	3	2	1	0
	%i	7 %r3	1													SA	VED	) ret	urn-	-ac	ddre	ess																
	%i	6 %r3	0													%	sfp-	-frar	ne j	ро	inte	er																
	%i	5 %r2	9																																			
1	%i	4 %r2	8																																			
Ν	%i	3 %r2	7																																			
	%i	2 %r2	6																																			
	%i	1 %r2	5																																			
	%i	0 %r2	4																																			
	%L	.7 %r2	3																																			
	%L	.6 %r2	2																																			
L	%L	.5 %r2	1																																			
0	%L	.4 %r2	о																																			
A	%L	.3 %r1	9									Ba	se-p	oin	ter 3	for \$	STF	/LDI	Р (0	or ç	gen	era	ıl-p	urpo	ose	e loo	cal)											
L	%L	.2 %r1	8									Ba	se-p	oin	ter 2	for	STF	/LDI	Р (0	or (	gen	era	ıl-p	urpo	ose	e loo	cal)											
	%L	.1 %r1	7									Ba	se-p	oin	ter 1	for	STF	/LDI	Р (0	or ç	gen	era	ıl-p	urpo	ose	e loo	cal)											
	%L	.0 %r1	6									Ba	se-p	oin	ter 0	for	STF	P/LDI	Р (0	or ç	gen	era	ıl-p	urpo	ose	e loo	cal)											
	%0	o7 %r1	5													cui	rren	t reti	ırn-	ac	ldre	ess																
	%0	6 %r1	4													9	∕₀sp	-Sta	ck F	Poi	inte	r																
	%0	5 %r1	3																																			
0	%0	04 %r1	2																																			
U T	%c	o3 %r1	1																																			
	%c	o2 %r1	оÌ																																			
	%0	o1 %r9	9																																			
	%c	0 %r8	3																																			
	%0	j7 %r7	7																																			
G	%g	,6 %r€	3																																			
Ľ	%g	,5 %r5	5																																			
0	%0	j4 %r4	1																																			
A	%g	j3 %r3	3																																			
L	%g	j2 %r2	2																																			
	%g	<b>j1 %r</b> 1	1																																			
	%g	j0 %r0	) I																																			
				31																					1	10												0
			_																											Κ	RE	GIS	ST	ΈR				
																		P	С																			
%ct	19	SET_IE		Any wi	ite (V	VRC	CTL)	оре	erati	ion t	to th	nis re	egist	ter s	sets	STA	TUS	S[15]	(IE	E) =	= 1.	Re	su	lt of	an	y re	ead	ор	erat	tior	n (l	RDO	СТ	Ľ) i	s ui	ndet	fine	d.
%ct	18	CLR_IE		Any wr	ite (V	VRC	CTL)	ope	erati	ion t	to th	nis re	egist	ter	lear	s ST	ATI	US[1	5] (	IE)	) =	0. F	Res	sult o	of a	any	rea	ad c	per	ati	ion	(RI	DC	CTL	) is	unc	lefi	ned.
%ct	17	DCACH	Е												data	cad	che	(DC)	ACH	HE	i) in	val	ida	te														
%ct	16	CPU_ID	,															CP	J IC	D	-																	
%ct	15	ICACHE												in	struc	tion	cac	he (	CA	C	HE)	in	vali	date	Э													
%ct	14	_															_	rese	rve	d -	_																	
%ct	13	_															_	rese	rve	d -	_																	
%ct	12	WVALIE	)																	-						1		н	L	MI	т		Τ		LO	LIN	лт	
%r1	11	ISTATI	s														1									Sav	/ed	Sta	tus				-					
%ct	10	STATUS	~														DC	: IC	IF				11	PRI		241	Ju	0.0		C	WF	>		Т	N	v	z	С
/00			-	31 30	29	28	27	26	25	24	23	22	21	20	) 19	18	17	16	15	; ; 1	14	13	12	111	111	10	9	8	7	Ť	6	5	T.	4	3	2	1	0
					1 <sup>-</sup>	-		-	-	1.11	1	1	1	1		1	1.1	1.5		1		-	1.17	1.	1	- 1	-		1.1	1	-	1 <sup>-</sup>	1		-			1 -

#### **K** Register

The K register is an 11-bit prefix value and is set to 0 by every instruction except PFX or PFXIO. A PFX or PFXIO instruction sets K directly from the IMM11 instruction field. The K register contains a non-zero value only for an instruction immediately following PFX or PFXIO.

A PFX or PFXIO instruction disables interrupts for one cycle, so the twoinstruction PFX or PFXIO sequence is an atomic CPU operation. Also, PFX or PXFIO sequence instruction pairs are skipped together by SKP-type conditional instructions.

The K register is not directly accessed by software, but is used indirectly. A MOVI instruction, for example, transfers all 11 bits of the K register into bits 15..5 of the destination register. This K-reading operation only yields a non-zero result when the previous instruction is PFX with a non-zero argument.

### %r0 (%g0) Register

This register is explicitly used as an argument or result for the instructions: STS16S, STS8S, ST8S, ST16S, ST8D, ST16D, FILL8, FILL16, MSTEP, and USR1-USR4.

#### **Program Counter**

The program counter (PC) register contains the byte-address of the currently executing instruction. Since all instructions must be half-word-aligned, the least-significant bit of the PC value is always 0.

The PC increments by two (PC  $\leftarrow$  PC + 2) after every instruction unless the PC is explicitly set. The BR, BSR, CALL, JMP, LRET, RET and TRET instructions modify the PC directly.

#### **Control Registers**

There are five defined control registers that are addressed independently from the general-purpose registers. The RDCTL and WRCTL instructions are the only instructions that can read or write to these control registers (meaning %ctl0 is unrelated to %g0).



17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DC	IC	E	IPRI								CWP			Ν	V	Z	С

#### Data Cache Enable (DC)

DC is the data-cache enable bit. This bit is present in Nios CPUs that have a data cache. When DC is 0, the data cache is disabled, and load instructions (LD, LDP, and LDS) act as if there is no cache memory. When DC is 1, load instructions check whether the data cache contains the addressed value. If the cache contains the addressed value ("hits"), the value from the cache is used instead of accessing the memory or peripherals. If the cache does not contain the addressed value ("misses"), the CPU accesses the desired memory or peripheral. Using a data cache can improve performance of systems with slow memories. Systems with fast, pipelined memories can improve as well. See "Cache Memory" on page 24 for more information.

When the CPU is reset, the data cache is disabled and DC is set to 0.

You must initialize the data cache before enabling it. See "DCACHE (%ctl7)" on page 20.

#### Instruction Cache Enable (IC)

IC is the instruction-cache enable bit. This bit is present in CPUs that have an instruction cache. When IC is 0, the instruction cache is disabled, and instruction fetches act as if there is no cache memory. When IC is 1, instruction fetches check whether the instruction cache contains the addressed instruction. If the cache contains the addressed instruction ("hits"), the instruction from the cache is used instead of accessing the memory. If the cache does not contain the addressed instruction ("misses"), the CPU fetches the instruction from memory. Using an instruction cache can improve performance of systems with slow memories. Systems with fast, pipelined memories can improve as well.

When the CPU is reset, the instruction cache is disabled and IC is set to 0.

You must initialize the instruction cache before enabling it. See "ICACHE (%ctl5)" on page 20.

#### **Interrupt Enable (IE)**

IE is the interrupt enable bit. When IE = 1, it enables external interrupts and internal exceptions. IE = 0 disables external interrupts and exceptions. Software TRAP instructions still execute normally even when IE = 0. You can set IE directly without affecting the rest of the STATUS register by writing to the SET\_IE (%ctl9) and CLR\_IE (%ctl8) control registers. When the CPU is reset, IE is set to 0 (interrupts disabled).

#### **Interrupt Priority (IPRI)**

IPRI contains the current running interrupt priority. When an exception is processed, the IPRI value is set to the exception number. See "Exceptions" on page 33 for more information. For external hardware interrupts, the IPRI value is set directly from the 6-bit hardware interrupt number. For TRAP instructions, the IPRI field is set directly from the IMM6 field of the instruction. For internal exceptions, the IPRI field is set from the predefined 6-bit exception number.

A hardware interrupt is not processed if its internal number is greater than or equal to IPRI or IE = 0. A TRAP instruction is processed unconditionally. IPRI disables interrupts above a certain number. For example, if IPRI is 3, then interrupts 0, 1 and 2 are processed, but all others (interrupts 3-63) are disabled. When the CPU is reset, IPRI is set to 63 (lowest-priority).

#### **Current Window Pointer (CWP)**

CWP points to the base of the sliding register window in the generalpurpose register file. Incrementing CWP moves the register window up 16 registers. Decrementing CWP moves the register window down 16 registers. CWP is decremented by SAVE instructions and incremented by RESTORE instructions.

Only specialized system software such as register window-management facilities should directly write values to CWP through WRCTL. Software normally modifies CWP by using SAVE and RESTORE instructions. When the CPU is reset, CWP is set to the largest valid value, HI\_LIMIT. For example, in a 256 register file size, there are 16 register windows. After reset, the WVALID register (%ct12) is set to 0x01C1 (that is, LO\_LIMIT = 1 and HI\_LIMIT = 14). See "WVALID (%ct2)" on page 19 for more information. For a 128 register option, HI\_LIMIT = 6; for 256 registers, HI\_LIMIT = 14; for 512 registers, HI\_LIMIT = 30. See Table 18 on page 51 for details.

#### **Condition Code Flags**

Some instructions modify the condition code flags. These flags are the four least significant bits of the status register as shown in Table 4.

Table 4. Condition Code Flags							
Flag	Bit	Result					
Ν	3	Sign of result, or most significant bit					
V	2	Arithmetic overflow—set if bit 31 of 32-bit result is different from sign of result computed with unlimited precision.					
Z	1	Result is 0					
С	0	Carry-out of addition, borrow-out of subtraction					

ISTATUS (%ctl1)

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ISTATUS is the saved copy of the STATUS register. When an exception is processed, the value of the STATUS register is copied into the ISTATUS register. This action allows the pre-exception value of the STATUS register to be restored before control returns to the interrupted program. See "Exceptions" on page 33 for more information. A return-from-trap (TRET) instruction automatically copies the ISTATUS register back into the STATUS register. Interrupts are disabled (IE = 0) when an exception is processed. Before re-enabling interrupts, an exception handler must preserve the value of the ISTATUS register. When the CPU is reset, ISTATUS is set to 0.

WVALID (%ctl2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNUSED					F	II_LIMI	Г		LO_LIMIT						

WVALID contains two values, HI\_LIMIT and LOW\_LIMIT. When a SAVE instruction decrements CWP from LOW\_LIMIT to LOW\_LIMIT –1 a register window underflow (exception #1) is generated. When a RESTORE instruction increments CWP from HI\_LIMIT to HI\_LIMIT +1, a register window overflow (exception #2) is generated. WVALID is configurable and may be read-only or read/write. When the CPU is reset, LO\_LIMIT is set to 1 and HI\_LIMIT is set to the highest valid window pointer ((register file size / 16) – 2).

ICACHE (%ctl5)

ICACHE is the instruction-cache's line-invalidate register. Writing an address to ICACHE invalidates the cache line that contains the addressed instruction. You must use ICACHE to initialize the instruction cache before enabling it. You must use ICACHE to inform the Nios processor's instruction cache when instructions are written to cached memory.

ICACHE is a write-only control register. Reading a value from ICACHE (by executing the sequence PFX 5; RDCTL) will produce an undefined value in the destination register.

Be aware that the instruction cache must be disabled (IC in STATUS must be 0) before writing to ICACHE.

#### CPU\_ID (%ctl6)

CPU\_ID contains a 16-bit constant value that identifies the version of Nios processor. Each released version returns a unique CPU\_ID. The CPU\_ID value is provided in the readme file distributed with each Nios processor release.

Bit 15 of CPU\_ID is always 0 for a 32-bit Nios CPU. Bits 14 through 12 are the major version number. The remaining bits are unique for each release of a major version. Bits 3 through 0 are the value 0x8 (1000 binary) for OpenCore<sup>®</sup> Plus evaluation Nios processors and a value other than 0x8 are for fully-functional processors.

#### DCACHE (%ctl7)

DCACHE is the data-cache's line-invalidate register. Writing an address to DCACHE invalidates the cache line that contains the addressed data. You must use DACHE to initialize the data cache before enabling it. Also, you can use DCACHE to inform the Nios processor's data cache that another master has written data to cached memory.

DCACHE is a write-only control register. Reading a value from DCACHE (by executing the sequence PFX 7; RDCTL) will produce an undefined value in the destination register.

Be aware that the data cache must be disabled (DC in STATUS must be 0) before writing to DCACHE. If large amounts of cache need to be invalidated, you may consider declaring the memory region volatile. This causes the C-compiler nios-elf-gcc to use PFXIO to access the data, thereby bypassing the cache and avoiding the need to invalidate.

1

CLR\_IE (%ctl8)

Any WRCTL operation to the CLR\_IE register clears the IE bit in the STATUS register (IE  $\leftarrow$  0) and the WRCTL value is ignored. A RDCTL operation from CLR\_IE produces an undefined result.

SET\_IE (%ctl9)

Any WRCTL operation to the SET\_IE register sets the IE bit in the STATUS register (IE  $\leftarrow$  1) and the WRCTL value is ignored. A RDCTL operation from SET\_IE produces an undefined result.

# Memory Access Overview

The Nios processor is little-endian. Data memory must occupy contiguous words. If the physical memory device is narrower than the word size, then the data bus should implement dynamic-bus sizing to simulate full-width data to the Nios CPU. Peripherals present their registers as word widths, padded by 0s in the most significant bits if the registers happen to be smaller than words. Table 5 and Table 6 show examples of the 32-bit Nios CPU word widths.

Table 5. Typical 32-bit Nios CPU Program/Data Memory at Address 0x0100								
Address		Contents						
	31 24	23 16	15 8	7 0				
0x0100	byte 3	byte 2	byte 1	byte 0				
0x0104	byte 7	byte 6	byte 5	byte 4				
0x0108	byte 11	byte 10	byte 9	byte 8				
0x010c	byte 15	byte 14	byte 13	byte 12				

Table 6. N-bit-wide Peripheral at Address 0x0100 (32-bit Nios CPU)								
Address	Contents							
	31 N	N-1	0					
0x0100	(zero padding)	register 0						
0x0104	(zero padding)	register 1						
0x0108	(zero padding)	register 2						
0x010c	(zero padding)	register 3						

### **Reading from Memory (or Peripherals)**

The Nios CPU can only perform aligned memory accesses. A 32-bit read operation can only read a full word starting at a byte address that is a multiple of 4. Instructions which read from memory always treat the low two bits of the address as 0. Instructions are provided for extracting particular bytes and half-words from words.

The simplest instruction that reads data from memory is the LD instruction. A typical example of this instruction is LD %g3, [%o4]. The first register operand, %g3, is the destination register, where data is loaded. The second register operand specifies a register containing an address to read from. This address is aligned to the nearest word meaning the lowest two bits are treated as if they are 0.

Quite often, however, software must read data smaller than 32 bits. The Nios CPU provides instructions for extracting individual bytes and half-words from words. The EXT8D instruction is used for extracting a byte, and the EXT16D instruction is used for extracting a half-word. A typical example of the EXT8D instruction is EXT8D %g3,%o4. The EXT8D instruction uses the lowest two bits of the second register operand to extract a byte from the first register operand, and replaces the entire contents of the first register operand with that byte. The EXT8D instruction extracts a byte as shown in Figure 1.





The assembly-language example in Code Example 1 shows how to read a single byte from memory, even if the address of the byte is not word-aligned.

#### Code Example 1. Reading a Single Byte from Memory

Contents of memory:

; 0 1 2 3 ;0x00001200 0x46 0x49 0x53 0x48 ;Instructions executed on a 32-bit Nios CPU ; Let's assume %o4 contains the address x00001202 LD %g3,[%o4] ; %g3 gets the contents of address 0x1200, ; so %g3 contains 0x48534946 EXT8D %g3,%o4 ; %g3 gets replaced with byte 2 from %g3, ; so %g3 contains 0x00000053

#### Writing to Memory (or Peripherals)

The Nios CPU can perform aligned writes to memory in widths of byte, half-word, or word. A word can be written to any address that is a multiple of 4 in one instruction. A half-word can be written to any address that is a multiple of 2 in two instructions. A byte can be written to any address in two instructions.

The lowest byte of a register can be written only to an address that is a multiple of 4; the middle-low byte of a register can be written only as an address that is a multiple of 4, plus 1, and so on.

The Nios CPU can also write the low half-word of a register to an address that is a multiple of four, and the high half-word of a register to an address which is a multiple of 4, plus 2.

The ST instruction writes a full word to a word aligned memory address from any register; the ST8D and ST16D instructions write a byte and half-word, respectively, with the alignment constraints described above, from register %r0.

Often it is necessary for software to write a particular byte or half-word to an arbitrary location in memory. The position within the source register may not correspond with the location in memory to be written. The FILL8 and FILL16 instructions takes the lowest byte or half-word, respectively, of a register and replicates it across register %r0. Code Example 2 shows how to write a single byte to memory, even if the address of the byte is not word-aligned.

#### Code Example 2. Single Byte Written to Memory—Address Not Word Aligned

;Contents of memory before: ; 0 1 2 3 ; 0x46 0x49 0x53 ;0x00001200 0x54; Let's assume %o4 contains the address 0x00001203 ; and that %g3 contains the value 0x00000BC FILL8 %r0,%g3 ; (First operand can only be %r0) ; replicate low byte of %g3 across %r0 ; so %r0 contains 0xBCBCBCBC ST8D [%o4],%r0 ; (Second operand can only be %r0) ; Stores the 3rd byte of %r0 to address 0x1203 ;Contents of memory after: ; ; 0 1 2 ;0x00001200 0x46 0x49 0x53 3 0xBC

# **Cache Memory**

The Nios CPU optionally has an instruction cache and a data cache. The data cache influences Nios memory access.

The data cache stores recently accessed data words and, whenever possible, uses the cached data value instead of performing a memory read cycle. The Nios CPU uses direct-mapped, the simplest cache implementation. This means that low bits of the data address are used to directly access the selected line of the cache memory as shown in Figure 2 on page 25. In a direct-mapped cache, data words whose addresses differ by a multiple of the cache size will be stored in the same cache line. To determine which of these words is stored in a line, the high bits of the word's address are stored as a tag along with the word's data and a valid bit.

Overview





#### Note:

 The total number of data cache lines is equal to the size of the data cache divided by four. The total number of instruction cache lines is equal to the size the size of the data cache divided by two.

When executing a load instruction (LD, LDP, or LDS), the Nios CPU compares the high bits of the load address with the selected cache line's tag. If the high bits match the tag and the line contains valid data, then the processor uses the cached data instead of reading memory, thereby accelerating processor performance. When the processor uses cached data, it is called a "hit." When the cache does not contain the desired data, it is called a "miss."

The Nios CPU uses *write-through*, the simplest cache policy. This means that all word-store instructions (ST, STP, and STS) store data to the cache and also perform a memory write cycle. The cache line that is written is determined by the same low bits of the data address that are used by word-load instructions, and so subsequent loads from the same address will hit. In addition to writing data to the cache, the high bits of the address are written as the data's tag, and the valid bit is set.

When the cache misses, the processor performs a memory read cycle, retrieves the desired data word, writes the word to the register indicated in the store instruction, and writes the data to the cache. That is, subsequent loads from the same memory address will hit.

### **Initializing Cache Memory**

You must initialize cache memory and enable it before it can be used. Initialize the data cache by writing a range of addresses to the DCACHE control register. Initialization clears the valid bits of all cache-memory lines to prevent uninitialized tag data from causing a false hit. The Data Cache Enable (DC) bit in the STATUS register must be zero during any write to the DCACHE register. Writing a value to DCACHE while DC = 1 will produce an undefined result. See"Data Cache Enable (DC)" on page 17 for more Data Cache Enable (DC) information.

Code Example 2 shows use of the cache control registers and status register to initialize and enable the instruction cache. Enabling the data cache is similar. The macros nm\_icache\_enable and nm\_icache\_disable use a combination of C and assembly language to read the status registers with RDCTL, change a single bit within it, and write it back out with WRCTL.

#### Code Example 2. Initializing Cache Memory

```
#define np_nios_icache_bit 0x00010000 // bit in control register 0
#define np_nios_dcache_bit 0x00020000 // bit in control register 0
#define np_nios_icache_reg 5
                                      // register to invalidate a line of icache
#define np_nios_dcache_reg 7 // register to invalidate a line of dcache
#define nm_icache_invalidate_line(byte_address) \
        asm("pfx 5 \n\t wrctl %0" : : "r" (byte_address));
#define nm icache enable()
        int status;
        asm("rdctl %0" : "=r" (status));
        status |= np_nios_icache_bit;
        asm("wrctl \sqrt[80]{n} \n\t nop" : : "r" (status));
#define nm_icache_disable()
        int status;
        asm("rdctl %0" : "=r" (status));
        status &= ~np_nios_icache_bit;
        asm("wrctl %0 \n\t nop" : : "r" (status));
void nr_icache_init(void)
        int i;
        nm_icache_disable();
        for(i = 0; i < nasys_icache_size; i+= nasys_icache_line_size)</pre>
                nm_icache_invalidate_line(i);
        nm_icache_enable();
        }
```

### Bypassing the Data Cache when Reading Peripherals

	Since repeated accesses to the same memory word cause the cache to hit, it would be undesirable for the data cache to intercept peripheral accesses. For example, repeated reads from a UART always load from the same data address but return different data each time. Allowing the data cache to intercept all reads after the first read would prevent proper operation. Peripheral reads need to be identified.
	Nios provides an instruction for disabling the data cache on an instruction-by-instruction basis. Any LD or LDP instruction immediately preceded by a PFXIO instruction will read data directly from memory (bypassing the cache), even if the cache is enabled. Any LD and LDP instruction, not immediately preceded by a PFXIO instruction, will use the data cache if it is enabled ( $DC = 1$ ).
	All LDS instructions always use the data cache if it is enabled. A PFXIO/LDS sequence will produce an undefined result. See "PFXIO" on page 95 for PFXIO instruction details.
	The Nios C-compiler (nios-elf-gcc) inserts PFXIO instructions as necessary to bypass the cache for any variable declared with the type-qualifier volatile. Any registers, variables, or buffers declared as volatile will not be cached.
Addressing	The topics in this section includes a description of the addressing modes:
Modes	<ul> <li>5/16-bit immediate</li> <li>Full width register-indirect</li> <li>Partial width register-indirect</li> <li>Full width register-indirect with offset</li> <li>Partial width register-indirect with offset</li> </ul>
	5/16-bit Immediate Value
	Many arithmatic and logical instructions take a 5 hit immediate value as

Many arithmetic and logical instructions take a 5-bit immediate value as an operand. The ADDI instruction, for example, has two operands: a register and a 5-bit immediate value. A 5-bit immediate value represents a constant from 0 to 31. To specify a constant value that requires from 6 to 16 bits (a number from 32 to 65,535), the 11-bit K register can be set using the PFX instruction. This value is concatenated with the 5-bit immediate value. The PFX instruction must be used directly before the instruction it modifies.

To support breaking 16-bit immediate constants into a PFX value and a 5-bit immediate value, the assembler provides the operators %hi() and %lo(). %hi(*x*) extracts the high 11 bits (bit 5..15) from constant *x*, and %lo(*x*) extracts the low 5 bits (0..4) from constant *x*.

Code Example 3 shows an ADDI instruction being used both with and without a PFX.

Code Example 3. The ADDI Instruction Used With/Without a PFX

```
; Assume %g3 contains the value 0x00000041
ADDI %g3,5 ; Add 5 to %g3
; so %g3 now contains 0x00000046
PFX %hi(0x1234) ; Load K with upper 11 bits of 0x1234
ADDI %g3,%lo(0x1234) ; Add low 5 bits of 0x1234 to %g3
; so the K register contained 0x091
; and the immediate operand of the ADDI
; instruction contained 0x14, which
; concatenated together make 0x00001234
; Now %g3 contains 0x0000127A
```

Besides arithmetic and logical instructions, several other instructions use immediate-mode constants of various widths, and the constant is not modified by the K register. See the description of each instruction in the "32-Bit Instruction Set" for a precise explanation of its operation. Table 7 shows instructions using 5/16-bit immediate values.

Table 7. Instructions Using 5/16-bit Immediate Values							
ADDI	AND <sup>(1)</sup>	ANDN <sup>(1)</sup>	ASRI				
CMPI	LSLI	LSRI	MOVI				
MOVHI	OR <sup>(1)</sup>	SUBI	XOR <sup>(1)</sup>				

Note:

AND, ANDN, OR, and XOR can only use PFX'd 16-bit immediate values. These
instructions act on two register operands if not preceded by a PFX instruction.

### Full Width Register-Indirect

The LD and ST instructions can load and store, respectively, a word to or from a register using another register to specify the address. See Table 8. The address is first aligned downward to a word-aligned address, as described in "Memory Access Overview" on page 21. The K register is treated as a signed offset, in words, from the word-aligned value of the address register. The offset range is (-4096..4092) bytes. The effective address is K (signed) x 4 + (address-register-value & 0xFFFFFFC).

Table 8. Instructions Using Register-Indirect Addressing						
Instruction	Data Register					
LD	Any	Any				
ST	Any	Any				

If the Nios processor includes a data cache, reading peripherals will require prefixing LD with PFXIO. See "Bypassing the Data Cache when Reading Peripherals" on page 27 for further information.

### Partial Width Register-Indirect

None of the 32-bit instructions read a partial word. To read a partial word, combine a full width register-indirect read instruction with an extraction instruction (EXT8D, EXT8S, EXT16D or EXT16S).

Several instructions can write a partial word. Each of these instructions has a static and a dynamic variant. The position within both the source register and the word of memory is determined by the low bits of an addressing register. In the case of a static variant, the position within both the source register and the word of memory is determined by a 1- or 2-bit immediate operand to the instruction. As with full width register-indirect addressing, the K register is treated as a signed offset in words from the word aligned value of the address register.

The partial width register-indirect instructions all use %r0 as the source of data to write as shown in Table 9. These instructions are convenient to use in conjunction with the FILL8 or FILL16 instructions.

Table 9. Instructions Using Partial Width Register-Indirect Addressing							
Instruction	Address Register	Data Register	Byte/Half-word Selection				
ST8S	Any	%r0	Immediate				
ST16S	Any	%r0	Immediate				
ST8D	Any	%r0	Low bits of address register				
ST16D	Any	%r0	Low bits of address register				

### Full Width Register-Indirect with Offset

The LDP, LDS, STP and STS instructions can load or store a full word to or from a register using another register to specify an address, and an immediate value to specify an offset, in words, from that address.

Unlike the LD and ST instructions, which can use any register to specify a memory address, these instructions may each only use particular registers for their address. The LDP and STP instructions may each only use the register %L0, %L1, %L2, or %L3 for their address registers. See Table 10. The LDS and STS instructions may only use the stack pointer, register %sp (equivalent to %o6), as their address register. These instructions each take a signed immediate index value that specifies an offset in words from the word-aligned address pointed in the address register.

Table 10. Instructions Using Full Width Register-Indirect with Offset Addressing								
Instruction	Address Register	Data Register	Offset Range without PFX or PFXIO					
LDP	%L0, %L1, %L2, %L3	Any	0124 bytes					
LDS	%sp	Any	01020 bytes					
STP	%L0, %L1, %L2, %L3	Any	0124 bytes					
STS	%sp	Any	01020 bytes					

### Partial Width Register-Indirect with Offset

There are no instructions that read a partial word from memory. To read a partial word, you may combine a full-width indexed register-indirect read instruction with an extraction instruction, EXT8D, EXT8S, EXT16D or EXT16S to write a partial word. You may use the STS8S and STS16S instructions (which use an immediate constant) to specify a byte or halfword offset, respectively, from the stack pointer to write the correspondingly aligned partial width of the source register %r0. See Table 11. These instructions may each only use the stack pointer, register %sp (equivalent to %o6), as their address register, and may only use register %r0 (equivalent to %g0, but must be called %r0 in the assembly instruction) as the data register. These instructions are convenient to use with the FILL8 or FILL16 instructions.

Table 11. Instructions Using Partial Width Register-Indirect with Offset Addressing							
Instruction	Address Register	Data Register	Byte/Half-word Selection	Index Range			
STS8S	%sp	%r0	Immediate	01023 bytes			
STS16S	%sp	%r0	Immediate	0511 half-words			

# Program-Flow Control

The topics in this section include a description of the following:

- Two relative-branch instructions (BR and BSR)
- Two absolute-jump instructions (JMP and CALL)
- Two trap instructions (TRET and TRAP)
- Five conditional instructions (SKP, SKP0, SKP1, SKPRZ and SKPRNZ)

### **Relative-Branch Instructions**

There are two relative-branch instructions: BR and BSR. The branch target address is computed from the current program-counter (that is, the address of the BR instruction itself) and the IMM11 instruction field. Details of the branch-offset computation are provided in the description of the BR and BSR instructions. BSR is identical to BR except that the return-address is saved in %o7. Details of the return-address computation are provided in the description of the BSR instruction. Both BR and BSR are unconditional. Conditional branches are implemented by preceding BR or BSR with a SKP-type instruction.

Both BR and BSR instructions have branch delay slot behavior: The instruction immediately following a BR or BSR is executed after BR or BSR, but before the instruction at the branch-target. See "Branch Delay Slots" on page 42 for more information. The branch range of the BR and BSR instructions is forward by 2048 bytes, or backwards by 2046 bytes relative to the address of the BR or BSR instruction.

### Absolute-Jump Instructions

There are two absolute (computed) jump instructions: JMP and CALL. The jump-target address is given by the contents of a general-purpose register. The register contents are left-shifted by one and transferred into the PC. CALL is identical to JMP except that the return-address is saved in %o7. Details of the return-address computation are provided in the description of the CALL instruction. Both JMP and CALL are unconditional. Conditional jumps are implemented by preceding JMP or CALL with a SKP-type instruction.

Both JMP and CALL instructions have branch delay slot behavior: The instruction immediately following a JMP or CALL is executed after JMP or CALL, but before the instruction at the jump-target. The LRET pseudo-instruction, which is an assembler alias for JMP %07, is conventionally used to return from subroutines.

### **Trap Instructions**

The Nios processor implements two instructions for software exception processing: TRAP and TRET. See "TRAP" on page 121 and "TRET" on page 122 for detailed descriptions of both these instructions. Unlike JMP and CALL, neither TRAP nor TRET has a branch delay-slot: The instruction immediately following TRAP is not executed until the exception-handler returns. The instruction immediately following TRET is not executed at all as part of TRET's operation.

### **Conditional Instructions**

There are five conditional instructions (SKPS, SKP0, SKP1, SKPRZ, and SKPRNZ). Each of these instructions has a converse assembler-alias pseudo-instruction (IFS, IF0, IF1, IFRZ, and IFRNZ, respectively). Each of these instructions tests a CPU-internal condition and then executes the next instruction or not, depending on the outcome. The operation of all five SKP-type instructions (and their pseudo-instruction aliases), are identical except for the particular test performed. In each case, the subsequent instruction is fetched from memory regardless of the test outcome. Depending on the outcome of the test, the subsequent instruction is either executed or cancelled. A cancelled instruction has no effect.



See the *Nios Embedded Processor Software Development Reference Manual* for more information about pseudo-instructions.

While SKPx and IFx type conditional instructions are often used to conditionalize jump (JMP, CALL) and branch (BR, BSR) instructions, they can be used to conditionalize any instruction. Conditionalized PFX or PFXIO instructions (PFX or PFXIO immediately after a SKPx or IFx instruction) present a special case; the next two instructions are either both cancelled or both executed. PFX or PFXIO instruction pairs are conditionalized as an atomic unit.

# **Exceptions**

The topics in this section include a description of the following:

- Exception vector table
- How external hardware interrupts, internal exceptions, register window underflow, register window overflow and TRAP instructions are handled
- Direct software exceptions (TRAP) and exception processing sequence

## **Exception Handling Overview**

The Nios processor allows up to 64 vectored exceptions. Exceptions can be enabled or disabled globally by the IE control-bit in the STATUS register, or selectively enabled on a priority basis by the IPRI field in the STATUS register. Exceptions can be generated from any of three sources: external hardware interrupts, internal exceptions or explicit software TRAP instructions.

The Nios exception-processing model allows precise handling of all internally generated exceptions. That is, the exception-transfer mechanism leaves the exception-handling subroutine with enough information to restore the status of the interrupted program as if nothing had happened. Internal exceptions are generated if a SAVE or RESTORE instruction causes a register-window underflow or overflow, respectively.

Exception-handling subroutines always execute in a newly opened register window, allowing very low interrupt latency. The exception handler does not need to manually preserve the interruptee's register contents.

The Nios processor has one non-maskable exception, interrupt priority 0, for use by the Nios on-chip instrumentation (OCI) debug module. The Nios OCI debug module is an intellectual property core designed by First Silicon Solutions (FS2) Inc. It is implemented as an FS2 OCI block that connects directly to signals internal to the Nios CPU. When triggered, this non-maskable exception interrupts execution, regardless of the values of IE or IPRI. The non-maskable exception is reserved for debug functionality, and is not accessible to users. User programs never handle non-maskable interrupts. After a non-maskable interrupt is serviced, the CPU always returns to its pre-exception status.

#### **Exception Vector Table**

The exception vector table is a set of 64 exception-handler addresses and each entry is 4 bytes. The base-address (VECBASE) of the exception vector table is configurable. For interrupt priorities 1 through 63, when the Nios CPU processes exception number *n*, the CPU fetches the *n*th entry from the exception vector table, doubles the fetched value, and then loads the result into the PC. The non-maskable interrupt 0 behaves differently and does not depend on entries in the vector table. The 0th vector table entry is unused.

The exception vector table can physically reside in RAM or ROM, depending on the hardware memory map of the target system. A ROM exception vector table does not require run-time initialization.

#### External Hardware Interrupt Sources

An external source can request a hardware interrupt by driving a 6-bit interrupt number on the Nios CPU irq\_number inputs while simultaneously asserting true (1) the Nios CPU irq input pin. In typical systems, the Nios CPU's irq and irq\_number inputs are driven by automatically-generated interconnect (bus) logic. As such, system peripherals typically have a single irq output. The automaticallygenerated bus logic converts multiple one-bit irq-sources into a single irqinput to the CPU, accompanied by an associated 6-bit irq\_number. The Nios CPU processes the indicated exception if the IE bit is true (1) and the requested interrupt number is smaller (higher priority) than the current value in the IPRI field of the STATUS register. The non-maskable exception, interrupts priority 0, is processed regardless of the value of the IE bit. Control is transferred to the exception handler whose number is given by the irq\_number inputs.

The Nios irq input is level sensitive. The irq and irq\_number inputs are sampled at the rising edge of each clock. External sources that generate interrupts should assert their irq output signals until the interrupt is acknowledged by software (such as by writing a register inside the interrupting peripheral to 0). Interrupts that are asserted and then de-asserted before the Nios CPU core can begin processing the exception are ignored.

### **Internal Exception Sources**

There are two sources of internal exceptions: register window-overflow and register window-underflow. The Nios processor architecture allows precise exception handling for all internally generated exceptions. In each case, it is possible for the exception handler to service the exceptional condition and resume normal execution of the interrupted program.

#### Register Window Underflow

The register window underflow exception is exception number 1. A register window-underflow exception occurs whenever the lowest valid register window is in use (CWP = LO\_LIMIT) and a SAVE instruction is issued. The SAVE instruction moves CWP below LO\_LIMIT and %sp is set per the normal operation of SAVE. A register window underflow exception is generated, which transfers control to an exception-handling subroutine before the instruction following SAVE is executed.

When a SAVE instruction causes a register window underflow exception, CWP is decremented only once before control is passed to the exceptionhandling subroutine. The CPU does not process a register window underflow exception if interrupts are disabled (IE = 0) or the current value in IPRI is less than or equal to 1.

The action taken by the underflow exception-handler subroutine depends upon the requirements of the system. For systems running larger or more complex code, the underflow (and overflow) handlers can implement a virtual register file that extends beyond the limits of the physical register file. When an underflow occurs, the underflow handler may, for example, save the current contents of the entire register file to memory and re-start CWP back at HI\_LIMIT, allowing room for code to continue opening register windows. Many embedded systems, on the other hand, might wish to tightly control stack usage and subroutine call-depth. Such systems might implement an underflow handler that prints an error message and exits the program.

The programmer determines the nature of and actions taken by the register window underflow exception handler. The Nios software development kit (SDK) includes, and automatically installs by default, a register window underflow handler that virtualizes the register file using the stack as temporary storage.

A register window underflow exception can only be generated by a SAVE instruction. Directly writing CWP (via a WRCTL instruction) to a value less than LO\_LIMIT does not cause a register window underflow exception. Executing a SAVE instruction when CWP is already below LO\_LIMIT does not generate a register window underflow exception.

#### Register Window Overflow

The register window overflow exception is exception number 2. A register window overflow exception occurs whenever the highest valid register window is in use (CWP = HI\_LIMIT) and a RESTORE instruction is issued. Control is transferred to an exception-handling subroutine before the instruction following RESTORE is executed.

When a register window overflow exception is taken, the exception handler sees CWP at HI\_LIMIT. You can think of CWP being incremented by the RESTORE instruction, but then immediately decremented as a consequence of normal exception processing.

The action taken by the overflow exception handler subroutine depends upon the requirements of the system. For systems running larger or more complex code, the overflow (and underflow) handlers can implement a virtual register file that extends beyond the limits of the physical register file. When an overflow occurs, such an overflow handler may, for example, reload the entire contents of the physical register file from the stack and restart CWP back at LO\_LIMIT. Many embedded systems, on the other hand, might wish to tightly control stack usage and subroutine call depth. Such systems might implement an overflow handler that prints an error message and exits the program.
The programmer determines the nature of any actions taken by the register window overflow exception handler. The Nios SDK automatically installs by default a register window overflow handler which virtualizes the register file using the stack.

A register window overflow exception can only be generated by a RESTORE instruction. Directly writing CWP (via a WRCTL instruction) to a value greater than HI\_LIMIT does not cause a register window overflow exception. Executing a RESTORE instruction when CWP is already above HI\_LIMIT does not generate a register window overflow exception.

#### **Direct Software Exceptions (TRAP Instructions)**

Software can directly request that control be transferred to an exception handler by issuing a TRAP instruction. The IMM6 field of the instruction gives the exception number. TRAP instructions are always processed, regardless of the setting of the IE or IPRI bits. TRAP instructions do not have a delay slot. The instruction immediately following a TRAP is not executed before control is transferred to the indicated exception-handler. A reference to the instruction following TRAP is saved in %o7, so a TRET instruction transfers control back to the instruction following TRAP at the conclusion of exception processing.

#### **Exception Processing Sequence**

When an exception is processed from any of the above sources, the following sequence occurs:

- 1. The contents of the STATUS register are copied into the ISTATUS register.
- 2. CWP is decremented, opening a new window for use by the exception-handler routine (This is not the case for register window underflow exceptions, where CWP was already decremented by the SAVE instruction that caused the exception).
- 3. IE is set to 0, disabling interrupts.
- 4. IPRI is set with the 6-bit number of the exception.
- 5. The address of the next non-executed instruction in the interrupted program is transferred into %07.
- 6. The start-address of the exception handler is fetched from the exception vector table and written into the PC.

7. After the exception handler finishes, a TRET instruction is issued to return control to the interrupted program.

#### Register Window Usage

All exception processing starts in a newly opened register window. This process decreases the complexity and latency of exception handlers because they are not responsible for maintaining the interruptee's register contents. An exception handler can freely use registers %00..%05 and %L0..%L7 in the newly opened window. An exception handler should not execute a SAVE instruction upon entry. The use of SAVE and RESTORE from within exception handlers is discussed later.

Because the transfer to an exception handler always opens a new register window, programs must always leave at least one register window available for exceptions. Setting LO-LIMIT to greater than zero guarantees that a new register window is available for exceptions. For example, whenever a program executes a SAVE instruction that would then use up the last register window (CWP = 0), a register-underflow trap is generated. The register-underflow handler itself executes in the final window (with CWP = 0).

Correctly written software never processes an exception when CWP is 0. CWP should be 0 only when an exception is being processed. New exception handlers must take certain well-defined precautions before re-enabling interrupts. See "Simple & Complex Exception Handlers" on page 40 for more information.

If the Nios OCI debug module is enabled in the Nios CPU core, the reset value for LO\_LIMIT is 2; otherwise, the reset value for LO\_LIMIT is 1. Safe usage of the Nios OCI debug module requires that LO\_LIMIT be 2, because the non-maskable exception must always have a register window available. For example, a program executing with CWP = 2 (LO\_LIMIT) may be interrupted, decrementing CWP to 1 (less than LO\_LIMIT) and transferring execution to the register window underflow interrupt service routine. Before this service routine completes, it could be interrupted by the non-maskable exception, decrementing CWP to 0. The non-maskable exception service routine can then execute safely in the last available register window with CWP = 0.

#### Status Preservation: ISTATUS Register

When an exception occurs, the interruptee's STATUS register is copied into the ISTATUS register. The STATUS register is then modified (IE set to 0, IPRI set, CWP decremented). The original contents of the STATUS register are preserved in the ISTATUS register. When exception processing returns control to the interruptee, the original program's STATUS register contents are restored from ISTATUS by the TRET instruction.

Interrupts are automatically disabled upon entry to an exception handler, so there is no danger of ISTATUS being overwritten by a subsequent interrupt or exception. The case of nested exception handlers (exception handlers that use or re-enable exceptions) is discussed in detail below. Nested exception handlers must explicitly preserve, maintain, and restore the contents of the ISTATUS register before and after enabling subsequent interrupts.

When the non-maskable exception (TRAP 0) is triggered, both STATUS and ISTATUS are saved. ISTATUS is saved because the non-maskable exception can interrupt an exception handler in progress. After the nonmaskable interrupt is serviced, the CPU returns to its pre-exception status, and STATUS and ISTATUS are restored.

#### **Return Address**

When an exception occurs, execution of the interrupted program is temporarily suspended. The instruction in the interrupted program that was preempted (that is, the instruction that would have executed, if the exception had not occurred) is taken as the return location for exception processing.

The return location is saved in %o7 (in the exception handler's newly opened register window) before control is transferred to the exception handler. The value stored in %o7 is the byte address of the return-instruction right-shifted by one place. This value is suitable directly for use as the target of a TRET instruction without modification. Exception handlers usually execute a TRET %o7 instruction to return control to the interrupted program.

#### Simple & Complex Exception Handlers

The Nios processor architecture permits efficient, simple exception handlers. The hardware itself accomplishes much of the status- and register-preservation overhead required by an exception handler. Simple exception handlers can substantially ignore all automatic aspects of exception handling. Complex exception handlers (such as nested exception handlers) must follow additional precautions.

#### Simple Exception Handlers

An exception handler is considered simple if it obeys the following rules:

- It does not re-enable interrupts.
- It does not use SAVE or RESTORE (either directly or by calling subroutines that use SAVE or RESTORE).
- It does not use any TRAP instructions (or call any subroutines that use TRAP instructions).
- It does not alter the contents of registers %g0..%g7, or %i0..%i7.

Any exception handler that obeys these rules need not take special precautions with ISTATUS or the return address in %o7. A simple exception handler need not be concerned with CWP or register-window management.

#### Complex Exception Handlers

An exception handler is considered complex if it violates any of the requirements of a simple exception handler, listed above. Complex exception handlers may allow nested exception handling and the execution of more complex code (such as subroutines that SAVE and RESTORE). A complex exception handler has the following additional responsibilities:

- It must preserve the contents of ISTATUS before re-enabling interrupts. For example, ISTATUS could be saved on the stack.
- It must check CWP before re-enabling interrupts to be sure CWP is at or above LO\_LIMIT. If CWP is below LO\_LIMIT, it must take an action to open up more available register windows (such as save the register file contents to RAM), or it must signal an error.
- It must re-enable interrupts subject to the above two conditions before executing any SAVE or RESTORE instructions or calling any subroutines that execute any SAVE or RESTORE instructions.
- Prior to returning control to the interruptee, it must restore the contents of the ISTATUS register, including any adjustments to CWP if the register-window has been deliberately shifted.

Prior to returning control to the interruptee, it must restore the contents of the interruptee's register window.

#### Pipeline Implementation

The Nios CPU is a pipelined RISC architecture as shown in Figure 3. The pipeline implementation is hidden from software except for branch delay slots and when CWP is modified by a WRCTL write.



#### **Direct CWP Manipulation**

Every WRCTL instruction that modifies the STATUS register (%ctl0) must be followed by a NOP instruction.

1

#### **Branch Delay Slots**

A branch delay slot is defined as the instruction immediately after a BR, BSR, CALL, or JMP instruction. A branch delay slot is executed after the branch instruction but before the branch-target instruction. Table 12 illustrates a branch delay-slot for a BR instruction.



After branch instruction (b) is taken, instruction (c) is executed before control is transferred to the branch target (e). The execution sequence of the above code fragment would be (a), (b), (c), and (e). Instruction (c) is instruction (b)'s branch delay slot. Instruction (d) is not executed. Most instructions can be used as a branch delay slot—the exceptions are:

BR BSR CALL IF1 IF0 IFRNZ IFRZ. IFS IMP LRET PFX PFXIO RET SKP1 SKP0 **SKPRNZ** SKPRZ SKPS TRET TRAP





This section provides a detailed description of the 32-bit Nios CPU instructions. The descriptions are arranged in alphabetical order according to instruction mnemonic. Each instruction page includes:

- Instruction mnemonic and description
- Description of operation
- Assembler syntax
- Syntax example
- Operation description
- Prefix actions
- Condition codes
- Delay slot behavior (when applicable)
- Instruction format
- Instruction fields

The  $\Delta$  symbol in the condition code flags table indicates flags are changed by the instruction.

Before the instruction set, the following tables are provided:

- Notation details table (Table 13)
- Instruction format (Table 14)
- **32-bit opcode table (Table 15)**
- GNU compiler/assembler pseudo instructions (Table 16)
- Nios operators understood by **nios-elf** [when available (Table 17)]
- Smallest Nios register file (Table 18)

<i>Tadie 13.</i> 1		1	1
Notation	Meaning	Notation	Meaning
$X \leftarrow Y$	X is written with Y	X >> n	The value X after being right-shifted n bit positions
Ø ← e	Expression e is evaluated, and the result is discarded	X << n	The value X after being left-shifted n bit positions
RA	One of the 32 visible registers, selected by the 5-bit a-field of the instruction word	bnX	The n <sup>th</sup> byte (8-bit field) within the full-width value X. ${}^{b0}X = X[70],$ ${}^{b1}X = X[158], {}^{b2}X = X[2316],$ and ${}^{b3}X = X[3124]$
RB	One of the 32 visible registers, selected by the 5-bit b-field of the instruction word	hnX	The n <sup>th</sup> half-word (16-bit field) within the full-width value X. $^{h0}X = X[150],$ $^{h1}X = X[3116]$
RP	One of the 4 pointer-enabled (P-type) registers, selected by the 2-bit p-field of the instruction word	X & Y	Bitwise logical AND
IMMn	An n-bit immediate value, embedded in the instruction word	X   Y	Bitwise logical OR
К	The 11-bit value held in the K register. (K can only be set by a PFX or PFXIO instruction)	X ⊕ Y	Bitwise logical exclusive OR
0xnn.mm	Hexadecimal notation (decimal points not significant, added for clarity)	~X	Bitwise logical NOT (one's complement)
X : Y	Bitwise-concatenation operator. For example: (0x12 : 0x34) = 0x1234	X	The absolute value of X (that is, -X if (X < 0), X otherwise).
{e1, e2}	Conditional expression. Evaluates to e2 if previous instruction was PFX or PFXIO, e1 otherwise	Mem32[X]	The aligned 32-bit word value stored in external memory, starting at byte address X
σ(X)	X after being sign-extended into a full register-sized signed integer	align32(X)	X & 0xFF.FF.FF.FC, which is the integer value X forced into full-word alignment via truncation
X[n]	The n <sup>th</sup> bit of X (n = 0 means LSB)	VECBASE	Byte address of Vector #0 in the interrupt vector table (VECBASE is configurable)
X[nm]	Consecutive bits n through m of X		·
С	The C (carry) flag in the STATUS register		
CTLk	One of the 2047 control registers selected by K		
PC	(Program Counter) Byte address of currently executing instruction.		

RR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
			op	56					В					Α			
Ri5	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
			op	6					IMM5					А			
Ri4	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	op6						0		IM	M4				Α			
RPi5	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
		op	54		F	D			В					А			
Ri6	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	ор5							IMN				Α					
Ri8	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
		ор3					IN	1M8						А			
i9	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
			op	06			IMM9									0	
i10	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
			op	6							IM	<b>/</b> 10					
i11	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
			op5							I	MM11						
Ri1u	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	ор6							op3u		IMM1u	0			Α			
Ri2u	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	op6							op3u IMM2u					A				

#### Table 14. Instruction Format (Part 1 of 2)

i8v	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
			op	6			op2v					IM	IMM8v					
i6v	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	op6					ор	2v	0	0			IM	/6v					
Rw	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
			op	6			op5w						А					
i4w	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
			op	6					op5w			0 IMM4w						
-																		
w	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	op6							op5w					0	0	0	0		

#### Table 14. Instruction Format (Part 2 of 2)

Oncodo	Mnomonio	Format	Summore
Ohcone	whemonic	Format	Summary
000000	ADD	RR	$RA \leftarrow RA + RB$
			Flags affected: N, V, C, Z
000001	ADDI	Ri5	$RA \leftarrow RA + (0 \times 00.00 : K : IMM5)$
			Flags affected: N, V, C, Z
000010	SUB	RR	$RA \leftarrow RA - RB$
			Flags affected: N, V, C, Z
000011	SUBI	Ri5	$RA \leftarrow RA - (0 \times 00.00 : K : IMM5)$
			Flags affected: N, V, C, Z
000100	CMP	RR	$\emptyset \leftarrow RA - RB$
			Flags affected: N, V, C, Z
000101	CMPI	Ri5	$\emptyset \leftarrow RA - (0 \times 00.00 : K : IMM5)$
			Flags affected: N, V, C, Z
000110	LSL	RR	$HA \leftarrow (HA << HB [40]),$
000444		D'-	
000111	LSLI	Ri5	$RA \leftarrow (RA << IMM5),$
001000			
001000	LSR	RR	$RA \leftarrow (RA >> RB [40]),$
001001		Dif	
001001	LSRI	RIS	$RA \leftarrow (R >> ININD),$
001010		DD	
001010	AGN	nn	$RA \leftarrow (RA >> RD [40]),$ Fill from left with RA[31]
001011		Dif	
001011	7011	1115	Fill from left with BA[31]
001100	MOV	BB	
001101	MOVI	Ri5	$RA \leftarrow (0 \times 00 \ 00 \cdot K \cdot IMM5)$
001110			$PA \leftarrow (0,00,00,11,1000)$
001110	AND	Bi5	$F_{A} \leftarrow F_{A} \otimes \{H_{B}, \{0, 00, 00, N, N$
001111		BB	$RA \leftarrow RA \otimes \mathbb{I}(IRB (0 \le 00, 00 : K : IMM5)))$
001111	ANDI	Bi5	Flags affected: N Z
010000	OB	BB	$BA \leftarrow BA \mid \{BB \mid (0 \times 00 \mid 00 : K : IMM5)\}$
010000	011	Ri5	Flags affected: N. Z
010001	XOR	BB.	$BA \leftarrow BA \oplus \{BB, (0 \times 00.00 : K : IMM5)\}$
		Ri5	Flags affected: N, Z
010010	BGEN	Ri5	$RA \leftarrow 2^{IMM5}$
010011	EXT8D	BB	$BA \leftarrow (0 \times 00.00.00)^{bn}BA$ where $n = BB[1,0]$
010100	SKP0	Bi5	Skip next instruction if: (BA [IMM5] == 0)
010101	SKP1	Bi5	Skip next instruction if: (BA [IMM5] 1)
010110			$PA \leftarrow Mom 22 \left[ a \log 22 (PR + (\sigma(K) \times 4)) \right]$
010110		nn	$\square A \leftarrow \text{ivenior [alignor(ND + (0(N) \times 4))]}$

Table 15. 32-bit Opcode Table (Part 2 of 3)											
Opcode	Mnemonic	Format	Summary								
010111	ST	RR	Mem32 [align32(RB + ( $\sigma(K) \times 4$ ))] $\leftarrow$ RA								
011000	STS8S	i10	<sup>bn</sup> Mem32 [align32(%sp + IMM10)] $\leftarrow$ <sup>bn</sup> %r0 where $n = IMM10[10]$								
011001	STS16S	i9	<sup>hn</sup> Mem32 [align32(%sp + IMM9 × 2)] $\leftarrow$ <sup>hn</sup> %r0 where <i>n</i> = IMM9[0]								
011010	EXT16D	RR	$RA \leftarrow (0 \times 00.00 : {}^{hn}RA)$ where $n = RB[1]$								
011011	MOVHI	Ri5	<sup>h1</sup> RA $\leftarrow$ (K : IMM5), <sup>h0</sup> RA unaffected								
011100	USR0	RR	$RA \leftarrow RA$ <user-defined operation=""> RB</user-defined>								
011101000	EXT8S	Ri1u	$RA \leftarrow (0 \times 00.00.00 : {}^{bn}RA)$ where $n = IMM2u$								
011101001	EXT16S	Ri1u	$RA \leftarrow (0 \times 00.00 : {}^{hn}RA)$ where $n = IMM1u$								
011101010			Unused								
011101011			Unused								
011101100	ST8S	Ri1u	<sup>bn</sup> Mem32 [align32(RA + ( $\sigma(K) \times 4$ ))] $\leftarrow$ <sup>bn</sup> %r0 where <i>n</i> = IMM2u								
011101101	ST16S	Ri1u	<sup>hn</sup> Mem32 [align32(RA + ( $\sigma(K) \times 4$ ))] $\leftarrow$ <sup>hn</sup> %r0 where <i>n</i> = IMM1u								
01111000	SAVE	i8v	$CWP \leftarrow CWP - 1; \ \%sp \leftarrow \%fp - (IMM8v \times 4)$ If (old-CWP == LO_LIMIT) then TRAP #1								
0111100100	TRAP	i6v	$\begin{split} \text{ISTATUS} &\leftarrow \text{STATUS}; \text{ IE} \leftarrow 0; \text{ CWP} \leftarrow \text{CWP} - 1; \\ \text{IPRI} \leftarrow \text{IMM6v}; \ \%\text{r15} \leftarrow ((\text{PC} + 2) >> 1); \\ \text{PC} \leftarrow \text{Mem32} \left[\text{VECBASE} + (\text{IMM6v} \times 4)\right] \times 2 \end{split}$								
01111100000	NOT	Rw	$RA \leftarrow \sim RA$								
01111100001	NEG	Rw	$RA \leftarrow 0 - RA$								
01111100010	ABS	Rw	$RA \leftarrow  RA $								
01111100011	SEXT8	Rw	$RA \leftarrow \sigma(^{b0}RA)$								
01111100100	SEXT16	Rw	$RA \leftarrow \sigma(^{h0}RA)$								
01111100101	RLC	Rw	$C \leftarrow msb$ (RA); RA $\leftarrow$ (RA << 1) : C Flag affected: C								
01111100110	RRC	Rw	$C \leftarrow RA[0]; RA \leftarrow C : (RA >> 1)$ Flag affected: C								
01111100111			Unused								
01111101000	SWAP	Rw	$RA \leftarrow {}^{h0}RA : {}^{h1}RA$								
01111101001	USR1	Rw	$RA \leftarrow RA$ <user-defined operation=""> R0</user-defined>								
01111101010	USR2	Rw	$RA \leftarrow RA$ <user-defined operation=""> R0</user-defined>								
01111101011	USR3	Rw	$RA \leftarrow RA$ <user-defined operation=""> R0</user-defined>								
01111101100	USR4	Rw	$RA \leftarrow RA$ <user-defined operation=""> R0</user-defined>								
01111101101	RESTORE	w	$CWP \leftarrow CWP + 1$ ; if (old-CWP == HI_LIMIT) then TRAP #2								
01111101110	TRET	Rw	$PC \leftarrow (RA \times 2); STATUS \leftarrow ISTATUS$								

Table 15. 32-b	nit Opcode Tabl	le (Part 3 d	of 3)
Opcode	Mnemonic	Format	Summary
01111101111			Unused
01111110000	ST8D	Rw	<sup>bn</sup> Mem32 [align32(RA +( $\sigma(K) \times 4$ ))] $\leftarrow {}^{bn}$ %r0 where $n = RA[10]$
01111110001	ST16D	Rw	<sup>hn</sup> Mem32 [align32(RA + ( $\sigma(K) \times 4$ ))] $\leftarrow$ <sup>hn</sup> %r0 where $n = RA[1]$
01111110010	FILL8	Rw	%r0 ← ( $^{b0}$ RA : $^{b0}$ RA : $^{b0}$ RA : $^{b0}$ RA )
01111110011	FILL16	Rw	$\%$ r0 $\leftarrow$ ( <sup>h0</sup> RA : <sup>h0</sup> RA)
01111110100	MSTEP	Rw	if (%r0[31] == 1) then %r0 ← (%r0 << 1) + RA else %r0 ← (%r0 << 1)
01111110101	MUL	Rw	$\%r0 \leftarrow (\%r0 \& 0x0000.ffff) \times (RA \& 0x0000.ffff)$
01111110110	SKPRZ	Rw	Skip next instruction if: (RA == 0)
01111110111	SKPS	i4w	Skip next instruction if condition encoded by IMM4w is true
01111111000	WRCTL	Rw	$CTLk \leftarrow RA$
01111111001	RDCTL	Rw	$RA \leftarrow CTLk$
01111111010	SKPRNZ	Rw	Skip next instruction if: (RA != 0)
01111111011			Unused
01111111100			Unused
01111111101			Unused
01111111110	JMP	Rw	$PC \leftarrow (RA \times 2)$
01111111111	CALL	Rw	$R15 \leftarrow ((PC + 4) >> 1);  PC \leftarrow (RA \times 2)$
100000	BR	i11	$PC \leftarrow PC + ((\sigma(IMM11) + 1) \times 2)$
100001			Unused
100010	BSR	i11	$\begin{array}{l} PC \leftarrow PC + ((\sigma(IMM11) + 1) \times 2); \\ \$ r15 \leftarrow ((PC + 4) >> 1) \end{array}$
10010	PFXIO	i11	$K \leftarrow IMM11$ (K set to zero after next instruction and forces subsequent memory load instruction to bypass the data cache)
10011	PFX	i11	K ← IMM11 (K set to zero after next instruction)
1010	STP	RPi5	$Mem32[align32(RP + (\sigma(K : IMM5) \times 4))] \leftarrow RA$
1011	LDP	RPi5	$RA \leftarrow Mem32 \ [align32(RP + (\sigma(K:IMM5) \times 4))]$
110	STS	Ri8	Mem32[align32(%sp + (IMM8 × 4) )] ← RA
111	LDS	Ri8	$RA \leftarrow Mem32  [align32(\%sp + (IMM8 \times 4))]$

The following pseudo-instructions are generated by **nios-elf-gcc** (GNU compiler) and understood by **nios-elf-as** (GNU assembler).

Table 16. GNU Compiler/Assembler Pseudo-Instructions										
Pseudo-Instruction	Equivalent Instruction	Notes								
LRET	JMP %07	LRET has no operands								
RET	JMP %i7	RET has no operands								
NOP	MOV %g0,%g0	NOP has no operands								
IF0 %rA,IMM5	SKP1 %rA,IMM5									
IF1 %rA,IMM5	SKP0 %rA,IMM5									
IFRZ%rA	SKPRNZ %rA									
IFRNZ %rA	SKPRZ %rA									
IFS cc_c	SKPS cc_nc									
IFS cc_nc	SKPS cc_c									
IFS cc_z	SKPS cc_nz									
IFS cc_nz	SKPS cc_z									
IFS cc_mi	SKPS cc_pl									
IFS cc_pl	SKPS cc_mi									
IFS ccge	SKPS cc_lt									
IFS cc_lt	SKPS cc_ge									
IFS cc_le	SKPS cc_gt									
IFS cc_gt	SKPS cc_le									
IFS cc_v	SKPS cc_nv									
IFS cc_nv	SKPS cc_v									
IFS cc_ls	SKPS cc_hi									
IFS cc_hi	SKPS cc_ls									

The following operators are understood by **nios-elf-as**. These operators may be used with constants and symbolic addresses, and can be correctly resolved either by the assembler or the linker.

Table 17. Nios	Table 17. Nios Operators												
Operator	Description	Operation											
%lo( <i>x</i> )	Extract low 5 bits of x	<i>x</i> & 0×0000001f											
%hi( <i>x</i> )	Extract bits 155 of x	( <i>x</i> >> 5) & 0×000007ff											
%xlo( <i>x</i> )	Extract bits 2016 of x	( <i>x</i> >> 16) & 0×0000001f											
%xhi( <i>x</i> )	Extract bits 3121 of x	( <i>x</i> >> 21) & 0×000007ff											
<i>x</i> @h	Half-word address of x	<i>x</i> >> 1											

(Internal Register File)	CWP=6 (HI_LIMIT)						
Reg[120127]	%i0%i7	7					
Reg[112119]	%L0%L7	CWP=5		◀	<ul> <li>Restore</li> </ul>		
Reg[104111]	%00%07	7 %i0%i7					
Reg[96103]		%L0%L7	CWP=4			Save ——	
Reg[8895]		%00%07	%i0%i7				
Reg[8087]			%L0%L7	CWP=3			
Reg[7279]			%00%07	%i0%i7			
Reg[6471]				%L0%L7	CWP=2		
Reg[5663]				%00%07	%i0%i7		
Reg[4855]					%L0%L7	(LOW_LIMIT)	
Reg[4047]					%00%07	%i0%i7	CWP=0
Reg[3239]						%L0%L7	or TRAP)
Reg[2431]						%00%07	%i0%i7
Reg[1623]							%L0%L7
Reg[815]							%00%07
Reg[07]	%g0%g	7 %g0%g7	%g0%g7	%g0%g7	%g0%g7	%g0%g7	%g0%g7

#### Table 18. Smallest Nios Register File

This shows the smallest Nios register file, which is 128 registers. Larger files have more register windows.

Register Groups for CWP=0								
%r24%r31	aka %i0%i7							
%r16%r23	aka %L0%L7							
%r8%r15	aka %o0%o7							
%r0%r7	aka %g0%g7							

# ABS

#### Absolute Value

Operat	ion:			$RA \leftarrow  RA $											
Assem	bler Sy	ntax:		ABS %rA											
Examp	le:			ABS %r6											
Descri	ption:			Calculate the absolute value of RA; store the result in RA.											
Condit	ion Co	des:		Flags: Unaffected											
				N V Z C											
Instruc	tion Fo	ormat:		Rw											
Instruc	tion Fi	elds:		A = F	Regis	ter ir	ndex o	of opera	and RA	L .					
15	14	13	12	11	10	ę	)	8	7	6	5	4	3		

Α

1 0

## ADD

Opera	tion:			RA	$\leftarrow$ RA -	⊦ RB										
Assen	nbler S	yntax:		ADD	ADD %rA,%rB											
Exam	ple:			ADD	ADD %L3,%g0 ; ADD %g0 to %L3											
Descr	iption:			Add	Adds the contents of register A to register B and stores the result in register A.											
Condi	tion Co	odes:		Flag	Flags:											
				Ν	V	Ζ	С									
				$\Delta$	$\Delta$ $\Delta$ $\Delta$ $\Delta$											
				N: F	N: Result bit 31											
				V: S	V: Signed-arithmetic overflow											
				Z: S	set if re	sult i	s zei	ro; cle	ared ot	herwis	e					
				C: C	Carry-o	ut of	addi	tion								
Instru	ction F	ormat		RR												
Instru	ction F	ields:		A =	Regist	er in	dex d	of RA	operan	d						
				B =	Regist	er in	dex d	of RB	operan	d						
15	14	13	12	11	10	9		8	7	6	5	4	3	2	1	0
0	0	0	0	0	0				В					Α		

# ADDI

#### Add Immediate

Opera	tion:			RA	$\leftarrow RA$ -	+ (0x00	.00 : K : I	MM5	5)							
Asser	nbler S	Syntax:		ADI	DI %rA	,IMM5	5									
Exam	ple:			Not	prece	ded by	/ PFX:									
				ADI Pre	DI %L5 <b>ceded</b>	, 6 by PF	<b>X</b> :	;	add	6 to	%L	5				
				PFX ADI	K %hi( )I %g3	1000) ,%lo(	(1000)	;	ADD	1000	to	%g3				
Descr	iption:			Not	prece	ded by	/ PFX:									
				Ado rang	ls 5-bit ge [03	immed 1].	iate value	e to re	egiste	er A, ste	ores	result	in regis	ster A. I	MM5 is	s in the
				Pre	ceded	by PF	X:									
				The	immeo	liate o	perand is	exte	nded	from §	5 to	16 bits	s by cor	ncatena	ting th	е
				con	tents of	the K-	register (	11 bi	ts) wi	th IMN	15 (5	5 bits).	The 16	-bit imn	nediate	e value
				(K :	IMM5)	is zero	o-extende	ed to	32 bit	ts and	adc	led to i	register	·A.		
Cond	tion Co	odes:		Flag	gs:											
				N Δ	V $\Delta$	<b>Ζ</b> 2										
				N: F	Result b	oit 31										
				V: 5	Signed-	arithm	etic overf	low								
				Z: S	Set if re	sult is a	zero; clea	ared	other	wise						
				C: 0	Carry-o	ut of a	dition									
Instru	ction F	ormat		Ri5												
Instru	ction F	ields:		A =	Regist	er inde	x of RA	opera	ind							
				IMN	/15 = 5-l	oit imm	ediate va	alue								
15	14	13	12	11	10	9	8	7	6	5	5	4	3	2	1	0
0	0	0	0	0	1		I	MM5						Α		

### **AND** Bitwise Logical AND

Opera	ation:			Not RA Pre RA	t prece $\leftarrow RA d$ ceded $\leftarrow RA d$	ded by & RB by PF) & (0x00	<b>PFX:</b> (: ).00 : K	: IMM5	)						
Asser	nbler S	yntax:		Not ANI Pre PF2 ANI	t prece ) %rA, ceded K %hi( ) %rA,	ded by %rB by PF) const %lo(c	PFX: (: ) onst)								
Exam	ple:			Not ANI Pre PF2 ANI	t <b>prece</b> 9 %g0, <b>ceded</b> K %hi( 9 %g0,	<b>ded by</b> %g1 <b>by PF)</b> 16383 %lo(1	<b>PFX:</b> (: ) 6383)		; %g;	0 gets D %g0	%gl with	& %g0 16383	i		
Descr	Description: Condition Codes:				t <b>prece</b> ically-A result in <b>ceded</b> RB op content ended to	ded by ND the n RA. by PF) erand is ts of the o 32 bit	PFX: individ (: s repla e K-reg s) is bit	dual bits ced by a ister (1 twise-Al	s in RA an imm 1 bits) NDed v	with th nediate with IMI with RA	e corre consta M5 (5 b , and th	spondi nt form its). Th ne resu	ng bits ed by c iis 16-bi It is writ	in RB; oncate it value ten bao	store nating (zero- ck into
Cond	ition Co	odes:		Flag N A N: I	gs: V – Result b	Z C ∆ –	;								
				Z: 8	Set if rea	sult is z	ero, cl	eared o	therwis	se					
Instru	ction F	ormat:		RR	, Ri5										
Instru	iction F	ields:		A = B = IMN	Registe Registe //5 = 5-b	er inde: er inde: pit imm	x of RA x of RE ediate	operar operar value	nd nd						
Not p	recedeo	l by PF	•X (RR)	)											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0			В					А		
Prece	ded by	PFX (F	łi5)												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

IMM5

**Altera Corporation** 

0

1

1

1

0

0

Α

# ANDN

#### **Bitwise Logical AND NOT**

Opera	tion:			Not	t prece	ded by	y PFX:									
				RA	$\leftarrow$ RA $\stackrel{\circ}{\rightarrow}$	ጲ ~RB <b>by D</b> E	v.									
				RA	← RA	<b>Бугг</b> &. ~(0х	<b>∧.</b> :00:00 · k	· IMN	15)							
Assen	nbler S	vntax		Not	prece	ded b	v PFX:		,							
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,		ymaxi		ANI	DN %rA	,%rB	<b>,</b>									
				Pre	ceded	by PF	X:									
				PFZ	K %hi(	const	=)									
_				ANI	DN %rA	,%lo	(const	)								
Examp	ple:			Not	t prece	ded b	y PFX:			_						
				ANI Pre	ON %g0 ceded	,%gl bv PF	×٠		; %	g0 get	s %g0	6~%	gl			
				PFX	<pre>Kulture K %hi()</pre>	16384	<b>A.</b> 1)									
				ANI	ANDN %g0,%lo(16384) ; clear bit 14 of %g0											
Descri	iption:			Not	lot preceded by PFX:											
				Log	gically-AND the individual bits in RA with the corresponding bits in the one's-											
				con	npleme	nt of F	B; store	the re	sult in F	RA.						
Preceded by PFX: The BB operand is replaced by an immedi																
				The	RB op	erand	is replac	ced by	an imm	nediate	consta	nt form	ed by c	oncate	nating	
				the	conten	IS OF TH	e K-regi	ster (1	I DITS) V		/15 (5 DI		S 16-DIT	Value I	s zero-	
				is w	ritten h	o 32 D ack in	to BA	DIIWISE	-inverte	ea ana	bitwise-	ANDed		A. The	result	
Condi	tion Co	-doc		Fla	ne.		10 TIA.									
Contai		Jue3.		N	ys. V	7 (	~									
						~										
				Δ	_	Δ ·	-									
				N: I	Result b	oit 31										
				Z: 8	Set if re	sult is	zero, cle	eared c	otherwis	se						
Instru	ction F	ormat	:	RR	, Ri5											
Instru	ction F	ields:		A =	Regist	er inde	ex of ope	erand F	٦A							
				B =	Regist	er inde	ex of ope	erand F	RB							
				IIVIN	/15 = 5-1	oit imn	iediate v	alue								
Not pr	ecede	d by P	FX (RR	)		_		_		_		_				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	1	1	1			В					A			

Prece	ded by	PFX (	Ri5)												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	IMM5							Α		

### **ASR** Arithmetic Shift Right

32-Bit Instruction Set

Opera Assen Exam	ition: nbler S ple:	Syntax:		RA ASR ASR	← (RA 8rA, 8L3,	>> <b>RB</b> [4 %rB %g0	0]), fi ;	ll from l shift	eft with %L3	h RA[31 right	l] by %	g0 bi	ts		
Descr	iption:			Arith Bits one	nmetica 315 d depen	ally shift of RB are ding on	right th e ignor the ori	e value ed. If th ginal sig	in RA e value gn of F	by the e in RB RA.	value o [40] is	1 RB; s 31, R/	tore th A is ze	ne resuli ero or ne	t in RA. egative
						y bit 31	9 28			<u></u>			2	1 0	]
Condi	tion Co	odes:		Flag N –	gs: Una V –	Iffected Z C 	]								
Instru	ction F	ormat	:	RR											
Instru	ction F	ields:		A = B =	Regist Regist	er index er index	of RA of RB	operan operan	d d						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0			В					Α		

57

# ASRI

#### Arithmetic Shift Right Immediate

Operation:	$RA \leftarrow (RA \gg IMM5)$ , fill from left with $RA[31]$
Assembler Syntax:	ASRI %rA,IMM5
Example:	ASRI %i5,6 ; shift %i5 right 6 bits
Description:	Arithmetically shift right the contents of RA by IMM5 bits. If IMM5 is 31, RA is zero
	or negative one depending on the original sign of RA.



Condi	tion Co	odes:		Flag N	gs: Una V –	affected Z (	d C -								
Instru Instru	Ri5 A = IMN	Regist 15 = 5-	er inde bit imn	ex of RA	A operar value	nd									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1			IMM5					А		

### BGEN Bit Generate

Operation:	$RA \leftarrow 2^{IMM5}$											
Assembler Syntax:	BGEN %rA,IMM5											
Example:	BGEN %g7,6 ; set %g7 to 64											
Description:	Sets RA to an integer power-of-two with the exponent given by IMM5. This is equivalent to setting a single bit in RA, and clearing the rest.											
Condition Codes:	Flags: Unaffected											
	N V Z C											
Instruction Format:	Ri5											
Instruction Fields:	A = Register index of RA operand											
	IMM5 = 5-bit immediate value											
15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0											
0 1 0 0	1 0 IMM5 A											

# BR

#### Branch

Opera	tion:			PC é	- PC +	((σ(IM	M11) +	1) << 1	)						
Assen	nbler S	yntax:		BR	addr										
Examp	ole:			BR	MainL	oop									
				NOP	•	; (de	lay s	lot)							
Descri	iption:			The (inst is tra	offset truction ansferr	given b s) relat ed to ir	ive to t istructi	11 is int he instru on at thi	erprete uction i is offse	ed as a mmedia et.	signed Itely fol	numbe llowing l	er of ha BR. Pro	lf-words ogram o	s ontrol
Condi	tion Co	odes:		Flag	js: Una	ffected									
				Ν	V	Z C	;								
				-	-										
Delay	Slot B	ehavio	r:	The	instruc	tion im	media	tely follo	wing E	BR (BR'	s delay	/ slot) is	execu	ted afte	er BR,
				but l	before	the des	tinatio	n instruc	tion. T	here are	e restri	ctions o	n whic	h instru	ctions
				may	be us	ed as a	delay	slot (se	e "Bra	nch Dela	ay Slot	s" on pa	age 42	).	
Instru	ction F	ormat		i11											
Instru	ction F	ields:		IMN	111 = 1	1-bit im	nmedia	te value	•						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0 IMM11										

### **BSR** Branch To Subroutine

Opera	tion:			%07	′ ← ((I	PC + 4) >	>>1)								
				PC	- PC	+ ((σ(IM	IM11) -	+ 1) << 1)	)						
Assem	bler S	yntax:		BSR	add	r									
Exam	ole:			BSR	Sen	dChara	lcter								
				NOP	•	; (de	alay s	slot)							
Descri	ption:			The (inst is tra BSF instr right with	offse tructio ansfer ansfer anstr structior t-shift out m	t given t ns) relat red to ir uction p n. The re ed value	by IMN tive to the struction of the store tion.	111 is inte the instru ion at this ur, which address is d in %07	erpret ction offse is the s shift is a c	ed as a s immedia et. The re address ed right o destinatic	signed tely fol turn-a of the one bit on suit	numbe llowing l ddress e secon t and st able for	er of ha BR. Pro is the a d subso ored in direct	If-words ogram c address equent %o7. 1 use by	ontrol of the The JMP
Condi	tion Co	odes:		Flac	ıs: Un	affected	4								
				N _	V -	Z (	> -								
Delay	Slot B	ehavio	r:	The BSF instr	instru R, but ructior	iction im before t ns may	nmedia he des be use	itely follo stination i ed as a de	wing I instruc elay sl	BSR (BS ction. Th lot (see "	R's de ere are Branc	elay slot e restric h Delay	t) is exe ctions c v Slots"	ecuted a on which on pag	after n le 42).
Instru	ction F	ormat		i11											
Instru	ction F	ields:		IMN	111 =	11-bit in	nmedia	ate value							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1						IMM11					

# CALL

#### **Call Subroutine**

Opera	tion:			%07	$V \leftarrow ((P \land P)) \rightarrow V$	C+4)>	>1)								
				PC •	⊢ (RA	<< 1)									
Assen	nbler S	yntax:		CAL	L %rA										
Examp	ple:			CAL	L %g0										
				NOF	)	; (de	lay s	lot)							
Descri	iption:			The add add bit a suita	value ress of ress of and sto able fo	of RA is the cal the sec red in % r direct	s shifte led sub cond su %o7. Th use by	d left by routine bseque ne right JMP w	y one a right-sl ent instr -shifted rithout r	nd tran hifted b uction. value nodifica	sferred y one b Return stored i ation.	into P( it. The r addres in %07	C. RA c return-a ss is shi is a de	contains address ifted rig stinatio	s the is the ht one n
Condi	tion Co	odes:		Flag	gs: Una	ffected									
				N _	V -	Z C	;								
Delay	Slot B	ehavio	r:	The	instruc	ction im	mediat	ely follo	owing C	ALL (C	ALL's d	delay sl	ot) is e	xecuted	d after
				CAL inst	L, but	before s may t	the des	stination I as a d	n instru Ielay slo	ction. T ot (see	here a "Branc	re restri h Delay	ictions <sup>,</sup> Slots"	on whic on pag	ch je 42).
Instru	ction F	ormat		Rw											
Instru	ction F	ields:		A =	Regist	er inde	x of ope	erand F	RA						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	1	1			А		

### CMP Compare

Operation:	$\varnothing \leftarrow RA - RB$										
Assembler Syntax:	CMP %rA,%rB										
Example:	CMP %g0,%g1 ; set flags by %g0 - %g1										
Description:	Subtract the contents of RB from RA, and discard the result. Set the condition codes according to the subtraction. Neither RA nor RB are altered.										
Condition Codes:	Flags:										
	$ \begin{array}{c cc} N & V & Z & C \\ \hline \Delta & \Delta & \Delta & \Delta \end{array} $										
	N: Result bit 31 V: Signed-arithmetic overflow Z: Set if result is zero; cleared otherwise C: Set if there was a borrow from the subtraction; cleared otherwise										
Instruction Format:	RR										
Instruction Fields: A = Register index of RA operand B = Register index of RB operand											
15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0										
0 0 0 1	0 0 B A										

# CMPI

#### **Compare Immediate**

Opera	tion:			$\varnothing \leftarrow \text{RA} - (0x00.00 : \text{K} : \text{IMM5})$											
Asser	nbler S	yntax:		CMF	8 & I	rA,IN	1M5								
Exam	ple:			Not	Not preceded by PFX:										
				CMF Pre	PI %i3 <b>ceded</b>	, 24 <b>by PF</b>	X:	;	compa	are %i	.3 to	24			
				CMF	91 %i4	,%100	(1000)	;	compa	are %i	.4 to	1000			
Descr	iption:			Not	prece	ded by	/ PFX:								
	•			Subtract a 5-bit immediate value given by IMM5 from RA, and discard the result.											
				Set the condition codes according to the subtraction. RA is not altered.											
				Preceded by PFX:											
				The	Immed	diate o	perand i	s exte	nded fro	om 5 to	16 bits	s by cor	ncatena	ting the	Э
				con	tents of	the K-	register	(11 bit	s) with	IMM5 (	5 bits).	The 16	-bit imn	nediate	value
				(K :	(K : IMM5) is zero-extended to 32 bits and subtracted from RA. Condition codes										
				are set and the result is discarded. RA is not altered.											
Condi	tion Co	odes:		Flags:											
				Ν	N V Z C										
				N: Result bit 31											
				V: Signed-arithmetic overflow											
				Z: Set if result is zero; cleared otherwise											
				C: 5	C: Set if there was a borrow from the subtraction; cleared otherwise										
Instru	ction F	ormat		Ri5											
Instru	ction F	ields:		A = Register index of RA operand											
				IMM5 = 5-bit immediate value											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1			IMM5					A		

### EXT16D Half-Word Extract (Dynamic)

Opera	<b>Operation:</b> $RA \leftarrow (0x00.00: {}^{hn}RA) \text{ where } n = RB[1]$																
Assen	nhler S	vntax		FYT	א ת16י	rz &ri	R			1							
Exam	ple:	ymux.		LD EXT	LD %i3,[%i4] ; get 32 bits from [%i4 & 0xFF.FF.FF. EXT16D %i3,%i4 ; extract short int at %i4												
Descr	iption:			Extr by b mor	Extracts one of the two half-words in RA. The half-word to-be-extracted is chosen by bit 1 of RB. The selected half-word is written into bits 150 of RA, and the more-significant bits 3116 are set to zero.												
						31				16 15	5			0			
		RA half word 1 half										half wor	word 0				
					RA after	31		0		 16 15	5	half wor	R	B[10]			
Condi	tion Co	odes:		Flag	Flags: Unaffected												
				N _	V -	Z C 											
Instru	ction F	ormat		RR													
Instru	ction F	ields:		A =	Regist	er index	c of oper	and R	A								
				B =	B = Register index of operand RB												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	0	1	0			В					А				

**Altera Corporation** 

# EXT16S

Half-Word Extract (Static)

Operation:

Assembler Syntax: Example: Description: EXT16S %rA, IMM1 EXT16S %L3,1 ; %L3 gets upper short int of itself Extracts one of the two half-words in RA. The half-word to-be-extracted is chosen by the one-bit immediate value IMM1. The selected half-word is written into bits



 $RA \leftarrow (0x00.00 : {^{hn}RA})$  where n = IMM1

### **EXT8D** Byte-Extract (Dynamic)



# EXT8S

Byte-Extract (Static)

#### Operation:

Assembler Syntax: Example: Description:  $RA \leftarrow (0x00.00.00 : {}^{bn}RA)$  where n = IMM2EXT8S %rA, IMM2 EXT8S %g6,3 ; %g6 gets the 3rd byte of itself

Extracts one of the four bytes in RA. The byte to-be-extracted is chosen by the immediate value IMM2 (byte 3 being the most-significant byte of RA). The selected byte is written into bits 7..0 of RA, and the more-significant bits 31..8 are set to zero.



### FILL16 Half-Word Fill

Opera	tion:			$R0 \leftarrow ({}^{h0}RA : {}^{h0}RA)$													
Assem	nbler S	yntax:		FIL	L16 %	r0,%r	A										
Example: FILL16 %r						r0,%i	3	; %r(	) gets	s 2 c	opies of %i3[015]						
								; fi	peran	nd must be %r0							
Descri	ption:			The least significant half-word of RA is copied into both half-w								-word p	ositior	S			
				in %	r0. %r	0 is the	only a	llowed	destina	tion op	perand	for FILL	instruc	tions.			
						21				16	15	<u>^</u>					
					RA	51		half word	1	10	15	hali	f word 0		0		
				b	efore			nali woru	I			lidii					
													_				
				B0							r	Ţ					
		after half word 0								half word 0							
						31				16	15				0		
Condi	tion Co	odes:		Flag	ıs: Una	ffected											
				N V Z C													
Instru	ction F	ormat		Bw													
Instruction Fields:			A =	Regist	er inde:	k of ope	erand F	RA									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	1	1	1	1	0	0	1	1			А				

## FILL8

**Byte-Fill** 

Operation:	$R0 \leftarrow ({}^{b0}RA : {}^{b0}RA : {}^{b0}RA : {}^{b0}RA)$											
Assembler Syntax:	FILL8 %r0,%rA											
Example:	FILL8 %r0,%o3 ; %r0 gets 4 copies of %o3[07]											
	; first operand must be %r0											

Description:

The least-significant byte of RA is copied into all four byte-positions in %r0. %r0 is the only allowed destination operand for FILL instructions.



## IFO

#### Execute if Register Bit is O

(Equivalent to SKP1 Instruction)

Opera	tion:			if (RA[IMM5] == 1) then begin if (Mem16[PC + 2] is PFX or PFXIO) then PC $\leftarrow$ PC + 6 else PC $\leftarrow$ PC + 4												
				end												
Asser	nbler S	yntax:		IF0	IFO %rA,IMM5											
Exam	ple:			IF0	) %o3,	21	1;									
Descr	iption:			Skip or F are	Skip next instruction if the single bit RA[IMM5] is 1. If the next instruction is PFX or PFXIO, then both PFX or PFXIO and the instruction following PFX or PFXIO are skipped together.											
Condi	tion Co	odes:		Flaç N –	Flags:         Unaffected           N         V         Z         C           -         -         -         -         -											
Instru	ction F	ormat		Ri5	Ri5											
Instru	ction F	ields:		A =	A = Begister index of operand BA											
					IMM5 = 5-bit immediate value											
					-											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	1	0	1			IMM5					А			

# IF1

#### Execute if Register Bit is 1

(Equivalent to SKP0 Instruction)

Opera	tion:			if $(RA[IMM5] == 0)$												
•				then begin												
				if (Mem16[PC + 2] is PFX or PFXIO)												
				then $PC \leftarrow PC + 6$ else $PC \leftarrow PC + 4$												
				end												
Asser	nbler S	Syntax:		IFI	IF1 %rA,IMM5											
Exam	ple:			IF1	IF1 %i3, 7											
				ADI	DI %g0	,1	; i	increm	ent i	f bit	7 is	set				
Descr	iption:			Skip next instruction if the single bit RA[IMM5] is 0. If the next instruction is PFX												
	•			or F	or PFXIO, then both PFX or PFXIO and the instruction following PFX or PFXIO											
				are	skippe	d toaet	her.						0			
Condi	tion C	odee		Flags: Unaffected												
Conta		Jues.		I IA												
				N	v	2 (	<i>.</i>									
				-	-		-									
Instru	ction F	ormat	:	Ri5												
Instru	ction F	ields:		A = Register index of operand RA												
				IMM5 = 5-bit immediate value												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	1	0	0			IMM5					Α			
## IFRNZ

### Execute if Register is not Zero

(Equivalent to SKPRZ Instruction)

Operat	tion:			if (R	$\mathbf{A} == 0$	)												
-				then	begin													
					if (Me	m16[PC	2 + 2] is	PFX or	PFXIO	)								
					then P	$C \leftarrow PC$	C + 6											
					else PO	$\mathbb{C} \leftarrow PC$	+ 4											
				end														
Assem	nbler S	yntax:		IFF	NZ %r	A												
Examp	ole:			IFF	NZ %o	3												
				BSF	Send	It	; on	ly bra	anch i	lf %o3	is n	ot ze	ro					
				NOF	NOP ; (delay slot) executed in either case Skin pextinstruction if <b>BA</b> is equal to zero. If the pextinstruction is <b>PEX</b> or <b>PEXIO</b>													
Descri	iption:			Skip	NOP       ; (delay slot) executed in either case         Skip next instruction if RA is equal to zero. If the next instruction is PFX or PFXIO													
				ther	n both F	PFX or	PFXIO	and the	e instruc	ction fo	llowing	PFX or	PFXIC	) are sł	kipped			
				toge	ether.													
Condit	tion Co	odes:		Flag	gs: Una	ffected												
				Ν	V	z c	;											
				-	-													
					<u> </u>													
Instruc	ction F	ormat	:	Rw														
Instruc	ction F	ields:		A =	Regist	er inde	x of op	erand F	RA									
					-													
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	1	1	1	1	1	1	0	1	1	0			Α					

## IFRZ

### Execute if Register is Zero

(Equivalent to SKPRNZ Instruction)

Opera	tion:			if (R	A != 0)										
•				then	begin										
					if (Me	m16[PC	2+2] is	PFX or	PFXIO	)					
					then P	$C \leftarrow PC$	2 + 6								
					else PO	$C \leftarrow PC$	+4								
				end											
Assen	nbler S	yntax:		IFR	Z%rA										
Exam	ple:			IFR	Z %g3										
				BSR	. Send	It	; on	ly ca	11 if	%g3 i	s zer	0			
				NOF	)		; (d	elay s	slot)	exect	ited i	n eit	her c	ase	
Descr	iption:			Skip	o next ir	nstructi	on if RA	is not :	zero. If	the nex	t instru	ction is	PFX or	PFXIC	), then
				both	n PFX d	or PFXI	O and	the inst	ruction	followi	ng PFX	or PF	XIO are	skippe	ed
				toge	ether.										
Condi	tion Co	odes:		Flag	gs: Una	ffected									
				Ν	V	z c	;								
				-	-										
Inchur	ation F			<b>D</b>											
instru		ormat		RW											
Instru	ction F	ields:		A =	Regist	er inde	x of ope	erand F	RA						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	1	0			А		

## IFS

#### **Conditionally Execute Next Instruction**

Opera	tion:			if (c	ondition	IMM	14 is fals	se)							
				then	begin										
					if (Me	m16[	PC + 2]	is PFX 01	PFXIO	)					
					then P	$C \leftarrow C$	PC + 6								
				and	else Po	$C \leftarrow I$	PC + 4								
Accor	nhlar S	wotay		TRC	, aa T	<b>M M M M</b>									
ASSEI		ymax.		168	, cc_1	MM4									
Exam	pie:			LFS	cc_n	.e									
				BSF	sena	Ξt	; 0	niy ca dolou	II II	Z IIG	ag se it od	t in cit	hora		
Deeer				NOP	, 		; (	ueray	SIUC)	exect			ner c	ase	16
Descr	iption:			Exe	cute ne			n if spec				, SKIP IT		on is tais	Se. If
				follo					-XIU, In	en bou			U and t	ne instri	uction
0					wing r			J ale Ski	ppeu io	yemen	•				
Condi	uon C	Jues:		Set	ings:										
					CC	_nc	0x0	(not C)							
F	The	ese con	dition		С	c_c	0x1	(C)							
-C.3	coc	les hav	e		сс	_nz	0x2	(not Z)							
	diff	erent			С	c_z	0x3	(Z)							
	nur for	neric va IFS and	aiues d		C	c_pl	0x4	(not N)							
	SK	PS			CC	_mi	0x5	(N)							
	inst	ruction	s.		C	c_lt	0x6	(N xor V	()						
					CC	_ge	0x7	(not (N	xor V))						
					CC	c_gt	0x8	(Not (Z	or (N xo	or V)))					
					C	c_le	0x9	(Z or (N	xorV))						
					cc	_nv	0xa	(not V)							
					C	C_V	0xb	(V)							
					C	c_hi	0xc	(not (C	or Z))						
					C	c_la	0xd	(C or Z)							
				Add	litional	alias	flags al	lowed:							
				cc_	cs = cc	_c	cc_n =	cc_mi	cc_c	c = cc_	nc	cc_vc =	cc_nv		
				cc_	eq = cc	_z	cc_vs :	= cc_v	cc_n	e = cc_	_nz	cc_p =	cc_pl		
				Coc	les mea	an ex	ecute if	. For exa	ample, i	fs cc_e	eq mea	ans exec	cute if e	qual	
Instru	ction F	ormat	:	i4w											
Instru	ction F	ields:		IMN	14 = 4-1	oit im	mediate	e value							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	0		IM	M4	

## JMP

### **Computed Jump**

Operation:	$PC \leftarrow (RA \ll 1)$
Assembler Syntax:	JMP %rA
Example:	JMP %o7 ; return
	NOP ; (delay slot)
Description:	Jump to the target-address given by (RA << 1). Note that the target address is always half-word aligned for any value of RA.
Condition Codes:	Flags: Unaffected
	N V Z C
Delay Slot Behavior:	The instruction immediately following JMP (JMP's delay slot) is executed after
	JMP, but before the destination instruction. There are restrictions on which
	instructions may be used as a delay slot (see "Branch Delay Slots" on page 42).
Instruction Format:	Rw
Instruction Fields:	A = Register index of operand RA
15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1	1 1 1 1 1 1 0 A

# Load 32-Bit Data From Memory

Opera	tion:			Not RA Pre RA	prece ← Men ceded ← Men	ded by Pl 132[align32 by PFX o 132[align32	<b>FX:</b> 2(RB)] or <b>PFX</b> 2(RB +	 ί <b>ΙΟ:</b> + σ(K):	× 4))]						
Assen	nbler S	Syntax:		LD	%rA,[	%rB]									
Exam	ple:			Not	prece	ded by Pl	FX:								
				LD Pre	%g0,[ <b>ceded</b>	%i3] <b>by PFX:</b>	;	load	word	at [	[%i3] i	nto %	g0		
				PFX	. 7		;	offs	et by	7 wc	ords				
				LD Pre	%g0,[ <b>ceded</b>	%i3] <b>by PFXIC</b>	; ):	load	word	at [	[%i3+28	] int	:o %g(	)	
				PFX	IO 0		;	forc	es LD	to k	oypass	the d	lata d	cache	
				LD	%g0,[	%i3]	;	load	word	at [	[%i3] i	nto 🖁	g0		
Descr	iption:			Not	prece	ded by Pl	FX:								
	If the cac bit ma the	ne Nios s a data che, the data va data va y come cache	CPU a 32- alue from	Loa alig Pre The offs the Pre Pre	ds a 32 ned adi ceded value et is ad resultir ceded ceding Nios C	-bit data v dress give <b>by PFX:</b> in K is sig ded to the g word-al <b>by PFXIC</b> LD by PF3 :PU with a	value f en by l n-exte base igned ): XIO is a data	rom m bits 31 ended a e-addre addre just like cache	emory i 2 of Ri and use ess RB i ss. e precei , the LD	nto RA B (the ed as a (bits 1 ding L ) bypa	A. Data is two LSE a word-se 0 ignore D by PF> asses the	s alway 3s of R caled, s ed), an ( (see a cache	s read B are i signed d data above)	offset. is read	word- ). This from t that,
Condi	tion C	odes:		Flag	gs: Una	ffected									
				N _	V -	Z C 									
Instru	ction F	ormat		RR											
Instru	ction F	ields:		A = B =	Regist Regist	er index o er index o	f oper f oper	and R. and R	A B						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0			В					А		

## LDP

### Load 32-Bit Data From Memory (Pointer Addressing Mode)

Opera	ition:			Not	preced	ded by	PFX:								
				RA	← Mem	32[alig	n32(RP -	+ (IMM	$(5 \times 4))$	]					
				Pred	ceded	by PFX	( or PF)	KIO:							
				RA	← Mem	32[alig	n32(RP -	+ (σ(K :	IMM5	() × 4))]					
Asser	nbler	Syntax:	:	LDP	%rA,	[%rP,	IMM5]								
Exam	ple:			Not	preced	ded by	PFX:								
				LDP	%03,	[%L2,	3]	; Loa ; sec	d %o3 ond r	from egist	1 [%L2 er op	+ 12 erand	] must r &L3	be	
				Pred	ceded	by PF)	<b>(</b> :	, 0110	OL C	ло, с	ы <b>т</b> , о	12,0	1 0115		
				PFX	%hi(	100)									
				LDP Pred	, <sub>%o3</sub>	[%L2, <b>by PF</b>	%lo(10 <b>(IO</b> :	0)]	; 1	oad %	o3 fr	om [%	L2 + 4	400]	
				PFX	IO %h	i(100	)								
				LDP	%03,	[%L2,	%lo(10	0)]	; 1	oad %	o3 fr	om [%	L2 +	400]	
Descr	iption	:		Not	preced	ded by	PFX:								
	If t ha ca bit ma the	the Nios is a data che, the c data va ay come e cache	CPU a 32- alue from	aligr a 5-l 32-b addi in a %L1 <b>Pred</b> A 16 The addd <b>Pred</b> Pred Pred	ned ado bit, uns bit data tionally single , %L2, ceded S-bit off: 16-bit ed to bit ceded ceding in a Ni	dress g igned, v value r allows instruct or %LC <b>by PF</b> set is fc offset ( its 312 <b>by PF</b> by PF)	word-sc may con a positi ion. The 3. (: ormed by K : IMM 2 of RP (IO: PFXIO	bits 31. aled off ne from ve 5-bit base- y conca 5) is sig to yield is just l	2 of F set giv the ca offset pointe tenatir gn-exte a wor ike pre	RP (the ren by I ache. T to be a r must ng the 1 ended t d-align ecceding	two LS MM5. If his inst pplied be one 1-bit K o 32 bi ed effe	Bs of F Nios h ruction to any c of the f -registe ts, mult ctive ac	RP are i as a da is simil of four b four reg er with II tiplied b ddress. (see ab	gnored ta cach ar to L base-po jisters: MM5 (5 by four, bove), e	bits). and
Cond	tion (	o doou		Elec		ffootod	, with a		ione, i		bypas	303 110		•	
Cond		odes:		N	V -	Z C									
Instru	ction	Format	:	RPi	5										
Instru	ction	Fields:		A =	Registe	er index	c of ope	rand R/	4						
				IMM	l5 = 5-k	oit imme	ediate v	alue							
				P =	Index o	of base	-pointer	registe	r, less	16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	F	)			IMM5					Α		

## LDS

### Load 32-Bit Data From Memory (Stack Addressing Mode)

Operation:			RA ←	Mem32	2[align3	32(%sp	) + (IMN	$(18 \times 4)$	)]					
Assembler S	yntax:		LDS 8	srA,[8	sp,I	MM8]								
Example:			LDS १	o1,[%	sp,3	]	; loa ; sec	d %o1 ond r	from regist	stacl	k + 12 n only	2 7 be <sup>9</sup>	tsp	
Description:			Loads	a 32-b	it data	value	from me	emory	into RA	. Data i	s alway	/s read	from a	word-
If the has cau	ne Nios a data che, the	CPU 32-	aligne an 8-b	d addre it, unsi	ess giv gned,	en by l word-s	oits 31 scaled c	2 of %: offset g	sp (the given by	two LSE / IMM8.	3s of %	sp are i	gnorec	l) plus
bit ma the	bit data value may come from the cache. Conventionally, software uses %o6 (aka %sp) as a stack-per single-instruction access to any data word at a known offse above %sp.											inter. L in a 1K	DS allo byte ra	ows Inge
Condition Co	odes:		Flags:	Unaffe	ected									
			N _	V Z 	C _	]								
Instruction F	ormat	:	Ri8											
Instruction F	ields:		A = Re IMM8	egister = 8-bit	index immed	of ope diate v	rand R/ alue	A						
15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 1	1				IMM	8						А		

# LRET

Equivalent to JMP %07

Operation:		PC ·	← (%07	/ << 1)									
Assembler Syntax	:	LRE	T										
Example:		LRE	ΞT	; re	turn								
		NOF	þ	; (đ	elay	slot)							
Description:		Jum alwa	np to th ays hai	e targe f-word	t-addre aligned	ss give for any	n by (% / value	607 << of %07	1). Note ′.	e that th	ne targe	et addro	ess is
Condition Codes:		Flaç	gs: Una	ffected	l								
Delay Slot Behavio	or:	N – The	V –	Z C 	; - Imediat	ely follo	wing Ll	RET (L	RET's (	delay sl	ot) is e	xecuted	d after
		LRE	ET, but	before	the des	stinatio	n instru	ction. T	here a	re restr	ictions	on whic	ch
		inst	ructions	s may l	be used	l as a d	elay slo	ot (see	"Branc	h Delay	v Slots"	on pag	je 42).
Instruction Format	:	Rw											
Instruction Fields:		Nor	ne (alwa	ays use	es %o7)	)							
15 14 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 1 1	1	1	1	1	1	1	1	0	0	1	1	1	1

## LSL Logical Shift Left

32-Bit Instruction Set

Operat	tion:			RA	← (RA	<< RB[4	40]), z	ero-fill f	rom rig	sht					
Assem	nbler S	yntax:		LSL	%rA,	%rB									
Examp	ole:			LSL	%L3,	%g0	;	Shift	%L3	left	by %g	0 bit	s		
Descri	ption:			The 31!	value i 5 of RE	in RA is 3 are igr	shiften nored).	d-left by	the nu	imber c	of bits ir	ndicate	d by R	B [40]	(bits
					31	30 29							2 1	0	
						<b>▲</b>						•••	• •		D
Condit	tion Co	odes:		Flag N	ıs: Una V –	ffected Z C 									
Instruc	ction F	ormat:		RR											
Instruc	ction F	ields:		A = B =	Registe Registe	er index er index	c of RA	operan operan	d d						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0			B					۸		

81

# LSLI

### Logical Shift Left Immediate

Opera	tion:			RA	← (RA	<< IM1	M5), ze	ro-fill fro	m righ	t					
Asser	nbler S	Syntax:		LSI	I %rA	, IMM5									
Exam	ple:			LSL	JI %i1	,6	;	Shift	%i1 ]	left k	by 6 b	its			
Descr	iption:			The	value	in RA i	s shifte	ed-left by	the n	umber	of bits ir	ndicate	d by IN	1M5.	
					31	30 29 • <b>-</b> •						••••••	2 1	0	0
Condi	ition C	odes:		Flag	gs: Una	ffected	I								
				N _	V _	z c	-								
Instru	ction F	ormat		Ri5											
Instru	ction F	ields:		A = IMN	Regist 15 = 5-l	er inde bit imm	x of RA ediate	A operar value	nd						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1			IMM5					А		

## LSR Logical Shift Right

Opera	tion:			RA «	- (RA	>> RB[	40]), z	ero-fill t	from le	ft					
Assen	nbler S	yntax:		LSR	%rA,	%rB									
Examp	ole:			LSR	%L3,	%g0	;	Shift	%L3	right	by %	g0 bit	ts		
Descri	iption:			The RB[(	value i 315] a	in RA is are igno	s shifteo ored). T	d-right b he resu	by the alt is ze	number ero-fillec	of bits I from tl	indicate ne left.	ed by F	RB [40	] (bits
					: L	31 30	29						2	1 0	1
Condi	tion Co	odes:		Flag	s: Una	ffected									
				N _	V -	Z C									
Instru	ction F	ormat:		RR											
Instru	ction F	ields:		A =	Regist	er inde>	c of RA	operar	d						
				B =	Regist	er inde>	c of RB	operar	d						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0			В					A		

## LSRI

### Logical Shift Right Immediate

Opera Assen Examp	tion: nbler S ple: intion:	Syntax:		RA LSR LSR The	← (RA I %rA I %g1	>> IMM , IMM5 , 6 in <b>BA</b> is	I5), zer ; shifte	ro-fill from Right-:	n left shift	t %g1	by 6	bits	ed by li	мма т	he
Desch	puon.			resu	ult is lef	31 30 3	29						2		]
Condi	tion Co	odes:		Flag N	gs: Una V	Iffected Z C	٦								
Instrue Instrue	ction F ction F	ormat: ields:		Ri5 A = IMN	Regist 15 = 5-1	er index bit imme	of RA	A operane value	b						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1			IMM5					A		

# MOV

**Register-to-Register Move** 

Opera	tion:			RA	← RB										
Assen	nbler S	Syntax:		MOV	%rA,	%rB									
Exam	ple:			MOV	800,	%L3	;	сору	%L3 i	into १	00				
Descr	iption:			Cop	y the c	ontent	s of RB	to RA.							
Condi	tion Co	odes:		Flag	gs: Una	ffected	l								
			MOV $\$rA$ , $\$rB$ MOV $\$o0$ , $\$L3$ ; copy $\$L3$ into $\$o0$ Copy the contents of RB to RA.       Flags: Unaffected         N       V       Z       C         -       -       -       -         at:       RR       s:       A = Register index of RA operand         B       Register index of RB operand       A = Register index of RB operand         A       1       0       0       B       A												
Instru	ction F	ormat	:	RR											
Instru	ction F	ields:		A = B =	Regist Regist	er inde er inde	x of RA x of RE	opera opera	nd nd						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0			В					Α		

## MOVHI

#### Move Immediate Into High Half-Word

Operation:	<sup>h1</sup> RA $\leftarrow$ (K : IMM5), <sup>h0</sup> RA unaffected												
Assembler Syntax:	MOVHI %rA,IMM5												
Example:	Not preceded by PFX:												
	MOVHI %g3,23 ; upper 16 bits of %g3 get 23 Preceded by PFX:												
	PFX %hi(100) MOVHI %g3,%lo(100) ; upper 16 bits of %g3 get 100												
Description:	Not preceded by PFX:												
	Copy IMM5 to the most significant half-word (bits 3116) of RA. The least significant half-word (bits 150) is unaffected.												
	Preceded by PFX:												
	The immediate operand is extended from 5 to 16 bits by concatenating the												
	contents of the K-register (11 bits) with IMM5 (5 bits). The 16-bit immediate value												
	(K : IMM5) is copied into the most significant half-word (bits 3116) of RA. The												
	least significant half-word (bits 150) is unaffected.												
Condition Codes:	Flags: Unaffected												
	N V Z C												
Instruction Format:	Ri5												
Instruction Fields:	A = Register index of operand RA												
	IMM5 = 5-bit immediate value												
15 14 13 1	2 11 10 9 8 7 6 5 4 3 2 1 0												
0 1 1	1 1 IMM5 A												

### MOVI Move Immediate

Opera	tion:			RA	$\leftarrow (0x0)$	0.00 : K	C : IMM	15)							
Asser	nbler S	Syntax:		MOV	VI %rA	,IMM5									
Exam	ple:			Not	prece	ded by	PFX:								
				МО∖ <b>Рге</b>	7I %o3 <b>ceded</b>	, 7 by PF3	<b>K</b> :		; 1	oad %o3	8 witł	ı 7			
				PFX MOV	K %hi( /I %o3	301) ,%lo(	301)		; 1	oad %o3	8 with	ı 301			
Descr	iption:			Not	prece	ded by	PFX:								
Condi	ition Co	odes:		Loa [0 <b>Pre</b> Loa [06 Flao	ds regi 31]) giv <b>ceded</b> ds regi 65535]) gs: Una V	ster RA en by I by PF3 ster RA given iffected Z C	A with a MM5. <b>K:</b> A with a by (K : I	a zero-e a zero-e IMM5).	xtenc	ded 5-bit ded 16-bi	immedi t imme	iate val diate va	ue (in tl alue (in	ne rang	e ge
				_	-										
Instru	ction F	ormat	:	Ri5											
Instru	ction F	ields:		A = IMN	Regist 15 = 5-l	er inde oit imm	x of R/ ediate	A operar value	nd						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1			IMM5					Α		

# **MSTEP**

### **Multiply-Step**

Opera	tion:			lf (F ther else	80[31] ≕ n R0 ← e R0 ←	= = 1) (R0 << (R0 <<	< 1) + F : 1)	<b>A</b>							
Assen	nbler S	yntax:		MST	'EP %r	A									
Exam	ple:			MST	'EP %g	1	; ac	cumula	ate pa	artial	-prod	uct			
Descr	iption:			Imp muli	lement: tiplican	s a sing d in RA	gle step . Resu	of an i It is acc	unsigne cumulat	ed mult ted into	iply. The %r0. R	e multip A is no	olier in <sup>c</sup> ot affect	%r0 an .ed.	d
				The entr resu	followi y, %r0 ılt is lef	ng cod and % t in %r(	e fragm r1 conta ).	ient imp ain the	olemen multipli	ts a 16 er and	-bit × 16 multiplie	-bit inte cand, r	o 32-bit especti <sup>,</sup>	multip vely. T	ly. On he
					SWAP MSTE MSTE A t MSTE ; 32	%r0 P %r1 P %r1 P %r1 otal of P %r1 -bit p	; Move <b>16 MS</b> produc	e mult <b>TEPs</b> ct lef	t in	er int %r0	o plac	ce			
Condi	tion Co	odes:		Flag	gs: Una	ffected									
				N _	V _	Z C									
Instru	ction F	ormat	:	Rw											
Instru	ction F	ields:		A =	Regist	er inde	x of ope	erand F	<b>R</b> A						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	0	0			А		

## MUL Multiply

Operation:	$R0 \leftarrow (R0 \& 0x0000.ffff) x (RA \& 0x0000.ffff)$
Assembler Syntax:	MUL %rA
Example:	MUL %i5
Description:	Multiply the low half-words of %r0 and %rA together, and put the 32 bit result into %r0. This performs an integer multiplication of two signed 16-bit numbers to produce a 32-bit signed result, or multiplication of two unsigned 16-bit numbers to produce an unsigned 32-bit result.
Condition Codes:	N         V         Z         C           -         -         -         -         -
Instruction Format:	Rw
Instruction Fields:	A = Register index of operand RA
15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1	1 1 1 0 1 0 1 A

# NEG

### **Arithmetic Negation**

Operat	tion:			RA ↔	- 0 -	- RA									
Assem	bler S	yntax:		NEG	%rA										
Examp	ole:			NEG	%04										
Descri	ption:			Nega	ate th	e val	ue of	RA. Pe	erform	two's c	omplen	nent ne	gation	of RA.	
Condit	ion Co	des:		Flag	s: Un	affec	ted								
				Ν	V	Ζ	С								
				-	-	-	-								
Instruc	tion F	ormat:		Rw											
Instruc	tion F	ields:		A = F	Regis	ter ir	dex o	of opera	and RA						
15	14	13	12	11	10	g	)	8	7	6	5	4	3	2	1

А

## NOP

### Equivalent to MOV %g0, %g0

Opera	tion:			Nor	e											
Assen	nbler S	yntax:		NOF	)											
Exam	ple:			NOF	)	; do	nothi	ng								
Descr	iption:			No	operati	on.										
Condi	tion Co	odes:		Flag	gs: Una	ffected										
				N V Z C												
Instru	ction F	ormat	:	RR												
Instru	ction F	ields:		Nor	e											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	

# NOT

### Logical Not

Operation:	$RA \leftarrow \sim RA$									
Assembler Syntax:	NOT %rA									
Example:	NOT %04									
Description:	Bitwise-invert the value of RA.									
Condition Codes:	Flags: Unaffected									
	N V Z C									
Instruction Format:	Rw									
Instruction Fields:	A = Register index of operand RA									

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

## **OR** Bitwise Logical OR

Opera	ition:			Not RA Pre RA	¢ prece ← RA   ceded ← RA	ded by RB by PF2 (0x00.0	<b>PFX:</b> <b>X:</b> 00 : K :	IMM5)							
Asser	nbler S	Syntax:		Not OR Pre PFX OR	prece %ra,% ceded %hi( %ra,%	ded by rb by PF const	PFX: X:								
Exam	ple:			Not OR Pre PFX OR	<pre>   prece   %i0,%   ceded   %hi(   %i0,% </pre>	ded by i1 <b>by PF</b> 3333) 10(33	<b>PFX:</b> <b>X:</b> 33)	;	OR %i	il int i0 wit	to %i0	3			
Descr	iption:			Not Log resu Pre The the exte RA.	prece ically-C ult in R ceded RB op conten ended t	ded by DR the A. by PF erand i ts of the o 32 bi	<b>PFX:</b> individu <b>X:</b> is repla e K-regi ts, then	ial bits i ced by ister (1 i bitwise	n RA w an imm I bits) w e-ORed	ith the rediate vith IMN I with R	corresp constar 15 (5 bit A. The	onding nt form s). This result	g bits in ed by o s 16-bi is writt	RB; si concat t value en bac	tore the enating is zero- k into
Condi	ition C	odes:		Flag N A N: F Z: S	gs: V – – Result t Set if re	Z = C $\Delta = -$ bit 31 sult is 2	- zero; clo	eared o	therwis	6e					
Instru	ction F	ormat		RR,	Ri5		, -								
Instru	ction F	ields		A = B = IMN	Regist Regist 15 = 5-l	er inde er inde bit imm	x of op x of op ediate	erand F erand F value	RA RB						
Not p	recede	d by P	FX (RR	l)											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0			В					Α		
Prece	ded by	PFX (	Ri5)												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0			IMM5					Α		

## PFX

### Prefix

Opera	tion:			K ←	$K \leftarrow IMM11$ (K set to zero by all other instructions)													
Assen	nbler S	yntax:		PFX	IMM12	1												
Exam	ple:			PFX	3	; ā	affec	ts nez	ct ins	struct	ion							
Descr	iption:			Loa K re othe defi	ds the 1 gister n er than F ned.	1-bit conay affe PFX and	onstan ect the d PFX	nt value next in IO. The	IMM11 structio result o	into th on. K is of two c	e K-reg set to z onsecu	iister. T zero aft itive PF	he valu er ever X instr	ue in the y instru uctions	e Iction is not			
Condi	tion Co	odes:		Flags: Unaffected N V Z C														
Instru	ction F	ormat	:	i11														
Instru	ction F	ields:		IMN	111 = 1	I-bit im	media	te value	)									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
1	0	0	1	1	1 IMM11													

## **PFXIO** Prefix with Cache Bypass

32-Bit Instruction Set

Operation: Assembler Syntax:		$K \leftarrow$ PFX	IMM11 IO IMM	(K set to 11	o zero	by all of	her ins	structions	5)				
Example:		PFX	IO 3	;	affe	ects n	ext :	instru	ction				
Description:	e K-reg t to zei e usec before	jister. T ro after d immed any oth	he valu every i diately her inst	ue in th nstruct before truction	e K ion either is								
Condition Codes:		PFX data Flag	IO force cache ( s: Unaff	s the su if prese ected	ubseq ent), e	uent LC ven if th	) or L[ e data	DP mem a-cache	ory-loa is ena	ad oper bled.	ation to	bypas	s the
		N _	V Z	C -	]								
Instruction Format:		i11											
Instruction Fields:		IMM	11 = 11	bit imm	nediate	e value							
15 14 <u>1</u> 3	12	11	10	9	8	7	6	5	4	3	2	1	0
1 0 0	1	0 IMM11											

# RDCTL

### **Read Control Register**

Opera	tion:			RA	← CTL	k									
Asser	nbler S	yntax:		RDC	TL %r	A									
Exam	ple:			Not	prece	ded by	PFX:								
				RDC Pre	TL %g ceded	7 by PFX	; <b>K:</b>	Loads	%g7	from S	STATUS	s reg	(%ctl	0)	
				PFX RDC	2 TL %g	7	;	Loads	%g7	from W	WALII	) reg	(%ctl	2)	
Descr	iption:			Not	prece	ded by	PFX:								
				Loa <b>Pre</b> Loa "Co	ds RA <b>ceded</b> ds RA ntrol Re	with the <b>by PF</b> with the egisters	e currei <b>K:</b> e currei s" on pa	nt conte nt conte age 16	ents of ents of for a li	the STA the con	ATUS r trol reg ntrol rec	egister jister s gisters	<sup>r</sup> (%ctl0) elected and the	by K. S ir indic	see es.
Condi	tion Co	odes:		Flac	as: Una	ffected						<b>,</b>			
				N 	V -	Z C	;								
Instru	ction F	ormat		Rw											
Instru	ction F	ields:		A =	Regist	er inde	x of op	erand F	RA						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	0	1			А		

## RESTORE

### **Restore Caller's Register Window**

Opera	tion:			CW if (o ther	P ← C\ ld-CWI ו TRAF	VP + 1 P == HI P #2	LIMIT	.)							
Assen	nbler S	Syntax:		RES	TORE										
Exam	ple:			RES	TORE	;	bump	up tl	ne reg	gister	wind	low			
Descr	iption:			Mov (froi ove	ves CW m the V rflow tr	/P up b VVALIE ap (TR	y one p ) regist AP #2)	oosition er) befo is gene	in the i bre the erated.	register RESTC	file. If DRE ins	CWP is struction	equal n, then	to HI_L a wind	-IMIT ow-
Condi	tion Co	odes:		Flag N	gs: Una V	Iffected Z C	 ; -								
Instru	ction F	ormat	:	w											
Instru	ction F	ields:		Nor	ie										
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	1	1	0	1	0	0	0	0	0

# RET

### Equivalent to JMP %i7

Operation:	$PC \leftarrow (\%i7 \ll 1)$
Assembler Syntax:	RET
Example:	RET ; return
	RESTORE ; (restores caller's register window)
Description:	Jump to the target-address given by (%i7 << 1). Note that the target address is always half-word aligned for any value of %i7.
Condition Codes:	Flags: Unaffected
	N V Z C 
Delay Slot Behavior:	The instruction immediately following RET (RET's delay slot) is executed after
	RET, but before the destination instruction. There are restrictions on which
	instructions may be used as a delay slot (see "Branch Delay Slots" on page 42).
Instruction Format:	Rw
Instruction Fields:	None (always uses %i7)
15 14 13 1	2 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1	1 1 1 1 1 1 1 0 1 1 1 1 1

## **RLC** Rotate Left Through Carry

32-Bit Instruction Set

Operation	on:			C ← RA	- RA[31 ← (RA	l] << 1) :	: C								
Assemb	bler S	yntax:		RLC	%rA										
Example	e:			RLC	%i4	;	rota	te %i	4 left	one	bit				
Descrip	tion:			Rota	ates the	e bits o	of RA le	ft by on	e posit	ion thro	ough th	e carry	flag.		
31 30 29 2 1 0														0	
Conditie	on Co	odes:		Flag N - C: E	gs: V –   Bit 31 o	Z C – ∆ f RA be	C A Defore ro	tating							
Instruct	tion F	ormat:		Rw				-							
Instruct	tion F	ields:		A =	Regist	er inde	x of ope	erand F	<b>A</b>						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	0	1	0	1			Α		

## RRC

### **Rotate Right Through Carry**

**Operation:** 

RA  $\leftarrow$  C : (RA >> 1)Assembler Syntax:RRC %rAExample:RRC %i4 ; rotate %i4 right one bitDescription:Rotates the bits of RA right by one position through the carry flag.

 $C \leftarrow RA[0]$ 



If Precede Condition	d by PF) Codes:	<b>(</b> :	Ur Fla N	naffecte ags: I V - –	ed Z	C A								
			C:	Bit 0 c	of RA be	efore ro	otating							
Instruction	Instruction Format:													
Instruction	Instruction Fields:			= Regi	ster ind	ex of o	perand	RA						
15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 1	1	1	1	1	0	0	1	1	0			А		

## SAVE

### Save Caller's Register Window

Opera	ation:			CW	$P \leftarrow CV$	WP – 1									
				%sp	o ← %fp	) – (IMI	$(48 \times 4)$								
				lf (o	d-CWI	> == L(	D_LIMI	T)							
				ther	n TRAF	° #1									
Asser	nbler S	Syntax:		SAV	/E %sp	,-IMM	8								
Exam	ple:			SAV	/E %sp	,-23		; sta	art s	subrout	ine w	ith n	ew re	gs	
								; fin	rst d	operanc	l can (	only i	be %s	р	
Descr	iption:			Mov (froi und	ves CW m the V lerflow t	'P dowr VVALIE trap (Ti	n by one D regist RAP #1	e positic er) befc ) is ger	on in th ore the nerate	he registe e SAVE i ed.	er file. If nstructi	CWP is	s equal n a wir	to LO_ ndow-	LIMIT
%sp (in the newly opened register window) is loaded with the value of %fp m IMM8 times 4. %fp in the new window is the same as %sp in the old (caller window.													minus er's)		
				SA\ disp fran	/E is co oosable ne	onventi set of	onally ι register	used up rs for the	on en e subr	itry to su outine a	broutine nd simu	es to op Itaneou	ben up usly ope	a new, en up a	stack-
Condi	ition C	odoe		Flag	ne Ilna	ffoctor	1								
Conu		oues.		1 14	ys. 011a										
				N		2 (	;								
				_	-		-								
Instru	ction F	ormat	:	i8v											
Instru	ction F	ields:		IMN	/18 = 8-1	oit imm	ediate	value							
15	14	10	10	44	10	0	0	7	e	F	4	2	0	4	0
10	14	13	12	11	10	9	0	/	0	5	4	3	2	I	U
0	1	1	1	1	0	0	0				IMI	NB			

## SEXT16

Sign Extend 16-bit Value

Operation:	$RA \leftarrow \sigma(^{h0}RA)$
Assembler Syntax:	SEXT16 %rA
Example:	SEXT16 %g3 ; convert signed short to signed long
Description:	Replace bits 1631 of RA with bit 15 of RA.
Condition Codes:	Flags: Unaffected
	N V Z C 
Instruction Format:	Rw
Instruction Fields:	A = Register index of operand RA

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

### SEXT8 Sign Extend 8-bit Value

Operation:	$RA \leftarrow \sigma(^{b0}RA)$													
Assembler Syntax:	SEXT8 %rA													
Example:	SEXT8 %o3 ; convert signed byte to signed long													
Description:	Replace bits 831 of RA with bit 7 of RA.													
Condition Codes:	Flags: Unaffected													
	N V Z C 													
Instruction Format:	Rw													
Instruction Fields:	A = Register index of operand RA													
	Gran and Andrews													
15 14 13 12	11 10 9 8 7 6 5 4 3 2 1													
0 1 1 1	1 1 0 0 0 1 1 A													

0

## **SKPO**

#### Skip If Register Bit Is O

(Equivalent to IF1 Instruction)

Opera	tion:			if (R	RA[IMM	[5] == 0	))								
				then	begin										
					if (Me	m16[PC	C+2] is	s PFX or	PFXIO	)					
					then P	$C \leftarrow PC$	C + 6								
					else PO	$C \leftarrow PC$	2+4								
				end											
Asser	nbler S	Syntax:		SKI	90 %rA	,IMM5									
Exam	ple:			SKI	20 %g6	,11	;	skip	if b	it 11	is cl	ear			
				ADI	DI %6,	1	;	incr	ement	if bi	it 11	is se	t		
Descr	iption:			Ski	p next ii	nstructi	ion if th	ne single	e bit RA		] is 0. If	the ne	xt instru	uction i	s PFX
	•			or F	PFXIO.	then bo	oth PF	X or PF	XIO an	d the in	Istructio	n follov	wina PF	X or F	FXIO
				are	skippe	d toget	her.						5		
Condi	ition Co	odes:		Fla	ns <sup>.</sup> Una	iffected									
oona		00001		N 1	yo. ona	7 0									
				IN		<u> </u>	,								
				-	-										
Instru	ction F	ormat	:	Ri5											
Instru	ction F	ields:		Δ -	Regist	er inde	x of or	erand F	A						
motra	ouoni	icius.			15 - 5-k	oit imm	n or op adiata	valuo							
					/10 = 0-1	JILIIIIII	eulate	value							
45	4.4	10	10		10	0	0	7	0	-	4	0	0		0
15	14	13	12	11	10	9	8	1	6	5	4	3	2	1	U
0	1	0	1	0	0			IMM5					Α		

SKP1

(Equivalent to IFO Instruction)

Opera	tion:			if (R	A[IMN	[5] == 1	)								
				then	begin										
					if (Me	m16[PC	2+2] i	s PFX or	PFXIO	)					
					then P	$C \leftarrow PC$	2 + 6								
					else P	$\mathbb{C} \leftarrow \mathrm{PC}$	+4								
				end											
Asser	nbler S	Syntax:		SKE	01 %rA	,IMM5									
Exam	ple:			SKE	v1 %o3	,21	;	skip	if 21	lst bi	t of	%o3 i	s set		
				ADI	)I %g0	, 1	;	incre	ement	if 21	st bi	t is	clear		
Descr	iption:			Skip or F	o next i PFXIO,	nstructi then bo	on if th oth PF	ne single X or PFX	bit RA	(IMM5] d the in	is 1. If structio	the ne	xt instru wing PF	uction i X or P	s PFX FXIO
				are	скірре	a togeti	ner.								
Condi	tion Co	odes:		Flag	gs: Una	ffected									
				N _	V -	Z C	;								
Instru	ction F	ormat		Ri5											
Inetru	ction F	iolde		Δ _	Rogist	or indo	v of or	orand B	Δ						
motru	cuoni	ieius.			16931 16 – 5 I	hit imm	n di of		7						
					15 = 5-1		euiale	value							
15	14	10	10	44	10	0	0	7	6	F	1	2	0	4	0
15	14	13	12	11	10	Э	8	1	0	3	4	3	2	I	U
0	1	0	1	0	1			IMM5					Α		

## SKPRNZ

#### Skip If Register Not Equal To 0

(Equivalent to IFRZ Instruction)

Opera	tion:			if (R	A != 0)										
•				then	begin										
					if (Me	m16[PC	2+2] is	PFX or	PFXIO	)					
					then P	$C \leftarrow PC$	C + 6								
					else Po	$\mathbb{C} \leftarrow PC$	2+4								
				end											
Assem	nbler S	yntax:		SKE	RNZ %	rA									
Examp	ole:			SKE	RNZ %	g3									
				BSF	Send	lIt	;	only (	call i	lf %g3	is z	ero			
				NOF	)		;	(dela	y slot	z) exe	ecuted	in e	ither	case	
Descri	ption:			Skip	o next ii	nstructi	on if RA	is not	zero. If	the nex	t instru	ction is	PFX or	PFXIC	), then
				botł	ו PFX מ	or PFX	O and	the inst	ruction	followi	ng PFX	and P	FXIO a	re skip	ped
				toge	ether.										
Condi	tion Co	odes:		Flag	gs: Una	ffected	l								
				Ν	V	z c	;								
				-	-										
				I											
Instrue	ction F	ormat		Rw											
Instrue	ction F	ields:		A =	Regist	er inde	x of op	erand F	RA						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	1	0			Α		

# SKPRZ

Skip If Register Equals O

Opera	tion:			if (RA == 0) then begin if (Mem16[PC + 2] is PFX or PFXIO) then PC $\leftarrow$ PC + 6 else PC $\leftarrow$ PC + 4											
Assen	nbler S	yntax:		SKPRZ %rA											
Examı	ple:			SKPRZ %o3 BSR SendIt ; only call if %o3 is not 0 NOP ; (delay slot) executed in either case											
Description:				Skip next instruction if RA is equal to zero. If the next instruction is PFX or PFXIO, then both PFX or PFXIO and the instruction following PFX or PFXIO are skipped together.											
Condi	tion Co	odes:		Flags: Unaffected N V Z C 											
Instru	ction F	ormat:	:	Rw											
Instru	ction F	ields:		A = Register index of operand RA											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	0			Α		

## SKPS

### **Skip On Condition Code**

Operation:				if (co then end	if (condition IMM4 is true) then begin if (Mem16[PC + 2] is PFX or PFXIO) then PC $\leftarrow$ PC + 6 else PC $\leftarrow$ PC + 4 end											
Assen	nbler S	SKF	SKPS cc IMM4													
Exam	ple:	SKF	- SKPS cc ne													
					BSR SendIt ; only call if Z flag clear											
					NOP ; (delay slot) executed in either case											
Descr	iption:	Skip PFX skip	Skip next instruction if specified condition is true. If the next instruction is PFX or PFXIO, then both PFX or PFXIO and the instruction following PFX or PFXIO are skipped together.													
Condition Codes:				Settings:												
					cc_c 0x0 (C)											
	The		-111		CC	_nc	0x1	(not C)								
	COC	ese con les hav	aition e		c	c_z	0x2	(Z)								
	diff	erent			cc_nz 0x3 (not Z)											
	nur for	neric va IES and	alues 1		CC	_mi	0x4	(N)								
SKPS					cc_pl			(not N)	(not N)							
	inst	truction	s.		CC	_ge	0x6	(not (N :	kor V))							
					cc_lt			(N xor V	(N xor V)							
					cc_le			(Z or (N	(Z or (N xor V))							
					cc_gt			(Not (Z or (N xorV)))								
					cc_v			(V)	(V)							
					cc_nv			(not V)								
					cc_la			(C or Z)								
					cc_hi			(not (C	or Z))							
				Additional alias flags allowed:												
				cc_	$cc\_cs = cc\_c  cc\_n = cc\_mi  cc\_cc = cc\_nc  cc\_vc = cc\_nv$											
				cc_	cc_eq = cc_z cc_vs = cc_v cc_ne = cc_nz cc_p = cc_pl											
				Codes mean skip if. For example, skps cc_eq means skip if equal												
Instru	ction F	ormat		i4w												
Instru	ction F	ields:		IMN	14 = 4-1	bit im	mediate	e value								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	1	1	1	1	0	1	1	1	0	IMM4				
# Store 32-bit Data To Memory

Opera	tion:			Not Mer Pree Mer	prece n32[alig ceded b n32[alig	<b>ded by</b> gn32(RB <b>y PFX:</b> gn32(RB	<b>PFX:</b> $[B] \leftarrow R$ $[B] + (\sigma(K))$	4 )×4))	]←RA						
Assen	nbler S	yntax:		ST	[%rB]	,%rA									
Exam	ple:			Not	prece	ded by	PFX:								
				ST Pre	[%g0] <b>ceded</b>	,%i3 <b>by PFX</b>	; (:	%g0	is po	ointer	r, %i3	stor	ed		
				PFX	3		;	off	set by	7 3 wo	ords				
				ST	[%g0]	,%i3	;	sto	re to	locat	ion %	g0 +	12		
Descr	iption:			Not	prece	ded by	PFX:								
				alig Pre The offs	res the ned ad ceded value et is ad	dress g by PFX in K is s ded to	iven by (: sign-ext the bas	bits 3 endec e-poir	12 of F and us	emory. RB (the ed as a ress RI	two LS a word- B (bits 1	Bs of F scaled, 10 igne	s writte RB are signec ored), a	ignored ignored d offset. and dat	word- 1). This a is
				writ	ten to t	ne resu	Iting wo	ord-ali	gned ad	dress.					
Condi	tion Co	odes:		Flaç N –	gs: Una V –	ffected Z C 									
Instru	ction F	ormat		RR											
Instru	ction F	ields		A = B =	Regist Regist	er inde» er inde»	c of ope	rand I rand I	RA RB						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1			В					А		

### ST16D

### Store 16-Bit Data To Memory (Computed Half-Word Pointer Address)

Opera	ation:			Not <sup>hn</sup> M Pre <sup>hn</sup> M	em32[a ceded em32[a	ded by lign32(l by PF) lign32(l	<b>PFX :</b> RA)] ← <b>K:</b> RA + (σ	$^{hn}R0 w$ (K) × 4	'here n ≕ ))] ← <sup>hn</sup>	= RA[1	] ere n =	RA[1]			
Asser	nbler S	yntax:		ST1	.6D [%	rA],%	r0								
Exam	ple:			Not	prece	ded by	PFX:								
				FII ST1	L16 % .6D [%	r0,%g o3],%	7 r0	; di ; si ; %:	uplica tore % r0 to	ate sh \$03[1] [%03]	th sh	of %g7 Nort i	acro nt fr	ss %r( om %r0	0
				Pre	ceded	by PF)	<b>K</b> :	; 50	econa	opera	ina ca		уре	910	
				FIL PFX	L16 %	r0,%g	3								
				ST1	.6D [%	03],%	r0	; sa ; 2	ame as 0 byte	s abov es in	ve, of memor	fset			
Descr	iption:			Not	prece	ded by	PFX:								
				Stor add (hal	res one ress gi <sup>r</sup> f-word	of the ven by 1 is the	two ha RA. Th e most-s	lf-word: e bits F significa	s of %r( RA[1] se ant). RA	0 to me elects v A[0] is ig	emory a vhich h gnored	it the ha alf-word	alf-worc d in %r(	I-aligne ) get st	d ored
				ST16D may be used in combination with FILL16 to implement a two-instruction half-word-store operation. Given a half-word held in bits 150 of any register $%rX$ , the following sequence writes this half-word to memory at the half-word-aligned address given by RA:											
					FILL ST16	16 %r D [%r.	0,%rX A],%r(	)							
				Pre	ceded	by PF)	<b>(</b> :								
				The offs add	value et is ad ress.	in K is a Ided to	sign-ex the bas	tended se-addr	and us ess RA	ed as a and d	a word- ata is w	scaled, vritten to	, signed o the re	l offset. sulting	This byte-
Cond	ition Co	odes:		Flag	gs: Una	ffected									
				Ν	V	z c	;								
Instru	iction F	ormat		Rw											
Instru	iction F	ields:		A =	Regist	er inde	x of ope	erand F	RA						
15	14	13	12	2 11 10 9 8 7 6 5 4 3 2 1 0											
0	1	1	1	2 11 10 9 8 7 6 5 4 3 2 1 0 1 1 1 1 0 0 0 1 A											

### **ST16S**

### Store 16-Bit Data To Memory (Static Half-Word-Offset Address)

Opera	tion:		Not <sup>hn</sup> Me Prec <sup>hn</sup> Me	preced em32[ali eded k em32[ali	led by ign32(R by PFX ign32(R	PFX: $A)] \leftarrow^{1}$ $A + (\sigma)$	$^{hn}R0$ wh (K) × 4)	lere n = )] ← <sup>hn</sup>	= IMM1 R0 wher	re n = I	MM1				
Asser	nbler S	Syntax:	ST1	- 6S [%1	A],%1	0,IMM	11	-							
Exam	ple:		ST1	6S [%g	98],%1	20,1									
Descr	iption:		<b>Not</b> Store giver whic	preced es one n by RA h half-v	l <b>ed by</b> of the t (IMN vord of	<b>PFX:</b> wo hali //1 x 2). %r0 is	f-words RA is p stored	of %r( presum (half-w	) to mer ed to ho ord #1	mory at old a wo is most	the ha ord-alig signific	lf-word ned ad cant).	I-aligneo Idress. I	d addre MM1 s	ss elects
			Prec A 12 offse as th addr	eded b -bit sign t (K:IM he half-v ess hel	oy PFX ned, ha M1) is word-al d in RA	alf-word half-wo ligned c A, which	I-scaled rd-scale offset fo n is pres	l offset ed (mu r the S sumed	is form Itiplied T opera to be w	ed by c by 2), s ation. T vord-aliq	oncate ign-ext his offs gned.	nating ended et is a	K with I to 32 bi pplied to	MM1. <sup>-</sup> its, and o the ba	This used ase-
			IMM (bas	1 selec e + offs	ts whic set).	h of the	e two ha	alf-wor	ds of %	r0 are s	stored a	at the ii	ndicated	d addre	SS
			(base + offset). ST16S may be used in combination with FILL16 to implement a half-word store operation to a half-word offset from a word-aligned base address. Given a half-word held in bits 150 of any register $%rX$ , the following sequence writes this half-word to memory at the half-word-aligned address given by RA + Y, where RA is presumed to hold a word- aligned pointer, and Y is an even, signed 13-bit byte offset:												
			FILL PFX ST16	16 %r0 Ymsbs S [%rA	),%rX A], %r(	), Y1	; Top ; Bit	11 bit 1 of	s of Y Y (= Y	, incl & 2)	. sign	bit.	(= (Y>>	≥2) & 0	x7FF)
			١	∕ (offset	12 t) =	11	10 9	8 Y	76 ímsbs -	5	4 3	2	1 0 Y1 0		
Condi	tion C	odes:	Flag N –	s: Unaf V 2	fected Z C 	]									
Instru	ction F	ormat:	Ri1u												
Instru	ction F	ields	A = I IMM	Registe 1 = 1-b	r index it imme	of ope ediate v	rand R/ alue	Ą							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	1	0	1	IMM1	0			А		
U	I	I	I	0	I	I	U	I		U			A		

### ST8D

### Store 8-Bit Data To Memory (Computed Byte-Pointer Address)

Opera	tion:			Not	preced	ed by P	FX:								
-				<sup>bn</sup> Mo Pree	em32[al <b>ceded</b>	lign32(I <b>by PFX</b>	RA)] ← 【:	<sup>bn</sup> R0 v	where n =	RA[1	0]				
				<sup>bn</sup> M	em32[al	lign32(H	$RA + \sigma($	$(K) \times 4$	$))] \leftarrow {}^{bn}F$	R0 wher	e n = R/	<b>A</b> [10]			
Assen	nbler S	yntax:		ST8	D [%r/	A],%r(	)								
Exam	ple:			Not	prece	ded by	PFX:								
				FIL ST8	L8 %r D [%o	0,%g7 3],%r	0	; c ; s ; s	duplica store % %r0 to	ate lo 803[1. [%03]	w byt .0]th	e of byte	%g7 a from	cross	%r0
				Pre	ceded	by PF)	<b>(</b> :	; :	second	opere	iliu ca	.11 0111	y be	010	
				FIL PFX	L8 %r 5	0,%g3									
				ST8	D [%o	3],%r	0	; :	same as	s abov	ve, of	fset			
								; 2	20 byte	es in	memor	У			
Descr	iption:			Not	prece	ded by	PFX:								
				Stor	res one	of the	four by	tes of	%r0 to r	nemory	at the	byte-ac	ddress	given b	y RA.
				The	two bi	ts RA[1	0] sel	ect wł	nich byte	in %r0	get sto	ored (by	/te 3 is	the mo	st-
				sigr	ificant)										
				ST8 byte follo	BD may e-store owing s	be use operati equenc	ed in co on. Giv ce write	mbina en a t s this	ation with byte held byte to r	n FILL8 I in bits memory	to impl 70 of at the	ement any reg byte-ad	a two-i gister % ddress	nstructi %r <i>X</i> , the given b	on e y RA:
					FTLL	8 %r0	.%rX								
					ST8D	[%rA	],%r0								
				Pre	ceded	bv PF)	<b>K</b> :								
				The	value i	n Kisı	used as	a wor	d-scaled	l, signe	d offset	t. This o	offset is	added	to the
				bas	e-addre	ess RA	and da	ta is v	written to	the re	sulting	byte-ad	ldress.		
Condi	tion Co	odes:		Flac	ıs: Una	ffected					-	-			
				N	v	z c	;								
				_											
Instru	ction F	ormat	:	Rw											
Instru	ction F	ields:		A =	Regist	er inde	x of op	erand	RA						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	0	0			А		

### ST8S

### Store 8-bit Data To Memory (Static Byte-Offset Address)

Operation:	Not precede <sup>bn</sup> Mem32[alia <b>Preceded b</b> bnMem32[alia	ed by P gn32(RA y PFX: gn32(RA	<b>FX:</b> Δ)] ← <sup>1</sup> Δ + (σ	$p^{n}$ R0 wh (K) × 4)	ere n  = )] ← br	· IMM2 IR0 when	ren =	IMM2							
Assembler Syntax:	ST8S [%rA	],%r0,	IMM2												
Example:	Not precede MOVI %g4, ST8S [%g4 Preceded b PFX 9 ST8S [%g4	ed by P 12 ],%r0, y PFX: ],%r0,	<b>FX:</b> 3 2	;	store	high byte	byte 2 of	of %r %r0 t	0 to o	mem[1:	2]				
Description:	Not precede Stores one o is presumed (byte #3 is n	ed by P of the fo to hold nost sign	<b>FX:</b> ur byte a wor nificar	; es of % rd-align nt).	nem[% r0 to m ed add	g4 + 3 emory a ress. IM	6 + 2 at the a IM2 se	2] address elects wł	given hich by	by RA⊣ yte of %	⊦ (IMM2 r0 is st	2). RA ored			
	Preceded b A 13-bit sign sign-extende applied to th	<b>Exceded by PFX:</b> 3-bit signed offset is formed by concatenating K with IMM2. This offset (K:IMM2) is n-extended to 32 bits and used as the byte-offset for the ST operation. The offset is blied to the base-address held in RA, which is presumed to be word-aligned. <i>M</i> 2 selects which of the four bytes of %r0 are stored at the indicated address (base + offset). BS may be used in combination with FILL8 to implement a byte-store operation to any 13-													
	IMM2 selects	which c	of the f	our byte	s of %r	0 are sto	red at	the indic	ated a	ddress (	base +	offset).			
	ST8S may be bit signed off register %rX + Y, where R	e used in set from , the foll A is pres	n com i a wo owing sumed	bination rd-align sequen I to hold	with F ed base ce write a word	ILL8 to in a addres es this b -aligned	mplem s. Give yte to pointe	ent a by en a byte memory er, and Y	te-stor e held at the is a sig	re operat in bits 7 address gned 13-	tion to a 0 of ai given bit byte	any 13- ny by RA offset:			
	FILL8 %r0,9 PFX Ymsbs ST8S [%rA]	∛rX , %r0,	Ylsbs	; Top ; ; Bi	11 bit .ts 1 a	s of Y, and 0 o	incl f Y (	. sign 1 = Y & 3	bit. 3)	(= (Y>>	2) & (	)x7FF)			
	Y (offse	12 t) =	11 	10 9	8 )	7 6 ⁄msbs -	5	4 3	2	1 0 Ylsbs	]				
Condition Codes:	Flags: Unaff	ected													
	N V Z	C C													
Instruction Format:	Ri2u														
Instruction Fields:	A = Register IMM2 = 2-bi	r index o t immed	of ope liate v	rand R <i>i</i> alue	٩										
15 14 13	12 11	10	9	8	7	6	5	4	3	2	1	0			
0 1 1	1 0	1	1	0	0	IMN	И2			А					

### STP

### Store 32-bit Data To Memory (Pointer Addressing Mode)

Opera	tion:			No	t prece	ded by	/ PFX:									
				Me	m32[alig	gn32(Rl	P + (IMN	(15 × 4)	)] ← R.	A						
				Pr	eceded	by PF	X:									
				Me	em32[alig	gn32(Rl	$P + (\sigma(K$	: IMM	5) × 4))	$] \leftarrow RA$	L					
Asser	nbler S	Syntax:		ST	P [%rP	,IMM5	5],%rA									
Exam	ple:			No	t prece	ded by	/ PFX:									
				ST Pr	P [%L2 eceded	, ३], १ by PF	3g3 <b>X:</b>		; St	ore %	g3 to	locat	ion [	%L2 +	12]	
				PF	X %hi(	102)										
				ST	P [%L2	,%lo(	102)],	%g3	; S	tore 9	≹g3 to	)				
_									; ⊥	ocatio	on [%]	.2 + 4	08]			
Descr	iption:			NC	ot prece	ded by	/ PFX:		<b>-</b> • • • •		Data :					
				50	ores the	32-DI	data vai	bite [2	1 A to n 1 21 of	DD (th	. Data I	s aiway	/S Writte	ianoroc	wora-	
				a f	5-bit, uns	sianed.	word-se	caled o	offset o	iven bv		303 011		ignored	i) pius	
				ть	ia inatru		oimilori		hut odd				the F h	it offer	tta ha	
				in an	This instruction is similar to ST, but additionally allows a positive 5-bit offset to be applied to any of four base-pointers in a single instruction. The base-pointer must											
				applied to any of four base-pointers in a single instruction. The base-pointer must be one of the four registers: %L0, %L1, %L2, or %L3.												
				Dr	acadad		<b>y</b> .		-							
				A 1	l 6-bit off	set is f	<b>^.</b> ormed b	v conc	atenat	ina the '	11-bit K	-reaiste	ər with I	MM5 (5	5 bits).	
				Th	e 16-bit	offset	(K : IMN	) is s	ign-ext	tended	to 32 bi	its, mul	tiplied b	by four,	and	
				ad	ded to b	its 31	2 of RP	to yiel	d a wo	rd-align	ned effe	ctive a	ddress.			
Condi	ition Co	odes:		Fla	ags: Una	ffected	ł									
				Ν	V	zc	>									
				_	-		-									
Instru	ction F	ormat	:	RF	Pi5											
Instru	ction F	ields:		A =	= Regist	er inde	x of ope	erand F	RA							
				IMM5 = 5-bit immediate value												
				P = Index of base-pointer register, less 16												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	0	11     10     9     8     7     6     5     4     3     2     1     0       P     IMM5     A     A												

### STS

### Store 32-bit Data To Memory (Stack Addressing Mode)

Operation:	$Mem32[align32(\%sp + (IMM8 \times 4))] \leftarrow RA$
Assembler Syntax:	STS [%sp,IMM8],%rA
Example:	STS [%sp,17],%i5 ; store %i5 at stack + 68
	; first register can only be %sp
Description:	Stores the 32-bit value in RA to memory. Data is always written to a word-aligned address given by bits 312 of %sp (the two LSBs of %sp are ignored) plus an 8-bit, unsigned, word-scaled offset given by IMM8.
	Conventionally, software uses %o6 (aka %sp) as a stack-pointer. STS allows single-instruction access to any data word at a known offset in a 1 Kbyte range above %sp.
Condition Codes:	Flags: Unaffected
	N V Z C
Instruction Format:	Ri8
Instruction Fields:	A = Register index for operand RA IMM8 = 8-bit immediate value
15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1 1 0	IMM8 A

### STS16S

### Store 16-bit Data To Memory (Stack Addressing Mode)

Opera	tion:			hnM	<sup>n</sup> Mem32[align32(%sp + IMM9 × 2)] $\leftarrow$ <sup>hn</sup> R0 where n = IMM9[0] TS16S [%sp, IMM9], %r0 TS16S [%sp, 7], %r0 ; can only be %sp and %r0 Stores one of the two half-words of %r0 to memory at the half-word-align address given by (%sp plus IMM9 × 2). The least-significant bit of IMM9 which half-word of %r0 is stored (half-word 1 is most significant). STS16s may be used in combination with FILL16 to implement a 16-bit speration to a half-word offset from the stack-pointer in a 1 Kbyte range. I half-word held in bits 150 of any register %rX, the following sequence wr half-word to memory at the half-word-offset Y from %sp (%sp presumed a word-aligned address):													
Assen	nbler S	yntax:		STS	516S [	%sp,I	MM9],	%r0										
Exam	ple:			STS	316S [	%sp,7	],%rC	)	; c	an onl	y be	%sp a	nd %r	0				
Descr	iption:			Stor add whie	res one ress gi <sup>r</sup> ch half-	of the ven by word of	two ha (%sp p f %r0 i	alf-words blus IMN s stored	s of %i /19 × 2] I (half-	r0 to me ). The le word 1 i	emory a east-sig s most	t the hand nificant signific	alf-word t bit of I cant).	d-aligne MM9 so	d elects			
				STS ope half half a w	S16s m ration t -word h -word t ord-alig	ay be u o a half neld in b o memo ned ad	sed in -word hits 15. ory at t dress)	combin offset fro .0 of any the half- :	ation v om the / regis word-e	with FIL e stack-p ter %r <i>X</i> , offset <i>Y</i>	L16 to i pointer the foll from %	implem in a 1 K lowing s sp (%s	ent a 1 (byte ra sequen sp pres	6-bit sto inge. Gi ice write umed to	ore iven a es this o hold			
					FILL16 %r0,%rX													
					FILL16 %r0,%rX STS16s [%sp,Y],%r0													
Condi	tion Co	odes:		Flaç N	gs: Una V –	Iffected Z C – –												
Instru	ction F	ormat	:	i9														
Instru	ction F	ields:		IMN	19 = 9-l	oit imme	ediate	value										
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	1	1	0	0	1					IMM9					0			

### STS8S

### Store 8-bit Data To Memory (Stack Addressing Mode)

Operation:	<sup>bn</sup> Mem32[align32(%sp + IMM10)] $\leftarrow$ <sup>bn</sup> R0 where n = IMM10[10]										
Assembler Syntax:	STS8S [%sp,IMM10],%r0										
Example:	STS8S [%sp,13],%r0 ; can only be %sp and %r0										
Description:	Stores one of the four bytes of %r0 to memory at the byte-address given by (%sp plus IMM10). The two least-significant bits of IMM10 selects which byte of %r0 is stored (byte 3 is most significant).										
	STS8S may be used in combination with FILL8 to implement a byte-store operation to a byte-offset from the stack-pointer in a 1Kbyte range. Given a byte held in bits 70 of any register $%rX$ , the following sequence writes this byte to memory at the byte-offset <i>Y</i> from $%sp$ (%sp presumed to hold a word-aligned address):										
	FILL8 %r0,%rX STS8S [%sp,Y],%r0										
Condition Codes:	Flags: Unaffected										
	N V Z C 										
Instruction Format:	i10										
Instruction Fields:	IMM10 = 10-bit immediate value										
15 14 13 12 <sup>-</sup>	11 10 9 8 7 6 5 4 3 2 1 0										
0 1 1 0	0 0 IMM10										

# SUB

### Subtract

Operation:			RA	$\leftarrow$ RA -	- RB											
Assembler	Syntax:		SUB	%rA,	%rB											
Example:			SUB	%i3,	%g0		; SUB	%g0 fr	om %i	3						
Description	:		Sub	tracts t	he c	ontents	s of RB fro	om RA,	stores	result i	ו RA.					
Condition C	odes:		Flag	js:												
			Ν	V	Ζ	С										
			Δ	Δ	Δ	Δ										
			N· F	I: Result bit 31												
			V: S	N: Result bit 31 J: Signed-arithmetic overflow												
			Z: S	V: Signed-arithmetic overflow Z: Set if result is zero; cleared otherwise												
			C: 5	Set if th	ere v	vas a b	orrow fro	m the s	ubtract	ion; cle	ared ot	herwise	Э			
Instruction	Format		RR													
Instruction	Fields:		A =	Regist	er ind	dex of	RA opera	nd								
			B =	Regist	er ind	dex of	RB opera	nd								
15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0 0	0	0	1	0			В					А				

### SUBI Subtract Immediate

Opera	tion:			RA	← RA -	- (0x00	.00 : K :	IMM5)							
Assen	nbler S	yntax:		sub	oi %rB	,IMM5	;								
Exam	ple:			Not	prece	ded by	PFX:								
				SUE Pre PFX SUE	8I %L5 <b>ceded</b> [ %hi( 8I %o3	,6 <b>by PF</b> 1000) ,%10(	<b>X:</b>		; s	subtract	: 6 fr : 1000	rom %L ) from	.5 1 %03		
Descr	iption:			Not	prece	ded by	PFX:		,						
Condi	tion Co	odes:		Sub in th <b>Pre</b> Con (K : Flag	e rang ceded Immed tents of IMM5) gs:	he imn e of [0. <b>by PF</b> diate op the K- is zero	nediate 31]. X: perand i register p-extend	value fr s exten (11 bits led to 3	rom f ided s) wit 2 bit	the conter from 5 to th IMM5 ( ts and sub	nts of F 16 bits 5 bits). otracted	RA. The s by cor The 16 d from r	e immeo ncatena -bit imn register	diate va ating the nediate A.	alue is e value
				Δ N: F V: S Z: S C: S	Δ Result b Signed-a Set if res	$\Delta$ $\Delta$ bit 31 arithme sult is z ere wa	etic over zero; cle s a borr	flow ared of ow fron	therv n the	wise e subtracti	ion; cle	ared ot	therwise	e	
Instru	ction F	ormat	:	Ri5											
Instru	ction F	ields:		Ri5 A = Register index of RA operand IMM5 = 5-bit immediate value											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1			IMM5					Α		

### **SWAP**

### Swap Register Half-Words

Opera	tion:			RA ·	$\leftarrow {}^{h0}R$	$A: {}^{h1}RA$	1								
Assen	nbler S	Syntax:		SWA	AP %rA	ł									
Exam	ple:			SWA	AP %g3	3	; Ex	chang	e two	half-	words	in %	g3		
Descr	iption:	1		Swa rest	aps (ex ult bacl	change c into R	es posit A.	ions) o	f the tw	o 16-bi	t half-w	ord val	ues in F	א. Wr	ites
Condi	tion C	odes:		Flaç N	gs: Una V –	affected Z (	1 > -								
Instru Instru	ction F ction F	Format Fields:	:	Rw A =	Regist	er inde	x of op	erand F	RA						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	1	0	0	0			Α		

### TRAP Unconditional Trap

Opera	tion:			IST	ATUS ←	- STAT	US								
				IE ↔	-0	VD 1									
				$CWF \leftarrow CWF = 1$ $IPRI \leftarrow IMM6$											
				1FK	$\%07 \leftarrow ((PC + 2) >> 1)$										
				PC •	← Men	32[VE	CBASE	+ (IMN	/16×4)]	<< 1					
Assen	nbler S	yntax:		TRA	P IMM	6									
Exam	ple:			TRA	AP 2		;invo	ke CW	P wind	low ov	erflow	exce	ption	hand	ller
Descr	iption:			CWP is decremented by one, opening a new register-window for the trap-handler. Interrupts are disabled (IE $\leftarrow$ 0). The pre-TRAP STATUS register is copied into the ISTATUS register.											
				Trai is re (VE add tran TRA retu	is read from the vector table which starts at the memory address VECBASE (VECBASE is configurable). A 32-bit value is fetched from the word-aligned address (VECBASE + IMM6 $\times$ 4). The fetched value is multiplied by two and transferred into PC. The address of the instruction immediately following the TRAP instruction is placed in %o7. The value in %o7 is suitable for use as a return-address for TRET without modification. The return-address convention for TRAP is different than BSR/CALL, because TRAP does not have a delay-slot.										
				A T IE b	RAP in oit in the	structio e STAT	on trans US reg	fers ex lister is	ecution	to the i	ndicate	d trap-ł	nandler	<sup>.</sup> even i	f the
				TRAP 0 corresponds to the Nios CPU's non-maskable exception, and it behaves differently than exceptions 1 through 63. TRAP 0 cannot be issued by user software.											
Condi	tion Co	des:		Flags: Unaffected											
				N V Z C											
Delay	Slot B	ehavior		TRAP does not have a delay slot. The instruction immediately following TRAP is not executed before the target trap-handler. The return-address used by TRET points to the instruction immediately following TRAP.											
Instru	ction F	ormat:		i6v											
Instru	ction F	ields:		IMN	16 = 6-	oit imm	ediate	value							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	1	0	0			IMN	16		

## TRET

### Trap Return

Opera	ition:			$PC \leftarrow (RA \ll 1)$													
				STA	$STATUS \leftarrow ISTATUS$												
Assen	nbler S	Syntax:		TRE	TRET %ra												
Example: TRET %07 ; return from TRAP																	
Descr	iption:			Exe	Execution is transferred to the address given by (RA << 1). The value written in												
				%07	%o7 by TRAP is suitable for use as a return-address without modification.												
				The TR/	The value in ISTATUS is copied into the STATUS register (this restores the pre- TRAP register window, because CWP is part of STATUS).												
Condi	ition Co	odes:		Flaç	Flags: Unaffected												
				N _	V 	Z C	;										
Instru	ction F	ormat	:	Rw													
Instru	ction F	ields:		A =	Regist	er inde:	x of ope	erand F	₹A								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	1	1	1	0	1	1	1	0			А				

# USRO

**User-defined Instruction** 

<b>Operation:</b> $RA \leftarrow RA < user-defined operation > RB$													
Assembler Synta:	:	USF	USR0 %rA, %rB										
Example:		USF	20 %o1	,%i6									
Description:		The This	The user can implement a custom operation in hardware and assign it to USR0. This operation uses 2 registers and places the result in the RA register.										
A custom instruction can be single-cycle or multi-cycle. It can PFX command to pass in an optional 11-bit value for use wi hardware block. Flags: Unaffected N V Z C 												fixed wi	ith the n
Instruction Forma	t:	RR	RR										
Instruction Fields	:	A = B =	Regist Regist	er inde: er inde:	x of op x of op	erand F erand F	A B						
15 14 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 1 1	1	0	0			В					А		

### **USRx** [x = 1,2,3,or 4]

**User-defined Instruction** 

Operation:	$RA \leftarrow RA \leq user-defined operation > R0$
Assembler Syntax:	USRx RA
Example:	USR2 %03
Description:	The user can implement a custom operation in hardware and assign it to USR1, USR2, USR3 or USR4. This operation uses 2 registers but one of them is always %r0. The result is placed in RA.
	A custom instruction can be single-cycle or multi-cycle. It can be prefixed with the PFX command to pass in an optional 11-bit value for use within the custom hardware block.
Condition Codes:	Flags: Unaffected       N     V     Z       -     -     -
Instruction Format: Instruction Fields:	Rw A = Register index of operand A

USR1															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	1	0	0	1			А		
USR2															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	1	0	1	0			А		
USR3															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	1	0	1	1			А		
USR4															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	1	1	0	0			Α		

### WRCTL Write Control Register

Opera	tion:			$CTLk \leftarrow RA$											
Asser	nbler S	Syntax:	:	WRC	CTL %r	A									
Exam	ple:			Not preceded by PFX:											
				WRC NOE	CTL %g	7	; w ; r	rites equir	%g7 t ed	to STA	ATUS r	eg			
				Pre	ceded	by PF	<b>X</b> :								
				PFX WRC	PFX 1 WRCTL %g7     ; writes %g7 to ISTATUS reg										
Descr	iption:			Not preceded by PFX:											
				Loads the STATUS register with RA. WRTCL to STATUS must be followed by a											
				NOP instruction.											
				<b>Preceded by PFX:</b> Writes the value in DA to the machine control register colocted by K. See Table 2.											
				on page 15 for a list of the machine-control register selected by K. See Table 3											
Cand				If the target of WRCTL is the STATUS register, then the condition-code flags are											
Cond		baes:		If the target of WHO I L is the STATUS register, then the condition-code flags are											
				target register, the condition codes are unaffected.											
Instru	ction F	ormat		Rw											
Inotru	otion F	loldou		A Desister index of energed DA											
mstru	CUON	ieius:		A = Hegister index of operand HA											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	0	0			Α		

### XOR

#### **Bitwise Logical Exclusive OR**

Opera	ation:			Not preceded by PFX: $RA \leftarrow RA \oplus RB$ Preceded by PFX: $RA \leftarrow RA \oplus (0x00.00 : K : IMM5)$												
Asser	nbler S	Syntax:		Not XOF Pre PFX XOF	Not preceded by PFX: XOR %ra,%rb Preceded by PFX: PFX %hi(const) XOR %rA,%lo(const)											
Exam	ple:			Not XOF Pre PFX	Not preceded by PFX:           XOR %g0,%g1         ; XOR %g1 into %g0           Preceded by PFX:           PFX %hi(16383)         ; XOR %c0 with 16383											
Descr	ription:			Not Log stor Pre Whe con 16-l	prece ically-e ically-e the re ceded en pref catena oit valu	ded by exclusive esult in by PF ixed, the ting the e is ze	/ PFX: /e-OR t RA. X: ne RB o e conte ro-exter	he indiv perand nts of th nded to	; XOF idual bi is repla ie K-req 32 bits	ts in R aced by gister ( a, then	A with th an imr 11 bits) bitwise	ne corre mediate with IN -exclusi	espondi e consta 1M5 (5 ve-ORe	ng bits Int form bits). T ed with	in RB; ned by his RA.	
Cond	ition C	odes:		Flaç N A N: F	ys: V – – – – –	Z C $\Delta$ -		( Into Th	<b>.</b> .							
Inotru	otion E	ormat		Z: Set if result is zero, cleared otherwise												
Instru	iction F	ormat		KR,	RR, Ri5											
instru		ielas:		A = B = IMN	Regist Regist 15 = 5-	er inde er inde bit imm	ex of op ex of op nediate	erand F erand F value	RB							
Not p	recede	d by P	FX (RR	:)												
15	14	13	12	, 11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	1		-	В	-	-		-	А		-	
Prece	ded by	PFX (	Ri5)	1	1	1					1					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	1			IMM5					А			





#### **Symbols**

%g0 register 16 %r0 register 16

#### A

ABS instruction 52 Absolute Value 52 Absolute-jump instructions 32 Add Immediate 54 ADD instruction 53 ADDI instruction 27, 28, 54 Addressing modes 27 AND instruction 28, 55 ANDN instruction 28, 56 Arithmetic Negation 90 Shift Right 57 Shift Right Immediate 58 ASR instruction 57 ASRI instruction 28, 58

#### B

BGEN instruction 59 Bit Generate 59 Bitwise Logical AND 55 AND NOT 56 Exclusive OR 126 OR 93 BR instruction 16, 31, 42, 60 Branch 60 Branch delay slots 32, 41, 42 Branch To Subroutine 61 BSR instruction 16, 31, 42, 61 Byte-Extract dynamic 67 static 68 Byte-Fill 70

### C

Cache Memory 24 Initializing cache memory 26 CALL instruction 16, 32, 42, 62 Call Subroutine 62 CLR\_IE (%ctl8) 21 CMP instruction 63 CMPI instruction 28, 64 Compare 63 Compare Immediate 64 Computed Jump 76 Condition code flags 19 Conditional instructions 33 Conditionally Execute Next Instruction 75 Control registers 16 CPU\_ID (%ctl6) 20 Current window pointer. See CWP. CWP description 18 manipulation, direct 41

#### D

Data Cache Enable. See DC. Data memory 21 DC 17 DCACHE (%ctl7) 20 Disable Data Cache During the Following Load Instruction Prefix 95

### E

Equivalent to IF0 (SKP1) 105 IF1 instruction (SKP0) 104

IFRZ (SKPRnz) 106 JMP %i7 (RET) 98 JMP %07 (LRET) 80 MOV %g0, %g0 (NOP) 91 SKP0 instruction (IF1) 72 SKP1 instruction (IF0) 71 SKPRnz instruction (IFRz) 74 SKPRz instruction (IFRnz) 73 Exception handlers complex 40 simple 40 Exceptions 33 handling 33 internal 35 processing sequence 37 vector table 34 Execute if Register Bit is 071 Register Bit is 172 Register is not Zero 73 Register is Zero 74 EXT16D instruction 29, 65 EXT16S instruction 29, 66 EXT8D instruction 29, 67 EXT8S instruction 29, 68

### F

FILL16 instruction 23, 30, 69 FILL8 instruction 23, 30, 70 Full width register indirect 29 indirect with offset 30

### G

General-purpose registers 14 GNU compiler/assembler pseudo-instructions 50

#### H

Half-Word Extract, Dynamic 65 Extract, Static 66 Fill 69 Hardware interrupt, external sources 34

### I

IC 17 ICACHE (%ctl5) 20 IE 17 IF0 instruction 33, 42, 71 equivalent (SKP1) 105 IF1 instruction 33, 42, 72 equivalent (SKP0) 104 IFRnz instruction 33, 42, 73 IFRz instruction 33, 42, 74 equivalent (SKPRnz) 106 IFS instruction 33, 42, 75 IMM11 instruction field 31 Instruction Cache Enable. See IC. Instructions 5/16-bit immediate value 27 about 14 absolute-jump 32 conditional 33 equivalents 50 format 45 notation details 44 pseudo 50 relative-branch 31 user-defined 123, 124 Internal exceptions 35 Interrupt enable. See IE. Interrupt priority. See IPRI. IPRI 18 ISTATUS register 19, 39

### J

JMP %i7, equivalent 98 %o7, equivalent 80 instruction 16, 32, 42, 76

### K

K register 16, 27

### L

LD instruction 22, 29, 77, 78 LDP instruction 30, 78 LDS instruction 30, 79 Load 32-Bit Data From Memory 77 Pointer Addressing Mode 78 Stack Addressing Mode 79 Logical Not 92 Shift Left 81 Shift Left 81 Shift Right 83 Shift Right 83 Shift Right Immediate 84 LRET instruction 16, 42, 80 LSL instruction 81 LSLI instruction 28, 82 LSR instruction 83 LSRI instruction 28, 84

#### Μ

Memory access 21 reading from 22 writing to 23 Modes, addressing 27 MOV %g0, equivalent 91 instruction 85 Move Immediate 87 Immediate Into High Half-Word 86 MOVHI instruction 28, 86 MOVI instruction 28,87 MSTEP instruction 88 MUL instruction 89 Multiply 89 Multiply-Step 88

#### Ν

NEG instruction 90 Nios instruction set 14 register file 51 Nios CPU architecture 13 block diagram 41 instructions 43–126 pipeline 41 NOP instruction 91 NOT instruction 92 Notation details, instructions 44

### 0

Opcode table 47 Operators 50 OR instruction 28, 93

#### Ρ

Partial width register indirect 29 indirect with offset 31 Peripherals reading from 22 writing to 23 PFX instruction 27, 42, 94 PFXIO instruction 20, 27, 29, 42, 95 Pipeline implementation 41 Prefix 94 Prefix with Cache Bypass 95 Program counter 16 Program-flow control 31 Programmer's model 15 Pseudo-instructions 50

### R

RDCTL instruction 16, 96 Read Control Register 96 Register window overflow 36 underflow 35 usage 38 Register-indirect full width 29 full width with offset 30 partial width 29 partial width 29 partial width 29 partial width with offset 31 Registers %g0 16 %r0 16 about 14 Index

control 16 file example 51 general-purpose 14 groups 14 ISTATUS 19, 39 K 16, 27 STATUS 19, 33 Register-to-Register Move 85 Relative-branch instructions 31 Restore Caller's Register Window 97 RESTORE instruction 18, 34, 97 RET instruction 16, 42, 98 Return address 39 RLC instruction 99 Rotate Left Through Carry 99 **Right Through Carry 100** RRC instruction 100

### S

Save Caller's Register Window 101 SAVE instruction 18, 34, 101 SET\_IE (%ctl9) 21 SEXT16 instruction 102 SEXT8 instruction 103 Sign Extend 16-bit Value 102 8-bit Value 103 Skip If Register Bit Is 0 104 If Register Bit Is 1 105 If Register Equals 0 107 If Register Not Equal To 0 106 On Condition Code 108 SKP0 instruction 33, 42, 104 equivalent (IF1) 72 SKP1 instruction 33, 42, 71, 105 equivalent (IF0) 71 SKPRnz instruction 33, 42, 106 equivalent (IFRz) 74 SKPRz instruction 33, 42, 73, 107 equivalent (IFRnz) 73 SKPS instruction 33, 42, 108 ST instruction 23, 29, 109 ST16D instruction 23, 30, 110

ST16S instruction 30, 111 ST8D instruction 30, 112 ST8Dinstruction 23 ST8S instruction 30, 113 STATUS (%ctl0) 16 register 19, 33 Status preservation, ISTATUS register 39 Store 16-Bit Data To Memory Computed Half-Word Pointer Address 110 Stack Addressing Mode 116 Static Half-Word-Offset Address 111 Store 32-Bit Data To Memory 109 Pointer Addressing Mode 114 Stack Addressing Mode 115 Store 8-Bit Data To Memory Computed Byte-Pointer Address 112 Stack Addressing Mode 117 Static Byte-Offset Address 113 STP instruction 30, 114 STS instruction 30, 115 STS16S instruction 116 STS8S instruction 117 SUB instruction 118 SUBI instruction 28, 119 Subtract 118 Subtract Immediate 119 SWAP instruction 120 Swap Register Half-Words 120

### Т

TRAP instruction 32, 37, 42, 121 Trap Return 122 TRET instruction 16, 19, 32, 37, 42, 122

### U

Unconditional Trap 121 User-defined Instruction 123, 124 USR0 instruction 123 USRx instruction 124

#### W

WRCTL instruction 16, 125 WRCTL write 41 Write Control Register 125 WVALID (%ctl2) 19

Х

XOR instruction 28, 126

