Nios®

Nios Embedded Processor

32-Bit Programmer’s Reference Manual

ALTERA®

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

Document Version: 3.1
Document Date: January 2003

MNL-NIOS32PROG-3.1

About this Manual

This manual provides comprehensive information about the Altera® Nios® 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. Revision 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 *(1)* (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: **Save As** 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_{MAX}, \QuartusII** directory, **d:** drive, **chiptrip.gdf** file.
Bold italic type	Book titles are shown in bold italic type with initial capital letters. Example: ***1999 Device Data Book***.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75 (High-Speed Board Design).*
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: <*file name*>, <*project name*>**.pof** 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™ 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.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
↵	The angled arrow indicates you should press the Enter key.
👣	The feet direct you to more information on a particular topic.

Notes:

Contents

32-Bit Instruction Set

List of Tables

Notes:

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.

Table 1. Nios CPU Architecture

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

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

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

In registers	%r24-%r31	or	%i0-%i7
Local registers	%r16-%r23	or	%L0-%L7
Out registers	%r8-%r15	or	%o0-%o7
Global 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.

Table 3. Programmer's Model

<table>
<tr><td colspan="3"></td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="8">IN</td><td>%i7</td><td>%r31</td><td colspan="32">SAVED return-address</td></tr>
<tr><td>%i6</td><td>%r30</td><td colspan="32">%fp—frame pointer</td></tr>
<tr><td>%i5</td><td>%r29</td><td colspan="32"></td></tr>
<tr><td>%i4</td><td>%r28</td><td colspan="32"></td></tr>
<tr><td>%i3</td><td>%r27</td><td colspan="32"></td></tr>
<tr><td>%i2</td><td>%r26</td><td colspan="32"></td></tr>
<tr><td>%i1</td><td>%r25</td><td colspan="32"></td></tr>
<tr><td>%i0</td><td>%r24</td><td colspan="32"></td></tr>
<tr><td rowspan="8">LOCAL</td><td>%L7</td><td>%r23</td><td colspan="32"></td></tr>
<tr><td>%L6</td><td>%r22</td><td colspan="32"></td></tr>
<tr><td>%L5</td><td>%r21</td><td colspan="32"></td></tr>
<tr><td>%L4</td><td>%r20</td><td colspan="32"></td></tr>
<tr><td>%L3</td><td>%r19</td><td colspan="32">Base-pointer 3 for STP/LDP (or general-purpose local)</td></tr>
<tr><td>%L2</td><td>%r18</td><td colspan="32">Base-pointer 2 for STP/LDP (or general-purpose local)</td></tr>
<tr><td>%L1</td><td>%r17</td><td colspan="32">Base-pointer 1 for STP/LDP (or general-purpose local)</td></tr>
<tr><td>%L0</td><td>%r16</td><td colspan="32">Base-pointer 0 for STP/LDP (or general-purpose local)</td></tr>
<tr><td rowspan="8">OUT</td><td>%o7</td><td>%r15</td><td colspan="32">current return-address</td></tr>
<tr><td>%o6</td><td>%r14</td><td colspan="32">%sp-Stack Pointer</td></tr>
<tr><td>%o5</td><td>%r13</td><td colspan="32"></td></tr>
<tr><td>%o4</td><td>%r12</td><td colspan="32"></td></tr>
<tr><td>%o3</td><td>%r11</td><td colspan="32"></td></tr>
<tr><td>%o2</td><td>%r10</td><td colspan="32"></td></tr>
<tr><td>%o1</td><td>%r9</td><td colspan="32"></td></tr>
<tr><td>%o0</td><td>%r8</td><td colspan="32"></td></tr>
<tr><td rowspan="8">GLOBAL</td><td>%g7</td><td>%r7</td><td colspan="32"></td></tr>
<tr><td>%g6</td><td>%r6</td><td colspan="32"></td></tr>
<tr><td>%g5</td><td>%r5</td><td colspan="32"></td></tr>
<tr><td>%g4</td><td>%r4</td><td colspan="32"></td></tr>
<tr><td>%g3</td><td>%r3</td><td colspan="32"></td></tr>
<tr><td>%g2</td><td>%r2</td><td colspan="32"></td></tr>
<tr><td>%g1</td><td>%r1</td><td colspan="32"></td></tr>
<tr><td>%g0</td><td>%r0</td><td colspan="32"></td></tr>
<tr><td colspan="3"></td><td colspan="21">31</td><td colspan="10">10</td><td>0</td></tr>
<tr><td colspan="3"></td><td colspan="21"></td><td colspan="11">K REGISTER</td></tr>
<tr><td colspan="3"></td><td colspan="32">PC</td></tr>
<tr><td>%ctl9</td><td colspan="2">SET_IE</td><td colspan="32">Any write (WRCTL) operation to this register sets STATUS[15] (IE) = 1. Result of any read operation (RDCTL) is undefined.</td></tr>
<tr><td>%ctl8</td><td colspan="2">CLR_IE</td><td colspan="32">Any write (WRCTL) operation to this register clears STATUS[15] (IE) = 0. Result of any read operation (RDCTL) is undefined.</td></tr>
<tr><td>%ctl7</td><td colspan="2">DCACHE</td><td colspan="32">data cache (DCACHE) invalidate</td></tr>
<tr><td>%ctl6</td><td colspan="2">CPU_ID</td><td colspan="32">CPU ID</td></tr>
<tr><td>%ctl5</td><td colspan="2">ICACHE</td><td colspan="32">instruction cache (ICACHE) invalidate</td></tr>
<tr><td>%ctl4</td><td colspan="2">—</td><td colspan="32">—reserved —</td></tr>
<tr><td>%ctl3</td><td colspan="2">—</td><td colspan="32">—reserved —</td></tr>
<tr><td>%ctl2</td><td colspan="2">WVALID</td><td colspan="22"></td><td colspan="5">HI_LIMIT</td><td colspan="5">LO_LIMIT</td></tr>
<tr><td>%ctl1</td><td colspan="2">ISTATUS</td><td colspan="14"></td><td colspan="18">Saved Status</td></tr>
<tr><td>%ctl0</td><td colspan="2">STATUS</td><td colspan="14"></td><td>DC</td><td>IC</td><td>IE</td><td colspan="6">IPRI</td><td colspan="5">CWP</td><td>N</td><td>V</td><td>Z</td><td>C</td></tr>
<tr><td colspan="3"></td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

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 two-instruction 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 ← 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).

STATUS (%ctl0)

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

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 general-purpose 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 (%ctl2)" 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
N	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
C	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						HI_LIMIT					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® 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.

CLR_IE (%ctl8)

Any WRCTL operation to the CLR_IE register clears the IE bit in the STATUS register (IE ← 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 ← 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

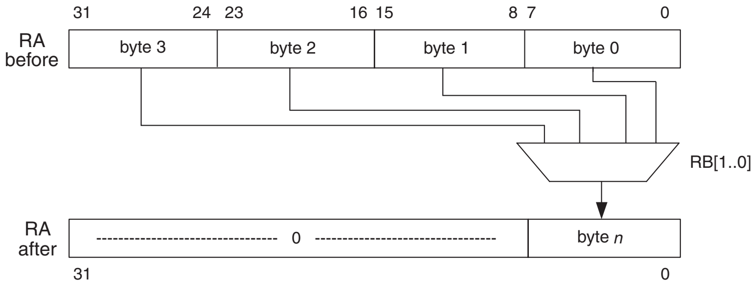
Reading from Memory (or Peripherals)

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.

Figure 1. EXT8D Instruction



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

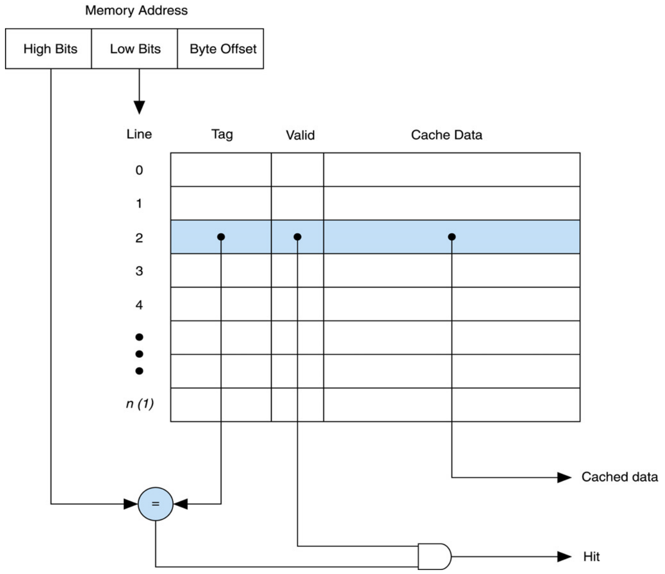
```
;Contents of memory before:
;
;                  0       1       2       3
;0x00001200     0x46    0x49    0x53    0x54

; Let's assume %o4 contains the address 0x00001203
; and that %g3 contains the value 0x000000BC
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       3
;0x00001200     0x46    0x49    0x53    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.

Figure 2. Accessing Lines of Cache with Low Bits of Word Address



Note:

(1) 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 %0 \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)
                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 Modes

The topics in this section includes a description of the addressing modes:

- 5/16-bit immediate
- Full width register-indirect
- Partial width register-indirect
- Full width register-indirect with offset
- Partial width register-indirect with offset

5/16-bit Immediate Value

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[(1)]	ANDN[(1)]	ASRI
CMPI	LSLI	LSRI	MOVI
MOVHI	OR[(1)]	SUBI	XOR[(1)]

Note:

(1) 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 & 0xFFFFFFFC).

Table 8. Instructions Using Register-Indirect Addressing

Instruction	Address Register	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	0..124 bytes
LDS	%sp	Any	0..1020 bytes
STP	%L0, %L1, %L2, %L3	Any	0..124 bytes
STS	%sp	Any	0..1020 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 half-word 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	0..1023 bytes
STS16S	%sp	%r0	Immediate	0..511 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 %o7, 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 automatically-generated bus logic converts multiple one-bit irq-sources into a single irq-input 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 exception-handling 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 %o7.

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 %o0..%o5 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 non-maskable 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:

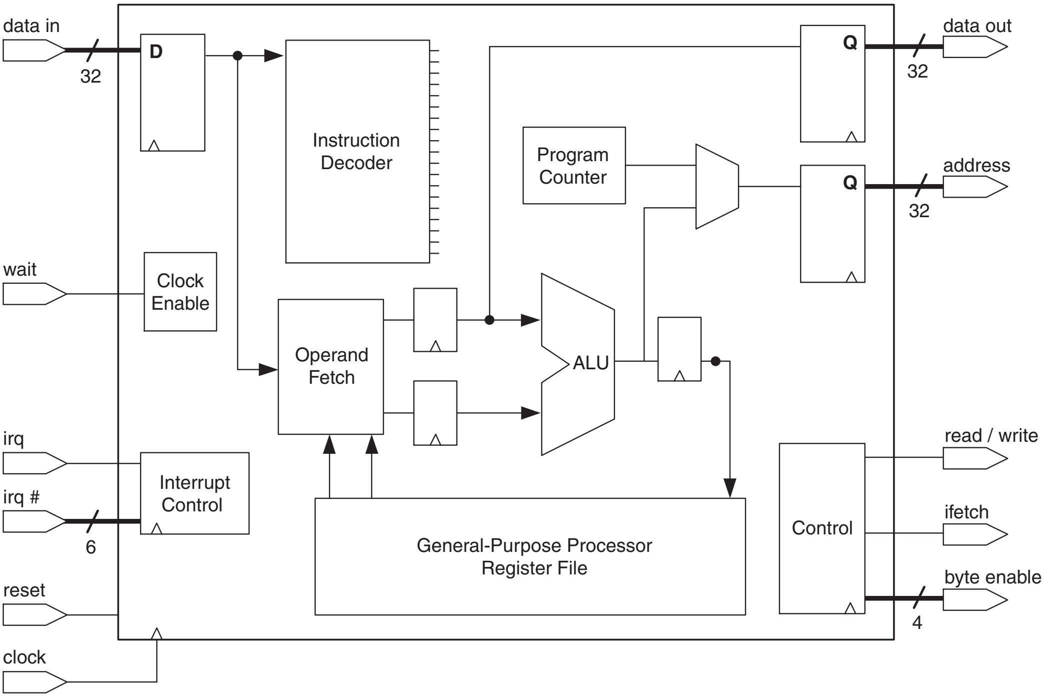
- It must preserve the contents of ISTATUS before re-enabling interrupts. For example, ISTATUS could be saved on the stack.
- It must check CWP before re-enabling interrupts to be sure CWP is at or above LO_LIMIT. If CWP is below LO_LIMIT, it must take an action to open up more available register windows (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.

Figure 3. Nios CPU Block Diagram



Direct CWP Manipulation

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

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.

Table 12. BR Branch Delay Slot Example

```
           …
(a)           ADD %g2, %g3
(b)           BR Target
(c)           ADD %g4, %g5    ← Branch Delay Slot
(d)           ADD %g6, %g7
           …
           Target:
(e)           ADD %g8, %g9
```

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
- JMP
- LRET
- PFX
- PFXIO
- RET
- SKP1
- SKP0
- SKPRNZ
- SKPRZ
- SKPS
- TRET
- TRAP

32-Bit Instruction Set

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 Δ 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)

Table 13. Notation Details

Notation	Meaning	Notation	Meaning
X ← Y	X is written with Y	X >> n	The value X after being right-shifted n bit positions
∅ ← 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	^{bn}X	The nth byte (8-bit field) within the full-width value X. ^{b0}X = X[7..0], ^{b1}X = X[15..8], ^{b2}X = X[23..16], and ^{b3}X = X[31..24]
RB	One of the 32 visible registers, selected by the 5-bit b-field of the instruction word	^{hn}X	The nth half-word (16-bit field) within the full-width value X. ^{h0}X = X[15..0], ^{h1}X = X[31..16]
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	$X \mid Y$	Bitwise logical OR
K	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	$\lvert X \rvert$	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 nth bit of X (n = 0 means LSB)	VECBASE	Byte address of Vector #0 in the interrupt vector table (VECBASE is configurable)
X[n..m]	Consecutive bits n through m of X		
C	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.		

Table 14. Instruction Format (Part 1 of 2)

<table>
<tr><td>RR</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="6">op6</td><td colspan="5">B</td><td colspan="5">A</td></tr>
<tr><td>Ri5</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="6">op6</td><td colspan="5">IMM5</td><td colspan="5">A</td></tr>
<tr><td>Ri4</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="6">op6</td><td>0</td><td colspan="4">IMM4</td><td colspan="5">A</td></tr>
<tr><td>RPi5</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="4">op4</td><td colspan="2">P</td><td colspan="5">B</td><td colspan="5">A</td></tr>
<tr><td>Ri6</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="5">op5</td><td colspan="6">IMM6</td><td colspan="5">A</td></tr>
<tr><td>Ri8</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="3">op3</td><td colspan="8">IMM8</td><td colspan="5">A</td></tr>
<tr><td>i9</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="6">op6</td><td colspan="9">IMM9</td><td>0</td></tr>
<tr><td>i10</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="6">op6</td><td colspan="10">IMM10</td></tr>
<tr><td>i11</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="5">op5</td><td colspan="11">IMM11</td></tr>
<tr><td>Ri1u</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="6">op6</td><td colspan="3">op3u</td><td>IMM1u</td><td>0</td><td colspan="5">A</td></tr>
<tr><td>Ri2u</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td colspan="6">op6</td><td colspan="3">op3u</td><td colspan="2">IMM2u</td><td colspan="5">A</td></tr>
</table>

Table 14. Instruction Format (Part 2 of 2)

i8v	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	op6						op2v		IMM8v							

i6v	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	op6						op2v		0	0	IMM6v					

Rw	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	op6						op5w					A				

i4w	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	op6						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	0

Table 15. 32-bit Opcode Table (Part 1 of 3)

Opcode	Mnemonic	Format	Summary
000000	ADD	RR	RA ← RA + RB Flags affected: N, V, C, Z
000001	ADDI	Ri5	RA ← RA + (0×00.00 : K : IMM5) Flags affected: N, V, C, Z
000010	SUB	RR	RA ← RA – RB Flags affected: N, V, C, Z
000011	SUBI	Ri5	RA ← RA – (0×00.00 : K : IMM5) Flags affected: N, V, C, Z
000100	CMP	RR	∅ ← RA – RB Flags affected: N, V, C, Z
000101	CMPI	Ri5	∅ ← RA – (0×00.00 : K : IMM5) Flags affected: N, V, C, Z
000110	LSL	RR	RA ← (RA << RB [4..0]), Zero-fill from right
000111	LSLI	Ri5	RA ← (RA << IMM5), Zero-fill from right
001000	LSR	RR	RA ← (RA >> RB [4..0]), Zero-fill from left
001001	LSRI	Ri5	RA ← (R >> IMM5), Zero-fill form left
001010	ASR	RR	RA ← (RA >> RB [4..0]), Fill from left with RA[31]
001011	ASRI	Ri5	RA ← (RA >> IMM5), Fill from left with RA[31]
001100	MOV	RR	RA ← RB
001101	MOVI	Ri5	RA ← (0×00.00 : K : IMM5)
001110	AND	RR Ri5	RA ← RA & {RB, (0×00.00 : K : IMM5)} Flags affected: N, Z
001111	ANDN	RR, Ri5	RA ← RA & ~({RB, (0×00.00 : K : IMM5)}) Flags affected: N, Z
010000	OR	RR, Ri5	RA ← RA | {RB, (0×00.00 : K : IMM5)} Flags affected: N, Z
010001	XOR	RR, Ri5	RA ← RA ⊕ {RB, (0×00.00 : K : IMM5)} Flags affected: N, Z
010010	BGEN	Ri5	RA ← 2^{IMM5}
010011	EXT8D	RR	RA ← (0×00.00.00 : ^{bn}RA) where *n* = RB[1..0]
010100	SKP0	Ri5	Skip next instruction if: (RA [IMM5] == 0)
010101	SKP1	Ri5	Skip next instruction if: (RA [IMM5] == 1)
010110	LD	RR	RA ← Mem32 [align32(RB + (σ(K) × 4))]

Table 15. 32-bit Opcode Table (Part 2 of 3)

Opcode	Mnemonic	Format	Summary
010111	ST	RR	Mem32 [align32(RB + (σ(K) × 4))] ← RA
011000	STS8S	i10	^{bn}Mem32 [align32(%sp + IMM10)] ← ^{bn}%r0 where n = IMM10[1..0]
011001	STS16S	i9	^{hn}Mem32 [align32(%sp + IMM9 × 2)] ← ^{hn}%r0 where n = IMM9[0]
011010	EXT16D	RR	RA ← (0×00.00 : ^{hn}RA) where n = RB[1]
011011	MOVHI	Ri5	^{h1}RA ← (K : IMM5), ^{h0}RA unaffected
011100	USR0	RR	RA ← RA <user-defined operation> RB
011101000	EXT8S	Ri1u	RA ← (0×00.00.00 : ^{bn}RA) where n = IMM2u
011101001	EXT16S	Ri1u	RA ← (0×00.00 : ^{hn}RA) where n = IMM1u
011101010			Unused
011101011			Unused
011101100	ST8S	Ri1u	^{bn}Mem32 [align32(RA + (σ(K) × 4))] ← ^{bn}%r0 where n = IMM2u
011101101	ST16S	Ri1u	^{hn}Mem32 [align32(RA + (σ(K) × 4))] ← ^{hn}%r0 where n = IMM1u
01111000	SAVE	i8v	CWP ← CWP – 1; %sp ← %fp – (IMM8v × 4) If (old-CWP == LO_LIMIT) then TRAP #1

0111100100	TRAP	i6v	ISTATUS ← STATUS; IE ← 0; CWP ← CWP – 1;
			IPRI ← IMM6v; %r15 ← ((PC + 2) >> 1) ;
			PC ← Mem32 [VECBASE + (IMM6v × 4)] × 2

01111100000	NOT	Rw	RA ← ~RA
01111100001	NEG	Rw	RA ← 0 – RA
01111100010	ABS	Rw	RA ← |RA|
01111100011	SEXT8	Rw	RA ← σ(^{b0}RA)
01111100100	SEXT16	Rw	RA ← σ(^{h0}RA)
01111100101	RLC	Rw	C ← msb (RA); RA ← (RA << 1) : C Flag affected: C
01111100110	RRC	Rw	C ← RA[0]; RA ← C : (RA >> 1) Flag affected: C
01111100111			Unused
01111101000	SWAP	Rw	RA ← ^{h0}RA : ^{h1}RA
01111101001	USR1	Rw	RA ← RA <user-defined operation> R0
01111101010	USR2	Rw	RA ← RA <user-defined operation> R0
01111101011	USR3	Rw	RA ← RA <user-defined operation> R0
01111101100	USR4	Rw	RA ← RA <user-defined operation> R0
01111101101	RESTORE	w	CWP ← CWP + 1; if (old-CWP == HI_LIMIT) then TRAP #2
01111101110	TRET	Rw	PC ← (RA × 2); STATUS ← ISTATUS

Table 15. 32-bit Opcode Table (Part 3 of 3)

Opcode	Mnemonic	Format	Summary
01111101111			Unused
01111110000	ST8D	Rw	^{bn}Mem32 [align32(RA +(σ(K) × 4))] ← ^{bn}%r0 where n = RA[1..0]
01111110001	ST16D	Rw	^{hn}Mem32 [align32(RA + (σ(K) × 4))] ← ^{hn}%r0 where n = RA[1]
01111110010	FILL8	Rw	%r0 ← (^{b0}RA : ^{b0}RA : ^{b0}RA : b^{0}RA)
01111110011	FILL16	Rw	%r0 ← (^{h0}RA : ^{h0}RA)
01111110100	MSTEP	Rw	if (%r0[31] == 1) then %r0 ← (%r0 << 1) + RA else %r0 ← (%r0 << 1)
01111110101	MUL	Rw	%r0 ← (%r0 & 0x0000.ffff) × (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 ← RA
01111111001	RDCTL	Rw	RA ← CTLk
01111111010	SKPRNZ	Rw	Skip next instruction if: (RA != 0)
01111111011			Unused
01111111100			Unused
01111111101			Unused
01111111110	JMP	Rw	PC ← (RA × 2)
01111111111	CALL	Rw	R15 ←((PC + 4) >> 1); PC ← (RA × 2)
100000	BR	i11	PC ← PC + ((σ(IMM11) + 1) × 2)
100001			Unused
100010	BSR	i11	PC ← PC + ((σ(IMM11) + 1) × 2); %r15 ← ((PC + 4) >> 1)
10010	PFXIO	i11	K ← 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 + (σ(K : IMM5) × 4))] ← RA
1011	LDP	RPi5	RA ← Mem32 [align32(RP + (σ(K : IMM5) × 4))]
110	STS	Ri8	Mem32[align32(%sp + (IMM8 × 4))] ← RA
111	LDS	Ri8	RA ← Mem32 [align32(%sp + (IMM8 × 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 %o7	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 Operators

Operator	Description	Operation
%lo(*x*)	Extract low 5 bits of *x*	*x* & 0×0000001f
%hi(*x*)	Extract bits 15..5 of *x*	(*x* >> 5) & 0×000007ff
%xlo(*x*)	Extract bits 20..16 of *x*	(*x* >> 16) & 0×0000001f
%xhi(*x*)	Extract bits 31..21 of *x*	(*x* >> 21) & 0×000007ff
x@h	Half-word address of *x*	*x* >> 1

Table 18. Smallest Nios Register File

← Restore

Save →

(Internal Register File)	CWP=6 (HI_LIMIT)	CWP=5	CWP=4	CWP=3	CWP=2	CWP=1 (LOW_LIMIT)	CWP=0 (Underflow or TRAP)
Reg[120..127]	%i0..%i7						
Reg[112..119]	%L0..%L7						
Reg[104..111]	%o0..%o7	%i0..%i7					
Reg[96..103]		%L0..%L7					
Reg[88..95]		%o0..%o7	%i0..%i7				
Reg[80..87]			%L0..%L7				
Reg[72..79]			%o0..%o7	%i0..%i7			
Reg[64..71]				%L0..%L7			
Reg[56..63]				%o0..%o7	%i0..%i7		
Reg[48..55]					%L0..%L7		
Reg[40..47]					%o0..%o7	%i0..%i7	
Reg[32..39]						%L0..%L7	
Reg[24..31]						%o0..%o7	%i0..%i7
Reg[16..23]							%L0..%L7
Reg[8..15]							%o0..%o7
Reg[0..7]	%g0..%g7	%g0..%g7	%g0..%g7	%g0..%g7	%g0..%g7	%g0..%g7	%g0..%g7

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

Operation: RA ← |RA|

Assembler Syntax: `ABS %rA`

Example: `ABS %r6`

Description: Calculate the absolute value of RA; store the result in 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	1	0	A				

ADD

Operation: RA ← RA + RB

Assembler Syntax: `ADD %rA,%rB`

Example: `ADD %L3,%g0      ; ADD %g0 to %L3`

Description: Adds the contents of register A to register B and stores the result in register A.

Condition Codes: Flags:

N	V	Z	C
Δ	Δ	Δ	Δ

N: Result bit 31
V: Signed-arithmetic overflow
Z: Set if result is zero; cleared otherwise
C: Carry-out of addition

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	0	0	0	B					A				

ADDI

Add Immediate

Operation: RA ← RA + (0x00.00 : K : IMM5)

Assembler Syntax:

```
ADDI %rA,IMM5
```

Example:

Not preceded by PFX:

```
ADDI %L5,6           ; add 6 to %L5
```

Preceded by PFX:

```
PFX %hi(1000)
ADDI %g3,%lo(1000)   ; ADD 1000 to %g3
```

Description:

Not preceded by PFX:

Adds 5-bit immediate value to register A, stores result in register A. IMM5 is in the range [0..31].

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 zero-extended to 32 bits and added to register A.

Condition Codes:

Flags:

N	V	Z	C
Δ	Δ	Δ	Δ

N: Result bit 31

V: Signed-arithmetic overflow

Z: Set if result is zero; cleared otherwise

C: Carry-out of addition

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	0	0	0	0	1	IMM5					A				

AND

Bitwise Logical AND

Operation:

Not preceded by PFX:

RA ← RA & RB

Preceded by PFX:

RA ← RA & (0x00.00 : K : IMM5)

Assembler Syntax:

Not preceded by PFX:

```
AND %rA,%rB
```

Preceded by PFX:

```
PFX %hi(const)
AND %rA,%lo(const)
```

Example:

Not preceded by PFX:

```
AND %g0,%g1             ; %g0 gets %g1 & %g0
```

Preceded by PFX:

```
PFX %hi(16383)
AND %g0,%lo(16383)      ; AND %g0 with 16383
```

Description:

Not preceded by PFX:

Logically-AND the individual bits in RA with the corresponding bits in RB; store the result in RA.

Preceded by PFX:

The RB operand is replaced by an immediate constant formed by concatenating the contents of the K-register (11 bits) with IMM5 (5 bits). This 16-bit value (zero-extended to 32 bits) is bitwise-ANDed with RA, and the result is written back into RA.

Condition Codes:

Flags:

N	V	Z	C
Δ	–	Δ	–

N: Result bit 31

Z: Set if result is zero, cleared otherwise

Instruction Format: RR, Ri5

Instruction Fields:

A = Register index of RA operand

B = Register index of RB operand

IMM5 = 5-bit immediate value

Not preceded by PFX (RR)

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

Preceded by PFX (Ri5)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	IMM5					A				

ANDN

Bitwise Logical AND NOT

Operation:

Not preceded by PFX:

RA ← RA & ~RB

Preceded by PFX:

RA ← RA & ~(0x00.00 : K : IMM5)

Assembler Syntax:

Not preceded by PFX:

```
ANDN %rA,%rB
```

Preceded by PFX:

```
PFX %hi(const)
ANDN %rA,%lo(const)
```

Example:

Not preceded by PFX:

```
ANDN %g0,%g1              ; %g0 gets %g0 & ~%g1
```

Preceded by PFX:

```
PFX %hi(16384)
ANDN %g0,%lo(16384)       ; clear bit 14 of %g0
```

Description:

Not preceded by PFX:

Logically-AND the individual bits in RA with the corresponding bits in the one's-complement of RB; store the result in RA.

Preceded by PFX:

The RB operand is replaced by an immediate constant formed by concatenating the contents of the K-register (11 bits) with IMM5 (5 bits). This 16-bit value is zero-extended to 32 bits, then bitwise-inverted and bitwise-ANDed with RA. The result is written back into RA.

Condition Codes:

Flags:

N	V	Z	C
Δ	–	Δ	–

N: Result bit 31

Z: Set if result is zero, cleared otherwise

Instruction Format: RR, Ri5

Instruction Fields:

A = Register index of operand RA

B = Register index of operand RB

IMM5 = 5-bit immediate value

Not preceded by PFX (RR)

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

Preceded 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					A				

ASR

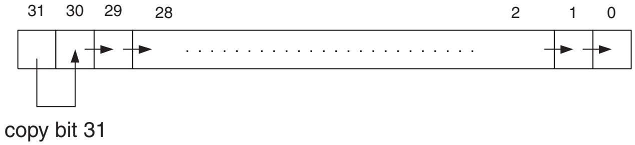
Arithmetic Shift Right

Operation: RA ← (RA >> RB[4..0]), fill from left with RA[31]

Assembler Syntax: `ASR %rA,%rB`

Example: `ASR %L3,%g0      ; shift %L3 right by %g0 bits`

Description: Arithmetically shift right the value in RA by the value of RB; store the result in RA. Bits 31..5 of RB are ignored. If the value in RB[4..0] is 31, RA is zero or negative one depending on the original sign of RA.



Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

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	1	0	1	0			B					A		

ASRI

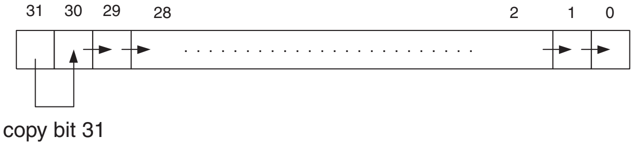
Arithmetic Shift Right Immediate

Operation: RA ← (RA >> 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.



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	0	1	0	1	1	IMM5					A				

BGEN

Bit Generate

Operation: RA ← 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

Operation: $PC \leftarrow PC + ((\sigma(IMM11) + 1) << 1)$

Assembler Syntax:

```
BR addr
```

Example:

```
BR MainLoop
NOP      ; (delay slot)
```

Description: The offset given by IMM11 is interpreted as a signed number of half-words (instructions) relative to the instruction immediately following BR. Program control is transferred to instruction at this offset.

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Delay Slot Behavior: The instruction immediately following BR (BR's delay slot) is executed after BR, 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: i11

Instruction Fields: IMM11 = 11-bit immediate value

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	IMM11										

BSR

Branch To Subroutine

Operation:

$\%o7 \leftarrow ((PC + 4) >> 1)$
$PC \leftarrow PC + ((\sigma(IMM11) + 1) << 1)$

Assembler Syntax:

```
BSR addr
```

Example:

```
BSR SendCharacter
NOP      ; (delay slot)
```

Description:

The offset given by IMM11 is interpreted as a signed number of half-words (instructions) relative to the instruction immediately following BR. Program control is transferred to instruction at this offset. The return-address is the address of the BSR instruction plus four, which is the address of the second subsequent instruction. The return-address is shifted right one bit and stored in %o7. The right-shifted value stored in %o7 is a destination suitable for direct use by JMP without modification.

Condition Codes:

Flags: Unaffected

N	V	Z	C
–	–	–	–

Delay Slot Behavior:

The instruction immediately following BSR (BSR’s delay slot) is executed after BSR, 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:

i11

Instruction Fields:

IMM11 = 11-bit immediate 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

Operation:

%o7 ← ((PC + 4) >> 1)
PC ← (RA << 1)

Assembler Syntax:

```
CALL %rA
```

Example:

```
CALL %g0
NOP     ; (delay slot)
```

Description:

The value of RA is shifted left by one and transferred into PC. RA contains the address of the called subroutine right-shifted by one bit. The return-address is the address of the second subsequent instruction. Return-address is shifted right one bit and stored in %o7. The right-shifted value stored in %o7 is a destination suitable for direct use by JMP without modification.

Condition Codes:

Flags: Unaffected

N	V	Z	C
–	–	–	–

Delay Slot Behavior:

The instruction immediately following CALL (CALL's delay slot) is executed after CALL, 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	1	A				

CMP

Compare

Operation: ∅ ← 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:

N	V	Z	C
Δ	Δ	Δ	Δ

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

Operation: $\varnothing \leftarrow RA - (0x00.00 : K : IMM5)$

Assembler Syntax: `CMPI & %rA,IMM5`

Example:

Not preceded by PFX:

```
CMPI %i3,24             ; compare %i3 to 24
```

Preceded by PFX:

```
PFX %hi(1000)
CMPI %i4,%lo(1000)      ; compare %i4 to 1000
```

Description:

Not preceded 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 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 zero-extended to 32 bits and subtracted from RA. Condition codes are set and the result is discarded. RA is not altered.

Condition Codes:

Flags:

N	V	Z	C
Δ	Δ	Δ	Δ

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: Ri5

Instruction Fields:

A = Register index of RA operand

IMM5 = 5-bit immediate value

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td colspan="5">IMM5</td><td colspan="5">A</td></tr>
</table>

EXT16D

Half-Word Extract (Dynamic)

Operation: $RA \leftarrow (0x00.00 : {}^{hn}RA)$ where $n = RB[1]$

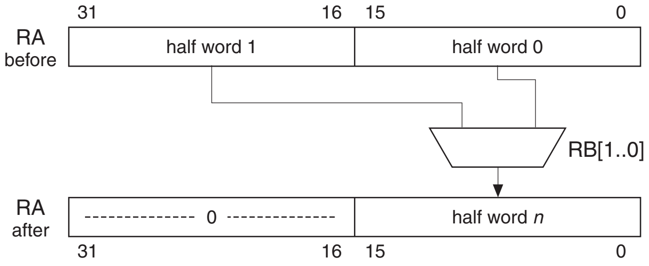
Assembler Syntax:

```
EXT16D %rA,%rB
```

Example:

```
LD %i3,[%i4]      ; get 32 bits from [%i4 & 0xFF.FF.FF.FC]
EXT16D %i3,%i4    ; extract short int at %i4
```

Description: 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 15..0 of RA, and the more-significant bits 31..16 are set to zero.



Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: RR

Instruction Fields:

A = Register index of operand RA
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			B					A		

EXT16S

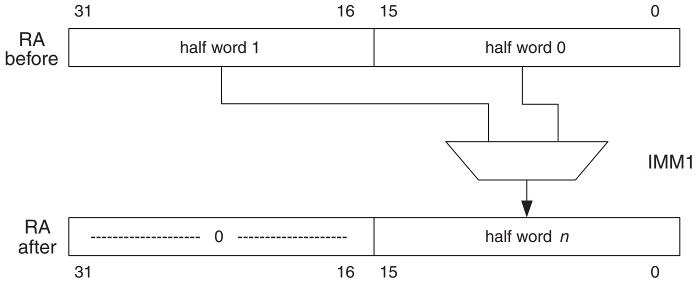
Half-Word Extract (Static)

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

Assembler Syntax: `EXT16S %rA,IMM1`

Example: `EXT16S %L3,1       ; %L3 gets upper short int of itself`

Description: 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 15..0 of RA, and the more significant bits 31..16 are set to zero.



Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: Ri1u

Instruction Fields: A = Register index of operand RA
IMM1 = 1-bit immediate value

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	0	1	IMM1	0	A				

EXT8D

Byte-Extract (Dynamic)

Operation: RA ← (0x00.00.00 : ^{bn}RA) where n = RB[1..0]

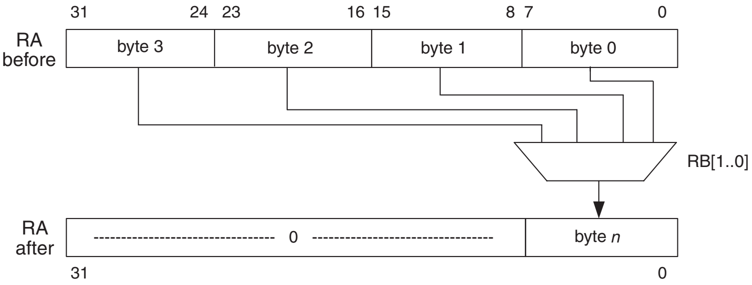
Assembler Syntax:

```
EXT8D %rA,%rB
```

Example:

```
LD %g4,[%i0]        ; get 32 bits from [%i0 & 0xFF.FF.FF.FC]
EXT8D %g4,%i0       ; extract the particular byte at %i0
```

Description: Extracts one of the four bytes in RA. The byte to-be-extracted is chosen by bits 1..0 of RB (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.



Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: RR

Instruction Fields:

A = Register index of operand RA
B = Register index of operand RB

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

EXT8S

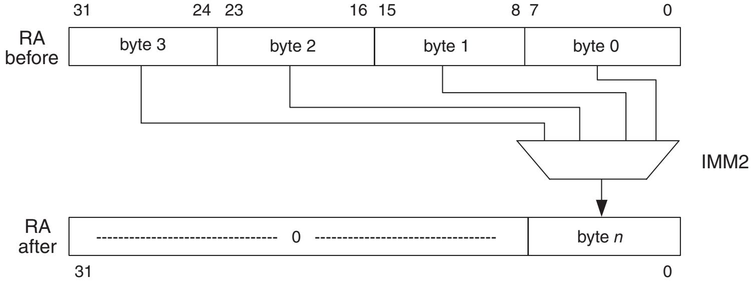
Byte-Extract (Static)

Operation: RA ← (0x00.00.00 : ^{bn}RA) where n = IMM2

Assembler Syntax: `EXT8S %rA,IMM2`

Example: `EXT8S %g6,3      ; %g6 gets the 3rd byte of itself`

Description: 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.



Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: Ri2u

Instruction Fields: A = Register index of operand RA
IMM2 = 2-bit immediate value

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	0	0	IMM2		A				

FILL16

Half-Word Fill

Operation: R0 ← (^{h0}RA : ^{h0}RA)

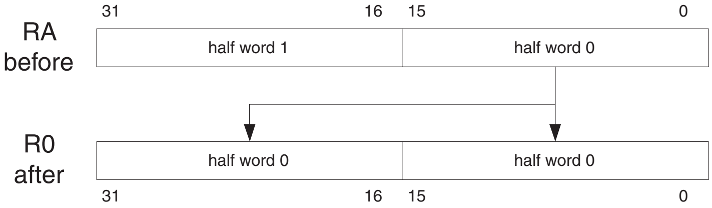
Assembler Syntax:

```
FILL16 %r0,%rA
```

Example:

```
FILL16 %r0,%i3      ; %r0 gets 2 copies of %i3[0..15]
                    ; first operand must be %r0
```

Description: The least significant half-word of RA is copied into both half-word positions in %r0. %r0 is the only allowed destination operand for FILL instructions.



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	1	0	0	1	1	A				

FILL8

Byte-Fill

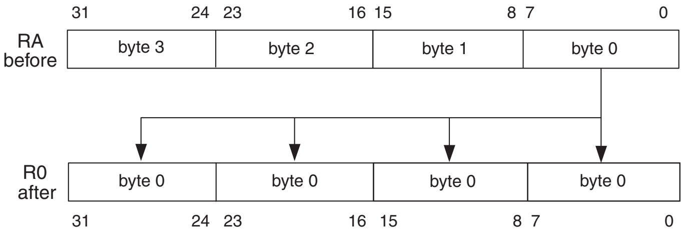
Operation: $R0 \leftarrow ({}^{b0}RA : {}^{b0}RA : {}^{b0}RA : {}^{b0}RA)$

Assembler Syntax: `FILL8 %r0,%rA`

Example:

```
FILL8 %r0,%o3      ; %r0 gets 4 copies of %o3[0..7]
                   ; 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.



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	1	0	0	1	0	A				

IF0

Execute if Register Bit is 0

(Equivalent to SKP1 Instruction)

Operation:

```
if (RA[IMM5] == 1)
then begin
        if (Mem16[PC + 2] is PFX or PFXIO)
        then PC ← PC + 6
        else PC ← PC + 4
end
```

Assembler Syntax:

```
IF0 %rA,IMM5
```

Example:

```
IF0 %o3,21      ;
ADDI %g0,1      ; increment if bit 21 is clear
```

Description: 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.

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	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	IMM5					A				

IF1

Execute if Register Bit is 1

(Equivalent to SKP0 Instruction)

Operation:

```
if (RA[IMM5] == 0)
then begin
        if (Mem16[PC + 2] is PFX or PFXIO)
        then PC ← PC + 6
        else PC ← PC + 4
end
```

Assembler Syntax:

```
IF1 %rA,IMM5
```

Example:

```
IF1 %i3, 7
ADDI %g0,1      ; increment if bit 7 is set
```

Description: Skip next instruction if the single bit RA[IMM5] is 0. 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: 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	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	IMM5					A				

IFRNZ

Execute if Register is not Zero

(Equivalent to SKPRZ Instruction)

Operation:

```
if (RA == 0)
then begin
    if (Mem16[PC + 2] is PFX or PFXIO)
    then PC ← PC + 6
    else PC ← PC + 4
end
```

Assembler Syntax:

```
IFRNZ %rA
```

Example:

```
IFRNZ %o3
BSR SendIt    ; only branch if %o3 is not zero
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.

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	1	0	1	1	0			A		

IFRZ

Execute if Register is Zero

(Equivalent to SKPRNZ Instruction)

Operation:

```
if (RA != 0)
then begin
        if (Mem16[PC + 2] is PFX or PFXIO)
        then PC ← PC + 6
        else PC ← PC + 4
end
```

Assembler Syntax:

```
IFRZ%rA
```

Example:

```
IFRZ %g3
BSR SendIt    ; only call if %g3 is zero
NOP           ; (delay slot) executed in either case
```

Description: Skip next instruction if RA is not zero. 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: 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	1	1	0	1	0	A				

IFS

Conditionally Execute Next Instruction

Operation:

```
if (condition IMM4 is false)
then begin
       if (Mem16[PC + 2] is PFX or PFXIO)
       then PC ← PC + 6
       else PC ← PC + 4
end
```

Assembler Syntax:

```
IFS cc_IMM4
```

Example:

```
IFS cc_ne
BSR SendIt     ; only call if Z flag set
NOP            ; (delay slot) executed in either case
```

Description: Execute next instruction if specified condition is true, skip if condition is false. 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:

These condition codes have different numeric values for IFS and SKPS instructions.

cc_nc	0x0	(not C)
cc_c	0x1	(C)
cc_nz	0x2	(not Z)
cc_z	0x3	(Z)
cc_pl	0x4	(not N)
cc_mi	0x5	(N)
cc_lt	0x6	(N xor V)
cc_ge	0x7	(not (N xor V))
cc_gt	0x8	(Not (Z or (N xor V)))
cc_le	0x9	(Z or (N xorV))
cc_nv	0xa	(not V)
cc_v	0xb	(V)
cc_hi	0xc	(not (C or Z))
cc_la	0xd	(C or Z)

Additional alias flags allowed:

cc_cs = cc_c	cc_n = cc_mi	cc_cc = cc_nc	cc_vc = cc_nv
cc_eq = cc_z	cc_vs = cc_v	cc_ne = cc_nz	cc_p = cc_pl

Codes mean execute if. For example, ifs cc_eq means execute if equal

Instruction Format: i4w

Instruction Fields: IMM4 = 4-bit immediate 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			

JMP

Computed Jump

Operation: PC ← (RA << 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				

LD

Load 32-Bit Data From Memory

Operation:

Not preceded by PFX:

$\mathrm{RA} \leftarrow \mathrm{Mem32[align32(RB)]}$

Preceded by PFX or PFXIO:

$\mathrm{RA} \leftarrow \mathrm{Mem32[align32(RB} + \sigma(\mathrm{K}) \times 4))]$

Assembler Syntax:

```
LD %rA,[%rB]
```

Example:

Not preceded by PFX:

```
LD %g0,[%i3]        ; load word at [%i3] into %g0
```

Preceded by PFX:

```
PFX 7               ; offset by 7 words
LD %g0,[%i3]        ; load word at [%i3+28] into %g0
```

Preceded by PFXIO:

```
PFXIO 0             ; forces LD to bypass the data cache
LD %g0,[%i3]        ; load word at [%i3] into %g0
```

Description:

☞ If the Nios CPU has a data cache, the 32-bit data value may come from the cache.

Not preceded by PFX:

Loads a 32-bit data value from memory into RA. Data is always read from a word-aligned address given by bits 31..2 of RB (the two LSBs of RB are ignored).

Preceded by PFX:

The value in K is sign-extended and used as a word-scaled, signed offset. This offset is added to the base-address RB (bits 1..0 ignored), and data is read from the resulting word-aligned address.

Preceded by PFXIO:

Preceding LD by PFXIO is just like preceding LD by PFX (see above), except that, in a Nios CPU with a data cache, the LD bypasses the cache.

Condition Codes:

Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format:

RR

Instruction Fields:

A = Register index of operand RA

B = Register index of operand RB

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

LDP

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

Operation:

Not preceded by PFX:

$RA \leftarrow \text{Mem32}[\text{align32}(RP + (IMM5 \times 4))]$

Preceded by PFX or PFXIO:

$RA \leftarrow \text{Mem32}[\text{align32}(RP + (\sigma(K : IMM5) \times 4))]$

Assembler Syntax:

```
LDP %rA,[%rP,IMM5]
```

Example:

Not preceded by PFX:

```
LDP %o3,[%L2,3]     ; Load %o3 from [%L2 + 12]
                    ; second register operand must be
                    ; one of %L0, %L1, %L2, or %L3
```

Preceded by PFX:

```
PFX %hi(100)
LDP %o3,[%L2,%lo(100)]    ; load %o3 from [%L2 + 400]
```

Preceded by PFXIO:

```
PFXIO %hi(100)
LDP %o3,[%L2,%lo(100)]    ; load %o3 from [%L2 + 400]
```

Description:

If the Nios CPU has a data cache, the 32-bit data value may come from the cache.

Not preceded by PFX:

Loads a 32-bit data value from memory into RA. Data is always read from a word-aligned address given by bits 31..2 of RP (the two LSBs of RP are ignored) plus a 5-bit, unsigned, word-scaled offset given by IMM5. If Nios has a data cache, the 32-bit data value may come from the cache. This instruction is similar to LD, 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 be one of the four registers: %L0, %L1, %L2, or %L3.

Preceded by PFX:

A 16-bit offset is formed by concatenating the 11-bit K-register with IMM5 (5 bits). The 16-bit offset (K : IMM5) is sign-extended to 32 bits, multiplied by four, and added to bits 31..2 of RP to yield a word-aligned effective address.

Preceded by PFXIO:

Preceding LDP by PFXIO is just like preceding LDP by PFX (see above), except that in a Nios CPU with a data cache, the LDP bypasses the cache.

Condition Codes:

Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format:

RPi5

Instruction Fields:

A = Register index of operand 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	1	P		IMM5					A				

LDS

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

Operation: RA ← Mem32[align32(%sp + (IMM8 × 4))]

Assembler Syntax:

```
LDS %rA,[%sp,IMM8]
```

Example:

```
LDS %o1,[%sp,3]     ; load %o1 from stack + 12
                    ; second register can only be %sp
```

Description:

Loads a 32-bit data value from memory into RA. Data is always read from a word-aligned address given by bits 31..2 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. LDS allows single-instruction access to any data word at a known offset in a 1Kbyte range above %sp.

☞ If the Nios CPU has a data cache, the 32-bit data value may come from the cache**.**

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: Ri8

Instruction Fields:

A = Register index of 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	1	IMM8								A				

LRET

Equivalent to JMP %o7

Operation: PC ← (%o7 << 1)

Assembler Syntax: `LRET`

Example:

```
LRET      ; return
NOP       ; (delay slot)
```

Description: Jump to the target-address given by (%o7 << 1). Note that the target address is always half-word aligned for any value of %o7.

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Delay Slot Behavior: The instruction immediately following LRET (LRET's delay slot) is executed after LRET, 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 %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

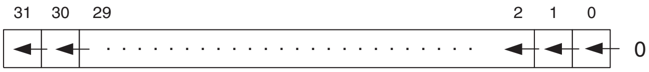
Operation: RA ← (RA << RB[4..0]), zero-fill from right

Assembler Syntax: `LSL %rA,%rB`

Example:

```
LSL %L3,%g0      ; Shift %L3 left by %g0 bits
```

Description: The value in RA is shifted-left by the number of bits indicated by RB [4..0] (bits 31..5 of RB are ignored).



Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

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	1	0	B					A				

LSLI

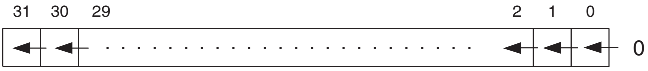
Logical Shift Left Immediate

Operation: RA ← (RA << IMM5), zero-fill from right

Assembler Syntax: `LSLI %rA,IMM5`

Example: `LSLI %i1,6      ; Shift %i1 left by 6 bits`

Description: The value in RA is shifted-left by the number of bits indicated by IMM5.



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	0	0	1	1	1	IMM5					A				

LSR

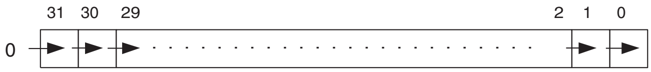
Logical Shift Right

Operation: RA ← (RA >> RB[4..0]), zero-fill from left

Assembler Syntax: `LSR %rA,%rB`

Example: `LSR %L3,%g0      ; Shift %L3 right by %g0 bits`

Description: The value in RA is shifted-right by the number of bits indicated by RB [4..0] (bits RB[31..5] are ignored). The result is zero-filled from the left.



Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

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	1	0	0	0	B					A				

LSRI

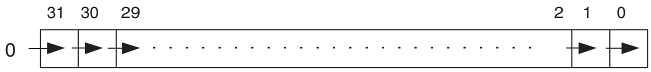
Logical Shift Right Immediate

Operation: RA ← (RA >> IMM5), zero-fill from left

Assembler Syntax: `LSRI %rA,IMM5`

Example: `LSRI %g1,6      ; Right-shift %g1 by 6 bits`

Description: The value in RA is shifted-right by the number of bits indicated by IMM5. The result is left-filled with zero.



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	0	1	0	0	1	IMM5					A				

MOV

Register-to-Register Move

Operation: RA ← RB

Assembler Syntax: `MOV %rA,%rB`

Example:

```
MOV %o0,%L3      ; copy %L3 into %o0
```

Description: Copy the contents of RB to RA.

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

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	1	1	0	0	B					A				

MOVHI

Move Immediate Into High Half-Word

Operation: $^{h1}RA \leftarrow (K : IMM5)$, ^{h0}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 31..16) of RA. The least significant half-word (bits 15..0) 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 31..16) of RA. The least significant half-word (bits 15..0) 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

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td colspan="5">IMM5</td><td colspan="5">A</td></tr>
</table>

MOVI

Move Immediate

Operation: RA ← (0x00.00 : K : IMM5)

Assembler Syntax: `MOVI %rA,IMM5`

Example: **Not preceded by PFX:**

```
MOVI %o3,7                ; load %o3 with 7
```

Preceded by PFX:

```
PFX %hi(301)
MOVI %o3,%lo(301)         ; load %o3 with 301
```

Description: **Not preceded by PFX:**

Loads register RA with a zero-extended 5-bit immediate value (in the range [0..31]) given by IMM5.

Preceded by PFX:

Loads register RA with a zero-extended 16-bit immediate value (in the range [0..65535]) given by (K : IMM5).

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: Ri5

Instruction Fields: A = Register index of RA operand
IMM5 = 5-bit immediate value

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td colspan="5">IMM5</td><td colspan="5">A</td></tr>
</table>

MSTEP

Multiply-Step

Operation:

```
If (R0[31]  = = 1)
then R0 ← (R0 << 1) + RA
else R0 ← (R0 << 1)
```

Assembler Syntax: `MSTEP %rA`

Example:

```
MSTEP %g1     ; accumulate partial-product
```

Description: Implements a single step of an unsigned multiply. The multiplier in %r0 and multiplicand in RA. Result is accumulated into %r0. RA is not affected.

The following code fragment implements a 16-bit × 16-bit into 32-bit multiply. On entry, %r0 and %r1 contain the multiplier and multiplicand, respectively. The result is left in %r0.

```
SWAP %r0 ; Move multiplier into place
MSTEP %r1
MSTEP %r1
MSTEP %r1
… A total of 16 MSTEPs …
MSTEP %r1
; 32-bit product left in %r0
```

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	1	0	1	0	0	A				

MUL

Multiply

Operation: R0 ← (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: 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	1	0	1	0	1	A				

NEG

Arithmetic Negation

Operation: $RA \leftarrow 0 - RA$

Assembler Syntax: `NEG %rA`

Example: `NEG %o4`

Description: Negate the value of RA. Perform two's complement negation 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	1	A				

NOP

Equivalent to MOV %g0, %g0

Operation: None

Assembler Syntax: `NOP`

Example: `NOP     ; do nothing`

Description: No operation.

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: RR

Instruction Fields: None

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 ← ~RA
Assembler Syntax:	`NOT %rA`
Example:	`NOT %o4`
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			A		

OR

Bitwise Logical OR

Operation:

Not preceded by PFX:

RA ← RA | RB

Preceded by PFX:

RA ← RA | (0x00.00 : K : IMM5)

Assembler Syntax:

Not preceded by PFX:

```
OR %ra,%rb
```

Preceded by PFX:

```
PFX %hi(const)
OR %ra,%lo(const)
```

Example:

Not preceded by PFX:

```
OR %i0,%i1             ; OR %i1 into %i0
```

Preceded by PFX:

```
PFX %hi(3333)
OR %i0,%lo(3333)       ; OR %i0 with 3333
```

Description:

Not preceded by PFX:

Logically-OR the individual bits in RA with the corresponding bits in RB; store the result in RA.

Preceded by PFX:

The RB operand is replaced by an immediate constant formed by concatenating the contents of the K-register (11 bits) with IMM5 (5 bits). This 16-bit value is zero-extended to 32 bits, then bitwise-ORed with RA. The result is written back into RA.

Condition Codes:

Flags:

N	V	Z	C
Δ	–	Δ	–

N: Result bit 31

Z: Set if result is zero; cleared otherwise

Instruction Format: RR, Ri5

Instruction Fields

A = Register index of operand RA

B = Register index of operand RB

IMM5 = 5-bit immediate value

Not preceded by PFX (RR)

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="5">B</td><td colspan="5">A</td></tr>
</table>

Preceded by PFX (Ri5)

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="5">IMM5</td><td colspan="5">A</td></tr>
</table>

PFX

Prefix

Operation: K ← IMM11 (K set to zero by all other instructions)

Assembler Syntax: `PFX IMM11`

Example:

```
PFX 3      ; affects next instruction
```

Description: Loads the 11-bit constant value IMM11 into the K-register. The value in the K register may affect the next instruction. K is set to zero after every instruction other than PFX and PFXIO. The result of two consecutive PFX instructions is not defined.

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: i11

Instruction Fields: IMM11 = 11-bit immediate value

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	IMM11										

PFXIO

Prefix with Cache Bypass

Operation: K ← IMM11 (K set to zero by all other instructions)

Assembler Syntax:

```
PFXIO IMM11
```

Example:

```
PFXIO 3       ; affects next instruction
```

Description: Loads the 11-bit constant value IMM11 into the K-register. The value in the K register may affect the next instruction. K is set to zero after every instruction other than PFX and PFXIO. PFXIO may only be used immediately before either an LD or LDP instruction. The result of PFXIO before any other instruction is undefined. See "PFX" on page 94.

PFXIO forces the subsequent LD or LDP memory-load operation to bypass the data cache (if present), even if the data-cache is enabled.

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: i11

Instruction Fields: IMM11 = 11-bit immediate value

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	IMM11										

RDCTL

Read Control Register

Operation: RA ← CTLk

Assembler Syntax:

```
RDCTL %rA
```

Example:

Not preceded by PFX:

```
RDCTL %g7       ; Loads %g7 from STATUS reg (%ctl0)
```

Preceded by PFX:

```
PFX 2
RDCTL %g7       ; Loads %g7 from WVALID reg (%ctl2)
```

Description:

Not preceded by PFX:

Loads RA with the current contents of the STATUS register (%ctl0).

Preceded by PFX:

Loads RA with the current contents of the control register selected by K. See “Control Registers” on page 16 for a list of control registers and their indices.

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	1	1	0	0	1	A				

RESTORE

Restore Caller's Register Window

Operation: CWP ← CWP + 1
if (old-CWP == HI_LIMIT)
then TRAP #2

Assembler Syntax: `RESTORE`

Example: `RESTORE     ; bump up the register window`

Description: Moves CWP up by one position in the register file. If CWP is equal to HI_LIMIT (from the WVALID register) before the RESTORE instruction, then a window-overflow trap (TRAP #2) is generated.

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: w

Instruction Fields: None

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 ← (%i7 << 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	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1

RLC

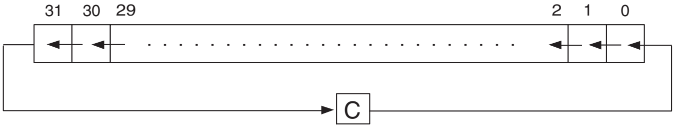
Rotate Left Through Carry

Operation: C ← RA[31]
RA ← (RA << 1) : C

Assembler Syntax: `RLC %rA`

Example: `RLC %i4     ; rotate %i4 left one bit`

Description: Rotates the bits of RA left by one position through the carry flag.



Condition Codes: Flags:

N	V	Z	C
–	–	–	Δ

C: Bit 31 of RA before rotating

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	1			A		

RRC

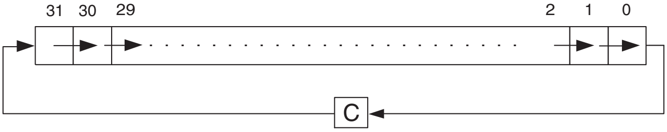
Rotate Right Through Carry

Operation: C ← RA[0]
RA ← C : (RA >> 1)

Assembler Syntax: `RRC %rA`

Example: `RRC %i4      ; rotate %i4 right one bit`

Description: Rotates the bits of RA right by one position through the carry flag.



If Preceded by PFX: Unaffected

Condition Codes: Flags:

N	V	Z	C
–	–	–	Δ

C: Bit 0 of RA before rotating

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	1	0	A				

SAVE

Save Caller's Register Window

Operation:

```
$CWP \leftarrow CWP - 1$
$\%sp \leftarrow \%fp - (IMM8 \times 4)$
If (old-CWP == LO_LIMIT)
then TRAP #1
```

Assembler Syntax:

```
SAVE %sp,-IMM8
```

Example:

```
SAVE %sp,-23       ; start subroutine with new regs
                   ; first operand can only be %sp
```

Description:

Moves CWP down by one position in the register file. If CWP is equal to LO_LIMIT (from the WVALID register) before the SAVE instruction, then a window-underflow trap (TRAP #1) is generated.

%sp (in the newly opened register window) is loaded with the value of %fp minus IMM8 times 4. %fp in the new window is the same as %sp in the old (caller's) window.

SAVE is conventionally used upon entry to subroutines to open up a new, disposable set of registers for the subroutine and simultaneously open up a stack-frame.

Condition Codes:

Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: i8v

Instruction Fields: IMM8 = 8-bit immediate value

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0	IMM8							

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 16..31 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	A				

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 8..31 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

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	1	1	A				

SKP0

Skip If Register Bit Is 0

(Equivalent to IF1 Instruction)

Operation:

```
if (RA[IMM5] == 0)
then begin
        if (Mem16[PC + 2] is PFX or PFXIO)
        then PC ← PC + 6
        else PC ← PC + 4
end
```

Assembler Syntax:

```
SKP0 %rA,IMM5
```

Example:

```
SKP0 %g6,11       ; skip if bit 11 is clear
ADDI %6,1         ; increment if bit 11 is set
```

Description: Skip next instruction if the single bit RA[IMM5] is 0. 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: 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	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0			IMM5					A		

SKP1

Skip If Register Bit Is 1

(Equivalent to IF0 Instruction)

Operation:

```
if (RA[IMM5] == 1)
then begin
    if (Mem16[PC + 2] is PFX or PFXIO)
    then PC ← PC + 6
    else PC ← PC + 4
end
```

Assembler Syntax:

```
SKP1 %rA,IMM5
```

Example:

```
SKP1 %o3,21      ; skip if 21st bit of %o3 is set
ADDI %g0, 1      ; increment if 21st bit is clear
```

Description: 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.

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	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1			IMM5					A		

SKPRNZ

Skip If Register Not Equal To 0

(Equivalent to IFRZ Instruction)

Operation:

```
if (RA != 0)
then begin
        if (Mem16[PC + 2] is PFX or PFXIO)
        then PC ← PC + 6
        else PC ← PC + 4
end
```

Assembler Syntax:

```
SKPRNZ %rA
```

Example:

```
SKPRNZ %g3
BSR SendIt      ; only call if %g3 is zero
NOP             ; (delay slot) executed in either case
```

Description: Skip next instruction if RA is not zero. If the next instruction is PFX or PFXIO, then both PFX or PFXIO and the instruction following PFX and PFXIO are skipped together.

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	1	1	0	1	0			A		

SKPRZ

Skip If Register Equals 0

Operation:

```
if (RA == 0)
then begin
       if (Mem16[PC + 2] is PFX or PFXIO)
       then PC ← PC + 6
       else PC ← PC + 4
end
```

Assembler Syntax:

```
SKPRZ %rA
```

Example:

```
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.

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	1	0	1	1	0	A				

SKPS

Skip On Condition Code

Operation:

```
if (condition IMM4 is true)
then begin
    if (Mem16[PC + 2] is PFX or PFXIO)
    then PC ← PC + 6
    else PC ← PC + 4
end
```

Assembler Syntax:

```
SKPS cc_IMM4
```

Example:

```
SKPS cc_ne
BSR SendIt      ; only call if Z flag clear
NOP             ; (delay slot) executed in either case
```

Description: 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:

These condition codes have different numeric values for IFS and SKPS instructions.

cc_c	0x0	(C)
cc_nc	0x1	(not C)
cc_z	0x2	(Z)
cc_nz	0x3	(not Z)
cc_mi	0x4	(N)
cc_pl	0x5	(not N)
cc_ge	0x6	(not (N xor V))
cc_lt	0x7	(N xor V)
cc_le	0x8	(Z or (N xor V))
cc_gt	0x9	(Not (Z or (N xorV)))
cc_v	0xa	(V)
cc_nv	0xb	(not V)
cc_la	0xc	(C or Z)
cc_hi	0xd	(not (C or Z))

Additional alias flags allowed:

cc_cs = cc_c	cc_n = cc_mi	cc_cc = cc_nc	cc_vc = cc_nv
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

Instruction Format: i4w

Instruction Fields: IMM4 = 4-bit immediate 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			

ST

Store 32-bit Data To Memory

Operation:

Not preceded by PFX:

Mem32[align32(RB)] ← RA

Preceded by PFX:

Mem32[align32(RB + (σ(K) × 4))] ← RA

Assembler Syntax:

```
ST [%rB],%rA
```

Example:

Not preceded by PFX:

```
ST [%g0],%i3      ; %g0 is pointer, %i3 stored
```

Preceded by PFX:

```
PFX 3             ; offset by 3 words
ST [%g0],%i3      ; store to location %g0 + 12
```

Description:

Not preceded by PFX:

Stores the 32-bit data value in RA to memory. Data is always written to a word-aligned address given by bits 31..2 of RB (the two LSBs of RB are ignored).

Preceded by PFX:

The value in K is sign-extended and used as a word-scaled, signed offset. This offset is added to the base-pointer address RB (bits 1..0 ignored), and data is written to the resulting word-aligned address.

Condition Codes:

Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: RR

Instruction Fields

A = Register index of operand RA

B = Register index of operand RB

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

ST16D

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

Operation:

Not preceded by PFX :

$^{hn}\mathrm{Mem32}[\mathrm{align32}(RA)] \leftarrow {}^{hn}R0$ where $n = RA[1]$

Preceded by PFX:

$^{hn}\mathrm{Mem32}[\mathrm{align32}(RA + (\sigma(K) \times 4))] \leftarrow {}^{hn}R0$ where $n = RA[1]$

Assembler Syntax:

```
ST16D [%rA],%r0
```

Example:

Not preceded by PFX:

```
FILL16 %r0,%g7       ; duplicate short of %g7 across %r0
ST16D [%o3],%r0      ; store %o3[1]th short int from
                     ; %r0 to [%o3]
                     ; second operand can only be %r0
```

Preceded by PFX:

```
FILL16 %r0,%g3
PFX 5
ST16D [%o3],%r0      ; same as above, offset
                     ; 20 bytes in memory
```

Description:

Not preceded by PFX:

Stores one of the two half-words of %r0 to memory at the half-word-aligned address given by RA. The bits RA[1] selects which half-word in %r0 get stored (half-word 1 is the most-significant). RA[0] is ignored.

ST16D may be used in combination with FILL16 to implement a two-instruction half-word-store operation. Given a half-word held in bits 15..0 of any register %r*X*, the following sequence writes this half-word to memory at the half-word-aligned address given by RA:

```
FILL16 %r0,%rX
ST16D [%rA],%r0
```

Preceded by PFX:

The value in K is sign-extended and used as a word-scaled, signed offset. This offset is added to the base-address RA and data is written to the resulting byte-address.

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	1	0	0	0	1	A				

ST16S

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

Operation:

Not preceded by PFX:

^{hn}Mem32[align32(RA)] ← ^{hn}R0 where n = IMM1

Preceded by PFX:

^{hn}Mem32[align32(RA + (σ(K) × 4))] ← ^{hn}R0 where n = IMM1

Assembler Syntax: `ST16S [%rA],%r0,IMM1`

Example: `ST16S [%g8],%r0,1`

Description:

Not preceded by PFX:

Stores one of the two half-words of %r0 to memory at the half-word-aligned address given by RA + (IMM1 x 2). RA is presumed to hold a word-aligned address. IMM1 selects which half-word of %r0 is stored (half-word #1 is most significant).

Preceded by PFX:

A 12-bit signed, half-word-scaled offset is formed by concatenating K with IMM1. This offset (K:IMM1) is half-word-scaled (multiplied by 2), sign-extended to 32 bits, and used as the half-word-aligned offset for the ST operation. This offset is applied to the base-address held in RA, which is presumed to be word-aligned.

IMM1 selects which of the two half-words of %r0 are stored at the indicated address (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 15..0 of any register %r*X*, 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:

```
FILL16 %r0,%rX
PFX Ymsbs             ; Top 11 bits of Y, incl. sign bit. (= (Y>>2) & 0x7FF)
ST16S [%rA], %r0, Y1  ; Bit 1 of Y (= Y & 2)
```

	12	11	10	9	8	7	6	5	4	3	2	1	0
Y (offset) =	--------------------- *Y*msbs ---------------------											*Y*1	0

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: Ri1u

Instruction Fields

A = Register index of operand RA

IMM1 = 1-bit immediate value

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	A				

ST8D

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

Operation:

Not preceded by PFX:

$^{bn}\mathrm{Mem32}[\mathrm{align32}(RA)] \leftarrow {}^{bn}R0$ where $n = RA[1..0]$

Preceded by PFX:

$^{bn}\mathrm{Mem32}[\mathrm{align32}(RA + \sigma(K) \times 4))] \leftarrow {}^{bn}R0$ where $n = RA[1..0]$

Assembler Syntax:

```
ST8D [%rA],%r0
```

Example:

Not preceded by PFX:

```
FILL8 %r0,%g7       ; duplicate low byte of %g7 across %r0
ST8D [%o3],%r0      ; store %o3[1..0]th byte from
                    ; %r0 to [%o3]
                    ; second operand can only be %r0
```

Preceded by PFX:

```
FILL8 %r0,%g3
PFX 5
ST8D [%o3],%r0      ; same as above, offset
                    ; 20 bytes in memory
```

Description:

Not preceded by PFX:

Stores one of the four bytes of %r0 to memory at the byte-address given by RA. The two bits RA[1..0] select which byte in %r0 get stored (byte 3 is the most-significant).

ST8D may be used in combination with FILL8 to implement a two-instruction byte-store operation. Given a byte held in bits 7..0 of any register %r*X*, the following sequence writes this byte to memory at the byte-address given by RA:

```
    FILL8 %r0,%rX
    ST8D [%rA],%r0
```

Preceded by PFX:

The value in K is used as a word-scaled, signed offset. This offset is added to the base-address RA and data is written to the resulting byte-address.

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	1	0	0	0	0	A				

ST8S

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

Operation:

Not preceded by PFX:
^{bn}Mem32[align32(RA)] ← ^{bn}R0 where n = IMM2
Preceded by PFX:
bnMem32[align32(RA + (σ(K) × 4))] ← bnR0 where n = IMM2

Assembler Syntax: `ST8S [%rA],%r0,IMM2`

Example:

Not preceded by PFX:

```
MOVI %g4,12
ST8S [%g4],%r0,3      ; store high byte of %r0 to mem[12]
```

Preceded by PFX:

```
PFX 9
ST8S [%g4],%r0,2      ; store byte 2 of %r0 to
                      ; mem[%g4 + 36 + 2]
```

Description:

Not preceded by PFX:
Stores one of the four bytes of %r0 to memory at the address given by RA + (IMM2). RA is presumed to hold a word-aligned address. IMM2 selects which byte of %r0 is stored (byte #3 is most significant).

Preceded by PFX:
A 13-bit signed offset is formed by concatenating K with IMM2. This offset (K:IMM2) is sign-extended to 32 bits and used as the byte-offset for the ST operation. The offset is applied to the base-address held in RA, which is presumed to be word-aligned.

IMM2 selects which of the four bytes of %r0 are stored at the indicated address (base + offset).

ST8S may be used in combination with FILL8 to implement a byte-store operation to any 13-bit signed offset from a word-aligned base address. Given a byte held in bits 7..0 of any register %r*X*, the following sequence writes this byte to memory at the address given by RA + *Y*, where RA is presumed to hold a word-aligned pointer, and *Y* is a signed 13-bit byte offset:

```
FILL8 %r0,%rX
PFX Ymsbs            ; Top 11 bits of Y, incl. sign bit. (= (Y>> 2) & 0x7FF)
ST8S [%rA], %r0, Ylsbs  ; Bits 1 and 0 of Y (= Y & 3)
```

	12	11	10	9	8	7	6	5	4	3	2	1	0
Y (offset) =	--------------------- *Y*msbs ---------------------											*Y*lsbs	

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: Ri2u

Instruction Fields: A = Register index of operand RA
IMM2 = 2-bit immediate value

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	IMM2		A				

STP

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

Operation:

Not preceded by PFX:

$\text{Mem32}[\text{align32}(\text{RP} + (\text{IMM5} \times 4))] \leftarrow \text{RA}$

Preceded by PFX:

$\text{Mem32}[\text{align32}(\text{RP} + (\sigma(\text{K} : \text{IMM5}) \times 4))] \leftarrow \text{RA}$

Assembler Syntax:

```
STP [%rP,IMM5],%rA
```

Example:

Not preceded by PFX:

```
STP [%L2,3],%g3         ; Store %g3 to location [%L2 + 12]
```

Preceded by PFX:

```
PFX %hi(102)
STP [%L2,%lo(102)],%g3   ; Store %g3 to
                         ; location [%L2 + 408]
```

Description:

Not preceded by PFX:

Stores the 32-bit data value in RA to memory. Data is always written to a word-aligned address given by bits [31..2] of RP (the two LSBs of RP are ignored) plus a 5-bit, unsigned, word-scaled offset given by IMM5.

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 be one of the four registers: %L0, %L1, %L2, or %L3.

Preceded by PFX:

A 16-bit offset is formed by concatenating the 11-bit K-register with IMM5 (5 bits). The 16-bit offset (K : IMM5) is sign-extended to 32 bits, multiplied by four, and added to bits 31..2 of RP to yield a word-aligned effective address.

Condition Codes:

Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format:

RPi5

Instruction Fields:

A = Register index of operand 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	P		IMM5					A				

STS

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

Operation: Mem32[align32(%sp + (IMM8 × 4))] ← 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 31..2 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)

Operation: ^{hn}Mem32[align32(%sp + IMM9 × 2)] ← ^{hn}R0 where n = IMM9[0]

Assembler Syntax: `STS16S [%sp,IMM9],%r0`

Example: `STS16S [%sp,7],%r0       ; can only be %sp and %r0`

Description: Stores one of the two half-words of %r0 to memory at the half-word-aligned address given by (%sp plus IMM9 × 2). The least-significant bit of IMM9 selects 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 store operation to a half-word offset from the stack-pointer in a 1 Kbyte range. Given a half-word held in bits 15..0 of any register %r*X*, the following sequence writes this half-word to memory at the half-word-offset *Y* from %sp (%sp presumed to hold a word-aligned address):

```
FILL16 %r0,%rX
STS16s [%sp,Y],%r0
```

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Instruction Format: i9

Instruction Fields: IMM9 = 9-bit immediate 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: ^{bn}Mem32[align32(%sp + IMM10)] ← ^{bn}R0 where n = IMM10[1..0]

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 7..0 of any register %r*X*, the following sequence writes this byte to memory at the byte-offset *Y* 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	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	IMM10									

SUB

Subtract

Operation: RA ← RA – RB

Assembler Syntax: `SUB %rA,%rB`

Example: `SUB %i3,%g0      ; SUB %g0 from %i3`

Description: Subtracts the contents of RB from RA, stores result in RA.

Condition Codes: Flags:

N	V	Z	C
Δ	Δ	Δ	Δ

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	0	1	0	B					A				

SUBI

Subtract Immediate

Operation: RA ← RA – (0x00.00 : K : IMM5)

Assembler Syntax:

```
subi %rB,IMM5
```

Example:

Not preceded by PFX:

```
SUBI %L5,6                ; subtract 6 from %L5
```

Preceded by PFX:

```
PFX %hi(1000)
SUBI %o3,%lo(1000)        ; subtract 1000 from %o3
```

Description:

Not preceded by PFX:

Subtracts the immediate value from the contents of RA. The immediate value is in the range of [0..31].

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 zero-extended to 32 bits and subtracted from register A.

Condition Codes: Flags:

N	V	Z	C
Δ	Δ	Δ	Δ

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: 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	0	0	0	1	1	IMM5					A				

SWAP

Swap Register Half-Words

Operation: $RA \leftarrow {}^{h0}RA : {}^{h1}RA$

Assembler Syntax: `SWAP %rA`

Example:

```
SWAP %g3      ; Exchange two half-words in %g3
```

Description: Swaps (exchanges positions) of the two 16-bit half-word values in RA. Writes result back into 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	1	0	0	0	A				

TRAP

Unconditional Trap

Operation:

ISTATUS ← STATUS
IE ← 0
CWP ← CWP – 1
IPRI ← IMM6
%o7 ← ((PC + 2) >> 1)
PC ← Mem32[VECBASE + (IMM6 × 4)] << 1

Assembler Syntax: `TRAP IMM6`

Example:

```
TRAP 2        ;invoke CWP window overflow exception handler
```

Description: CWP is decremented by one, opening a new register-window for the trap-handler. Interrupts are disabled (IE ← 0). The pre-TRAP STATUS register is copied into the ISTATUS register.

Transfer execution to trap handler number IMM6. The address of the trap-handler 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 × 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 TRAP instruction transfers execution to the indicated trap-handler even if the IE bit in the STATUS register is 0.

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.

Condition Codes: Flags: Unaffected

N	V	Z	C
–	–	–	–

Delay Slot Behavior 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.

Instruction Format: i6v

Instruction Fields: IMM6 = 6-bit immediate 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	IMM6					

TRET

Trap Return

Operation: PC ← (RA << 1)
STATUS ← ISTATUS

Assembler Syntax: `TRET %ra`

Example:

```
TRET %o7      ; return from TRAP
```

Description: Execution is transferred to the address given by (RA << 1). The value written in %o7 by TRAP is suitable for use as a return-address without modification.

The value in ISTATUS is copied into the STATUS register (this restores the pre-TRAP register window, because CWP is part of STATUS).

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	1	1	1	0	A				

USR0

User-defined Instruction

Operation: RA ← RA <user-defined operation> RB

Assembler Syntax: `USR0 %rA, %rB`

Example: `USR0 %o1,%i6`

Description: 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 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	C
–	–	–	–

Instruction Format: RR

Instruction Fields: A = Register index of operand RA
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	1	0	0			B					A		

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

User-defined Instruction

Operation: RA ← RA <user-defined operation> R0

Assembler Syntax: `USRx RA`

Example: `USR2 %o3`

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	C
–	–	–	–

Instruction Format: Rw

Instruction Fields: 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	A				

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	A				

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	A				

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	A				

WRCTL

Write Control Register

Operation: CTLk ← RA

Assembler Syntax: `WRCTL %rA`

Example: **Not preceded by PFX:**

```
WRCTL %g7       ; writes %g7 to STATUS reg
NOP             ; required
```

Preceded by PFX:

```
PFX 1
WRCTL %g7       ; writes %g7 to ISTATUS reg
```

Description: **Not preceded by PFX:**

Loads the STATUS register with RA. WRTCL to STATUS must be followed by a NOP instruction.

Preceded by PFX:

Writes the value in RA to the machine-control register selected by K. See Table 3 on page 15 for a list of the machine-control registers and their indices.

Condition Codes: If the target of WRCTL is the STATUS register, then the condition-code flags are directly set by the WRCTL operation from bits RA[3..0]. For any other WRCTL target register, the condition codes are unaffected.

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	0	0	0	A				

XOR

Bitwise Logical Exclusive OR

Operation:

Not preceded by PFX:

$RA \leftarrow RA \oplus RB$

Preceded by PFX:

$RA \leftarrow RA \oplus (0x00.00 : K : IMM5)$

Assembler Syntax:

Not preceded by PFX:

```
XOR %ra,%rb
```

Preceded by PFX:

```
PFX %hi(const)
XOR %rA,%lo(const)
```

Example:

Not preceded by PFX:

```
XOR %g0,%g1              ; XOR %g1 into %g0
```

Preceded by PFX:

```
PFX %hi(16383)
XOR %o0,%lo(16383)       ; XOR %o0 with 16383
```

Description:

Not preceded by PFX:

Logically-exclusive-OR the individual bits in RA with the corresponding bits in RB; store the result in RA.

Preceded by PFX:

When prefixed, the RB operand is replaced by an immediate constant formed by concatenating the contents of the K-register (11 bits) with IMM5 (5 bits). This 16-bit value is zero-extended to 32 bits, then bitwise-exclusive-ORed with RA. The result is written back into RA.

Condition Codes:

Flags:

N	V	Z	C
Δ	–	Δ	–

N: Result bit 31

Z: Set if result is zero, cleared otherwise

Instruction Format:

RR, Ri5

Instruction Fields:

A = Register index of operand RA

B = Register index of operand RB

IMM5 = 5-bit immediate value

Not preceded by PFX (RR)

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

Preceded by PFX (Ri5)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	IMM5					A				

Notes: