Instruction Set Reference

2015.04.02

NII51017

Subscribe Send Feedback

This section introduces the Nios® II instruction word format and provides a detailed reference of the Nios II instruction set.

Word Formats

There are three types of Nios II instruction word format: I-type, R-type, and J-type.

I-Type

The defining characteristic of the I-type instruction word format is that it contains an immediate value embedded within the instruction word. I-type instructions words contain:

- A 6-bit opcode field OP
- Two 5-bit register fields A and B
- A 16-bit immediate data field IMM16

In most cases, fields A and IMM16 specify the source operands, and field B specifies the destination register. IMM16 is considered signed except for logical operations and unsigned comparisons.

I-type instructions include arithmetic and logical operations such as `addi` and `andi`; branch operations; load and store operations; and cache management operations.

Table 1: I-Type Instruction Format

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										OP					

R-Type

The defining characteristic of the R-type instruction word format is that all arguments and results are specified as registers. R-type instructions contain:

- A 6-bit opcode field OP
- Three 5-bit register fields A, B, and C
- An 11-bit opcode-extension field OPX

In most cases, fields A and B specify the source operands, and field C specifies the destination register.

© 2015 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, ENPIRION, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

ISO 9001:2008 Registered

Some R-Type instructions embed a small immediate value in the five low-order bits of OPX. Unused bits in OPX are always 0.

R-type instructions include arithmetic and logical operations such as `add` and `nor`; comparison operations such as `cmpeq` and `cmplt`; the `custom` instruction; and other operations that need only register operands.

Table 2: R-Type Instruction Format

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

J-Type

J-type instructions contain:

- A 6-bit opcode field
- A 26-bit immediate data field

J-type instructions, such as `call` and `jmpi`, transfer execution anywhere within a 256-MB range.

Table 3: J-Type Instruction Format

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IMM26															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM26										OP					

Instruction Opcodes

The OP field in the Nios II instruction word specifies the major class of an opcode as listed in the two tables below. Most values of OP are encodings for I-type instructions. One encoding, OP = 0x00, is the J-type instruction `call`. Another encoding, OP = 0x3a, is used for all R-type instructions, in which case, the OPX field differentiates the instructions. All undefined encodings of OP and OPX are reserved.

Table 4: OP Encodings

OP	Instruction	OP	Instruction	OP	Instruction	OP	Instruction
0x00	call	0x10	cmplti	0x20	cmpeqi	0x30	cmpltui
0x01	jmpi	0x11		0x21		0x31	
0x02		0x12		0x22		0x32	custom
0x03	ldbu	0x13	initda	0x23	ldbuio	0x33	initd
0x04	addi	0x14	ori	0x24	muli	0x34	orhi

OP	Instruction	OP	Instruction	OP	Instruction	OP	Instruction
0x05	stb	0x15	stw	0x25	stbio	0x35	stwio
0x06	br	0x16	blt	0x26	beq	0x36	bltu
0x07	ldb	0x17	ldw	0x27	ldbio	0x37	ldwio
0x08	cmpgei	0x18	cmpnei	0x28	cmpgeui	0x38	rdprs
0x09		0x19		0x29		0x39	
0x0A		0x1A		0x2A		0x3A	R-type
0x0B	ldhu	0x1B	flushda	0x2B	ldhuio	0x3B	flushd
0x0C	andi	0x1C	xori	0x2C	andhi	0x3C	xorhi
0x0D	sth	0x1D		0x2D	sthio	0x3D	
0x0E	bge	0x1E	bne	0x2E	bgeu	0x3E	
0x0F	ldh	0x1F		0x2F	ldhio	0x3F	

Table 5: OPX Encodings for R-Type Instructions

OPX	Instruction	OPX	Instruction	OPX	Instruction	OPX	Instruction
0x00		0x10	cmplt	0x20	cmpeq	0x30	cmpltu
0x01	eret	0x11		0x21		0x31	add
0x02	roli	0x12	slli	0x22		0x32	
0x03	rol	0x13	sll	0x23		0x33	
0x04	flushp	0x14	wrprs	0x24	divu	0x34	break
0x05	ret	0x15		0x25	div	0x35	
0x06	nor	0x16	or	0x26	rdctl	0x36	sync
0x07	mulxuu	0x17	mulxsu	0x27	mul	0x37	
0x08	cmpge	0x18	cmpne	0x28	cmpgeu	0x38	
0x09	bret	0x19		0x29	initi	0x39	sub
0x0A		0x1A	srli	0x2A		0x3A	srai
0x0B	ror	0x1B	srl	0x2B		0x3B	sra
0x0C	flushi	0x1C	nextpc	0x2C		0x3C	
0x0D	jmp	0x1D	callr	0x2D	trap	0x3D	
0x0E	and	0x1E	xor	0x2E	wrctl	0x3E	
0x0F		0x1F	mulxss	0x2F		0x3F	

Assembler Pseudo-Instructions

Pseudo-instructions are used in assembly source code like regular assembly instructions. Each pseudo-instruction is implemented at the machine level using an equivalent instruction. The `movia` pseudo-

instruction is the only exception, being implemented with two instructions. Most pseudo-instructions do not appear in disassembly views of machine code.

Table 6: Assembler Pseudo-Instructions

Pseudo-Instruction	Equivalent Instruction
`bgt rA, rB, label`	`blt rB, rA, label`
`bgtu rA, rB, label`	`bltu rB, rA, label`
`ble rA, rB, label`	`bge rB, rA, label`
`bleu rA, rB, label`	`bgeu rB, rA, label`
`cmpgt rC, rA, rB`	`cmplt rC, rB, rA`
`cmpgti rB, rA, IMMED`	`cmpgei rB, rA, (IMMED+1)`
`cmpgtu rC, rA, rB`	`cmpltu rC, rB, rA`
`cmpgtui rB, rA, IMMED`	`cmpgeui rB, rA, (IMMED+1)`
`cmple rC, rA, rB`	`cmpge rC, rB, rA`
`cmplei rB, rA, IMMED`	`cmplti rB, rA, (IMMED+1)`
`cmpleu rC, rA, rB`	`cmpgeu rC, rB, rA`
`cmpleui rB, rA, IMMED`	`cmpltui rB, rA, (IMMED+1)`
`mov rC, rA`	`add rC, rA, r0`
`movhi rB, IMMED`	`orhi rB, r0, IMMED`
`movi rB, IMMED`	`addi, rB, r0, IMMED`
`movia rB, label`	`orhi rB, r0, %hiadj(label)` `addi, rB, r0, %lo(label)`
`movui rB, IMMED`	`ori rB, r0, IMMED`
`nop`	`add r0, r0, r0`
`subi rB, rA, IMMED`	`addi rB, rA, (-IMMED)`

Refer to the *Application Binary Interface* chapter of the *Nios II Processor Reference Handbook* for more information about global pointers.

Related Information

Application Binary Interface

Assembler Macros

The Nios II assembler provides macros to extract halfwords from labels and from 32-bit immediate values. These macros return 16-bit signed values or 16-bit unsigned values depending on where they are used. When used with an instruction that requires a 16-bit signed immediate value, these macros return a value ranging from –32768 to 32767. When used with an instruction that requires a 16-bit unsigned immediate value, these macros return a value ranging from 0 to 65535.

Table 7: Assembler Macros

Macro	Description	Operation
`%lo(immed32)`	Extract bits [15..0] of immed32	immed32 & 0xFFFF
`%hi(immed32)`	Extract bits [31..16] of immed32	(immed32 >> 16) & 0xFFFF
`%hiadj(immed32)`	Extract bits [31..16] and adds bit 15 of immed32	((immed32 >> 16) & 0xFFFF) + ((immed32 >> 15) & 0x1)
`%gprel(immed32)`	Replace the immed32 address with an offset from the global pointer	immed32 –_gp

Refer to the *Application Binary Interface* chapter of the *Nios II Processor Reference Handbook* for more information about global pointers.

Related Information

Application Binary Interface

Instruction Set Reference

The following pages list all Nios II instruction mnemonics in alphabetical order.

Table 8: Notation Conventions

Notation	Meaning
X ← Y	X is written with Y
PC ← X	The program counter (PC) is written with address X; the instruction at X is the next instruction to execute
PC	The address of the assembly instruction in question
rA, rB, rC	One of the 32-bit general-purpose registers
prs.rA	General-purpose register rA in the previous register set
IMMn	An n-bit immediate value, embedded in the instruction word
IMMED	An immediate value
X_n	The nth bit of X, where n = 0 is the LSB
$X_{n..m}$	Consecutive bits n through m of X
0xNNMM	Hexadecimal notation
X : Y	Bitwise concatenation For example, (0x12 : 0x34) = 0x1234
σ(X)	The value of X after being sign-extended to a full register-sized signed integer
X >> n	The value X after being right-shifted n bit positions
X << n	The value X after being left-shifted n bit positions
X & Y	Bitwise logical AND

Notation	Meaning
X | Y	Bitwise logical OR
X ^ Y	Bitwise logical XOR
~X	Bitwise logical NOT (one's complement)
Mem8[X]	The byte located in data memory at byte address X
Mem16[X]	The halfword located in data memory at byte address X
Mem32[X]	The word located in data memory at byte address X
label	An address label specified in the assembly file
(signed) rX	The value of rX treated as a signed number
(unsigned) rX	The value of rX treated as an unsigned number

Note: All register operations apply to the current register set, except as noted.

The following exceptions are not listed for each instruction because they can occur on any instruction fetch:

- Supervisor-only instruction address
- Fast TLB miss (instruction)
- Double TLB miss (instruction)
- TLB permission violation (execute)
- MPU region violation (instruction)

For information about these and all Nios II exceptions, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

Related Information

Programming Model

add

Instruction	add
Operation	rC ← rA + rB
Assembler Syntax	`add rC, rA, rB`
Example	`add r6, r7, r8`
Description	Calculates the sum of rA and rB. Stores the result in rC. Used for both signed and unsigned addition.

Usage	Carry Detection (unsigned operands):
	Following an add operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. The following code shows both cases:
	`add rC, rA, rB`
	`cmpltu rD, rC, rA`
	`add rC, rA, rB`
	`bltu rC, rA, label`
	`# The original add operation`
	`# rD is written with the carry bit`
	`# The original add operation`
	`# Branch if carry generated`
	Overflow Detection (signed operands):
	An overflow is detected when two positives are added and the sum is negative, or when two negatives are added and the sum is positive. The overflow condition can control a conditional branch, as shown in the following code:
	`add rC, rA, rB`
	`xor rD, rC, rA`
	`xor rE, rC, rB`
	`and rD, rD, rE`
	`blt rD, r0,label`
	`# The original add operation`
	`# Compare signs of sum and rA`
	`# Compare signs of sum and rB`
	`# Combine comparisons`
	`# Branch if overflow occurred`

Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x31
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x31					0					0x3A					

addi

Instruction	addi
Operation	rB ← rA + σ(IMM16)
Assembler Syntax	`addi rB, rA, IMM16`
Example	`addi r6, r7, -100`
Description	Sign-extends the 16-bit immediate value and adds it to the value of rA. Stores the sum in rB.

Usage	Carry Detection (unsigned operands):
	Following an `addi` operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. The following code shows both cases:
	`addi rB, rA, IMM16`
	`cmpltu rD, rB, rA`
	`addi rB, rA, IMM16`
	`bltu rB, rA, label`
	`# The original add operation`
	`# rD is written with the carry bit`
	`# The original add operation`
	`# Branch if carry generated`
	Overflow Detection (signed operands):
	An overflow is detected when two positives are added and the sum is negative, or when two negatives are added and the sum is positive. The overflow condition can control a conditional branch, as shown in the following code:
	`addi rB, rA, IMM16`
	`xor rC, rB, rA`
	`xorhi rD, rB, IMM16`
	`and rC, rC, rD`
	`blt rC, r0,label`
	`# The original add operation`
	`# Compare signs of sum and rA`
	`# Compare signs of sum and IMM16`
	`# Combine comparisons`
	`# Branch if overflow occurred`

Exceptions	None
Instruction Type	I

Instruction Fields	A = Register index of operand rA B = Register index of operand rB IMM16 = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x04					

and

Instruction	bitwise logical and
Operation	rC ← rA & rB
Assembler Syntax	`and rC, rA, rB`
Example	`and r6, r7, r8`
Description	Calculates the bitwise logical AND of rA and rB and stores the result in rC.
Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x0e
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0e					0					0x3A					

andhi

Instruction	bitwise logical and immediate into high halfword
Operation	rB ← rA & (IMM16 : 0x0000)

Assembler Syntax	`andhi rB, rA, IMM16`
Example	`andhi r6, r7, 100`
Description	Calculates the bitwise logical AND of rA and (IMM16 : 0x0000) and stores the result in rB.
Exceptions	None
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x2c					

andi

Instruction	bitwise logical and immediate
Operation	rB ← rA & (0x0000 : IMM16)
Assembler Syntax	`andi rB, rA, IMM16`
Example	`andi r6, r7, 100`
Description	Calculates the bitwise logical AND of rA and (0x0000 : IMM16) and stores the result in rB.
Exceptions	None
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

Bit Fields															
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x0c					

beq

Instruction	branch if equal
Operation	if (rA == rB) then PC ← PC + 4 + σ(IMM16) else PC ← PC + 4
Assembler Syntax	`beq rA, rB, label`
Example	`beq r6, r7, label`
Description	If rA == rB, then `beq` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `beq`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.
Exceptions	Misaligned destination address
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x26					

bge

Instruction	branch if greater than or equal signed

Operation	if ((signed) rA >= (signed) rB) then PC ← PC + 4 + σ(IMM16) else PC ← PC + 4
Assembler Syntax	`bge rA, rB, label`
Example	`bge r6, r7, top_of_loop`
Description	If (signed) rA >= (signed) rB, then `bge` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `bge`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.
Exceptions	Misaligned destination address
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x0e					

bgeu

Instruction	branch if greater than or equal unsigned
Operation	if ((unsigned) rA >= (unsigned) rB) then PC ← PC + 4 + σ(IMM16) else PC ← PC + 4
Assembler Syntax	`bgeu rA, rB, label`
Example	`bgeu r6, r7, top_of_loop`

Description	If (unsigned) rA >= (unsigned) rB, then `bgeu` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `bgeu`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.
Exceptions	Misaligned destination address
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x2e					

bgt

Instruction	branch if greater than signed
Operation	if ((signed) rA > (signed) rB) then PC ← label else PC ← PC + 4
Assembler Syntax	`bgt rA, rB, label`
Example	`bgt r6, r7, top_of_loop`
Description	If (signed) rA > (signed) rB, then `bgt` transfers program control to the instruction at label.
Pseudo-instruction	`bgt` is implemented with the `blt` instruction by swapping the register operands.

bgtu

Instruction	branch if greater than unsigned

Operation	if ((unsigned) rA > (unsigned) rB) then PC ← label else PC ← PC + 4
Assembler Syntax	`bgtu rA, rB, label`
Example	`bgtu r6, r7, top_of_loop`
Description	If (unsigned) rA > (unsigned) rB, then `bgtu` transfers program control to the instruction at label.
Pseudo-instruction	`bgtu` is implemented with the `bltu` instruction by swapping the register operands.

ble

Instruction	branch if less than or equal signed
Operation	if ((signed) rA <= (signed) rB) then PC ← label else PC ← PC + 4
Assembler Syntax	`ble rA, rB, label`
Example	`ble r6, r7, top_of_loop`
Description	If (signed) rA <= (signed) rB, then `ble` transfers program control to the instruction at label.
Pseudo-instruction	`ble` is implemented with the `bge` instruction by swapping the register operands.

bleu

Instruction	branch if less than or equal to unsigned
Operation	if ((unsigned) rA <= (unsigned) rB) then PC ← label else PC ← PC + 4
Assembler Syntax	`bleu rA, rB, label`
Example	`bleu r6, r7, top_of_loop`

Description	If (unsigned) rA <= (unsigned) rB, then `bleu` transfers program counter to the instruction at label.
Pseudo-instruction	`bleu` is implemented with the `bgeu` instruction by swapping the register operands.

blt

Instruction	branch if less than signed
Operation	if ((signed) rA < (signed) rB) then PC ← PC + 4 + σ(IMM16) else PC ← PC + 4
Assembler Syntax	`blt rA, rB, label`
Example	`blt r6, r7, top_of_loop`
Description	If (signed) rA < (signed) rB, then `blt` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `blt`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.
Exceptions	Misaligned destination address
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x16					

bltu

Instruction	branch if less than unsigned

Operation	if ((unsigned) rA < (unsigned) rB) then PC ← PC + 4 + σ(IMM16) else PC ← PC + 4
Assembler Syntax	`bltu rA, rB, label`
Example	`bltu r6, r7, top_of_loop`
Description	If (unsigned) rA < (unsigned) rB, then `bltu` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `bltu`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.
Exceptions	Misaligned destination address
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x36					

bne

Instruction	branch if not equal
Operation	if (rA != rB) then PC ← PC + 4 + σ(IMM16) else PC ← PC + 4
Assembler Syntax	`bne rA, rB, label`
Example	`bne r6, r7, top_of_loop`

Description	If rA != rB, then bne transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following bne. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.
Exceptions	Misaligned destination address
Instruction Type	I
Instruction Fields	A = Register index of operand rA B = Register index of operand rB IMM16 = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x1e					

br

Instruction	unconditional branch
Operation	PC ← PC + 4 + σ(IMM16)
Assembler Syntax	`br label`
Example	`br top_of_loop`
Description	Transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following br. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.
Exceptions	Misaligned destination address
Instruction Type	I
Instruction Fields	IMM16 = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0					0					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x06					

break

Instruction	debugging breakpoint
Operation	`bstatus` ← `status` PIE ← 0 U ← 0 `ba` ← PC + 4 PC ← break handler address
Assembler Syntax	`break` `break imm5`
Example	`break`
Description	Breaks program execution and transfers control to the debugger break-processing routine. Saves the address of the next instruction in register `ba` and saves the contents of the `status` register in `bstatus`. Disables interrupts, then transfers execution to the break handler. The 5-bit immediate field `imm5` is ignored by the processor, but it can be used by the debugger. `break` with no argument is the same as `break 0`.
Usage	`break` is used by debuggers exclusively. Only debuggers should place `break` in a user program, operating system, or exception handler. The address of the break handler is specified with the Nios_II Processor parameter editor in Qsys. Some debuggers support `break` and `break 0` instructions in source code. These debuggers treat the `break` instruction as a normal breakpoint.
Exceptions	Break
Instruction Type	R
Instruction Fields	`IMM5` = Type of breakpoint

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0					0					0x1e					0x34
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x34					IMM5					0x3a					

bret

Instruction	breakpoint return
Operation	`status` ← `bstatus` PC ← `ba`
Assembler Syntax	`bret`
Example	`bret`
Description	Copies the value of `bstatus` to the `status` register, then transfers execution to the address in `ba`.
Usage	`bret` is used by debuggers exclusively and should not appear in user programs, operating systems, or exception handlers.
Exceptions	Misaligned destination address Supervisor-only instruction
Instruction Type	R
Instruction Fields	None

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x1e					0					0x1e					0x09
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x09					0					0x3a					

call

Instruction	call subroutine
Operation	`ra` ← PC + 4 PC ← ($PC_{31..28}$: IMM26 x 4)

Assembler Syntax	`call label`
Example	`call write_char`
Description	Saves the address of the next instruction in register `ra`, and transfers execution to the instruction at address ($PC_{31..28}$: IMM26 x 4).
Usage	`call` can transfer execution anywhere within the 256-MB range determined by $PC_{31..28}$. The Nios II GNU linker does not automatically handle cases in which the address is out of this range.
Exceptions	None
Instruction Type	J
Instruction Fields	`IMM26` = 26-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IMM26															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM26										0					

callr

Instruction	call subroutine in register
Operation	`ra` ← PC + 4 PC ← rA
Assembler Syntax	`callr rA`
Example	`callr r6`
Description	Saves the address of the next instruction in the return address register, and transfers execution to the address contained in register rA.
Usage	`callr` is used to dereference C-language function pointers.
Exceptions	Misaligned destination address
Instruction Type	R

Instruction Fields	A = Register index of operand rA

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					0x1f					0x1d
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1d					0					0x3a					

cmpeq

Instruction	compare equal
Operation	if (rA == rB) then rC ← 1 else rC ← 0
Assembler Syntax	`cmpeq rC, rA, rB`
Example	`cmpeq r6, r7, r8`
Description	If rA == rB, then stores 1 to rC; otherwise, stores 0 to rC.

Usage	`cmpeq` performs the == operation of the C programming language. Also, `cmpeq` can be used to implement the C logical negation operator "!".
	`cmpeq rC, rA, r0`
	`# Implements rC = !rA`

Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x20
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x20					0					0x3a					

cmpeqi

Instruction	compare equal immediate
Operation	if (rA σ(IMM16)) then rB ← 1 else rB ← 0
Assembler Syntax	`cmpeqi rB, rA, IMM16`
Example	`cmpeqi r6, r7, 100`
Description	Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA == σ(IMM16), `cmpeqi` stores 1 to rB; otherwise stores 0 to rB.
Usage	`cmpeqi` performs the == operation of the C programming language.
Exceptions	None
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x20					

cmpge

Instruction	compare greater than or equal signed
Operation	if ((signed) rA >= (signed) rB) then rC ← 1 else rC ← 0
Assembler Syntax	`cmpge rC, rA, rB`
Example	`cmpge r6, r7, r8`

Description	If rA >= rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage	`cmpge` performs the signed >= operation of the C programming language.
Exceptions	None
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x08
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x08					0					0x3a					

cmpgei

Instruction	compare greater than or equal signed immediate
Operation	if ((signed) rA >= (signed) σ(IMM16)) then rB ← 1 else rB ← 0
Assembler Syntax	`cmpgei rB, rA, IMM16`
Example	`cmpgei r6, r7, 100`
Description	Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA >= σ(IMM16), then `cmpgei` stores 1 to rB; otherwise stores 0 to rB.
Usage	`cmpgei` performs the signed >= operation of the C programming language.
Exceptions	None
Instruction Type	R

Instruction Fields	A = Register index of operand rA B = Register index of operand rB IMM16 = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x08					

cmpgeu

Instruction	compare greater than or equal unsigned
Operation	if ((unsigned) rA >= (unsigned) rB) then rC ← 1 else rC ← 0
Assembler Syntax	`cmpgeu rC, rA, rB`
Example	`cmpgeu r6, r7, r8`
Description	If rA >= rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage	`cmpgeu` performs the unsigned >= operation of the C programming language.
Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x28
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x28					0					0x3a					

cmpgeui

Instruction	compare greater than or equal unsigned immediate
Operation	if ((unsigned) rA >= (unsigned) (0x0000 : IMM16)) then rB ← 1 else rB ← 0
Assembler Syntax	`cmpgeui rB, rA, IMM16`
Example	`cmpgeui r6, r7, 100`
Description	Zero-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA >= (0x0000 : IMM16), then `cmpgeui` stores 1 to rB; otherwise stores 0 to rB.
Usage	`cmpgeui` performs the unsigned >= operation of the C programming language.
Exceptions	None
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x28					

cmpgt

Instruction	compare greater than signed
Operation	if ((signed) rA > (signed) rB) then rC ← 1 else rC ← 0
Assembler Syntax	`cmpgt rC, rA, rB`
Example	`cmpgt r6, r7, r8`

Description	If rA > rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage	`cmpgt` performs the signed > operation of the C programming language.
Pseudo-instruction	`cmpgt` is implemented with the `cmplt` instruction by swapping its rA and rB operands.

cmpgti

Instruction	compare greater than signed immediate
Operation	if ((signed) rA > (signed) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax	`cmpgti rB, rA, IMMED`
Example	`cmpgti r6, r7, 100`
Description	Sign-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA > σ(IMMED), then `cmpgti` stores 1 to rB; otherwise stores 0 to rB.
Usage	`cmpgti` performs the signed > operation of the C programming language. The maximum allowed value of IMMED is 32766. The minimum allowed value is –32769.
Pseudo-instruction	`cmpgti` is implemented using a `cmpgei` instruction with an IMM16 immediate value of IMMED + 1.

cmpgtu

Instruction	compare greater than unsigned
Operation	if ((unsigned) rA > (unsigned) rB) then rC ← 1 else rC ← 0
Assembler Syntax	`cmpgtu rC, rA, rB`
Example	`cmpgtu r6, r7, r8`

Description	If rA > rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage	`cmpgtu` performs the unsigned > operation of the C programming language.
Pseudo-instruction	`cmpgtu` is implemented with the `cmpltu` instruction by swapping its rA and rB operands.

cmpgtui

Instruction	compare greater than unsigned immediate
Operation	if ((unsigned) rA > (unsigned) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax	`cmpgtui rB, rA, IMMED`
Example	`cmpgtui r6, r7, 100`
Description	Zero-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA > IMMED, then `cmpgtui` stores 1 to rB; otherwise stores 0 to rB.
Usage	`cmpgtui` performs the unsigned > operation of the C programming language. The maximum allowed value of IMMED is 65534. The minimum allowed value is 0.
Pseudo-instruction	`cmpgtui` is implemented using a `cmpgeui` instruction with an IMM16 immediate value of IMMED + 1.

cmple

Instruction	compare less than or equal signed
Operation	if ((signed) rA <= (signed) rB) then rC ← 1 else rC ← 0
Assembler Syntax	`cmple rC, rA, rB`
Example	`cmple r6, r7, r8`

Description	If rA <= rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage	`cmple` performs the signed <= operation of the C programming language.
Pseudo-instruction	`cmple` is implemented with the `cmpge` instruction by swapping its rA and rB operands.

cmplei

Instruction	compare less than or equal signed immediate
Operation	if ((signed) rA < (signed) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax	`cmplei rB, rA, IMMED`
Example	`cmplei r6, r7, 100`
Description	Sign-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA <= σ(IMMED), then `cmplei` stores 1 to rB; otherwise stores 0 to rB.
Usage	`cmplei` performs the signed <= operation of the C programming language. The maximum allowed value of IMMED is 32766. The minimum allowed value is –32769.
Pseudo-instruction	`cmplei` is implemented using a `cmplti` instruction with an IMM16 immediate value of IMMED + 1.

cmpleu

Instruction	compare less than or equal unsigned
Operation	if ((unsigned) rA < (unsigned) rB) then rC ← 1 else rC ← 0
Assembler Syntax	`cmpleu rC, rA, rB`
Example	`cmpleu r6, r7, r8`

Description	If rA <= rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage	`cmpleu` performs the unsigned <= operation of the C programming language.
Pseudo-instruction	`cmpleu` is implemented with the `cmpgeu` instruction by swapping its rA and rB operands.

cmpleui

Instruction	compare less than or equal unsigned immediate
Operation	if ((unsigned) rA <= (unsigned) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax	`cmpleui rB, rA, IMMED`
Example	`cmpleui r6, r7, 100`
Description	Zero-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA <= IMMED, then `cmpleui` stores 1 to rB; otherwise stores 0 to rB.
Usage	`cmpleui` performs the unsigned <= operation of the C programming language. The maximum allowed value of IMMED is 65534. The minimum allowed value is 0.
Pseudo-instruction	`cmpleui` is implemented using a `cmpltui` instruction with an IMM16 immediate value of IMMED + 1.

cmplt

Instruction	compare less than signed
Operation	if ((signed) rA < (signed) rB) then rC ← 1 else rC ← 0
Assembler Syntax	`cmplt rC, rA, rB`
Example	`cmplt r6, r7, r8`

Description	If rA < rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage	`cmplt` performs the signed < operation of the C programming language.
Exceptions	None
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x10
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x10					0					0x3a					

cmplti

Instruction	compare less than signed immediate
Operation	if ((signed) rA < (signed) σ(IMM16)) then rB ← 1 else rB ← 0
Assembler Syntax	`cmplti rB, rA, IMM16`
Example	`cmplti r6, r7, 100`
Description	Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA < σ(IMM16), then `cmplti` stores 1 to rB; otherwise stores 0 to rB.
Usage	`cmplti` performs the signed < operation of the C programming language.
Exceptions	None
Instruction Type	I

Instruction Fields	A = Register index of operand rA B = Register index of operand rB IMM16 = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x10					

cmpltu

Instruction	compare less than unsigned
Operation	if ((unsigned) rA < (unsigned) rB) then rC ← 1 else rC ← 0
Assembler Syntax	`cmpltu rC, rA, rB`
Example	`cmpltu r6, r7, r8`
Description	If rA < rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage	`cmpltu` performs the unsigned < operation of the C programming language.
Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x30
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x30					0					0x3a					

cmpltui

Instruction	compare less than unsigned immediate
Operation	if ((unsigned) rA < (unsigned) (0x0000 : IMM16)) then rB ← 1 else rB ← 0
Assembler Syntax	`cmpltui rB, rA, IMM16`
Example	`cmpltui r6, r7, 100`
Description	Zero-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA < (0x0000 : IMM16), then `cmpltui` stores 1 to rB; otherwise stores 0 to rB.
Usage	`cmpltui` performs the unsigned < operation of the C programming language.
Exceptions	None
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x30					

cmpne

Instruction	compare not equal
Operation	if (rA != rB) then rC ← 1 else rC ← 0
Assembler Syntax	`cmpne rC, rA, rB`
Example	`cmpne r6, r7, r8`

Description	If rA != rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage	`cmpne` performs the != operation of the C programming language.
Exceptions	None
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x18
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x18					0					0x3a					

cmpnei

Instruction	compare not equal immediate
Operation	if (rA != σ(IMM16)) then rB ← 1 else rB ← 0
Assembler Syntax	`cmpnei rB, rA, IMM16`
Example	`cmpnei r6, r7, 100`
Description	Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA != σ(IMM16), then `cmpnei` stores 1 to rB; otherwise stores 0 to rB.
Usage	`cmpnei` performs the != operation of the C programming language.
Exceptions	None
Instruction Type	I

Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x18					

custom

Instruction	custom instruction
Operation	if c == 1 then rC ← f_N(rA, rB, A, B, C) else Ø ← f_N(rA, rB, A, B, C)
Assembler Syntax	`custom N, xC, xA, xB` Where xA means either general purpose register rA, or custom register cA.
Example	`custom 0, c6, r7, r8`
Description	The `custom` opcode provides access to up to 256 custom instructions allowed by the Nios II architecture. The function implemented by a custom instruction is user-defined and is specified with the Nios_II Processor parameter editor in Qsys. The 8-bit immediate N field specifies which custom instruction to use. Custom instructions can use up to two parameters, xA and xB, and can optionally write the result to a register xC.
Usage	To access a custom register inside the custom instruction logic, clear the bit readra, readrb, or writerc that corresponds to the register field. In assembler syntax, the notation cN refers to register N in the custom register file and causes the assembler to clear the c bit of the opcode. For example, `custom 0, c3, r5, r0` performs custom instruction 0, operating on general-purpose registers r5 and r0, and stores the result in custom register 3.
Exceptions	None

Instruction Type	R
Instruction Fields	`A` = Register index of operand A `B` = Register index of operand B `C` = Register index of operand C `readra` = 1 if instruction uses rA, 0 otherwise `readrb` = 1 if instruction uses rB, 0 otherwise `writerc` = 1 if instruction provides result for rC, 0 otherwise `N` = 8-bit number that selects instruction

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					readra
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
readrb	readrc	N								0x32					

div

Instruction	divide
Operation	rC ← rA ÷ rB
Assembler Syntax	`div rC, rA, rB`
Example	`div r6, r7, r8`
Description	Treating rA and rB as signed integers, this instruction divides rA by rB and then stores the integer portion of the resulting quotient to rC. After attempted division by zero, the value of rC is undefined. There is no divide-by-zero exception. After dividing –2147483648 by –1, the value of rC is undefined (the number +2147483648 is not representable in 32 bits). There is no overflow exception. Nios II processors that do not implement the `div` instruction cause an unimplemented instruction exception.

Usage	Remainder of Division:
	If the result of the division is defined, then the remainder can be computed in rD using the following instruction sequence:
	`div rC, rA, rB`
	`mul rD, rC, rB`
	`sub rD, rA, rD`
	`# The original div operation`
	`# rD = remainder`

Exceptions	Division error Unimplemented instruction
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x18
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x25					0					0x3a					

divu

Instruction	divide unsigned
Operation	rC ← rA ÷ rB
Assembler Syntax	`divu rC, rA, rB`
Example	`divu r6, r7, r8`
Description	Treating rA and rB as unsigned integers, this instruction divides rA by rB and then stores the integer portion of the resulting quotient to rC. After attempted division by zero, the value of rC is undefined. There is no divide-by-zero exception. Nios II processors that do not implement the `divu` instruction cause an unimplemented instruction exception.

Usage	Remainder of Division:
	If the result of the division is defined, then the remainder can be computed in rD using the following instruction sequence:
	`divu rC, rA, rB`
	`mul rD, rC, rB`
	`sub rD, rA, rD`
	`# The original divu operation`
	`# rD = remainder`

Exceptions	Division error Unimplemented instruction
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x24
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x24					0					0x3a					

eret

Instruction	exception return
Operation	`status` ← `estatus` PC ← `ea`
Assembler Syntax	`eret`
Example	`eret`
Description	Copies the value of `estatus` into the `status` register, and transfers execution to the address in `ea`.

Usage	Use `eret` to return from traps, external interrupts, and other exception handling routines. Note that before returning from hardware interrupt exceptions, the exception handler must adjust the ea register.
Exceptions	Misaligned destination address Supervisor-only instruction
Instruction Type	R
Instruction Fields	None

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x1d					0x1e					C					0x01
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x01					0					0x3a					

flushd

Instruction	flush data cache line
Operation	Flushes the data cache line associated with address rA + σ(IMM16).
Assembler Syntax	`flushd IMM16(rA)`
Example	`flushd -100(r6)`

Description	If the Nios II processor implements a direct mapped data cache, `flushd` writes the data cache line that is mapped to the specified address back to memory if the line is dirty, and then clears the data cache line. Unlike `flushda`, `flushd` writes the dirty data back to memory even when the addressed data is not currently in the cache. This process comprises the following steps: • Compute the effective address specified by the sum of rA and the signed 16-bit immediate value. • Identify the data cache line associated with the computed effective address. Each data cache effective address comprises a `tag` field and a `line` field. When identifying the data cache line, `flushd` ignores the `tag` field and only uses the `line` field to select the data cache line to clear. • Skip comparing the cache line tag with the effective address to determine if the addressed data is currently cached. Because `flushd` ignores the cache line tag, `flushd` flushes the cache line regardless of whether the specified data location is currently cached. • If the data cache line is dirty, write the line back to memory. A cache line is dirty when one or more words of the cache line have been modified by the processor, but are not yet written to memory. • Clear the valid bit for the line. If the Nios II processor core does not have a data cache, the `flushd` instruction performs no operation.
Usage	Use `flushd` to write dirty lines back to memory even if the addressed memory location is not in the cache, and then flush the cache line. By contrast, refer to "flushda flush data cache address", "initd initialize data cache line", and "initda initialize data cache address" for other cache-clearing options. For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
Exceptions	None
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x3b					

Related Information

- **Cache and Tightly-Coupled Memory**
- **flushda** on page 41
- **initda** on page 46
- **initd** on page 44

flushda

Instruction	flush data cache address
Operation	Flushes the data cache line currently caching address rA + σ(IMM16)
Assembler Syntax	`flushda IMM16(rA)`
Example	`flushda -100(r6)`

Description	If the Nios II processor implements a direct mapped data cache, `flushda` writes the data cache line that is mapped to the specified address back to memory if the line is dirty, and then clears the data cache line. Unlike `flushd`, `flushda` writes the dirty data back to memory only when the addressed data is currently in the cache. This process comprises the following steps: • Compute the effective address specified by the sum of rA and the signed 16-bit immediate value. • Identify the data cache line associated with the computed effective address. Each data cache effective address comprises a `tag` field and a `line` field. When identifying the line, `flushda` uses both the `tag` field and the `line` field. • Compare the cache line tag with the effective address to determine if the addressed data is currently cached. If the `tag` fields do not match, the effective address is not currently cached, so the instruction does nothing. • If the data cache line is dirty and the `tag` fields match, write the dirty cache line back to memory. A cache line is dirty when one or more words of the cache line have been modified by the processor, but are not yet written to memory. • Clear the valid bit for the line. If the Nios II processor core does not have a data cache, the `flushda` instruction performs no operation.
Usage	Use `flushda` to write dirty lines back to memory only if the addressed memory location is currently in the cache, and then flush the cache line. By contrast, refer to "flushd flush data cache line", "initd initialize data cache line", and "initda initialize data cache address" for other cache-clearing options. For more information on the Nios II data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
Exceptions	Supervisor-only data address Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)

Instruction Type	I
Instruction Fields	A = Register index of operand rA IMM16 = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x1b					

Related Information

- **Cache and Tightly-Coupled Memory**
- **initda** on page 46
- **initd** on page 44
- **flushd** on page 39

flushi

Instruction	flush instruction cache line
Operation	Flushes the instruction cache line associated with address rA.
Assembler Syntax	`flushi rA`
Example	`flushi r6`
Description	Ignoring the tag, `flushi` identifies the instruction cache line associated with the byte address in rA, and invalidates that line. If the Nios II processor core does not have an instruction cache, the `flushi` instruction performs no operation. For more information about the data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					0					0x0c
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0c					0					0x3a					

Related Information

Cache and Tightly-Coupled Memory

flushp

Instruction	flush pipeline
Operation	Flushes the processor pipeline of any prefetched instructions.
Assembler Syntax	`flushp`
Example	`flushp`
Description	Ensures that any instructions prefetched after the `flushp` instruction are removed from the pipeline.
Usage	Use `flushp` before transferring control to newly updated instruction memory.
Exceptions	None
Instruction Type	R
Instruction Fields	None

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					0					0x04
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x04					0					0x3a					

initd

Instruction	initialize data cache line
Operation	Initializes the data cache line associated with address rA + σ(IMM16).

Assembler Syntax	`initd IMM16(rA)`
Example	`initd 0(r6)`
Description	If the Nios II processor implements a direct mapped data cache, `initd` clears the data cache line without checking for (or writing) a dirty data cache line that is mapped to the specified address back to memory. Unlike `initda`, `initd` clears the cache line regardless of whether the addressed data is currently cached. This process comprises the following steps: • Compute the effective address specified by the sum of rA and the signed 16-bit immediate value. • Identify the data cache line associated with the computed effective address. Each data cache effective address comprises a `tag` field and a `line` field. When identifying the line, `initd` ignores the `tag` field and only uses the `line` field to select the data cache line to clear. • Skip comparing the cache line tag with the effective address to determine if the addressed data is currently cached. Because `initd` ignores the cache line tag, `initd` flushes the cache line regardless of whether the specified data location is currently cached. • Skip checking if the data cache line is dirty. Because `initd` skips the dirty cache line check, data that has been modified by the processor, but not yet written to memory is lost. • Clear the valid bit for the line. If the Nios II processor core does not have a data cache, the `initd` instruction performs no operation.
Usage	Use `initd` after processor reset and before accessing data memory to initialize the processor's data cache. Use `initd` with caution because it does not write back dirty data. By contrast, refer to "flushd flush data cache line", "flushda flush data cache address", and "initda initialize data cache address" for other cache-clearing options. Altera recommends using `initd` only when the processor comes out of reset. For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
Exceptions	Supervisor-only instruction
Instruction Type	I

Instruction Fields	`A` = Register index of operand rA `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x33					

Related Information

- **Cache and Tightly-Coupled Memory**
- **flushda** on page 41
- **initda** on page 46
- **flushd** on page 39

initda

Instruction	initialize data cache address
Operation	Initializes the data cache line currently caching address rA + σ(IMM16)
Assembler Syntax	`initda IMM16(rA)`
Example	`initda -100(r6)`

Description	If the Nios II processor implements a direct mapped data cache, `initda` clears the data cache line without checking for (or writing) a dirty data cache line that is mapped to the specified address back to memory. Unlike `initd`, `initda` clears the cache line only when the addressed data is currently cached. This process comprises the following steps: • Compute the effective address specified by the sum of rA and the signed 16-bit immediate value. • Identify the data cache line associated with the computed effective address. Each data cache effective address comprises a `tag` field and a `line` field. When identifying the line, `initda` uses both the `tag` field and the `line` field. • Compare the cache line tag with the effective address to determine if the addressed data is currently cached. If the `tag` fields do not match, the effective address is not currently cached, so the instruction does nothing. • Skip checking if the data cache line is dirty. Because `initd` skips the dirty cache line check, data that has been modified by the processor, but not yet written to memory is lost. • Clear the valid bit for the line. If the Nios II processor core does not have a data cache, the `initda` instruction performs no operation.
Usage	Use `initda` to skip writing dirty lines back to memory and to flush the cache line only if the addressed memory location is currently in the cache. By contrast, refer to "flushd flush data cache line", "flushda flush data cache address", and "initd initialize data cache line" on page 8–55 for other cache-clearing options. Use `initda` with caution because it does not write back dirty data. For more information on the Nios II data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
Exceptions	Supervisor-only data address Fast TLB miss (data) Double TLB miss (data) MPU region violation (data) Unimplemented instruction

Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x13					

Related Information

- **Cache and Tightly-Coupled Memory**
- **flushda** on page 41
- **initd** on page 44
- **flushd** on page 39

initi

Instruction	initialize instruction cache line
Operation	Initializes the instruction cache line associated with address rA.
Assembler Syntax	`initi rA`
Example	`initi r6`
Description	Ignoring the tag, `initi` identifies the instruction cache line associated with the byte address in `ra`, and `initi` invalidates that line. If the Nios II processor core does not have an instruction cache, the `initi` instruction performs no operation.
Usage	This instruction is used to initialize the processor's instruction cache. Immediately after processor reset, use `initi` to invalidate each line of the instruction cache. For more information on instruction cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
Exceptions	Supervisor-only instruction

Instruction Type	R
Instruction Fields	A = Register index of operand rA

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					0					0x29
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x29					0					0x3a					

Related Information

Cache and Tightly-Coupled Memory

jmp

Instruction	computed jump
Operation	PC ← rA
Assembler Syntax	`jmp rA`
Example	`jmp r12`
Description	Transfers execution to the address contained in register rA.
Usage	It is illegal to jump to the address contained in register r31. To return from subroutines called by `call` or `callr`, use `ret` instead of `jmp`.
Exceptions	Misaligned destination address
Instruction Type	R
Instruction Fields	A = Register index of operand rA

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					0					0x0d
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0d					0					0x3a					

jmpi

Instruction	jump immediate

Operation	PC ← ($PC_{31..28}$: IMM26 x 4)
Assembler Syntax	`jmpi label`
Example	`jmpi write_char`
Description	Transfers execution to the instruction at address ($PC_{31..28}$: IMM26 x 4).
Usage	`jmpi` is a low-overhead local jump. `jmpi` can transfer execution anywhere within the 256-MB range determined by $PC_{31..28}$. The Nios II GNU linker does not automatically handle cases in which the address is out of this range.
Exceptions	None
Instruction Type	J
Instruction Fields	`IMM26` = 26-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IMM26															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM26										0x01					

Idb / Idbio

Instruction	load byte from memory or I/O peripheral
Operation	rB ← σ(Mem8[rA + σ(IMM16)])
Assembler Syntax	`ldb rB, byte_offset(rA)` `ldbio rB, byte_offset(rA)`
Example	`ldb r6, 100(r5)`
Description	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the desired memory byte, sign extending the 8-bit value to 32 bits. In Nios II processor cores with a data cache, this instruction may retrieve the desired data from the cache instead of from memory.

Usage	Use the `ldbio` instruction for peripheral I/O. In processors with a data cache, `ldbio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldbio` acts like `ldb`. For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
Exceptions	Supervisor-only data address Misaligned data address TLB permission violation (read) Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Table 9: ldb

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x07					

Table 10: ldbio

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x27					

Related Information

Cache and Tightly-Coupled Memory

ldbu / ldbuio

Instruction	load unsigned byte from memory or I/O peripheral
Operation	rB ← 0x000000 : Mem8[rA + σ(IMM16)]
Assembler Syntax	`ldbu rB, byte_offset(rA)` `ldbuio rB, byte_offset(rA)`
Example	`ldbu r6, 100(r5)`
Description	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the desired memory byte, zero extending the 8-bit value to 32 bits.
Usage	In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldbuio` instruction for peripheral I/O. In processors with a data cache, `ldbuio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldbuio` acts like `ldbu`. For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
Exceptions	Supervisor-only data address Misaligned data address TLB permission violation (read) Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Table 11: ldbu

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					

Bit Fields															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x03					

Table 12: ldbuio

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x23					

Related Information

Cache and Tightly-Coupled Memory

ldh / ldhio

Instruction	load halfword from memory or I/O peripheral
Operation	rB ← σ(Mem16[rA + σ(IMM16)])
Assembler Syntax	`ldh rB, byte_offset(rA)` `ldhio rB, byte_offset(rA)`
Example	`ldh r6, 100(r5)`
Description	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the memory halfword located at the effective byte address, sign extending the 16-bit value to 32 bits. The effective byte address must be halfword aligned. If the byte address is not a multiple of 2, the operation is undefined.
Usage	In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldhio` instruction for peripheral I/O. In processors with a data cache, `ldhio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldhio` acts like `ldh`. For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Exceptions	Supervisor-only data address Misaligned data address TLB permission violation (read) Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Table 13: ldh

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x0f					

Table 14: ldhio

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x2f					

Related Information

Cache and Tightly-Coupled Memory

ldhu / ldhuio

Instruction	load unsigned halfword from memory or I/O peripheral
Operation	rB ← 0x0000 : Mem16[rA + σ(IMM16)]
Assembler Syntax	`ldhu rB, byte_offset(rA)` `ldhuio rB, byte_offset(rA)`

Example	`ldhu r6, 100(r5)`
Description	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the memory halfword located at the effective byte address, zero extending the 16-bit value to 32 bits. The effective byte address must be halfword aligned. If the byte address is not a multiple of 2, the operation is undefined.
Usage	In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldhuio` instruction for peripheral I/O. In processors with a data cache, `ldhuio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldhuio` acts like `ldhu`. For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
Exceptions	Supervisor-only data address Misaligned data address TLB permission violation (read) Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Table 15: ldhu

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x0b					

Table 16: ldhuio

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x2b					

Related Information

Cache and Tightly-Coupled Memory

ldw / ldwio

Instruction	load 32-bit word from memory or I/O peripheral
Operation	rB ← Mem32[rA + σ(IMM14)]
Assembler Syntax	`ldw rB, byte_offset(rA)` `ldwio rB, byte_offset(rA)`
Example	`ldw r6, 100(r5)`
Description	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the memory word located at the effective byte address. The effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.
Usage	In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldwio` instruction for peripheral I/O. In processors with a data cache, `ldwio` bypasses the cache and memory. Use the `ldwio` instruction for peripheral I/O. In processors with a data cache, `ldwio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldwio` acts like `ldw`. For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Exceptions	Supervisor-only data address Misaligned data address TLB permission violation (read) Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Table 17: ldw

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x17					

Table 18: ldwio

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x37					

Related Information

Cache and Tightly-Coupled Memory

mov

Instruction	move register to register
Operation	rC ← rA
Assembler Syntax	`mov rC, rA`
Example	`mov r6, r7`

Description	Moves the contents of rA to rC.
Pseudo-instruction	`mov` is implemented as `add rC, rA, r0`.

movhi

Instruction	move immediate into high halfword
Operation	rB ← (IMMED : 0x0000)
Assembler Syntax	`movhi rB, IMMED`
Example	`movhi r6, 0x8000`
Description	Writes the immediate value IMMED into the high halfword of rB, and clears the lower halfword of rB to 0x0000.

Usage	The maximum allowed value of IMMED is 65535. The minimum allowed value is 0. To load a 32-bit constant into a register, first load the upper 16 bits using a `movhi` pseudo-instruction. The `%hi()` macro can be used to extract the upper 16 bits of a constant or a label. Then, load the lower 16 bits with an `ori` instruction. The `%lo()` macro can be used to extract the lower 16 bits of a constant or label as shown in the following code:
	`movhi rB, %hi(value)`
	`ori rB, rB, %lo(value)`
	An alternative method to load a 32-bit constant into a register uses the %hiadj() macro and the `addi` instruction as shown in the following code:
	`movhi rB, %hiadj(value)`
	`addi rB, rB, %lo(value)`

Pseudo-instruction	`movhi` is implemented as `orhi rB, r0, IMMED`.

movi

Instruction	move signed immediate into word
Operation	rB ← σ(IMMED)
Assembler Syntax	`movi rB, IMMED`
Example	`movi r6, -30`

Description	Sign-extends the immediate value IMMED to 32 bits and writes it to rB.
Usage	The maximum allowed value of IMMED is 32767. The minimum allowed value is –32768. To load a 32-bit constant into a register, refer to the `movhi` instruction.
Pseudo-instruction	`movi` is implemented as `addi rB, r0, IMMED`.

movia

Instruction	move immediate address into word
Operation	rB ← label
Assembler Syntax	`movia rB, label`
Example	`movia r6, function_address`
Description	Writes the address of label to rB.

Pseudo-instruction	`movia` is implemented as:
	`orhi rB, r0, %hiadj(label)`
	`addi rB, rB, %lo(label)`

movui

Instruction	move unsigned immediate into word
Operation	rB ← (0x0000 : IMMED)
Assembler Syntax	`movui rB, IMMED`
Example	`movui r6, 100`
Description	Zero-extends the immediate value IMMED to 32 bits and writes it to rB.
Usage	The maximum allowed value of IMMED is 65535. The minimum allowed value is 0. To load a 32-bit constant into a register, refer to the `movhi` instruction.
Pseudo-instruction	`movui` is implemented as `ori rB, r0, IMMED`.

mul

Instruction	multiply
Operation	rC ← (rA x rB) $_{31..0}$
Assembler Syntax	`mul rC, rA, rB`
Example	`mul r6, r7, r8`
Description	Multiplies rA times rB and stores the 32 low-order bits of the product to rC. The result is the same whether the operands are treated as signed or unsigned integers. Nios II processors that do not implement the `mul` instruction cause an unimplemented instruction exception.

Usage	Carry Detection (unsigned operands):
	Before or after the multiply operation, the carry out of the MSB of rC can be detected using the following instruction sequence:
	`mul rC, rA, rB`
	`mulxuu rD, rA, rB`
	`cmpne rD, rD, r0`
	`# The mul operation (optional)`
	`# rD is nonzero if carry occurred`
	`# rD is 1 if carry occurred, 0 if not`
	The `mulxuu` instruction writes a nonzero value into rD if the multiplication of unsigned numbers generates a carry (unsigned overflow). If a 0/1 result is desired, follow the `mulxuu` with the `cmpne` instruction.
	Overflow Detection (signed operands):
	After the multiply operation, overflow can be detected using the following instruction sequence:
	`mul rC, rA, rB`
	`cmplt rD, rC, r0`
	`mulxss rE, rA, rB`
	`add rD, rD, rE`
	`cmpne rD, rD, r0`
	`# The original mul operation`
	`# rD is nonzero if overflow`
	`# rD is 1 if overflow, 0 if not`
	The `cmplt-mulxss-add` instruction sequence writes a nonzero value into rD if the product in rC cannot be represented in 32 bits (signed overflow). If a 0/1 result is desired, follow the instruction sequence with the `cmpne` instruction.

Exceptions	Unimplemented instruction
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x27
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x27					0					0x3a					

muli

Instruction	multiply immediate
Operation	rB ← (rA x σ(IMM16)) $_{31..0}$
Assembler Syntax	`muli rB, rA, IMM16`
Example	`muli r6, r7, -100`
Description	Sign-extends the 16-bit immediate value IMM16 to 32 bits and multiplies it by the value of rA. Stores the 32 low-order bits of the product to rB. The result is independent of whether rA is treated as a signed or unsigned number. Nios II processors that do not implement the `muli` instruction cause an unimplemented instruction exception. Carry Detection and Overflow Detection: For a discussion of carry and overflow detection, refer to the `mul` instruction.
Exceptions	Unimplemented instruction
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x24					

mulxss

Instruction	multiply extended signed/signed
Operation	rC ← ((signed) rA) x ((signed) rB)) $_{63..32}$
Assembler Syntax	`mulxss rC, rA, rB`
Example	`mulxss r6, r7, r8`
Description	Treating rA and rB as signed integers, `mulxss` multiplies rA times rB, and stores the 32 high-order bits of the product to rC. Nios II processors that do not implement the `mulxss` instruction cause an unimplemented instruction exception.
Usage	Use `mulxss` and `mul` to compute the full 64-bit product of two 32-bit signed integers. Furthermore, `mulxss` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit integers, each contained in a pair of 32-bit registers, (S1 : U1) and (S2 : U2), their 128-bit product is (U1 x U2) + ((S1 x U2) << 32) + ((U1 x S2) << 32) + ((S1 x S2) << 64). The `mulxss` and `mul` instructions are used to calculate the 64-bit product S1 x S2.
Exceptions	Unimplemented instruction
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x1f
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1f					0					0x3a					

mulxsu

Instruction	multiply extended signed/unsigned
Operation	rC ← ((signed) rA) x ((unsigned) rB)) $_{63..32}$

Assembler Syntax	`mulxsu rC, rA, rB`
Example	`mulxsu r6, r7, r8`
Description	Treating rA as a signed integer and rB as an unsigned integer, `mulxsu` multiplies rA times rB, and stores the 32 high-order bits of the product to rC. Nios II processors that do not implement the `mulxsu` instruction cause an unimplemented instruction exception.
Usage	`mulxsu` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit integers, each contained in a pair of 32-bit registers, (S1 : U1) and (S2 : U2), their 128-bit product is: (U1 x U2) + ((S1 x U2) << 32) + ((U1 x S2) << 32) + ((S1 x S2) << 64). The `mulxsu` and `mul` instructions are used to calculate the two 64-bit products S1 x U2 and U1 x S2.
Exceptions	Unimplemented instruction
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x17
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x17					0					0x3a					

mulxuu

Instruction	multiply extended unsigned/unsigned
Operation	rC ← ((unsigned) rA) x ((unsigned) rB)) $_{63..32}$
Assembler Syntax	`mulxuu rC, rA, rB`
Example	`mulxuu r6, r7, r8`

Description	Treating rA and rB as unsigned integers, `mulxuu` multiplies rA times rB and stores the 32 high-order bits of the product to rC. Nios II processors that do not implement the `mulxuu` instruction cause an unimplemented instruction exception.
Usage	Use `mulxuu` and `mul` to compute the 64-bit product of two 32-bit unsigned integers. Furthermore, `mulxuu` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit signed integers, each contained in a pair of 32-bit registers, (S1 : U1) and (S2 : U2), their 128-bit product is (U1 x U2) + ((S1 x U2) << 32) + ((U1 x S2) << 32) + ((S1 x S2) << 64). The `mulxuu` and `mul` instructions are used to calculate the 64-bit product U1 x U2. `mulxuu` also can be used as part of the calculation of a 128-bit product of two 64-bit unsigned integers. Given two 64-bit unsigned integers, each contained in a pair of 32-bit registers, (T1 : U1) and (T2 : U2), their 128-bit product is (U1 x U2) + ((U1 x T2) << 32) + ((T1 x U2) << 32) + ((T1 x T2) << 64). The `mulxuu` and `mul` instructions are used to calculate the four 64-bit products U1 x U2, U1 x T2, T1 x U2, and T1 x T2.
Exceptions	Unimplemented instruction
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x07
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x07					0					0x3a					

nextpc

Instruction	get address of following instruction
Operation	rC ← PC + 4

Assembler Syntax	`nextpc rC`
Example	`nextpc r6`
Description	Stores the address of the next instruction to register rC.
Usage	A relocatable code fragment can use `nextpc` to calculate the address of its data segment. `nextpc` is the only way to access the PC directly.
Exceptions	None
Instruction Type	R
Instruction Fields	`C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0					0					C					0x1c
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1c					0					0x3a					

nop

Instruction	no operation
Operation	None
Assembler Syntax	`nop`
Example	`nop`
Description	`nop` does nothing.
Pseudo-instruction	`nop` is implemented as `add r0, r0, r0`.

nor

Instruction	bitwise logical nor
Operation	rC ← ~(rA | rB)
Assembler Syntax	`nor rC, rA, rB`
Example	`nor r6, r7, r8`

Description	Calculates the bitwise logical NOR of rA and rB and stores the result in rC.
Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	A					B					C					0x06						0					0x3a					

or

Instruction	bitwise logical or
Operation	rC ← rA | rB
Assembler Syntax	`or rC, rA, rB`
Example	`or r6, r7, r8`
Description	Calculates the bitwise logical OR of rA and rB and stores the result in rC.
Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x16					0					0x3a					

orhi

Instruction	bitwise logical or immediate into high halfword

Operation	rB ← rA | (IMM16 : 0x0000)
Assembler Syntax	`orhi rB, rA, IMM16`
Example	`orhi r6, r7, 100`
Description	Calculates the bitwise logical OR of rA and (IMM16 : 0x0000) and stores the result in rB.
Exceptions	None
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x34					

ori

Instruction	bitwise logical or immediate
Operation	rB ← rA | (0x0000 : IMM16)
Assembler Syntax	`ori rB, rA, IMM16`
Example	`ori r6, r7, 100`
Description	Calculates the bitwise logical OR of rA and (0x0000 : IMM16) and stores the result in rB.
Exceptions	None
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x14					

rdctl

Instruction	read from control register
Operation	rC ← ctlN
Assembler Syntax	`rdctl rC, ctlN`
Example	`rdctl r3, ctl31`
Description	Reads the value contained in control register ctlN and writes it to register rC.
Exceptions	Supervisor-only instruction
Instruction Type	R
Instruction Fields	`C` = Register index of operand rC `N` = Control register index of operand ctlN

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0					0					C					0x26
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x26					N					0x3a					

rdprs

Instruction	read from previous register set
Operation	rB ← prs.rA + σ(IMM16)
Assembler Syntax	`rdprs rB, rA, IMM16`
Example	`rdprs r6, r7, 0`

Description	Sign-extends the 16-bit immediate value IMM16 to 32 bits, and adds it to the value of rA from the previous register set. Places the result in rB in the current register set.
Usage	The previous register set is specified by status.PRS. By default, status.PRS indicates the register set in use before an exception, such as an external interrupt, caused a register set change. To read from an arbitrary register set, software can insert the desired register set number in `status.PRS` prior to executing `rdprs`. If shadow register sets are not implemented on the Nios II core, `rdprs` is an illegal instruction.
Exceptions	Supervisor-only instruction Illegal instruction
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x38					

ret

Instruction	return from subroutine
Operation	PC ← `ra`
Assembler Syntax	`ret`
Example	`ret`
Description	Transfers execution to the address in `ra`.
Usage	Any subroutine called by `call` or `callr` must use `ret` to return.

Exceptions	Misaligned destination address
Instruction Type	R
Instruction Fields	None

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x1f					0					0					0x05
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x05					0					0x3a					

rol

Instruction	rotate left
Operation	rC ← rA rotated left $rB_{4..0}$ bit positions
Assembler Syntax	`rol rC, rA, rB`
Example	`rol r6, r7, r8`
Description	Rotates rA left by the number of bits specified in $rB_{4..0}$ and stores the result in rC. The bits that shift out of the register rotate into the least-significant bit positions. Bits 31–5 of rB are ignored.
Exceptions	None
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x03
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x03					0					0x3a					

roli

Instruction	rotate left immediate

Operation	rC ← rA rotated left IMM5 bit positions
Assembler Syntax	`roli rC, rA, IMM5`
Example	`roli r6, r7, 3`
Description	Rotates rA left by the number of bits specified in IMM5 and stores the result in rC. The bits that shift out of the register rotate into the least-significant bit positions.
Usage	In addition to the rotate-left operation, roli can be used to implement a rotate-right operation. Rotating left by (32 – IMM5) bits is the equivalent of rotating right by IMM5 bits.
Exceptions	None
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `C` = Register index of operand rC `IMM5` = 5-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					C					0x02
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x02					IMM5					0x3a					

ror

Instruction	rotate right
Operation	rC ← rA rotated right $rB_{4..0}$ bit positions
Assembler Syntax	`ror rC, rA, rB`
Example	`ror r6, r7, r8`
Description	Rotates rA right by the number of bits specified in $rB_{4..0}$ and stores the result in rC. The bits that shift out of the register rotate into the most-significant bit positions. Bits 31– 5 of rB are ignored.
Exceptions	None

Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x0b
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0b					0					0x3a					

sll

Instruction	shift left logical
Operation	rC ← rA << ($rB_{4..0}$)
Assembler Syntax	`sll rC, rA, rB`
Example	`sll r6, r7, r8`
Description	Shifts rA left by the number of bits specified in $rB_{4..0}$ (inserting zeroes), and then stores the result in rC. `sll` performs the << operation of the C programming language.
Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x13
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x13					0					0x3a					

slli

Instruction	shift left logical immediate
Operation	rC ← rA << IMM5
Assembler Syntax	`slli rC, rA, IMM5`
Example	`slli r6, r7, 3`
Description	Shifts rA left by the number of bits specified in IMM5 (inserting zeroes), and then stores the result in rC.
Usage	`slli` performs the << operation of the C programming language.
Exceptions	None
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `C` = Register index of operand rC `IMM5` = 5-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					C					0x12
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x12					IMM5					0x3a					

sra

Instruction	shift right arithmetic
Operation	rC ← (signed) rA >> ((unsigned) $rB_{4..0}$)
Assembler Syntax	`sra rC, rA, rB`
Example	`sra r6, r7, r8`
Description	Shifts rA right by the number of bits specified in $rB_{4..0}$ (duplicating the sign bit), and then stores the result in rC. Bits 31–5 are ignored.
Usage	`sra` performs the signed >> operation of the C programming language.

Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x3b
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3b					0					0x3a					

srai

Instruction	shift right arithmetic immediate
Operation	rC ← (signed) rA >> ((unsigned) IMM5)
Assembler Syntax	`srai rC, rA, IMM5`
Example	`srai r6, r7, 3`
Description	Shifts rA right by the number of bits specified in IMM5 (duplicating the sign bit), and then stores the result in rC.
Usage	`srai` performs the signed >> operation of the C programming language.
Exceptions	None
Instruction Type	R
Instruction Fields	A = Register index of operand rA C = Register index of operand rC IMM5 = 5-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					C					0x3a
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bit Fields		
0x3a	IMM5	0x3a

srl

Instruction	shift right logical
Operation	rC ← (unsigned) rA >> ((unsigned) $rB_{4..0}$)
Assembler Syntax	`srl rC, rA, rB`
Example	`srl r6, r7, r8`
Description	Shifts rA right by the number of bits specified in $rB_{4..0}$ (inserting zeroes), and then stores the result in rC. Bits 31–5 are ignored.
Usage	`srl` performs the unsigned >> operation of the C programming language.
Exceptions	None
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x1b
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1b					0					0x3a					

srli

Instruction	shift right logical immediate
Operation	rC ← (unsigned) rA >> ((unsigned) IMM5)
Assembler Syntax	`srli rC, rA, IMM5`
Example	`srli r6, r7, 3`

Description	Shifts rA right by the number of bits specified in IMM5 (inserting zeroes), and then stores the result in rC.
Usage	`srli` performs the unsigned >> operation of the C programming language.
Exceptions	None
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `C` = Register index of operand rC `IMM5` = 5-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x1a
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1a					IMM5					0x3a					

stb / stbio I

Instruction	store byte to memory or I/O periphera
Operation	Mem8[rA + σ(IMM16)] ← $rB_{7..0}$
Assembler Syntax	`stb rB, byte_offset(rA)` `stbio rB, byte_offset(rA)`
Example	`stb r6, 100(r5)`
Description	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Stores the low byte of rB to the memory byte specified by the effective address.
Usage	In processors with a data cache, this instruction may not generate an Avalon-MM bus cycle to noncache data memory immediately. Use the `stbio` instruction for peripheral I/O. In processors with a data cache, `stbio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `stbio` acts like `stb`.

Exceptions	Supervisor-only data address Misaligned data address TLB permission violation (write) Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Table 19: stb

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x05					

Table 20: stbio

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x25					

sth / sthio

Instruction	store halfword to memory or I/O peripheral
Operation	Mem16[rA + σ(IMM16)] ← $rB_{15..0}$
Assembler Syntax	`sth rB, byte_offset(rA)` `sthio rB, byte_offset(rA)`
Example	`sth r6, 100(r5)`

Description	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Stores the low halfword of rB to the memory location specified by the effective byte address. The effective byte address must be halfword aligned. If the byte address is not a multiple of 2, the operation is undefined.
Usage	In processors with a data cache, this instruction may not generate an Avalon-MM data transfer immediately. Use the sthio instruction for peripheral I/O. In processors with a data cache, `sthio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `sthio` acts like `sth`.
Exceptions	Supervisor-only data address Misaligned data address TLB permission violation (write) Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Table 21: sth

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x0d					

Table 22: sthio

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bit Fields	
IMM16	0x2d

stw / stwio

Instruction	store word to memory or I/O peripheral
Operation	Mem32[rA + σ(IMM16)] ← rB
Assembler Syntax	`stw rB, byte_offset(rA)` `stwio rB, byte_offset(rA)`
Example	`stw r6, 100(r5)`
Description	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Stores rB to the memory location specified by the effective byte address. The effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.
Usage	In processors with a data cache, this instruction may not generate an Avalon-MM data transfer immediately. Use the `stwio` instruction for peripheral I/O. In processors with a data cache, `stwio` bypasses the cache and is guaranteed to generate an Avalon-MM bus cycle. In processors without a data cache, `stwio` acts like `stw`.
Exceptions	Supervisor-only data address Misaligned data address TLB permission violation (write) Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

Table 23: stw

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x15					

Table 24: stwio

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x35					

sub

Instruction	subtract
Operation	rC ← rA – rB
Assembler Syntax	`sub rC, rA, rB`
Example	`sub r6, r7, r8`
Description	Subtract rB from rA and store the result in rC.

Usage	Carry Detection (unsigned operands):
	The carry bit indicates an unsigned overflow. Before or after a sub operation, a carry out of the MSB can be detected by checking whether the first operand is less than the second operand. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. Both cases are shown in the following code:
	`sub rC, rA, rB`
	`cmpltu rD, rA, rB`
	`sub rC, rA, rB`
	`bltu rA, rB, label`
	`# The original sub operation (optional)`
	`# rD is written with the carry bit`
	`# The original sub operation (optional)`
	`# Branch if carry generated`
	Overflow Detection (signed operands):
	Detect overflow of signed subtraction by comparing the sign of the difference that is written to rC with the signs of the operands. If rA and rB have different signs, and the sign of rC is different than the sign of rA, an overflow occurred. The overflow condition can control a conditional branch, as shown in the following code:
	`sub rC, rA, rB`
	`xor rD, rA, rB`
	`xor rE, rA, rC`
	`and rD, rD, rE`
	`blt rD, r0, label`
	`# The original sub operation`
	`# Compare signs of rA and rB`
	`# Compare signs of rA and rC`
	`# Combine comparisons`
	`# Branch if overflow occurred`

Exceptions	None
Instruction Type	R

Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x39
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x39					0					0x3a					

subi

Instruction	subtract immediate
Operation	rB ← rA – σ(IMMED)
Assembler Syntax	`subi rB, rA, IMMED`
Example	`subi r8, r8, 4`
Description	Sign-extends the immediate value IMMED to 32 bits, subtracts it from the value of rA and then stores the result in rB.
Usage	The maximum allowed value of IMMED is 32768. The minimum allowed value is –32767.
Pseudo-instruction	`subi` is implemented as `addi rB, rA, -IMMED`

sync

Instruction	memory synchronization
Operation	None
Assembler Syntax	`sync`
Example	`sync`
Description	Forces all pending memory accesses to complete before allowing execution of subsequent instructions. In processor cores that support in-order memory accesses only, this instruction performs no operation.

Exceptions	None
Instruction Type	R
Instruction Fields	None

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0					0					0					0x36
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x36					0					0x3a					

trap

Instruction	trap
Operation	`estatus` ← `status` PIE ← 0 U ← 0 `ea` ← PC + 4 PC ← exception handler address
Assembler Syntax	`trap` `trap imm5`
Example	`trap`
Description	Saves the address of the next instruction in register `ea`, saves the contents of the `status` register in `estatus`, disables interrupts, and transfers execution to the exception handler. The address of the exception handler is specified with the Nios_II Processor parameter editor in Qsys. The 5-bit immediate field `imm5` is ignored by the processor, but it can be used by the debugger. `trap` with no argument is the same as `trap 0`.
Usage	To return from the exception handler, execute an `eret` instruction.
Exceptions	Trap
Instruction Type	R

Instruction Fields	`IMM5` = Type of breakpoint

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0					0					0x1d					0x2d
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2d					IMM5					0x3a					

wrctl

Instruction	write to control register
Operation	ctlN ← rA
Assembler Syntax	`wrctl ctlN, rA`
Example	`wrctl ctl6, r3`
Description	Writes the value contained in register rA to the control register ctlN.
Exceptions	Supervisor-only instruction
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `N` = Control register index of operand ctlN

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					0					0x2e
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2e					N					0x3a					

wrprs

Instruction	write to previous register set
Operation	prs.rC ← rA
Assembler Syntax	`wrprs rC, rA`
Example	`wrprs r6, r7`

Description	Copies the value of rA in the current register set to rC in the previous register set. This instruction can set r0 to 0 in a shadow register set.
Usage	The previous register set is specified by status.PRS. By default, status.PRS indicates the register set in use before an exception, such as an external interrupt, caused a register set change. To write to an arbitrary register set, software can insert the desired register set number in `status.PRS` prior to executing `wrprs`. System software must use `wrprs` to initialize `r0` to 0 in each shadow register set before using that register set. If shadow register sets are not implemented on the Nios II core, `wrprs` is an illegal instruction.
Exceptions	Supervisor-only instruction Illegal instruction
Instruction Type	R
Instruction Fields	`A` = Register index of operand rA `C` = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					0					C					0x14
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x14					0					0x3a					

xor

Instruction	bitwise logical exclusive or
Operation	rC ← rA ^ rB
Assembler Syntax	`xor rC, rA, rB`
Example	`xor r6, r7, r8`
Description	Calculates the bitwise logical exclusive-or of rA and rB and stores the result in rC.
Exceptions	None

Instruction Type	R
Instruction Fields	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					C					0x1e
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1e					0					0x3a					

xorhi

Instruction	bitwise logical exclusive or immediate into high halfword
Operation	rB ← rA ^ (IMM16 : 0x0000)
Assembler Syntax	`xorhi rB, rA, IMM16`
Example	`xorhi r6, r7, 100`
Description	Calculates the bitwise logical exclusive XOR of rA and (IMM16 : 0x0000) and stores the result in rB.
Exceptions	None
Instruction Type	I
Instruction Fields	A = Register index of operand rA B = Register index of operand rB IMM16 = 16-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x3c					

xori

Instruction	bitwise logical exclusive or immediate

Operation	rB ← rA ^ (0x0000 : IMM16)
Assembler Syntax	`xori rB, rA, IMM16`
Example	`xori r6, r7, 100`
Description	Calculates the bitwise logical exclusive OR of rA and (0x0000 : IMM16) and stores the result in rB.
Exceptions	None
Instruction Type	I
Instruction Fields	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit unsigned immediate value

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A					B					IMM16					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM16										0x1c					

Document Revision History

Table 25: Document Revision History

Date	Version	Changes
April 2015	2015.04.02	Maintenance release.
February 2014	13.1.0	Removed references to SOPC Builder.
May 2011	11.0.0	Maintenance release.
December 2010	10.1.0	Corrected comments delimiter (#) in instruction usage.
July 2010	10.0.0	Corrected typographical error in `cmpgei` instruction type.
November 2009	9.1.0	Added shadow register sets and external interrupt controller support, including `rdprs` and `wrprs` instructions.
March 2009	9.0.0	Backwards-compatible change to the `eret` instruction B field encoding.
November 2008	8.1.0	Maintenance release.

Date	Version	Changes
May 2008	8.0.0	• Added MMU. • Added an Exceptions section to all instructions.
October 2007	7.2.0	Added `jmpi` instruction.
May 2007	7.1.0	• Added table of contents to Introduction section. • Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Maintenance release.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	• Correction to the `blt` instruction. • Added U bit operation for `break` and `trap` instructions.
July 2005	5.0.1	• Added new `flushda` instruction. • Updated `flushd` instruction. • Instruction Opcode table updated with `flushda` instruction.
May 2005	5.0.0	Maintenance release.
December 2004	1.2	• `break` instruction update. • `srli` instruction correction.
September 2004	1.1	Updates for Nios II 1.01 release.
May 2004	1.0	Initial release.