Introduction

This section introduces the Nios® II instruction-word format and provides a detailed reference of the Nios II instruction set. This chapter contains the following sections:

- “Word Formats” on page 8–1
- “Instruction Opcodes” on page 8–2
- “Assembler Pseudo-Instructions” on page 8–3
- “Assembler Macros” on page 8–4
- “Instruction Set Reference” on page 8–4

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 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.

The I-type instruction format is:

```
A B 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. Some R-Type instructions embed a small immediate value in the low-order bits of OPX.

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

The R-type instruction format is:

| 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  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 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 MByte range.

The J-type instruction format is:

| 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| IMMED26 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| OP |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Instruction Opcodes**

The OP field in the Nios II instruction word specifies the major class of an opcode as shown in Table 8–1 and Table 8–2. 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 8–1. OP Encodings (Part 1 of 2)**

<table>
<thead>
<tr>
<th>OP</th>
<th>Instruction</th>
<th>OP</th>
<th>Instruction</th>
<th>OP</th>
<th>Instruction</th>
<th>OP</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>call</td>
<td>0x010</td>
<td>cmplti</td>
<td>0x020</td>
<td>cmpee</td>
<td>0x030</td>
<td>cmpltu</td>
</tr>
<tr>
<td>0x01</td>
<td>jmpi</td>
<td>0x011</td>
<td>cmplti</td>
<td>0x021</td>
<td>cmpee</td>
<td>0x031</td>
<td>custom</td>
</tr>
<tr>
<td>0x02</td>
<td>stb</td>
<td>0x012</td>
<td>cmplt</td>
<td>0x022</td>
<td>cmpee</td>
<td>0x032</td>
<td>custom</td>
</tr>
<tr>
<td>0x03</td>
<td>ldbu</td>
<td>0x013</td>
<td>cmpltu</td>
<td>0x023</td>
<td>cmpee</td>
<td>0x033</td>
<td>custom</td>
</tr>
<tr>
<td>0x04</td>
<td>addi</td>
<td>0x014</td>
<td>initda</td>
<td>0x024</td>
<td>cmpee</td>
<td>0x034</td>
<td>cmpltu</td>
</tr>
<tr>
<td>0x05</td>
<td>stb</td>
<td>0x015</td>
<td>or</td>
<td>0x025</td>
<td>cmpee</td>
<td>0x035</td>
<td>cmpltu</td>
</tr>
<tr>
<td>0x06</td>
<td>br</td>
<td>0x016</td>
<td>or</td>
<td>0x026</td>
<td>cmpee</td>
<td>0x036</td>
<td>cmpltu</td>
</tr>
<tr>
<td>0x07</td>
<td>ldb</td>
<td>0x017</td>
<td>stw</td>
<td>0x027</td>
<td>cmpee</td>
<td>0x037</td>
<td>cmpltu</td>
</tr>
<tr>
<td>0x08</td>
<td>cmpgei</td>
<td>0x018</td>
<td>stbio</td>
<td>0x028</td>
<td>cmpee</td>
<td>0x038</td>
<td>cmpltu</td>
</tr>
<tr>
<td>0x09</td>
<td>cmpgei</td>
<td>0x019</td>
<td>stwio</td>
<td>0x029</td>
<td>cmpee</td>
<td>0x039</td>
<td>cmpltu</td>
</tr>
<tr>
<td>0x0A</td>
<td>cmpgei</td>
<td>0x01A</td>
<td>stwio</td>
<td>0x02A</td>
<td>cmpee</td>
<td>0x03A</td>
<td>cmpltu</td>
</tr>
<tr>
<td></td>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 8–3. Assembler Pseudo-Instructions (Part 1 of 2)

<table>
<thead>
<tr>
<th>Pseudo-Instruction</th>
<th>Equivalent Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgt rA, rB, label</td>
<td>blt rB, rA, label</td>
</tr>
<tr>
<td>bgtu rA, rB, label</td>
<td>bltu rB, rA, label</td>
</tr>
<tr>
<td>ble rA, rB, label</td>
<td>bge rB, rA, label</td>
</tr>
<tr>
<td>bleu rA, rB, label</td>
<td>bgeu rB, rA, label</td>
</tr>
<tr>
<td>cmpgt rC, rA, rB</td>
<td>cmplt rC, rB, rA</td>
</tr>
</tbody>
</table>
Assembler Macros

The Nios II assembler provides macros to extract halfwords from labels and from 32-bit immediate values. Table 8–4 lists the available macros. 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 8–4. Assembler Macros

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>%lo(imm32)</td>
<td>Extract bits [15..0] of imm32</td>
<td>imm32 &amp; 0xffff</td>
</tr>
<tr>
<td>%hi(imm32)</td>
<td>Extract bits [31..16] of imm32</td>
<td>(imm32 &gt;&gt; 16) &amp; 0xffff</td>
</tr>
<tr>
<td>%hiadj(imm32)</td>
<td>Extract bits [31..16] and adds bit 15 of imm32</td>
<td>((imm32 &gt;&gt; 16) &amp; 0xffff) + ((imm32 &gt;&gt; 15) &amp; 0x1)</td>
</tr>
<tr>
<td>%gprel(imm32)</td>
<td>Replace the imm32 address with an offset from the global pointer (1)</td>
<td>imm32 – _gp</td>
</tr>
</tbody>
</table>

Note to Table 8–4:
(1) Refer to the Application Binary Interface chapter of the Nios II Processor Reference Handbook for more information about global pointers.

Instruction Set Reference

The following pages list all Nios II instruction mnemonics in alphabetical order. Table 8–5 shows the notation conventions used to describe instruction operation.
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 details on these and all Nios II exceptions, refer to the Programming Model chapter of the Nios II Processor Reference Handbook.
add

Operation:

\[ r_C \leftarrow r_A + r_B \]

Assembler Syntax:

\texttt{add rC, rA, rB}

Example:

\texttt{add r6, r7, r8}

Description:
Calculates the sum of \( r_A \) and \( r_B \). Stores the result in \( r_C \). Used for both signed and unsigned addition.

Usage:

Carry Detection (unsigned operands):
Following an \texttt{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. Both cases are shown below.

\begin{verbatim}
add rC, rA, rB ; The original add operation
cmpltu rD, rC, rA ; rD is written with the carry bit
add rC, rA, rB ; The original add operation
bltu rC, rA, label ; Branch if carry was generated
\end{verbatim}

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 below.

\begin{verbatim}
add rC, rA, rB ; The original add operation
xor rD, rC, rA ; Compare signs of sum and \( r_A \)
xor rE, rC, rB ; Compare signs of sum and \( r_B \)
and rD, rD, rE ; Combine comparisons
blt rD, r0, label ; Branch if overflow occurred
\end{verbatim}

Exceptions:
None

Instruction Type:
\texttt{R}

Instruction Fields:
- \( A = \) Register index of operand \( r_A \)
- \( B = \) Register index of operand \( r_B \)
- \( C = \) Register index of operand \( r_C \)

\begin{verbatim}
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
\hline
A | B | C | 0x31 | 0 | 0x3a
\end{verbatim}
**addi (add immediate)**

**Operation:**
\[ rB \leftarrow rA + \sigma(\text{IMM16}) \]

**Assembler Syntax:**
\[
\text{addi } rB, rA, \text{ IMM16}
\]

**Example:**
\[
\text{addi } r6, r7, \text{ -100}
\]

**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. Both cases are shown below.

\[
\begin{align*}
\text{addi } rB, rA, \text{ IMM16} & \quad ; \text{The original add operation} \\
\text{cmpltu } rD, rB, rA & \quad ; rD \text{ is written with the carry bit}
\end{align*}
\]

\[
\begin{align*}
\text{addi } rB, rA, \text{ IMM16} & \quad ; \text{The original add operation} \\
\text{bltu } rB, rA, \text{ label} & \quad ; \text{Branch if carry was generated}
\end{align*}
\]

**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 below.

\[
\begin{align*}
\text{addi } rB, rA, \text{ IMM16} & \quad ; \text{The original add operation} \\
\text{xor } rC, rB, rA & \quad ; \text{Compare signs of sum and } rA \\
\text{xorhi } rD, rB, \text{ IMM16} & \quad ; \text{Compare signs of sum and IMM16} \\
\text{and } rC, rC, rD & \quad ; \text{Combine comparisons} \\
\text{blt } rC, r0, \text{ label} & \quad ; \text{Branch if overflow occurred}
\end{align*}
\]

**Exceptions:**
None

**Instruction Type:** `I`

**Instruction Fields:**
- \( A = \) Register index of operand \( rA \)
- \( B = \) Register index of operand \( rB \)
- \( \text{IMM16} = 16\text{-bit signed immediate value} \)

| 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  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

© March 2009 Altera Corporation
and

bitwise logical and

Operation: \( rC \leftarrow rA \& rB \)

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 \( rC \)

| 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  | 0x0e | 0  | 0x3a |

3 1 3 0 2 9 2 8 2 7 2 6 2 5 2 4 2 3 2 2 2 1 2 0 1 9 1 8 1 7 1 6 1 5 1 4 1 3 1 2 1 1 1 0 9 8 7 6 5 4 3 2 1 0
andhi

bitwise logical and immediate into high halfword

Operation:

\[ r_B \leftarrow r_A \& (IMM16 : 0x0000) \]

Assembler Syntax:

`andhi rB, rA, IMM16`

Example:

`andhi r6, r7, 100`

Description:

Calculates the bitwise logical AND of \( r_A \) and \( (IMM16 : 0x0000) \) and stores the result in \( r_B \).

Exceptions:

None

Instruction Type:

I

Instruction Fields:

- \( A = \) Register index of operand \( r_A \)
- \( B = \) Register index of operand \( r_B \)
- \( IMM16 = 16\)-bit unsigned immediate value

| 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 |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | IMM16 |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 0x2c |
andi

Operation: \( rB \leftarrow rA \& (0x0000 : \text{IMM16}) \)

Assembler Syntax: `andi rB, rA, IMM16`

Example: `andi r6, r7, 100`

Description: Calculates the bitwise logical AND of \( rA \) and \((0x0000 : \text{IMM16})\) and stores the result in \( rB \).

Exceptions: None

Instruction Type: I

Instruction Fields:
- \( A \) = Register index of operand \( rA \)
- \( B \) = Register index of operand \( rB \)
- \( \text{IMM16} \) = 16-bit unsigned immediate value

| 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  | IMM16 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     | 0x0c |
beq branch if equal

**Operation:**

If \( rA = rB \)  
then \( PC \leftarrow PC + 4 + \sigma(IMM16) \)  
else \( PC \leftarrow PC + 4 \)

**Assembler Syntax:**

\( \text{beq } rA, rB, \text{ label} \)

**Example:**

\( \text{beq } r6, r7, \text{ label} \)

**Description:**

If \( rA = rB \), then \( \text{beq} \) 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 \( \text{beq} \). 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 \( rB \)  
\( \text{IMM16} = 16\text{-bit signed immediate value} \)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>IMM16</td>
<td>0x26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
bge 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

| 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  | IMM16 | 0x0e |
### bgeu  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

| 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  | IMM16 | 0x2e |
bgt

branch if greater than signed

**Operation:**

\[
\text{if } ((\text{signed}) \ rA > (\text{signed}) \ rB) \\
\text{then } PC \leftarrow \text{label} \\
\text{else } PC \leftarrow PC + 4
\]

**Assembler Syntax:**

\[ \text{bgt } rA, rB, \text{ label} \]

**Example:**

\[ \text{bgt } r6, r7, \text{ top_of_loop} \]

**Description:**

If (signed) \( rA > \) (signed) \( rB \), then \textit{bgt} transfers program control to the instruction at label.

**Pseudo-instruction:**

\text{bgt} is implemented with the \textit{blt} instruction by swapping the register operands.
bgtu \hspace{1cm} \textbf{branch if greater than unsigned}

**Operation:**
\[
\text{if } ((\text{unsigned}) \ rA > (\text{unsigned}) \ rB) \\
\text{then } PC \leftarrow \text{label} \\
\text{else } PC \leftarrow PC + 4
\]

**Assembler Syntax:**
```
\text{bgtu} \ rA, \ rB, \text{label}
```

**Example:**
```
bgtu r6, r7, top_of_loop
```

**Description:** If (unsigned) \( rA > (\text{unsigned}) \ rB \), then \textbf{bgtu} transfers program control to the instruction at \text{label}.

**Pseudo-instruction:** \textbf{bgtu} is implemented with the \textit{bltu} instruction by swapping the register operands.
ble 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 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 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

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td>IMM16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x16</td>
</tr>
</tbody>
</table>
### bltu

**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
MM16 = 16-bit signed immediate value

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td>IMM16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**bne**                      
**branch if not equal**

**Operation:**

if \( rA \neq rB \)  
then \( PC \leftarrow PC + 4 + \sigma(IMM16) \)  
else \( PC \leftarrow PC + 4 \)

**Assembler Syntax:**

\[ bne \ rA, \ rB, \ label \]

**Example:**

\[ bne \ r6, \ r7, \ top\_of\_loop \]

**Description:**

If \( rA \neq rB \), then \texttt{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 \texttt{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:**

\( A = \text{Register index of operand } rA \)

\( B = \text{Register index of operand } rB \)

\( \text{IMM16 = 16-bit signed immediate value} \)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td>IMM16</td>
<td>0x1e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
br

Operation: \( \text{PC} \leftarrow \text{PC} + 4 + \sigma \text{ (IMM16)} \)

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: \( \text{IMM16} = 16\text{-bit signed immediate value} \)
break

**Operation:**

\[
\begin{align*}
\text{bstatus} & \leftarrow \text{status} \\
\text{PIE} & \leftarrow 0 \\
U & \leftarrow 0 \\
\text{ba} & \leftarrow \text{PC} + 4 \\
\text{PC} & \leftarrow \text{break handler address}
\end{align*}
\]

**Assembler Syntax:**

- `break`
- `break imm5`
- `break`

**Example:**

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 at system generation time.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1e</td>
<td>0x34</td>
</tr>
<tr>
<td>IMM5</td>
<td>0x3a</td>
</tr>
</tbody>
</table>
### bret

**Operation:**

\[
\text{status} \leftarrow \text{bstatus} \\
\text{PC} \leftarrow \text{ba}
\]

**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

| 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0x1e | 0 | 0 | 0x09 | 0 | 0x3a |
**call**

**Operation:**

\[ ra \leftarrow PC + 4 \]

\[ PC \leftarrow (PC_{31..28} : IMM26 \times 4) \]

**Assembler Syntax:**

call label

call write_char

**Description:**

Saves the address of the next instruction in register ra, and transfers execution to the instruction at address \( PC_{31..28} : IMM26 \times 4 \).

**Usage:**

call can transfer execution anywhere within the 256 MByte 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

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IMM26</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
callr (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

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>0x1f</th>
<th>0x1d</th>
<th>0</th>
<th>0x3a</th>
</tr>
</thead>
</table>

© March 2009 Altera Corporation

Nios II Processor Reference Handbook
**cmpeq**

**Operation:**
if \( rA \equiv rB \)
then \( rC \leftarrow 1 \)
else \( rC \leftarrow 0 \)

**Assembler Syntax:**
cmpeq rC, rA, rB

cmpeq r6, r7, r8

**Example:**
cmpeq r6, r7, r8

**Description:**
If \( rA \equiv rB \), then stores 1 to \( rC \); otherwise, stores 0 to \( rC \).

**Usage:**
cmpeq performs the \( \equiv \) operation of the C programming language. Also, cmpeq can be used to implement the C logical-negation operator “!”.

cmpeq rC, rA, r0 ; Implements \( rC = \neg rA \)

**Exceptions:**
None

**Instruction Type:**
\( R \)

**Instruction Fields:**
A = Register index of operand \( rA \)
B = Register index of operand \( rB \)
C = Register index of operand \( rC \)
cmpeqi  

compare equal immediate

**Operation:**
if \( rA = (IMM16) \)
then \( rB \leftarrow 1 \)
else \( rB \leftarrow 0 \)

**Assembler Syntax:**
cmpeqi rB, rA, IMM16

cmpeqi r6, r7, 100

**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 \( rA = \sigma (IMM16) \), 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

| 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  | IMM16 | 0x20 |
**cmpge**

**compare greater than or equal signed**

**Operation:**

\[
\text{if } ((\text{signed}) \ rA \geq (\text{signed}) \ rB) \\
\text{then } rC \leftarrow 1 \\
\text{else } rC \leftarrow 0
\]

**Assembler Syntax:**

`cmpge rC, rA, rB`

**Example:**

`cmpge r6, r7, r8`

**Description:**

If \( rA \geq rB \), then stores 1 to \( rC \); otherwise stores 0 to \( rC \).

**Usage:**

`cmpge` performs the signed \( \geq \) 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 \( rC \)

| 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  | 0x08 | 0 | 0x3a |
cmpgei  

**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

cmpgei r6, r7, 100

**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

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |
| A | B | IMM16 | 0x08 |
```
cmpgeu 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

| 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  | 0x28 | 0  | 0x3a |
cmpgeui compare greater than or equal unsigned immediate

**Operation:**

\[
\text{if } ((\text{unsigned}) \ rA \geq (\text{unsigned}) \ (0x0000 : \text{IMM16})) \\
\text{then } rB \leftarrow 1 \\
\text{else } rB \leftarrow 0
\]

**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 \( rA \geq (0x0000 : \text{IMM16}) \), then cmpgeui stores 1 to rB; otherwise stores 0 to rB.

**Usage:**

\( \text{cmpgeui} \) performs the unsigned \( \geq \) operation of the C programming language.

**Exceptions:**

None

**Instruction Type:**

I

**Instruction Fields:**

- \( A = \text{Register index of operand rA} \)
- \( B = \text{Register index of operand rB} \)
- \( \text{IMM16} = 16\text{-bit unsigned immediate value} \)

| 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  | IMM16 | 0x28 |
cmpgt

**Operation:**

\[
\text{if } (\text{(signed) } rA > \text{(signed) } rB) \\
\text{then } rC \leftarrow 1 \\
\text{else } rC \leftarrow 0
\]

**Asmbebler Syntax:**

\[
\text{cmpgt } rC, rA, rB
\]

**Example:**

\[
\text{cmpgt } r6, r7, r8
\]

**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 compare greater than signed immediate

Operation:
\[
\text{if } ((\text{signed}) \ rA > (\text{signed}) \ \text{IMMED}) \\
\text{then } rB \leftarrow 1 \\
\text{else } rB \leftarrow 0
\]

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 \( rA > (\text{signed}) \ \text{IMMED} \), then \text{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 \text{cmpgei} instruction with an IMM16 immediate value of IMMED + 1.
cmpgtu  

**Operation:**

\[
\text{if } ((\text{unsigned } rA > \text{unsigned } rB) \\
\text{then } rC \leftarrow 1 \\
\text{else } rC \leftarrow 0
\]

**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:**

\textit{cmpgtu} performs the unsigned > operation of the C programming language.

**Pseudo-instruction:**

\textit{cmpgtu} is implemented with the \textit{cmpltu} instruction by swapping its \( rA \) and \( rB \) operands.
cmpgtui  compare greater than unsigned immediate

Operation:  
\[
\begin{align*}
\text{if } ((\text{unsigned}) rA > (\text{unsigned}) \text{ IMMED}) \\
\text{then } rB &\leftarrow 1 \\
\text{else } rB &\leftarrow 0
\end{align*}
\]

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 \textit{cmpgtui} stores 1 to rB; otherwise stores 0 to rB.

Usage:  
\textit{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:  
\textit{cmpgtui} is implemented using a \textit{cmpgeui} instruction with an IMM16 immediate value of IMMED + 1.
### cmple

**Operation:**

\[
\text{if } ((\text{signed}) \ rA <= (\text{signed}) \ rB) \\
\text{then } rC \leftarrow 1 \\
\text{else } rC \leftarrow 0
\]

**Assembler Syntax:**

\[
\text{cmple } rC, rA, rB
\]

**Example:**

\[
\text{cmple } r6, r7, r8
\]

**Description:**

If \( rA \leq rB \), then stores 1 to \( rC \); otherwise stores 0 to \( rC \).

**Usage:**

\text{cmple} performs the signed \( \leq \) operation of the C programming language.

**Pseudo-instruction:**

\text{cmple} is implemented with the \text{cmpge} instruction by swapping its \( rA \) and \( rB \) operands.
**cmplei**  
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 ≤ (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**

**Operation:**

\[
\text{if } ((\text{unsigned}) \ rA < (\text{unsigned}) \ rB) \\
\text{then } rC \leftarrow 1 \\
\text{else } rC \leftarrow 0
\]

**Assembler Syntax:**

```
cmpleu \ rC, \ rA, \ rB
```

**Example:**

```
cmpleu \ r6, \ r7, \ r8
```

**Description:**

If \( rA \leq rB \), then stores 1 to \( rC \); otherwise stores 0 to \( rC \).

**Usage:**

\( \text{cmpleu} \) performs the unsigned \( \leq \) operation of the C programming language.

**Pseudo-instruction:**

\( \text{cmpleu} \) is implemented with the \( \text{cmpgeu} \) instruction by swapping its \( rA \) and \( rB \) operands.
**cmpleui**  
**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

**Operation:**
\[
\text{if ((signed) } rA < (\text{signed) } rB) \\
\text{then } rC \leftarrow 1 \\
\text{else } rC \leftarrow 0
\]

**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:**
\texttt{cmpl} 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 \( rC \)

| 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  | 0x10 | 0 | 0x3a |
cmplti  

cmpare less than signed immediate

**Operation:**

\[
\text{if } ((\text{signed}) \ rA < (\text{signed}) \ \sigma (\text{IMM16}))
\text{ then } rB \leftarrow 1
\text{ else } rB \leftarrow 0
\]

**Assembler Syntax:**

\[
\text{cmplti } rB, rA, \text{ IMM16}
\]

**Example:**

\[
\text{cmplti } r6, r7, 100
\]

**Description:**
Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If \( rA < \sigma (\text{IMM16}) \), then \text{cmplti} stores 1 to \( rB \); otherwise stores 0 to \( rB \).

**Usage:**
\text{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

| 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  | IMM16 | 0x10 |
**cmpltu**

**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

| 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   | 0x30 | 0   | 0x3a |
cmpltui: compare less than unsigned immediate

**Operation:**
if ((unsigned) rA < (unsigned) (0x0000 : IMM16))
then rB ← 1
else rB ← 0

**Assembler Syntax:**
cmpltui rB, rA, IMM16

cmpltui r6, r7, 100

**Example:**
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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>IMM16</th>
<th>0x30</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### cmpne

**Operation:**

\[
\text{if (rA} \neq \text{rB)} \\
\text{then } rC \leftarrow 1 \\
\text{else } rC \leftarrow 0
\]

**Assembler Syntax:**

\[
\text{cmpne } rC, rA, rB
\]

**Example:**

\[
\text{cmpne } r6, r7, r8
\]

**Description:**

If \( rA \neq rB \), then stores 1 to \( rC \); otherwise stores 0 to \( rC \).

**Usage:**

\text{cmpne} performs the \( \neq \) 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 \( rC \)

| 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  | 0x18 | 0  | 0x3a |

Nios II Processor Reference Handbook © March 2009 Altera Corporation
cmpnei  

**Operation:**

\[
\text{if } (rA != \sigma(\text{IMM16})) \\
\text{then } rB \leftarrow 1 \\
\text{else } rB \leftarrow 0
\]

**Assembler Syntax:**

\[
cmpnei \ rB, rA, \text{IMM16}
\]

**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 \( rA \neq \sigma(\text{IMM16}) \), 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

```
   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 | IMM16 | 0x18 |
```
**custom**

**custom instruction**

**Operation:**

if \( c = 1 \)
then \( rC \leftarrow f_N(rA, rB, A, B, C) \)
else \( \emptyset \leftarrow f_N(rA, rB, A, B, C) \)

**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 at system generation time. 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
- \( \text{readra} = 1 \) if instruction uses \( rA \), 0 otherwise
- \( \text{readrb} = 1 \) if instruction uses \( rB \), 0 otherwise
- \( \text{writerc} = 1 \) if instruction provides result for \( rC \), 0 otherwise
- \( N \) = 8-bit number that selects instruction

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| A  | B  | C  | ra | rb | rc | N  | 0x32 |
**div**

**Operation:**
\[ rC \leftarrow rA \div rB \]

**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:

\[
\begin{align*}
div & \ rC, \ rA, \ rB & & ; \text{The original div operation} \\
mul & \ rD, \ rC, \ rB \\
sub & \ rD, \ rA, \ rD & & ; \ rD = \text{remainder}
\end{align*}
\]

**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 \( rC \)

| 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| \( A \) | \( B \) | \( C \) | \( 0x25 \) | \( 0 \) | \( 0x3a \) |
**divu**

**Operation:**
\[ r_C \leftarrow r_A \div r_B \]

**Assembler Syntax:**
`divu r_C, r_A, r_B`

**Example:**
`divu r6, r7, r8`

**Description:**
Treating \( r_A \) and \( r_B \) as unsigned integers, this instruction divides \( r_A \) by \( r_B \) and then stores the integer portion of the resulting quotient to \( r_C \). After attempted division by zero, the value of \( r_C \) 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 \( r_D \) using the following instruction sequence:

```
divu r_C, r_A, r_B ; The original divu operation
mul r_D, r_C, r_B
sub r_D, r_A, r_D ; r_D = remainder
```

**Exceptions:**

- Division error
- Unimplemented instruction

**Instruction Type:**

- \( R \)

**Instruction Fields:**

- \( A = \) Register index of operand \( r_A \)
- \( B = \) Register index of operand \( r_B \)
- \( C = \) Register index of operand \( r_C \)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0x24</th>
<th></th>
<th>0x3a</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
eret

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

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1d</td>
<td>0x1e</td>
<td>0</td>
<td>0x01</td>
<td>0</td>
<td>0x3a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
flushd

Flushes the data cache line associated with address rA + σ(IMM16).

Assembler Syntax:

flushd IMM16(rA)
flushd -100(r6)

Example:

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 have not yet been 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” on page 8–51, “initd initialize data cache line” on page 8–54, and “initda initialize data cache address” on page 8–55 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

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td>IMM16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x3b</td>
<td></td>
</tr>
</tbody>
</table>

Nios II Processor Reference Handbook © March 2009 Altera Corporation
flushda  

flush data cache address

**Operation:**
Flushes the data cache line currently caching address \( rA + \sigma (IMM16) \)

**Assembler Syntax:**
- `flushda IMM16(rA)`
- `flushda -100(r6)`

**Example:**
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” on page 8–50, “initd initialize data cache line” on page 8–54, and “initda initialize data cache address” on page 8–55 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>A</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>IMM16</td>
</tr>
<tr>
<td>28 - 0</td>
<td>0x1b</td>
</tr>
</tbody>
</table>
flushi

flushi 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

| 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  | 0  | 0  | 0x0c| 0  | 0x3a|
flushp

Operation: Flushing the processor pipeline of any pre-fetched instructions.

Assembler Syntax: flushp

Example: flushp

Description: Ensures that any instructions pre-fetched 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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0x40 | 0  | 0x3a |
initd  initialize data cache line

**Operation:**
Initializes the data cache line associated with address \( rA + \sigma \) (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” on page 8–50, “flushda flush data cache address” on page 8–51, and “initda initialize data cache address” on page 8–55 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

| 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  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | IMM16 | 0x33 |
**initda**

**initialize data cache address**

**Operation:** Initializes the data cache line currently caching address rA + σ (IMM16)

**Assembler Syntax:**
- `initda IMM16(rA)`
- `initda -100(r6)`

**Example:**
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” on page 8–50, “flushda flush data cache address” on page 8–51, and “initd initialize data cache line” on page 8–54 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

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IMM16</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>0x13</td>
</tr>
</tbody>
</table>
**initi**

**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

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>0</th>
<th>0x29</th>
<th>0</th>
<th>0x3a</th>
</tr>
</thead>
</table>

---

### Table of Instruction Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>31-28</td>
</tr>
<tr>
<td>0</td>
<td>27-24</td>
</tr>
<tr>
<td>0</td>
<td>23-20</td>
</tr>
<tr>
<td>0x29</td>
<td>19-16</td>
</tr>
<tr>
<td>0</td>
<td>15-12</td>
</tr>
<tr>
<td>0x3a</td>
<td>11-8</td>
</tr>
</tbody>
</table>

---
jmp

Operation: \( PC \leftarrow rA \)

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

```
|   | 0 | 0 | 0x0d | 0 | 0x3a |
```
jmp\text{I} \quad \text{jump immediate}

\textbf{Operation:} \quad \text{PC} \leftarrow (\text{PC}_{31..28} : \text{IMM26} \times 4)

\textbf{Assembler Syntax:} \quad \text{jmpi label}

\textbf{Example:} \quad \text{jmpi write_char}

\textbf{Description:} \quad \text{Transfers execution to the instruction at address (PC}_{31..28} : \text{IMM26} \times 4).

\textbf{Usage:} \quad \text{jmpi} \text{ is a low-overhead local jump. jmpi can transfer execution anywhere within the 256 MByte range determined by PC}_{31..28}. \text{ The Nios II GNU linker does not automatically handle cases in which the address is out of this range.}

\textbf{Exceptions:} \quad \text{None}

\textbf{Instruction Type:} \quad \text{J}

\textbf{Instruction Fields:} \quad \text{IMM26} = 26\text{-bit unsigned immediate value}

| 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | IMM26 |    | 0x01 |
ldb / ldbio  
load byte from memory or I/O peripheral

**Operation:**
\[ r_B \leftarrow \sigma (\text{Mem}_8[r_A + \sigma (\text{IMM16})]) \]

**Assembler Syntax:**
- ldb rB, byte_offset(rA)
- ldbio rB, byte_offset(rA)

**Example:**
\[ \text{ldb} \ r6, \ 100(r5) \]

**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

**Instruction Format:**

\[
\begin{array}{cccc}
A & B & \text{IMM16} & 0x07 \\
\end{array}
\]

Instruction format for ldb

\[
\begin{array}{cccc}
A & B & \text{IMM16} & 0x27 \\
\end{array}
\]

Instruction format for ldbio
ldbu / ldbuio

**load unsigned byte from memory or I/O peripheral**

**Operation:**

\[ rB \leftarrow 0x000000 : \text{Mem8}[rA + \sigma(\text{IMM16})] \]

**Assembler Syntax:**

- `ldbu rB, byte_offset(rA)`
- `ldbuio rB, byte_offset(rA)`
- `ldbu r6, 100(r5)`

**Example:**

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

---

### Instruction format for `ldbu`

| 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  | IMM16 | 0x03 |

### Instruction format for `ldbuio`

| 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  | IMM16 | 0x23 |
ldh / ldhio

load halfword from memory or I/O peripheral

Operation:

\[ r_B \leftarrow \sigma (\text{Mem16}[r_A + \sigma (\text{IMM16})]) \]

Assembler Syntax:

- \text{ldh } r_B, \text{byte_offset}(r_A)
- \text{ldhio } r_B, \text{byte_offset}(r_A)

Example:

- \text{ldh } r_6, 100(r_5)

Description:

Computes the effective byte address specified by the sum of \( r_A \) and the instruction’s signed 16-bit immediate value. Loads register \( r_B \) 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 \text{ldhio} instruction for peripheral I/O. In processors with a data cache, \text{ldhio} bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, \text{ldhio} acts like \text{ldh}.

For more information on data cache, refer to the \textit{Cache and Tightly Coupled Memory} chapter of the \textit{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 \( r_A \)
- \( B = \) Register index of operand \( r_B \)
- \( \text{IMM16} = \) 16-bit signed immediate value

<table>
<thead>
<tr>
<th>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</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

Instruction format for \text{ldh}

<table>
<thead>
<tr>
<th>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</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

Instruction format for \text{ldhio}
ldhu / ldhuio

**load unsigned halfword from memory or I/O peripheral**

**Operation:**

\[ rB \leftarrow 0x0000 : \text{Mem16}[rA + \sigma(\text{IMM16})] \]

**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 \( rB \)
- \( \text{IMM16} = 16\text{-bit signed immediate value} \)

**Instruction Format:**

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

Instruction format for `ldhu`

| 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----| 0x2b |

Instruction format for `ldhuio`
ldw / ldwio

**load 32-bit word from memory or I/O peripheral**

**Operation:**

\[ rB \leftarrow \text{Mem32}[rA + \sigma(\text{IMM14})] \]

**Assembler Syntax:**

- `ldw rB, byte_offset(rA)`
- `ldwio rB, byte_offset(rA)`
- `ldw r6, 100(r5)`

**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, `ldw` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldw` 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

**Instruction format for `ldw`**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>IMM16</td>
<td>0x17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Instruction format for `ldwio`**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>IMM16</td>
<td>0x37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
mov  

Operation:  \( rC \leftarrow rA \)

Assembler Syntax:  \( \text{mov } rC, rA \)

Example:  \( \text{mov } r6, r7 \)

Description:  Moves the contents of \( rA \) to \( rC \).

Pseudo-instruction:  \( \text{mov} \) is implemented as \( \text{add } rC, rA, r0. \)
movhi  move immediate into high halfword

**Operation:**

\[ rB \leftarrow (IMMED : 0x0000) \]

**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 below.

```
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 below.

```
movhi rB, %hiadj(value)
addi rB, rB, %lo(value)
```

**Pseudo-instruction:**

`: movhi is implemented as orhi rB, r0, IMMED.`
movi

move signed immediate into word

Operation: \[ rB \leftarrow \sigma(\text{IMMED}) \]

Assembler Syntax: \text{movi \ rB, IMMED}

Example: \text{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 \text{addi \ rB, r0, IMMED.}
**movia**

**move immediate address into word**

**Operation:**

\[ rB \leftarrow \text{label} \]

**Assembler Syntax:**

`movia rB, label`

**Example:**

`movia r6, function_address`

**Description:**

Writes the address of label to rB.

**Pseudo-instruction:**

`movia` is implemented as:

\[
\begin{align*}
\text{orhi} & \hspace{1em} rB, r0, \%\text{hiadj}(\text{label}) \\
\text{addi} & \hspace{1em} rB, rB, \%\text{lo}(\text{label})
\end{align*}
\]
movui  move unsigned immediate into word

**Operation:**

\[ rB \leftarrow (0x0000 : \text{IMMED}) \]

**Assembler Syntax:**

\[
\text{movui } rB, \text{ IMMED} \\
\text{movui } r6, 100
\]

**Example:**

**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:**

\[
\text{movui is implemented as ori } rB, r0, \text{ IMMED.}
\]
mul

Operation: \[ rC \leftarrow (rA \times rB)_{31.0} \]

Assembler Syntax:

\[ \text{mul } rC, rA, rB \]

Example:

\[ \text{mul } r6, r7, r8 \]

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 \texttt{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:

\[
\begin{align*}
\text{mul } rC, rA, rB & ; \text{ The mul operation (optional)} \\
\text{mulxuu } rD, rA, rB & ; rD \text{ is non-zero if carry occurred} \\
\text{cmpne } rD, rD, r0 & ; rD \text{ is 1 if carry occurred, 0 if not}
\end{align*}
\]

The \texttt{mulxuu} instruction writes a non-zero value into \( rD \) if the multiplication of unsigned numbers will generate a carry (unsigned overflow). If a 0/1 result is desired, follow the \texttt{mulxuu} with the \texttt{cmpne} instruction.

 Overflow Detection (signed operands):

After the multiply operation, overflow can be detected using the following instruction sequence:

\[
\begin{align*}
\text{mul } rC, rA, rB & ; \text{ The original mul operation} \\
\text{cmplt } rD, rC, r0 & \\
\text{mulxss } rE, rA, rB & \\
\text{add } rD, rD, rE & ; rD \text{ is non-zero if overflow} \\
\text{cmpne } rD, rD, r0 & ; rD \text{ is 1 if overflow, 0 if not}
\end{align*}
\]

The \texttt{cmplt-mulxss-add} instruction sequence writes a non-zero 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 \texttt{cmpne} instruction.

Exceptions:

 Unimplemented instruction

Instruction Type: \( R \)

Instruction Fields:

\[ A = \text{Register index of operand } rA \]

\[ B = \text{Register index of operand } rB \]

\[ C = \text{Register index of operand } rC \]
### muli

**Operation:**
\[ r_B \leftarrow (r_A \times \sigma(IMM16))_{31..0} \]

**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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>IMM16</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>0x24</td>
<td></td>
</tr>
</tbody>
</table>
mulxss  multiply extended signed/signed

Operation:  \[ r_C \leftarrow ((\text{signed}) \, r_A) \times ((\text{signed}) \, r_B) \]_{63..32}

Assembler Syntax:  \text{mulxss} \, r_C, \, r_A, \, r_B

Example:  \text{mulxss} \, r_6, \, r_7, \, r_8

Description:  Treating \( r_A \) and \( r_B \) as signed integers, \text{mulxss} multiplies \( r_A \times r_B \), and stores the 32 high-order bits of the product to \( r_C \).

Nios II processors that do not implement the \text{mulxss} instruction cause an unimplemented-instruction exception.

Usage:  Use \text{mulxss} and \text{mul} to compute the full 64-bit product of two 32-bit signed integers. Furthermore, \text{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, \((S_1 : U_1)\) and \((S_2 : U_2)\), their 128-bit product is \((U_1 \times U_2) + ((S_1 \times U_2) << 32) + ((U_1 \times S_2) << 32) + ((S_1 \times S_2) << 64)\). The \text{mulxss} and \text{mul} instructions are used to calculate the 64-bit product \( S_1 \times S_2 \).

Exceptions:  Unimplemented instruction

Instruction Type:  \( R \)

Instruction Fields:  
A = Register index of operand \( r_A \)
B = Register index of operand \( r_B \)
C = Register index of operand \( r_C \)

| 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  | 0x1f | 0 | 0x3a |
mulxsu  

multiply extended signed/unsigned

Operation:
\[ rC \leftarrow ((\text{signed} \ rA) \times (\text{unsigned} \ rB))_{63:32} \]

Assembler Syntax: 
mulxsu rC, rA, rB

Example:
mulxsu r6, r7, r8

Description:
Treating rA as a signed integer and rB as an unsigned integer, \texttt{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 \texttt{mulxsu} instruction cause an unimplemented-instruction exception.

Usage:
\texttt{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: \((U1 \times U2) + ((S1 \times U2) \ll 32) + ((U1 \times S2) \ll 32) + ((S1 \times S2) \ll 64)\). The \texttt{mulxsu} and \texttt{mul} instructions are used to calculate the two 64-bit products \(S1 \times U2\) and \(U1 \times S2\).

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

| 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  | 0x17 | 0 | 0x3a |
mulxuu

**multiply extended unsigned/unsigned**

**Operation:**

\[ r_C \leftarrow ((\text{unsigned}) \, r_A) \times ((\text{unsigned}) \, r_B) \] 

**Assembler Syntax:**

`mulxuu r_C, r_A, r_B`

**Example:**

`mulxuu r_6, r_7, r_8`

**Description:**

Treating \( r_A \) and \( r_B \) as unsigned integers, \( \text{mulxuu} \) multiplies \( r_A \) times \( r_B \) and stores the 32 high-order bits of the product to \( r_C \).

Nios II processors that do not implement the \( \text{mulxuu} \) instruction cause an unimplemented-instruction exception.

**Usage:**

Use \( \text{mulxuu} \) and \( \text{mul} \) to compute the 64-bit product of two 32-bit unsigned integers. Furthermore, \( \text{mulxuu} \) 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, \((S_1 : U_1)\) and \((S_2 : U_2)\), their 128-bit product is \((U_1 \times U_2) + ((S_1 \times U_2) \ll 32) + ((U_1 \times S_2) \ll 32) + ((S_1 \times S_2) \ll 64)\). The \( \text{mulxuu} \) and \( \text{mul} \) instructions are used to calculate the 64-bit product \( U_1 \times U_2 \).

\( \text{mulxuu} \) 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, \((T_1 : U_1)\) and \((T_2 : U_2)\), their 128-bit product is \((U_1 \times U_2) + ((U_1 \times T_2) \ll 32) + ((T_1 \times U_2) \ll 32) + ((T_1 \times T_2) \ll 64)\). The \( \text{mulxuu} \) and \( \text{mul} \) instructions are used to calculate the four 64-bit products \( U_1 \times U_2, U_1 \times T_2, T_1 \times U_2, \) and \( T_1 \times T_2 \).

**Exceptions:**

Unimplemented instruction

**Instruction Type:**

R

**Instruction Fields:**

\( A = \) Register index of operand \( r_A \)

\( B = \) Register index of operand \( r_B \)

\( C = \) Register index of operand \( r_C \)
nextpc  

get address of following instruction

Operation:  \( rC \leftarrow PC + 4 \)

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:  

\[
\begin{array}{cccccccccccccccccccc}
\hline
0 & 0 & C & 0x1c & 0 & 0x3a \\
\end{array}
\]
### nop

**no operation**

<table>
<thead>
<tr>
<th>Operation:</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler Syntax:</td>
<td><code>nop</code></td>
</tr>
<tr>
<td>Example:</td>
<td><code>nop</code></td>
</tr>
<tr>
<td>Description:</td>
<td><code>nop</code> does nothing.</td>
</tr>
<tr>
<td>Pseudo-instruction:</td>
<td><code>nop</code> is implemented as <code>add r0, r0, r0</code>.</td>
</tr>
</tbody>
</table>
nor

bitwise logical nor

Operation: \( rC \leftarrow \neg(rA \lor rB) \)
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 \( rC \)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**or**

**Operation:**
\[ rC \leftarrow rA \mid rB \]

**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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>0x16</th>
<th>0</th>
<th>0x3a</th>
</tr>
</thead>
</table>
**orhi**  
**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
**ori** bitwise logical or immediate

**Operation:**
\[ r_B \leftarrow r_A \mid (0x0000 : \text{IMM16}) \]

**Assembler Syntax:**
```
ori r_B, r_A, IMM16
```

**Example:**
```
ori r6, r7, 100
```

**Description:**
Calculates the bitwise logical OR of \( r_A \) and \((0x0000 : \text{IMM16})\) and stores the result in \( r_B \).

**Exceptions:**
None

**Instruction Type:**
I

**Instruction Fields:**
- \( A = \) Register index of operand \( r_A \)
- \( B = \) Register index of operand \( r_B \)
- \( \text{IMM16} = 16\)-bit unsigned immediate value

| 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  | IMM16 | 0x14 |
**rdctl**

**read from control register**

**Operation:**
\[ rC \leftarrow \text{ctlN} \]

**Assembler Syntax:**
\[ \text{rdctl } rC, \text{ ctlN} \]

**Example:**
\[ \text{rdctl } r3, \text{ ctl31} \]

**Description:**
Reads the value contained in control register \( \text{ctlN} \) 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 \( \text{ctlN} \)

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

Operation: \( PC \leftarrow ra \)

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

| 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|
| 0x1f | 0 | 0 | 0x05 | 0 | 0x3a |
rol

**Operation:**
rC ← rA rotated left rB<sub>4..0</sub> 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<sub>4..0</sub> 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

```
A  B  C  0x03  0  0x3a
```
roli  

**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

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>C</td>
<td>0x02</td>
<td>IMM5</td>
<td>0x3a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ror**  

**Operation:**  
\[ rC \leftarrow rA \text{ rotated right } rB_{4..0} \text{ bit positions} \]

**Assembler Syntax:**  
\texttt{ror rC, rA, rB}

**Example:**  
\texttt{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:**  
\( A = \text{Register index of operand } rA \)
\( B = \text{Register index of operand } rB \)
\( C = \text{Register index of operand } rC \)

| 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  | 0x0b | 0  | 0x3a |

Nios II Processor Reference Handbook © March 2009 Altera Corporation
**sll**  \( \text{shift left logical} \)

**Operation:**  \( rC \leftarrow rA \ll (rB_{4..0}) \)

**Assembler Syntax:**  \( \text{sll } rC, rA, rB \)

**Example:**  \( \text{sll } r6, r7, r8 \)

**Description:** Shifts \( rA \) left by the number of bits specified in \( rB_{4..0} \) (inserting zeroes), and then stores the result in \( rC \). \( \text{sll} \) performs the \( \ll \) operation of the C programming language.

**Exceptions:** None

**Instruction Type:** R

**Instruction Fields:**

- \( A = \text{Register index of operand } rA \)
- \( B = \text{Register index of operand } rB \)
- \( C = \text{Register index of operand } rC \)

| 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  | 0x13 | 0  | 0x3a |
slli

shift left logical immediate

Operation: \( r_C \leftarrow r_A \ll \text{IMM5} \)

Assembler Syntax: \text{slli \ rC, \ rA, \ IMM5}

Example: \text{slli \ r6, \ r7, \ 3}

Description: Shifts \( r_A \) left by the number of bits specified in \( \text{IMM5} \) (inserting zeroes), and then stores the result in \( r_C \).

Usage: \text{slli} performs the \( \ll \) operation of the C programming language.

Exceptions: None

Instruction Type: R

Instruction Fields:
- \( A = \) Register index of operand \( r_A \)
- \( C = \) Register index of operand \( r_C \)
- \( \text{IMM5} = \) 5-bit unsigned immediate value

\[
\begin{array}{c|c|c|c|c}
31 & 30 & 29 & 28 & 27 \\
26 & 25 & 24 & 23 & 22 \\
21 & 20 & 19 & 18 & 17 \\
16 & 15 & 14 & 13 & 12 \\
11 & 10 & 9 & 8 & 7 \\
6 & 5 & 4 & 3 & 2 \\
1 & 0 & & & \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
A & 0 & C & 0x12 & \text{IMM5} & 0x3a \\
\end{array}
\]
sra

**Operation:**
\[ rC \leftarrow \text{(signed) } rA \gg ((\text{unsigned}) \ rB_{4..0}) \]

**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 \( rC \)
### srai

**Operation:**
\[ rC \leftarrow \text{(signed) } rA \gg ((\text{unsigned}) \text{ IMM5}) \]

**Assembler Syntax:**
`srai rC, rA, IMM5`

**Example:**
`srai r6, r7, 3`

**Description:**
Shifts \( rA \) right by the number of bits specified in \( \text{IMM5} \) (duplicating the sign bit), and then stores the result in \( rC \).

**Usage:**
srai performs the signed \( \gg \) operation of the C programming language.

**Exceptions:**
None

**Instruction Type:**
R

**Instruction Fields:**
- \( A \) = Register index of operand \( rA \)
- \( C \) = Register index of operand \( rC \)
- \( \text{IMM5} \) = 5-bit unsigned immediate value

|   | 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 | 0  |    | C  | 0x3a|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| IMM5 | 0x3a |
srl shift right logical

Operation: \[ r_C \leftarrow \text{(unsigned) } r_A \gg ((\text{unsigned}) \; r_B_{4..0}) \]
Assembler Syntax: \text{srl } r_C, r_A, r_B
Example: \text{srl } r_6, r_7, r_8
Description: Shifts \( r_A \) right by the number of bits specified in \( r_B_{4..0} \) (inserting zeroes), and then stores the result in \( r_C \). Bits 31–5 are ignored.
Usage: \text{srl} performs the unsigned >> operation of the C programming language.
Exceptions: None
Instruction Type: R
Instruction Fields:
\begin{align*}
A &= \text{Register index of operand } r_A \\
B &= \text{Register index of operand } r_B \\
C &= \text{Register index of operand } r_C
\end{align*}
srli

Operation: \( r_C \leftarrow (\text{unsigned}) \ r_A \gg ((\text{unsigned}) \ IMM5) \)

Assembler Syntax: srli r_C, r_A, IMM5

Example: srli r6, r7, 3

Description: Shifts \( r_A \) right by the number of bits specified in \( IMM5 \) (inserting zeroes), and then stores the result in \( r_C \).

Usage: srli performs the unsigned >> operation of the C programming language.

Exceptions: None

Instruction Type: R

Instruction Fields:
- \( A \) = Register index of operand \( r_A \)
- \( C \) = Register index of operand \( r_C \)
- \( IMM5 \) = 5-bit unsigned immediate value

| 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  | 0  | C  | 0x1a| IMM5| 0x3a|
**stb / stbio**

**store byte to memory or I/O peripheral**

**Operation:**

\[
\text{Mem8}[rA + \sigma(\text{IMM16})] \leftarrow rB_{7:0}
\]

**Assembler Syntax:**

- `stb rB, byte_offset(rA)`
- `stbio rB, byte_offset(rA)`
- `stb r6, 100(r5)`

**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 non-cache 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 \(rB\)
- \(\text{IMM16} = 16\)-bit signed immediate value

**Instruction format for stb:**

| 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  | IMM16 | | 0x05 |

**Instruction format for stbio:**

| 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  | IMM16 | | 0x25 |
sth / sthio

store halfword to memory or I/O peripheral

**Operation:**
\[
\text{Mem16}[rA + \sigma(IMM16)] \leftarrow rB_{15..0}
\]

**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

---

**Instruction format for sth**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td>IMM16</td>
<td>0x0d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Instruction format for sthio**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td>IMM16</td>
<td>0x2d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
stw / stwio  

**store word to memory or I/O peripheral**

**Operation:**

\[ \text{Mem32}[rA + \sigma(\text{IMM16})] \leftarrow rB \]

**Assembler Syntax:**

- `stw rB, byte_offset(rA)`
- `stwio rB, byte_offset(rA)`
- `stw r6, 100(r5)`

**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:**

- \( \text{A} = \text{Register index of operand } rA \)
- \( \text{B} = \text{Register index of operand } rB \)
- \( \text{IMM16} = \text{16-bit signed immediate value} \)

### Instruction Format for `stw`

```
  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 | IMM16     | 0x15      |
```

### Instruction Format for `stwio`

```
  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 | IMM16     | 0x35      |
```
### sub

**Operation:**

\[ \text{rC} \leftarrow \text{rA} - \text{rB} \]

**Assembler Syntax:**

```
sub rC, rA, rB
```

**Example:**

```
sub r6, r7, r8
```

**Description:**

Subtract \( \text{rB} \) from \( \text{rA} \) and store the result in \( \text{rC} \).

**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 below.

```
sub rC, rA, rB ; The original sub operation (optional)
cmpltu rD, rA, rB ; rD is written with the carry bit
sub rC, rA, rB ; The original sub operation (optional)
bltu rA, rB, label ; Branch if carry was generated
```

**Overflow Detection (signed operands):**

Detect overflow of signed subtraction by comparing the sign of the difference that is written to \( \text{rC} \) with the signs of the operands. If \( \text{rA} \) and \( \text{rB} \) have different signs, and the sign of \( \text{rC} \) is different than the sign of \( \text{rA} \), an overflow occurred. The overflow condition can control a conditional branch, as shown below.

```
sub rC, rA, rB ; The original sub operation
xor rD, rA, rB ; Compare signs of \( \text{rA} \) and \( \text{rB} \)
xor rE, rA, rC ; Compare signs of \( \text{rA} \) and \( \text{rC} \)
and rD, rD, rE ; Combine comparisons
blt rD, r0, label ; Branch if overflow occurred
```

**Exceptions:**

None

**Instruction Type:**

R

**Instruction Fields:**

- \( A \) = Register index of operand \( \text{rA} \)
- \( B \) = Register index of operand \( \text{rB} \)
- \( C \) = Register index of operand \( \text{rC} \)

| 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  | 0x39 | 0  | 0x3a |
subi  

**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**

<table>
<thead>
<tr>
<th>Operation:</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler Syntax:</td>
<td>sync</td>
</tr>
<tr>
<td>Example:</td>
<td>sync</td>
</tr>
</tbody>
</table>

**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

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

memory synchronization
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 at system generation time.

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
wrctl

write to control register

Operation: \( \text{ctlN} \leftarrow rA \)

Assembler Syntax: \text{wrctl ctlN, rA}

Example: \text{wrctl ctl6, r3}

Description: Writes the value contained in register \( rA \) to the control register \( \text{ctlN} \).

Exceptions: Supervisor-only instruction

Instruction Type: \( R \)

Instruction Fields:
- \( A = \) Register index of operand \( rA \)
- \( N = \) Control register index of operand \( \text{ctlN} \)

| 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  | 0  | 0  | 0x2e| N  | 0x3a|
**xor**

**Operation:**
\[ rC \leftarrow rA \oplus rB \]

**Assembler Syntax:**
\[ \text{xor } rC, rA, rB \]

**Example:**
\[ \text{xor } r6, r7, r8 \]

**Description:**
Calculates the bitwise logical exclusive XOR 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  | 0x1e | 0   | 0x3a |
xorhi bitwise logical exclusive or immediate into high halfword

Operation: \[ rB \leftarrow rA \oplus (IMM16 : 0x0000) \]

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

| 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  | IMM16 | 0x3c |
xori bitwise logical exclusive or immediate

Operation: \( rB \leftarrow rA \oplus (0x0000 : \text{IMM16}) \)

Assembler Syntax: `xori rB, rA, IMM16`

Example: `xori r6, r7, 100`

Description: Calculates the bitwise logical exclusive OR of \( rA \) and \((0x0000 : \text{IMM16})\) and stores the result in \( rB \).

Exceptions: None

Instruction Type: I

Instruction Fields:
- \( A \) = Register index of operand \( rA \)
- \( B \) = Register index of operand \( rB \)
- \( \text{IMM16} \) = 16-bit unsigned immediate value

| 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  | IMM16 | 0x1c |
Referenced Documents

This chapter references the following documents:

- **Programming Model** chapter of the Nios II Processor Reference Handbook
- **Application Binary Interface** chapter of the Nios II Processor Reference Handbook
- **Cache and Tightly Coupled Memory** chapter of the Nios II Software Developer’s Handbook

Document Revision History

Table 8–6 shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date &amp; Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>March 2009 v9.0.0</td>
<td>Backward-compatible change to the eret instruction B field encoding.</td>
<td>—</td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Maintenance release.</td>
<td>—</td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>Added an Exceptions section to all instructions.</td>
<td>Added MMU.</td>
</tr>
<tr>
<td>October 2007 v7.2.0</td>
<td>Added jmpi instruction.</td>
<td>—</td>
</tr>
<tr>
<td>May 2007 v7.1.0</td>
<td>■ Added table of contents to Introduction section.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Added Referenced Documents section.</td>
<td></td>
</tr>
<tr>
<td>March 2007 v7.0.0</td>
<td>Maintenance release.</td>
<td>—</td>
</tr>
<tr>
<td>November 2006 v6.1.0</td>
<td>Maintenance release.</td>
<td>—</td>
</tr>
<tr>
<td>May 2006 v6.0.0</td>
<td>Maintenance release.</td>
<td>—</td>
</tr>
<tr>
<td>October 2005 v5.1.0</td>
<td>■ Correction to the blt instruction.</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>■ Added U bit operation for break and trap instructions.</td>
<td></td>
</tr>
<tr>
<td>July 2005 v5.0.1</td>
<td>■ Added new flushda instruction.</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>■ Updated flushd instruction.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Instruction Opcode table updated with flushda instruction.</td>
<td></td>
</tr>
<tr>
<td>May 2005 v5.0.0</td>
<td>Maintenance release.</td>
<td>—</td>
</tr>
<tr>
<td>December 2004 v1.2</td>
<td>■ break instruction update.</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>■ srl instruction correction.</td>
<td>—</td>
</tr>
</tbody>
</table>
Table 8–6. Document Revision History (Part 2 of 2)

<table>
<thead>
<tr>
<th>Date &amp; Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 2004 v1.1</td>
<td>Updates for Nios II 1.01 release.</td>
<td>—</td>
</tr>
<tr>
<td>May 2004 v1.0</td>
<td>Initial release.</td>
<td>—</td>
</tr>
</tbody>
</table>