


This section provides additional information about the Nios® II processor.

This section includes the following chapters:

- [Chapter 5, Nios II Core Implementation Details](#)
- [Chapter 6, Nios II Processor Revision History](#)
- [Chapter 7, Application Binary Interface](#)
- [Chapter 8, Instruction Set Reference](#)

Introduction

This document describes all of the Nios® II processor core implementations available at the time of publishing. This document describes only implementation-specific features of each processor core. All cores support the Nios II instruction set architecture.

 For more information regarding the Nios II instruction set architecture, refer to the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*.

For common core information and details on a specific core, refer to the appropriate section:

- “Device Family Support” on page 5–3
- “Nios II/f Core” on page 5–3
- “Nios II/s Core” on page 5–12
- “Nios II/e Core” on page 5–18

Table 5–1 compares the objectives and features of each Nios II processor core. The table is designed to help system designers choose the core that best suits their target application.

Table 5–1. Nios II Processor Cores (Part 1 of 3)

Feature		Core		
		Nios II/e	Nios II/s	Nios II/f
Objective		Minimal core size	Small core size	Fast execution speed
Performance	DMIPS/MHz (1)	0.15	0.74	1.16
	Max. DMIPS (2)	31	127	218
	Max. f_{MAX} (2)	200 MHz	165 MHz	185 MHz
Area		< 700 LEs; < 350 ALMs	< 1400 LEs; < 700 ALMs	Without MMU or MPU: < 1800 LEs; < 900 ALMs With MMU: < 3000 LEs; < 1500 ALMs With MPU: < 2400 LEs; < 1200 ALMs
Pipeline		1 stage	5 stages	6 stages
External Address Space		2 GBytes	2 GBytes	2 GBytes without MMU 4 GBytes with MMU

Table 5-1. Nios II Processor Cores (Part 2 of 3)

Feature		Core		
		Nios II/e	Nios II/s	Nios II/f
Instruction Bus	Cache	–	512 bytes to 64 KBytes	512 bytes to 64 KBytes
	Pipelined Memory Access	–	Yes	Yes
	Branch Prediction	–	Static	Dynamic
	Tightly-Coupled Memory	–	Optional	Optional
Data Bus	Cache	–	–	512 bytes to 64 KBytes
	Pipelined Memory Access	–	–	–
	Cache Bypass Methods	–	–	<ul style="list-style-type: none"> ■ I/O instructions ■ Bit-31 cache bypass ■ Optional MMU
	Tightly-Coupled Memory	–	–	Optional
Arithmetic Logic Unit	Hardware Multiply	–	3-cycle (3)	1-cycle (3)
	Hardware Divide	–	Optional	Optional
	Shifter	1 cycle-per-bit	3-cycle shift (3)	1-cycle barrel shifter (3)
JTAG Debug Module	JTAG interface, run control, software breakpoints	Optional	Optional	Optional
	Hardware Breakpoints	–	Optional	Optional
	Off-Chip Trace Buffer	–	Optional	Optional
Memory Management Unit		–	–	Optional
Memory Protection Unit		–	–	Optional
Exception Handling	Exception Types	Software trap, unimplemented instruction, illegal instruction, hardware interrupt	Software trap, unimplemented instruction, illegal instruction, hardware interrupt	Software trap, unimplemented instruction, illegal instruction, supervisor-only instruction, supervisor-only instruction address, supervisor-only data address, misaligned destination address, misaligned data address, division error, fast TLB miss, double TLB miss, TLB permission violation, MPU region violation, hardware interrupt
	Integrated Interrupt Controller	Yes	Yes	Yes
User Mode Support		No; Permanently in supervisor mode	No; Permanently in supervisor mode	Yes; When MMU or MPU present

Table 5-1. Nios II Processor Cores (Part 3 of 3)

Feature	Core		
	Nios II/e	Nios II/s	Nios II/f
Custom Instruction Support	Yes	Yes	Yes

Notes to Table 5-1:

- (1) DMIPS performance for the Nios II/s and Nios II/f cores depends on the hardware multiply option.
- (2) Using the fastest hardware multiply option, and targeting a Stratix II FPGA in the fastest speed grade.
- (3) Multiply and shift performance depends on which hardware multiply option is used. If no hardware multiply option is used, multiply operations are emulated in software, and shift operations require one cycle per bit. For details, refer to the arithmetic logic unit description for each core.

Device Family Support

All Nios II cores provide the same support for target Altera device families. Nios II cores provide either full or preliminary device family support, as described below:

- Full support means the Nios II cores meet all functional and timing requirements for the device family and may be used in production designs
- Preliminary support means the Nios II cores meet all functional requirements, but might still be undergoing timing analysis for the device family; they may be used in production designs with caution.

Table 5-2 shows the level of support offered to each of the Altera device families by the Nios II cores.

Table 5-2. Device Family Support

Device Family	Support
Arria™ GX	Full
Stratix® IV	Preliminary
Stratix III	Full
Stratix II	Full
Stratix II GX	Full
Stratix GX	Full
Stratix	Full
Hardcopy® II	Full
HardCopy	Full
Cyclone® III	Full
Cyclone II	Full
Cyclone	Full
Other device families	No support

Nios II/f Core

The Nios II/f fast core is designed for high execution performance. Performance is gained at the expense of core size. The base Nios II/f core, without the memory management unit (MMU) or memory protection unit (MPU), is approximately 25% larger than the Nios II/s core. Altera designed the Nios II/f core with the following design goals in mind:

- Maximize the instructions-per-cycle execution efficiency
- Maximize f_{MAX} performance of the processor core

The resulting core is optimal for performance-critical applications, as well as for applications with large amounts of code and/or data, such as systems running a full-featured operating system.

Overview

The Nios II/f core:

- Has separate instruction and data caches
- Provides optional MMU to support operating systems that require an MMU
- Provides optional MPU to support operating systems and runtime environments that desire memory protection but do not need virtual memory management
- Can access up to 2 GBytes of external address space when no MMU is present and 4 GBytes when the MMU is present
- Supports optional tightly-coupled memory for instructions and data
- Employs a 6-stage pipeline to achieve maximum DMIPS/MHz
- Performs dynamic branch prediction
- Provides hardware multiply, divide, and shift options to improve arithmetic performance
- Supports the addition of custom instructions
- Supports the JTAG debug module
- Supports optional JTAG debug module enhancements, including hardware breakpoints and real-time trace

The following sections discuss the noteworthy details of the Nios II/f core implementation. This document does not discuss low-level design issues or implementation details that do not affect Nios II hardware or software designers.

Arithmetic Logic Unit

The Nios II/f core provides several arithmetic logic unit (ALU) options to improve the performance of multiply, divide, and shift operations.

Multiply and Divide Performance

The Nios II/f core provides the following hardware multiplier options:

- DSP Block—Includes DSP block multipliers available on the target device. This option is available only on Altera FPGAs that have DSP Blocks.
- Embedded Multipliers—Includes dedicated embedded multipliers available on the target device. This option is available only on Altera FPGAs that have embedded multipliers.
- Logic Elements—Includes hardware multipliers built from logic element (LE) resources.

- None—Does not include multiply hardware. In this case, multiply operations are emulated in software.

The Nios II/f core also provides a hardware divide option that includes LE-based divide circuitry in the ALU.

Including an ALU option improves the performance of one or more arithmetic instructions.



The performance of the embedded multipliers differ, depending on the target FPGA family.

Table 5-3 lists the details of the hardware multiply and divide options.

Table 5-3. Hardware Multiply and Divide Details for the Nios II/f Core

ALU Option	Hardware Details	Cycles per Instruction	Result Latency Cycles	Supported Instructions
No hardware multiply or divide	Multiply and divide instructions generate an exception	–	–	None
Logic elements	ALU includes 32 x 4-bit multiplier	11	+2	mul, muli
DSP block on Stratix, Stratix II and Stratix III families	ALU includes 32 x 32-bit multiplier	1	+2	mul, muli, mulxss, mulxsu, mulxuu
Embedded multipliers on Cyclone II and Cyclone III families	ALU includes 32 x 16-bit multiplier	5	+2	mul, muli
Hardware divide	ALU includes multicycle divide circuit	4 – 66	+2	div, divu

The cycles per instruction value determines the maximum rate at which the ALU can dispatch instructions and produce each result. The latency value determines when the result becomes available. If there is no data dependency between the results and operands for back-to-back instructions, then the latency does not affect throughput. However, if an instruction depends on the result of an earlier instruction, then the processor stalls through any result latency cycles until the result is ready.

In the following code example, a multiply operation (with 1 instruction cycle and 2 result latency cycles) is followed immediately by an add operation that uses the result of the multiply. On the Nios II/f core, the `addi` instruction, like most ALU instructions, executes in a single cycle. However, in this code example, execution of the `addi` instruction is delayed by two additional cycles until the multiply operation completes.

```
mul r1, r2, r3      ; r1 = r2 * r3
addi r1, r1, 100   ; r1 = r1 + 100 (Depends on result of mul)
```

In contrast, the following code does not stall the processor.

```
mul r1, r2, r3      ; r1 = r2 * r3
or r5, r5, r6       ; No dependency on previous results
or r7, r7, r8       ; No dependency on previous results
addi r1, r1, 100    ; r1 = r1 + 100 (Depends on result of mul)
```

Shift and Rotate Performance

The performance of shift operations depends on the hardware multiply option. When a hardware multiplier is present, the ALU achieves shift and rotate operations in one or two clock cycles. Otherwise, the ALU includes dedicated shift circuitry that achieves one-bit-per-cycle shift and rotate performance. Refer to [Table 5-9 on page 5-11](#) for details.

Memory Access

The Nios II/f core provides optional instruction and data caches. The cache size for each is user-definable, between 512 bytes and 64 KBytes.

The memory address width in the Nios II/f core depends on whether the optional MMU is present. Without an MMU, the Nios II/f core supports the bit-31 cache bypass method for accessing I/O on the data master port. Therefore addresses are 31 bits wide, reserving bit 31 for the cache bypass function. With an MMU, cache bypass is a function of the memory partition and the contents of the translation lookaside buffer (TLB). Therefore bit-31 cache bypass is disabled, and 32 address bits are available to address memory.

Instruction and Data Master Ports

The instruction master port is a pipelined Avalon® Memory-Mapped (Avalon-MM) master port. If the core includes data cache with a line size greater than four bytes, then the data master port is a pipelined Avalon-MM master port. Otherwise, the data master port is not pipelined.

The instruction and data master ports on the Nios II/f core are optional. A master port can be excluded, as long as the core includes at least one tightly-coupled memory to take the place of the missing master port.



Although the Nios II processor can operate entirely out of tightly-coupled memory without the need for Avalon-MM instruction or data masters, software debug is not possible when either the Avalon-MM instruction or data master is omitted.

Support for pipelined Avalon-MM transfers minimizes the impact of synchronous memory with pipeline latency. The pipelined instruction and data master ports can issue successive read requests before prior requests complete.

Instruction and Data Caches

This section first describes the similar characteristics of the instruction and data cache memories, and then describes the differences.

Both the instruction and data cache addresses are divided into fields based on whether or not an MMU is present in your system. [Table 5-4](#) shows the cache byte address fields for systems without an MMU present.

Table 5-4. Cache Byte Address Fields

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
tag												line										offset									

Table 5-5 shows the cache virtual byte address fields for systems with an MMU present. Table 5-6 shows the cache physical byte address fields for systems with an MMU present.

Table 5-5. Cache Virtual Byte Address Fields

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
												line												offset							

Table 5-6. Cache Physical Byte Address Fields

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
tag																											offset				

Instruction Cache

The instruction cache memory has the following characteristics:

- Direct-mapped cache implementation.
- 32 bytes (8 words) per cache line.
- The instruction master port reads an entire cache line at a time from memory, and issues one read per clock cycle.
- Critical word first.
- Virtually-indexed, physically-tagged, when MMU present.

The size of the tag field depends on the size of the cache memory and the physical address size. The size of the line field depends only on the size of the cache memory. The offset field is always five bits (i.e., a 32-byte line). The maximum instruction byte address size is 31 bits in systems without an MMU present. In systems with an MMU, the maximum instruction byte address size is 32 bits and the tag field always includes all the bits of the physical frame number (PFN).

The instruction cache is optional. However, excluding instruction cache from the Nios II/f core requires that the core include at least one tightly-coupled instruction memory.

Data Cache

The data cache memory has the following characteristics:

- Direct-mapped cache implementation
- Configurable line size of 4, 16, or 32 bytes
- The data master port reads an entire cache line at a time from memory, and issues one read per clock cycle.
- Write-back
- Write-allocate (i.e., on a store instruction, a cache miss allocates the line for that address)
- Virtually-indexed, physically-tagged, when MMU present

The size of the tag field depends on the size of the cache memory and the physical address size. The size of the line field depends only on the size of the cache memory. The size of the offset field depends on the line size. Line sizes of 4, 16, and 32 bytes have offset widths of 2, 4, and 5 bits respectively. The maximum data byte address size is 31 bits in systems without an MMU present. In systems with an MMU, the maximum data byte address size is 32 bits and the tag field always includes all the bits of the PFN.

The data cache is optional. If the data cache is excluded from the core, the data master port can also be excluded.

The Nios II instruction set provides several different instructions to clear the data cache. There are two important questions to answer when determining the instruction to use. Do you need to consider the tag field when looking for a cache match? Do you need to write dirty cache lines back to memory before clearing? Table 5-8 shows the most appropriate instruction to use for each case.

Table 5-7. Data Cache Clearing Instructions

	Ignore Tag Field	Consider Tag Field
Write Dirty Lines	flushd	flushda
Don't Write Dirty Lines	initd	initda



The 4-byte line data cache implementation substitutes the `flushd` instruction for the `flushda` instruction and triggers an unimplemented instruction exception for the `initda` instruction. The 16-byte and 32-byte line data cache implementations fully support the `flushda` and `initda` instructions.



For more information regarding the Nios II instruction set, refer to the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*.

The Nios II/f core implements all the data cache bypass methods.



For information regarding the data cache bypass methods, refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

Mixing cached and uncached accesses to the same cache line can result in invalid data reads. For example, the following sequence of events causes cache incoherency.

1. The Nios II core writes data to cache, creating a dirty data cache line.
2. The Nios II core reads data from the same address, but bypasses the cache.

Avoid mixing cached and uncached accesses to the same cache line. If it is necessary to mix cached and uncached data accesses, flush the corresponding line of the data cache after completing the cached accesses and before performing the uncached accesses.

Bursting

When the data cache is enabled, you can enable bursting on the data master port. Consult the documentation for memory devices connected to the data master port to determine whether bursting will improve performance.

Tightly-Coupled Memory

The Nios II/f core provides optional tightly-coupled memory interfaces for both instructions and data. A Nios II/f core can use up to four each of instruction and data tightly-coupled memories. When a tightly-coupled memory interface is enabled, the Nios II core includes an additional memory interface master port. Each tightly-coupled memory interface must connect directly to exactly one memory slave port.

When tightly-coupled memory is present, the Nios II core decodes addresses internally to determine if requested instructions or data reside in tightly-coupled memory. If the address resides in tightly-coupled memory, the Nios II core fetches the instruction or data through the tightly-coupled memory interface. Software accesses tightly-coupled memory with the usual load and store instructions, such as `ldw` or `ldwio`.

Accessing tightly-coupled memory bypasses cache memory. The processor core functions as if cache were not present for the address span of the tightly-coupled memory. Instructions for managing cache, such as `initd` and `flushd`, do not affect the tightly-coupled memory, even if the instruction specifies an address in tightly-coupled memory.

When the MMU is present, tightly-coupled memories are always mapped into the kernel partition and can only be accessed in supervisor mode.

Memory Management Unit

The Nios II/f core provides options to improve the performance of the Nios II MMU.



For details on the MMU architecture, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

Micro Translation Lookaside Buffers

The translation lookaside buffer (TLB) consists of one main TLB stored in on-chip RAM and two separate micro TLBs (μ TLB) for instructions (μ ITLB) and data (μ DTLB) stored in LE-based registers.

The μ TLBs have a configurable number of entries and are fully associative. The default configuration has 6 μ DTLB entries and 4 μ ITLB entries. The hardware chooses the least-recently used μ TLB entry when loading a new entry.

The μ TLBs are not visible to software. They act as an inclusive cache of the main TLB. The processor firsts look for a hit in the μ TLB. If it misses, it then looks for a hit in the main TLB. If the main TLB misses, the processor takes an exception. If the main TLB hits, the TLB entry is copied into the μ TLB for future accesses.

The hardware automatically flushes the μ TLB on each TLB write operation and on a `wrc1` to the `tlbmisc` register in case the process identifier (PID) has changed.

Memory Protection Unit

The Nios II/f core provides options to improve the performance of the Nios II MPU. For details on the MPU architecture, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

Execution Pipeline

This section provides an overview of the pipeline behavior for the benefit of performance-critical applications. Designers can use this information to minimize unnecessary processor stalling. Most application programmers never need to analyze the performance of individual instructions.

The Nios II/f core employs a 6-stage pipeline. The pipeline stages are listed in [Table 5-8](#).

Table 5-8. Implementation Pipeline Stages for Nios II/f Core

Stage Letter	Stage Name
F	Fetch
D	Decode
E	Execute
M	Memory
A	Align
W	Writeback

Up to one instruction is dispatched and/or retired per cycle. Instructions are dispatched and retired in-order. Dynamic branch prediction is implemented using a 2-bit branch history table. The pipeline stalls for the following conditions:

- Multi-cycle instructions
- Avalon-MM instruction master port read accesses
- Avalon-MM data master port read/write accesses
- Data dependencies on long latency instructions (e.g., load, multiply, shift).

Pipeline Stalls

The pipeline is set up so that if a stage stalls, no new values enter that stage or any earlier stages. No “catching up” of pipeline stages is allowed, even if a pipeline stage is empty.

Only the A-stage and D-stage are allowed to create stalls.

The A-stage stall occurs if any of the following conditions occurs:

- An A-stage memory instruction is waiting for Avalon-MM data master requests to complete. Typically this happens when a load or store misses in the data cache, or a `flushd` instruction needs to write back a dirty line.
- An A-stage shift/rotate instruction is still performing its operation. This only occurs with the multi-cycle shift circuitry (i.e., when the hardware multiplier is not available).
- An A-stage divide instruction is still performing its operation. This only occurs when the optional divide circuitry is available.
- An A-stage multi-cycle custom instruction is asserting its stall signal. This only occurs if the design includes multi-cycle custom instructions.

The D-stage stall occurs if an instruction is trying to use the result of a late result instruction too early and no M-stage pipeline flush is active. The late result instructions are loads, shifts, rotates, `rdctl`, multiplies (if hardware multiply is supported), divides (if hardware divide is supported), and multi-cycle custom instructions (if present).

Branch Prediction

The Nios II/f core performs dynamic branch prediction to minimize the cycle penalty associated with taken branches.

Instruction Performance

All instructions take one or more cycles to execute. Some instructions have other penalties associated with their execution. Late result instructions have two cycles placed between them and an instruction that uses their result. Instructions that flush the pipeline cause up to three instructions after them to be cancelled. This creates a three-cycle penalty and an execution time of four cycles. Instructions that require Avalon-MM transfers are stalled until any required Avalon-MM transfers (up to one write and one read) are completed.

Execution performance for all instructions is shown in [Table 5-9](#).

Table 5-9. Instruction Execution Performance for Nios II/f Core 4byte/line data cache (Part 1 of 2)

Instruction	Cycles	Penalties
Normal ALU instructions (e.g., <code>add</code> , <code>cmplt</code>)	1	
Combinatorial custom instructions	1	
Multi-cycle custom instructions	> 1	Late result
Branch (correctly predicted, taken)	2	
Branch (correctly predicted, not taken)	1	
Branch (mis-predicted)	4	Pipeline flush
<code>trap</code> , <code>break</code> , <code>eret</code> , <code>bret</code> , <code>flushp</code> , <code>wrctl</code> ; illegal and unimplemented instructions	4	Pipeline flush
<code>call</code> , <code>jmp</code>	2	
<code>jmp</code> , <code>ret</code> , <code>callr</code>	3	
<code>rdctl</code>	1	Late result
<code>load</code> (without Avalon-MM transfer)	1	Late result
<code>load</code> (with Avalon-MM transfer)	> 1	Late result
<code>store</code> (without Avalon-MM transfer)	1	
<code>store</code> (with Avalon-MM transfer)	> 1	
<code>flushd</code> , <code>flushda</code> (without Avalon-MM transfer)	2	
<code>flushd</code> , <code>flushda</code> (with Avalon-MM transfer)	> 2	
<code>initd</code> , <code>initda</code>	2	
<code>flushi</code> , <code>initi</code>	4	
Multiply	(1)	Late result
Divide	(1)	Late result
Shift/rotate (with hardware multiply using embedded multipliers)	1	Late result
Shift/rotate (with hardware multiply using LE-based multipliers)	2	Late result

Table 5-9. Instruction Execution Performance for Nios II/f Core 4byte/line data cache (Part 2 of 2)

Instruction	Cycles	Penalties
Shift/rotate (without hardware multiply present)	1 - 32	Late result
All other instructions	1	

Note to Table 5-9:

(1) Depends on the hardware multiply or divide option. Refer to [Table 5-3 on page 5-5](#) for details.

Exception Handling

The Nios II/f core supports the following exception types:

- Hardware interrupt
- Software trap
- Illegal instruction
- Unimplemented instruction
- Supervisor-only instruction
- Supervisor-only instruction address
- Supervisor-only data address
- Misaligned data address
- Misaligned destination address
- Division error
- Fast TLB miss
- Double TLB miss
- TLB permission violation
- MPU region violation

JTAG Debug Module

The Nios II/f core supports the JTAG debug module to provide a JTAG interface to software debugging tools. The Nios II/f core supports an optional enhanced interface that allows real-time trace data to be routed out of the processor and stored in an external debug probe.



The Nios II MMU does not support the JTAG debug module trace.

Nios II/s Core

The Nios II/s standard core is designed for small core size. On-chip logic and memory resources are conserved at the expense of execution performance. The Nios II/s core uses approximately 20% less logic than the Nios II/f core, but execution performance also drops by roughly 40%. Altera designed the Nios II/s core with the following design goals in mind:

- Do not cripple performance for the sake of size.

- Remove hardware features that have the highest ratio of resource usage to performance impact.

The resulting core is optimal for cost-sensitive, medium-performance applications. This includes applications with large amounts of code and/or data, such as systems running an operating system in which performance is not the highest priority.

Overview

The Nios II/s core:

- Has an instruction cache, but no data cache
- Can access up to 2 Gbytes of external address space
- Supports optional tightly-coupled memory for instructions
- Employs a 5-stage pipeline
- Performs static branch prediction
- Provides hardware multiply, divide, and shift options to improve arithmetic performance
- Supports the addition of custom instructions
- Supports the JTAG debug module
- Supports optional JTAG debug module enhancements, including hardware breakpoints and real-time trace

The following sections discuss the noteworthy details of the Nios II/s core implementation. This document does not discuss low-level design issues or implementation details that do not affect Nios II hardware or software designers.

Arithmetic Logic Unit

The Nios II/s core provides several ALU options to improve the performance of multiply, divide, and shift operations.

Multiply and Divide Performance

The Nios II/s core provides the following hardware multiplier options:

- DSP Block—Includes DSP block multipliers available on the target device. This option is available only on Altera FPGAs that have DSP Blocks.
- Embedded Multipliers—Includes dedicated embedded multipliers available on the target device. This option is available only on Altera FPGAs that have embedded multipliers.
- Logic Elements—Includes hardware multipliers built from logic element (LE) resources.
- None—Does not include multiply hardware. In this case, multiply operations are emulated in software.

The Nios II/s core also provides a hardware divide option that includes LE-based divide circuitry in the ALU.

Including an ALU option improves the performance of one or more arithmetic instructions.



The performance of the embedded multipliers differ, depending on the target FPGA family.

Table 5-10 lists the details of the hardware multiply and divide options.

Table 5-10. Hardware Multiply and Divide Details for the Nios II/s Core

ALU Option	Hardware Details	Cycles per instruction	Supported Instructions
No hardware multiply or divide	Multiply and divide instructions generate an exception	–	None
LE-based multiplier	ALU includes 32 x 4-bit multiplier	11	mul, muli
Embedded multiplier on Stratix, Stratix II and Stratix III families	ALU includes 32 x 32-bit multiplier	3	mul, muli, mulxss, mulxsu, mulxuu
Embedded multiplier on Cyclone II and Cyclone III families	ALU includes 32 x 16-bit multiplier	5	mul, muli
Hardware divide	ALU includes multicycle divide circuit	4 – 66	div, divu

Shift and Rotate Performance

The performance of shift operations depends on the hardware multiply option. When a hardware multiplier is present, the ALU achieves shift and rotate operations in three or four clock cycles. Otherwise, the ALU includes dedicated shift circuitry that achieves one-bit-per-cycle shift and rotate performance. Refer to Table 5-13 on page 5-17 for details.

Memory Access

The Nios II/s core provides instruction cache, but no data cache. The instruction cache size is user-definable, between 512 bytes and 64 KBytes. The Nios II/s core can address up to 2 Gbyte of external memory. The Nios II architecture reserves the most-significant bit of data addresses for the bit-31 cache bypass method. In the Nios II/s core, bit 31 is always zero.



For information regarding data cache bypass methods, refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

Instruction and Data Master Ports

The instruction port on the Nios II/s core is optional. The instruction master port can be excluded, as long as the core includes at least one tightly-coupled instruction memory. The instruction master port is a pipelined Avalon-MM master port.

Support for pipelined Avalon-MM transfers minimizes the impact of synchronous memory with pipeline latency. The pipelined instruction master port can issue successive read requests before prior requests complete.

The data master port on the Nios II/s core is always present.

Instruction Cache

The instruction cache for the Nios II/s core is nearly identical to the instruction cache in the Nios II/f core. The instruction cache memory has the following characteristics:

- Direct-mapped cache implementation
- The instruction master port reads an entire cache line at a time from memory, and issues one read per clock cycle.
- Critical word first

Table 5-11 shows the instruction byte address fields.

Table 5-11. Instruction Byte Address Fields

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
tag													line											offset							

The size of the tag field depends on the size of the cache memory and the physical address size. The size of the line field depends only on the size of the cache memory. The offset field is always five bits (i.e., a 32-byte line). The maximum instruction byte address size is 31 bits.

The instruction cache is optional. However, excluding instruction cache from the Nios II/s core requires that the core include at least one tightly-coupled instruction memory.

Tightly-Coupled Memory

The Nios II/s core provides optional tightly-coupled memory interfaces for instructions. A Nios II/s core can use up to four tightly-coupled instruction memories. When a tightly-coupled memory interface is enabled, the Nios II core includes an additional memory interface master port. Each tightly-coupled memory interface must connect directly to exactly one memory slave port.

When tightly-coupled memory is present, the Nios II core decodes addresses internally to determine if requested instructions reside in tightly-coupled memory. If the address resides in tightly-coupled memory, the Nios II core fetches the instruction through the tightly-coupled memory interface. Software does not require awareness of whether code resides in tightly-coupled memory or not.

Accessing tightly-coupled memory bypasses cache memory. The processor core functions as if cache were not present for the address span of the tightly-coupled memory. Instructions for managing cache, such as `init_i` and `flush_i`, do not affect the tightly-coupled memory, even if the instruction specifies an address in tightly-coupled memory.

Execution Pipeline

This section provides an overview of the pipeline behavior for the benefit of performance-critical applications. Designers can use this information to minimize unnecessary processor stalling. Most application programmers never need to analyze the performance of individual instructions, and live happy lives without ever studying Table 5-12.

The Nios II/s core employs a 5-stage pipeline. The pipeline stages are listed in Table 5-12.

Table 5-12. Implementation Pipeline Stages for Nios II/s Core

Stage Letter	Stage Name
F	Fetch
D	Decode
E	Execute
M	Memory
W	Writeback

Up to one instruction is dispatched and/or retired per cycle. Instructions are dispatched and retired in-order. Static branch prediction is implemented using the branch offset direction; a negative offset (backward branch) is predicted as taken, and a positive offset (forward branch) is predicted as not-taken. The pipeline stalls for the following conditions:

- Multi-cycle instructions (e.g., shift/rotate without hardware multiply)
- Avalon-MM instruction master port read accesses
- Avalon-MM data master port read/write accesses
- Data dependencies on long latency instructions (e.g., load, multiply, shift operations)

Pipeline Stalls

The pipeline is set up so that if a stage stalls, no new values enter that stage or any earlier stages. No “catching up” of pipeline stages is allowed, even if a pipeline stage is empty.

Only the M-stage is allowed to create stalls.

The M-stage stall occurs if any of the following conditions occurs:

- An M-stage load/store instruction is waiting for Avalon-MM data master transfer to complete.
- An M-stage shift/rotate instruction is still performing its operation when using the multi-cycle shift circuitry (i.e., when the hardware multiplier is not available).
- An M-stage shift/rotate/multiply instruction is still performing its operation when using the hardware multiplier (which takes three cycles).
- An M-stage multi-cycle custom instruction is asserting its stall signal. This only occurs if the design includes multi-cycle custom instructions.

Branch Prediction

The Nios II/s core performs static branch prediction to minimize the cycle penalty associated with taken branches.

Instruction Performance

All instructions take one or more cycles to execute. Some instructions have other penalties associated with their execution. Instructions that flush the pipeline cause up to three instructions after them to be cancelled. This creates a three-cycle penalty and an execution time of four cycles. Instructions that require an Avalon-MM transfer are stalled until the transfer completes.

Execution performance for all instructions is shown in [Table 5-13](#).

Table 5-13. Instruction Execution Performance for Nios II/s Core

Instruction	Cycles	Penalties
Normal ALU instructions (e.g., <code>add</code> , <code>cmplt</code>)	1	
Combinatorial custom instructions	1	
Multi-cycle custom instructions	> 1	
Branch (correctly predicted taken)	2	
Branch (correctly predicted not taken)	1	
Branch (mispredicted)	4	Pipeline flush
<code>trap</code> , <code>break</code> , <code>eret</code> , <code>bret</code> , <code>flushp</code> , <code>wrctl</code> , <code>unimplemented</code>	4	Pipeline flush
<code>jmp</code> , <code>jmp_i</code> , <code>ret</code> , <code>call</code> , <code>callr</code>	4	Pipeline flush
<code>rdctl</code>	1	
<code>load</code> , <code>store</code>	> 1	
<code>flushi</code> , <code>initi</code>	4	
Multiply	(1)	
Divide	(1)	
Shift/rotate (with hardware multiply using embedded multipliers)	3	
Shift/rotate (with hardware multiply using LE-based multipliers)	4	
Shift/rotate (without hardware multiply present)	1 to 32	
All other instructions	1	

Note to [Table 5-13](#):

(1) Depends on the hardware multiply or divide option. Refer to [Table 5-10](#) on page 5-14 for details.

Exception Handling

The Nios II/s core supports the following exception types:

- Hardware interrupt
- Software trap
- Illegal instruction
- Unimplemented instruction

JTAG Debug Module

The Nios II/s core supports the JTAG debug module to provide a JTAG interface to software debugging tools. The Nios II/s core supports an optional enhanced interface that allows real-time trace data to be routed out of the processor and stored in an external debug probe.

Nios II/e Core

The Nios II/e economy core is designed to achieve the smallest possible core size. Altera designed the Nios II/e core with a singular design goal: reduce resource utilization any way possible, while still maintaining compatibility with the Nios II instruction set architecture. Hardware resources are conserved at the expense of execution performance. The Nios II/e core is roughly half the size of the Nios II/s core, but the execution performance is substantially lower.

The resulting core is optimal for cost-sensitive applications as well as applications that require simple control logic.

Overview

The Nios II/e core:

- Executes at most one instruction per six clock cycles
- Can access up to 2 Gbytes of external address space
- Supports the addition of custom instructions
- Supports the JTAG debug module
- Does not provide hardware support for potential unimplemented instructions
- Has no instruction cache or data cache
- Does not perform branch prediction

The following sections discuss the noteworthy details of the Nios II/e core implementation. This document does not discuss low-level design issues, or implementation details that do not affect Nios II hardware or software designers.

Arithmetic Logic Unit

The Nios II/e core does not provide hardware support for any of the potential unimplemented instructions. All unimplemented instructions are emulated in software.

The Nios II/e core employs dedicated shift circuitry to perform shift and rotate operations. The dedicated shift circuitry achieves one-bit-per-cycle shift and rotate operations.

Memory Access

The Nios II/e core does not provide instruction cache or data cache. All memory and peripheral accesses generate an Avalon-MM transfer. The Nios II/e core can address up to 2 Gbytes of external memory. The Nios II architecture reserves the most-significant bit of data addresses for the bit-31 cache bypass method. In the Nios II/e core, bit 31 is always zero.



For information regarding data cache bypass methods, refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

Instruction Execution Stages

This section provides an overview of the pipeline behavior as a means of estimating assembly execution time. Most application programmers never need to analyze the performance of individual instructions.

Instruction Performance

The Nios II/e core dispatches a single instruction at a time, and the processor waits for an instruction to complete before fetching and dispatching the next instruction. Because each instruction completes before the next instruction is dispatched, branch prediction is not necessary. This greatly simplifies the consideration of processor stalls. Maximum performance is one instruction per six clock cycles. To achieve six cycles, the Avalon-MM instruction master port must fetch an instruction in one clock cycle. A stall on the Avalon-MM instruction master port directly extends the execution time of the instruction.

Execution performance for all instructions is shown in [Table 5-14](#).

Table 5-14. Instruction Execution Performance for Nios II/e Core

Instruction	Cycles
Normal ALU instructions (e.g., add, cmlt)	6
branch, jmp, jmp, ret, call, callr	6
trap, break, eret, bret, flushp, wrctl, rdctl, unimplemented	6
load word	6 + Duration of Avalon-MM read transfer
load halfword	9 + Duration of Avalon-MM read transfer
load byte	10 + Duration of Avalon-MM read transfer
store	6 + Duration of Avalon-MM write transfer
Shift, rotate	7 to 38
All other instructions	6
Combinatorial custom instructions	6
Multi-cycle custom instructions	≥6

Exception Handling

The Nios II/e core supports the following exception types:

- Hardware interrupt
- Software trap
- Illegal instruction
- Unimplemented instruction

JTAG Debug Module

The Nios II/e core supports the JTAG debug module to provide a JTAG interface to software debugging tools. The JTAG debug module on the Nios II/e core does not support hardware breakpoints or trace.

Referenced Documents

This chapter references the following documents:

- *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*
- *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*
- *Programming Model* chapter of the *Nios II Processor Reference Handbook*

Document Revision History

Table 5-15 shows the revision history for this document.

Table 5-15. Document Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Maintenance release.	—
November 2008 v8.1.0	Maintenance release.	—
May 2008 v8.0.0	Added text for MMU and MPU.	Added MMU and MPU
October 2007 v7.2.0	Added <code>jmp_i</code> instruction to tables.	—
May 2007 v7.1.0	<ul style="list-style-type: none"> ■ Added table of contents to Introduction section. ■ Added Referenced Documents section. 	—
March 2007 v7.0.0	Add preliminary Cyclone III device family support	Cyclone III device family
November 2006 v6.1.0	Add preliminary Stratix III device family support	Stratix III device family
May 2006 v6.0.0	Performance for <code>flushi</code> and <code>initi</code> instructions changes from 1 to 4 cycles for Nios II/s and Nios II/f cores.	—
October 2005 v5.1.0	Maintenance release.	—

Table 5-15. Document Revision History (Part 2 of 2)

Date & Document Version	Changes Made	Summary of Changes
May 2005 v5.0.0	Updates to Nios II/f and Nios II/s cores. Added tightly-coupled memory and new data cache options. Corrected cycle counts for shift/rotate operations.	—
December 2004 v1.2	Updates to Multiply and Divide Performance section for Nios II/f and Nios II/s cores.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

Introduction

Each release of the Nios® II Embedded Design Suite (EDS) introduces improvements to the Nios II processor, the software development tools, or both. This document catalogs the history of revisions to the Nios II processor; it does not track revisions to development tools, such as the Nios II integrated development environment (IDE). This chapter contains the following sections:

- “Nios II Versions” on page 6-1
- “Architecture Revisions” on page 6-2
- “Core Revisions” on page 6-3
- “JTAG Debug Module Revisions” on page 6-6

Improvements to the Nios II processor might affect:

- Features of the Nios II architecture—An example of an architecture revision is adding instructions to support floating-point arithmetic.
- Implementation of a specific Nios II core—An example of a core revision is increasing the maximum possible size of the data cache memory for the Nios II/f core.
- Features of the JTAG debug module—An example of a JTAG debug module revision is adding an additional trigger input to the JTAG debug module, allowing it to halt processor execution on a new type of trigger event.

Altera implements Nios II revisions such that code written for an existing Nios II core also works on future revisions of the same core.

Nios II Versions

The number for any version of the Nios II processor is determined by the version of the Nios II EDS. For example, in the Nios II EDS version 8.0, all Nios II cores are also version 8.0.

Table 6-1 lists the version numbers of all releases of the Nios II processor.

Table 6-1. Nios II Processor Revision History (Part 1 of 2)

Version	Release Date	Notes
8.0	May 2008	<ul style="list-style-type: none"> ■ Added an optional memory management unit (MMU). ■ Added an optional memory protection unit (MPU). ■ Added advanced exception checking. ■ Added the <code>initda</code> instruction.
7.2	October 2007	Added the <code>jmp<i>i</i></code> instruction.
7.1	May 2007	No changes.
7.0	March 2007	No changes.

Table 6-1. Nios II Processor Revision History (Part 2 of 2)

Version	Release Date	Notes
6.1	November 2006	No changes.
6.0	May 2006	The name Nios II Development Kit describing the software development tools changed to Nios II Embedded Design Suite.
5.1 SP1	January 2006	Bug fix for Nios II/f core.
5.1	October 2005	No changes.
5.0	May 2005	<ul style="list-style-type: none"> ■ Changed version nomenclature. Altera® now aligns the Nios II processor version with Altera's Quartus II® software version. ■ Memory structure enhancements: <ol style="list-style-type: none"> (1) Added tightly-coupled memory. (2) Made data cache line size configurable. (3) Made cache optional in Nios II/f and Nios II/s cores. ■ Support for HardCopy® devices.
1.1	December 2004	<ul style="list-style-type: none"> ■ Minor enhancements to the architecture: Added <code>cpuid</code> control register, and updated the <code>break</code> instruction. ■ Increased user control of multiply and shift hardware in the arithmetic logic unit (ALU) for Nios II/s and Nios II/f cores. ■ Minor bug fixes.
1.01	September 2004	■ Minor bug fixes.
1.0	May 2004	Initial release of the Nios II processor.

Architecture Revisions

Architecture revisions augment the fundamental capabilities of the Nios II architecture, and affect all Nios II cores. A change in the architecture mandates a revision to all Nios II cores to accommodate the new architectural enhancement. For example, when Altera adds a new instruction to the instruction set, Altera consequently must update all Nios II cores to recognize the new instruction. [Table 6-2](#) lists revisions to the Nios II architecture.

Table 6-2. Nios II Architecture Revisions (Part 1 of 2)

Version	Release Date	Notes
8.0	May 2008	<ul style="list-style-type: none"> ■ Added an optional MMU. ■ Added an optional MPU. ■ Added advanced exception checking to detect division errors, illegal instructions, misaligned memory accesses, and provide extra exception information. ■ Added the <code>initda</code> instruction.
7.2	October 2007	Added the <code>jmp</code> instruction.
7.1	May 2007	No changes.
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	Added optional <code>cpu_resetrequest</code> and <code>cpu_resettaken</code> signals to all processor cores.

Table 6-2. Nios II Architecture Revisions (Part 2 of 2)

Version	Release Date	Notes
5.1	October 2005	No changes.
5.0	May 2005	Added the <code>flushda</code> instruction.
1.1	December 2004	<ul style="list-style-type: none"> ■ Added <code>cpuid</code> control register. ■ Updated <code>break</code> instruction specification to accept an immediate argument for use by debugging tools.
1.01	September 2004	No changes.
1.0	May 2004	Initial release of the Nios II processor architecture.

Core Revisions

Core revisions introduce changes to an existing Nios II core. Core revisions most commonly fix identified bugs, or add support for an architecture revision. Not every Nios II core is revised with every release of the Nios II architecture.

Nios II/f Core

Table 6-3 lists revisions to the Nios II/f core.

Table 6-3. Nios II/f Core Revisions (Part 1 of 2)

Version	Release Date	Notes
8.0	May 2008	<ul style="list-style-type: none"> ■ Implemented the optional MMU. ■ Implemented the optional MPU. ■ Implemented advanced exception checking. ■ Implemented the <code>initda</code> instruction.
7.2	October 2007	Implemented the <code>jmp_i</code> instruction.
7.1	May 2007	No changes.
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	Cycle count for <code>flushi</code> and <code>initi</code> instructions changes from 1 to 4 cycles.
5.1 SP1	January 2006	<p>Bug Fix:</p> <p>Back-to-back store instructions can cause memory corruption to the stored data. If the first store is not to the last word of a cache line and the second store is to the last word of the line, memory corruption occurs.</p>
5.1	October 2005	No changes.

Table 6-3. Nios II/f Core Revisions (Part 2 of 2)

Version	Release Date	Notes
5.0	May 2005	<ul style="list-style-type: none"> ■ Added optional tightly-coupled memory ports. Designers can add zero to four tightly-coupled instruction master ports, and zero to four tightly-coupled data master ports. ■ Made the data cache line size configurable. Designers can configure the data cache with the following line sizes: 4, 16, or 32 bytes. Previously, the data cache line size was fixed at 4 bytes. ■ Made instruction and data caches optional (previously, cache memories were always present). If the instruction cache is not present, the Nios II core does not have an instruction master port, and must use a tightly-coupled instruction memory. ■ Full support for HardCopy devices (previous versions required a work around to support HardCopy devices).
1.1	December 2004	<ul style="list-style-type: none"> ■ Added user-configurable options affecting multiply and shift operations. Now designers can choose one of three options: <ol style="list-style-type: none"> (1) Use embedded multiplier resources available in the target device family (previously available). (2) Use logic elements to implement multiply and shift hardware (new option). (3) Omit multiply hardware. Shift operations take one cycle per bit shifted; multiply operations are emulated in software (new option). ■ Added <code>cpuid</code> control register. ■ Bug Fix: Interrupts that were disabled by <code>wrctl ienable</code> remained enabled for one clock cycle following the <code>wrctl</code> instruction. Now the instruction following such a <code>wrctl</code> cannot be interrupted.
1.01	September 2004	<ul style="list-style-type: none"> ■ Bug Fixes: <ol style="list-style-type: none"> (1) When a store to memory is followed immediately in the pipeline by a load from the same memory location, and the memory location is held in the data cache, the load may return invalid data. This situation can occur in C code compiled with optimization off (-O0). (2) The SOPC Builder top-level system module included an extra, unnecessary output port for systems with very small address spaces.
1.0	May 2004	Initial release of the Nios II/f core.

Nios II/s Core

Table 6-4 lists revisions to the Nios II/s core.

Table 6-4. Nios II/s Core Revisions (Part 1 of 2)

Version	Release Date	Notes
8.0	May 2008	Implemented the illegal instruction exception.
7.2	October 2007	Implemented the <code>jmp</code> instruction.
7.1	May 2007	No changes.
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	Cycle count for <code>flushi</code> and <code>initi</code> instructions changes from 1 to 4 cycles.

Table 6-4. Nios II/s Core Revisions (Part 2 of 2)

Version	Release Date	Notes
5.1	October 2005	No changes.
5.0	May 2005	<ul style="list-style-type: none"> ■ Added optional tightly-coupled memory ports. Designers can add zero to four tightly-coupled instruction master ports. ■ Made instruction cache optional (previously instruction cache was always present). If the instruction cache is not present, the Nios II core does not have an instruction master port, and must use a tightly-coupled instruction memory. ■ Full support for HardCopy devices (previous versions required a work around to support HardCopy devices).
1.1	December 2004	<ul style="list-style-type: none"> ■ Added user-configurable options affecting multiply and shift operations. Now designers can choose one of three options: <ol style="list-style-type: none"> (1) Use embedded multiplier resources available in the target device family (previously available). (2) Use logic elements to implement multiply and shift hardware (new option). (3) Omit multiply hardware. Shift operations take one cycle per bit shifted; multiply operations are emulated in software (new option). ■ Added user-configurable option to include divide hardware in the ALU. Previously this option was available for only the Nios II/f core. ■ Added <code>cpuid</code> control register.
1.01	September 2004	<p>Bug fix:</p> <p>The SOPC Builder top-level system module included an extra, unnecessary output port for systems with very small address spaces.</p>
1.0	May 2004	Initial release of the Nios II/s core.

Nios II/e Core

Table 6-5 lists revisions to the Nios II/e core.

Table 6-5. Nios II/e Core Revisions

Version	Release Date	Notes
8.0	May 2008	Implemented the illegal instruction exception.
7.2	October 2007	Implemented the <code>jmp_i</code> instruction.
7.1	May 2007	No changes.
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	No changes.
5.1	October 2005	No changes.
5.0	May 2005	Full support for HardCopy devices (previous versions required a work around to support HardCopy devices).
1.1	December 2004	Added <code>cpuid</code> control register.
1.01	September 2004	<p>Bug fix:</p> <p>The SOPC Builder top-level system module included an extra, unnecessary output port for systems with very small address spaces.</p>
1.0	May 2004	Initial release of the Nios II/e core.

JTAG Debug Module Revisions

JTAG debug module revisions augment the debug capabilities of the Nios II processor, or fix bugs isolated within the JTAG debug module logic.

Table 6–6 lists revisions to the JTAG debug module.

Table 6–6. JTAG Debug Module Revisions

Version	Release Date	Notes
8.0	May 2008	No changes.
7.2	October 2007	No changes.
7.1	May 2007	No changes.
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	No changes.
5.1	October 2005	No changes.
5.0	May 2005	Full support for HardCopy devices (previous versions of the JTAG debug module did not support HardCopy devices).
1.1	December 2004	Bug fix: When using the Nios II/s and Nios II/f cores, hardware breakpoints may have falsely triggered when placed on the instruction sequentially following a <code>jmp</code> , <code>trap</code> , or any branch instruction.
1.01	September 2004	<ul style="list-style-type: none"> ■ Feature enhancements: <ol style="list-style-type: none"> (1) Added the ability to trigger based on the instruction address. Uses include triggering trace control (trace on/off), sequential triggers, and trigger in/out signal generation. (2) Enhanced trace collection such that collection can be stopped when the trace buffer is full without halting the Nios II processor. (3) Armed triggers – Enhanced trigger logic to support two levels of triggers, or "armed triggers"; enabling the use of "Event A then event B" trigger definitions. ■ Bug fixes: <ol style="list-style-type: none"> (1) On the Nios II/s core, trace data sometimes recorded incorrect addresses during interrupt processing. (2) Under certain circumstances, captured trace data appeared to start earlier or later than the desired trigger location. (3) During debugging, the processor would hang if a hardware breakpoint and an interrupt occurred simultaneously.
1.0	May 2004	Initial release of the JTAG debug module.

Referenced Documents

This chapter references no other documents.

Document Revision History

Table 6–7 shows the revision history for this document.

Table 6-7. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Maintenance release.	—
November 2008 v8.1.0	Maintenance release.	—
May 2008 v8.0.0	<ul style="list-style-type: none"> ■ Added MMU information. ■ Added MPU information. ■ Added advanced exception checking information. ■ Added <code>initda</code> instruction information. 	Added MMU, MPU, advanced exception checking, and <code>initda</code> instruction.
October 2007 v7.2.0	<ul style="list-style-type: none"> ■ Added <code>jmp_i</code> instruction information. ■ Added exception handling information. 	Added <code>jmp_i</code> instruction
May 2007 v7.1.0	<ul style="list-style-type: none"> ■ Updated tables to reflect no changes to cores. ■ Added table of contents to Introduction section. ■ Added Referenced Documents section. 	—
March 2007 v7.0.0	Updated tables to reflect no changes to cores.	—
November 2006 v6.1.0	Updated tables to reflect no changes to cores.	—
May 2006 v6.0.0	Updates for Nios II cores version 6.0.	—
October 2005 v5.1.0	Updates for Nios II cores version 5.1.	—
May 2005 v5.0.0	Updates for Nios II cores version 5.0.	—
December 2004 v1.1	Updates for Nios II cores version 1.1.	—
September 2004 v1.0	Initial release.	—

This chapter describes the Application Binary Interface (ABI) for the Nios® II processor. The ABI describes:

- How data is arranged in memory
- Behavior and structure of the stack
- Function calling conventions

This chapter contains the following sections:

- “Data Types” on page 7-1
- “Memory Alignment” on page 7-1
- “Register Usage” on page 7-2
- “Stacks” on page 7-3
- “Arguments and Return Values” on page 7-6
- “Relocation” on page 7-8

Data Types

Table 7-1 shows the size and representation of the C/C++ data types for the Nios II processor.

Table 7-1. Representation of Data Types

Type	Size (Bytes)	Representation
char, signed char	1	two's complement (ASCII)
unsigned char	1	binary (ASCII)
short, signed short	2	two's complement
unsigned short	2	binary
int, signed int	4	two's complement
unsigned int	4	binary
long, signed long	4	two's complement
unsigned long	4	binary
float	4	IEEE
double	8	IEEE
pointer	4	binary
long long	8	two's complement
unsigned long long	8	binary

Memory Alignment

Contents in memory are aligned as follows:

- A function must be aligned to a minimum of 32-bit boundary.
- The minimum alignment of a data element is its natural size. A data element larger than 32 bits need only be aligned to a 32-bit boundary.
- Structures, unions, and strings must be aligned to a minimum of 32 bits.
- Bit fields inside structures are always 32-bit aligned.

Register Usage

The ABI adds additional usage conventions to the Nios II register file defined in the *Programming Model* chapter of the *Nios II Processor Reference Handbook*. The ABI uses the registers as shown in [Table 7-2](#).

Table 7-2. Nios II ABI Register Usage (Part 1 of 2)

Register	Name	Used by Compiler	Callee Saved (1)	Normal Usage
r0	zero	✓		0x00000000
r1	at			Assembler temporary
r2		✓		Return value (least-significant 32 bits)
r3		✓		Return value (most-significant 32 bits)
r4		✓		Register arguments (first 32 bits)
r5		✓		Register arguments (second 32 bits)
r6		✓		Register arguments (third 32 bits)
r7		✓		Register arguments (fourth 32 bits)
r8		✓		Caller-saved general-purpose registers
r9		✓		
r10		✓		
r11		✓		
r12		✓		
r13		✓		
r14		✓		
r15		✓		
r16		✓	✓	Callee-saved general-purpose registers
r17		✓	✓	
r18		✓	✓	
r19		✓	✓	
r20		✓	✓	
r21		✓	✓	
r22		✓	✓	
r23		✓	✓	
r24	et			Exception temporary
r25	bt			Break temporary
r26	gp	✓		Global pointer
r27	sp	✓		Stack pointer

Table 7-2. Nios II ABI Register Usage (Part 2 of 2)

Register	Name	Used by Compiler	Callee Saved (1)	Normal Usage
r28	fp	✓		Frame pointer (2)
r29	ea			Exception return address
r30	ba			Break return address
r31	ra	✓		Return address

Notes to Table 7-2:

- (1) A function can use one of these registers if it saves it first. The function must restore the register's original value before exiting.
- (2) If the frame pointer is not used, the register is available as a temporary register. Refer to "Frame Pointer Elimination" on page 7-4.

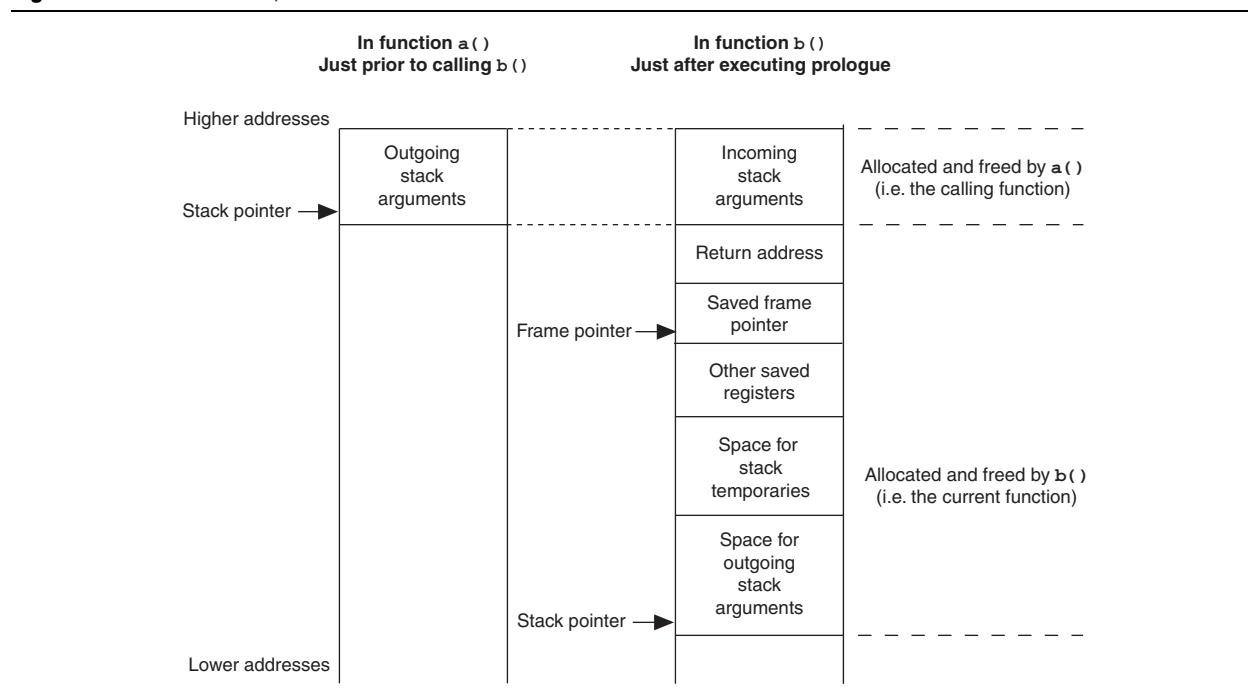
The endianness of values greater than 8 bits is little endian. The upper 8 bits of a value are stored at the higher byte address.

Stacks

The stack grows downward (i.e. towards lower addresses). The stack pointer points to the last used slot. The frame pointer points to the saved frame pointer near the top of the stack frame.

Figure 7-1 shows an example of the structure of a current frame. In this case, function a () calls function b (), and the stack is shown before the call and after the prologue in the called function has completed.

Figure 7-1. Stack Pointer, Frame Pointer and the Current Frame



Each section of the current frame is aligned to a 32-bit boundary. The ABI requires the stack pointer be 32-bit aligned at all times.

Frame Pointer Elimination

The frame pointer is provided for debugger support. If you are not using a debugger, you can optimize your code by eliminating the frame pointer, using the `-fomit-frame-pointer` compiler option. When the frame pointer is eliminated, register `fp` is available as a temporary register.

Call Saved Registers

The compiler is responsible for saving registers that need to be saved in a function. If there are any such registers, they are saved on the stack, from high to low addresses, in the following order: `ra`, `fp`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`, `r16`, `r17`, `r18`, `r19`, `r20`, `r21`, `r22`, `r23`, `r24`, `r25`, `gp`, and `sp`. Stack space is not allocated for registers that are not saved.

Further Examples of Stacks

There are a number of special cases for stack layout, which are described in this section.

Stack Frame for a Function With `alloca()`

The Nios II stack frame implementation provides support for the `alloca()` function, defined in the Berkeley Software Distribution (BSD) extension to C, and implemented by the gcc compiler. Figure 7-2 depicts what the frame looks like after `alloca()` is called. The space allocated by `alloca()` replaces the outgoing arguments and the outgoing arguments get new space allocated at the bottom of the frame.


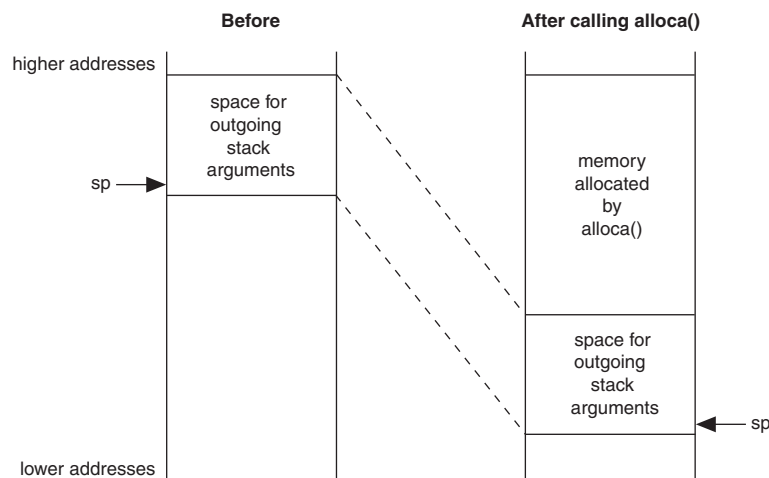
 The Nios II C/C++ compiler maintains a frame pointer for any function that calls `alloca()`, even if `-fomit-frame-pointer` is specified.

Figure 7-2. Stack Frame after Calling `alloca()`

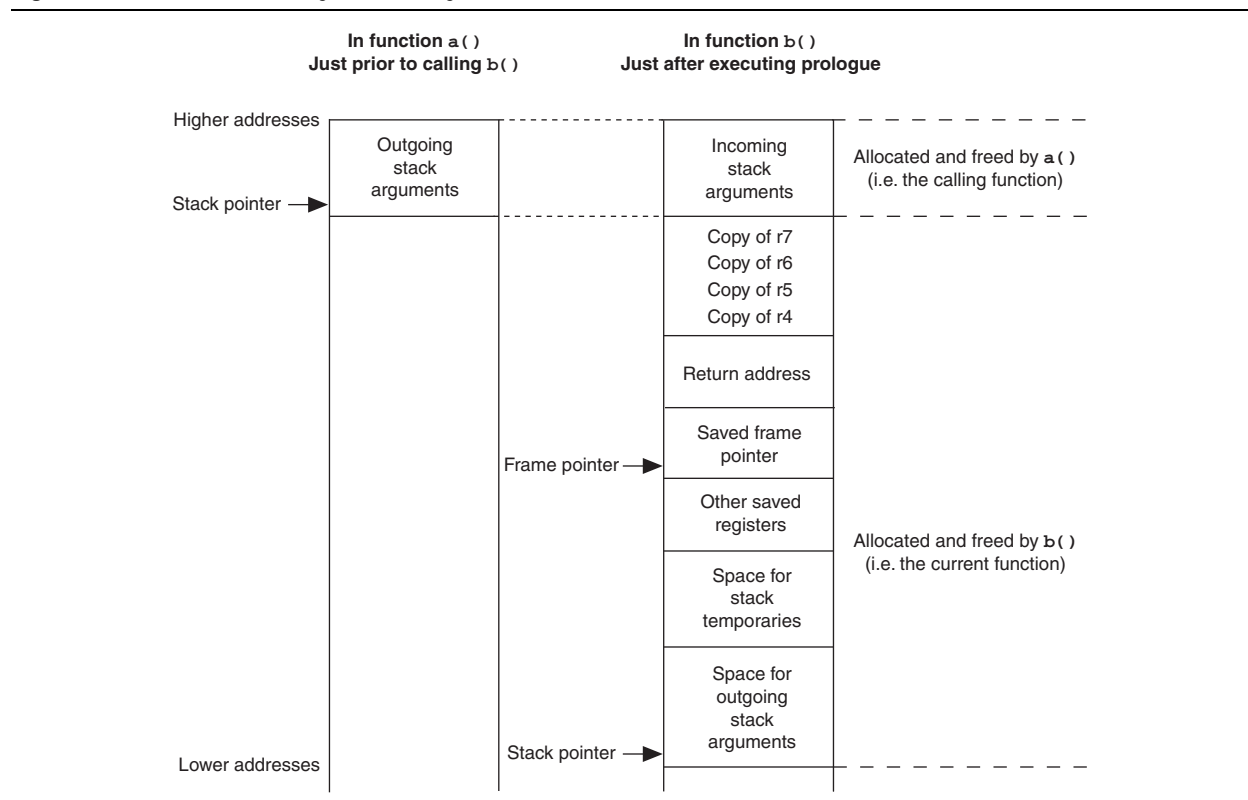


Stack Frame for a Function with Variable Arguments

Functions that take variable arguments (`varargs`) still have their first 16 bytes of arguments arriving in registers `r4` through `r7`, just like other functions.

In order for `varargs` to work, functions that take variable arguments allocate 16 extra bytes of storage on the stack. They copy to the stack the first 16 bytes of their arguments from registers `r4` through `r7` as shown in [Figure 7-3](#).

Figure 7-3. Stack Frame Using Variable Arguments



Stack Frame for a Function with Structures Passed By Value

Functions that take `struct` value arguments still have their first 16 bytes of arguments arriving in registers `r4` through `r7`, just like other functions.

If part of a structure is passed using registers, the function might need to copy the register contents back to the stack. This operation is similar to that required in the variable arguments case as shown in [Figure 7-3](#).

Function Prologues

The Nios II C/C++ compiler generates function prologues that allocate the stack frame of a function for storage of stack temporaries and outgoing arguments. In addition, each prologue is responsible for saving the state of the calling function. This entails saving certain registers on the stack. These registers, the callee-saved registers, are listed in [Table 7-2 on page 7-2](#). A function prologue is required to save a callee-saved register only if the function uses the register.

Given the function prologue algorithm, when doing a back trace, a debugger can disassemble instructions and reconstruct the processor state of the calling function.



An even better way to find out what the prologue has done is to use information stored in the DWARF2 debugging fields of the executable and linkable format (.elf) file.

The instructions found in a Nios II function prologue perform the following tasks:

- Adjust the stack pointer (to allocate the frame)
- Store registers to the frame
- Set the frame pointer to the location of the saved frame pointer

Example 7-1 shows a function prologue.

Example 7-1. A function prologue

```
/* Adjust the stack pointer */
addi    sp, sp, -16    /* make a 16-byte frame */

/* Store registers to the frame */
stw     ra, 12(sp)    /* store the return address */
stw     fp, 8(sp)     /* store the frame pointer*/
stw     r16, 4(sp)    /* store callee-saved register */
stw     r17, 0(sp)    /* store callee-saved register */

/* Set the new frame pointer */
addi    fp, sp, 8
```

Prologue Variations

The following variations can occur in a prologue:

- If the function's frame size is greater than 32,767 bytes, extra temporary registers are used in the calculation of the new stack pointer as well as for the offsets of where to store callee-saved registers. The extra registers are needed because of the maximum size of immediate values allowed by the Nios II processor.
- If the frame pointer is not in use, the final instruction, recalculating the frame pointer, is not generated.
- If variable arguments are used, extra instructions store the argument registers on the stack.
- If the compiler designates the function as a leaf function, the return address is not saved.
- If optimizations are on, especially instruction scheduling, the order of the instructions might change and become interlaced with instructions located after the prologue.

Arguments and Return Values

This section discusses the details of passing arguments to functions and returning values from functions.

Arguments

The first 16 bytes to a function are passed in registers r4 through r7. The arguments are passed as if a structure containing the types of the arguments were constructed, and the first 16 bytes of the structure are located in r4 through r7.

A simple example:

```
int function (int a, int b);
```

The equivalent structure representing the arguments is:

```
struct { int a; int b; };
```

The first 16 bytes of the struct are assigned to r4 through r7. Therefore r4 is assigned the value of *a* and r5 the value of *b*.

The first 16 bytes to a function taking variable arguments are passed the same way as a function not taking variable arguments. The called function must clean up the stack as necessary to support the variable arguments. Refer to [“Stack Frame for a Function with Variable Arguments”](#) on page 7-4.

Return Values

Return values of types up to 8 bytes are returned in r2 and r3. For return values greater than 8 bytes, the caller must allocate memory for the result and must pass the address of the result memory as a hidden zero argument.

The hidden zero argument is best explained through an example.

Example 7-2. Returned struct

```
/* b() computes a structure-type result and returns it */  
STRUCT b(int i, int j)  
{  
    ...  
    return result;  
}  
  
void a(...)  
{  
    ...  
    value = b(i, j);  
}
```

In [Example 7-2](#), if the result type is no larger than 8 bytes, `b()` returns its result in r2 and r3.

If the return type is larger than 8 bytes, the Nios II C/C++ compiler treats this program as if `a()` had passed a pointer to `b()`. [Example 7-3](#) shows how the Nios II C/C++ compiler sees the code in [Example 7-2](#).

Example 7-3. Returned struct is Larger than 8 Bytes

```

void b(STRUCT *p_result, int i, int j)
{
    ...
    *p_result = result;
}

void a(...)
{
    STRUCT value;
    ...
    b(*value, i, j);
}

```

Relocation

In a Nios II object file, each relocatable address reference possesses a relocation type. The relocation type specifies how to calculate the relocated address. Table 7-3 lists the calculation for address relocation for each Nios II relocation type. The bit mask specifies where the address is found in the instruction.

Table 7-3. Nios II Relocation Calculation (Part 1 of 2)

Name	Value	Overflow check (1)	Relocated Address R (2)	Bit Mask M	Bit Shift B
R_NIOS2_NONE	0	n/a	None	n/a	n/a
R_NIOS2_S16	1	Yes	S + A	0x003FFFC0	6
R_NIOS2_U16	2	Yes	S + A	0x003FFFC0	6
R_NIOS2_PCREL16	3	Yes	((S + A) - 4) - PC	0x003FFFC0	6
R_NIOS2_CALL26	4	No	(S + A) >> 2	0xFFFFFC0	6
R_NIOS2_IMM5	5	Yes	(S + A) & 0x1F	0x000007C0	6
R_NIOS2_CACHE_OPX	6	Yes	(S + A) & 0x1F	0x07C00000	22
R_NIOS2_IMM6	7	Yes	(S + A) & 0x3F	0x00000FC0	6
R_NIOS2_IMM8	8	Yes	(S + A) & 0xFF	0x00003FC0	6
R_NIOS2_HI16	9	No	((S + A) >> 16) & 0xFFFF	0x003FFFC0	6
R_NIOS2_LO16	10	No	(S + A) & 0xFFFF	0x003FFFC0	6
R_NIOS2_HIADJ16	11	No	(((S+A) >> 16) & 0xFFFF) + (((S+A) >> 15) & 0x1) & 0xFFFF	0x003FFFC0	6
R_NIOS2_BFD_RELOC_32	12	No	S + A	0xFFFFFFFF	0
R_NIOS2_BFD_RELOC_16	13	Yes	(S + A) & 0xFFFF	0x0000FFFF	0
R_NIOS2_BFD_RELOC_8	14	Yes	(S + A) & 0xFF	0x000000FF	0
R_NIOS2_GPREL	15	No	(S + A - GP) & 0xFFFF	0x003FFFC0	6
R_NIOS2_GNU_VTINHERIT	16	n/a	None	n/a	n/a
R_NIOS2_GNU_VTENTRY	17	n/a	None	n/a	n/a
R_NIOS2_UJMP	18	No	((S + A) >> 16) & 0xFFFF, (S + A + 4) & 0xFFFF	0x003FFFC0	6

Table 7-3. Nios II Relocation Calculation (Part 2 of 2)

Name	Value	Overflow check (1)	Relocated Address R (2)	Bit Mask M	Bit Shift B
R_NIOS2_CJMP	19	No	$((S + A) \gg 16) \& 0xFFFF,$ $(S + A + 4) \& 0xFFFF$	0x003FFFC0	6
R_NIOS2_CALLR	20	No	$((S + A) \gg 16) \& 0xFFFF)$ $(S + A + 4) \& 0xFFFF$	0x003FFFC0	6
R_NIOS2_ALIGN	21	n/a	None	n/a	n/a
R_NIOS2_ILLEGAL	22	n/a	None	n/a	n/a

Notes to Table 7-3:

- (1) For relocation types where no overflow check is performed, the relocated address is truncated to fit the instruction.
- (2) S: Symbol address, A: Addend, PC: Program counter, GP: Global pointer

With the information in Table 7-3, any Nios II instruction can be relocated by manipulating it as an unsigned 32-bit integer, as follows:

$$Xr = ((R \ll B) \& M \mid (X \& \sim M));$$

where:

- R is the relocated address, calculated as shown in Table 7-3
- B is the bit shift shown in Table 7-3
- M is the bit mask shown in Table 7-3
- X is the original instruction
- Xr is the relocated instruction

Validated Relocation Types

The Nios II C/C++ compiler generates and uses a subset of the available relocation types. The following five types are used frequently and have been thoroughly validated:

- R_NIOS2_HIADJ16
- R_NIOS2_LO16
- R_NIOS2_CALL26
- R_NIOS2_GPREL
- R_NIOS2_BFD_RELOC32

Other relocation types are not supported.

Referenced Documents

This chapter references the following documents:

- *Programming Model* chapter of the *Nios II Processor Reference Handbook*

Document Revision History

Table 7-4 shows the revision history for this document.

Table 7-4. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Maintenance release.	—
November 2008 v8.1.0	Maintenance release.	—
May 2008 v8.0.0	Frame pointer description updated Relocation table added	Frame pointer implementation redefined
October 2007 v7.2.0	Maintenance release.	—
May 2007 v7.1.0	<ul style="list-style-type: none"> ■ Added table of contents to Introduction section. ■ Added Referenced Documents section. 	—
March 2007 v7.0.0	Maintenance release.	—
November 2006 v6.1.0	Maintenance release.	—
May 2006 v6.0.0	Maintenance release.	—
October 2005 v5.1.0	Maintenance release.	—
May 2005 v5.0.0	Maintenance release.	—
May 2004 v1.0	Initial release.	—

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

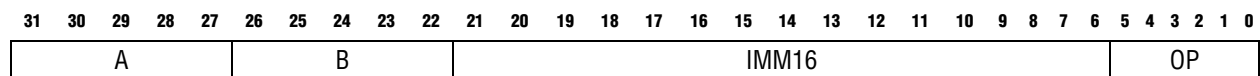
The defining characteristic of the I-type instruction-word format is that it contains an immediate value embedded within the instruction word. I-type instructions words contain:

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

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

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

The I-type instruction format is:



R-Type

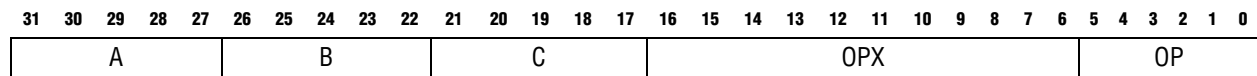
The defining characteristic of the R-type instruction-word format is that all arguments and results are specified as registers. R-type instructions contain:

- 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 as `cmpeq` and `cmplt`; the custom instruction; and other operations that need only register operands.

The R-type instruction format is:



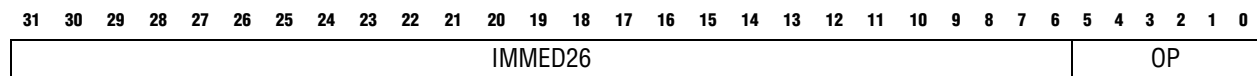
J-Type

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:



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)

OP	Instruction	OP	Instruction	OP	Instruction	OP	Instruction
0x00	<code>call</code>	0x10	<code>cmplti</code>	0x20	<code>cmpeqi</code>	0x30	<code>cmpltui</code>
0x01	<code>jmp<i>i</i></code>	0x11		0x21		0x31	
0x02		0x12		0x22		0x32	<code>custom</code>
0x03	<code>ldbu</code>	0x13	<code>initda</code>	0x23	<code>ldbuio</code>	0x33	<code>initd</code>
0x04	<code>addi</code>	0x14	<code>ori</code>	0x24	<code>muli</code>	0x34	<code>orhi</code>
0x05	<code>stb</code>	0x15	<code>stw</code>	0x25	<code>stbio</code>	0x35	<code>stwio</code>
0x06	<code>br</code>	0x16	<code>blt</code>	0x26	<code>beq</code>	0x36	<code>bltu</code>
0x07	<code>ldb</code>	0x17	<code>ldw</code>	0x27	<code>ldbio</code>	0x37	<code>ldwio</code>
0x08	<code>cmpgei</code>	0x18	<code>cmpnei</code>	0x28	<code>cmpgeui</code>	0x38	
0x09		0x19		0x29		0x39	
0x0A		0x1A		0x2A		0x3A	R-type

Table 8-1. OP Encodings (Part 2 of 2)

OP	Instruction	OP	Instruction	OP	Instruction	OP	Instruction
0x0B	ldhu	0x1B	flushda	0x2B	ldhuio	0x3B	flushd
0x0C	andi	0x1C	xori	0x2C	andhi	0x3C	xorhi
0x0D	sth	0x1D		0x2D	sthio	0x3D	
0x0E	bge	0x1E	bne	0x2E	bgeu	0x3E	
0x0F	ldh	0x1F		0x2F	ldhio	0x3F	

Table 8-2. OPX Encodings for R-Type Instructions

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

Assembler Pseudo-Instructions

Table 8-3 lists pseudo-instructions available in Nios II assembly language. Pseudo-instructions are used in assembly source code like regular assembly instructions. Each pseudo-instruction is implemented at the machine level using an equivalent instruction. The `movia` pseudo-instruction is the only exception, being implemented with two instructions. Most pseudo-instructions do not appear in disassembly views of machine code.

Table 8-3. Assembler Pseudo-Instructions (Part 1 of 2)

Pseudo-Instruction	Equivalent Instruction
<code>bgt rA, rB, label</code>	<code>blt rB, rA, label</code>
<code>bgtu rA, rB, label</code>	<code>bltu rB, rA, label</code>
<code>ble rA, rB, label</code>	<code>bge rB, rA, label</code>
<code>bleu rA, rB, label</code>	<code>bgeu rB, rA, label</code>
<code>cmpgt rC, rA, rB</code>	<code>cmplt rC, rB, rA</code>

Table 8-3. Assembler Pseudo-Instructions (Part 2 of 2)

Pseudo-Instruction	Equivalent Instruction
cmpgti rB, rA, IMMED	cmpgei rB, rA, (IMMED+1)
cmpgtu rC, rA, rB	cmpltu rC, rB, rA
cmpgtui rB, rA, IMMED	cmpgeui rB, rA, (IMMED+1)
cmple rC, rA, rB	cmpge rC, rB, rA
cmplei rB, rA, IMMED	cmplti rB, rA, (IMMED+1)
cmpleu rC, rA, rB	cmpgeu rC, rB, rA
cmpleui rB, rA, IMMED	cmpltui rB, rA, (IMMED+1)
mov rC, rA	add rC, rA, r0
movhi rB, IMMED	orhi rB, r0, IMMED
movi rB, IMMED	addi, rB, r0, IMMED
movia rB, label	orhi rB, r0, %hiadj(label) addi, rB, r0, %lo(label)
movui rB, IMMED	ori rB, r0, IMMED
nop	add r0, r0, r0
subi rB, rA, IMMED	addi rB, rA, (-IMMED)

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

Macro	Description	Operation
%lo(immed32)	Extract bits [15..0] of immed32	immed32 & 0xffff
%hi(immed32)	Extract bits [31..16] of immed32	(immed32 >> 16) & 0xffff
%hiadj(immed32)	Extract bits [31..16] and adds bit 15 of immed32	((immed32 >> 16) & 0xffff) + ((immed32 >> 15) & 0x1)
%gp[rel](immed32)	Replace the immed32 address with an offset from the global pointer (1)	immed32 -_gp

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.

Table 8-5. Notation Conventions

Notation	Meaning
$X \leftarrow Y$	X is written with Y
$PC \leftarrow X$	The program counter (PC) is written with address X; the instruction at X will be the next instruction to execute
PC	The address of the assembly instruction in question
rA, rB, rC	One of the 32-bit general-purpose registers
IMM n	An n -bit immediate value, embedded in the instruction word
IMMED	An immediate value
X_n	The n^{th} bit of X, where $n = 0$ is the LSB
$X_{n..m}$	Consecutive bits n through m of X
0xNNMM	Hexadecimal notation
X : Y	Bitwise concatenation For example, (0x12 : 0x34) = 0x1234
$\alpha(X)$	The value of X after being sign-extended to a full register-sized signed integer
$X \gg n$	The value X after being right-shifted n bit positions
$X \ll n$	The value X after being left-shifted n bit positions
$X \& Y$	Bitwise logical AND
$X Y$	Bitwise logical OR
$X \wedge Y$	Bitwise logical XOR
$\sim X$	Bitwise logical NOT (one's complement)
Mem8[X]	The byte located in data memory at byte-address X
Mem16[X]	The halfword located in data memory at byte-address X
Mem32[X]	The word located in data memory at byte-address X
label	An address label specified in the assembly file
(signed) rX	The value of rX treated as a signed number
(unsigned) rX	The value of rX treated as an unsigned number

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**add****Operation:** $rC \leftarrow rA + rB$ **Assembler Syntax:** `add rC, rA, rB`**Example:** `add r6, r7, r8`**Description:** Calculates the sum of rA and rB. Stores the result in rC. Used for both signed and unsigned addition.**Usage: Carry Detection (unsigned operands):**

Following an `add` operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. Both cases are shown below.

```
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
```

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.

```
add rC, rA, rB           ; The original add operation
xor rD, rC, rA          ; Compare signs of sum and rA
xor rE, rC, rB          ; Compare signs of sum and rB
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 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					0x31					0					0x3a						

addi

add immediate

- Operation:** $rB \leftarrow rA + \sigma(\text{IMM16})$
- Assembler Syntax:** `addi rB, rA, IMM16`
- Example:** `addi r6, r7, -100`
- Description:** Sign-extends the 16-bit immediate value and adds it to the value of rA. Stores the sum in rB.

Usage: **Carry Detection (unsigned operands):**

Following an `addi` operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. Both cases are shown below.

```
addi rB, rA, IMM16      ; The original add operation
cmlptu rD, rB, rA      ; rD is written with the carry bit
```

```
addi rB, rA, IMM16      ; The original add operation
bltu rB, rA, label     ; Branch if carry was generated
```

Overflow Detection (signed operands):

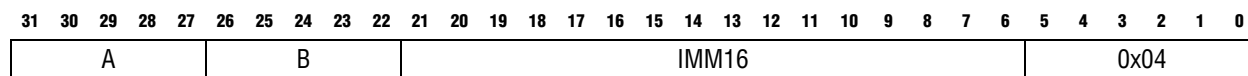
An overflow is detected when two positives are added and the sum is negative, or when two negatives are added and the sum is positive. The overflow condition can control a conditional branch, as shown below.

```
addi rB, rA, IMM16      ; The original add operation
xor rC, rB, rA          ; Compare signs of sum and rA
xorhi rD, rB, IMM16     ; Compare signs of sum and IMM16
and rC, rC, rD          ; Combine comparisons
blt rC, r0, label       ; Branch if overflow occurred
```

Exceptions: None

Instruction Type: I

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



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											

andhi bitwise logical and immediate into high halfword

Operation: $rB \leftarrow rA \& (IMM16 : 0x0000)$

Assembler Syntax: `andhi rB, rA, IMM16`

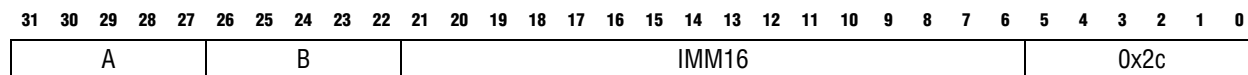
Example: `andhi r6, r7, 100`

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

Exceptions: None

Instruction Type: I

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



andi**bitwise logical and immediate**

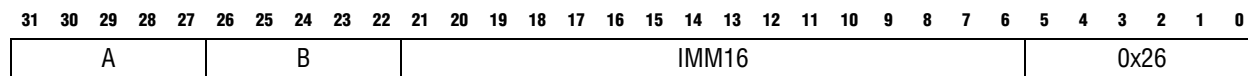
Operation:	$rB \leftarrow rA \& (0x0000 : IMM16)$
Assembler Syntax:	<code>andi rB, rA, IMM16</code>
Example:	<code>andi r6, r7, 100</code>
Description:	Calculates the bitwise logical AND of rA and (0x0000 : IMM16) and stores the result in rB.
Exceptions:	None
Instruction Type:	I
Instruction Fields:	A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

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



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

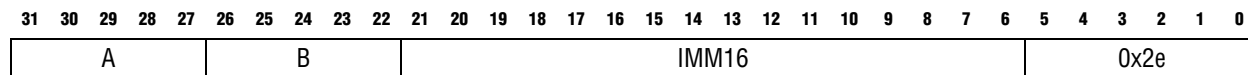
Example: bgeu r6, r7, top_of_loop

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



bgt**branch if greater than signed**

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

bgtu

branch if greater than unsigned

Operation:	if ((unsigned) rA > (unsigned) rB) then PC ← label else PC ← PC + 4
Assembler Syntax:	bgtu rA, rB, label
Example:	bgtu r6, r7, top_of_loop
Description:	If (unsigned) rA > (unsigned) rB, then <code>bgtu</code> transfers program control to the instruction at label.
Pseudo-instruction:	<code>bgtu</code> is implemented with the <code>b1tu</code> 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.

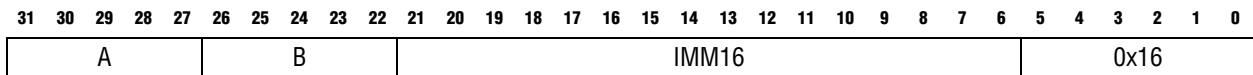
bleu

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 \leftarrow PC + 4 + σ (IMM16) else PC \leftarrow PC + 4
Assembler Syntax:	blt rA, rB, label
Example:	blt r6, r7, top_of_loop
Description:	If (signed) rA < (signed) rB, then <code>blt</code> 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 <code>blt</code> . 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



bltu

branch if less than unsigned

Operation: if ((unsigned) rA < (unsigned) rB)
 then PC ← PC + 4 + σ(IMM16)
 else PC ← PC + 4

Assembler Syntax: bltu rA, rB, label

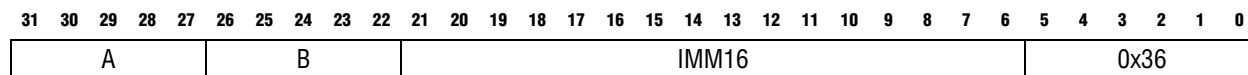
Example: bltu r6, r7, top_of_loop

Description: If (unsigned) rA < (unsigned) rB, then bltu transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following bltu. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.

Exceptions: Misaligned destination address

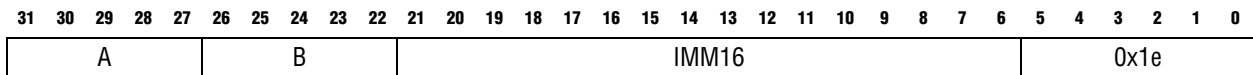
Instruction Type: I

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



bne**branch if not equal**

Operation:	if ($rA \neq rB$) then $PC \leftarrow PC + 4 + \sigma(\text{IMM16})$ else $PC \leftarrow PC + 4$
Assembler Syntax:	<code>bne rA, rB, label</code>
Example:	<code>bne r6, r7, top_of_loop</code>
Description:	If $rA \neq rB$, then <code>bne</code> transfers program control to the instruction at <code>label</code> . In the instruction encoding, the offset given by <code>IMM16</code> is treated as a signed number of bytes relative to the instruction immediately following <code>bne</code> . The two least-significant bits of <code>IMM16</code> 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



br

unconditional branch

Operation: $PC \leftarrow PC + 4 + \sigma(IMM16)$

Assembler Syntax: `br label`

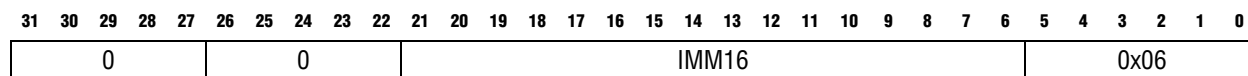
Example: `br top_of_loop`

Description: Transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `br`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.

Exceptions: Misaligned destination address

Instruction Type: I

Instruction Fields: IMM16 = 16-bit signed immediate value



break**debugging breakpoint**

Operation: $bstatus \leftarrow status$
 $PIE \leftarrow 0$
 $U \leftarrow 0$
 $ba \leftarrow PC + 4$
 $PC \leftarrow \text{break handler address}$

Assembler Syntax: `break`
`break imm5`

Example: `break`

Description: Breaks program execution and transfers control to the debugger break-processing routine. Saves the address of the next instruction in register `ba` and saves the contents of the `status` register in `bstatus`. Disables interrupts, then transfers execution to the break handler.

The 5-bit immediate field `imm5` is ignored by the processor, but it can be used by the debugger.

`break` with no argument is the same as `break 0`.

Usage: `break` is used by debuggers exclusively. Only debuggers should place `break` in a user program, operating system, or exception handler. The address of the break handler is specified 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

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					0x1e					0x34					IMM5					0x3a						

bret

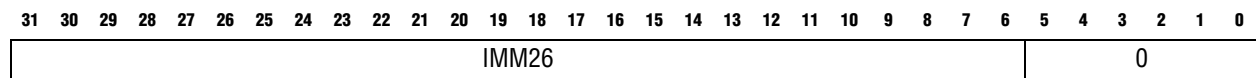
breakpoint return

Operation:	$status \leftarrow bstatus$ $PC \leftarrow ba$
Assembler Syntax:	bret
Example:	bret
Description:	Copies the value of <code>bstatus</code> to the <code>status</code> register, then transfers execution to the address in <code>ba</code> .
Usage:	<code>bret</code> 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**call subroutine**

Operation:	$ra \leftarrow PC + 4$ $PC \leftarrow (PC_{31..28} : IMM26 \times 4)$
Assembler Syntax:	<code>call label</code>
Example:	<code>call write_char</code>
Description:	Saves the address of the next instruction in register <code>ra</code> , and transfers execution to the instruction at address $(PC_{31..28} : IMM26 \times 4)$.
Usage:	<code>call</code> 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



callr

call subroutine in register

Operation: $ra \leftarrow PC + 4$
 $PC \leftarrow rA$

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

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				0x1f				0x1d				0				0x3a											

cmpeq**compare equal**

Operation:	if (rA == rB) then rC ← 1 else rC ← 0
Assembler Syntax:	cmpeq rC, rA, rB
Example:	cmpeq r6, r7, r8
Description:	If rA == rB, then stores 1 to rC; otherwise, stores 0 to rC.
Usage:	cmpeq performs the == operation of the C programming language. Also, cmpeq can be used to implement the C logical-negation operator "!". cmpeq rC, rA, r0 ; Implements rC = !rA
Exceptions:	None
Instruction Type:	R
Instruction Fields:	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

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				0x20				0				0x3a											

cmpeqi

compare equal immediate

- Operation:** if ($rA \sigma(\text{IMM16})$)
 then $rB \leftarrow 1$
 else $rB \leftarrow 0$
- Assembler Syntax:** `cmpeqi rB, rA, IMM16`
- Example:** `cmpeqi r6, r7, 100`
- Description:** Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If $rA == \sigma(\text{IMM16})$, `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:	if ((signed) rA >= (signed) rB) then rC ← 1 else rC ← 0
Assembler Syntax:	cmpge rC, rA, rB
Example:	cmpge r6, r7, r8
Description:	If rA >= rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage:	cmpge performs the signed >= operation of the C programming language.
Exceptions:	None
Instruction Type:	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 \leftarrow 1
 else rB \leftarrow 0

Assembler Syntax: `cmpgei rB, rA, IMM16`

Example: `cmpgei r6, r7, 100`

Description: Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA >= α (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

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												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: if ((unsigned) rA >= (unsigned) (0x0000 : IMM16))
 then rB ← 1
 else rB ← 0

Assembler Syntax: `cmpgeui rB, rA, IMM16`

Example: `cmpgeui r6, r7, 100`

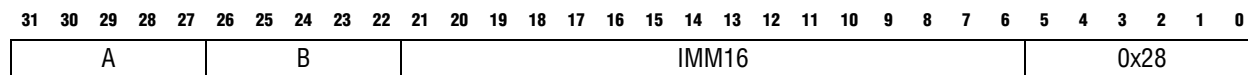
Description: Zero-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA >= (0x0000 : IMM16), then `cmpgeui` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmpgeui` performs the unsigned >= operation of the C programming language.

Exceptions: None

Instruction Type: I

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



cmpgt**compare greater than signed**

Operation:	if ((signed) rA > (signed) rB) then rC ← 1 else rC ← 0
Assembler Syntax:	cmpgt rC, rA, rB
Example:	cmpgt r6, r7, r8
Description:	If rA > rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage:	cmpgt performs the signed > operation of the C programming language.
Pseudo-instruction:	cmpgt is implemented with the <code>cmplt</code> instruction by swapping its rA and rB operands.

cmpgti

compare greater than signed immediate

Operation:	if ((signed) rA > (signed) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax:	<code>cmpgti rB, rA, IMMED</code>
Example:	<code>cmpgti r6, r7, 100</code>
Description:	Sign-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If $rA > \alpha(\text{IMMED})$, then <code>cmpgti</code> stores 1 to rB; otherwise stores 0 to rB.
Usage:	<code>cmpgti</code> 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:	<code>cmpgti</code> is implemented using a <code>cmpgei</code> instruction with an IMM16 immediate value of IMMED + 1.

cmpgtu**compare greater than unsigned**

Operation:	if ((unsigned) rA > (unsigned) rB) then rC ← 1 else rC ← 0
Assembler Syntax:	cmpgtu rC, rA, rB
Example:	cmpgtu r6, r7, r8
Description:	If rA > rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage:	cmpgtu performs the unsigned > operation of the C programming language.
Pseudo-instruction:	cmpgtu is implemented with the <code>cmpltu</code> instruction by swapping its rA and rB operands.

cmpgtui

compare greater than unsigned immediate

Operation:	if ((unsigned) rA > (unsigned) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax:	cmpgtui rB, rA, IMMED
Example:	cmpgtui r6, r7, 100
Description:	Zero-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA > IMMED, then <code>cmpgtui</code> stores 1 to rB; otherwise stores 0 to rB.
Usage:	<code>cmpgtui</code> 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:	<code>cmpgtui</code> is implemented using a <code>cmpgeui</code> instruction with an IMM16 immediate value of IMMED + 1.

cmple**compare less than or equal signed**

Operation:	if ((signed) rA <= (signed) rB) then rC ← 1 else rC ← 0
Assembler Syntax:	cmple rC, rA, rB
Example:	cmple r6, r7, r8
Description:	If rA <= rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage:	cmple performs the signed <= operation of the C programming language.
Pseudo-instruction:	cmple is implemented with the cmpge instruction by swapping its rA and rB operands.

cmplei

compare less than or equal signed immediate

Operation:	if ((signed) rA < (signed) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax:	<code>cmplei rB, rA, IMMED</code>
Example:	<code>cmplei r6, r7, 100</code>
Description:	Sign-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If $rA \leq \alpha(\text{IMMED})$, then <code>cmplei</code> stores 1 to rB; otherwise stores 0 to rB.
Usage:	<code>cmplei</code> performs the signed \leq operation of the C programming language. The maximum allowed value of IMMED is 32766. The minimum allowed value is -32769.
Pseudo-instruction:	<code>cmplei</code> is implemented using a <code>cmplti</code> instruction with an IMM16 immediate value of IMMED + 1.

cmpleu**compare less than or equal unsigned**

Operation:	if ((unsigned) rA < (unsigned) rB) then rC ← 1 else rC ← 0
Assembler Syntax:	cmpleu rC, rA, rB
Example:	cmpleu r6, r7, r8
Description:	If rA ≤ rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage:	cmpleu performs the unsigned ≤ operation of the C programming language.
Pseudo-instruction:	cmpleu is implemented with the cmpgeu instruction by swapping its rA and rB operands.

cmpleui

compare less than or equal unsigned immediate

Operation:	if ((unsigned) rA <= (unsigned) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax:	<code>cmpleui rB, rA, IMMED</code>
Example:	<code>cmpleui r6, r7, 100</code>
Description:	Zero-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA <= IMMED, then <code>cmpleui</code> stores 1 to rB; otherwise stores 0 to rB.
Usage:	<code>cmpleui</code> 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:	<code>cmpleui</code> is implemented using a <code>cmpltui</code> instruction with an IMM16 immediate value of IMMED + 1.

cmplt**compare less than signed**

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

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

compare less than signed immediate

Operation: if ((signed) rA < (signed) σ (IMM16))
 then rB \leftarrow 1
 else rB \leftarrow 0

Assembler Syntax: `cmplti rB, rA, IMM16`

Example: `cmplti r6, r7, 100`

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

Usage: `cmplti` performs the signed < operation of the C programming language.

Exceptions: None

Instruction Type: I

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

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

Example: cmpltui r6, r7, 100

Description: Zero-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA < (0x0000 : IMM16), then cmpltui stores 1 to rB; otherwise stores 0 to rB.

Usage: cmpltui performs the unsigned < operation of the C programming language.

Exceptions: None

Instruction Type: I

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

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												0x30											

cmpne**compare not equal**

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

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													

cmpnei

compare not equal immediate

- Operation:** if ($rA \neq \sigma(\text{IMM16})$)
 then $rB \leftarrow 1$
 else $rB \leftarrow 0$
- Assembler Syntax:** `cmpnei rB, rA, IMM16`
- Example:** `cmpnei r6, r7, 100`
- Description:** Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If $rA \neq \sigma(\text{IMM16})$, 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:	<code>custom N, xC, xA, xB</code> Where xA means either general purpose register rA, or custom register cA.
Example:	<code>custom 0, c6, r7, r8</code>
Description:	The <code>custom</code> 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 <code>readra</code> , <code>readrb</code> , or <code>writerc</code> that corresponds to the register field. In assembler syntax, the notation <code>cN</code> refers to register N in the custom register file and causes the assembler to clear the <code>c</code> bit of the opcode. For example, <code>custom 0, c3, r5, r0</code> 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 <code>readra</code> = 1 if instruction uses rA, 0 otherwise <code>readrb</code> = 1 if instruction uses rB, 0 otherwise <code>writerc</code> = 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

divide

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:

```
div rC, rA, rB      ; The original div operation
mul rD, rC, rB
sub rD, rA, rD      ; rD = remainder
```

Exceptions: Division error
 Unimplemented instruction

Instruction Type: R

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

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**divide unsigned****Operation:** $rC \leftarrow rA \div rB$ **Assembler Syntax:** `divu rC, rA, rB`**Example:** `divu r6, r7, r8`**Description:** Treating *rA* and *rB* as unsigned integers, this instruction divides *rA* by *rB* and then stores the integer portion of the resulting quotient to *rC*. After attempted division by zero, the value of *rC* is undefined. There is no divide-by-zero exception.Nios II processors that do not implement the `divu` instruction cause an unimplemented-instruction exception.**Usage:** Remainder of Division:If the result of the division is defined, then the remainder can be computed in *rD* using the following instruction sequence:

```
divu rC, rA, rB    ; The original divu operation
mul  rD, rC, rB
sub  rD, rA, rD    ; rD = remainder
```

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

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								0x24				0				0x3a			

eret

exception return

Operation:	$status \leftarrow estatus$ $PC \leftarrow ea$
Assembler Syntax:	eret
Example:	eret
Description:	Copies the value of <code>estatus</code> into the <code>status</code> register, and transfers execution to the address in <code>ea</code> .
Usage:	Use <code>eret</code> 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 <code>ea</code> register.
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
0x1d				0x1e				0				0x01				0				0x3a											

flushd**flush data cache line**

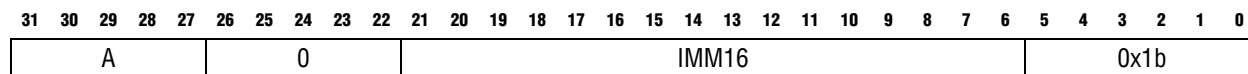
Operation:	Flushes the data cache line associated with address $rA + \sigma$ (IMM16).
Assembler Syntax:	<code>flushd IMM16 (rA)</code>
Example:	<code>flushd -100 (r6)</code>
Description:	<p>If the Nios II processor implements a direct mapped data cache, <code>flushd</code> writes the data cache line that is mapped to the specified address back to memory if the line is dirty, and then clears the data cache line. Unlike <code>flushda</code>, <code>flushd</code> writes the dirty data back to memory even when the addressed data is not currently in the cache. This process comprises the following steps:</p> <ul style="list-style-type: none"> ■ Compute the effective address specified by the sum of <code>rA</code> 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 <code>tag</code> field and a <code>line</code> field. When identifying the data cache line, <code>flushd</code> ignores the <code>tag</code> field and only uses the <code>line</code> 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 <code>flushd</code> ignores the cache line tag, <code>flushd</code> 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. <p>If the Nios II processor core does not have a data cache, the <code>flushd</code> instruction performs no operation.</p>
Usage:	<p>Use <code>flushd</code> 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.</p> <p>For more information on data cache, refer to the Cache and Tightly Coupled Memory chapter of the <i>Nios II Software Developer's Handbook</i>.</p>
Exceptions:	None
Instruction Type:	I
Instruction Fields:	<p>A = Register index of operand <code>rA</code></p> <p>IMM16 = 16-bit signed immediate value</p>

	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												0x3b											

flushda

flush data cache address

Operation:	Flushes the data cache line currently caching address $rA + \sigma$ (IMM16)
Assembler Syntax:	<code>flushda IMM16 (rA)</code>
Example:	<code>flushda -100 (r6)</code>
Description:	<p>If the Nios II processor implements a direct mapped data cache, <code>flushda</code> 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 <code>flushd</code>, <code>flushda</code> writes the dirty data back to memory only when the addressed data is currently in the cache. This process comprises the following steps:</p> <ul style="list-style-type: none"> ■ Compute the effective address specified by the sum of <code>rA</code> 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 <code>tag</code> field and a <code>line</code> field. When identifying the line, <code>flushda</code> uses both the <code>tag</code> field and the <code>line</code> field. ■ Compare the cache line tag with the effective address to determine if the addressed data is currently cached. If the <code>tag</code> 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 <code>tag</code> 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. <p>If the Nios II processor core does not have a data cache, the <code>flushda</code> instruction performs no operation.</p>
Usage:	<p>Use <code>flushda</code> 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.</p> <p>For more information on the Nios II data cache, refer to the <i>Cache and Tightly Coupled Memory</i> chapter of the <i>Nios II Software Developer's Handbook</i>.</p>
Exceptions:	<p>Supervisor-only data address</p> <p>Fast TLB miss (data)</p> <p>Double TLB miss (data)</p> <p>MPU region violation (data)</p>
Instruction Type:	I
Instruction Fields:	<p>A = Register index of operand <code>rA</code></p> <p>IMM16 = 16-bit signed immediate value</p>



flushi**flush instruction cache line**

Operation:	Flushes the instruction-cache line associated with address rA.
Assembler Syntax:	<code>flushi rA</code>
Example:	<code>flushi r6</code>
Description:	<p>Ignoring the tag, <code>flushi</code> identifies the instruction-cache line associated with the byte address in rA, and invalidates that line.</p> <p>If the Nios II processor core does not have an instruction cache, the <code>flushi</code> instruction performs no operation.</p> <p>For more information about the data cache, refer to the <i>Cache and Tightly Coupled Memory</i> chapter of the <i>Nios II Software Developer's Handbook</i>.</p>
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

flush pipeline

- Operation:** Flushes 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			0x04			0			0x3a																	

initd**initialize data cache line**

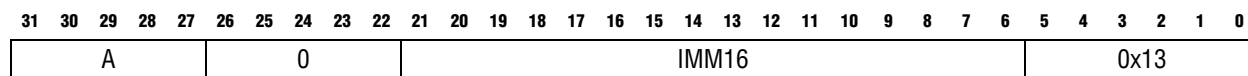
Operation:	Initializes the data cache line associated with address $rA + \sigma$ (IMM16).
Assembler Syntax:	<code>initd IMM16(rA)</code>
Example:	<code>initd 0(r6)</code>
Description:	<p>If the Nios II processor implements a direct mapped data cache, <code>initd</code> clears the data cache line without checking for (or writing) a dirty data cache line that is mapped to the specified address back to memory. Unlike <code>initda</code>, <code>initd</code> clears the cache line regardless of whether the addressed data is currently cached. This process comprises the following steps:</p> <ul style="list-style-type: none"> ■ Compute the effective address specified by the sum of <code>rA</code> 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 <code>tag</code> field and a <code>line</code> field. When identifying the line, <code>initd</code> ignores the <code>tag</code> field and only uses the <code>line</code> 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 <code>initd</code> ignores the cache line tag, <code>initd</code> flushes the cache line regardless of whether the specified data location is currently cached. ■ Skip checking if the data cache line is dirty. Because <code>initd</code> 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. <p>If the Nios II processor core does not have a data cache, the <code>initd</code> instruction performs no operation.</p>
Usage:	<p>Use <code>initd</code> after processor reset and before accessing data memory to initialize the processor's data cache. Use <code>initd</code> 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 <code>initd</code> only when the processor comes out of reset.</p> <p>For more information on data cache, refer to the Cache and Tightly Coupled Memory chapter of the <i>Nios II Software Developer's Handbook</i>.</p>
Exceptions:	Supervisor-only instruction
Instruction Type:	I
Instruction Fields:	<p>A = Register index of operand <code>rA</code></p> <p>IMM16 = 16-bit signed immediate value</p>

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 $rA + \sigma$ (IMM16)
Assembler Syntax:	<code>initda IMM16 (rA)</code>
Example:	<code>initda -100 (r6)</code>
Description:	<p>If the Nios II processor implements a direct mapped data cache, <code>initda</code> 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 <code>initd</code>, <code>initda</code> clears the cache line only when the addressed data is currently cached. This process comprises the following steps:</p> <ul style="list-style-type: none"> ■ Compute the effective address specified by the sum of <code>rA</code> 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 <code>tag</code> field and a <code>line</code> field. When identifying the line, <code>initda</code> uses both the <code>tag</code> field and the <code>line</code> field. ■ Compare the cache line tag with the effective address to determine if the addressed data is currently cached. If the <code>tag</code> 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 <code>initd</code> 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. <p>If the Nios II processor core does not have a data cache, the <code>initda</code> instruction performs no operation.</p>
Usage:	<p>Use <code>initda</code> 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 <code>initda</code> with caution because it does not write back dirty data.</p> <p>For more information on the Nios II data cache, refer to the <i>Cache and Tightly Coupled Memory</i> chapter of the <i>Nios II Software Developer's Handbook</i>.</p>
Exceptions:	<p>Supervisor-only data address Fast TLB miss (data) Double TLB miss (data) MPU region violation (data) Unimplemented instruction</p>
Instruction Type:	I
Instruction Fields:	<p>A = Register index of operand <code>rA</code> IMM16 = 16-bit signed immediate value</p>



init**initialize instruction cache line**

Operation:	Initializes the instruction-cache line associated with address rA.
Assembler Syntax:	<code>init rA</code>
Example:	<code>init r6</code>
Description:	<p>Ignoring the tag, <code>init</code> identifies the instruction-cache line associated with the byte address in rA, and <code>init</code> invalidates that line.</p> <p>If the Nios II processor core does not have an instruction cache, the <code>init</code> instruction performs no operation.</p>
Usage:	<p>This instruction is used to initialize the processor's instruction cache. Immediately after processor reset, use <code>init</code> to invalidate each line of the instruction cache.</p> <p>For more information on instruction cache, refer to the <i>Cache and Tightly Coupled Memory</i> chapter of the <i>Nios II Software Developer's Handbook</i>.</p>
Exceptions:	Supervisor-only instruction
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			0x29			0			0x3a																	

jmp

computed jump

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

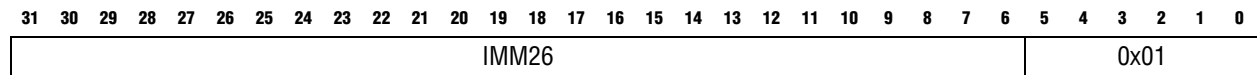
Instruction Type: R

Instruction Fields: A = Register index of operand rA

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				0x0d				0				0x3a											

jmp**jump immediate**

Operation:	$PC \leftarrow (PC_{31..28} : IMM26 \times 4)$
Assembler Syntax:	<code>jmp label</code>
Example:	<code>jmp write_char</code>
Description:	Transfers execution to the instruction at address $(PC_{31..28} : IMM26 \times 4)$.
Usage:	<code>jmp</code> is a low-overhead local jump. <code>jmp</code> 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



ldb / ldbio

load byte from memory or I/O peripheral

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

Assembler Syntax: `ldb rB, byte_offset(rA)`
`ldbio rB, byte_offset(rA)`

Example: `ldb r6, 100(r5)`

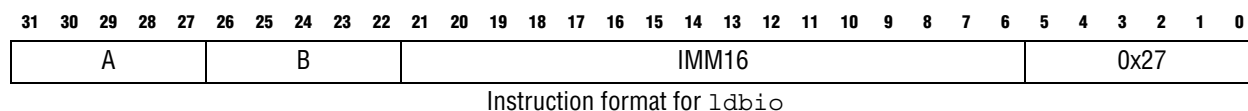
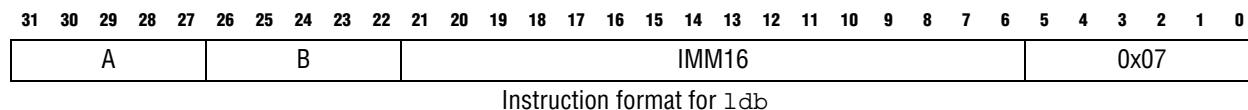
Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the desired memory byte, sign extending the 8-bit value to 32 bits. In Nios II processor cores with a data cache, this instruction may retrieve the desired data from the cache instead of from memory.

Usage: Use the `ldbio` instruction for peripheral I/O. In processors with a data cache, `ldbio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldbio` acts like `ldb`.
 For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Exceptions: Supervisor-only data address
 Misaligned data address
 TLB permission violation (read)
 Fast TLB miss (data)
 Double TLB miss (data)
 MPU region violation (data)

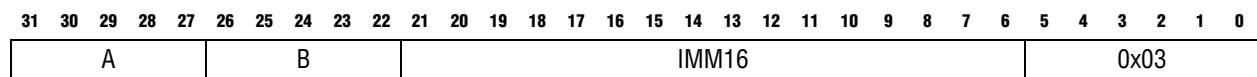
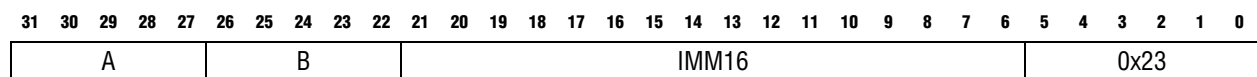
Instruction Type: I

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



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)`**Example:** `ldbu r6, 100(r5)`**Description:** Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the desired memory byte, zero extending the 8-bit value to 32 bits.**Usage:** In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldbuio` instruction for peripheral I/O. In processors with a data cache, `ldbuio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldbuio` acts like `ldbu`.For more information on data cache, refer to the [Cache and Tightly Coupled Memory](#) chapter of the *Nios II Software Developer's Handbook*.

- Exceptions:**
- Supervisor-only data address
 - Misaligned data address
 - TLB permission violation (read)
 - Fast TLB miss (data)
 - Double TLB miss (data)
 - MPU region violation (data)

Instruction Type: I**Instruction Fields:**
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate valueInstruction format for `ldbu`Instruction format for `ldbuio`

ldh / ldhio

load halfword from memory or I/O peripheral

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

Assembler Syntax:
`ldh rB, byte_offset(rA)`
`ldhio rB, byte_offset(rA)`

Example: `ldh r6, 100(r5)`

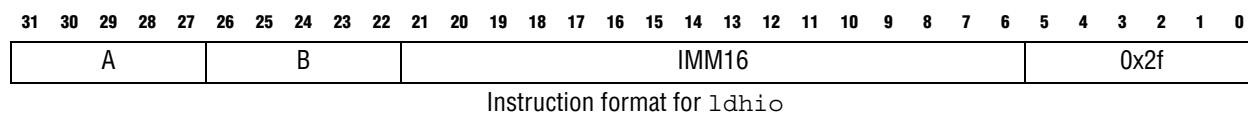
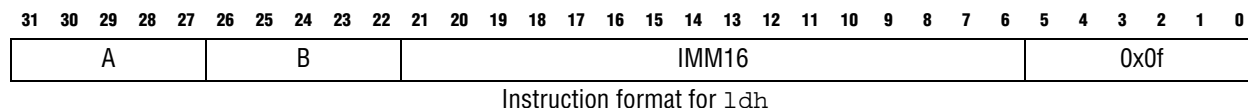
Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the memory halfword located at the effective byte address, sign extending the 16-bit value to 32 bits. The effective byte address must be halfword aligned. If the byte address is not a multiple of 2, the operation is undefined.

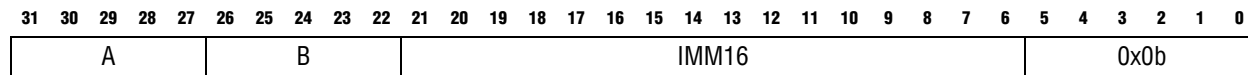
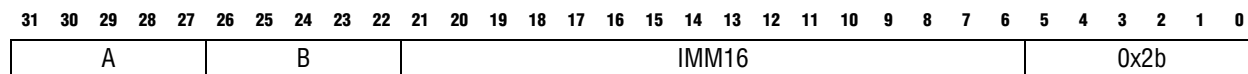
Usage: In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldhio` instruction for peripheral I/O. In processors with a data cache, `ldhio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldhio` acts like `ldh`.
 For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

- Exceptions:**
- Supervisor-only data address
 - Misaligned data address
 - TLB permission violation (read)
 - Fast TLB miss (data)
 - Double TLB miss (data)
 - MPU region violation (data)

Instruction Type: I

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



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
IMM16 = 16-bit signed immediate valueInstruction format for `ldhu`Instruction format for `ldhuio`

ldw / ldwio

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

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

Assembler Syntax:
`ldw rB, byte_offset(rA)`
`ldwio rB, byte_offset(rA)`

Example:
`ldw r6, 100(r5)`

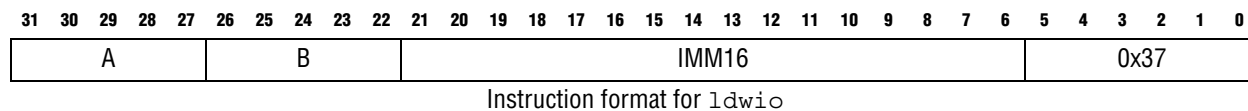
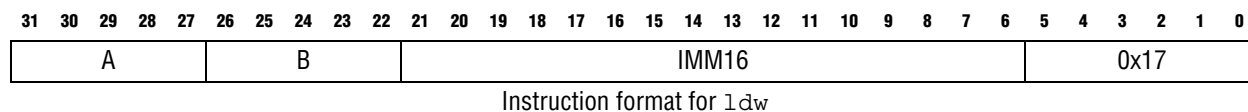
Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the memory word located at the effective byte address. The effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.

Usage: In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldwio` instruction for peripheral I/O. In processors with a data cache, `ldwio` bypasses the cache and memory. Use the `ldwio` instruction for peripheral I/O. In processors with a data cache, `ldwio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldwio` acts like `ldw`.
 For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Exceptions: Supervisor-only data address
 Misaligned data address
 TLB permission violation (read)
 Fast TLB miss (data)
 Double TLB miss (data)
 MPU region violation (data)

Instruction Type: I

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



mov**move register to register**

Operation:	$rC \leftarrow rA$
Assembler Syntax:	<code>mov rC, rA</code>
Example:	<code>mov r6, r7</code>
Description:	Moves the contents of rA to rC.
Pseudo-instruction:	<code>mov</code> is implemented as <code>add rC, rA, r0</code> .

movhi

move immediate into high halfword

Operation: $rB \leftarrow (\text{IMMED} : 0x0000)$

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:	<code>movi rB, IMMED</code>
Example:	<code>movi r6, -30</code>
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 <code>movhi</code> instruction.
Pseudo-instruction:	<code>movi</code> is implemented as <code>addi rB, r0, IMMED</code> .

movia

move immediate address into word

Operation:	$rB \leftarrow \text{label}$
Assembler Syntax:	<code>movia rB, label</code>
Example:	<code>movia r6, function_address</code>
Description:	Writes the address of label to rB.
Pseudo-instruction:	movia is implemented as: <code>orhi rB, r0, %hiadj(label)</code> <code>addi rB, rB, %lo(label)</code>

movui**move unsigned immediate into word**

Operation:	$rB \leftarrow (0x0000 : IMMED)$
Assembler Syntax:	<code>movui rB, IMMED</code>
Example:	<code>movui r6, 100</code>
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 <code>movhi</code> instruction.
Pseudo-instruction:	<code>movui</code> is implemented as <code>ori rB, r0, IMMED</code> .

mul

multiply

Operation: $rC \leftarrow (rA \times rB)_{31..0}$

Assembler Syntax: `mul rC, rA, rB`

Example: `mul r6, r7, r8`

Description: Multiplies rA times rB and stores the 32 low-order bits of the product to rC. The result is the same whether the operands are treated as signed or unsigned integers.
 Nios II processors that do not implement the mul instruction cause an unimplemented-instruction exception.

Usage: **Carry Detection (unsigned operands):**
 Before or after the multiply operation, the carry out of the MSB of rC can be detected using the following instruction sequence:

```
mul rC, rA, rB          ; The mul operation (optional)
mulxuu rD, rA, rB      ; rD is non-zero if carry occurred
cmpne rD, rD, r0       ; rD is 1 if carry occurred, 0 if not
```

The 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 mulxuu with the cmpne instruction.

Overflow Detection (signed operands):
 After the multiply operation, overflow can be detected using the following instruction sequence:

```
mul rC, rA, rB          ; The original mul operation
cmlt rD, rC, r0         ; rD is non-zero if overflow
mulxss rE, rA, rB      ; rE is non-zero if overflow
add rD, rD, rE         ; rD is 1 if overflow, 0 if not
cmpne rD, rD, r0       ; rD is 1 if overflow, 0 if not
```

The cmlt-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 cmpne instruction.

Exceptions: Unimplemented instruction

Instruction Type: R

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

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				0x27				0				0x3a											

mul**multiply immediate**

Operation: $rB \leftarrow (rA \times \alpha(\text{IMM16}))_{31:0}$

Assembler Syntax: `mul rB, rA, IMM16`

Example: `mul 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 `mul` instruction cause an unimplemented-instruction exception.

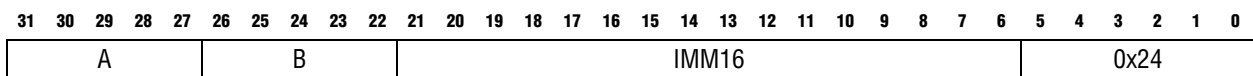
Carry Detection and Overflow Detection:

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



mulxss

multiply extended signed/signed

Operation: $rC \leftarrow ((\text{signed}) rA) \times ((\text{signed}) rB)_{63..32}$

Assembler Syntax: `mulxss rC, rA, rB`

Example: `mulxss r6, r7, r8`

Description: Treating rA and rB as signed integers, `mulxss` multiplies rA times rB, and stores the 32 high-order bits of the product to rC.

Nios II processors that do not implement the `mulxss` instruction cause an unimplemented-instruction exception.

Usage: Use `mulxss` and `mul` to compute the full 64-bit product of two 32-bit signed integers. Furthermore, `mulxss` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit integers, each contained in a pair of 32-bit registers, (S1 : U1) and (S2 : U2), their 128-bit product is $(U1 \times U2) + ((S1 \times U2) \ll 32) + ((U1 \times S2) \ll 32) + ((S1 \times S2) \ll 64)$. The `mulxss` and `mul` instructions are used to calculate the 64-bit product $S1 \times S2$.

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				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, `mulxsu` multiplies `rA` times `rB`, and stores the 32 high-order bits of the product to `rC`.

Nios II processors that do not implement the `mulxsu` instruction cause an unimplemented-instruction exception.

Usage: `mulxsu` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit integers, each contained in a pair of 32-bit registers, (`S1 : U1`) and (`S2 : U2`), their 128-bit product is: $(U1 \times U2) + ((S1 \times U2) \ll 32) + ((U1 \times S2) \ll 32) + ((S1 \times S2) \ll 64)$. The `mulxsu` and `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:** $rC \leftarrow ((\text{unsigned}) rA) \times ((\text{unsigned}) rB))_{63..32}$
- Assembler Syntax:** `mulxuu rC, rA, rB`
- Example:** `mulxuu r6, r7, r8`
- Description:** Treating rA and rB as unsigned integers, `mulxuu` multiplies rA times rB and stores the 32 high-order bits of the product to rC.

Nios II processors that do not implement the `mulxuu` instruction cause an unimplemented-instruction exception.

- Usage:** Use `mulxuu` and `mul` to compute the 64-bit product of two 32-bit unsigned integers. Furthermore, `mulxuu` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit signed integers, each contained in a pair of 32-bit registers, (S1 : U1) and (S2 : U2), their 128-bit product is $(U1 \times U2) + ((S1 \times U2) \ll 32) + ((U1 \times S2) \ll 32) + ((S1 \times S2) \ll 64)$. The `mulxuu` and `mul` instructions are used to calculate the 64-bit product $U1 \times U2$.
- `mulxuu` also can be used as part of the calculation of a 128-bit product of two 64-bit unsigned integers. Given two 64-bit unsigned integers, each contained in a pair of 32-bit registers, (T1 : U1) and (T2 : U2), their 128-bit product is $(U1 \times U2) + ((U1 \times T2) \ll 32) + ((T1 \times U2) \ll 32) + ((T1 \times T2) \ll 64)$. The `mulxuu` and `mul` instructions are used to calculate the four 64-bit products $U1 \times U2$, $U1 \times T2$, $T1 \times U2$, and $T1 \times T2$.

- Exceptions:** Unimplemented instruction

- Instruction Type:** R

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

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				0x07				0				0x3a											

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:** 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
0				0				C				0x1c				0				0x3a											

nop

no operation

Operation:	None
Assembler Syntax:	<code>nop</code>
Example:	<code>nop</code>
Description:	<code>nop</code> does nothing.
Pseudo-instruction:	<code>nop</code> is implemented as <code>add r0, r0, r0</code> .

nor**bitwise logical nor**

Operation:	$rC \leftarrow \sim(rA \mid rB)$
Assembler Syntax:	<code>nor rC, rA, rB</code>
Example:	<code>nor r6, r7, r8</code>
Description:	Calculates the bitwise logical NOR of rA and rB and stores the result in rC.
Exceptions:	None
Instruction Type:	R
Instruction Fields:	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

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

or

bitwise logical or

Operation: $rC \leftarrow rA \mid rB$

Assembler Syntax: `or rC, rA, rB`

Example: `or r6, r7, r8`

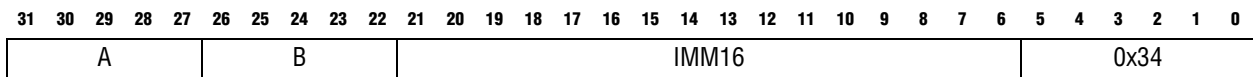
Description: Calculates the bitwise logical OR of rA and rB and stores the result in rC.

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

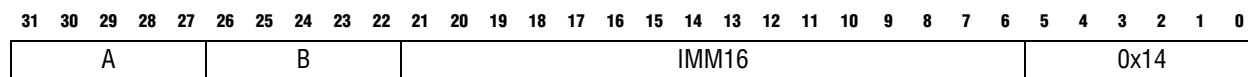
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					C					0x16				0				0x3a								

orhi**bitwise logical or immediate into high halfword****Operation:** $rB \leftarrow rA \mid (\text{IMM16} : 0x0000)$ **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:** $rB \leftarrow rA \mid (0x0000 : IMM16)$
- Assembler Syntax:** `ori rB, rA, IMM16`
- Example:** `ori r6, r7, 100`
- Description:** Calculates the bitwise logical OR of rA and (0x0000 : IMM16) and stores the result in rB.
- Exceptions:** None
- Instruction Type:** I
- Instruction Fields:**
 A = Register index of operand rA
 B = Register index of operand rB
 IMM16 = 16-bit unsigned immediate value



rdctl**read from control register**

Operation: $rC \leftarrow ctIN$

Assembler Syntax: `rdctl rC, ctIN`

Example: `rdctl r3, ct131`

Description: Reads the value contained in control register `ctIN` 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 `ctIN`

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

return from subroutine

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

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					0x03					0			0x3a								

rol

rotate left immediate

- Operation:** $rC \leftarrow rA$ rotated left IMM5 bit positions
- Assembler Syntax:** `rol rC, rA, IMM5`
- Example:** `rol 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, `rol` 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

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					0x02					IMM5					0x3a						

ror**rotate right**

Operation: $rC \leftarrow rA$ rotated right $rB_{4..0}$ bit positions

Assembler Syntax: `ror rC, rA, rB`

Example: `ror r6, r7, r8`

Description: Rotates rA right by the number of bits specified in $rB_{4..0}$ and stores the result in rC . The bits that shift out of the register rotate into the most-significant bit positions. Bits 31–5 of rB are ignored.

Exceptions: None

Instruction Type: R

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

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								

sll

shift left logical

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

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

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				0x12				IMM5				0x3a											

sra

shift right arithmetic

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

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					0x3b					0			0x3a								

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

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

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				0x1b				0				0x3a											

srli**shift right logical immediate****Operation:** $rC \leftarrow (\text{unsigned}) rA \gg ((\text{unsigned}) IMM5)$ **Assembler Syntax:** `srli rC, rA, IMM5`**Example:** `srli r6, r7, 3`**Description:** Shifts rA right by the number of bits specified in IMM5 (inserting zeroes), and then stores the result in rC.**Usage:** `srli` performs the unsigned `>>` operation of the C programming language.**Exceptions:** None**Instruction Type:** R**Instruction Fields:**
A = Register index of operand rA
C = Register index of operand rC
IMM5 = 5-bit unsigned immediate value

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

Assembler Syntax: `stb rB, byte_offset(rA)`
`stbio rB, byte_offset(rA)`

Example: `stb r6, 100(r5)`

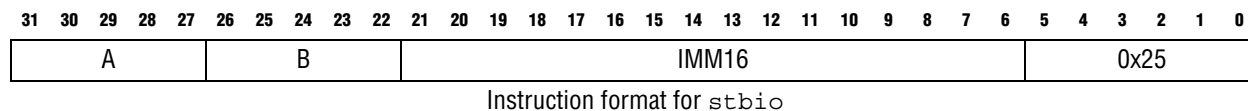
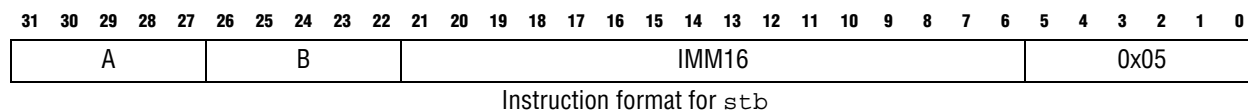
Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Stores the low byte of rB to the memory byte specified by the effective address.

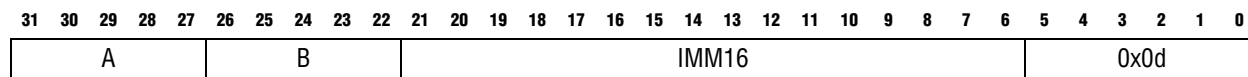
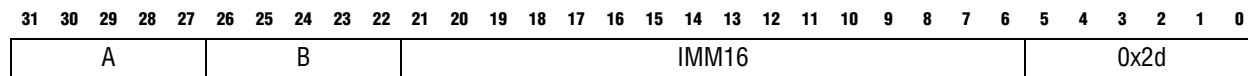
Usage: In processors with a data cache, this instruction may not generate an Avalon-MM bus cycle to non-cache data memory immediately. Use the `stbio` instruction for peripheral I/O. In processors with a data cache, `stbio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. 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
 IMM16 = 16-bit signed immediate value

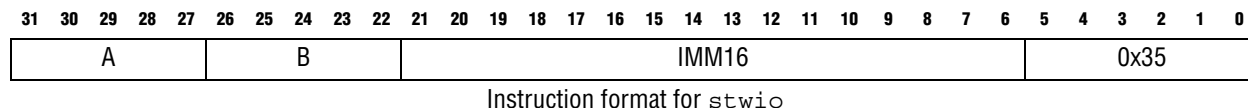
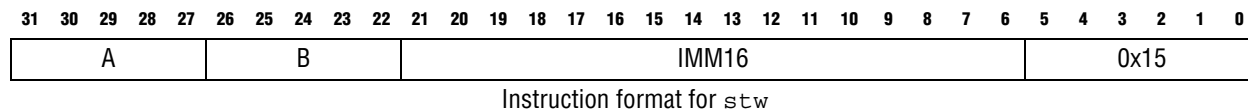


sth / sthio**store halfword to memory or I/O peripheral****Operation:** $\text{Mem16}[\text{rA} + \sigma(\text{IMM16})] \leftarrow \text{rB}_{15:0}$ **Assembler Syntax:**
`sth rB, byte_offset(rA)`
`sthio rB, byte_offset(rA)`**Example:** `sth r6, 100(r5)`**Description:** Computes the effective byte address 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 valueInstruction format for `sth`Instruction format for `sthio`

stw / stwio

store word to memory or I/O peripheral

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



sub**subtract**

Operation: $rC \leftarrow rA - rB$

Assembler Syntax: `sub rC, rA, rB`

Example: `sub r6, r7, r8`

Description: Subtract rB from rA and store the result in rC.

Usage: **Carry Detection (unsigned operands):**

The carry bit indicates an unsigned overflow. Before or after a `sub` operation, a carry out of the MSB can be detected by checking whether the first operand is less than the second operand. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. Both cases are shown 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 rC with the signs of the operands. If rA and rB have different signs, and the sign of rC is different than the sign of rA, an overflow occurred. The overflow condition can control a conditional branch, as shown below.

```
sub rC, rA, rB      ; The original sub operation
xor rD, rA, rB      ; Compare signs of rA and rB
xor rE, rA, rC      ; Compare signs of rA and 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 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								0x39				0				0x3a			

subi

subtract immediate

Operation:	$rB \leftarrow rA - \sigma(\text{IMMED})$
Assembler Syntax:	<code>subi rB, rA, IMMED</code>
Example:	<code>subi r8, r8, 4</code>
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:	<code>subi</code> is implemented as <code>addi rB, rA, -IMMED</code>

sync**memory synchronization**

Operation:	None
Assembler Syntax:	<code>sync</code>
Example:	<code>sync</code>
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				0x36				0				0x3a											

trap

trap

Operation: $estatus \leftarrow status$
 $PIE \leftarrow 0$
 $U \leftarrow 0$
 $ea \leftarrow PC + 4$
 $PC \leftarrow \text{exception handler address}$

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

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				0x1d				0x2d				IMM5				0x3a											

wrctl**write to control register**

Operation: $ctlN \leftarrow rA$

Assembler Syntax: `wrctl ctlN, rA`

Example: `wrctl ctl6, r3`

Description: Writes the value contained in register rA to the control register ctlN.

Exceptions: Supervisor-only instruction

Instruction Type: R

Instruction Fields: A = Register index of operand rA
N = Control register index of operand ctlN

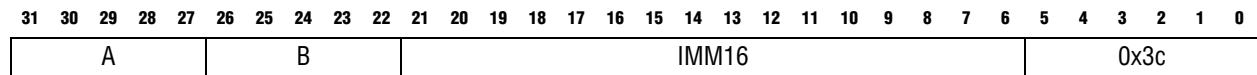
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A				0				0				0x2e				N				0x3a											

XOR

bitwise logical exclusive or

- Operation:** $rC \leftarrow rA \wedge rB$
- Assembler Syntax:** `xor rC, rA, rB`
- Example:** `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:** $rB \leftarrow rA \wedge (\text{IMM16} : 0x0000)$ **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

xori

bitwise logical exclusive or immediate

Operation: $rB \leftarrow rA \wedge (0x0000 : IMM16)$

Assembler Syntax: `xori rB, rA, IMM16`

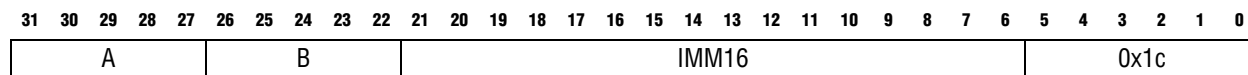
Example: `xori r6, r7, 100`

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

Exceptions: None

Instruction Type: I

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



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 8-6. Document Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Backward-compatible change to the <code>eret</code> instruction B field encoding.	—
November 2008 v8.1.0	Maintenance release.	—
May 2008 v8.0.0	Added an Exceptions section to all instructions.	Added MMU.
October 2007 v7.2.0	Added <code>jmp_i</code> instruction.	—
May 2007 v7.1.0	<ul style="list-style-type: none"> ■ Added table of contents to Introduction section. ■ Added Referenced Documents section. 	—
March 2007 v7.0.0	Maintenance release.	—
November 2006 v6.1.0	Maintenance release.	—
May 2006 v6.0.0	Maintenance release.	—
October 2005 v5.1.0	<ul style="list-style-type: none"> ■ Correction to the <code>blt</code> instruction. ■ Added U bit operation for <code>break</code> and <code>trap</code> instructions. 	—
July 2005 v5.0.1	<ul style="list-style-type: none"> ■ Added new <code>flushda</code> instruction. ■ Updated <code>flushd</code> instruction. ■ Instruction Opcode table updated with <code>flushda</code> instruction. 	—
May 2005 v5.0.0	Maintenance release.	—
December 2004 v1.2	<ul style="list-style-type: none"> ■ <code>break</code> instruction update. ■ <code>srl_i</code> instruction correction. 	—

Table 8-6. Document Revision History (Part 2 of 2)

Date & Document Version	Changes Made	Summary of Changes
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

