8. Instruction Set Reference

NII51017-11.0.0

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 8–3
- "Assembler Pseudo-Instructions" on page 8–4
- "Assembler Macros" on page 8–5
- "Instruction Set Reference" on page 8–5

Word Formats

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

I-Type

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

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

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

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

Table 8–1 shows the I-type instruction format.

Table 8–1. I-Type Instruction Format

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

R-Type

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

- A 6-bit opcode field OP
- Three 5-bit register fields A, B, and C

- An 11-bit opcode-extension field OPX

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

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

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

Table 8–2 shows the R-type instruction format.

Table 8–2. R-Type Instruction Format

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

J-Type

J-type instructions contain:

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

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

Table 8–3 shows the J-type instruction format.

Table 8–3. J-Type Instruction Format

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

add add

Operation: rC ← rA + rB

Assembler Syntax: `add rC, rA, rB`

Example: `add r6, r7, r8`

Description: Calculates the sum of rA and rB. Stores the result in rC. Used for both signed and unsigned addition.

Usage: Carry Detection (unsigned operands):

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

```
add rC, rA, rB              # 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 generated
```

Overflow Detection (signed operands):

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

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

Usage:

Carry Detection (unsigned operands):

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

```
addi rB, rA, IMM16          # The original add operation
cmpltu 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 generated
```

Overflow Detection (signed operands):

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

```
addi rB, rA, IMM16          # 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

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																0x04					

and **bitwise logical and**

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

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 ← 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

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

andi — bitwise logical and immediate

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

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 ← PC + 4 + σ (IMM16)
	else PC ← PC + 4

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

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																0x26					

bge **branch if greater than or equal signed**

Operation: if ((signed) rA >= (signed) rB)
then PC ← PC + 4 + σ (IMM16)
else PC ← PC + 4

Assembler Syntax: `bge rA, rB, label`

Example: `bge r6, r7, top_of_loop`

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

Exceptions: Misaligned destination address

Instruction Type: I

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

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

bgeu — branch if greater than or equal unsigned

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

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

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

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

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																0x16					

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

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																0x36					

bne **branch if not equal**

Operation:	if (rA != rB)
	then PC ← PC + 4 + σ (IMM16)
	else PC ← PC + 4

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

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`																`0x1e`					

call call subroutine

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

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

callr — call subroutine in register

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

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					

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:	I
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					

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					

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					

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					

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					

div divide

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

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

Usage: Remainder of Division:

If the result of the division is defined, then the remainder can be computed in rD using the following instruction sequence:

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

Usage: Remainder of Division:

If the result of the division is defined, then the remainder can be computed in rD using the following instruction sequence:

```
divu rC, rA, rB      # 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					

jmp **computed jump**

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

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					

jmpi jump immediate

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

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

ldb / ldbio — load byte from memory or I/O peripheral

Operation:	rB ← σ (Mem8[rA + σ (IMM16)])

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

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

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																0x07					

Instruction format for `ldb`

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																0x27					

Instruction format for `ldbio`

ldbu / ldbuio — load unsigned byte from memory or I/O peripheral

Operation:	rB ← 0x000000 : Mem8[rA + σ (IMM16)]

Assembler Syntax:	`ldbu rB, byte_offset(rA)`
	`ldbuio rB, byte_offset(rA)`

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					IMM16																0x03					

Instruction format for `ldbu`

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

Instruction format for `ldbuio`

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

Operation:	rB ← Mem32[rA + σ (IMM14)]
Assembler Syntax:	`ldw rB, byte_offset(rA)`
	`ldwio rB, byte_offset(rA)`
Example:	`ldw r6, 100(r5)`
Description:	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the memory word located at the effective byte address. The effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.
Usage:	In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldwio` instruction for peripheral I/O. In processors with a data cache, `ldwio` bypasses the cache and memory. Use the `ldwio` instruction for peripheral I/O. In processors with a data cache, `ldwio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldwio` acts like `ldw`.
	For more information on data cache, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
Exceptions:	Supervisor-only data address
	Misaligned data address
	TLB permission violation (read)
	Fast TLB miss (data)
	Double TLB miss (data)
	MPU region violation (data)
Instruction Type:	I
Instruction Fields:	`A` = Register index of operand rA
	`B` = Register index of operand rB
	`IMM16` = 16-bit signed immediate value

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																0x17					

Instruction format for `ldw`

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																0x37					

Instruction format for `ldwio`

mul — multiply

Operation:	rC ← $(rA \times rB)_{31..0}$
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 nonzero if carry occurred
cmpne rD, rD, r0        # rD is 1 if carry occurred, 0 if not
```

The `mulxuu` instruction writes a nonzero value into rD if the multiplication of unsigned numbers generates a carry (unsigned overflow). If a 0/1 result is desired, follow the `mulxuu` with the `cmpne` instruction.

Overflow Detection (signed operands):

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

```
mul rC, rA, rB          # The original mul operation
cmplt rD, rC, r0
mulxss rE, rA, rB
add rD, rD, rE          # rD is nonzero if overflow
cmpne rD, rD, r0        # rD is 1 if overflow, 0 if not
```

The `cmplt`–`mulxss`–`add` instruction sequence writes a nonzero value into rD if the product in rC cannot be represented in 32 bits (signed overflow). If a 0/1 result is desired, follow the instruction sequence with the `cmpne` instruction.

Exceptions: Unimplemented instruction

Instruction Type: R

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

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					

muli — multiply immediate

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

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																0x24					

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 × U2) + ((S1 × U2) << 32) + ((U1 × S2) << 32) + ((S1 × S2) << 64). The `mulxss` and `mul` instructions are used to calculate the 64-bit product S1 × 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 ← $((\text{signed}) \text{ rA}) \times ((\text{unsigned}) \text{ rB}))_{63..32}$
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 × U2) + ((S1 × U2) << 32) + ((U1 × S2) << 32) + ((S1 × S2) << 64). The `mulxsu` and `mul` instructions are used to calculate the two 64-bit products S1 × U2 and U1 × 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 ← $((\text{unsigned})\ rA) \times ((\text{unsigned})\ rB))_{63..32}$
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 × U2) + ((S1 × U2) << 32) + ((U1 × S2) << 32) + ((S1 × S2) << 64). The `mulxuu` and `mul` instructions are used to calculate the 64-bit product U1 × 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 × U2) + ((U1 × T2) << 32) + ((T1 × U2) << 32) + ((T1 × T2) << 64). The `mulxuu` and `mul` instructions are used to calculate the four 64-bit products U1 × U2, U1 × T2, T1 × U2, and T1 × 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					

nor — bitwise logical nor

Operation:	rC ← ~(rA | rB)
Assembler Syntax:	`nor rC, rA, rB`
Example:	`nor r6, r7, r8`
Description:	Calculates the bitwise logical NOR of rA and rB and stores the result in rC.
Exceptions:	None
Instruction Type:	R
Instruction Fields:	`A` = Register index of operand rA
	`B` = Register index of operand rB
	`C` = Register index of operand rC

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

or **bitwise logical or**

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

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

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																0x34					

ori bitwise logical or immediate

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

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

ret return from subroutine

Operation:	PC ← `ra`
Assembler Syntax:	`ret`
Example:	`ret`
Description:	Transfers execution to the address in `ra`.
Usage:	Any subroutine called by `call` or `callr` must use `ret` to return.
Exceptions:	Misaligned destination address
Instruction Type:	R
Instruction Fields:	None

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 ← 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					

roli rotate left immediate

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

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					

sll shift left logical

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

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					

sra — shift right arithmetic

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

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					

srl **shift right logical**

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

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					

stb / stbio — store byte to memory or I/O peripheral

Operation:	Mem8[rA + σ (IMM16)] ← $rB_{7..0}$
Assembler Syntax:	`stb rB, byte_offset(rA)`
	`stbio rB, byte_offset(rA)`
Example:	`stb r6, 100(r5)`
Description:	Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Stores the low byte of rB to the memory byte specified by the effective address.
Usage:	In processors with a data cache, this instruction may not generate an Avalon-MM bus cycle to noncache data memory immediately. Use the `stbio` instruction for peripheral I/O. In processors with a data cache, `stbio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `stbio` acts like `stb`.
Exceptions:	Supervisor-only data address Misaligned data address TLB permission violation (write) Fast TLB miss (data) Double TLB miss (data) MPU region violation (data)
Instruction Type:	I
Instruction Fields:	`A` = Register index of operand rA `B` = Register index of operand rB `IMM16` = 16-bit signed immediate value

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

Instruction format for `stb`

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

Instruction format for `stbio`

stw / stwio store word to memory or I/O peripheral

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

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

Instruction format for `stw`

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

Instruction format for `stwio`

sub subtract

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

Usage:

Carry Detection (unsigned operands):

The carry bit indicates an unsigned overflow. Before or after a `sub` operation, a carry out of the MSB can be detected by checking whether the first operand is less than the second operand. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. Both cases are shown in the following code:

```
sub rC, rA, rB         # 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 generated
```

Overflow Detection (signed operands):

Detect overflow of signed subtraction by comparing the sign of the difference that is written to rC with the signs of the operands. If rA and rB have different signs, and the sign of rC is different than the sign of rA, an overflow occurred. The overflow condition can control a conditional branch, as shown in the following code:

```
sub rC, rA, rB         # 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					

xor — bitwise logical exclusive or

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

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

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

xori — bitwise logical exclusive or immediate

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

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

Document Revision History

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

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

Date	Version	Changes
May 2011	11.0.0	Maintenance release.
December 2010	10.1.0	Corrected comments delimiter (#) in instruction usage.
July 2010	10.0.0	Corrected typographical error in `cmpgei` instruction type.
November 2009	9.1.0	Added shadow register sets and external interrupt controller support, including `rdprs` and `wrprs` instructions.
March 2009	9.0.0	Backwards-compatible change to the `eret` instruction B field encoding.
November 2008	8.1.0	Maintenance release.
May 2008	8.0.0	■ Added MMU. ■ Added an Exceptions section to all instructions.
October 2007	7.2.0	Added `jmpi` instruction.
May 2007	7.1.0	■ Added table of contents to Introduction section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Maintenance release.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	■ Correction to the `blt` instruction. ■ Added U bit operation for `break` and `trap` instructions.