

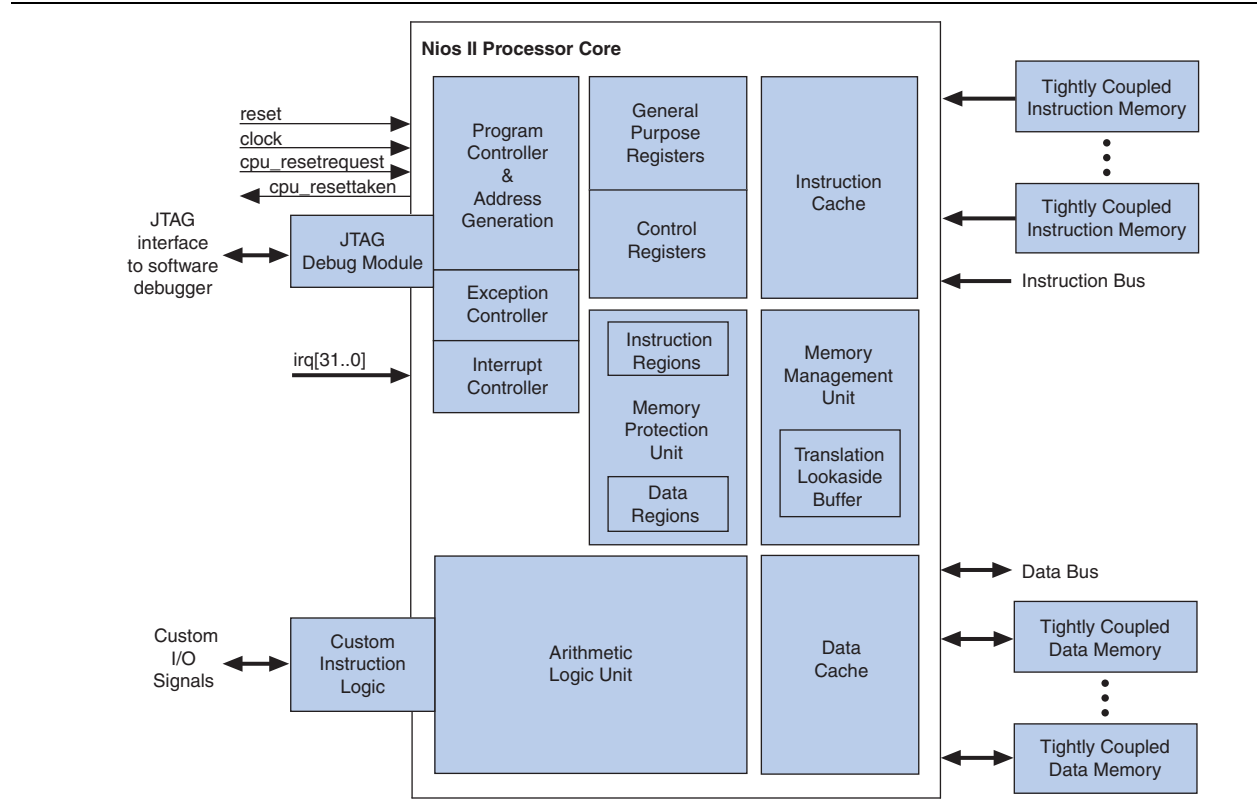
## Introduction

This chapter describes the hardware structure of the Nios® II processor, including a discussion of all the functional units of the Nios II architecture and the fundamentals of the Nios II processor hardware implementation. This chapter contains the following sections:

- “Processor Implementation” on page 2–3
- “Register File” on page 2–3
- “Arithmetic Logic Unit” on page 2–3
- “Reset and Debug Signals” on page 2–7
- “Exception and Interrupt Controller” on page 2–7
- “Memory and I/O Organization” on page 2–9
- “JTAG Debug Module” on page 2–16

The Nios II architecture describes an instruction set architecture (ISA). The ISA in turn necessitates a set of functional units that implement the instructions. A Nios II processor core is a hardware design that implements the Nios II instruction set and supports the functional units described in this document. The processor core does not include peripherals or the connection logic to the outside world. It includes only the circuits required to implement the Nios II architecture.

Figure 2–1 shows a block diagram of the Nios II processor core.

**Figure 2-1.** Nios II Processor Core Block Diagram

The Nios II architecture defines the following functional units:

- Register file
- Arithmetic logic unit (ALU)
- Interface to custom instruction logic
- Exception controller
- Interrupt controller
- Instruction bus
- Data bus
- Memory management unit (MMU)
- Memory protection unit (MPU)
- Instruction and data cache memories
- Tightly-coupled memory interfaces for instructions and data
- JTAG debug module

The following sections discuss hardware implementation details related to each functional unit.


## Processor Implementation

The functional units of the Nios II architecture form the foundation for the Nios II instruction set. However, this does not indicate that any unit is implemented in hardware. The Nios II architecture describes an instruction set, not a particular hardware implementation. A functional unit can be implemented in hardware, emulated in software, or omitted entirely.

A Nios II implementation is a set of design choices embodied by a particular Nios II processor core. All implementations support the instruction set defined in the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*. Each implementation achieves specific objectives, such as smaller core size or higher performance. This allows the Nios II architecture to adapt to the needs of different target applications.

Implementation variables generally fit one of three trade-off patterns: more-or-less of a feature; inclusion-or-exclusion of a feature; hardware implementation or software emulation of a feature. An example of each trade-off follows:

- More or less of a feature—For example, to fine-tune performance, you can increase or decrease the amount of instruction cache memory. A larger cache increases execution speed of large programs, while a smaller cache conserves on-chip memory resources.
- Inclusion or exclusion of a feature—For example, to reduce cost, you can choose to omit the JTAG debug module. This decision conserves on-chip logic and memory resources, but it eliminates the ability to use a software debugger to debug applications.
- Hardware implementation or software emulation—For example, in control applications that rarely perform complex arithmetic, you can choose for the division instruction to be emulated in software. Removing the divide hardware conserves on-chip resources but increases the execution time of division operations.

 For details of which Nios II cores supports what features, refer to the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*. For complete details of user-selectable parameters for the Nios II processor, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

## Register File

The Nios II architecture supports a flat register file, consisting of thirty two 32-bit general-purpose integer registers, and up to thirty two 32-bit control registers. The architecture supports supervisor and user modes that allow system code to protect the control registers from errant applications.

The Nios II architecture allows for the future addition of floating-point registers.

## Arithmetic Logic Unit

The Nios II ALU operates on data stored in general-purpose registers. ALU operations take one or two inputs from registers, and store a result back in a register. The ALU supports the data operations shown in [Table 2-1](#).

**Table 2-1.** Operations Supported by the Nios II ALU


| Category         | Details   |
|------------------|---|
| Arithmetic       | The ALU supports addition, subtraction, multiplication, and division on signed and unsigned operands.   |
| Relational       | The ALU supports the equal, not-equal, greater-than-or-equal, and less-than relational operations ( ==, != >=, < ) on signed and unsigned operands.   |
| Logical          | The ALU supports AND, OR, NOR, and XOR logical operations.  |
| Shift and Rotate | The ALU supports shift and rotate operations, and can shift/rotate data by 0 to 31 bit positions per instruction. The ALU supports arithmetic shift right and logical shift right/left. The ALU supports rotate left/right. |

To implement any other operation, software computes the result by performing a combination of the fundamental operations in [Table 2-1](#).

## Unimplemented Instructions


Some Nios II processor core implementations do not provide hardware to support the entire Nios II instruction set. In such a core, instructions without hardware support are known as unimplemented instructions.

The processor generates an exception whenever it issues an unimplemented instruction so your exception handler can call a routine that emulates the operation in software. Therefore, unimplemented instructions do not affect the programmer's view of the processor.

 For a list of potential unimplemented instructions, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

## Custom Instructions

The Nios II architecture supports user-defined custom instructions. The Nios II ALU connects directly to custom instruction logic, enabling you to implement in hardware operations that are accessed and used exactly like native instructions.

 For further information, refer to the *Nios II Custom Instruction User Guide*.

## Floating-Point Instructions

The Nios II architecture supports single precision floating-point instructions as specified by the IEEE Std 754-1985. The basic set of floating-point custom instructions includes single precision floating-point addition, subtraction, and multiplication. Floating-point division is available as an extension to the basic instruction set. These floating-point instructions are implemented as custom instructions. [Table 2-2](#) provides a detailed description of the conformance to IEEE 754-1985.


**Table 2-2.** Hardware Conformance with IEEE 754-1985 Floating-Point Standard

| Feature                          |                        | Implementation  |
|----------------------------------|------------------------|---|
| Operations (1)                   | Addition               | Implemented   |
|                                  | Subtraction            | Implemented   |
|                                  | Multiplication         | Implemented   |
|                                  | Division               | Optional  |
| Precision                        | Single                 | Implemented   |
|                                  | Double                 | Not implemented. Double precision operations are implemented in software.   |
| Exception conditions             | Invalid operation      | Result is Not a Number (NaN)  |
|                                  | Division by zero       | Result is $\pm$ infinity  |
|                                  | Overflow               | Result is $\pm$ infinity  |
|                                  | Inexact                | Result is a normal number   |
|                                  | Underflow              | Result is $\pm$ 0   |
| Rounding Modes                   | Round to nearest       | Implemented   |
|                                  | Round toward zero      | Not implemented   |
|                                  | Round toward +infinity | Not implemented   |
|                                  | Round toward -infinity | Not implemented   |
| NaN                              | Quiet                  | Implemented   |
|                                  | Signaling              | Not implemented   |
| Subnormal (denormalized) numbers |                        | Subnormal operands are treated as zero. The floating-point custom instructions do not generate subnormal numbers. |
| Software exceptions              |                        | Not implemented. IEEE 754-1985 exception conditions are detected and handled as shown elsewhere in this table.    |
| Status flags                     |                        | Not implemented. IEEE 754-1985 exception conditions are detected and handled as shown elsewhere in this table.    |


**Notes to Table 2-2:**

- (1) The Nios II Embedded Design Suite (EDS) provides software implementations of primitive floating-point operations other than addition, subtraction, multiplication, and division. This includes operations such as floating-point conversions and comparisons. The software implementations of these primitives are 100% compliant with IEEE 754-1985.

You can add floating-point custom instructions to any Nios II processor core using the Nios II Processor MegaWizard interface. The floating-point division hardware requires more resources than the other instructions. The MegaWizard interface allows you to omit the floating-point division hardware for cases in which code running on your hardware design does not make heavy use of floating-point division. When you omit the floating-point divide instruction, the Nios II compiler implements floating-point division in software.

 To add floating-point custom instructions to your Nios II processor core, refer to “Custom Instructions Page” in the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

The Nios II floating-point custom instructions are based on the Altera floating-point megafunctions.

 For details on each individual floating-point megafunction, including acceleration factors and device resource usage, refer to the [Literature: Megafunctions](#) page on the Altera website.


The Nios II software development tools recognize C code that takes advantage of the floating-point instructions present in the processor core. When the floating-point custom instructions are present in your target hardware, the Nios II compiler compiles your code to use the custom instructions for floating-point operations, including addition, subtraction, multiplication, division and the newlib math library.

### Software Development Considerations

The best choice for your hardware design depends on a balance among floating-point usage, hardware resource usage, and performance. While the floating-point custom instructions speed up floating-point arithmetic, they substantially add to the size of your hardware design. If resource usage is an issue, consider reworking your algorithms to minimize floating-point arithmetic.

You can use `#pragma` directives in your software to compare hardware and software implementations of the floating-point instructions. The following `#pragma` directives instruct the Nios II compiler to ignore the floating-point instructions and generate software implementations. The scope of these `#pragma` directives is the entire C file.

- `#pragma no_custom_fadds`—Forces software implementation of floating-point add
- `#pragma no_custom_fsubs`—Forces software implementation of floating-point subtract
- `#pragma no_custom_fmuls`—Forces software implementation of floating-point multiply
- `#pragma no_custom_fdivs`—Forces software implementation of floating-point divide

 The Nios II instruction set simulator (ISS) does not support custom instructions. If you need to run your software on the ISS, disable the floating-point custom instructions in software with the `#pragma` directives.

All the floating-point custom instructions are single-precision operations. Double-precision operations are implemented in software. By default, the Nios II compiler treats floating-point constants as double-precision numbers. To use the floating-point custom instructions for operations with floating-point constants, append an “f” to the constant. This tells the compiler to treat the constant as a single-precision floating-point value, rather than promote the other variables in your equation to double precision numbers. [Table 2-3](#) shows code examples using constants.

**Table 2-3.** Floating-Point Constant Examples

| Example Code                | Precision | Floating-Point Custom Instruction Usage |
|-----------------------------|-----------|---|
| <code>y = x × 4.67;</code>  | Double    | No                                      |
| <code>y = x × 4.67f;</code> | Single    | Yes                                     |


## Reset and Debug Signals

The Nios II processor core supports the following two kinds of reset signals:

- `reset`—This is a global hardware reset signal that forces the processor core to reset immediately.
- `cpu_resetrequest`—This is an optional, local reset signal that causes the processor to reset without affecting other components in the Nios II system. The processor finishes executing any instructions in the pipeline, and then enters the reset state. This process can take several clock cycles, so be sure to continue asserting the `cpu_resetrequest` signal until the processor core asserts a `cpu_resettaken` signal.


The processor core asserts a `cpu_resettaken` signal for 1 cycle when the reset is complete and then periodically if `cpu_resetrequest` remains asserted. The processor remains in the reset state for as long as `cpu_resetrequest` is asserted. While the processor is in the reset state, it periodically reads from the reset address. It discards the result of the read, and remains in the reset state.

The processor does not respond to `cpu_resetrequest` when the processor is under the control of the JTAG debug module, that is, when the processor is paused. The processor responds to the `cpu_resetrequest` signal if the signal is asserted when the JTAG debug module relinquishes control, both momentarily during each single step as well as when you resume execution.

 For more information on adding reset signals to the Nios II processor, refer to “Advanced Features Page” in the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

The Nios II processor core also supports debug signals.

- `debugreq`—This is an optional signal that temporarily suspends the processor for debugging purposes. When you assert the signal, the processor pauses in the same manner as when a breakpoint is encountered, transfers execution to the routine located at the break address, and asserts a `debugack` signal. Asserting the `debugreq` signal when the processor is already paused has no effect.

 For more information on the break vector and adding debug signals to the Nios II processor, refer to “JTAG Debug Module Page” in the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

## Exception and Interrupt Controller

### Exception Controller

The Nios II architecture provides a simple, non-vectorized exception controller to handle all exception types. Each exception, including hardware interrupts, causes the processor to transfer execution to an exception address. An exception handler at this address determines the cause of the exception and dispatches an appropriate exception routine.

Exception addresses are specified in SOPC Builder at system generation time.

All exceptions are precise. Precise means that the processor has completed execution of all instructions preceding the faulting instruction and not started execution of instructions following the faulting instruction. Precise exceptions allow the processor to resume program execution once the exception handler clears the exception.

## Integral Interrupt Controller

The Nios II architecture supports 32 external hardware interrupts. The processor core has 32 level-sensitive interrupt request (IRQ) inputs, `irq0` through `irq31`, providing a unique input for each interrupt source. IRQ priority is determined by software. The architecture supports nested interrupts.

The software can enable and disable any interrupt source individually through the `ienable` control register, which contains an interrupt-enable bit for each of the IRQ inputs. Software can enable and disable interrupts globally using the PIE bit of the `status` control register. A hardware interrupt is generated if and only if all of the following conditions are true:

- The PIE bit of the `status` register is 1
- An interrupt-request input, `irq<n>`, is asserted
- The corresponding bit `n` of the `ienable` register is 1

## Interrupt Vector Custom Instruction

The Nios II processor core offers an interrupt vector custom instruction which accelerates interrupt vector dispatch. Include this custom instruction to reduce your program's interrupt latency.

The interrupt vector custom instruction is based on a priority encoder with one input for each interrupt connected to the Nios II processor. The cost of the interrupt vector custom instruction depends on the number of interrupts connected to the Nios II processor. The worst case is a system with 32 interrupts. In this case, the interrupt vector custom instruction consumes about 50 logic elements (LEs).

If you have a large number of interrupts connected, adding the interrupt vector custom instruction to your system might lower  $f_{MAX}$ .



For guidance in adding the interrupt vector custom instruction to the Nios II processor, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

Table 2-4 details the implementation of the interrupt vector custom instruction.

**Table 2-4.** Interrupt Vector Custom Instruction (Part 1 of 2)

ALT\_CI\_EXCEPTION\_VECTOR\_N

**Operation:** if (`ipending == 0`) | (`estatus.PIE == 0`)  
then `rC` ← negative value  
else `rC` ←  $8 \times$  bit # of the least-significant 1 bit of the `ipending` register (`ct14`)


**Assembler Syntax:** `custom ALT_CI_EXCEPTION_VECTOR_N, rC, r0, r0`

**Example:** `custom ALT_CI_EXCEPTION_VECTOR_N, et, r0, r0`  
`blt et, r0, not_irq`

**Table 2-4.** Interrupt Vector Custom Instruction (Part 2 of 2)

|                            |  |
|----------------------------|--|
| <b>Description:</b>        | The interrupt vector custom instruction accelerates interrupt vector dispatch. This custom instruction identifies the highest priority interrupt, generates the vector table offset, and stores this offset to rC. The instruction generates a negative offset if there is no hardware interrupt (that is, the exception is caused by a software condition, such as a trap). |
| <b>Usage:</b>              | The interrupt vector custom instruction is used exclusively by the exception handler.  |
| <b>Exceptions:</b>         | None   |
| <b>Instruction Type:</b>   | R  |
| <b>Instruction Fields:</b> | C = Register index of operand rC<br>N = Value of ALT_CI_EXCEPTION_VECTOR_N   |

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |      |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|------|----|----|---|---|---|---|---|---|---|---|---|---|
| 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  |    |    |    | 0  | 0  | 1  | N  |    |    |    | 0x32 |    |    |   |   |   |   |   |   |   |   |   |   |

 For an explanation of the instruction reference format, refer to the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*.

## Memory and I/O Organization

This section explains hardware implementation details of the Nios II memory and I/O organization. The discussion covers both general concepts true of all Nios II processor systems, as well as features that might change from system to system.

The flexible nature of the Nios II memory and I/O organization are the most notable difference between Nios II processor systems and traditional microcontrollers. Because Nios II processor systems are configurable, the memories and peripherals vary from system to system. As a result, the memory and I/O organization varies from system to system.

A Nios II core uses one or more of the following to provide memory and I/O access:

- Instruction master port—An Avalon® Memory-Mapped (Avalon-MM) master port that connects to instruction memory via system interconnect fabric
- Instruction cache—Fast cache memory internal to the Nios II core
- Data master port—An Avalon-MM master port that connects to data memory and peripherals via system interconnect fabric
- Data cache—Fast cache memory internal to the Nios II core
- Tightly-coupled instruction or data memory port—Interface to fast on-chip memory outside the Nios II core

The Nios II architecture hides the hardware details from the programmer, so programmers can develop Nios II applications without specific knowledge of the hardware implementation.


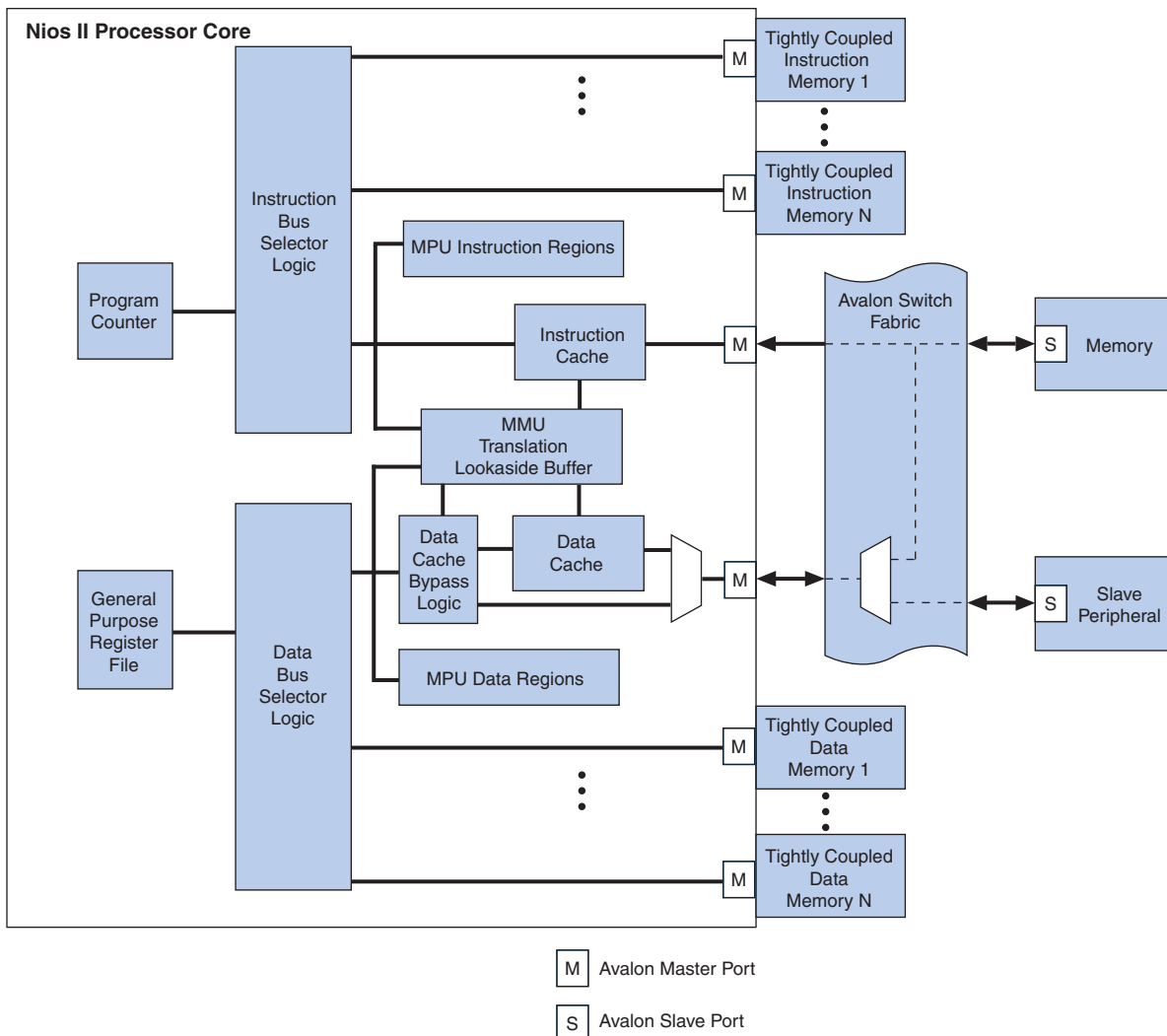
 For details that affect programming issues, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

Figure 2-2 shows a diagram of the memory and I/O organization for a Nios II processor core.

**Figure 2-2.** Nios II Memory and I/O Organization



## Instruction and Data Buses

The Nios II architecture supports separate instruction and data buses, classifying it as a Harvard architecture. Both the instruction and data buses are implemented as Avalon-MM master ports that adhere to the Avalon-MM interface specification. The data master port connects to both memory and peripheral components, while the instruction master port connects only to memory components.

 Refer to the *Avalon Interface Specifications* for details of the Avalon-MM interface.

## Memory and Peripheral Access

The Nios II architecture provides memory-mapped I/O access. Both data memory and peripherals are mapped into the address space of the data master port. The Nios II architecture is little endian. Words and halfwords are stored in memory with the more-significant bytes at higher addresses.

The Nios II architecture does not specify anything about the existence of memory and peripherals; the quantity, type, and connection of memory and peripherals are system-dependent. Typically, Nios II processor systems contain a mix of fast on-chip memory and slower off-chip memory. Peripherals typically reside on-chip, although interfaces to off-chip peripherals also exist.

## Instruction Master Port

The Nios II instruction bus is implemented as a 32-bit Avalon-MM master port. The instruction master port performs a single function: it fetches instructions to be executed by the processor. The instruction master port does not perform any write operations.

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 and increases the overall  $f_{MAX}$  of the system. The instruction master port can issue successive read requests before data has returned from prior requests. The Nios II processor can prefetch sequential instructions and perform branch prediction to keep the instruction pipe as active as possible.

The instruction master port always retrieves 32 bits of data. The instruction master port relies on dynamic bus-sizing logic contained in the system interconnect fabric. By virtue of dynamic bus sizing, every instruction fetch returns a full instruction word, regardless of the width of the target memory. Consequently, programs do not need to be aware of the widths of memory in the Nios II processor system.

The Nios II architecture supports on-chip cache memory for improving average instruction fetch performance when accessing slower memory. Refer to “[Cache Memory](#)” on page 2-12 for details. The Nios II architecture supports tightly-coupled memory, which provides guaranteed low-latency access to on-chip memory. Refer to “[Tightly-Coupled Memory](#)” on page 2-13 for details.

## Data Master Port

The Nios II data bus is implemented as a 32-bit Avalon-MM master port. The data master port performs two functions:

- Read data from memory or a peripheral when the processor executes a load instruction
- Write data to memory or a peripheral when the processor executes a store instruction

Byte-enable signals on the master port specify which of the four byte-lane(s) to write during store operations. When the Nios II core is configured with a data cache line size greater than four bytes, the data master port supports pipelined Avalon-MM transfers. When the data cache line size is only four bytes, any memory pipeline latency is perceived by the data master port as wait states. Load and store operations can complete in a single clock-cycle when the data master port is connected to zero-wait-state memory.

The Nios II architecture supports on-chip cache memory for improving average data transfer performance when accessing slower memory. Refer to “Cache Memory” on page 2-12 for details. The Nios II architecture supports tightly-coupled memory, which provides guaranteed low-latency access to on-chip memory. Refer to “Tightly-Coupled Memory” on page 2-13 for details.

### Shared Memory for Instructions and Data

Usually the instruction and data master ports share a single memory that contains both instructions and data. While the processor core has separate instruction and data buses, the overall Nios II processor system might present a single, shared instruction/data bus to the outside world. The outside view of the Nios II processor system depends on the memory and peripherals in the system and the structure of the system interconnect fabric.

The data and instruction master ports never cause a gridlock condition in which one port starves the other. For highest performance, assign the data master port higher arbitration priority on any memory that is shared by both instruction and data master ports.

## Cache Memory

The Nios II architecture supports cache memories on both the instruction master port (instruction cache) and the data master port (data cache). Cache memory resides on-chip as an integral part of the Nios II processor core. The cache memories can improve the average memory access time for Nios II processor systems that use slow off-chip memory such as SDRAM for program and data storage.

The instruction and data caches are enabled perpetually at run-time, but methods are provided for software to bypass the data cache so that peripheral accesses do not return cached data. Cache management and cache coherency are handled by software. The Nios II instruction set provides instructions for cache management.

### Configurable Cache Memory Options

The cache memories are optional. The need for higher memory performance (and by association, the need for cache memory) is application dependent. Many applications require the smallest possible processor core, and can trade-off performance for size.

A Nios II processor core might include one, both, or neither of the cache memories. Furthermore, for cores that provide data and/or instruction cache, the sizes of the cache memories are user-configurable. The inclusion of cache memory does not affect the functionality of programs, but it does affect the speed at which the processor fetches instructions and reads/writes data.

### Effective Use of Cache Memory

The effectiveness of cache memory to improve performance is based on the following premises:

- Regular memory is located off-chip, and access time is long compared to on-chip memory
- The largest, performance-critical instruction loop is smaller than the instruction cache
- The largest block of performance-critical data is smaller than the data cache

Optimal cache configuration is application specific, although you can make decisions that are effective across a range of applications. For example, if a Nios II processor system includes only fast, on-chip memory (i.e., it never accesses slow, off-chip memory), an instruction or data cache is unlikely to offer any performance gain. As another example, if the critical loop of a program is 2 KBytes, but the size of the instruction cache is 1 KByte, an instruction cache does not improve execution speed. In fact, an instruction cache may degrade performance in this situation.

If an application always requires certain data or sections of code to be located in cache memory for performance reasons, the tightly-coupled memory feature might provide a more appropriate solution. Refer to “[Tightly-Coupled Memory](#)” on page 2-13 for details.

### Cache Bypass Methods

The Nios II architecture provides the following methods for bypassing the data cache:

- I/O load and store instructions
- Bit-31 cache bypass

#### I/O Load and Store Instructions Method

The load and store I/O instructions such as `ldio` and `stio` bypass the data cache and force an Avalon-MM data transfer to a specified address.

#### The Bit-31 Cache Bypass Method

The bit-31 cache bypass method on the data master port uses bit 31 of the address as a tag that indicates whether the processor should transfer data to/from cache, or bypass it. This is a convenience for software, which might need to cache certain addresses and bypass others. Software can pass addresses as parameters between functions, without having to specify any further information about whether the addressed data is cached or not.



To determine which cores implement which cache bypass methods, refer to the [Nios II Core Implementation Details](#) chapter of the *Nios II Processor Reference Handbook*.

## Tightly-Coupled Memory

Tightly-coupled memory provides guaranteed low-latency memory access for performance-critical applications. Compared to cache memory, tightly-coupled memory provides the following benefits:

- Performance similar to cache memory
- Software can guarantee that performance-critical code or data is located in tightly-coupled memory
- No real-time caching overhead, such as loading, invalidating, or flushing memory

Physically, a tightly-coupled memory port is a separate master port on the Nios II processor core, similar to the instruction or data master port. A Nios II core can have zero, one, or multiple tightly-coupled memories. The Nios II architecture supports tightly-coupled memories for both instruction and data access. Each tightly-coupled memory port connects directly to exactly one memory with guaranteed low, fixed latency. The memory is external to the Nios II core and is usually located on chip.

### Accessing Tightly-Coupled Memory

Tightly-coupled memories occupy normal address space, the same as other memory devices connected via system interconnect fabric. The address ranges for tightly-coupled memories (if any) are determined at system generation time.

Software accesses tightly-coupled memory using regular load and store instructions. From the software's perspective, there is no difference accessing tightly-coupled memory compared to other memory.

### Effective Use of Tightly-Coupled Memory

A system can use tightly-coupled memory to achieve maximum performance for accessing a specific section of code or data. For example, interrupt-intensive applications can place exception handler code into a tightly-coupled memory to minimize interrupt latency. Similarly, compute-intensive digital signal processing (DSP) applications can place data buffers into tightly-coupled memory for the fastest possible data access.

If the application's memory requirements are small enough to fit entirely on chip, it is possible to use tightly-coupled memory exclusively for code and data. Larger applications must selectively choose what to include in tightly-coupled memory to maximize the cost-performance trade-off.

## Address Map

The address map for memories and peripherals in a Nios II processor system is design dependent. You specify the address map at system generation time.

There are three addresses that are part of the processor and deserve special mention:

- Reset address
- Exception address
- Break handler address


Programmers access memories and peripherals by using macros and drivers. Therefore, the flexible address map does not affect application developers.

## Memory Management Unit

The optional Nios II MMU provides the following features and functionality:


- Virtual to physical address mapping
- Memory protection
- 32-bit virtual and physical addresses, mapping a 4 GByte virtual address space into as much as 4 GBytes of physical memory
- 4 KByte page and frame size
- Low 512 MBytes of physical address space available for direct access

- Hardware translation lookaside buffers (TLBs), accelerating address translation
  - Separate TLBs for instruction and data accesses
  - Read, write, and execute permissions controlled per page
  - Default caching behavior controlled per page
  - TLBs acting as  $n$ -way set-associative caches for software page tables
  - TLB sizes and associativities configurable at system generation
- Format of page tables (or equivalent data structures) determined by system software
- Replacement policy for TLB entries determined by system software
- Write policy for TLB entries determined by system software
- All exceptions precise

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

You can optionally include the MMU when you instantiate the Nios II processor in your Nios II hardware system. When present, the MMU is always enabled, and the data and instruction caches are virtually-indexed, physically-tagged caches. Several parameters are available, allowing you to optimize the MMU for your system needs.


For complete details of user-selectable parameters for the Nios II MMU, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

 The Nios II MMU is optional and mutually exclusive from the Nios II MPU. Nios II systems can include either an MMU or MPU, but cannot include both an MMU and MPU in the same design.

## Memory Protection Unit


The optional Nios II MPU provides the following features and functionality:

- Memory protection
- Up to 32 instruction regions and 32 data regions
- Variable instruction and data region sizes
- Amount of region memory defined by size or upper address limit
- Read and write access permissions for data regions
- Execute access permissions for instruction regions
- Overlapping regions
- All exceptions precise

 For further details on the MPU implementation, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

You can optionally include the MPU when you instantiate the Nios II processor in your Nios II hardware system. When present, the MPU is always enabled. Several parameters are available, allowing you to optimize the MPU for your system needs.


 For complete details of user-selectable parameters for the Nios II MPU, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

 The Nios II MPU is optional and mutually exclusive from the Nios II MMU. Nios II systems can include either an MPU or MMU, but cannot include both an MPU and MMU in the same design.

## JTAG Debug Module

The Nios II architecture supports a JTAG debug module that provides on-chip emulation features to control the processor remotely from a host PC. PC-based software debugging tools communicate with the JTAG debug module and provide facilities, such as the following features:

- Downloading programs to memory
- Starting and stopping execution
- Setting breakpoints and watchpoints
- Analyzing registers and memory
- Collecting real-time execution trace data

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

The debug module connects to the JTAG circuitry in an Altera® FPGA. External debugging probes can then access the processor via the standard JTAG interface on the FPGA. On the processor side, the debug module connects to signals inside the processor core. The debug module has non-maskable control over the processor, and does not require a software stub linked into the application under test. All system resources visible to the processor in supervisor mode are available to the debug module. For trace data collection, the debug module stores trace data in memory either on-chip or in the debug probe.

The debug module gains control of the processor either by asserting a hardware break signal, or by writing a break instruction into program memory to be executed. In both cases, the processor transfers execution to the routine located at the break address. The break address is specified in SOPC Builder at system generation time.

Soft-core processors such as the Nios II processor offer unique debug capabilities beyond the features of traditional, fixed processors. The soft-core nature of the Nios II processor allows you to debug a system in development using a full-featured debug core, and later remove the debug features to conserve logic resources. For the release version of a product, the JTAG debug module functionality can be reduced, or removed altogether.

The following sections describe the capabilities of the Nios II JTAG debug module hardware. The usage of all hardware features is dependent on host software, such as the Nios II IDE, which manages the connection to the target processor and controls the debug process.

## JTAG Target Connection

The JTAG target connection provides the ability to connect to the processor through the standard JTAG pins on the Altera FPGA. This provides basic capabilities to start and stop the processor, and examine and edit registers and memory. The JTAG target connection is the minimum requirement for the Nios II IDE flash programmer.



While the processor has no minimum clock frequency requirements, Altera recommends that your design's system clock frequency be at least four times the JTAG clock frequency to ensure that the on-chip instrumentation (OCI) core functions properly.

## Download and Execute Software

Downloading software refers to the ability to download executable code and data to the processor's memory via the JTAG connection. After downloading software to memory, the JTAG debug module can then exit debug mode and transfer execution to the start of executable code.

## Software Breakpoints

Software breakpoints allow you to set a breakpoint on instructions residing in RAM. The software breakpoint mechanism writes a break instruction into executable code stored in RAM. When the processor executes the break instruction, control is transferred to the JTAG debug module.

## Hardware Breakpoints

Hardware breakpoints allow you to set a breakpoint on instructions residing in nonvolatile memory, such as flash memory. The hardware breakpoint mechanism continuously monitors the processor's current instruction address. If the instruction address matches the hardware breakpoint address, the JTAG debug module takes control of the processor.

Hardware breakpoints are implemented using the JTAG debug module's hardware trigger feature.

## Hardware Triggers

Hardware triggers activate a debug action based on conditions on the instruction or data bus during real-time program execution. Triggers can do more than halt processor execution. For example, a trigger can be used to enable trace data collection during real-time processor execution.

Table 2-5 lists all the conditions that can cause a trigger. Hardware trigger conditions are based on either the instruction or data bus. Trigger conditions on the same bus can be logically ANDed, enabling the JTAG debug module to trigger, for example, only on write cycles to a specific address.

**Table 2-5.** Trigger Conditions

| Condition           | Bus               | Description   |
|---------------------|-------------------|---|
| Specific address    | Data, Instruction | Trigger when the bus accesses a specific address.   |
| Specific data value | Data              | Trigger when a specific data value appears on the bus.  |
| Read cycle          | Data              | Trigger on a read bus cycle.  |
| Write cycle         | Data              | Trigger on a write bus cycle.   |
| Armed               | Data, Instruction | Trigger only after an armed trigger event. Refer to “Armed Triggers” on page 2-18.                                  |
| Range               | Data              | Trigger on a range of address values, data values, or both. Refer to “Triggering on Ranges of Values” on page 2-18. |

When a trigger condition occurs during processor execution, the JTAG debug module triggers an action, such as halting execution, or starting trace capture. Table 2-6 lists the trigger actions supported by the Nios II JTAG debug module.

**Table 2-6.** Trigger Actions

| Action           | Description  |
|------------------|--|
| Break            | Halt execution and transfer control to the JTAG debug module.  |
| External trigger | Assert a trigger signal output. This trigger output can be used, for example, to trigger an external logic analyzer. |
| Trace on         | Turn on trace collection.  |
| Trace off        | Turn off trace collection.   |
| Trace sample (1) | Store one sample of the bus to trace buffer.   |
| Arm              | Enable an armed trigger.   |

**Notes to Table 2-6:**

(1) Only conditions on the data bus can trigger this action.

**Armed Triggers**

The JTAG debug module provides a two-level trigger capability, called armed triggers. Armed triggers enable the JTAG debug module to trigger on event B, only after event A. In this example, event A causes a trigger action that enables the trigger for event B.

**Triggering on Ranges of Values**

The JTAG debug module can trigger on ranges of data or address values on the data bus. This mechanism uses two hardware triggers together to create a trigger condition that activates on a range of values within a specified range.

**Trace Capture**

Trace capture refers to ability to record the instruction-by-instruction execution of the processor as it executes code in real-time. The JTAG debug module offers the following trace features:

- Capture execution trace (instruction bus cycles).
- Capture data trace (data bus cycles).

- For each data bus cycle, capture address, data, or both.
- Start and stop capturing trace in real time, based on triggers.
- Manually start and stop trace under host control.
- Optionally stop capturing trace when trace buffer is full, leaving the processor executing.
- Store trace data in on-chip memory buffer in the JTAG debug module. (This memory is accessible only through the JTAG connection.)
- Store trace data to larger buffers in an off-chip debug probe.

Certain trace features require additional licensing or debug tools from third-party debug providers. For example, an on-chip trace buffer is a standard feature of the Nios II processor, but using an off-chip trace buffer requires additional debug software and hardware provided by First Silicon Solutions (FS2) or Lauterbach GmbH.



For details, refer to the FS2 website ([www.fs2.com](http://www.fs2.com)) and the Lauterbach GmbH website ([www.lauterbach.com](http://www.lauterbach.com)).

### Execution vs. Data Trace

The JTAG debug module supports tracing the instruction bus (execution trace), the data bus (data trace), or both simultaneously. Execution trace records only the addresses of the instructions executed, enabling you to analyze where in memory (i.e., in which functions) code executed. Data trace records the data associated with each load and store operation on the data bus.

The JTAG debug module can filter the data bus trace in real time to capture the following:

- Load addresses only
- Store addresses only
- Both load and store addresses
- Load data only
- Load address and data
- Store address and data
- Address and data for both loads and stores
- Single sample of the data bus upon trigger event

### Trace Frames

A frame is a unit of memory allocated for collecting trace data. However, a frame is not an absolute measure of the trace depth.

To keep pace with the processor executing in real time, execution trace is optimized to store only selected addresses, such as branches, calls, traps, and interrupts. From these addresses, host-side debug software can later reconstruct an exact instruction-by-instruction execution trace. Furthermore, execution trace data is stored in a compressed format, such that one frame represents more than one instruction. As a result of these optimizations, the actual start and stop points for trace collection during execution might vary slightly from the user-specified start and stop points.

Data trace stores 100% of requested loads and stores to the trace buffer in real time. When storing to the trace buffer, data trace frames have lower priority than execution trace frames. Therefore, while data frames are always stored in chronological order, execution and data trace are not guaranteed to be exactly synchronized with each other.

## Referenced Documents

This chapter references the following documents:

- *Programming Model* chapter of the *Nios II Processor Reference Handbook*
- *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*
- *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*
- *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*
- *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Custom Instruction User Guide*
- *Avalon Interface Specifications*
- *AN 391: Profiling Nios II Systems*
- *Literature: Megafunctions* page on the Altera website

## Document Revision History

Table 2-7 shows the revision history for this document.

**Table 2-7.** Document Revision History (Part 1 of 2)

| Date & Document Version | Changes Made  | Summary of Changes |
|-------------------------|---|--------------------|
| March 2009<br>v9.0.0    | Maintenance release.  | —                  |
| November 2008<br>v8.1.0 | <ul style="list-style-type: none"> <li>■ Expanded floating-point instructions information.</li> <li>■ Updated description of optional <code>cpu_resetrequest</code> and <code>cpu_resettaken</code> signals.</li> <li>■ Added description of optional <code>debugreq</code> and <code>debugack</code> signals.</li> </ul> | —                  |
| May 2008<br>v8.0.0      | Added MMU and MPU sections.   | Added MMU and MPU. |

**Table 2-7.** Document Revision History (Part 2 of 2)

| Date & Document Version | Changes Made   | Summary of Changes                         |
|-------------------------|--|--|
| October 2007<br>v7.2.0  | Maintenance release.   | —  |
| May 2007<br>v7.1.0      | <ul style="list-style-type: none"> <li>■ Added table of contents to Introduction section.</li> <li>■ Added Referenced Documents section.</li> </ul>  | —  |
| March 2007<br>v7.0.0    | Maintenance release.   | —  |
| November 2006<br>v6.1.0 | Described interrupt vector custom instruction.   | Interrupt vector custom instruction added. |
| May 2006<br>v6.0.0      | <ul style="list-style-type: none"> <li>■ Added description of optional <code>cpu_resetrequest</code> and <code>cpu_resettaken</code>.</li> <li>■ Added section on single precision floating-point instructions.</li> </ul> | —  |
| October 2005<br>v5.1.0  | Maintenance release.   | —  |
| May 2005<br>v5.0.0      | Added tightly-coupled memory.  | —  |
| December 2004<br>v1.2   | Added new control register <code>ctl5</code> .   | —  |
| September 2004<br>v1.1  | Updates for Nios II 1.01 release.  | —  |
| May 2004<br>v1.0        | Initial release.   | —  |

