

Introduction

The 3.0 version of the Nios[®] CPU is a pipelined general-purpose RISC microprocessor. The Nios processor supports both 32-bit and 16-bit architectural variants and both 16 and 32-bit variants use 16-bit instructions. The principal features of the Nios instruction set architecture are:

- *Large, windowed register file*—The Nios CPU implementations can include up to 512 internal general-purpose registers. The compiler uses the internal registers to accelerate subroutine calls and local variable access.
- *Simple, complete instruction set*—Both 32-bit and 16-bit Nios CPU variants use 16-bit-wide instructions. 16-bit instructions reduce code footprint and instruction-memory bandwidth.
- *Powerful addressing modes*—The Nios instruction set includes Load and Store instructions that the compiler uses to accelerate structure access and local-variable (stack) access.
- *Extensibility*—Users can incorporate custom logic directly into the Nios arithmetic logic unit (ALU). The automatically-generated software development kit (SDK) includes macros for accessing custom instruction hardware for C and assembly-language programs.



See the *Nios Embedded Processor 16-Bit Programmer's Reference Manual* and the *Nios Embedded Processor 32-Bit Programmer's Reference Manual* for an extensive discussion of the Nios 32-bit and 16-bit instruction set architecture. The latest revisions of all Nios documentation are at <http://www.altera.com/literature/lit-nio.html>.

Nios 3.0 CPU Implementation Details

The implementation of Nios processor version 3.0 is a five-stage pipeline with separate instruction and data-memory masters (Harvard memory architecture). Both instruction and data-memory control ports are implemented as Avalon™ bus masters. Using the SOPC Builder system design tool, you can specify connections between both Nios bus masters and any Avalon slaves in your system such as memory and peripherals. SOPC Builder automatically inserts arbiters as required.



See the *Avalon Bus Specification Reference Manual* for details on the signals and timing of Avalon bus transfers. See the *System Design using SOPC Builder* chapter in the *Quartus II Development Software Handbook, Volume 1* for details on connecting and generating Nios processor systems.

Instruction Bus-Master

The Nios instruction master is a 16-bit wide, latency-aware Avalon bus master. This master is only used to fetch instructions from memory that will be executed by the Nios CPU. This master never performs any write operations. Because this master is latency-aware, it can perform posted-read operations to latent memory devices. The instruction master issues new read requests before data is returned by the bus. Nios uses a branch-not-taken prediction scheme to issue speculative addresses. Native support for latency minimizes the impact of latent memory and increases the overall f_{Max} of the system as a whole. Penalties occur only for mis-predicted branches. The Nios instruction master can optionally implement an on-chip cache memory for improving average instruction fetch performance when accessing slower memory. See “[Cache Memory](#)” on page 3.

The Avalon bus generated by SOPC Builder automatically includes dynamic bus-sizing logic. Consequently, the Nios instruction master can be connected to 8- 16- and 32-bit-wide memory.



See the *Avalon Bus Specification Reference Manual* for details on the signals and timing of Avalon bus transfers.

Data Bus-Master

The Nios data master is 3.0 bits wide for 32-bit architecture variants and 16-bits wide for 16-bit architecture variants. The data master is used for three purposes:

1. To fetch data from memory when the CPU executes a load-instruction (LD, LDP, LDS).
2. To write data to memory when the CPU executes a store-instruction (ST, STP, STS, ST8s, ST8d, ST16s, ST16d, STS8s, STS16s).
3. To fetch interrupt-vectors from the interrupt-vector table when the CPU executes a TRAP instruction or processes an internal or external exception.

Because it is not meaningful to predict data addresses or continue execution before data is retrieved, the Nios data master does not support latency. Consequently, slave latency is seen by the data master as wait-states. Load and store operations can be completed in a single clock-cycle when the Nios data master is connected to zero-wait-state memory. This assumes there is no arbitration conflicts with other masters, including the CPU's own instruction-master.

For highest performance, the data master should be assigned higher arbitration priority on any slave that is shared by both instruction and data masters.

Cache Memory

The Nios CPU has optional cache memories on both the instruction master (instruction cache) and the data master (data cache). Users enable the hardware for instruction cache, data cache, or both, using the Nios processor configuration wizard in SOPC Builder. The cache memory size is configurable. Cache memory will reside on-chip.

Both instruction cache and data cache are direct-mapped caches, the simplest implementation. Instructions and control registers in the Nios architecture provide facilities for the user to enable and disable instruction cache and data cache usage at runtime, as well as invalidate entries in the cache memory.

In the event of a cache hit when the cache is enabled, memory load operations execute in a single clock cycle. Cache misses introduce additional cycles of latency. When the cache is disabled (either by disabling the cache hardware or by temporarily disabling the cache in software), memory accesses do not incur extra latency. When the cache is enabled, memory store operations incur one or two additional cycles of latency.



See the *Nios Embedded Processor 32-Bit Programmer's Reference Manual* for information on using the instruction and data cache memories.



The cache memory is only available in the 32-bit Nios CPU architecture. The Nios cache memories can only be used when targeting the Stratix[®] II, Stratix, Stratix GX, or Cyclone[™] device families.

Shift Unit

The Nios processor uses fixed (non-configurable) barrel-shifter logic that executes all shift instructions (ASR, ASRI, ASL, ASLI, LSL, LSLI, RLC and RRC) in two clock cycles, regardless of the shift-distance.

Multiply Support

The Nios processor supports three different options for implementing integer multiplication operations from software:

1. *The MUL Instruction*—A 32-bit Nios CPU can optionally be configured to include a hardware 16x16 → 32 integer multiplier. This hardware multiplier is used by the MUL instruction to compute a 32-bit result in three or less clock cycles. When the MUL option is selected, the MUL instruction uses the C-runtime libraries in the automatically-generated SDK to implement multiplication.



This option is not supported in the 16-bit Nios instruction set.

The MUL instruction takes 1 to 3 clock cycles to compute a 32-bit result, depending on the device family used. For example, when using the MUL option in Altera® Stratix® devices, the hardware multiplier is implemented in the Stratix fast DSP blocks, and the operation can complete in a single clock cycle. In other device families, the hardware multiplier is implemented in general logic elements, the MUL instruction may take up to 3 clock cycles to complete.

2. *The MSTEP Instruction*— A 32-bit Nios CPU can optionally be configured to include hardware for executing one step of a 16x16 multiply. This hardware is used by the MSTEP instruction to produce a partial multiplication result in two clocks. When the MSTEP option is selected, the C-runtime libraries in the automatically-generated SDK implements multiplication operations. It does this by issuing strings of successive MSTEP operations such as 16-sequential MSTEPs to perform a 16x16 → 32 multiplication. The amount of logic resources needed to implement MSTEP is less than 5% of the total CPU logic. Since the amount is so small, the MSTEP option is the default.



This option is not supported in the 16-bit Nios instruction set.

3. *Software Multiplication*—The C-runtime libraries in the automatically generated SDK implement integer multiplication with sequences of shift and add instructions when neither MSTEP nor MUL options are enabled. Software multiplication gives the smallest possible CPU logic utilization, but the slowest execution of multiplication operations.

Interrupt Support

Version 3.0 of the Nios processor allows optional removal of support for TRAP instructions, hardware interrupts, and internal exceptions. This option is intended for use only in Nios systems where the processor is implementing a very simple control function (that is, not running complex software). When so configured, the Nios processor will not:

- Include an irq input-pin
- Process exceptions (TRAP instruction behavior is undefined)
- Generate exceptions when a SAVE or RESTORE instruction underflows/overflows the register file

Select this option only if:

- You want the smallest possible Nios CPU core.
- You know absolutely that your application software does not generate register-window underflow/overflow exceptions. That is, the subroutine call-depth is less than the number of available register-windows.
- You do not have any hardware interrupt sources.
- Your assembly-language code does not include TRAP instructions.



The Nios SDK does not generate TRAP instructions in compiled code. By default, interrupt support is enabled. TRAPs, interrupts, and exceptions work exactly as described in the *Nios Embedded Processor 16-Bit Programmer's Reference Manual* and the *Nios Embedded Processor 32-Bit Programmer's Reference Manual*.

Nios On-Chip Instrumentation Debug Module

The Nios CPU has an optional on-chip JTAG debug module that allows communication with the CPU via the standard JTAG pins on the Altera device. The debug module is an intellectual property (IP) core designed by First Silicon Solutions (FS2) Inc. It is implemented as an FS2 On-Chip Instrumentation (OCI™) block that connects directly to signals internal to the Nios CPU. Hereafter, the debug module is referred to as the Nios OCI debug module.

The Nios OCI debug module may be configured to implement run control, hardware breakpoints and software trace. The OCI debug module stores trace data in either on-chip memory or in an external system analyzer (for example, the ISA-NIOS in-system analyzer by FS2). The Nios OCI debug module can read and write registers and memory, allowing easy download of software to memory and examination of registers during program execution. The Nios OCI debug module has non-maskable control of CPU execution. It does not require a software stub to operate, and can seize control at any time during software execution.

Some advanced debug features require external JTAG-based debug hardware and/or a third-party debugger. In addition to debug functionality, the Nios OCI debug module can be used for stdio. See “[Software Subroutines](#)” for more details.



See the *Nios Embedded Processor Software Development Reference Manual* for information about using the Nios OCI debug module for software debugging.

Software Subroutines

The following sections describes the Nios library software subroutines available for using the Nios OCI debug module and for manipulating the Nios instruction and data caches.

JTAG Character I/O Subroutines

The Nios CPU can communicate with a host computer connected to the OCI debug module via a JTAG download cable. When the Nios OCI debug module is enabled and Host Communication (`maincomm_slave`) is set to `oci_core` in SOPC Builder, the SDK generated by SOPC Builder includes software subroutines for stdio via the OCI debug module. The subroutines do not conflict with JTAG debug, and can be used simultaneously while the OCI debug module is performing debug operations.



Using JTAG for character I/O is significantly slower than UART communication.

Table 1 lists the software subroutines available in the custom SDK library. These functions are declared in the SDK include file **excalibur.h**. `nasys_printf_uart` (in **excalibur.h**) will be defined to be the base address of the Nios OCI debug module (even though the OCI debug module is not a UART). Therefore, `printf` can be used to send formatted strings to a host computer via the OCI debug module.

<i>Table 1. JTAG Software Subroutines</i>	
Subroutine	Description
<code>nr_jtag_rxchar</code>	Reads a character from the Nios OCI debug module whose base address is passed as an argument.
<code>nr_jtag_tx_ready</code>	Returns non-zero (true) if JTAG IO is ready to transmit a character, otherwise zero (false).
<code>nr_jtag_txchar</code>	Sends a single character to the Nios OCI debug module whose base address is passed as an argument.
<code>nr_jtag_txcr</code>	Sends a carriage return and line feed to the Nios OCI debug module with base address defined by <code>nasys_printf_uart</code>
<code>nr_jtag_txhex</code>	Prints an integer value, in hexadecimal, to the Nios OCI debug module with base address defined by <code>nasys_printf_uart</code> .
<code>nr_jtag_txhex16</code>	Prints the value of a short integer, in hexadecimal, to the Nios OCI debug module with base address defined by <code>nasys_printf_uart</code> .
<code>nr_jtag_txhex32</code>	Prints the value of a long integer, in hexadecimal, to the Nios OCI debug module with base address defined by <code>nasys_printf_uart</code> .
<code>nr_jtag_txstring</code>	Prints a null-terminated string to the Nios OCI debug module with base address defined by <code>nasys_printf_uart</code> .



See the *Nios Embedded Processor Software Development Reference Manual* for `printf` implementation details and communication with a host computer.

nr_jtag_rxchar

This subroutine reads a character from the Nios OCI debug module. The Nios OCI debug module's base address is passed in as pointer `ociBase`. If no character is waiting, `nr_jtag_rxchar` returns -1. If zero is passed for the base address, `nr_jtag_rxchar` reads a character from the Nios OCI debug module at the base address defined by `nasys_printf_uart` (in **excalibur.h**).

Syntax

```
int nr_jtag_rxchar( const void *ociBase );
```

Parameter

The `ociBase` parameter is a pointer to the Nios OCI debug module.

Example

```
#include "excalibur.h"

void main()
{
    int c;
    printf( "Please enter a character:\n" );

    while( ( c = nr_jtag_rxchar( 0 ) ) == -1 )
        ; // wait for valid input

    printf( "Your character is:\t%c\n", c );
}
```

nr_jtag_tx_ready

This subroutine returns a non-zero value (true) if JTAG IO is ready to transmit a character. If JTAG IO is not ready to transmit (because the host is not receiving previously transmitted characters), it will return zero (false). The Nios OCI debug module base address should be passed as `ociBase`. If zero is passed as the base address, the base address will default to `nasys_printf_uart` (in **excalibur.h**).

Syntax

```
int nr_jtag_tx_ready( const void* ociBase );
```

Parameter

The `ociBase` parameter is a pointer to the Nios OCI debug module.

Example

```
#include "excalibur.h"

#define kLineWidth  77
#define kLineCount  100

#define kTimeout    1000

int my_jtag_txchar( int c, int timeout )
{
    int ms = nr_timer_milliseconds();
```

```

int ready = 0;

while ( nr_timer_milliseconds() < ( ms + timeout ) )
{
    if ( ready = nr_jtag_tx_ready( na_cpu_oci_core ) )
        break;
}

if ( ready )
    nr_jtag_txchar( c, na_cpu_oci_core );
return ready;
}

int my_jtag_txcr( int timeout )
{
    if ( ! my_jtag_txchar( 13, timeout ) )
        return 0;

    return my_jtag_txchar( 10, timeout );
}

int main()
{
    char c;
    int i,j;
    int mix;

    // install the timer ISR and start timing
    nr_timer_milliseconds();

    printf( "\n\nPress character, or <space> for mix:" );

    while( ( c = nr_rxchar() ) < 0 )
;

    printf( "%c\n\n",c );

    // Don't show unprintables
    if( c < 32 )
        c = '.';

    mix = c==' ';

    for( i = 0; i < kLineCount; i++ )
    {
        for( j = 0; j < kLineWidth; j++ )
        {
            if( mix )
            {
                c++;

                if( c >= 127 )
                    c = 33;
            }
            if ( ! my_jtag_txchar( c, kTimeout ) )
            {
                printf( "ERROR: my_jtag_txchar exceeded timeout (%i
ms)\n", kTimeout );
                return;
            }
        }
    }
}

```

```

    }
    my_jtag_txcr( kTimeout );
  }
  my_jtag_txcr( kTimeout );
  my_jtag_txcr( kTimeout );

  printf( "\n\nDone.\n" );
}

```

nr_jtag_txchar

This subroutine sends a single character, *c*, to the Nios OCI debug module whose base address is passed as *ociBase*. If zero is passed for the base address, *nr_jtag_txchar* sends a character to the Nios OCI debug module at the base address defined by *nasys_printf_uart* (in **excalibur.h**).



nr_jtag_rxchar will not return until the OCI transmit readybit is clear. If the host is not receiving characters, *nr_jtag_txchar* will loop forever. This will affect all other subroutines *nr_jtag_tx* that call *nr_jtag_txchar*.

Syntax

```
void nr_jtag_txchar( int c, const void *ociBase );
```

Parameters

<i>nr_jtag_txchar</i> Parameters	
Parameter Name	Description
<i>c</i>	Character to be sent
<i>ociBase</i>	Pointer to the Nios OCI debug module

nr_jtag_txcr

This subroutine sends a carriage return and line feed to the Nios OCI debug module with base address defined by *nasys_printf_uart* (in **excalibur.h**).

Syntax

```
void nr_jtag_txcr( void );
```

nr_jtag_txhex

This subroutine prints the integer value of *x* in hexadecimal to the Nios OCI debug module with base address defined by `nasys_printf_uart` (in **excalibur.h**). The range for a 16-bit Nios CPU is 0x0000 to 0xFFFF, and for a 32-bit Nios CPU is 0x00000000 to 0xFFFFFFFF.

Syntax

```
void nr_jtag_txhex( int x );
```

Parameter

The *x* parameter is an integer value to be sent to the Nios OCI debug module.

nr_jtag_txhex16

This subroutine prints the 16-bit value of *x* in hexadecimal to the Nios OCI debug module with base address defined by `nasys_printf_uart` (in **excalibur.h**). The range is from 0x0000 to 0xFFFF.

Syntax

```
void nr_jtag_txhex16( short x );
```

Parameter

The *x* parameter is a 16-bit integer value to be sent to the Nios OCI debug module.

nr_jtag_txhex32

This subroutine prints the 32-bit value of *x* in hexadecimal to the to Nios OCI debug module with base address defined by `nasys_printf_uart` (in **excalibur.h**). The range is from 0x00000000 to 0xFFFFFFFF. This subroutine is not available on a 16-bit Nios CPU.

Syntax

```
void nr_jtag_txhex32( long x );
```

Parameter

The *x* parameter is a 32-bit integer value to be sent to the Nios OCI debug module.

nr_jtag_txstring

This subroutine prints the null-terminated string *s* to the Nios OCI debug module with base address defined by `nasys_printf_uart` (in **excalibur.h**).

Syntax

```
void nr_jtag_txstring( const char *s );
```

Parameter

The *s* parameter is a pointer to a null-terminated character string.

Cache Management Software Subroutines

The job of the data cache and one of the jobs of the `nios-elf-gcc` optimizing compiler is to reduce bus traffic to memory. Both the data cache and the compiler do this by keeping track of recent reads and writes to particular memory locations, and using those recent reads—or even implied values—instead of actual memory accesses when possible.

This strategy assumes that memory doesn't change except when the software running changes it. This strategy fails if the memory is shared with another CPU that can change it, or if the locations accessed are hardware peripherals.

For such accesses, you must tell the compiler both not to optimize and not to access the data cache by declaring the type of accessed memory `volatile`. This causes the compiler to always access the memory, and also – if the `-mdcache` option is used, to emit `PFXIO` instructions.

Here is an example of the `volatile` keyword for a peripheral structure.

```
// UART Registers
typedef volatile struct
{
int np_uartrxdata;      // Read-only, 8-bit
int np_uarttxdata;     // Write-only, 8-bit
int np_uartstatus;    // Read-only, 8-bit
int np_uartcontrol;   // Read/Write, 9-bit
int np_uartdivisor;   // Read/Write, 16-bit, optional
int np_uartendofpacket; // Read/Write, end-of-packet character
} np_uart;
```

The following subroutines are available in the Nios library for manipulating the Nios instruction and data caches. These subroutines only work correctly in a Nios system which has been built with the corresponding cache enabled.

void nr_ichache_init(void);

This subroutine clears the instruction cache and enables it. The Nios library calls this subroutine before your main() subroutine on systems which have an instruction cache.

void nr_dcache_init(void);

This subroutine clears the data cache and enables it. The Nios libraries call this subroutine before your main() subroutine on systems which have a data cache.

*void nr_ichache_invalidate_lines(void *low_address, void *high_address);*

This subroutine clears a portion or all of the instruction cache that corresponds to the address range passed. The range includes the low address and excludes the high address.

*void nr_dcache_invalidate_lines(void *low_address, void *high_address);*

This subroutine clears a portion or all of the data cache that corresponds to the address range passed. The range includes the low address excludes the high address.

void nr_dcache_enable(void);

This subroutine enables the data cache.

void nr_dcache_disable(void);

This subroutine disables the data cache.

void nr_ichache_enable(void);

This subroutine enables the instruction cache.

void nr_ichache_disable(void);

This subroutine disables the instruction cache.

PTF Assignments

The Nios 3.0 CPU configuration options are determined by assignments in your system PTF file. Each Nios CPU will have a corresponding MODULE section in your system PTF containing the assignment:

```
class = "altera_nios";
```

Table 2 lists the CPU's PTF assignments followed by a detailed description. These assignments are found in the MODULE/WIZARD_SCRIPT_ARGUMENTS section unless otherwise noted and are used by the Nios generator-program to control HDL and SDK generation.

Assignment	Type	Allowed Values	Default	Requires
CPU-related				
cache_dcache_size_k	Integer	(1)	0	cache_has_dcache
cache_has_dcache	Boolean	1, 0	0	
cache_has_icache	Boolean	1, 0	0	
cache_icache_size_k	Integer	(1)	0	cache_has_icache
CPU_Architecture	String	nios_32, nios_16	nios_32	
implement_forward_b1	Boolean	1, 0	1	
mstep	Boolean	1, 0	1	
multiply	Boolean	1, 0	0	
num_regs	Integer	128, 256, 512	256	
reset_offset	hexadecimal integer (byte-offset)	any hexadecimal integer in the range 0-0xFFFFFFFF	None - (must be specified)	
reset_slave	String (module/slave section path name)	slash-delimited ("/") path to slave section	None - (must be specified)	
rom_decoder	Boolean	1, 0	1	
support_interrupts	Boolean	1, 0	1	
support_rlc_rcc	Boolean	1, 0	0	
vecbase_offset	hexadecimal integer (byte-offset)	any hexadecimal integer in the range 0-0xFFFFFFFF	None - (must be specified)	
vecbase_slave	String (module/slave section path name)	slash-delimited ("/") path to slave section	None - (must be specified)	
wvalid_wr	Boolean	1, 0	0	

<i>Table 2. CPU PTF Assignments (Part 2 of 2)</i>				
Assignment	Type	Allowed Values	Default	Requires
SDK-related				
datamem_slave	String (module/slave section path name)	slash-delimited ("/") path to slave section	None - (must be specified)	
debugcomm_slave	String (module/slave section path name)	slash-delimited ("/") path to slave section	None - (must be specified)	
germs_monitor_id	String (10 characters or fewer)	Any ASCII string, max 10 characters	"" (empty string)	
maincomm_slave	String (module/slave section path name)	slash-delimited ("/") path to slave section	None - (must be specified)	
mainmem_slave	String (module/slave section path name)	slash-delimited ("/") path to slave section	None - (must be specified)	
Debug-related				
include_oci				
Other				
smart_recompile	Boolean	1, 0	0	

(1) Any power of 2 (i.e. 0,1,2,4,8,16,32). If `cache_hex_(x)` cache is on, the cpu assignment must be > 0.

cache_dcache_size_k Assignment

This assignment assigns the data cache size, in kilobytes. Acceptable values for this assignment must be powers of two. For instance, a value of 1 creates a data cache of 1 KB (1024 bytes), which equates to 256 cache lines (one 32-bit word per line). The data cache may not be bigger than one quarter of the addressable space of the Nios data master. For instance, if the data master can address locations between `0x0` and `0xffff` (a 4 KB range), the data cache size is restricted to 1 KB. Thus, the upper limit of the cache size is design-dependent.

cache_has_dcachel Assignment

This assignment enables use of the data cache. The data cache option is only valid for the 32-bit Nios architecture; a 16-bit Nios CPU cannot enable the `cache_has_dcachel`. This assignment must be used in conjunction with the `cache_dcachel_size_k` assignment. The data cache option is only available when targeting the Stratix II, Stratix, Stratix GX, and Cyclone device families.

cache_has_icachel Assignment

This assignment enables use of the instruction cache. The instruction cache option is only valid for the 32-bit Nios architecture; a 16-bit Nios CPU cannot enable `cache_has_icachel`. This assignment must be used in conjunction with the `cache_icachel_size_k` assignment. The data cache option is only available when targeting the Stratix II, Stratix, Stratix GX, and Cyclone devices families.

cache_icachel_size_k Assignment

This assignment assigns the instruction cache size, in kilobytes. Acceptable values for this assignment must be powers of two. For instance, a value of 1 creates a data cache of 1 KB (1024 bytes), which equates to 512 cache lines (one 16-bit instruction word per line). The data cache may not be bigger than one quarter of the addressable space of the Nios instruction master. For instance, if the instruction master can addresses locations between 0x0 and 0xfff (a 4 KB range), the instruction cache size is restricted to 1 KB. Thus, the upper limit of the cache size is design-dependent.

CPU_Architecture Assignment

This assignment selects which architecture variant (32-bit/16-bit) will be generated. When `CPU_Architecture` is `nios_16`, logic will be generated that implements the 16-bit instruction set architecture (16-bit-wide registers, ALU, and data bus-master), as described in the *Nios Embedded Processor 16-Bit Programmer's Reference Manual*. Also, the `Data_Width` assignment in the MASTER data master/SYSTEM_BUILDER_INFO section must be set to 16.

When `CPU_Architecture` is `nios_32`, logic will be generated that implements the 32-bit instruction set architecture (32-bit-wide registers, ALU, and data bus-master), as described in the *Nios Embedded Processor 32-Bit Programmer's Reference Manual*. Also, the `Data_Width` assignment in the MASTER data master/SYSTEM_BUILDER_INFO section must be set to 32.

In either case, the SDK associated with this CPU will be generated with the correct libraries and default compiler options necessary to create software for the selected architecture variant.

implement_forward_b1

This assignment controls the CPU's handling of certain types of pipeline-data hazards. The CPU always hides all pipeline implementation details from software, and behaves exactly as indicated in the *Nios Embedded Processor 32-Bit Programmers' Reference Manual* and *Nios Embedded Processor 16-Bit Programmer's Reference Manual* regardless of this setting.

When `implement_forward_b1 = 1`, the CPU will include a data-forwarding multiplexer on the B input of the ALU. This allows the CPU to process an instruction stream where the B operand of an instruction is modified by the previous instruction without introducing a stall cycle in between. This reduces the number pipeline stalls at the expense of some additional logic (~32 LEs) and a potential reduction in f_{Max} .

Most compiled code does not encounter this class of data hazard very often. Performance on the Dhrystone benchmark (at a fixed clock frequency) is improved by about three percent when `implement_forward_b1 = 1`. When `implement_forward_b1 = 0`, the CPU will stall for one clock-cycle when the operand of an instruction was modified by its immediate predecessor.

mstep Assignment

This assignment may be set to 1 only for 32-bit Nios cores. When `mstep = 1`, the generated Nios CPU includes hardware that implements the MSTEP instruction. The generated SDK makes use of MSTEP in the C-runtime libraries. When `mstep = 0`, the MSTEP instruction is not implemented, and the C-runtime libraries will not use it.

multiply Assignment

This assignment may be set to 1 only for 32-bit Nios cores. When `multiply = 1`, the generated Nios CPU includes hardware that implements the MUL instruction, and the generated SDK makes use of MUL in the C-runtime libraries. When `multiply = 0`, the MUL instruction is not implemented, and the C-runtime libraries will not use it. Issuing the MUL instruction when `multiply = 0` produces an undefined result.

num_regs Assignment

This assignment value determines the total number of registers implemented in the CPU's windowed-register file. Larger register files use more on-chip memory resources. Larger register files allow deeper subroutine-nesting before a register-window underflow exception is generated. With 128 registers, software can call subroutines 7-levels deep before an underflow exception is generated. With 512 registers, software can call subroutines 31-levels deep before an underflow exception is generated. Software that frequently generates underflow/overflow exceptions may experience significant performance degradation. In practice, even complicated software can live comfortably in 512 registers although recursive algorithms can cause even simple software to execute very deep-subroutine nesting. The best setting for `num_regs` depends on the complexity and performance requirements of the application software and the amount of on-chip memory resources available.

reset_offset Assignment

See “[reset_slave Assignment](#)” on page 18 for `reset_offset` assignment information.

reset_slave Assignment

The Nios CPU logic is generated with a built-in reset address. The value of the reset address is derived from two PTF assignments: `reset_slave` and `reset_offset`. The `reset_slave` assignment is used to extract a base-address from the slave port of the specified slave-port on the specified module. The `reset_offset` assignment specifies a byte-offset that is applied to the indicated device's base-address.

`<reset-address> = base-address-of-reset_slave + reset_offset;`

After system reset is asserted and released, the CPU will fetch its first instruction from the computed reset-address. For example, consider a system with a Nios CPU and an on-chip memory module named `boot_rom` mapped at base-address `0x400`. In this example, suppose the on-chip memory module has a single slave-port named `s1`. Suppose that this memory contains the CPU's startup program based at the beginning of the memory.

To construct a CPU which executes this startup program at reset, that is with a reset-address of `0x400`, set the following PTF-assignments:

```
reset_slave = "boot_rom/s1";  
  
reset_offset = "0x0";
```

rom_decoder Assignment

The Nios CPU includes internal instruction-decode logic that takes the incoming 16-bit instruction word as input and produces approximately 80 control signals that determine the operation performed by the data path logic. A significant portion of the instruction-decoder logic can be implemented either as logic elements (LEs) or as an internal ROM memory. When `rom_decoder = 1`, embedded memory is used in the instruction logic, saving approximately 52-58 LEs. The exact savings depends on which other CPU options have been selected. When `rom_decoder = 0`, no memory resources are used in the instruction decoder.

support_interrupts Assignment

When the assignment `support_interrupts = 0`, the Nios CPU will have neither `irq` nor `irqnumber` input signals on its Avalon bus master interface and therefore cannot receive hardware interrupts. The CPU will not generate internal exceptions for register-window underflows and overflows. The results of executing a TRAP instruction is undefined.

In short, the CPU does not support interrupts, exceptions, or TRAP instructions. See [“Interrupt Support” on page 5](#) for a description of when systems may use a CPU with no interrupt support.

When `support_interrupts = 1`, the Nios CPU will support interrupts, exceptions, and traps exactly as described in the *Nios Embedded Processor 32-Bit Programmer’s Reference Manual* and *Nios Embedded Processor 16-Bit Programmer’s Reference Manual*. By default, interrupt support is enabled, and `support_interrupts` is 1.

support_rlc_rrc Assignment

These instructions rotate a register right or left (respectively) by one bit through the carry flag. These instructions are never used by the compiler. Support for these instructions requires some dedicated logic (between 12-21) that can usually be saved. When assignment `support_rlc_rrc = 0`, the CPU does not support these instructions, and the behavior of both RRC and RLC instructions is undefined. When assignment `support_rlc_rrc = 1`, the CPU supports RLC and RRC instructions. Use this setting if you have written any assembly-language software that explicitly uses these instructions.



See the complete Nios instruction set including the RRC and RLC instructions in the *Nios Embedded Processor 32-Bit Programmer’s Reference Manual* and *Nios Embedded Processor 16-Bit Programmer’s Reference Manual*.

vecbase_offset Assignment

See “[vecbase_slave Assignment](#)” on page 20 for more information.

vecbase_slave Assignment

The Nios CPU logic is generated with a built-in vector-table base address. The vector-table address is derived from two PTF assignments: `vecbase_slave` and `vecbase_offset`. The `vecbase_slave` assignment is used to extract a base address from the slave port of the specified slave port on the specified module. The `vecbase_offset` assignment specifies a byte offset which is applied to the indicated devices' base address.

$$\langle \text{vector-table-base-address} \rangle = \text{base-address-of-vecbase_slave} + \text{vecbase_offset};$$

Whenever an exception is processed due to external hardware interrupt, internal exception, or execution of a TRAP instruction, the CPU fetches exception-handler addresses from the vector table at this computed address. For example, consider a system with a Nios CPU and a 4K memory module named `general_purpose_ram` mapped at base address `0x1000`. In this example, suppose the on-chip memory module has a single slave port named `s1`. Suppose the highest 256 bytes, the size of the vector-table for 32-bit Nios CPU, is used as the vector table. To construct a CPU that uses the memory region starting at `0x1F00` as its vector table, set the following PTF assignments:

```
vecbase_slave = "general_purpose_ram/s1";
```

```
vecbase_offset = "0xF00";
```



The *Nios Embedded Processor 16-Bit Programmer's Reference Manual* and *Nios Embedded Processor 32-Bit Programmer's Reference Manual* contains a detailed description of the exception-handling process.

wvalid_wr Assignment

Control-register number 2 (`%ctl2`) is the WVALID register. Software can always read this register to determine the limits of travel on current window pointer (CWP). WVALID determines when underflow and overflow exceptions are generated. A detailed description of the WVALID register appears in the *Nios Embedded Processor 32-Bit Programmer's Reference Manual* and *Nios Embedded Processor 16-Bit Programmer's Reference Manual*. Regardless of the setting of `wvalid_wr`, the WVALID register is initialized at reset so that the entire windowed register file is available to software (`LO_LIMIT = 1` and `HI_LIMIT = <number-of-registers> - 2`). The vast majority of compiled programs never need to alter the initially-set WVALID value.

The default CWP-manager library included in the automatically-generated SDK never writes the WVALID register. When the PTF-parameter `wvalid_wr = 0`, the WVALID register is not writable. A WRCTL instruction with `%ctrl2` as the destination has no effect. By implementing WVALID read-only, the total logic utilization of the CPU is reduced by approximately 15 LEs. Also, WVALID is protected from accidental modification by errant software.

When `wvalid_wr = 1`, WVALID is implemented as a readable and writable register. Software can modify WVALID by performing a WRCTL instruction with `%ctrl2` as the destination. Some real-time operating system (RTOS) software may modify WVALID, for example, partitioning the register file into kernel and user segments. Such RTOS software will require a writable WVALID register (`wvalid_wr = 1`) in order to run correctly.



Check the manufacturer's documentation for your RTOS software to determine its CPU hardware requirements.

datamem_slave Assignment

The Nios CPU is generated along with a custom SDK including libraries, example programs, and utilities. C programs are, by default, compiled with their stack, heap (malloc data) and writable global variables within the system's designated data-memory device. This PTF assignment is used to designate the memory device where, by default, compiled programs' writable data will be located. The generated SDK is aware of the vector table, and will locate the stack and heap to avoid conflicts.

debugcomm_slave Assignment

The Nios CPU is generated along with a custom SDK including libraries, example programs, and utilities. One such utility is a resident GDB stub which communicates with host debugging software over a designated serial device (Avalon UART). The PTF assignment `debugcomm_slave` designates which Avalon UART is used by the debug stub. It can be the same UART as the `maincomm_slave`, but it is generally easier to debug systems with separate `debugcomm_slave` and `maincomm_slave` UARTs.



The designated device must be an Avalon UART.

germs_monitor_id Assignment

The Nios CPU is generated along with a custom SDK including libraries, example programs, and utilities. One of the several programs generated as part of the Nios SDK is a customized GERMS monitor (boot-monitor program). The GERMS monitor prints a message at start up. The contents of this message are taken from the CPU's `germs_monitor_id` assignment. It is often useful to set different id messages to display different hardware revision levels of your system.

maincomm_slave Assignment

The Nios CPU is generated along with a custom SDK including libraries, example programs, and utilities. Some libraries and utilities implicitly make use of a standard output or console device, for example, the `printf` library. The SDK uses a designated Avalon slave peripheral as the default I/O device for many text input-output functions. The PTF assignment `maincomm_slave` designates which slave peripheral is used by the generated SDK for default text I/O.



There are two designated slave peripherals: OCI debug module and UART. The designated slave peripheral must have the PTF assignment `Is Printable Device = 1`.

mainmem_slave Assignment

The Nios CPU is generated along with a custom SDK including libraries, example programs, and utilities. C programs are, by default, compiled to reside at a base address within the system's designated main-memory device. This PTF assignment is used to designate the memory device where, by default, compiled programs will be located. The generated SDK is aware of the vector table, and will set the base address of compiled programs to avoid the vector-table if, for example, the vector table is mapped at the beginning of the designated main memory device.

include_oci Assignment

The Nios on-chip instrumentation (OCI) Debug Module is enabled with the `include_oci` parameter. The Nios OCI debug module must be enabled and configured using the Nios CPU configuration wizard. Several other PTF assignments are used to specify parameters for the OCI debug module. Only the configurations specified by SOPC Builder are supported. These PTF assignments are: `include_oci`, `oci_offchip_trace`, `oci_onchip_trace`, `oci_num_xbrk`, `oci_type`, `oci_num_dbrk`, `oci_dbrk_trace`, `oci_dbrk_pairs`, and `oci_debugreq_signals`.

When `include_oci = 1`, the Nios CPU includes on-chip logic and memory resources required to implement JTAG-based software debug and optional trace features. In addition, the CPU is generated with an Avalon slave port named `oci_core`. Using the SOPC Builder GUI, designers can assign any valid base address to the `oci_core` slave port. Only debug tools access the `oci_core` memory space; software designers should not access `oci_memory`.

smart_recompile Assignment

This assignment, when set (1), enables a make-like functionality of the Nios generator program. The HDL representing the Nios CPU (and associated memory contents and simulation files) will be generated only if the parameters that influence the CPU have changed.

When the option is set to the default setting, clear (0), the CPU is always regenerated, even if doing so results in output files identical to the previous generation.



101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
[Applications Hotline:](#)
(800) 800-EPLD
[Literature Services:](#)
lit_req@altera.com

Copyright © 2003 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



I.S. EN ISO 9001