Nios II MPU Usage

© March 2010 AN-540-1.0

Introduction

This application note covers the basic features of the Nios® II processor's optional memory protection unit (MPU), describing how to use it without the support of an operating system (OS). When the Nios II MPU is enabled and properly configured, it monitors all processor data and instruction accesses and triggers exceptions when illegal accesses are attempted.

This application note includes two design examples, with notes about how the examples work. These examples walk you through making use of the Nios II processor's MPU in an environment based on the Altera® hardware abstraction layer (HAL), without an OS. One of the examples uses the MPU to detect the following three issues commonly seen when debugging embedded systems:

- Stack overflow
- Null pointer
- Wild pointer

Do not confuse the MPU with the Nios II memory management unit (MMU). The MPU does not provide memory mapping or management.

Requirements

To use this application note effectively, you need to be familiar with the following topics:

- The basic purpose and architecture of the Nios II MPU

  For a detailed description of the Nios II MPU, refer to "Memory Protection Unit" in the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

- Creating and configuring Nios II systems with SOPC Builder.

  For information about creating and configuring Nios II systems, refer to the *Nios II Hardware Development Tutorial* and to *Volume 4: SOPC Builder* in the *Quartus II Handbook*.

To work with this application note's design examples and software examples, you need the following items:

- The Nios II Embedded Evaluation Kit (NEEK), Cyclone® III Edition

  The design examples use only on-chip hardware resources. Therefore, it is easy to port the designs to a different hardware platform if necessary.

- Quartus® II software version 9.1 or higher.

- Nios II Embedded Design Suite (EDS) version 9.1 or higher.
- The design example archive file, **an540_91.zip**. This file is available next to the link to this document on the Literature: Nios II Processor page of the Altera website.

  Unzip **an540_91.zip** to a working directory on your computer. We refer to this directory throughout this application note as <*design examples*>. Be sure to preserve the directory structure of the extracted software archive. Extraction creates a directory structure tree under <*design examples*> with the following subdirectories:

  - **MPU_Design_limit/software_examples/app/mpu_basic**
  - **MPU_Design_limit/software_examples/app/mpu_exc_detection**
  - **MPU_Design_limit/software_examples/bsp/mpu_example_bsp**
  - **MPU_Design_msk/software_examples/app/mpu_basic**
  - **MPU_Design_msk/software_examples/app/mpu_exc_detection**
  - **MPU_Design_msk/software_examples/bsp/mpu_example_bsp**

  ☞ The working directory name you choose must not contain any spaces.

☞ After extracting **an540_91.zip**, refer to <*design examples*>**/ReadMe.txt** for a list of any required software patches or other updated information. If a patch is required, install it according to the instructions in **ReadMe.txt**.

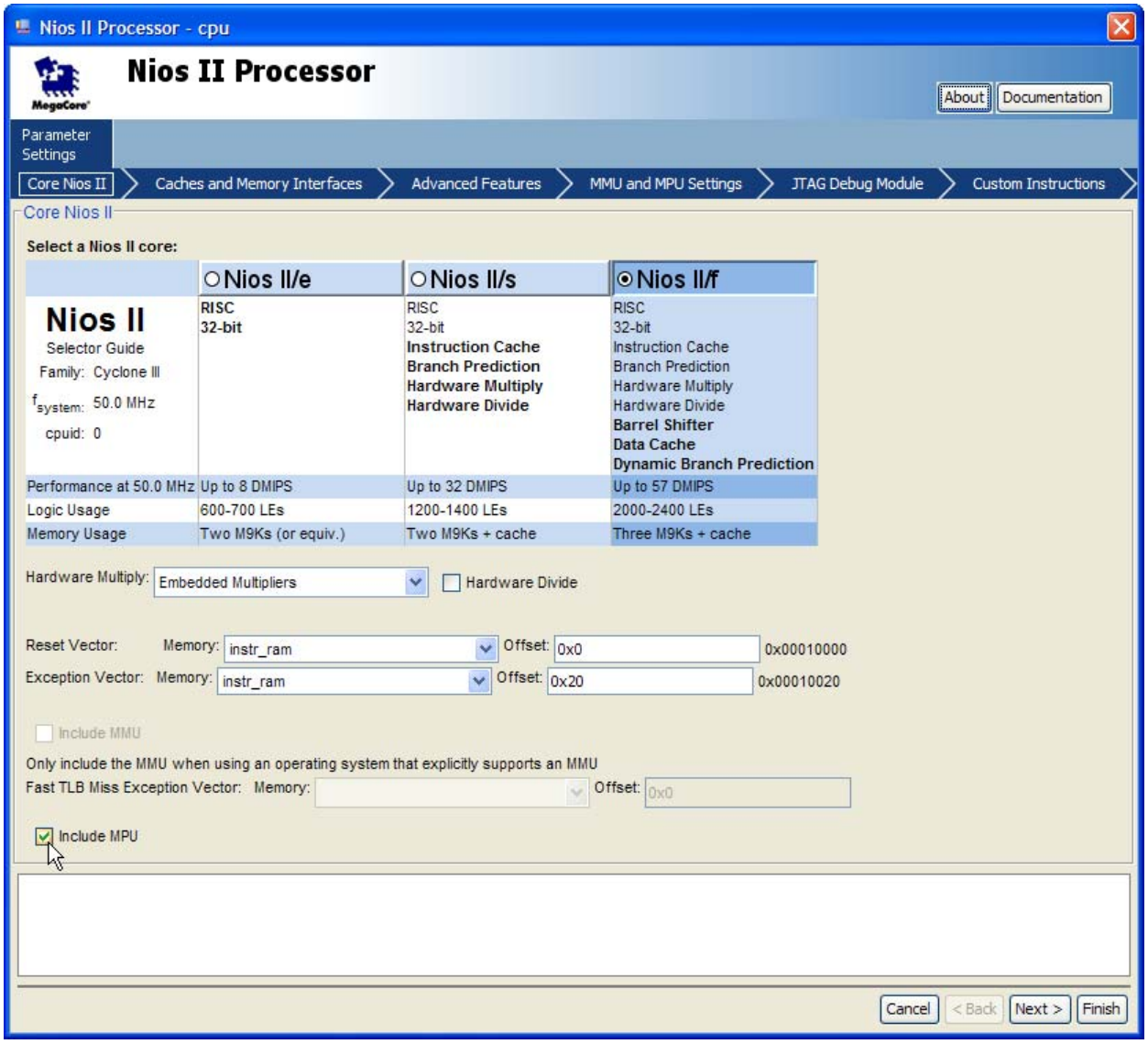
General Usage

This section describes the process of configuring the Nios II MPU hardware and writing software to support it.

Adding the MPU Hardware

To add an MPU to your system, you must use a Nios II/f core. In SOPC Builder, enable the MPU by turning on **Include MPU** in the **Core Nios II** tab of the Nios II MegaWizard™ interface, as shown in Figure 1.

Figure 1. Enabling the MPU in the Nios II/f Processor Core



Use the **MMU and MPU Settings** tab, as shown in Figure 2, to configure the MPU.

Figure 2. MMU and MPU Settings Tab

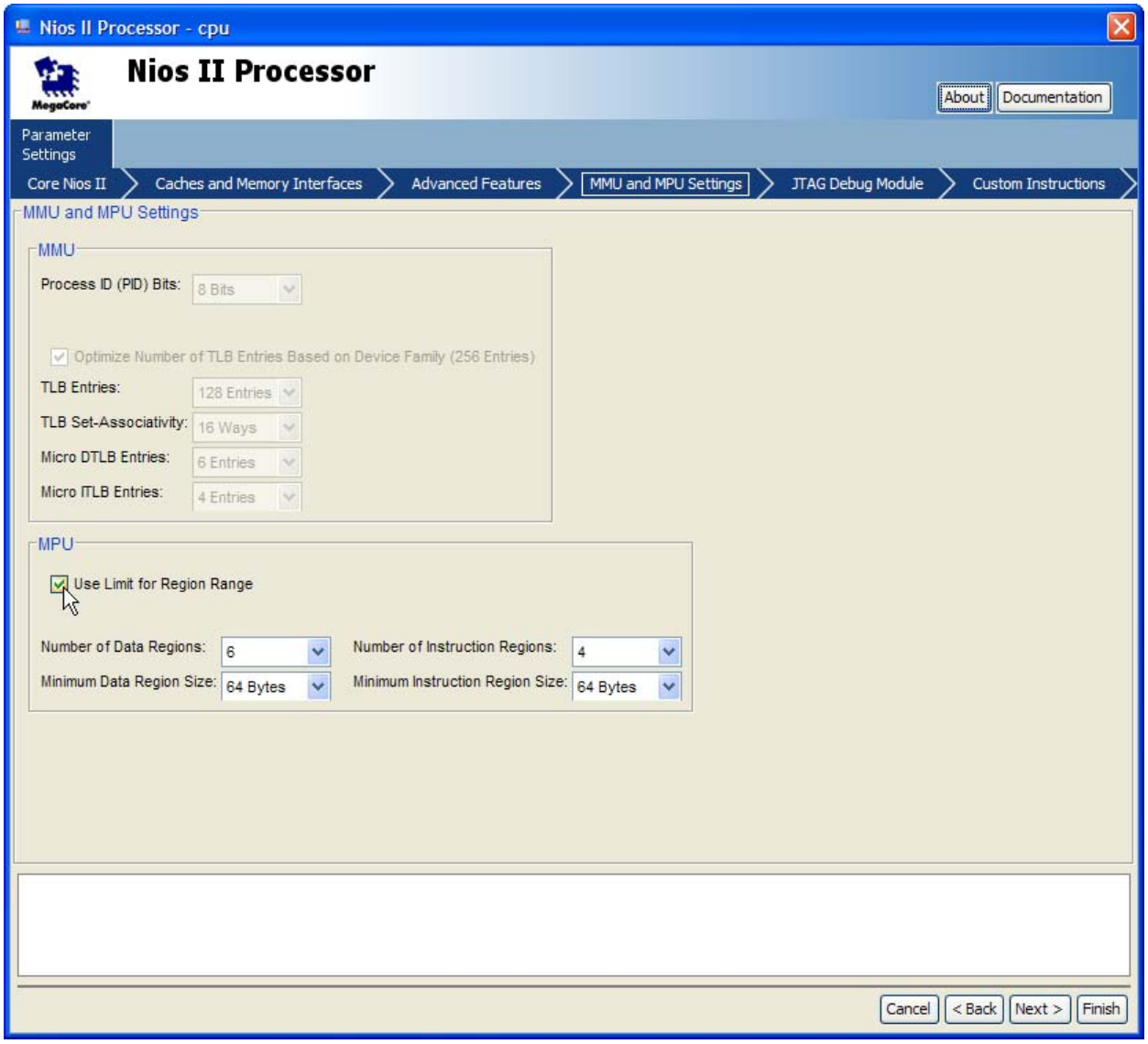


Table 1 summarizes the MPU options.

Table 1. MPU Configuration Options

Option	Allowed Values	Default Value
Use Limit for Region Range	Off or On	Off
Number of Data Regions	2—32	8
Number of Instruction Regions	2—32	8
Minimum Data Region Size	64 bytes—1 MB	4 KB
Minimum Instruction Region Size	64 bytes—1 MB	4 KB

You can configure the MPU to define the size of its memory regions in either of the following ways:

- Define region size by specifying an address mask
- Define region size by specifying the end address

By default, the MPU defines region sizes with an address mask. To define region sizes with an end address, turn on **Use Limit for Region Range**. For detailed information about the two methods of specifying region size, refer to "MPU Register Details" on page 6.

The minimum region size is crucial to understanding MPU run-time configuration. The minimum region size, <*min_region*>, specifies the granularity of the MPU memory map. The size of any particular memory region must be an integer multiple of <*min_region*>.

Most of the MPU parameters controlled by software are based on the minimum region size. You can specify separate values of <*min_region*> for data and instruction regions.

For simplicity, this application note's design examples have <*min_region*> = 64 for both data and instruction regions.

Writing Software for the MPU

This section describes the process of writing software to configure and manage the Nios II MPU.

MPU Programming Guidelines

Software is responsible for enabling and configuring the MPU as well as maintaining MPU region information. In a single-threaded operating environment (such as the Altera HAL), use a global data structure to store the MPU region information.

The Nios II MPU must be disabled before software attempts to configure it.

Software normally initializes the MPU after reset. If it is necessary to change MPU regions or region permissions after reset, software also reinitializes the MPU.

Every region supported by the MPU must be either configured or disabled before allowing application code to execute. Leaving a region enabled and unconfigured results in undefined behavior. For details about how to disable an MPU region, refer to "Defining Regions with mpubase and mpuacc" on page 10.

Depending on the complexity of your software, you might need to define several MPU configurations, each with a different set of regions or region permissions. This technique is typically used by an operating system. For details, refer to "Operating Systems and the MPU".

Operating Systems and the MPU

Even if you are not using an operating system, it is helpful to understand the techniques that an OS uses to manage an MPU.

When an operating system uses an MPU, it typically defines two or more MPU configurations. One configuration defines the permissions that the MPU applies to operating system or kernel level accesses. One or more configurations define the permissions available to user or application processes. The OS might also define additional configurations for non-user purposes. For example, there might be a special factory task that can modify system-critical information like product serial numbers or media access control (MAC) addresses in flash or other nonvolatile memory. Such a task is likely to need a special set of memory and device permissions.

The operating system disables the MPU, reconfigures it, and then re-enables it whenever the processor needs to run in a different MPU configuration. For example, the OS might need to change MPU configurations upon the following types of events:

- Exception
- Return from exception
- Operating system call
- Return from operating system call

The exact circumstances under which MPU reconfiguration is required depends on the OS implementation and settings.

MPU Register Details

This section describes the register maps, the meanings of the register fields, and how the register fields are used.

When you initialize the MPU you use two registers: `mpubase` and `mpuacc`.

Register mpubase Usage

Table 2 shows the layout of the `mpubase` register.

Table 2. mpubase Control Register Fields

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

Notes to Table 2:

(1) This field size is variable. Unused upper bits and unused lower bits must be written as zero.

(2) This field size is variable. Unused upper bits must be written as zero.

Table 3 gives details of the fields defined in the `mpubase` register.

Table 3. mpubase Control Register Field Descriptions

Field	Description	Access	Reset
BASE	BASE represents the base memory address of the region identified by the INDEX and D fields.	Read/Write	0
INDEX	INDEX is the region index number.	Read/Write	0
D	D is the region access bit. When D = 1, INDEX refers to a data region. When D = 0, INDEX refers to an instruction region.	Read/Write	0

You specify an MPU region by writing a value representing the region's base address to the BASE field, a unique index to the INDEX field, and the region type (data or instruction) to field D.

The BASE field represents the region's base address, in the form described by Equation 1. The BASE field can only represent addresses aligned to an integer multiple of <*min_region*>. For example, if the minimum region size is 16 kilobytes (KB), regions can be located at addresses such as 0x0, 0x4000, 0x8000, ...

Equation 1. Base Address Computation

$$BASE = \langle base\ address \rangle / \langle min\_region \rangle$$

For example, if the region starts at 0x1000 and the minimum region size is 64 bytes, set the BASE field to 0x40, which is 0x1000/64.

The INDEX field provides a unique identifier for the region. INDEX also specifies the priority of the region. The lower the index value, the higher the region's priority.

Use the D field to specify the region type: data or instruction.

Register mpuacc Usage

mpuacc has two possible layouts, depending on the SOPC Builder generation-time option **Use limit for region range**, as described in "Adding the MPU Hardware" on page 2. This option controls whether the mpuacc register contains a MASK or LIMIT field. Table 4 shows the layout of the mpuacc register with the MASK field.

Table 4. mpuacc Control Register Fields for MASK Variation

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	MASK *(1)*																									C	PERM			RD	WR

Note to Table 4:

(1) This field size is variable. Unused upper bits and unused lower bits must be written as zero.

Table 5 shows the layout of the mpuacc register with the LIMIT field.

Table 5. mpuacc Control Register Fields for LIMIT Variation

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
LIMIT *(1)*																										C	PERM			RD	WR

Note to Table 5:

(1) This field size is variable. Unused upper bits and unused lower bits must be written as zero.

Table 6 provides details of the fields defined in the mpuacc register.

Table 6. mpuacc Control Register Field Descriptions

Field	Description	Access	Reset
MASK *(1)*	MASK specifies the size of the region.	Read/Write	0
LIMIT *(1)*	LIMIT specifies the upper address limit of the region.	Read/Write	0
C	C is the data cacheable flag. C only applies to MPU data regions and determines the default cacheability of a data region. When C = 0, the data region is uncacheable. When C = 1, the data region is cacheable.	Read/Write	0
PERM	PERM specifies the access permissions for the region.	Read/Write	0
RD	RD is the read region flag. When RD = 1, wrctl instructions to the mpuacc register perform a read operation.	Write	0
WE	WR is the write region flag. When WR = 1, wrctl instructions to the mpuacc register perform a write operation.	Write	0

Note to Table 6:

(1) The MASK and LIMIT fields are mutually exclusive. Refer to Table 4 and Table 5.

If the mpuacc register is configured with the MASK field, the MASK field represents the size of your region. The value of MASK is defined in Equation 2.

Equation 2. Computing Region Mask

$$MASK = 0x1FFFFFF << \log_2 (<region\_size> >> 6)$$

Table 7 lists every possible MASK value for an MPU configured with a 64-byte minimum region size.

Table 7. MASK Encodings for 64-byte Minimum Region (Part 1 of 2)

MASK Encoding	Region Size
0x1FFFFFF	64 bytes
0x1FFFFFE	128 bytes
0x1FFFFFC	256 bytes
0x1FFFFF8	512 bytes
0x1FFFFF0	1 KByte
0x1FFFFE0	2 KB
0x1FFFFC0	4 KB
0x1FFFF80	8 KB
0x1FFFF00	16 KB
0x1FFFE00	32 KB
0x1FFFC00	64 KB
0x1FFF800	128 KB
0x1FFF000	256 KB
0x1FFE000	512 KB
0x1FFC000	1 MB
0x1FF8000	2 MB

Table 7. MASK Encodings for 64-byte Minimum Region (Part 2 of 2)

MASK Encoding	Region Size
0x1FF0000	4 MB
0x1FE0000	8 MB
0x1FC0000	16 MB
0x1F80000	32 MB
0x1F00000	64 MB
0x1E00000	128 MB
0x1C00000	256 MB
0x1800000	512 MB
0x1000000	1 GB
0x0000000	2 GB

If the `mpuacc` register is configured with the `LIMIT` field, `LIMIT` represents the address immediately following the upper end of your region. For example, suppose the MPU's minimum region size is 64 bytes, and you need to set up the following region:

- The region starts at 0x1000
- The region ends at 0x1FFF

To set up the desired region, configure `mpubase.BASE` and `mpuacc.LIMIT` as shown in the following list:

- Set `mpubase.BASE` to 0x40, which is 0x1000/64
- Set `mpuacc.LIMIT` to 0x80, which is 0x2000/64

Use the `C` field to specify whether a data region is to be cached. Usually, you set `C` for memory regions and clear it for regions representing registers or general-purpose memory-mapped I/O.

The `PERM` field defines the permissions for the region, as shown in Table 8 and Table 9.

Table 8. Instruction Region Permission Encodings

PERM Encoding *(1)*	Supervisor Permissions	User Permissions
000	None	None
001	Execute	None
010	Execute	Execute

Note to Table 8:

(1) PERM values represented in binary

Table 9. Data Region Permission Encodings

PERM Encoding *(1)*	Supervisor Permissions	User Permissions
000	None	None
001	Read	None
010	Read	Read
100	Read/Write	None
101	Read/Write	Read
110	Read/Write	Read/Write

Note to Table 9:

(1) PERM values represented in binary

Defining Regions with mpubase and mpuacc

The `mpubase` register works in conjunction with the `mpuacc` register to set and retrieve MPU region information. Use the `RD` and `WR` fields of `mpuacc` to instruct the MPU to perform an MPU region read or write, as shown in the following list:

- Set `mpuacc.RD` = 1 to perform an MPU region read operation.
- Set `mpuacc.WR` = 1 to perform an MPU region write operation.

Simultaneously setting both the `RD` and `WR` fields to 1 results in undefined behavior.

An MPU region must be disabled if it is not in use. To disable a region, software sets up the following conditions:

- `mpubase.BASE` is any nonzero value.
- If the MPU is configured to define region size by mask, `mpuacc.MASK` represents 0x80000000, which is 2^{31} (the size of the Nios II address space). For example, if the minimum region size is 64, or 0x40 bytes, `mpuacc.MASK` is 0x80000000 / 0x40, or 0x20000000.
- If the MPU is configured to define region size by limit, `mpuacc.LIMIT` = 0.

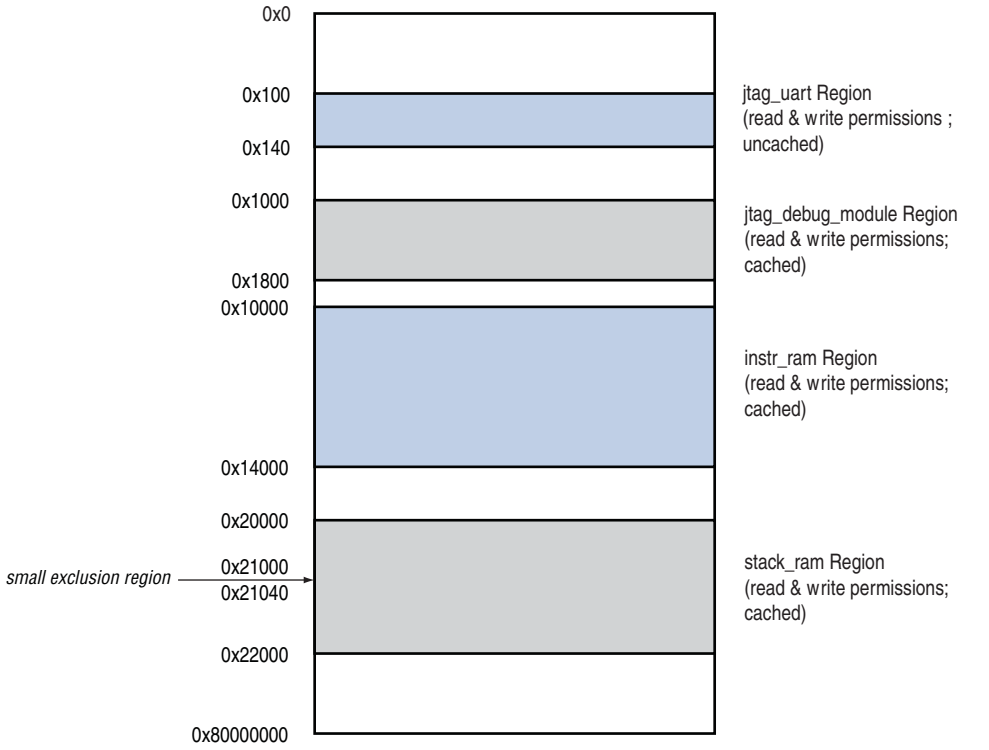
Region Layout Considerations

This section describes how to select MPU region locations and sizes to make the most effective use of the MPU. For information about the mechanics of setting up MPU regions, refer to "MPU Register Details" on page 6.

Each region size must be an integer power of two. You must ensure that each region is aligned to an address that is an integer multiple of its size.

Figure 3 and Figure 4 illustrate the regions configured by the software examples accompanying this application note. Refer to the example code and comments for details about how and why these regions are configured as they are.

Figure 3. MPU Data Region Example (Addresses not to scale) *(1)*

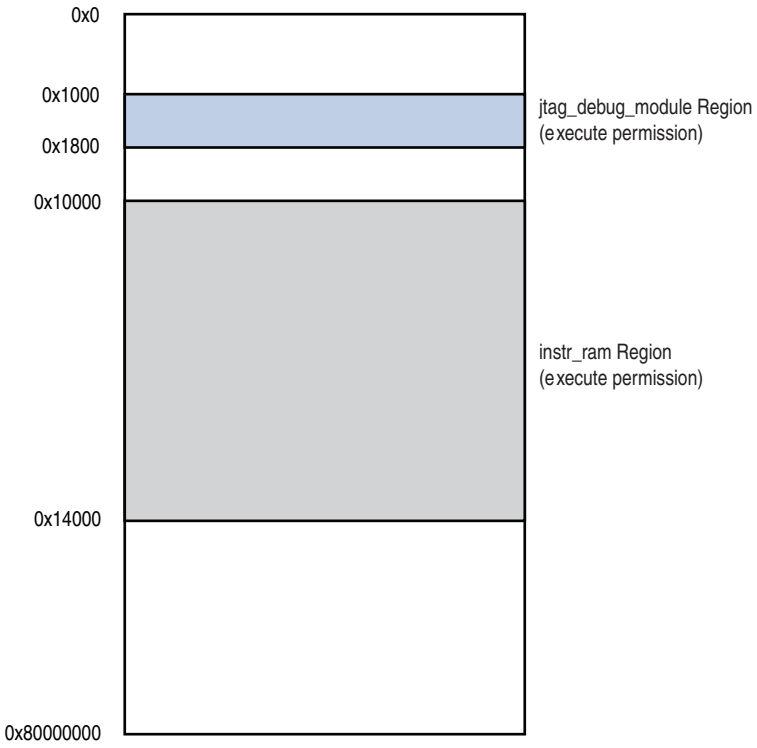


Note to Figure 3:

(1) A low-priority exclusion region spans the entire 2 GB address space from 0x0 to 0x80000000.

Regions can overlap. For example, you can place a higher-priority region inside a lower-priority region. `region[3]` in mpu_utils.c illustrates this technique, creating a small exclusion region from 0x21000 to 0x21040, as shown in Figure 3. Any access to addresses in the 0x21000 to 0x21040 range is controlled by the exclusion region rather than the stack_ram region (`region[4]`), because the exclusion region has the higher priority.

Figure 4. MPU Instruction Region Example (Addresses not to scale) *(1)*



Note to Figure 4:

(1) A low-priority exclusion region spans the entire 2 GB address space from 0x0 to 0x80000000.

Flow Summary

In a Nios II system with an MPU, whenever MPU initialization or reinitialization is required, the software is responsible for the following tasks:

1. Ensure that the MPU is disabled.

   ☞ At system reset, the MPU is disabled by default. At other times, software must disable the MPU before reconfiguring regions.

2. Initialize and configure the MPU with region information.
3. Enable the MPU prior to executing task-specific or single-threaded application code.

Design Examples

The design examples accompanying this application note illustrate the use of the Nios II MPU in a single-threaded environment, such as the Altera HAL.

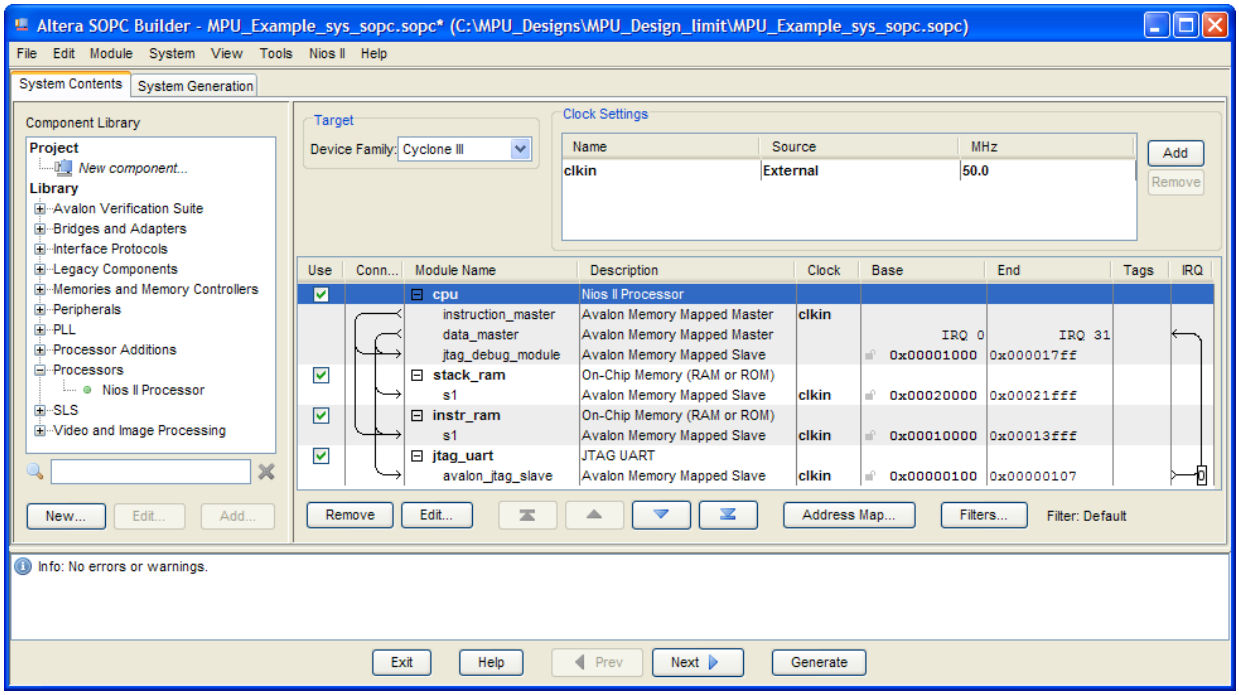
Example Hardware

The simple hardware designs, emphasizing MPU usage, are easily portable to other hardware platforms. There are two design examples, both targeting the NEEK. In one, the MPU specifies region sizes by mask, and in the other the MPU specifies region sizes by limit. Aside from this detail of MPU instantiation, the two designs are identical.

The address map is designed to make MPU configuration very straightforward. For instance, the **instr_ram** and **stack_ram** memories reside on valid region boundaries, and the JTAG UART base address is unique and aligned to a valid region boundary, as illustrated in Figure 3.

Figure 5 illustrates one of the design examples as it appears in SOPC Builder. The hardware addresses fall on valid MPU region boundaries. While this constraint is not required, it is more convenient for the software engineer.

Figure 5. MPU Example Hardware System



Software

The design files accompanying this application note include the following example software projects:

- **mpu_basic**—Configures the MPU with several data and instruction regions, and prints a simple message.
- **mpu_exc_detection**—Configures the MPU with the same data instruction regions as in **mpu_basic**, and sets up an exception handler to detect the following conditions:
  - Null pointer
  - Wild pointer
  - Stack overflow

The software examples in each subdirectory are identical. The code is written to detect the whether the MPU is configured for mask or limit region sizes, and to behave appropriately.

The **mpu_exc_detection** example detects stack overflow by creating a small high-priority exclusion data region in the middle of a larger data region where both the stack and the heap reside. Whenever the stack grows downwards or the heap grows upwards into this exclusion region, the MPU triggers an exception and the software detects it.

The **mpu_exc_detection** example detects null pointer usage by making sure that no regions include offset 0x0. The example system is designed such that no components (memory or otherwise) are located at this offset. If software attempts to access address 0x0, the MPU triggers an exception, allowing the software to recover. If you ensure that memories are preinitialized to zero, null pointer detection helps protects against uninitialized data access.

The **mpu_exc_detection** example detects wild pointer usage by creating very large low-priority exclusion regions covering the majority of the memory map. In this way, if the Nios II processor attempts to access an address outside of valid memory and peripheral I/O address space, the MPU triggers an exception and software can detect it.

Both of these software examples use the MPU utility functions and macros in **mpu_utils.c** and **mpu_utils.h**. In both examples, initialization and reinitialization are handled by two functions: one for data regions, and one for instruction regions. In most real-world systems, a single function is sufficient to handle initialization and reinitialization for both types of regions.

MPU Utilities

You can find helpful MPU utility functions and macros in the **mpu_utils.c** and **mpu_utils.h** files in each software example. The following functions are the most important for you to understand:

- `nios2_mpu_data_init()`—A system-specific function. In your own code, write an equivalent function to specify the MPU data regions in your design.
- `nios2_mpu_inst_init()`—A system-specific function. In your own code, write an equivalent function to specify the MPU instruction regions in your design.

- `nios2_mpu_load_region()`—Configures an MPU region with specific parameters.
- `nios2_mpu_enable()`—Enables the entire MPU.
- `nios2_mpu_disable()`—Disables the entire MPU.

Each utility function makes use of the `Nios2MPURegion` data structure shown in Example 1.

Example 1. Nios2MPURegion Data Structure

```
typedef struct
{
    unsigned int base;
    unsigned int index;
    unsigned int mask;
    unsigned int c;
    unsigned int perm;
} Nios2MPURegion;
```

Example 2 shows `nios2_mpu_inst_init()` for the **mpu_basic** software example. The constants `NIOS2_MPU_NUM_INST_REGIONS` and `NIOS2_MPU_REGION_USES_LIMIT` are defined in **system.h**.

In Example 2, `region[0]` grants execution access to the **instr_ram** memory in both user and supervisor modes, as shown in Figure 4 on page 12. `region[1]` grants execution access to the break and trace memory (starting at 0x1000) in both modes. The other two MPU instruction regions grant no execution permissions to the entire Nios II address space. Because their priorities, 2 and 3, are lower than the first two regions, the code stored in the **instr_ram** runs, and the break and trace features work correctly. However, if code attempts to execute outside those regions, the MPU triggers an exception.

The final statement in `nios2_mpu_inst_init()` calls `nios2_mpu_load_region()` to configure the region with the information contained in the structure.

Example 2. nios2_mpu_inst_init() in the mpu_basic Software Example

```
void nios2_mpu_inst_init()
{
    unsigned int mask;
    Nios2MPURegion region[NIOS2_MPU_NUM_INST_REGIONS];

    //Main instruction region.
    region[0].index = 0;
    region[0].base = 0x400; // Byte Address 0x10000
#ifdef NIOS2_MPU_REGION_USES_LIMIT
    region[0].mask = 0x500; // Byte Address 0x14000
#else
    region[0].mask = 0x1ffff00;
#endif
    region[0].c = 1;
    region[0].perm = MPU_INST_PERM_SUPER_EXEC_USER_EXEC;

    //Instruction region for break address.
    region[1].index = 1;
    region[1].base = 0x40; // Byte Address 0x1000
#ifdef NIOS2_MPU_REGION_USES_LIMIT
    region[1].mask = 0x60; // Byte Address 0x1800
#else
    region[1].mask = 0x1ffffe0;
#endif
    region[1].c = 1;
    region[1].perm = MPU_INST_PERM_SUPER_EXEC_USER_EXEC;

    //Rest of the regions are maximally sized and permissive.
#ifdef NIOS2_MPU_REGION_USES_LIMIT
    mask = 0x2000000;
#else
    mask = 0x0;
#endif
    unsigned int num_of_region = NIOS2_MPU_NUM_INST_REGIONS;
    unsigned int index;
    for (index = 2; index < num_of_region; index++){
        region[index].base = 0x0;
        region[index].index = index;
        region[index].mask = mask;
        region[index].c = 0;
        region[index].perm = MPU_INST_PERM_SUPER_NONE_USER_NONE;
    }

    nios2_mpu_load_region(region, num_of_region, 0);
}
```

Example 3 shows the function prototype for `nios2_mpu_load_region()`.

Example 3. nios2_mpu_load_region()

```
void nios2_mpu_load_region (
    Nios2MPURegion region[],
    unsigned int num_of_region,
    unsigned int d);
```

The following list shows the arguments to `nios2_mpu_load_region()`:

- `Nios2MPURegion`—An array of data structures, each representing an MPU region
- `num_of_region`—The number of regions
- `d`—The region type (instruction or data)

`nios2_mpu_load_region()` configures the MPU according to the arguments passed by the calling function.

The MPU is disabled by default at system restart. After the MPU is configured, the example uses `nios2_mpu_enable()` and `nios2_mpu_disable()` to enable and disable the MPU. Whenever you reconfigure the MPU, you must first disable it, and re-enable it after configuring.

The software examples accompanying this application note are commented to help you understand how each example works. Most of the complexity of managing the MPU and its regions is embodied in the MPU utility functions and macros in **mpu_utils.c** and **mpu_utils.h**, allowing you to focus on the top-level software flow.

Building the Software

To create and build a software example, execute the following steps:

1. Identify the directory containing the software example that you want to run, based on the hardware example that you want to use. For example, to run the **mpu_basic** software example on the **MPU_Design_limit** hardware design example, the directory is *<design examples>*/**MPU_Design_limit/software_examples/app/mpu_basic.**
2. Use one of the following methods to open the Nios II Command Shell:
   - In the Windows operating system, on the Start menu, point to **Programs** > **Altera** > **Nios II EDS** *<version>*, and click **Nios II** *<version>* **Command Shell**.
   - In the Linux operating system, in a command shell, execute the following commands:

     ```
     cd $SOPC_KIT_NIOS2↵
     ./sdk_shell↵
     ```
3. Change directories to the software example directory identified in Step 1.
4. Type the following command:

   ```
   ./create-this-app↵
   ```
5. After the projects are generated and built, configure your board with the hardware image and run the software with the following commands:

   ```
   nios2-configure-sof -C ../../../↵
   nios2-download -g <example>.elf && nios2-terminal↵
   ```

Each software example displays information on the screen. The output from the **mpu_basic** example resembles Example 4.

Example 4. mpu_basic Console Output

```
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Pausing target processor: OK
Initializing CPU cache (if present)
OK
Downloaded 3KB in 0.0s
Verified OK
Starting processor at address 0x00010020
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

Hello from a simple MPU-Enabled Nios II System!.
val1 = 0xfeedface, val2 = 0xfeedface, val3 = 0x@.
```

The output from the **mpu_exc_detection** example resembles Example 5.

Example 5. mpu_exc_detection Console Output

```
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Pausing target processor: OK
Initializing CPU cache (if present)
OK
Downloaded 5KB in 0.0s
Verified OK
Starting processor at address 0x00010110
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

Hello from a simple MPU-Enabled Nios II System!.
        Starting some exceptions tests.
=====
MPU NULL data pointer test.
MPU NULL data pointer test passed!
MPU wild pointer test.
MPU wild pointer test passed!
MPU stack overflow test.
MPU stack overflow test passed!
=====
        Exception Tests ended.
Now exiting program.
```

For further details, refer to the source code and the <*design examples*>**/ReadMe.txt** file accompanying the examples.

If the software example appears to hang, verify that you have configured your board with the correct **.sof**.

Conclusion

After you have studied the code and understand the design examples described in this application note, you have the skills to use the Nios II MPU successfully in your HAL-based design. These examples illustrate the basics of how to use `mpubase` and `mpuacc` to configure your MPU prior to enabling it.

Referenced Documents

This application note refers to the following documents:

- *Nios II Hardware Development Tutorial*
- *Programming Model* chapter of the *Nios II Processor Reference Handbook*
- *Volume 4: SOPC Builder* in the *Quartus II Handbook*

Document Revision History

Table 10 shows the revision history for this application note.

Table 10. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2010 v1.0	Initial release.	—