Xilinx Standalone Library Documentation

Standalone v6.5

UG647 (2017.4) December 20, 2017

XILINX
ALL PROGRAMMABLE™

Table of Contents

Chapter 3: Cortex R5 Processor API

Chapter 4: ARM Processor Common API

Chapter 5: Cortex A9 Processor API

Chapter 6: Cortex A53 32-bit Processor API

XILINX
ALL PROGRAMMABLE™

Chapter 1

Xilinx Hardware Abstraction Layer API

Overview

This section describes the Xilinx® Hardware Abstraction Layer API, These APIs are applicable for all processors supported by Xilinx.

Modules

- Assert APIs
- IO interfacing APIs
- Definitions for available xilinx platforms
- Data types for Xilinx Software IP Cores
- Customized APIs for memory operations
- Xilinx software status codes
- Test utilities for memory and caches

Assert APIs

Overview

The `xil_assert.h` file contains the assert related functions.

Macros

- #define Xil_AssertVoid(Expression)
- #define Xil_AssertNonvoid(Expression)
- #define Xil_AssertVoidAlways()
- #define Xil_AssertNonvoidAlways()

Typedefs

- typedef void(* Xil_AssertCallback) (const char8 *File, s32 Line)

Functions

- void Xil_Assert (const char8 *File, s32 Line)
- void XNullHandler (void *NullParameter)
- void Xil_AssertSetCallback (Xil_AssertCallback Routine)

Variables

- u32 Xil_AssertStatus
- s32 Xil_AssertWait

Macro Definition Documentation

#define Xil_AssertVoid(*Expression*)

This assert macro is to be used for void functions. This in conjunction with the Xil_AssertWait boolean can be used to accomodate tests so that asserts which fail allow execution to continue.

Parameters

Expression	expression to be evaluated. If it evaluates to false, the assert occurs.

Returns

Returns void unless the Xil_AssertWait variable is true, in which case no return is made and an infinite loop is entered.

#define Xil_AssertNonvoid(*Expression*)

This assert macro is to be used for functions that do return a value. This in conjunction with the Xil_AssertWait boolean can be used to accomodate tests so that asserts which fail allow execution to continue.

Parameters

Expression	expression to be evaluated. If it evaluates to false, the assert occurs.

Returns

Returns 0 unless the Xil_AssertWait variable is true, in which case no return is made and an infinite loop is entered.

#define Xil_AssertVoidAlways( )

Always assert. This assert macro is to be used for void functions. Use for instances where an assert should always occur.

Returns

Returns void unless the Xil_AssertWait variable is true, in which case no return is made and an infinite loop is entered.

#define Xil_AssertNonvoidAlways( )

Always assert. This assert macro is to be used for functions that do return a value. Use for instances where an assert should always occur.

Returns

Returns void unless the Xil_AssertWait variable is true, in which case no return is made and an infinite loop is entered.

Typedef Documentation

typedef void(* Xil_AssertCallback) (const char8 *File, s32 Line)

This data type defines a callback to be invoked when an assert occurs. The callback is invoked only when asserts are enabled

Function Documentation

void Xil_Assert (const char8 * *File*, s32 *Line*)

Implement assert. Currently, it calls a user-defined callback function if one has been set. Then, it potentially enters an infinite loop depending on the value of the Xil_AssertWait variable.

Parameters

file	filename of the source
line	linenumber within File

Returns

None.

Note

None.

void XNullHandler (void * *NullParameter*)

Null handler function. This follows the XInterruptHandler signature for interrupt handlers. It can be used to assign a null handler (a stub) to an interrupt controller vector table.

Parameters

NullParameter	arbitrary void pointer and not used.

Returns

None.

Note

None.

void Xil_AssertSetCallback (Xil_AssertCallback *Routine*)

Set up a callback function to be invoked when an assert occurs. If a callback is already installed, then it will be replaced.

Parameters

routine	callback to be invoked when an assert is taken

Returns

None.

Note

This function has no effect if NDEBUG is set

Variable Documentation

u32 Xil_AssertStatus

This variable allows testing to be done easier with asserts. An assert sets this variable such that a driver can evaluate this variable to determine if an assert occurred.

s32 Xil_AssertWait

This variable allows the assert functionality to be changed for testing such that it does not wait infinitely. Use the debugger to disable the waiting during testing of asserts.

IO interfacing APIs

Overview

The `xil_io.h` file contains the interface for the general IO component, which encapsulates the Input/Output functions for processors that do not require any special I/O handling.

Functions

- u16 Xil_EndianSwap16 (u16 Data)
- u32 Xil_EndianSwap32 (u32 Data)
- static INLINE u8 Xil_In8 (UINTPTR Addr)
- static INLINE u16 Xil_In16 (UINTPTR Addr)
- static INLINE u32 Xil_In32 (UINTPTR Addr)
- static INLINE u64 Xil_In64 (UINTPTR Addr)
- static INLINE void Xil_Out8 (UINTPTR Addr, u8 Value)
- static INLINE void Xil_Out16 (UINTPTR Addr, u16 Value)
- static INLINE void Xil_Out32 (UINTPTR Addr, u32 Value)
- static INLINE void Xil_Out64 (UINTPTR Addr, u64 Value)
- static INLINE u16 Xil_In16LE (UINTPTR Addr)
- static INLINE u32 Xil_In32LE (UINTPTR Addr)
- static INLINE void Xil_Out16LE (UINTPTR Addr, u16 Value)
- static INLINE void Xil_Out32LE (UINTPTR Addr, u32 Value)
- static INLINE u16 Xil_In16BE (UINTPTR Addr)
- static INLINE u32 Xil_In32BE (UINTPTR Addr)
- static INLINE void Xil_Out16BE (UINTPTR Addr, u16 Value)
- static INLINE void Xil_Out32BE (UINTPTR Addr, u32 Value)

Function Documentation

u16 Xil_EndianSwap16 (u16 *Data*)

Perform a 16-bit endian converion.

Parameters

Data	16 bit value to be converted

Returns

converted value.

u32 Xil_EndianSwap32 (u32 *Data*)

Perform a 32-bit endian converion.

Parameters

Data	32 bit value to be converted

Returns

converted value.

static INLINE u8 Xil_In8 (UINTPTR *Addr*) [static]

Performs an input operation for an 8-bit memory location by reading from the specified address and returning the Value read from that address.

Parameters

Addr	contains the address to perform the input operation at.

Returns

The Value read from the specified input address.

Note

None.

static INLINE u16 Xil_In16 (UINTPTR *Addr*) [static]

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the Value read from that address.

Parameters

Addr	contains the address to perform the input operation at.

Returns

The Value read from the specified input address.

Note

None.

static INLINE u32 Xil_In32 (UINTPTR *Addr*) [static]

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the Value read from that address.

Parameters

Addr	contains the address to perform the input operation at.

Returns

The Value read from the specified input address.

Note

None.

static INLINE u64 Xil_In64 (UINTPTR *Addr*) [static]

Performs an input operation for a 64-bit memory location by reading the specified Value to the the specified address.

Parameters

Addr	contains the address to perform the output operation at.
Value	contains the Value to be output at the specified address.

Returns

None.

Note

None.

static INLINE void Xil_Out8 (UINTPTR *Addr,* u8 *Value*) [static]

Performs an output operation for an 8-bit memory location by writing the specified Value to the the specified address.

Parameters

Addr	contains the address to perform the output operation at.
Value	contains the Value to be output at the specified address.

Returns

None.

Note

None.

static INLINE void Xil_Out16 (UINTPTR *Addr,* u16 *Value*) [static]

Performs an output operation for a 16-bit memory location by writing the specified Value to the the specified address.

Parameters

Addr	contains the address to perform the output operation at.
Value	contains the Value to be output at the specified address.

Returns

None.

Note

None.

static INLINE void Xil_Out32 (UINTPTR *Addr*, u32 *Value*) [static]

Performs an output operation for a 32-bit memory location by writing the specified Value to the the specified address.

Parameters

Addr	contains the address to perform the output operation at.
Value	contains the Value to be output at the specified address.

Returns

None.

Note

None.

static INLINE void Xil_Out64 (UINTPTR *Addr*, u64 *Value*) [static]

Performs an output operation for a 64-bit memory location by writing the specified Value to the the specified address.

Parameters

Addr	contains the address to perform the output operation at.
Value	contains the Value to be output at the specified address.

Returns

None.

Note

None.

static INLINE u16 Xil_In16LE (UINTPTR *Addr*) [static]

Perform a little-endian input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is big-endian, the return value is the byte-swapped value read from the address.

static INLINE u32 Xil_In32LE (UINTPTR *Addr*) [static]

Perform a little-endian input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is big-endian, the return value is the byte-swapped value read from the address.

static INLINE void Xil_Out16LE (UINTPTR *Addr,* u16 *Value*) [static]

Perform a little-endian output operation for a 16-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is big-endian, the byteswapped value is written to the address.

static INLINE void Xil_Out32LE (UINTPTR *Addr,* u32 *Value*) [static]

Perform a little-endian output operation for a 32-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is big-endian, the byteswapped value is written to the address.

static INLINE u16 Xil_In16BE (UINTPTR *Addr*) [static]

Perform an big-endian input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is little-endian, the return value is the byte-swapped value read from the address.

static INLINE u32 Xil_In32BE (UINTPTR *Addr*) [static]

Perform a big-endian input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is little-endian, the return value is the byte-swapped value read from the address.

static INLINE void Xil_Out16BE (UINTPTR *Addr,* u16 *Value*) [static]

Perform a big-endian output operation for a 16-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is little-endian, the byteswapped value is written to the address.

static INLINE void Xil_Out32BE (UINTPTR *Addr,* u32 *Value*) [static]

Perform a big-endian output operation for a 32-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is little-endian, the byteswapped value is written to the address.

Definitions for available xilinx platforms

Overview

The `xplatform_info.h` file contains definitions for various available Xilinx® platforms.

Functions

- u32 XGetPlatform_Info ()
- u32 XGetPSVersion_Info ()
- u32 XGet_Zynq_UltraMp_Platform_info ()

Function Documentation

u32 XGetPlatform_Info ( )

This API is used to provide information about platform.

Parameters

None.	

Returns

The information about platform defined in xplatform_info.h

u32 XGetPSVersion_Info ( )

This API is used to provide information about PS Silicon version.

Parameters

None.	

Returns

The information about PS Silicon version.

u32 XGet_Zynq_UltraMp_Platform_info ( )

This API is used to provide information about zynq ultrascale MP platform.

Parameters

None.	

Returns

The information about zynq ultrascale MP platform defined in xplatform_info.h

Data types for Xilinx Software IP Cores

Overview

The `xil_types.h` file contains basic types for Xilinx® software IP cores. These data types are applicable for all processors supported by Xilinx.

Macros

- #define XIL_COMPONENT_IS_READY
- #define XIL_COMPONENT_IS_STARTED

New types

New simple types.

- typedef uint8_t u8
- typedef uint16_t **u16**
- typedef uint32_t **u32**
- typedef char char8
- typedef int8_t **s8**

- typedef int16_t **s16**
- typedef int32_t **s32**
- typedef int64_t **s64**
- typedef uint64_t **u64**
- typedef int **sint32**
- typedef intptr_t **INTPTR**
- typedef uintptr_t **UINTPTR**
- typedef ptrdiff_t **PTRDIFF**
- typedef long **LONG**
- typedef unsigned long **ULONG**
- typedef void(* XInterruptHandler) (void *InstancePtr)
- typedef void(* XExceptionHandler) (void *InstancePtr)
- #define **__XUINT64__**
- #define XUINT64_MSW(x)
- #define XUINT64_LSW(x)
- #define **ULONG64_HI_MASK**
- #define **ULONG64_LO_MASK**
- #define UPPER_32_BITS(n)
- #define LOWER_32_BITS(n)

Macro Definition Documentation

#define XIL_COMPONENT_IS_READY

component has been initialized

#define XIL_COMPONENT_IS_STARTED

component has been started

#define XUINT64_MSW(*x*)

Return the most significant half of the 64 bit data type.

Parameters

x	is the 64 bit word.

Returns

The upper 32 bits of the 64 bit word.

#define XUINT64_LSW(x)

Return the least significant half of the 64 bit data type.

Parameters

x	is the 64 bit word.

Returns

The lower 32 bits of the 64 bit word.

#define UPPER_32_BITS(n)

return bits 32-63 of a number

Parameters

n	: the number we're accessing

Returns

bits 32-63 of number

Note

A basic shift-right of a 64- or 32-bit quantity. Use this to suppress the "right shift count >= width of type" warning when that quantity is 32-bits.

#define LOWER_32_BITS(n)

return bits 0-31 of a number

Parameters

n	: the number we're accessing

Returns

bits 0-31 of number

Typedef Documentation

typedef uint8_t u8

guarded against xbasic_types.h.

typedef char char8

xbasic_types.h does not typedef s* or u64

typedef void(* XInterruptHandler) (void *InstancePtr)

This data type defines an interrupt handler for a device. The argument points to the instance of the component

typedef void(* XExceptionHandler) (void *InstancePtr)

This data type defines an exception handler for a processor. The argument points to the instance of the component

Customized APIs for memory operations

Overview

The `xil_mem.h` file contains prototypes for function related to memory operations. These APIs are applicable for all processors supported by Xilinx®.

Functions

- void Xil_MemCpy (void *dst, const void *src, u32 cnt)

Function Documentation

void Xil_MemCpy (void * *dst*, const void * *src*, u32 *cnt*)

This function copies memory from once location to other.

Parameters

dst	pointer pointing to destination memory
src	pointer pointing to source memory
cnt	32 bit length of bytes to be copied

Xilinx software status codes

Overview

The `xstatus.h` file contains Xilinx® software status codes. Status codes have their own data type called `int`. These codes are used throughout the Xilinx device drivers.

Test utilities for memory and caches

Overview

The `xil_testcache.h`, `xil_testio.h` and the `xil_testmem.h` files contain utility functions to test cache and memory. Details of supported tests and subtests are listed below.

- **Cache test** : xil_testcache.h contains utility functions to test cache.
- **I/O test** : The `Xil_testio.h` file contains endian related memory IO functions. A subset of the memory tests can be selected or all of the tests can be run in order. If there is an error detected by a subtest, the test stops and the failure code is returned. Further tests are not run even if all of the tests are selected.
- **Memory test** : The `xil_testmem.h` file contains utility functions to test memory. A subset of the memory tests can be selected or all of the tests can be run in order. If there is an error detected by a subtest, the test stops and the failure code is returned. Further tests are not run even if all of the tests are selected. Following are descriptions of Memory test subtests:
  - XIL_TESTMEM_ALLMEMTESTS: Runs all of the subtests.
  - XIL_TESTMEM_INCREMENT: Incrementing Value Test. This test starts at `XIL_TESTMEM_INIT_VALUE` and uses the incrementing value as the test value for memory.
  - XIL_TESTMEM_WALKONES: Walking Ones Test. This test uses a walking 1 as the test value for memory.

    ```
    location 1 = 0x00000001
    location 2 = 0x00000002
    ...
    ```

  - XIL_TESTMEM_WALKZEROS: Walking Zero's Test. This test uses the inverse value of the walking ones test as the test value for memory.

    ```
    location 1 = 0xFFFFFFFE
    location 2 = 0xFFFFFFFD
    ...
    ```

  - XIL_TESTMEM_INVERSEADDR: Inverse Address Test. This test uses the inverse of the address of the location under test as the test value for memory.
  - XIL_TESTMEM_FIXEDPATTERN: Fixed Pattern Test. This test uses the provided patters as the test value for memory. If zero is provided as the pattern the test uses `0xDEADBEEF`.

WARNING: *The tests are **DESTRUCTIVE**. Run before any initialized memory spaces have been set up. The address provided to the memory tests is not checked for validity except for the NULL case. It is possible to provide a code-space pointer for this test to start with and ultimately destroy executable code causing random failures.*

Note

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XIL_TESTMEM_WALKONES and XIL_TESTMEM_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XIL_TESTMEM_INCREMENT and XIL_TESTMEM_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

Functions

- s32 Xil_TestIO8 (u8 *Addr, s32 Length, u8 Value)
- s32 Xil_TestIO16 (u16 *Addr, s32 Length, u16 Value, s32 Kind, s32 Swap)
- s32 Xil_TestIO32 (u32 *Addr, s32 Length, u32 Value, s32 Kind, s32 Swap)
- s32 Xil_TestMem32 (u32 *Addr, u32 Words, u32 Pattern, u8 Subtest)
- s32 Xil_TestMem16 (u16 *Addr, u32 Words, u16 Pattern, u8 Subtest)
- s32 Xil_TestMem8 (u8 *Addr, u32 Words, u8 Pattern, u8 Subtest)

Memory subtests

- #define XIL_TESTMEM_ALLMEMTESTS
- #define **XIL_TESTMEM_INCREMENT**
- #define **XIL_TESTMEM_WALKONES**
- #define **XIL_TESTMEM_WALKZEROS**
- #define **XIL_TESTMEM_INVERSEADDR**
- #define **XIL_TESTMEM_FIXEDPATTERN**
- #define **XIL_TESTMEM_MAXTEST**

Macro Definition Documentation

#define XIL_TESTMEM_ALLMEMTESTS

See the detailed description of the subtests in the file description.

Function Documentation

s32 Xil_TestIO8 (u8 * *Addr*, s32 *Length*, u8 *Value*)

Perform a destructive 8-bit wide register IO test where the register is accessed using Xil_Out8 and Xil_In8, and comparing the written values by reading them back.

Parameters

Addr	a pointer to the region of memory to be tested.
Length	Length of the block.
Value	constant used for writting the memory.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

s32 Xil_TestIO16 (u16 * *Addr*, s32 *Length*, u16 *Value*, s32 *Kind*, s32 *Swap*)

Perform a destructive 16-bit wide register IO test. Each location is tested by sequentially writing a 16-bit wide register, reading the register, and comparing value. This function tests three kinds of register IO functions, normal register IO, little-endian register IO, and big-endian register IO. When testing little/big-endian IO, the function performs the following sequence, Xil_Out16LE/Xil_Out16BE, Xil_In16, Compare In-Out values, Xil_Out16, Xil_In16LE/Xil_In16BE, Compare In-Out values. Whether to swap the read-in value before comparing is controlled by the 5th argument.

Parameters

Addr	a pointer to the region of memory to be tested.
Length	Length of the block.
Value	constant used for writting the memory.
Kind	Type of test. Acceptable values are: XIL_TESTIO_DEFAULT, XIL_TESTIO_LE, XIL_TESTIO_BE.
Swap	indicates whether to byte swap the read-in value.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

s32 Xil_TestIO32 (u32 * *Addr*, s32 *Length*, u32 *Value*, s32 *Kind*, s32 *Swap*)

Perform a destructive 32-bit wide register IO test. Each location is tested by sequentially writing a 32-bit wide regsiter, reading the register, and comparing value. This function tests three kinds of register IO functions, normal register IO, little-endian register IO, and big-endian register IO. When testing little/big-endian IO, the function perform the following sequence, Xil_Out32LE/ Xil_Out32BE, Xil_In32, Compare, Xil_Out32, Xil_In32LE/Xil_In32BE, Compare. Whether to swap the read-in value *before comparing is controlled by the 5th argument.

Parameters

Addr	a pointer to the region of memory to be tested.
Length	Length of the block.
Value	constant used for writting the memory.
Kind	type of test. Acceptable values are: XIL_TESTIO_DEFAULT, XIL_TESTIO_LE, XIL_TESTIO_BE.
Swap	indicates whether to byte swap the read-in value.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

s32 Xil_TestMem32 (u32 * *Addr*, u32 *Words*, u32 *Pattern*, u8 *Subtest*)

Perform a destructive 32-bit wide memory test.

Parameters

Addr	pointer to the region of memory to be tested.
Words	length of the block.
Pattern	constant used for the constant pattern test, if 0, 0xDEADBEEF is used.
Subtest	test type selected. See xil_testmem.h for possible values.

Returns

- 0 is returned for a pass
- 1 is returned for a failure

Note

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** Width, the patterns used in XIL_TESTMEM_WALKONES and XIL_TESTMEM_WALKZEROS will repeat on a boundry of a power of two making it more difficult to detect addressing errors. The XIL_TESTMEM_INCREMENT and XIL_TESTMEM_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

s32 Xil_TestMem16 (u16 * *Addr,* u32 *Words,* u16 *Pattern,* u8 *Subtest*)

Perform a destructive 16-bit wide memory test.

Parameters

Addr	pointer to the region of memory to be tested.
Words	length of the block.
Pattern	constant used for the constant Pattern test, if 0, 0xDEADBEEF is used.
Subtest	type of test selected. See xil_testmem.h for possible values.

Returns

```
- -1 is returned for a failure
- 0 is returned for a pass
```

Note

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** Width, the patterns used in XIL_TESTMEM_WALKONES and XIL_TESTMEM_WALKZEROS will repeat on a boundry of a power of two making it more difficult to detect addressing errors. The XIL_TESTMEM_INCREMENT and XIL_TESTMEM_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

s32 Xil_TestMem8 (u8 * *Addr,* u32 *Words,* u8 *Pattern,* u8 *Subtest*)

Perform a destructive 8-bit wide memory test.

Parameters

Addr	pointer to the region of memory to be tested.
Words	length of the block.
Pattern	constant used for the constant pattern test, if 0, 0xDEADBEEF is used.
Subtest	type of test selected. See xil_testmem.h for possible values.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

Note

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** Width, the patterns used in XIL_TESTMEM_WALKONES and XIL_TESTMEM_WALKZEROS will repeat on a boundry of a power of two making it more difficult to detect addressing errors. The XIL_TESTMEM_INCREMENT and XIL_TESTMEM_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

XILINX
ALL PROGRAMMABLE™

Chapter 2

MicroBlaze Processor API

Overview

This section provides a linked summary and detailed descriptions of the MicroBlaze Processor APIs.

Modules

- MicroBlaze Pseudo-asm Macros and Interrupt handling APIs
- MicroBlaze exception APIs
- MicroBlaze Processor Cache APIs
- MicroBlaze Processor FSL Macros
- MicroBlaze PVR access routines and macros
- Sleep Routines for MicroBlaze

MicroBlaze Pseudo-asm Macros and Interrupt handling APIs

Overview

Standalone includes macros to provide convenient access to various registers in the MicroBlaze processor. Some of these macros are very useful within exception handlers for retrieving information about the exception.Also, the interrupt handling functions help manage interrupt handling on MicroBlaze processor devices.To use these functions, include the header file mb_interface.h in your source code

Functions

- void microblaze_register_handler (XInterruptHandler Handler, void *DataPtr)
- void microblaze_register_exception_handler (u32 ExceptionId, Xil_ExceptionHandler Handler, void *DataPtr)

MicroBlaze pseudo-asm macros

The following is a summary of the MicroBlaze processor pseudo-asm macros.

- #define mfgpr(rn)
- #define mfmsr()
- #define mfear()
- #define **mfeare**()
- #define mfesr()
- #define mffsr()

Macro Definition Documentation

#define mfgpr(*rn*)

Return value from the general purpose register (GPR) rn.

Parameters

rn	General purpose register to be read.

#define mfmsr( )

Return the current value of the MSR.

Parameters

None	

#define mfear( )

Return the current value of the Exception Address Register (EAR).

Parameters

None	

#define mfesr( )

Return the current value of the Exception Status Register (ESR).

Parameters

None	

#define mffsr( )

Return the current value of the Floating Point Status (FPS).

Parameters

None	

Function Documentation

void microblaze_register_handler (XInterruptHandler *Handler,* void * *DataPtr*)

Registers a top-level interrupt handler for the MicroBlaze. The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

Parameters

Handler	Top level handler.
DataPtr	a reference to data that will be passed to the handler when it gets called.

Returns

None.

void microblaze_register_exception_handler (u32 *ExceptionId,* Xil_ExceptionHandler *Handler,* void * *DataPtr*)

Registers an exception handler for the MicroBlaze. The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

Parameters

ExceptionId	is the id of the exception to register this handler for.
Top	level handler.
DataPtr	is a reference to data that will be passed to the handler when it gets called.

Returns

None.

Note

None.

MicroBlaze exception APIs

Overview

The `xil_exception.h` file, available in the <install-directory>/src/MicroBlaze folder, contains MicroBlaze specific exception related APIs and macros. Application programs can use these APIs for various exception related operations. For example, enable exception, disable exception, register exception hander.

Note

To use exception related functions, `xil_exception.h` must be added in source code

Data Structures

- struct MB_ExceptionVectorTableEntry

Typedefs

- typedef void(∗ Xil_ExceptionHandler) (void ∗Data)
- typedef void(∗ XInterruptHandler) (void ∗InstancePtr)

Functions

- void Xil_ExceptionInit (void)
- void Xil_ExceptionEnable (void)
- void Xil_ExceptionDisable (void)
- void Xil_ExceptionRegisterHandler (u32 Id, Xil_ExceptionHandler Handler, void ∗Data)
- void Xil_ExceptionRemoveHandler (u32 Id)

Data Structure Documentation

struct MB_ExceptionVectorTableEntry

Currently HAL is an augmented part of standalone BSP, so the old definition of MB_ExceptionVectorTableEntry is used here.

Typedef Documentation

typedef void(* Xil_ExceptionHandler) (void *Data)

This typedef is the exception handler function.

typedef void(* XInterruptHandler) (void *InstancePtr)

This data type defines an interrupt handler for a device. The argument points to the instance of the component

Function Documentation

void Xil_ExceptionInit (void)

Initialize exception handling for the processor. The exception vector table is setup with the stub handler for all exceptions.

Parameters

None.	

Returns

None.

void Xil_ExceptionEnable (void)

Enable Exceptions.

Returns

None.

void Xil_ExceptionDisable (void)

Disable Exceptions.

Parameters

None.	

Returns

None.

void Xil_ExceptionRegisterHandler (u32 *Id,* Xil_ExceptionHandler *Handler,* void * *Data*)

Makes the connection between the Id of the exception source and the associated handler that is to run when the exception is recognized. The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

Parameters

Id	contains the 32 bit ID of the exception source and should be XIL_EXCEPTION_INT or be in the range of 0 to XIL_EXCEPTION_LAST. See xil_mach_exception.h for further information.
Handler	handler function to be registered for exception
Data	a reference to data that will be passed to the handler when it gets called.

void Xil_ExceptionRemoveHandler (u32 *Id*)

Removes the handler for a specific exception Id. The stub handler is then registered for this exception Id.

Parameters

Id	contains the 32 bit ID of the exception source and should be XIL_EXCEPTION_INT or in the range of 0 to XIL_EXCEPTION_LAST. See xexception_l.h for further information.

MicroBlaze Processor Cache APIs

Overview

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Note

Macros

- void Xil_L1DCacheInvalidate()
- void Xil_L2CacheInvalidate()
- void Xil_L1DCacheInvalidateRange(Addr, Len)
- void Xil_L2CacheInvalidateRange(Addr, Len)
- void Xil_L1DCacheFlushRange(Addr, Len)

- void Xil_L2CacheFlushRange(Addr, Len)
- void Xil_L1DCacheFlush()
- void Xil_L2CacheFlush()
- void Xil_L1ICacheInvalidateRange(Addr, Len)
- void Xil_L1ICacheInvalidate()
- void Xil_L1DCacheEnable()
- void Xil_L1DCacheDisable()
- void Xil_L1ICacheEnable()
- void Xil_L1ICacheDisable()
- void Xil_DCacheEnable()
- void Xil_ICacheEnable()

Functions

- void Xil_DCacheDisable (void)
- void Xil_ICacheDisable (void)

Macro Definition Documentation

void Xil_L1DCacheInvalidate( )

Invalidate the entire L1 data cache. If the cacheline is modified (dirty), the modified contents are lost.

Parameters

None.	

Returns

None.

Note

Processor must be in real mode.

void Xil_L2CacheInvalidate( )

Invalidate the entire L2 data cache. If the cacheline is modified (dirty),the modified contents are lost.

Parameters

None.	

Returns

None.

Note

Processor must be in real mode.

void Xil_L1DCacheInvalidateRange(*Addr, Len*)

Invalidate the L1 data cache for the given address range. If the bytes specified by the address (Addr) are cached by the L1 data cache, the cacheline containing that byte is invalidated.If the cacheline is modified (dirty), the modified contents are lost.

Parameters

Addr	is address of ragne to be invalidated.
Len	is the length in bytes to be invalidated.

Returns

None.

Note

Processor must be in real mode.

void Xil_L2CacheInvalidateRange(*Addr, Len*)

Invalidate the L1 data cache for the given address range. If the bytes specified by the address (Addr) are cached by the L1 data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost.

Parameters

Addr	address of ragne to be invalidated.
Len	length in bytes to be invalidated.

Returns

None.

Note

Processor must be in real mode.

void Xil_L1DCacheFlushRange(*Addr, Len*)

Flush the L1 data cache for the given address range. If the bytes specified by the address (Addr) are cached by the data cache, and is modified (dirty), the cacheline will be written to system memory.The cacheline will also be invalidated.

Parameters

Addr	the starting address of the range to be flushed.
Len	length in byte to be flushed.

Returns

None.

void Xil_L2CacheFlushRange(*Addr, Len*)

Flush the L2 data cache for the given address range. If the bytes specified by the address (Addr) are cached by the data cache, and is modified (dirty), the cacheline will be written to system memory. The cacheline will also be invalidated.

Parameters

Addr	the starting address of the range to be flushed.
Len	length in byte to be flushed.

Returns

None.

void Xil_L1DCacheFlush( )

Flush the entire L1 data cache. If any cacheline is dirty, the cacheline will be written to system memory. The entire data cache will be invalidated.

Returns

None.

void Xil_L2CacheFlush( )

Flush the entire L2 data cache. If any cacheline is dirty, the cacheline will be written to system memory. The entire data cache will be invalidated.

Returns

None.

void Xil_L1ICacheInvalidateRange(*Addr, Len*)

Invalidate the instruction cache for the given address range.

Parameters

Addr	is address of ragne to be invalidated.
Len	is the length in bytes to be invalidated.

Returns

None.

void Xil_L1ICacheInvalidate( )

Invalidate the entire instruction cache.

Parameters

None	

Returns

None.

void Xil_L1DCacheEnable( )

Enable the L1 data cache.

Returns

None.

void Xil_L1DCacheDisable( )

Disable the L1 data cache.

Returns

None.

Note

This is processor specific.

void Xil_L1ICacheEnable( )

Enable the instruction cache.

Returns

None.

Note

This is processor specific.

void Xil_L1ICacheDisable( )

Disable the L1 Instruction cache.

Returns

None.

Note

This is processor specific.

void Xil_DCacheEnable( )

Enable the data cache.

Parameters

None	

Returns

None.

void Xil_ICacheEnable( )

Enable the instruction cache.

Parameters

None	

Returns

None.

Note

Function Documentation

void Xil_DCacheDisable (void)

Disable the data cache.

Parameters

None	

Returns

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None	

Returns

None.

MicroBlaze Processor FSL Macros

Overview

Standalone includes macros to provide convenient access to accelerators connected to the MicroBlaze Fast Simplex Link (FSL) Interfaces.To use these functions, include the header file fsl.h in your source code

Macros

- #define getfslx(val, id, flags)
- #define putfslx(val, id, flags)
- #define tgetfslx(val, id, flags)
- #define tputfslx(id, flags)
- #define getdfslx(val, var, flags)
- #define putdfslx(val, var, flags)
- #define tgetdfslx(val, var, flags)
- #define tputdfslx(var, flags)

Macro Definition Documentation

#define getfslx(*val, id, flags*)

Performs a get function on an input FSL of the MicroBlaze processor

Parameters

val	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
id	FSL identifier
flags	valid FSL macro flags

#define putfslx(*val, id, flags*)

Performs a put function on an input FSL of the MicroBlaze processor

Parameters

val	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
id	FSL identifier
flags	valid FSL macro flags

#define tgetfslx(*val, id, flags*)

Performs a test get function on an input FSL of the MicroBlaze processor

Parameters

val	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
id	FSL identifier
flags	valid FSL macro flags

#define tputfslx(*id, flags*)

Performs a put function on an input FSL of the MicroBlaze processor

Parameters

id	FSL identifier
flags	valid FSL macro flags

#define getdfslx(*val, var, flags*)

Performs a get function on an input FSL of the MicroBlaze processor

Parameters

val	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
var	FSL identifier
flags	valid FSL macro flags

#define putdfslx(*val, var, flags*)

Performs a put function on an input FSL of the MicroBlaze processor

Parameters

val	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
var	FSL identifier
flags	valid FSL macro flags

#define tgetdfslx(*val, var, flags*)

Performs a test get function on an input FSL of the MicroBlaze processor;

Parameters

val	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
var	FSL identifier
flags	valid FSL macro flags

#define tputdfslx(*var, flags*)

Performs a put function on an input FSL of the MicroBlaze processor

Parameters

var	FSL identifier
flags	valid FSL macro flags

MicroBlaze PVR access routines and macros

Overview

MicroBlaze processor v5.00.a and later versions have configurable Processor Version Registers (PVRs). The contents of the PVR are captured using the pvr_t data structure, which is defined as an array of 32-bit words, with each word corresponding to a PVR register on hardware. The number of PVR words is determined by the number of PVRs configured in the hardware. You should not attempt to access PVR registers that are not present in hardware, as the pvr_t data structure is resized to hold only as many PVRs as are present in hardware. To access information in the PVR:

1. Use the microblaze_get_pvr() function to populate the PVR data into a pvr_t data structure.
2. In subsequent steps, you can use any one of the PVR access macros list to get individual data stored in the PVR.
3. pvr.h header file must be included to source to use PVR macros.

Macros

- #define MICROBLAZE_PVR_IS_FULL(_pvr)
- #define MICROBLAZE_PVR_USE_BARREL(_pvr)
- #define MICROBLAZE_PVR_USE_DIV(_pvr)
- #define MICROBLAZE_PVR_USE_HW_MUL(_pvr)
- #define MICROBLAZE_PVR_USE_FPU(_pvr)
- #define MICROBLAZE_PVR_USE_ICACHE(_pvr)
- #define MICROBLAZE_PVR_USE_DCACHE(_pvr)
- #define MICROBLAZE_PVR_MICROBLAZE_VERSION(_pvr)
- #define MICROBLAZE_PVR_USER1(_pvr)
- #define MICROBLAZE_PVR_USER2(_pvr)
- #define MICROBLAZE_PVR_D_LMB(_pvr)
- #define MICROBLAZE_PVR_D_PLB(_pvr)
- #define MICROBLAZE_PVR_I_LMB(_pvr)
- #define MICROBLAZE_PVR_I_PLB(_pvr)
- #define MICROBLAZE_PVR_INTERRUPT_IS_EDGE(_pvr)
- #define MICROBLAZE_PVR_EDGE_IS_POSITIVE(_pvr)
- #define MICROBLAZE_PVR_INTERCONNECT(_pvr)
- #define MICROBLAZE_PVR_USE_MUL64(_pvr)
- #define MICROBLAZE_PVR_OPCODE_0x0_ILLEGAL(_pvr)
- #define MICROBLAZE_PVR_UNALIGNED_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_ILL_OPCODE_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_IPLB_BUS_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_DPLB_BUS_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_DIV_ZERO_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_FPU_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_FSL_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_DEBUG_ENABLED(_pvr)
- #define MICROBLAZE_PVR_NUMBER_OF_PC_BRK(_pvr)
- #define MICROBLAZE_PVR_NUMBER_OF_RD_ADDR_BRK(_pvr)
- #define MICROBLAZE_PVR_NUMBER_OF_WR_ADDR_BRK(_pvr)
- #define MICROBLAZE_PVR_FSL_LINKS(_pvr)
- #define MICROBLAZE_PVR_ICACHE_ADDR_TAG_BITS(_pvr)
- #define MICROBLAZE_PVR_ICACHE_ALLOW_WR(_pvr)
- #define MICROBLAZE_PVR_ICACHE_LINE_LEN(_pvr)
- #define MICROBLAZE_PVR_ICACHE_BYTE_SIZE(_pvr)
- #define MICROBLAZE_PVR_DCACHE_ADDR_TAG_BITS(_pvr)
- #define MICROBLAZE_PVR_DCACHE_ALLOW_WR(_pvr)
- #define MICROBLAZE_PVR_DCACHE_LINE_LEN(_pvr)
- #define MICROBLAZE_PVR_DCACHE_BYTE_SIZE(_pvr)
- #define MICROBLAZE_PVR_ICACHE_BASEADDR(_pvr)
- #define MICROBLAZE_PVR_ICACHE_HIGHADDR(_pvr)
- #define MICROBLAZE_PVR_DCACHE_BASEADDR(_pvr)

- #define MICROBLAZE_PVR_DCACHE_HIGHADDR(_pvr)
- #define MICROBLAZE_PVR_TARGET_FAMILY(_pvr)
- #define MICROBLAZE_PVR_MSR_RESET_VALUE(_pvr)
- #define MICROBLAZE_PVR_MMU_TYPE(_pvr)

Functions

- int microblaze_get_pvr (pvr_t *pvr)

Macro Definition Documentation

#define MICROBLAZE_PVR_IS_FULL(*_pvr*)

Return non-zero integer if PVR is of type FULL, 0 if basic

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_BARREL(*_pvr*)

Return non-zero integer if hardware barrel shifter present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_DIV(*_pvr*)

Return non-zero integer if hardware divider present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_HW_MUL(*_pvr*)

Return non-zero integer if hardware multiplier present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_FPU(_pvr)

Return non-zero integer if hardware floating point unit (FPU) present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_ICACHE(_pvr)

Return non-zero integer if I-cache present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_DCACHE(_pvr)

Return non-zero integer if D-cache present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_MICROBLAZE_VERSION(_pvr)

Return MicroBlaze processor version encoding. Refer to the MicroBlaze Processor Reference Guide (UG081) for mappings from encodings to actual hardware versions.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USER1(_pvr)

Return the USER1 field stored in the PVR.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USER2(_pvr)

Return the USER2 field stored in the PVR.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_D_LMB(_pvr)

Return non-zero integer if Data Side PLB interface is present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_D_PLB(_pvr)

Return non-zero integer if Data Side PLB interface is present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_I_LMB(_pvr)

Return non-zero integer if Instruction Side Local Memory Bus (LMB) interface present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_I_PLB(_pvr)

Return non-zero integer if Instruction Side PLB interface present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_INTERRUPT_IS_EDGE(_pvr)

Return non-zero integer if interrupts are configured as edge-triggered.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_EDGE_IS_POSITIVE(_pvr)

Return non-zero integer if interrupts are configured as positive edge triggered.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_INTERCONNECT(_pvr)

Return non-zero if MicroBlaze processor has PLB interconnect; otherwise return zero.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_USE_MUL64(_pvr)

Return non-zero integer if MicroBlaze processor supports 64-bit products for multiplies.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_OPCODE_0x0_ILLEGAL(_pvr)

Return non-zero integer if opcode 0x0 is treated as an illegal opcode. multiplies.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_UNALIGNED_EXCEPTION(_pvr)

Return non-zero integer if unaligned exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ILL_OPCODE_EXCEPTION(_pvr)

Return non-zero integer if illegal opcode exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_IPLB_BUS_EXCEPTION(_pvr)

Return non-zero integer if I-PLB exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DPLB_BUS_EXCEPTION(_pvr)

Return non-zero integer if I-PLB exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DIV_ZERO_EXCEPTION(_pvr)

Return non-zero integer if divide by zero exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_FPU_EXCEPTION(_pvr)

Return non-zero integer if FPU exceptions are supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_FSL_EXCEPTION(_pvr)

Return non-zero integer if FSL exceptions are present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DEBUG_ENABLED(_pvr)

Return non-zero integer if debug is enabled.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_NUMBER_OF_PC_BRK(_pvr)

Return the number of hardware PC breakpoints available.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_NUMBER_OF_RD_ADDR_BRK(_pvr)

Return the number of read address hardware watchpoints supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_NUMBER_OF_WR_ADDR_BRK(_pvr)

Return the number of write address hardware watchpoints supported.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_FSL_LINKS(_pvr)

Return the number of FSL links present.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_ADDR_TAG_BITS(_pvr)

Return the number of address tag bits for the I-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_ALLOW_WR(_pvr)

Return non-zero if writes to I-caches are allowed.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_LINE_LEN(_pvr)

Return the length of each I-cache line in bytes.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_BYTE_SIZE(_pvr)

Return the size of the D-cache in bytes.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_ADDR_TAG_BITS(_pvr)

Return the number of address tag bits for the D-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_ALLOW_WR(_pvr)

Return non-zero if writes to D-cache are allowed.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_LINE_LEN(_pvr)

Return the length of each line in the D-cache in bytes.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_BYTE_SIZE(_pvr)

Return the size of the D-cache in bytes.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_BASEADDR(_pvr)

Return the base address of the I-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_ICACHE_HIGHADDR(_pvr)

Return the high address of the I-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_BASEADDR(_pvr)

Return the base address of the D-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_DCACHE_HIGHADDR(_pvr)

Return the high address of the D-cache.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_TARGET_FAMILY(_pvr)

Return the encoded target family identifier.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_MSR_RESET_VALUE(_pvr)

Refer to the MicroBlaze Processor Reference Guide (UG081) for mappings from encodings to target family name strings.

Parameters

_pvr	pvr data structure

#define MICROBLAZE_PVR_MMU_TYPE(_pvr)

Returns the value of C_USE_MMU. Refer to the MicroBlaze Processor Reference Guide (UG081) for mappings from MMU type values to MMU function.

Parameters

_pvr	pvr data structure

Function Documentation

int microblaze_get_pvr (pvr_t * *pvr*)

Populate the PVR data structure to which pvr points with the values of the hardware PVR registers.

Parameters

pvr-	address of PVR data structure to be populated

Returns

0 - SUCCESS -1 - FAILURE

Sleep Routines for MicroBlaze

Overview

microblaze_sleep.h contains microblaze sleep APIs. These APIs provides delay for requested duration.

Note

microblaze_sleep.h may contain architecture-dependent items.

Functions

- void MB_Sleep (u32 MilliSeconds) __attribute__((__deprecated__))

Function Documentation

void MB_Sleep (u32 *MilliSeconds*)

Provides delay for requested duration..

Parameters

MilliSeconds-	Delay time in milliseconds.

Returns

None.

Note

Instruction cache should be enabled for this to work.

XILINX
ALL PROGRAMMABLE™

Chapter 3

Cortex R5 Processor API

Overview

Standalone BSP contains boot code, cache, exception handling, file and memory management, configuration, time and processor-specific include functions. It supports gcc compiler. This section provides a linked summary and detailed descriptions of the Cortex R5 processor APIs.

Modules

Cortex R5 Processor Boot Code

Overview

The `boot.S` file contains a minimal set of code for transferring control from the processor's reset location to the start of the application. The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)
3. Disable instruction cache, data cache and MPU
4. Invalidate instruction and data cache
5. Configure MPU with short descriptor translation table format and program base address of translation table

6. Enable data cache, instruction cache and MPU
7. Enable Floating point unit
8. Transfer control to _start which clears BSS sections and jumping to main application

Cortex R5 Processor MPU specific APIs

Overview

MPU functions provides access to MPU operations such as enable MPU, disable MPU and set attribute for section of memory. Boot code invokes Init_MPU function to configure the MPU. A total of 10 MPU regions are allocated with another 6 being free for users. Overview of the memory attributes for different MPU regions is as given below,

	Memory Range	Attributes of MPURegion	Note
DDR	0x00000000 - 0x7FFFFFFF	Normal write-back Cacheable	For a system where DDR is less than 2GB, region after DDR and before PL is marked as undefined in translation table
PL	0x80000000 - 0xBFFFFFFF	Strongly Ordered	
QSPI	0xC0000000 - 0xDFFFFFFF	Device Memory	
PCIe	0xE0000000 - 0xEFFFFFFF	Device Memory	
STM_CORESIGHT	0xF8000000 - 0xF8FFFFFF	Device Memory	

	Memory Range	Attributes of MPURegion	Note
RPU_R5_GIC	0xF9000000 - 0xF90FFFFF	Device Memory	
FPS	0xFD000000 - 0xFDFFFFFF	Device Memory	
LPS	0xFE000000 - 0xFFFFFFFF	Device Memory	0xFE000000 - 0xFEFFFFFF upper LPS slaves, 0xFF000000 - 0xFFFFFFFF lower LPS slaves
OCM	0xFFFC0000 - 0xFFFFFFFF	Normal write-back Cacheable	

Functions

- void Xil_SetTlbAttributes (INTPTR Addr, u32 attrib)
- void Xil_EnableMPU (void)
- void Xil_DisableMPU (void)
- void Xil_SetMPURegion (INTPTR addr, u64 size, u32 attrib)

Function Documentation

void Xil_SetTlbAttributes (INTPTR *addr*, u32 *attrib*)

This function sets the memory attributes for a section covering 1MB, of memory in the translation table.

Parameters

Addr	32-bit address for which memory attributes need to be set.
attrib	Attribute for the given memory region.

Returns

None.

void Xil_EnableMPU (void)

Enable MPU for Cortex R5 processor. This function invalidates I cache and flush the D Caches, and then enables the MPU.

Parameters

None.	

Returns

None.

void Xil_DisableMPU (void)

Disable MPU for Cortex R5 processors. This function invalidates I cache and flush the D Caches, and then disabes the MPU.

Parameters

None.	

Returns

None.

void Xil_SetMPURegion (INTPTR *addr*, u64 *size*, u32 *attrib*)

Set the memory attributes for a section of memory in the translation table.

Parameters

Addr	32-bit address for which memory attributes need to be set..
size	size is the size of the region.
attrib	Attribute for the given memory region.

Returns

None.

Cortex R5 Processor Cache Functions

Overview

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Functions

- void Xil_DCacheEnable (void)
- void Xil_DCacheDisable (void)
- void Xil_DCacheInvalidate (void)
- void Xil_DCacheInvalidateRange (INTPTR adr, u32 len)
- void Xil_DCacheFlush (void)
- void Xil_DCacheFlushRange (INTPTR adr, u32 len)
- void Xil_DCacheInvalidateLine (INTPTR adr)
- void Xil_DCacheFlushLine (INTPTR adr)
- void Xil_DCacheStoreLine (INTPTR adr)
- void Xil_ICacheEnable (void)
- void Xil_ICacheDisable (void)
- void Xil_ICacheInvalidate (void)
- void Xil_ICacheInvalidateRange (INTPTR adr, u32 len)
- void Xil_ICacheInvalidateLine (INTPTR adr)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheDisable (void)

Disable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheInvalidate (void)

Invalidate the entire Data cache.

Parameters

None.	

Returns

None.

void Xil_DCacheInvalidateRange (INTPTR *adr,* u32 *len*)

Invalidate the Data cache for the given address range. If the bytes specified by the address (adr) are cached by the Data cache,the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of range to be invalidated in bytes.

Returns

None.

void Xil_DCacheFlush (void)

Flush the entire Data cache.

Parameters

None.	

Returns

None.

void Xil_DCacheFlushRange (INTPTR *adr*, u32 *len*)

Flush the Data cache for the given address range. If the bytes specified by the address (adr) are cached by the Data cache, the cacheline containing those bytes is invalidated.If the cacheline is modified (dirty), the written to system memory before the lines are invalidated.

Parameters

adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes

Returns

None.

void Xil_DCacheInvalidateLine (INTPTR *adr*)

Invalidate a Data cache line. If the byte specified by the address (adr) is cached by the data cache, the cacheline containing that byte is invalidated.If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_DCacheFlushLine (INTPTR *adr*)

Flush a Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_DCacheStoreLine (INTPTR *adr*)

Store a Data cache line. If the byte specified by the address (adr) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory.After the store completes, the cacheline is marked as unmodified (not dirty).

Parameters

adr	32bit address of the data to be stored

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

None.	

Returns

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None.	

Returns

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	

Returns

None.

void Xil_ICacheInvalidateRange (INTPTR *adr,* u32 *len*)

Invalidate the instruction cache for the given address range. If the bytes specified by the address (adr) are cached by the Data cache, the cacheline containing that byte is invalidated. If the cachelineis modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

void Xil_ICacheInvalidateLine (INTPTR *adr*)

Invalidate an instruction cache line.If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Parameters

adr	32bit address of the instruction to be invalidated.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

Cortex R5 Time Functions

Overview

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 32-bit counter in TTC. The `sleep.c`, `usleep.c` file and the corresponding `sleep.h` include file implement sleep functions. Sleep functions are implemented as busy loops.

Functions

- void XTime_StartTimer (void)
- void XTime_SetTime (XTime Xtime_Global)
- void XTime_GetTime (XTime *Xtime_Global)

Function Documentation

void XTime_StartTimer (void)

Starts the TTC timer 3 counter 0 if present and if it is not already running with desired parameters for sleep functionalities.

Parameters

None.	

Returns

None.

Note

When this function is called by any one processor in a multi- processor environment, reference time will reset/lost for all processors.

void XTime_SetTime (XTime *Xtime_Global*)

TTC Timer runs continuously and the time can not be set as desired. This API doesn't contain anything. It is defined to have uniformity across platforms.

Parameters

Xtime_Global	32 bit value to be written to the timer counter register.

Returns

None.

Note

In multiprocessor environment reference time will reset/lost for all processors, when this function called by any one processor.

void XTime_GetTime (XTime * *Xtime_Global*)

Get the time from the timer counter register.

Parameters

Xtime_Global	Pointer to the 32 bit location to be updated with the time current value of timer counter register.

Returns

None.

Cortex R5 Event Counters Functions

Overview

Cortex R5 event counter functions can be utilized to configure and control the Cortex-R5 performance monitor events. Cortex-R5 Performance Monitor has 6 event counters which can be used to count a variety of events described in Coretx-R5 TRM. xpm_counter.h defines configurations XPM_CNTRCFGx which can be used to program the event counters to count a set of events.

Note

It doesn't handle the Cortex-R5 cycle counter, as the cycle counter is being used for time keeping.

Functions

- void Xpm_SetEvents (s32 PmcrCfg)
- void Xpm_GetEventCounters (u32 *PmCtrValue)

Function Documentation

void Xpm_SetEvents (s32 *PmcrCfg*)

This function configures the Cortex R5 event counters controller, with the event codes, in a configuration selected by the user and enables the counters.

Parameters

PmcrCfg	Configuration value based on which the event counters are configured.XPM_CNTRCFG* values defined in xpm_counter.h can be utilized for setting configuration

Returns

None.

void Xpm_GetEventCounters (u32 * *PmCtrValue*)

This function disables the event counters and returns the counter values.

Parameters

PmCtrValue	Pointer to an array of type u32 PmCtrValue[6]. It is an output parameter which is used to return the PM counter values.

Returns

None.

Cortex R5 Processor Specific Include Files

Overview

The `xpseudo_asm.h` file includes `xreg_cortexr5.h` and `xpseudo_asm_gcc.h`.
The `xreg_cortexr5.h` include file contains the register numbers and the register bits for the ARM Cortex-R5 processor.
The `xpseudo_asm_gcc.h` file contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization,or cache manipulation. These inline assembler instructions can be used from drivers and user applications written in C.

XILINX
ALL PROGRAMMABLE™

Chapter 4

ARM Processor Common API

Overview

This section provides a linked summary and detailed descriptions of the ARM Processor Common APIs.

Modules

- ARM Processor Exception Handling

ARM Processor Exception Handling

Overview

ARM processors specific exception related APIs for cortex A53,A9 and R5 can utilized for enabling/disabling IRQ, registering/removing handler for exceptions or initializing exception vector table with null handler.

Macros

- #define Xil_ExceptionEnableMask(Mask)
- #define Xil_ExceptionEnable()
- #define Xil_ExceptionDisableMask(Mask)
- #define Xil_ExceptionDisable()
- #define Xil_EnableNestedInterrupts()
- #define Xil_DisableNestedInterrupts()

Typedefs

- typedef void(* Xil_ExceptionHandler) (void *data)

Functions

- void Xil_ExceptionRegisterHandler (u32 Exception_id, Xil_ExceptionHandler Handler, void *Data)

- void Xil_ExceptionRemoveHandler (u32 Exception_id)
- void Xil_ExceptionInit (void)
- void Xil_DataAbortHandler (void *CallBackRef)
- void Xil_PrefetchAbortHandler (void *CallBackRef)
- void Xil_UndefinedExceptionHandler (void *CallBackRef)

Macro Definition Documentation

#define Xil_ExceptionEnableMask(*Mask*)

Enable Exceptions.

Parameters

Mask	for exceptions to be enabled.

Returns

None.

Note

If bit is 0, exception is enabled. C-Style signature: void Xil_ExceptionEnableMask(Mask)

#define Xil_ExceptionEnable( )

Enable the IRQ exception.

Returns

None.

Note

None.

#define Xil_ExceptionDisableMask(*Mask*)

Disable Exceptions.

Parameters

Mask	for exceptions to be enabled.

Returns

None.

Note

If bit is 1, exception is disabled. C-Style signature: Xil_ExceptionDisableMask(Mask)

#define Xil_ExceptionDisable( )

Disable the IRQ exception.

Returns

None.

Note

None.

#define Xil_EnableNestedInterrupts( )

Enable nested interrupts by clearing the I and F bits in CPSR. This API is defined for cortex-a9 and cortex-r5.

Returns

None.

Note

This macro is supposed to be used from interrupt handlers. In the interrupt handler the interrupts are disabled by default (I and F are 1). To allow nesting of interrupts, this macro should be used. It clears the I and F bits by changing the ARM mode to system mode. Once these bits are cleared and provided the preemption of interrupt conditions are met in the GIC, nesting of interrupts will start happening. Caution: This macro must be used with caution. Before calling this macro, the user must ensure that the source of the current IRQ is appropriately cleared. Otherwise, as soon as we clear the I and F bits, there can be an infinite loop of interrupts with an eventual crash (all the stack space getting consumed).

#define Xil_DisableNestedInterrupts( )

Disable the nested interrupts by setting the I and F bits. This API is defined for cortex-a9 and cortex-r5.

Returns

None.

Note

This macro is meant to be called in the interrupt service routines. This macro cannot be used independently. It can only be used when nesting of interrupts have been enabled by using the macro Xil_EnableNestedInterrupts(). In a typical flow, the user first calls the Xil_EnableNestedInterrupts in the ISR at the appropriate point. The user then must call this macro before exiting the interrupt service routine. This macro puts the ARM back in IRQ/FIQ mode and hence sets back the I and F bits.

Typedef Documentation

typedef void(* Xil_ExceptionHandler) (void *data)

This typedef is the exception handler function.

Function Documentation

void Xil_ExceptionRegisterHandler (u32 *Exception_id,* Xil_ExceptionHandler *Handler,* void * *Data*)

Register a handler for a specific exception. This handler is being called when the processor encounters the specified exception.

Parameters

exception_id	contains the ID of the exception source and should be in the range of 0 to XIL_EXCEPTION_ID_LAST. See xil_exception.h for further information.
Handler	to the Handler for that exception.
Data	is a reference to Data that will be passed to the Handler when it gets called.

Returns

None.

Note

None.

void Xil_ExceptionRemoveHandler (u32 *Exception_id*)

Removes the Handler for a specific exception Id. The stub Handler is then registered for this exception Id.

Parameters

exception_id	contains the ID of the exception source and should be in the range of 0 to XIL_EXCEPTION_ID_LAST. See xil_exception.h for further information.

Returns

None.

Note

None.

void Xil_ExceptionInit (void)

The function is a common API used to initialize exception handlers across all supported arm processors. For ARM Cortex-A53, Cortex-R5, and Cortex-A9, the exception handlers are being initialized statically and this function does not do anything. However, it is still present to take care of backward compatibility issues (in earlier versions of BSPs, this API was being used to initialize exception handlers).

Parameters

None.	

Returns

None.

Note

None.

void Xil_DataAbortHandler (void * *CallBackRef*)

Default Data abort handler which prints data fault status register through which information about data fault can be acquired

Parameters

None	

Returns

None.

Note

None.

void Xil_PrefetchAbortHandler (void * *CallBackRef*)

Default Prefetch abort handler which prints prefetch fault status register through which information about instruction prefetch fault can be acquired

Parameters

None	

Returns

None.

Note

None.

void Xil_UndefinedExceptionHandler (void * *CallBackRef*)

Default undefined exception handler which prints address of the undefined instruction if debug prints are enabled

Parameters

None	

Returns

None.

Note

None.

XILINX
ALL PROGRAMMABLE™

Chapter 5

Cortex A9 Processor API

Overview

Standalone BSP contains boot code, cache, exception handling, file and memory management, configuration, time and processor-specific include functions. It supports gcc compilers.

Modules

Cortex A9 Processor Boot Code

Overview

The `boot.S` file contains a minimal set of code for transferring control from the processor reset location to the start of the application. The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Invalidate instruction cache, data cache and TLBs
3. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)
4. Configure MMU with short descriptor translation table format and program base address of translation table
5. Enable data cache, instruction cache and MMU

6. Enable Floating point unit
7. Transfer control to _start which clears BSS sections, initializes global timer and runs global constructor before jumping to main application

The `translation_table.S` file contains a static page table required by MMU for cortex-A9. This translation table is flat mapped (input address = output address) with default memory attributes defined for zynq architecture. It utilizes short descriptor translation table format with each section defining 1MB of memory. The overview of translation table memory attributes is described below.

	Memory Range	**Definition in Translation Table**	**Note**
DDR	0x00000000 - 0x3FFFFFFF	Normal write-back Cacheable	For a system where DDR is less than 1GB, region after DDR and before PL is marked as undefined/reserved in translation table
PL	0x40000000 - 0xBFFFFFFF	Strongly Ordered	
Reserved	0xC0000000 - 0xDFFFFFFF	Unassigned	
Memory mapped devices	0xE0000000 - 0xE02FFFFF	Device Memory	
Reserved	0xE0300000 - 0xE0FFFFFF	Unassigned	
NAND, NOR	0xE1000000 - 0xE3FFFFFF	Device memory	
SRAM	0xE4000000 - 0xE5FFFFFF	Normal write-back Cacheable	
Reserved	0xE6000000 - 0xF7FFFFFF	Unassigned	
AMBA APB Peripherals	0xF8000000 - 0xF8FFFFFF	Device Memory	0xF8000C00 - 0xF8000FFF, 0xF8010000 -0xF88FFFFF and 0xF8F03000 to 0xF8FFFFFF are reserved but due to granual size of 1MB, it is not possible to define separate regions for them

	Memory Range	Definition in Translation Table	Note
Reserved	0xF9000000 - 0xFBFFFFFF	Unassigned	
Linear QSPI - XIP	0xFC000000 - 0xFDFFFFFF	Normal write-through cacheable	
Reserved	0xFE000000 - 0xFFEFFFFF	Unassigned	
OCM	0xFFF00000 - 0xFFFFFFFF	Normal inner write-back cacheable	0xFFF00000 to 0xFFFB0000 is reserved but due to 1MB granual size, it is not possible to define separate region for it

Cortex A9 Processor Cache Functions

Overview

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Functions

- void Xil_DCacheEnable (void)
- void Xil_DCacheDisable (void)
- void Xil_DCacheInvalidate (void)
- void Xil_DCacheInvalidateRange (INTPTR adr, u32 len)
- void Xil_DCacheFlush (void)
- void Xil_DCacheFlushRange (INTPTR adr, u32 len)
- void Xil_ICacheEnable (void)
- void Xil_ICacheDisable (void)
- void Xil_ICacheInvalidate (void)
- void Xil_ICacheInvalidateRange (INTPTR adr, u32 len)
- void Xil_DCacheInvalidateLine (u32 adr)
- void Xil_DCacheFlushLine (u32 adr)
- void Xil_DCacheStoreLine (u32 adr)
- void Xil_ICacheInvalidateLine (u32 adr)
- void Xil_L1DCacheEnable (void)
- void Xil_L1DCacheDisable (void)

- void Xil_L1DCacheInvalidate (void)
- void Xil_L1DCacheInvalidateLine (u32 adr)
- void Xil_L1DCacheInvalidateRange (u32 adr, u32 len)
- void Xil_L1DCacheFlush (void)
- void Xil_L1DCacheFlushLine (u32 adr)
- void Xil_L1DCacheFlushRange (u32 adr, u32 len)
- void Xil_L1DCacheStoreLine (u32 adr)
- void Xil_L1ICacheEnable (void)
- void Xil_L1ICacheDisable (void)
- void Xil_L1ICacheInvalidate (void)
- void Xil_L1ICacheInvalidateLine (u32 adr)
- void Xil_L1ICacheInvalidateRange (u32 adr, u32 len)
- void Xil_L2CacheEnable (void)
- void Xil_L2CacheDisable (void)
- void Xil_L2CacheInvalidate (void)
- void Xil_L2CacheInvalidateLine (u32 adr)
- void Xil_L2CacheInvalidateRange (u32 adr, u32 len)
- void Xil_L2CacheFlush (void)
- void Xil_L2CacheFlushLine (u32 adr)
- void Xil_L2CacheFlushRange (u32 adr, u32 len)
- void Xil_L2CacheStoreLine (u32 adr)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheDisable (void)

Disable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheInvalidate (void)

Invalidate the entire Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheInvalidateRange (INTPTR *adr,* u32 *len*)

Invalidate the Data cache for the given address range. If the bytes specified by the address range are cached by the Data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and NOT written to the system memory before the lines are invalidated.
In this function, if start address or end address is not aligned to cache-line, particular cache-line containing unaligned start or end address is flush first and then invalidated the others as invalidating the same unaligned cache line may result into loss of data. This issue raises few possibilities.
If the address to be invalidated is not cache-line aligned, the following choices are available:

1. Invalidate the cache line when required and do not bother much for the side effects. Though it sounds good, it can result in hard-to-debug issues. The problem is, if some other variable are allocated in the same cache line and had been recently updated (in cache), the invalidation would result in loss of data.
2. Flush the cache line first. This will ensure that if any other variable present in the same cache line and updated recently are flushed out to memory. Then it can safely be invalidated. Again it sounds good, but this can result in issues. For example, when the invalidation happens in a typical ISR (after a DMA transfer has updated the memory), then flushing the cache line means, loosing data that were updated recently before the ISR got invoked.

Linux prefers the second one. To have uniform implementation (across standalone and Linux), the second option is implemented. This being the case, follwoing needs to be taken care of:

1. Whenever possible, the addresses must be cache line aligned. Please nore that, not just start address, even the end address must be cache line aligned. If that is taken care of, this will always work.
2. Avoid situations where invalidation has to be done after the data is updated by peripheral/DMA directly into the memory. It is not tough to achieve (may be a bit risky). The common use case to do invalidation is when a DMA happens. Generally for such use cases, buffers can be allocated first and then start the DMA. The practice that needs to be followed here is, immediately after buffer allocation and before starting the DMA, do the invalidation. With this approach, invalidation need not to be done after the DMA transfer is over.

This is going to always work if done carefully. However, the concern is, there is no guarantee that invalidate has not needed to be done after DMA is complete. For example, because of some reasons if the first cache line or last cache line (assuming the buffer in question comprises of multiple cache lines) are brought into cache (between the time it is invalidated and DMA completes) because of some speculative prefetching or reading data for a variable present in the same cache line, then we will have to invalidate the cache after DMA is complete.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_DCacheFlush (void)

Flush the entire Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheFlushRange (INTPTR *adr*, u32 *len*)

Flush the Data cache for the given address range. If the bytes specified by the address range are cached by the data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to the system memory before the lines are invalidated.

Parameters

adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes.

Returns

None.

Note

None.

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidateRange (INTPTR *adr,* u32 *len*)

Invalidate the instruction cache for the given address range. If the instructions specified by the address range are cached by the instrunction cache, the cachelines containing those instructions are invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_DCacheInvalidateLine (u32 *adr*)

Invalidate a Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to the system memory before the line is invalidated.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_DCacheFlushLine (u32 *adr*)

Flush a Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_DCacheStoreLine (u32 *adr*)

Store a Data cache line. If the byte specified by the address (adr) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Parameters

adr	32bit address of the data to be stored.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_ICacheInvalidateLine (u32 *adr*)

Invalidate an instruction cache line. If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Parameters

adr	32bit address of the instruction to be invalidated.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_L1DCacheEnable (void)

Enable the level 1 Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L1DCacheDisable (void)

Disable the level 1 Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L1DCacheInvalidate (void)

Invalidate the level 1 Data cache.

Parameters

None.	

Returns

None.

Note

In Cortex A9, there is no cp instruction for invalidating the whole D-cache. This function invalidates each line by set/way.

void Xil_L1DCacheInvalidateLine (u32 *adr*)

Invalidate a level 1 Data cache line. If the byte specified by the address (Addr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data to be invalidated.

Returns

None.

Note

The bottom 5 bits are set to 0, forced by architecture.

void Xil_L1DCacheInvalidateRange (u32 *adr,* u32 *len*)

Invalidate the level 1 Data cache for the given address range. If the bytes specified by the address range are cached by the Data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and NOT written to the system memory before the lines are invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_L1DCacheFlush (void)

Flush the level 1 Data cache.

Parameters

None.	

Returns

None.

Note

In Cortex A9, there is no cp instruction for flushing the whole D-cache. Need to flush each line.

void Xil_L1DCacheFlushLine (u32 *adr*)

Flush a level 1 Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

The bottom 5 bits are set to 0, forced by architecture.

void Xil_L1DCacheFlushRange (u32 *adr,* u32 *len*)

Flush the level 1 Data cache for the given address range. If the bytes specified by the address range are cached by the Data cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to system memory before the lines are invalidated.

Parameters

adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes.

Returns

None.

Note

None.

void Xil_L1DCacheStoreLine (u32 *adr*)

Store a level 1 Data cache line. If the byte specified by the address (adr) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Parameters

Address	to be stored.

Returns

None.

Note

The bottom 5 bits are set to 0, forced by architecture.

void Xil_L1ICacheEnable (void)

Enable the level 1 instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L1ICacheDisable (void)

Disable level 1 the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L1ICacheInvalidate (void)

Invalidate the entire level 1 instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L1ICacheInvalidateLine (u32 *adr*)

Invalidate a level 1 instruction cache line. If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Parameters

adr	32bit address of the instruction to be invalidated.

Returns

None.

Note

The bottom 5 bits are set to 0, forced by architecture.

void Xil_L1ICacheInvalidateRange (u32 *adr,* u32 *len*)

Invalidate the level 1 instruction cache for the given address range. If the instrucions specified by the address range are cached by the instruction cache, the cacheline containing those bytes are invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_L2CacheEnable (void)

Enable the L2 cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L2CacheDisable (void)

Disable the L2 cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L2CacheInvalidate (void)

Invalidate the entire level 2 cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L2CacheInvalidateLine (u32 *adr*)

Invalidate a level 2 cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data/instruction to be invalidated.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_L2CacheInvalidateRange (u32 *adr,* u32 *len*)

Invalidate the level 2 cache for the given address range. If the bytes specified by the address range are cached by the L2 cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and are NOT written to system memory before the lines are invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_L2CacheFlush (void)

Flush the entire level 2 cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_L2CacheFlushLine (u32 *adr*)

Flush a level 2 cache line. If the byte specified by the address (adr) is cached by the L2 cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data/instruction to be flushed.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_L2CacheFlushRange (u32 *adr,* u32 *len*)

Flush the level 2 cache for the given address range. If the bytes specified by the address range are cached by the L2 cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to the system memory before the lines are invalidated.

Parameters

adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes.

Returns

None.

Note

None.

void Xil_L2CacheStoreLine (u32 *adr*)

Store a level 2 cache line. If the byte specified by the address (adr) is cached by the L2 cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Parameters

adr	32bit address of the data/instruction to be stored.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

Cortex A9 Processor MMU Functions

Overview

MMU functions equip users to enable MMU, disable MMU and modify default memory attributes of MMU table as per the need.

Functions

- void Xil_SetTlbAttributes (INTPTR Addr, u32 attrib)
- void Xil_EnableMMU (void)
- void Xil_DisableMMU (void)

Function Documentation

void Xil_SetTlbAttributes (INTPTR *Addr*, u32 *attrib*)

This function sets the memory attributes for a section covering 1MB of memory in the translation table.

Parameters

Addr	32-bit address for which memory attributes need to be set.
attrib	Attribute for the given memory region. xil_mmu.h contains definitions of commonly used memory attributes which can be utilized for this function.

Returns

None.

Note

The MMU or D-cache does not need to be disabled before changing a translation table entry.

void Xil_EnableMMU (void)

Enable MMU for cortex A9 processor. This function invalidates the instruction and data caches, and then enables MMU.

Parameters

None.	

Returns

None.

void Xil_DisableMMU (void)

Disable MMU for Cortex A9 processors. This function invalidates the TLBs, Branch Predictor Array and flushed the D Caches before disabling the MMU.

Parameters

None.	

Returns

None.

Note

When the MMU is disabled, all the memory accesses are treated as strongly ordered.

Cortex A9 Time Functions

Overview

xtime_l.h provides access to the 64-bit Global Counter in the PMU. This counter increases by one at every two processor cycles. These functions can be used to get/set time in the global timer.

Functions

- void XTime_SetTime (XTime Xtime_Global)
- void XTime_GetTime (XTime *Xtime_Global)

Function Documentation

void XTime_SetTime (XTime *Xtime_Global*)

Set the time in the Global Timer Counter Register.

Parameters

Xtime_Global	64-bit Value to be written to the Global Timer Counter Register.

Returns

None.

Note

When this function is called by any one processor in a multi- processor environment, reference time will reset/lost for all processors.

void XTime_GetTime (XTime * *Xtime_Global*)

Get the time from the Global Timer Counter Register.

Parameters

Xtime_Global	Pointer to the 64-bit location which will be updated with the current timer value.

Returns

None.

Note

None.

Cortex A9 Event Counter Function

Overview

Cortex A9 event counter functions can be utilized to configure and control the Cortex-A9 performance monitor events.
Cortex-A9 performance monitor has six event counters which can be used to count a variety of events described in Coretx-A9 TRM. xpm_counter.h defines configurations XPM_CNTRCFGx which can be used to program the event counters to count a set of events.

Note

It doesn't handle the Cortex-A9 cycle counter, as the cycle counter is being used for time keeping.

Functions

- void Xpm_SetEvents (s32 PmcrCfg)
- void Xpm_GetEventCounters (u32 *PmCtrValue)

Function Documentation

void Xpm_SetEvents (s32 *PmcrCfg*)

This function configures the Cortex A9 event counters controller, with the event codes, in a configuration selected by the user and enables the counters.

Parameters

PmcrCfg	Configuration value based on which the event counters are configured. XPM_CNTRCFG* values defined in xpm_counter.h can be utilized for setting configuration.

Returns

None.

Note

None.

void Xpm_GetEventCounters (u32 * *PmCtrValue*)

This function disables the event counters and returns the counter values.

Parameters

PmCtrValue	Pointer to an array of type u32 PmCtrValue[6]. It is an output parameter which is used to return the PM counter values.

Returns

None.

Note

None.

PL310 L2 Event Counters Functions

Overview

xl2cc_counter.h contains APIs for configuring and controlling the event counters in PL310 L2 cache controller. PL310 has two event counters which can be used to count variety of events like DRHIT, DRREQ, DWHIT, DWREQ, etc. xl2cc_counter.h contains definitions for different configurations which can be used for the event counters to count a set of events.

Functions

- void XL2cc_EventCtrInit (s32 Event0, s32 Event1)
- void XL2cc_EventCtrStart (void)
- void XL2cc_EventCtrStop (u32 *EveCtr0, u32 *EveCtr1)

Function Documentation

void XL2cc_EventCtrInit (s32 *Event0*, s32 *Event1*)

This function initializes the event counters in L2 Cache controller with a set of event codes specified by the user.

Parameters

Event0	Event code for counter 0.
Event1	Event code for counter 1.

Returns

None.

Note

The definitions for event codes XL2CC_∗ can be found in xl2cc_counter.h.

void XL2cc_EventCtrStart (void)

This function starts the event counters in L2 Cache controller.

Parameters

None.	

Returns

None.

Note

None.

void XL2cc_EventCtrStop (u32 ∗ *EveCtr0*, u32 ∗ *EveCtr1*)

This function disables the event counters in L2 Cache controller, saves the counter values and resets the counters.

Parameters

EveCtr0	Output parameter which is used to return the value in event counter 0. EveCtr1: Output parameter which is used to return the value in event counter 1.

Returns

None.

Note

None.

Cortex A9 Processor and pl310 Errata Support

Overview

Various ARM errata are handled in the standalone BSP. The implementation for errata handling follows ARM guidelines and is based on the open source Linux support for these errata.

Note

The errata handling is enabled by default. To disable handling of all the errata globally, un-define the macro ENABLE_ARM_ERRATA in xil_errata.h. To disable errata on a per-erratum basis, un-define relevant macros in xil_errata.h.

errata_definitions

The errata conditions handled in the standalone BSP are listed below

- #define **ENABLE_ARM_ERRATA**
- #define CONFIG_ARM_ERRATA_742230
- #define CONFIG_ARM_ERRATA_743622
- #define CONFIG_ARM_ERRATA_775420
- #define CONFIG_ARM_ERRATA_794073
- #define CONFIG_PL310_ERRATA_588369
- #define CONFIG_PL310_ERRATA_727915
- #define CONFIG_PL310_ERRATA_753970

Macro Definition Documentation

#define CONFIG_ARM_ERRATA_742230

Errata No: 742230 Description: DMB operation may be faulty

#define CONFIG_ARM_ERRATA_743622

Errata No: 743622 Description: Faulty hazard checking in the Store Buffer may lead to data corruption.

#define CONFIG_ARM_ERRATA_775420

Errata No: 775420 Description: A data cache maintenance operation which aborts, might lead to deadlock

#define CONFIG_ARM_ERRATA_794073

Errata No: 794073 Description: Speculative instruction fetches with MMU disabled might not comply with architectural requirements

#define CONFIG_PL310_ERRATA_588369

PL310 L2 Cache Errata Errata No: 588369 Description: Clean & Invalidate maintenance operations do not invalidate clean lines

#define CONFIG_PL310_ERRATA_727915

Errata No: 727915 Description: Background Clean and Invalidate by Way operation can cause data corruption

#define CONFIG_PL310_ERRATA_753970

Errata No: 753970 Description: Cache sync operation may be faulty

Cortex A9 Processor Specific Include Files

The xpseudo_asm.h includes xreg_cortexa9.h and xpseudo_asm_gcc.h.
The xreg_cortexa9.h file contains definitions for inline assembler code. It provides inline definitions for Cortex A9 GPRs, SPRs, MPE registers, co-processor registers and Debug registers.
The xpseudo_asm_gcc.h contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation etc. These inline assembler instructions can be used from drivers and user applications written in C.

Chapter 6

Cortex A53 32-bit Processor API

Overview

Cortex-A53 standalone BSP contains two separate BSPs for 32-bit mode and 64-bit mode. The 32-bit mode of cortex-A53 is compatible with ARMv7-A architecture.

Modules

- Cortex A53 32-bit Processor Boot Code
- Cortex A53 32-bit Processor Cache Functions
- Cortex A53 32-bit Processor MMU Handling
- Cortex A53 32-bit Mode Time Functions
- Cortex A53 32-bit Processor Specific Include Files

Cortex A53 32-bit Processor Boot Code

Overview

The `boot.S` file contains a minimal set of code for transferring control from the processor reset location to the start of the application. The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Invalidate instruction cache, data cache and TLBs
3. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)
4. Program counter frequency
5. Configure MMU with short descriptor translation table format and program base address of translation table
6. Enable data cache, instruction cache and MMU

7. Transfer control to _start which clears BSS sections and runs global constructor before jumping to main application

The `translation_table.S` file contains a static page table required by MMU for cortex-A53. This translation table is flat mapped (input address = output address) with default memory attributes defined for zynq ultrascale+ architecture. It utilizes short descriptor translation table format with each section defining 1MB of memory. The overview of translation table memory attributes is described below.

	Memory Range	**Definition in Translation Table**	**Note**
DDR	0x00000000 - 0x7FFFFFFF	Normal write-back Cacheable	For a system where DDR is less than 2GB, region after DDR and before PL is marked as undefined/reserved in translation table
PL	0x80000000 - 0xBFFFFFFF	Strongly Ordered	
QSPI, lower PCIe	0xC0000000 - 0xEFFFFFFF	Device Memory	
Reserved	0xF0000000 - 0xF7FFFFFF	Unassigned	
STM Coresight	0xF8000000 - 0xF8FFFFFF	Device Memory	
GIC	0xF9000000 - 0xF90FFFFF	Device memory	
Reserved	0xF9100000 - 0xFCFFFFFF	Unassigned	
FPS, LPS slaves	0xFD000000 - 0xFFBFFFFF	Device memory	
CSU, PMU	0xFFC00000 - 0xFFDFFFFF	Device Memory	This region contains CSU and PMU memory which are marked as Device since it is less than 1MB and falls in a region with device memory
TCM, OCM	0xFFE00000 - 0xFFFFFFFF	Normal write-back cacheable	

Cortex A53 32-bit Processor Cache Functions

Overview

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Functions

- void Xil_DCacheEnable (void)
- void Xil_DCacheDisable (void)
- void Xil_DCacheInvalidate (void)
- void Xil_DCacheInvalidateRange (INTPTR adr, u32 len)
- void Xil_DCacheInvalidateLine (u32 adr)
- void Xil_DCacheFlush (void)
- void Xil_DCacheFlushLine (u32 adr)
- void Xil_ICacheEnable (void)
- void Xil_ICacheDisable (void)
- void Xil_ICacheInvalidate (void)
- void Xil_ICacheInvalidateRange (INTPTR adr, u32 len)
- void Xil_ICacheInvalidateLine (u32 adr)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheDisable (void)

Disable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheInvalidate (void)

Invalidate the Data cache. The contents present in the data cache are cleaned and invalidated.

Parameters

None.	

Returns

None.

Note

In Cortex-A53, functionality to simply invalide the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

void Xil_DCacheInvalidateRange (INTPTR *adr,* u32 *len*)

Invalidate the Data cache for the given address range. The cachelines present in the adderss range are cleaned and invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

In Cortex-A53, functionality to simply invalide the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

void Xil_DCacheInvalidateLine (u32 *adr*)

Invalidate a Data cache line. The cacheline is cleaned and invalidated.

Parameters

adr	32 bit address of the data to be invalidated.

Returns

None.

Note

In Cortex-A53, functionality to simply invalide the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

void Xil_DCacheFlush (void)

Flush the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheFlushLine (u32 *adr*)

Flush a Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data to be flushed.

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidateRange (INTPTR *adr,* u32 *len*)

Invalidate the instruction cache for the given address range. If the instructions specified by the address range are cached by the instrunction cache, the cachelines containing those instructions are invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_ICacheInvalidateLine (u32 *adr*)

Invalidate an instruction cache line. If the instruction specified by the address is cached by the instruction cache, the cachecline containing that instruction is invalidated.

Parameters

adr	32bit address of the instruction to be invalidated..

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

Cortex A53 32-bit Processor MMU Handling

Overview

MMU functions equip users to enable MMU, disable MMU and modify default memory attributes of MMU table as per the need.

Functions

- void Xil_SetTlbAttributes (INTPTR Addr, u32 attrib)
- void Xil_EnableMMU (void)
- void Xil_DisableMMU (void)

Function Documentation

void Xil_SetTlbAttributes (INTPTR *Addr,* u32 *attrib*)

This function sets the memory attributes for a section covering 1MB of memory in the translation table.

Parameters

Addr	32-bit address for which the attributes need to be set.
attrib	Attributes for the specified memory region. xil_mmu.h contains commonly used memory attributes definitions which can be utilized for this function.

Returns

None.

Note

The MMU or D-cache does not need to be disabled before changing a translation table entry.

void Xil_EnableMMU (void)

Enable MMU for Cortex-A53 processor in 32bit mode. This function invalidates the instruction and data caches before enabling MMU.

Parameters

None.	

Returns

None.

void Xil_DisableMMU (void)

Disable MMU for Cortex A53 processors in 32bit mode. This function invalidates the TLBs, Branch Predictor Array and flushed the data cache before disabling the MMU.

Parameters

None.	

Returns

None.

Note

When the MMU is disabled, all the memory accesses are treated as strongly ordered.

Cortex A53 32-bit Mode Time Functions

Overview

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 64-bit generic counter in Cortex-A53. The `sleep.c`, `usleep.c` file and the corresponding `sleep.h` include file implement sleep functions. Sleep functions are implemented as busy loops.

Functions

- void XTime_StartTimer (void)
- void XTime_SetTime (XTime Xtime_Global)
- void XTime_GetTime (XTime *Xtime_Global)

Function Documentation

void XTime_StartTimer (void)

Start the 64-bit physical timer counter.

Parameters

None.	

Returns

None.

Note

The timer is initialized only if it is disabled. If the timer is already running this function does not perform any operation.

void XTime_SetTime (XTime *Xtime_Global*)

Timer of A53 runs continuously and the time can not be set as desired. This API doesn't contain anything. It is defined to have uniformity across platforms.

Parameters

Xtime_Global	64bit Value to be written to the Global Timer Counter Register.

Returns

None.

Note

None.

void XTime_GetTime (XTime * *Xtime_Global*)

Get the time from the physical timer counter register.

Parameters

Xtime_Global	Pointer to the 64-bit location to be updated with the current value in physical timer counter.

Returns

None.

Note

None.

Cortex A53 32-bit Processor Specific Include Files

The `xreg_cortexa53.h` file contains definitions for inline assembler code. It provides inline definitions for Cortex A53 GPRs, SPRs and floating point registers.
The `xpseudo_asm_gcc.h` contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation. These inline assembler instructions can be used from drivers and user applications written in C.

XILINX
ALL PROGRAMMABLE™

Chapter 7

Cortex A53 64-bit Processor API

Overview

Cortex-A53 standalone BSP contains two separate BSPs for 32-bit mode and 64-bit mode. The 64-bit mode of cortex-A53 contains ARMv8-A architecture. This section provides a linked summary and detailed descriptions of the Cortex A53 64-bit Processor APIs.

Modules

Cortex A53 64-bit Processor Boot Code

Overview

The `boot.S` file contains a minimal set of code for transferring control from the processor reset location to the start of the application. The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Cortex-A53 starts execution from EL3 and currently application is also run from EL3. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Set reset vector table base address
3. Program stack pointer for EL3
4. Routing of interrupts to EL3
5. Enable ECC protection
6. Program generic counter frequency

7. Invalidate instruction cache, data cache and TLBs
8. Configure MMU registers and program base address of translation table
9. Transfer control to _start which clears BSS sections and runs global constructor before jumping to main application

Cortex A53 64-bit Processor Cache Functions

Overview

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Functions

- void Xil_DCacheEnable (void)
- void Xil_DCacheDisable (void)
- void Xil_DCacheInvalidate (void)
- void Xil_DCacheInvalidateRange (INTPTR adr, INTPTR len)
- void Xil_DCacheInvalidateLine (INTPTR adr)
- void Xil_DCacheFlush (void)
- void Xil_DCacheFlushLine (INTPTR adr)
- void Xil_ICacheEnable (void)
- void Xil_ICacheDisable (void)
- void Xil_ICacheInvalidate (void)
- void Xil_ICacheInvalidateRange (INTPTR adr, INTPTR len)
- void Xil_ICacheInvalidateLine (INTPTR adr)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheDisable (void)

Disable the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheInvalidate (void)

Invalidate the Data cache. The contents present in the cache are cleaned and invalidated.

Parameters

None.	

Returns

None.

Note

In Cortex-A53, functionality to simply invalide the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

void Xil_DCacheInvalidateRange (INTPTR *adr,* INTPTR *len*)

Invalidate the Data cache for the given address range. The cachelines present in the adderss range are cleaned and invalidated.

Parameters

adr	64bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

In Cortex-A53, functionality to simply invalide the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

void Xil_DCacheInvalidateLine (INTPTR *adr*)

Invalidate a Data cache line. The cacheline is cleaned and invalidated.

Parameters

adr	64bit address of the data to be flushed.

Returns

None.

Note

In Cortex-A53, functionality to simply invalide the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

void Xil_DCacheFlush (void)

Flush the Data cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_DCacheFlushLine (INTPTR *adr*)

Flush a Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

adr	64bit address of the data to be flushed.

Returns

None.

Note

The bottom 6 bits are set to 0, forced by architecture.

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	

Returns

None.

Note

None.

void Xil_ICacheInvalidateRange (INTPTR *adr,* INTPTR *len*)

Invalidate the instruction cache for the given address range. If the instructions specified by the address range are cached by the instrunction cache, the cachelines containing those instructions are invalidated.

Parameters

adr	64bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_ICacheInvalidateLine (INTPTR *adr*)

Invalidate an instruction cache line. If the instruction specified by the parameter adr is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Parameters

adr	64bit address of the instruction to be invalidated.

Returns

None.

Note

The bottom 6 bits are set to 0, forced by architecture.

Cortex A53 64-bit Processor MMU Handling

Overview

MMU function equip users to modify default memory attributes of MMU table as per the need.

Functions

- void Xil_SetTlbAttributes (INTPTR Addr, u64 attrib)

Function Documentation

void Xil_SetTlbAttributes (INTPTR *Addr,* u64 *attrib*)

brief It sets the memory attributes for a section, in the translation table. If the address (defined by Addr) is less than 4GB, the memory attribute(attrib) is set for a section of 2MB memory. If the address (defined by Addr) is greater than 4GB, the memory attribute (attrib) is set for a section of 1GB memory.

Parameters

Addr	64-bit address for which attributes are to be set.
attrib	Attribute for the specified memory region. xil_mmu.h contains commonly used memory attributes definitions which can be utilized for this function.

Returns

None.

Note

The MMU and D-cache need not be disabled before changing an translation table attribute.

Cortex A53 64-bit Mode Time Functions

Overview

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 64-bit generic counter in Cortex-A53. The `sleep.c`, `usleep.c` file and the corresponding `sleep.h` include file implement sleep functions. Sleep functions are implemented as busy loops.

Functions

- void XTime_StartTimer (void)
- void XTime_SetTime (XTime Xtime_Global)
- void XTime_GetTime (XTime *Xtime_Global)

Function Documentation

void XTime_StartTimer (void)

Start the 64-bit physical timer counter.

Parameters

None.	

Returns

None.

Note

The timer is initialized only if it is disabled. If the timer is already running this function does not perform any operation.

void XTime_SetTime (XTime *Xtime_Global*)

Timer of A53 runs continuously and the time can not be set as desired. This API doesn't contain anything. It is defined to have uniformity across platforms.

Parameters

Xtime_Global	64bit value to be written to the physical timer counter register.

Returns

None.

Note

None.

void XTime_GetTime (XTime * *Xtime_Global*)

Get the time from the physical timer counter register.

Parameters

Xtime_Global	Pointer to the 64-bit location to be updated with the current value of physical timer counter register.

Returns

None.

Note

None.

Cortex A53 64-bit Processor Specific Include Files

The `xreg_cortexa53.h` file contains definitions for inline assembler code. It provides inline definitions for Cortex A53 GPRs, SPRs and floating point registers.
The `xpseudo_asm_gcc.h` contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation. These inline assembler instructions can be used from drivers and user applications written in C.

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support .

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Please Read: Important Legal Notices

Automotive Applications Disclaimer

AUTOMOTIVE PRODUCTS (IDENTIFIED AS ”XA” IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE (”SAFETY APPLICATION”) UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD (”SAFETY DESIGN”). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.