Xilinx Standalone Library Documentation

Standalone Library v7.2

UG647 (v2020.1) June 3, 2020

Table of Contents

Chapter 1

Xilinx Hardware Abstraction Layer API

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

Assert APIs and Macros

The xil_assert.h file contains assert related functions and macros. Assert APIs/Macros specifies that a application program satisfies certain conditions at particular points in its execution. These function can be used by application programs to ensure that, application code is satisfying certain conditions.

Table 1: **Quick Function Reference**

Type	Name	Arguments
void	Xil_Assert	file line
void	XNullHandler	void * NullParameter
void	Xil_AssertSetCallback	routine

Functions

Xil_Assert

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.

Note: None.

Prototype

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

Parameters

The following table lists the `Xil_Assert` function arguments.

Table 2: **Xil_Assert Arguments**

Name	Description
file	filename of the source
line	linenumber within File

Returns

None.

XNullHandler

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.

***Note*:** None.

Prototype

```
void XNullHandler(void *NullParameter);
```

Parameters

The following table lists the `XNullHandler` function arguments.

Table 3: **XNullHandler Arguments**

Name	Description
NullParameter	arbitrary void pointer and not used.

Returns

None.

Xil_AssertSetCallback

Set up a callback function to be invoked when an assert occurs.

If a callback is already installed, then it will be replaced.

Note: This function has no effect if NDEBUG is set

Prototype

```
void Xil_AssertSetCallback(Xil_AssertCallback Routine);
```

Parameters

The following table lists the `Xil_AssertSetCallback` function arguments.

Table 4: **Xil_AssertSetCallback Arguments**

Name	Description
routine	callback to be invoked when an assert is taken

Returns

None.

Register IO interfacing APIs

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

Table 5: **Quick Function Reference**

Type	Name	Arguments
u16	Xil_EndianSwap16	u16 Data
u32	Xil_EndianSwap32	u32 Data
INLINE u8	Xil_In8	UINTPTR Addr
INLINE u16	Xil_In16	UINTPTR Addr
INLINE u32	Xil_In32	UINTPTR Addr
INLINE u64	Xil_In64	UINTPTR Addr

Table 5: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
INLINE void	Xil_Out8	UINTPTR Addr u8 Value
INLINE void	Xil_Out16	UINTPTR Addr u16 Value
INLINE void	Xil_Out32	UINTPTR Addr u32 Value
INLINE void	Xil_Out64	UINTPTR Addr u64 Value
INLINE u32	Xil_SecureOut32	UINTPTR Addr u32 Value
INLINE u16	Xil_In16BE	void
INLINE u32	Xil_In32BE	void
INLINE void	Xil_Out16BE	void
INLINE void	Xil_Out32BE	void

Functions

Xil_EndianSwap16

Perform a 16-bit endian conversion.

Prototype

```
u16 Xil_EndianSwap16(u16 Data);
```

Parameters

The following table lists the `Xil_EndianSwap16` function arguments.

Table 6: **Xil_EndianSwap16 Arguments**

Name	Description
Data	16 bit value to be converted

Returns

16 bit Data with converted endianness

Xil_EndianSwap32

Perform a 32-bit endian conversion.

Prototype

```
u32 Xil_EndianSwap32(u32 Data);
```

Parameters

The following table lists the `Xil_EndianSwap32` function arguments.

Table 7: **Xil_EndianSwap32 Arguments**

Name	Description
Data	32 bit value to be converted

Returns

32 bit data with converted endianness

Xil_In8

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

Prototype

```
INLINE u8 Xil_In8(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In8` function arguments.

Table 8: **Xil_In8 Arguments**

Name	Description
Addr	contains the address to perform the input operation

Returns

The 8 bit Value read from the specified input address.

Xil_In16

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

Prototype

```
INLINE u16 Xil_In16(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In16` function arguments.

Table 9: **Xil_In16 Arguments**

Name	Description
Addr	contains the address to perform the input operation

Returns

The 16 bit Value read from the specified input address.

Xil_In32

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

Prototype

```
INLINE u32 Xil_In32(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In32` function arguments.

Table 10: **Xil_In32 Arguments**

Name	Description
Addr	contains the address to perform the input operation

Returns

The 32 bit Value read from the specified input address.

Xil_In64

Performs an input operation for a memory location by reading the 64 bit Value read from that address.

Prototype

```
INLINE u64 Xil_In64(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In64` function arguments.

Table 11: **Xil_In64 Arguments**

Name	Description
Addr	contains the address to perform the input operation

Returns

The 64 bit Value read from the specified input address.

Xil_Out8

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

Prototype

```
INLINE void Xil_Out8(UINTPTR Addr, u8 Value);
```

Parameters

The following table lists the `Xil_Out8` function arguments.

Table 12: **Xil_Out8 Arguments**

Name	Description
Addr	contains the address to perform the output operation
Value	contains the 8 bit Value to be written at the specified address.

Returns

None.

Xil_Out16

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

Prototype

```
INLINE void Xil_Out16(UINTPTR Addr, u16 Value);
```

Parameters

The following table lists the `Xil_Out16` function arguments.

Table 13: **Xil_Out16 Arguments**

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

Returns

None.

Xil_Out32

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

Prototype

```
INLINE void Xil_Out32(UINTPTR Addr, u32 Value);
```

Parameters

The following table lists the `Xil_Out32` function arguments.

Table 14: **Xil_Out32 Arguments**

Name	Description
Addr	contains the address to perform the output operation
Value	contains the 32 bit Value to be written at the specified address.

Returns

None.

Xil_Out64

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

Prototype

```
INLINE void Xil_Out64(UINTPTR Addr, u64 Value);
```

Parameters

The following table lists the `Xil_Out64` function arguments.

Table 15: **Xil_Out64 Arguments**

Name	Description
Addr	contains the address to perform the output operation
Value	contains 64 bit Value to be written at the specified address.

Returns

None.

Xil_SecureOut32

Performs an output operation for a memory location by writing the 32 bit Value to the the specified address and then reading it back to verify the value written in the register.

Prototype

```
INLINE u32 Xil_SecureOut32(UINTPTR Addr, u32 Value);
```

Parameters

The following table lists the `Xil_SecureOut32` function arguments.

Table 16: **Xil_SecureOut32 Arguments**

Name	Description
Addr	contains the address to perform the output operation
Value	contains 32 bit Value to be written at the specified address

Returns

Returns Status

- XST_SUCCESS on success
- XST_FAILURE on failure

Definitions for available xilinx platforms

The xplatform_info.h file contains definitions for various available Xilinx platforms. Also, it contains prototype of APIs, which can be used to get the platform information.

Table 17: **Quick Function Reference**

Type	Name	Arguments
u32	XGetPlatform_Info	None.

Functions

XGetPlatform_Info

This API is used to provide information about platform.

Prototype

```
u32 XGetPlatform_Info(void);
```

Parameters

The following table lists the `XGetPlatform_Info` function arguments.

Table 18: **XGetPlatform_Info Arguments**

Name	Description
None.	

Returns

The information about platform defined in xplatform_info.h

Data types for Xilinx Software IP Cores

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

Customized APIs for Memory Operations

The xil_mem.h file contains prototype for functions related to memory operations. These APIs are applicable for all processors supported by Xilinx.

Table 19: **Quick Function Reference**

Type	Name	Arguments
void	Xil_MemCpy	void * dst const void * src u32 cnt

Functions

Xil_MemCpy

This function copies memory from once location to other.

Prototype

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

Parameters

The following table lists the `Xil_MemCpy` function arguments.

Table 20: **Xil_MemCpy Arguments**

Name	Description
dst	pointer pointing to destination memory

Table 20: **Xil_MemCpy Arguments** *(cont'd)*

Name	Description
src	pointer pointing to source memory
cnt	32 bit length of bytes to be copied

Xilinx software status codes

The xstatus.h file contains the Xilinx software status codes.These codes are used throughout the Xilinx device drivers.

Test Utilities for Memory and Caches

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.

The xil_testcache.h file contains utility functions to test cache.

The xil_testio.h file contains utility functions to test 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.

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 list describes the supported memory tests:

- XIL_TESTMEM_ALLMEMTESTS: This test runs all of the subtests.
- XIL_TESTMEM_INCREMENT: This test starts at 'XIL_TESTMEM_INIT_VALUE' and uses the incrementing value as the test value for memory.
- XIL_TESTMEM_WALKONES: Also known as the Walking ones test. This test uses a walking '1' as the test value for memory.

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

- XIL_TESTMEM_WALKZEROS: Also known as the 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: Also known as the 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: Also known as the 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".

CAUTION! *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.

Table 21: **Quick Function Reference**

Type	Name	Arguments
s32	Xil_TestDCacheRange	void
s32	Xil_TestDCacheAll	void
s32	Xil_TestICacheRange	void
s32	Xil_TestICacheAll	void
s32	Xil_TestIO8	u8 * Addr s32 Length u8 Value
s32	Xil_TestIO16	u16 * Addr s32 Length u16 Value s32 Kind s32 Swap

Table 21: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
s32	Xil_TestIO32	u32 * Addr s32 Length u32 Value s32 Kind s32 Swap

Functions

Xil_TestIO8

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.

Prototype

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

Parameters

The following table lists the `Xil_TestIO8` function arguments.

Table 22: **Xil_TestIO8 Arguments**

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

Returns

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

Xil_TestIO16

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.

Prototype

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

Parameters

The following table lists the `Xil_TestIO16` function arguments.

Table 23: **Xil_TestIO16 Arguments**

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

Xil_TestIO32

Perform a destructive 32-bit wide register IO test.

Each location is tested by sequentially writing a 32-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 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.

Prototype

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

Parameters

The following table lists the `Xil_TestIO32` function arguments.

Table 24: **Xil_TestIO32 Arguments**

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

MicroBlaze Processor API

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

MicroBlaze Pseudo-asm Macros and Interrupt Handling APIs

MicroBlaze BSP 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

Table 25: **Quick Function Reference**

Type	Name	Arguments
void	microblaze_enable_interrupts	void
void	microblaze_disable_interrupts	void
void	microblaze_enable_icache	void
void	microblaze_disable_icache	void
void	microblaze_enable_dcache	void
void	microblaze_disable_dcache	void
void	microblaze_enable_exceptions	void
void	microblaze_disable_exceptions	void

Table 25: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	microblaze_register_handler	`XInterruptHandler` Handler void * DataPtr
void	microblaze_register_exception_handler	u32 ExceptionId Top void * DataPtr
void	microblaze_invalidate_icache	void
void	microblaze_invalidate_dcache	void
void	microblaze_flush_dcache	void
void	microblaze_invalidate_icache_range	void
void	microblaze_invalidate_dcache_range	void
void	microblaze_flush_dcache_range	void
void	microblaze_scrub	void
void	microblaze_invalidate_cache_ext	void
void	microblaze_flush_cache_ext	void
void	microblaze_flush_cache_ext_range	void
void	microblaze_invalidate_cache_ext_range	void
void	microblaze_update_icache	void
void	microblaze_init_icache_range	void
void	microblaze_update_dcache	void
void	microblaze_init_dcache_range	void

Functions

microblaze_register_handler

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.

Prototype

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

Parameters

The following table lists the `microblaze_register_handler` function arguments.

Table 26: **microblaze_register_handler Arguments**

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

Returns

None.

microblaze_register_exception_handler

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.

None.

***Note*:**

Prototype

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

Parameters

The following table lists the `microblaze_register_exception_handler` function arguments.

Table 27: **microblaze_register_exception_handler Arguments**

Name	Description
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.

MicroBlaze exception APIs

The xil_exception.h file contains MicroBlaze specific exception related APIs and macros. Application programs can use these APIs/Macros for various exception related operations (i.e. enable exception, disable exception, register exception handler etc.)

***Note*:** To use exception related functions, the xil_exception.h file must be added in source code

Table 28: **Quick Function Reference**

Type	Name	Arguments
void	microblaze_enable_exceptions	void
void	microblaze_disable_exceptions	void
void	microblaze_enable_interrupts	void
void	microblaze_disable_interrupts	void
void	Xil_ExceptionNullHandler	void * Data
void	Xil_ExceptionInit	None.
void	Xil_ExceptionEnable	void
void	Xil_ExceptionDisable	None.
void	Xil_ExceptionRegisterHandler	u32 Id `Xil_ExceptionHandler` Handler void * Data

Table 28: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_ExceptionRemoveHandler	u32 Id

Functions

Xil_ExceptionNullHandler

This function is a stub handler that is the default handler that gets called if the application has not setup a handler for a specific exception.

The function interface has to match the interface specified for a handler even though none of the arguments are used.

Prototype

```
void Xil_ExceptionNullHandler(void *Data);
```

Parameters

The following table lists the `Xil_ExceptionNullHandler` function arguments.

Table 29: **Xil_ExceptionNullHandler Arguments**

Name	Description
Data	unused by this function.

Xil_ExceptionInit

Initialize exception handling for the processor.

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

Prototype

```
void Xil_ExceptionInit(void);
```

Parameters

The following table lists the `Xil_ExceptionInit` function arguments.

Table 30: **Xil_ExceptionInit Arguments**

Name	Description
None.	

Returns

None.

Xil_ExceptionEnable

Enable Exceptions.

Prototype

```
void Xil_ExceptionEnable(void);
```

Returns

None.

Xil_ExceptionDisable

Disable Exceptions.

Prototype

```
void Xil_ExceptionDisable(void);
```

Parameters

The following table lists the `Xil_ExceptionDisable` function arguments.

Table 31: **Xil_ExceptionDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_ExceptionRegisterHandler

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.

Prototype

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

Parameters

The following table lists the `Xil_ExceptionRegisterHandler` function arguments.

Table 32: **Xil_ExceptionRegisterHandler Arguments**

Name	Description
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.

Xil_ExceptionRemoveHandler

Removes the handler for a specific exception Id.

The stub handler is then registered for this exception Id.

Prototype

```
void Xil_ExceptionRemoveHandler(u32 Id);
```

Parameters

The following table lists the `Xil_ExceptionRemoveHandler` function arguments.

Table 33: **Xil_ExceptionRemoveHandler Arguments**

Name	Description
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 FSL Macros

MicroBlaze BSP 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

MicroBlaze PVR access routines and macros

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.

Table 34: **Quick Function Reference**

Type	Name	Arguments
int	microblaze_get_pvr	pvr-

Functions

microblaze_get_pvr

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

Prototype

```
int microblaze_get_pvr(pvr_t *pvr);
```

Parameters

The following table lists the `microblaze_get_pvr` function arguments.

Table 35: **microblaze_get_pvr Arguments**

Name	Description
pvr-	address of PVR data structure to be populated

Returns

0 - SUCCESS -1 - FAILURE

Sleep Routines for MicroBlaze

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

***Note*:** The microblaze_sleep.h file may contain architecture-dependent items.

Table 36: **Quick Function Reference**

Type	Name	Arguments
void	MB_Sleep	MilliSeconds-

Functions

MB_Sleep

Provides delay for requested duration.

***Note*:** Instruction cache should be enabled for this to work.

Prototype

```
void MB_Sleep(u32 MilliSeconds) __attribute__((__deprecated__));
```

Parameters

The following table lists the `MB_Sleep` function arguments.

Table 37: **MB_Sleep Arguments**

Name	Description
MilliSeconds-	Delay time in milliseconds.

Returns

None.

Chapter 3

Cortex R5 Processor API

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.

Cortex R5 Processor Boot Code

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

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
DDR	0x00000000 - 0x7FFFFFFF	Normal write-back Cacheable

	Memory Range	Attributes of MPURegion
PL	0x80000000 - 0xBFFFFFFF	Strongly Ordered
QSPI	0xC0000000 - 0xDFFFFFFF	Device Memory
PCIe	0xE0000000 - 0xEFFFFFFF	Device Memory
STM_CORESIGHT	0xF8000000 - 0xF8FFFFFF	Device Memory
RPU_R5_GIC	0xF9000000 - 0xF90FFFFF	Device memory
FPS	0xFD000000 - 0xFDFFFFFF	Device Memory
LPS	0xFE000000 - 0xFFFFFFFF	Device Memory
OCM	0xFFFC0000 - 0xFFFFFFFF	Normal write-back Cacheable

Note: For a system where DDR is less than 2GB, region after DDR and before PL is marked as undefined in translation table. Memory range 0xFE000000-0xFEFFFFFF is allocated for upper LPS slaves, where as memory region 0xFF000000-0xFFFFFFFF is allocated for lower LPS slaves.

Table 38: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	INTPTR Addr u32 attrib
void	Xil_EnableMPU	None.
void	Xil_DisableMPU	None.
u32	Xil_SetMPURegion	Addr u64 size u32 attrib
u32	Xil_UpdateMPUConfig	u32 reg_num INTPTR address u32 size u32 attrib
void	Xil_GetMPUConfig	XMpu_Config mpuconfig
u32	Xil_GetNumOfFreeRegions	none
u32	Xil_GetNextMPURegion	none
u32	Xil_DisableMPURegionByRegNum	u32 reg_num
u16	Xil_GetMPUFreeRegMask	none

Table 38: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	Xil_SetMPURegionByRegNum	u32 reg_num address u64 size u32 attrib
void *	Xil_MemMap	void

Functions

Xil_SetTlbAttributes

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

Prototype

```
void Xil_SetTlbAttributes(INTPTR Addr, u32 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 39: **Xil_SetTlbAttributes Arguments**

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

Returns

None.

Xil_EnableMPU

Enable MPU for Cortex R5 processor.

This function invalidates I cache and flush the D Caches, and then enables the MPU.

Prototype

```
void Xil_EnableMPU(void);
```

Parameters

The following table lists the `Xil_EnableMPU` function arguments.

Table 40: **Xil_EnableMPU Arguments**

Name	Description
None.	

Returns

None.

Xil_DisableMPU

Disable MPU for Cortex R5 processors.

This function invalidates I cache and flush the D Caches, and then disabes the MPU.

Prototype

```
void Xil_DisableMPU(void);
```

Parameters

The following table lists the `Xil_DisableMPU` function arguments.

Table 41: **Xil_DisableMPU Arguments**

Name	Description
None.	

Returns

None.

Xil_SetMPURegion

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

Prototype

```
u32 Xil_SetMPURegion(INTPTR addr, u64 size, u32 attrib);
```

Parameters

The following table lists the `Xil_SetMPURegion` function arguments.

Table 42: **Xil_SetMPURegion Arguments**

Name	Description
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.

Xil_UpdateMPUConfig

Update the MPU configuration for the requested region number in the global MPU configuration table.

Prototype

```
u32 Xil_UpdateMPUConfig(u32 reg_num, INTPTR address, u32 size, u32 attrib);
```

Parameters

The following table lists the `Xil_UpdateMPUConfig` function arguments.

Table 43: **Xil_UpdateMPUConfig Arguments**

Name	Description
reg_num	The requested region number to be updated information for.
address	32 bit address for start of the region.
size	Requested size of the region.
attrib	Attribute for the corresponding region.

Returns

XST_FAILURE: When the requested region number if 16 or more. XST_SUCCESS: When the MPU configuration table is updated.

Xil_GetMPUConfig

The MPU configuration table is passed to the caller.

Prototype

```
void Xil_GetMPUConfig(XMpu_Config mpuconfig);
```

Parameters

The following table lists the `Xil_GetMPUConfig` function arguments.

Table 44: **Xil_GetMPUConfig Arguments**

Name	Description
mpuconfig	This is of type XMpu_Config which is an array of 16 entries of type structure representing the MPU config table

Returns

none

Xil_GetNumOfFreeRegions

Returns the total number of free MPU regions available.

Prototype

```
u32 Xil_GetNumOfFreeRegions(void);
```

Parameters

The following table lists the `Xil_GetNumOfFreeRegions` function arguments.

Table 45: **Xil_GetNumOfFreeRegions Arguments**

Name	Description
none	

Returns

Number of free regions available to users

Xil_GetNextMPURegion

Returns the next available free MPU region.

Prototype

```
u32 Xil_GetNextMPURegion(void);
```

Parameters

The following table lists the `Xil_GetNextMPURegion` function arguments.

Table 46: **Xil_GetNextMPURegion Arguments**

Name	Description
none	

Returns

The free MPU region available

Xil_DisableMPURegionByRegNum

Disables the corresponding region number as passed by the user.

Prototype

```
u32 Xil_DisableMPURegionByRegNum(u32 reg_num);
```

Parameters

The following table lists the `Xil_DisableMPURegionByRegNum` function arguments.

Table 47: **Xil_DisableMPURegionByRegNum Arguments**

Name	Description
reg_num	The region number to be disabled

Returns

XST_SUCCESS: If the region could be disabled successfully XST_FAILURE: If the requested region number is 16 or more.

Xil_GetMPUFreeRegMask

Returns the total number of free MPU regions available in the form of a mask.

A bit of 1 in the returned 16 bit value represents the corresponding region number to be available. For example, if this function returns 0xC0000, this would mean, the regions 14 and 15 are available to users.

Prototype

```
u16 Xil_GetMPUFreeRegMask(void);
```

Parameters

The following table lists the `Xil_GetMPUFreeRegMask` function arguments.

Table 48: **Xil_GetMPUFreeRegMask Arguments**

Name	Description
none	

Returns

The free region mask as a 16 bit value

Xil_SetMPURegionByRegNum

Enables the corresponding region number as passed by the user.

Prototype

```
u32 Xil_SetMPURegionByRegNum(u32 reg_num, INTPTR addr, u64 size, u32
attrib);
```

Parameters

The following table lists the `Xil_SetMPURegionByRegNum` function arguments.

Table 49: **Xil_SetMPURegionByRegNum Arguments**

Name	Description
reg_num	The region number to be enabled
address	32 bit address for start of the region.
size	Requested size of the region.
attrib	Attribute for the corresponding region.

Returns

XST_SUCCESS: If the region could be created successfully XST_FAILURE: If the requested region number is 16 or more.

Cortex R5 Processor Cache Functions

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.

Table 50: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheFlush	None.
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	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len
void	Xil_ICacheInvalidateLine	INTPTR adr

Functions

Xil_DCacheEnable

Enable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 51: **Xil_DCacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 52: **Xil_DCacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the entire Data cache.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 53: **Xil_DCacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

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.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 54: **Xil_DCacheInvalidateRange Arguments**

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

Returns

None.

Xil_DCacheFlush

Flush the entire Data cache.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 55: **Xil_DCacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheFlushRange

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.

Prototype

```
void Xil_DCacheFlushRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 56: **Xil_DCacheFlushRange Arguments**

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

Returns

None.

Xil_DCacheInvalidateLine

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.

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

Prototype

```
void Xil_DCacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 57: **Xil_DCacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheFlushLine

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.

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

Prototype

```
void Xil_DCacheFlushLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 58: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheStoreLine

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).

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

Prototype

```
void Xil_DCacheStoreLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheStoreLine` function arguments.

Table 59: **Xil_DCacheStoreLine Arguments**

Name	Description
adr	32bit address of the data to be stored

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 60: **Xil_ICacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 61: **Xil_ICacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 62: **Xil_ICacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

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.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 63: **Xil_ICacheInvalidateRange Arguments**

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

Returns

None.

Xil_ICacheInvalidateLine

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.

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

Prototype

```
void Xil_ICacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 64: **Xil_ICacheInvalidateLine Arguments**

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

Returns

None.

Cortex R5 Time Functions

The xtime_l.h provides access to 32-bit TTC timer counter. These functions can be used by applications to track the time.

Table 65: **Quick Function Reference**

Type	Name	Arguments
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_SetTime

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.

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

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 66: **XTime_SetTime Arguments**

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

Returns

None.

XTime_GetTime

Get the time from the timer counter register.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 67: **XTime_GetTime Arguments**

Name	Description
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

Cortex R5 event counter functions can be utilized to configure and control the Cortex-R5 performance monitor events. Cortex-R5 Performance Monitor has 3 event counters which can be used to count a variety of events described in Coretx-R5 TRM. The xpm_counter.h file defines configurations XPM_CNTRCFGx which can be used to program the event counters to count a set of events.

Table 68: **Quick Function Reference**

Type	Name	Arguments
void	Xpm_SetEvents	s32 PmcrCfg
void	Xpm_GetEventCounters	u32 * PmCtrValue
u32	Xpm_DisableEvent	Event
u32	Xpm_SetUpAnEvent	Event
u32	Xpm_GetEventCounter	Event Pointer
void	Xpm_DisableEventCounters	None.

Table 68: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xpm_EnableEventCounters	None.
void	Xpm_ResetEventCounters	None.
void	Xpm_SleepPerfCounter	u32 delay u64 frequency

Functions

Xpm_SetEvents

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

Prototype

```
void Xpm_SetEvents(s32 PmcrCfg);
```

Parameters

The following table lists the `Xpm_SetEvents` function arguments.

Table 69: **Xpm_SetEvents Arguments**

Name	Description
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.

Xpm_GetEventCounters

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

Prototype

```
void Xpm_GetEventCounters(u32 *PmCtrValue);
```

Parameters

The following table lists the `Xpm_GetEventCounters` function arguments.

Table 70: **Xpm_GetEventCounters Arguments**

Name	Description
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.

Xpm_DisableEvent

Disables the requested event counter.

Note: None.

Prototype

```
u32 Xpm_DisableEvent(u32 EventHandlerId);
```

Parameters

The following table lists the `Xpm_DisableEvent` function arguments.

Table 71: **Xpm_DisableEvent Arguments**

Name	Description
Event	Counter ID. The counter ID is the same that was earlier returned through a call to Xpm_SetUpAnEvent. Cortex-R5 supports only 3 counters. The valid values are 0, 1, or 2.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if the passed Counter ID is invalid (i.e. greater than 2).

Xpm_SetUpAnEvent

Sets up one of the event counters to count events based on the Event ID passed.

For supported Event IDs please refer xpm_counter.h. Upon invoked, the API searches for an available counter. After finding one, it sets up the counter to count events for the requested event.

Note: None.

Prototype

```
u32 Xpm_SetUpAnEvent(u32 EventID);
```

Parameters

The following table lists the `Xpm_SetUpAnEvent` function arguments.

Table 72: **Xpm_SetUpAnEvent Arguments**

Name	Description
Event	ID. For valid values, please refer xpm_counter.h.

Returns

- Counter Number if successful. For Cortex-R5, valid return values are 0, 1, or 2.
- XPM_NO_COUNTERS_AVAILABLE (0xFF) if all counters are being used

Xpm_GetEventCounter

Reads the counter value for the requested counter ID.

This is used to read the number of events that has been counted for the requsted event ID. This can only be called after a call to Xpm_SetUpAnEvent.

Note: None.

Prototype

```
u32 Xpm_GetEventCounter(u32 EventHandlerId, u32 *CntVal);
```

Parameters

The following table lists the `Xpm_GetEventCounter` function arguments.

Table 73: **Xpm_GetEventCounter Arguments**

Name	Description
Event	Counter ID. The counter ID is the same that was earlier returned through a call to Xpm_SetUpAnEvent. Cortex-R5 supports only 3 counters. The valid values are 0, 1, or 2.
Pointer	to a 32 bit unsigned int type. This is used to return the event counter value.

Returns

- XST_SUCCESS if successful.

- XST_FAILURE if the passed Counter ID is invalid (i.e. greater than 2).

Xpm_DisableEventCounters

This function disables the Cortex R5 event counters.

Prototype

```
void Xpm_DisableEventCounters(void);
```

Parameters

The following table lists the `Xpm_DisableEventCounters` function arguments.

Table 74: **Xpm_DisableEventCounters Arguments**

Name	Description
None.	

Returns

None.

Xpm_EnableEventCounters

This function enables the Cortex R5 event counters.

Prototype

```
void Xpm_EnableEventCounters(void);
```

Parameters

The following table lists the `Xpm_EnableEventCounters` function arguments.

Table 75: **Xpm_EnableEventCounters Arguments**

Name	Description
None.	

Returns

None.

Xpm_ResetEventCounters

This function resets the Cortex R5 event counters.

Prototype

```
void Xpm_ResetEventCounters(void);
```

Parameters

The following table lists the `Xpm_ResetEventCounters` function arguments.

Table 76: **Xpm_ResetEventCounters Arguments**

Name	Description
None.	

Returns

None.

Xpm_SleepPerfCounter

This is helper function used by sleep/usleep APIs to generate delay in sec/usec.

Prototype

```
void Xpm_SleepPerfCounter(u32 delay, u64 frequency);
```

Parameters

The following table lists the `Xpm_SleepPerfCounter` function arguments.

Table 77: **Xpm_SleepPerfCounter Arguments**

Name	Description
delay	- delay time in sec/usec
frequency	- Number of countes in second/micro second

Returns

None.

Cortex R5 Processor Specific Include Files

The xpseudo_asm.h includes xreg_cortexr5.h and xpseudo_asm_gcc.h.

The xreg_cortexr5.h file contains definitions for inline assembler code. It provides inline definitions for Cortex R5 GPRs, SPRs,co-processor registers and Debug register

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.

Cortex R5 peripheral definitions

The xparameters_ps.h file contains the canonical definitions and constant declarations for peripherals within hardblock, attached to the ARM Cortex R5 core. These definitions can be used by drivers or applications to access the peripherals.

Chapter 4

ARM Processor Common API

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

ARM Processor Exception Handling

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.

Table 78: **Quick Function Reference**

Type	Name	Arguments
void	Xil_ExceptionRegisterHandler	exception_id `Xil_ExceptionHandler` Handler void * Data
void	Xil_ExceptionRemoveHandler	exception_id
void	Xil_GetExceptionRegisterHandler	exception_id `Xil_ExceptionHandler` * Handler void ** Data
void	Xil_ExceptionInit	None.
void	Xil_DataAbortHandler	void
void	Xil_PrefetchAbortHandler	void
void	Xil_UndefinedExceptionHandler	void

Functions

Xil_ExceptionRegisterHandler

Register a handler for a specific exception.

This handler is being called when the processor encounters the specified exception.

Note: None.

Prototype

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

Parameters

The following table lists the `Xil_ExceptionRegisterHandler` function arguments.

Table 79: **Xil_ExceptionRegisterHandler Arguments**

Name	Description
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.

Xil_ExceptionRemoveHandler

Removes the Handler for a specific exception Id.

The stub Handler is then registered for this exception Id.

Note: None.

Prototype

```
void Xil_ExceptionRemoveHandler(u32 Exception_id);
```

Parameters

The following table lists the `Xil_ExceptionRemoveHandler` function arguments.

Table 80: **Xil_ExceptionRemoveHandler Arguments**

Name	Description
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.

Xil_GetExceptionRegisterHandler

Get a handler for a specific exception.

This handler is being called when the processor encounters the specified exception.

Note: None.

Prototype

```
void Xil_GetExceptionRegisterHandler(u32 Exception_id, Xil_ExceptionHandler
*Handler, void **Data);
```

Parameters

The following table lists the `Xil_GetExceptionRegisterHandler` function arguments.

Table 81: **Xil_GetExceptionRegisterHandler Arguments**

Name	Description
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.

Xil_ExceptionInit

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).

***Note*:** None.

Prototype

```
void Xil_ExceptionInit(void);
```

Parameters

The following table lists the `Xil_ExceptionInit` function arguments.

Table 82: **Xil_ExceptionInit Arguments**

Name	Description
None.	

Returns

None.

Xil_DataAbortHandler

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

Prototype

```
void Xil_DataAbortHandler(void *CallBackRef);
```

Xil_PrefetchAbortHandler

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

Prototype

```
void Xil_PrefetchAbortHandler(void *CallBackRef);
```

Xil_UndefinedExceptionHandler

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

Prototype

```
void Xil_UndefinedExceptionHandler(void *CallBackRef);
```

Chapter 5

Cortex A9 Processor API

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

Cortex A9 Processor Boot Code

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

None.

Note:

translation_table.S 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
DDR	0x00000000 - 0x3FFFFFFF	Normal write-back Cacheable
PL	0x40000000 - 0xBFFFFFFF	Strongly Ordered

	Memory Range	Definition in Translation Table
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
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

For region 0x00000000 - 0x3FFFFFFF, a system where DDR is less than 1GB, region after DDR and before PL is marked as undefined/reserved in translation table. In 0xF8000000 - 0xF8FFFFFF, 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. For region 0xFFF00000 - 0xFFFFFFFF, 0xFFF00000 to 0xFFFB0000 is reserved but due to 1MB granual size, it is not possible to define separate region for it

***Note*:**

Cortex A9 Processor Cache Functions

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.

Table 83: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len

Table 83: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushRange	INTPTR adr u32 len
void	Xil_ICacheEnable	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
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	None.
void	Xil_L1DCacheDisable	None.
void	Xil_L1DCacheInvalidate	None.
void	Xil_L1DCacheInvalidateLine	u32 adr
void	Xil_L1DCacheInvalidateRange	u32 adr u32 len
void	Xil_L1DCacheFlush	None.
void	Xil_L1DCacheFlushLine	u32 adr
void	Xil_L1DCacheFlushRange	u32 adr u32 len

Table 83: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_L1DCacheStoreLine	Address
void	Xil_L1ICacheEnable	None.
void	Xil_L1ICacheDisable	None.
void	Xil_L1ICacheInvalidate	None.
void	Xil_L1ICacheInvalidateLine	u32 adr
void	Xil_L1ICacheInvalidateRange	u32 adr u32 len
void	Xil_L2CacheEnable	None.
void	Xil_L2CacheDisable	None.
void	Xil_L2CacheInvalidate	None.
void	Xil_L2CacheInvalidateLine	u32 adr
void	Xil_L2CacheInvalidateRange	u32 adr u32 len
void	Xil_L2CacheFlush	None.
void	Xil_L2CacheFlushLine	u32 adr
void	Xil_L2CacheFlushRange	u32 adr u32 len
void	Xil_L2CacheStoreLine	u32 adr

Functions

Xil_DCacheEnable

Enable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 84: **Xil_DCacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 85: **Xil_DCacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the entire Data cache.

Note: None.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 86: **Xil_DCacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

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. data. This issue raises few possibilities. work.

1. 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. 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.

***Note*:** None.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 87: **Xil_DCacheInvalidateRange Arguments**

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

Returns

None.

Xil_DCacheFlush

Flush the entire Data cache.

Note: None.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 88: **Xil_DCacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheFlushRange

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.

Note: None.

Prototype

```
void Xil_DCacheFlushRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 89: **Xil_DCacheFlushRange Arguments**

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

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 90: **Xil_ICacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 91: **Xil_ICacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

***Note*:** None.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 92: **Xil_ICacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

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.

***Note*:** None.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 93: **Xil_ICacheInvalidateRange Arguments**

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

Returns

None.

Xil_DCacheInvalidateLine

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.

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

Prototype

```
void Xil_DCacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 94: **Xil_DCacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheFlushLine

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.

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

Prototype

```
void Xil_DCacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 95: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheStoreLine

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).

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

Prototype

```
void Xil_DCacheStoreLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheStoreLine` function arguments.

Table 96: **Xil_DCacheStoreLine Arguments**

Name	Description
adr	32bit address of the data to be stored.

Returns

None.

Xil_ICacheInvalidateLine

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.

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

Prototype

```
void Xil_ICacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 97: **Xil_ICacheInvalidateLine Arguments**

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

Returns

None.

Xil_L1DCacheEnable

Enable the level 1 Data cache.

***Note*:** None.

Prototype

```
void Xil_L1DCacheEnable(void);
```

Parameters

The following table lists the `Xil_L1DCacheEnable` function arguments.

Table 98: **Xil_L1DCacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_L1DCacheDisable

Disable the level 1 Data cache.

***Note*:** None.

Prototype

```
void Xil_L1DCacheDisable(void);
```

Parameters

The following table lists the `Xil_L1DCacheDisable` function arguments.

Table 99: **Xil_L1DCacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_L1DCacheInvalidate

Invalidate the level 1 Data cache.

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

Prototype

```
void Xil_L1DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_L1DCacheInvalidate` function arguments.

Table 100: **Xil_L1DCacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_L1DCacheInvalidateLine

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.

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

Prototype

```
void Xil_L1DCacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1DCacheInvalidateLine` function arguments.

Table 101: **Xil_L1DCacheInvalidateLine Arguments**

Name	Description
adr	32bit address of the data to be invalidated.

Returns

None.

Xil_L1DCacheInvalidateRange

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.

Note: None.

Prototype

```
void Xil_L1DCacheInvalidateRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L1DCacheInvalidateRange` function arguments.

Table 102: **Xil_L1DCacheInvalidateRange Arguments**

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

Returns

None.

Xil_L1DCacheFlush

Flush the level 1 Data cache.

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

Prototype

```
void Xil_L1DCacheFlush(void);
```

Parameters

The following table lists the `Xil_L1DCacheFlush` function arguments.

Table 103: **Xil_L1DCacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_L1DCacheFlushLine

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.

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

Prototype

```
void Xil_L1DCacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1DCacheFlushLine` function arguments.

Table 104: **Xil_L1DCacheFlushLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_L1DCacheFlushRange

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.

***Note*:** None.

Prototype

```
void Xil_L1DCacheFlushRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L1DCacheFlushRange` function arguments.

Table 105: **Xil_L1DCacheFlushRange Arguments**

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

Returns

None.

Xil_L1DCacheStoreLine

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).

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

Prototype

```
void Xil_L1DCacheStoreLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1DCacheStoreLine` function arguments.

Table 106: **Xil_L1DCacheStoreLine Arguments**

Name	Description
Address	to be stored.

Returns

None.

Xil_L1ICacheEnable

Enable the level 1 instruction cache.

***Note*:** None.

Prototype

```
void Xil_L1ICacheEnable(void);
```

Parameters

The following table lists the `Xil_L1ICacheEnable` function arguments.

Table 107: **Xil_L1ICacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_L1ICacheDisable

Disable level 1 the instruction cache.

Note: None.

Prototype

```
void Xil_L1ICacheDisable(void);
```

Parameters

The following table lists the `Xil_L1ICacheDisable` function arguments.

Table 108: **Xil_L1ICacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_L1ICacheInvalidate

Invalidate the entire level 1 instruction cache.

Note: None.

Prototype

```
void Xil_L1ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_L1ICacheInvalidate` function arguments.

Table 109: **Xil_L1ICacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_L1ICacheInvalidateLine

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.

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

Prototype

```
void Xil_L1ICacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1ICacheInvalidateLine` function arguments.

Table 110: **Xil_L1ICacheInvalidateLine Arguments**

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

Returns

None.

Xil_L1ICacheInvalidateRange

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.

***Note*:** None.

Prototype

```
void Xil_L1ICacheInvalidateRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L1ICacheInvalidateRange` function arguments.

Table 111: **Xil_L1ICacheInvalidateRange Arguments**

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

Returns

None.

Xil_L2CacheEnable

Enable the L2 cache.

Note: None.

Prototype

```
void Xil_L2CacheEnable(void);
```

Parameters

The following table lists the `Xil_L2CacheEnable` function arguments.

Table 112: **Xil_L2CacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_L2CacheDisable

Disable the L2 cache.

Note: None.

Prototype

```
void Xil_L2CacheDisable(void);
```

Parameters

The following table lists the `Xil_L2CacheDisable` function arguments.

Table 113: **Xil_L2CacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_L2CacheInvalidate

Invalidate the entire level 2 cache.

Note: None.

Prototype

```
void Xil_L2CacheInvalidate(void);
```

Parameters

The following table lists the `Xil_L2CacheInvalidate` function arguments.

Table 114: **Xil_L2CacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_L2CacheInvalidateLine

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.

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

Prototype

```
void Xil_L2CacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_L2CacheInvalidateLine` function arguments.

Table 115: **Xil_L2CacheInvalidateLine Arguments**

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

Returns

None.

Xil_L2CacheInvalidateRange

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.

Note: None.

Prototype

```
void Xil_L2CacheInvalidateRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L2CacheInvalidateRange` function arguments.

Table 116: **Xil_L2CacheInvalidateRange Arguments**

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

Returns

None.

Xil_L2CacheFlush

Flush the entire level 2 cache.

Note: None.

Prototype

```
void Xil_L2CacheFlush(void);
```

Parameters

The following table lists the `Xil_L2CacheFlush` function arguments.

Table 117: **Xil_L2CacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_L2CacheFlushLine

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.

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

Prototype

```
void Xil_L2CacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_L2CacheFlushLine` function arguments.

Table 118: **Xil_L2CacheFlushLine Arguments**

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

Returns

None.

Xil_L2CacheFlushRange

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.

Note: None.

Prototype

```
void Xil_L2CacheFlushRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L2CacheFlushRange` function arguments.

Table 119: **Xil_L2CacheFlushRange Arguments**

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

Returns

None.

Xil_L2CacheStoreLine

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).

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

Prototype

```
void Xil_L2CacheStoreLine(u32 adr);
```

Parameters

The following table lists the `Xil_L2CacheStoreLine` function arguments.

Table 120: **Xil_L2CacheStoreLine Arguments**

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

Returns

None.

Cortex A9 Processor MMU Functions

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

Table 121: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	INTPTR Addr u32 attrib
void	Xil_EnableMMU	None.
void	Xil_DisableMMU	None.
void *	Xil_MemMap	UINTPTR PhysAddr size_t size u32 flags

Functions

Xil_SetTlbAttributes

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

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

Prototype

```
void Xil_SetTlbAttributes(INTPTR Addr, u32 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 122: **Xil_SetTlbAttributes Arguments**

Name	Description
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.

Xil_EnableMMU

Enable MMU for cortex A9 processor.

This function invalidates the instruction and data caches, and then enables MMU.

Prototype

```
void Xil_EnableMMU(void);
```

Parameters

The following table lists the `Xil_EnableMMU` function arguments.

Table 123: **Xil_EnableMMU Arguments**

Name	Description
None.	

Returns

None.

Xil_DisableMMU

Disable MMU for Cortex A9 processors.

This function invalidates the TLBs, Branch Predictor Array and flushed the D Caches before disabling the MMU.

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

Prototype

```
void Xil_DisableMMU(void);
```

Parameters

The following table lists the `Xil_DisableMMU` function arguments.

Table 124: **Xil_DisableMMU Arguments**

Name	Description
None.	

Returns

None.

Xil_MemMap

Memory mapping for Cortex A9 processor.

***Note*:** : Previously this was implemented in libmetal. Move to embeddedsw as this functionality is specific to A9 processor.

Prototype

```
void * Xil_MemMap(UINTPTR PhysAddr, size_t size, u32 flags);
```

Parameters

The following table lists the `Xil_MemMap` function arguments.

Table 125: **Xil_MemMap Arguments**

Name	Description
PhysAddr	is physical address.
size	is size of region.
flags	is flags used to set translation table.

Returns

Pointer to virtual address.

Cortex A9 Time Functions

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.

Table 126: **Quick Function Reference**

Type	Name	Arguments
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_SetTime

Set the time in the Global Timer Counter Register.

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

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 127: **XTime_SetTime Arguments**

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

Returns

None.

XTime_GetTime

Get the time from the Global Timer Counter Register.

Note: None.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 128: **XTime_GetTime Arguments**

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

Returns

None.

Cortex A9 Event Counter Function

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.

Table 129: **Quick Function Reference**

Type	Name	Arguments
void	Xpm_SetEvents	s32 PmcrCfg
void	Xpm_GetEventCounters	u32 * PmCtrValue

Functions

Xpm_SetEvents

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

Note: None.

Prototype

```
void Xpm_SetEvents(s32 PmcrCfg);
```

Parameters

The following table lists the `Xpm_SetEvents` function arguments.

Table 130: **Xpm_SetEvents Arguments**

Name	Description
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.

Xpm_GetEventCounters

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

Note: None.

Prototype

```
void Xpm_GetEventCounters(u32 *PmCtrValue);
```

Parameters

The following table lists the `Xpm_GetEventCounters` function arguments.

Table 131: **Xpm_GetEventCounters Arguments**

Name	Description
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.

PL310 L2 Event Counters Functions

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.

Table 132: **Quick Function Reference**

Type	Name	Arguments
void	XL2cc_EventCtrInit	s32 Event0 s32 Event1
void	XL2cc_EventCtrStart	None.
void	XL2cc_EventCtrStop	u32 * EveCtr0

Functions

XL2cc_EventCtrInit

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

***Note*:** The definitions for event codes XL2CC_* can be found in xl2cc_counter.h.

Prototype

```
void XL2cc_EventCtrInit(s32 Event0, s32 Event1);
```

Parameters

The following table lists the `XL2cc_EventCtrInit` function arguments.

Table 133: **XL2cc_EventCtrInit Arguments**

Name	Description
Event0	Event code for counter 0.

Table 133: **XL2cc_EventCtrInit Arguments** *(cont'd)*

Name	Description
Event1	Event code for counter 1.

Returns

None.

XL2cc_EventCtrStart

This function starts the event counters in L2 Cache controller.

Note: None.

Prototype

```
void XL2cc_EventCtrStart(void);
```

Parameters

The following table lists the `XL2cc_EventCtrStart` function arguments.

Table 134: **XL2cc_EventCtrStart Arguments**

Name	Description
None.	

Returns

None.

XL2cc_EventCtrStop

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

Note: None.

Prototype

```
void XL2cc_EventCtrStop(u32 *EveCtr0, u32 *EveCtr1);
```

Parameters

The following table lists the `XL2cc_EventCtrStop` function arguments.

Table 135: **XL2cc_EventCtrStop Arguments**

Name	Description
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.

Cortex A9 Processor and pl310 Errata Support

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.

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

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.

Cortex A53 32-bit Processor Boot Code

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

Cortex A53 32-bit Processor Cache Functions

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.

Table 136: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	None.

Table 136: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushRange	INTPTR adr u32 len
void	Xil_DCacheInvalidateLine	u32 adr
void	Xil_DCacheFlushLine	u32 adr
void	Xil_ICacheInvalidateLine	u32 adr
void	Xil_ICacheEnable	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len

Functions

Xil_DCacheEnable

Enable the Data cache.

***Note*:** None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 137: **Xil_DCacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 138: **Xil_DCacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the Data cache.

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

Note: In Cortex-A53, functionality to simply invalid 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.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 139: **Xil_DCacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

Invalidate the Data cache for the given address range.

The cachelines present in the adderss range are cleaned and invalidated

@notice In Cortex-A53, functionality to simply invalid 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.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 140: **Xil_DCacheInvalidateRange Arguments**

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

Returns

None.

Xil_DCacheFlush

Flush the Data cache.

@notice None.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 141: **Xil_DCacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheFlushRange

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 system memory before the lines are invalidated.

@notice None.

Prototype

```
void Xil_DCacheFlushRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 142: **Xil_DCacheFlushRange Arguments**

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

Returns

None.

Xil_DCacheInvalidateLine

Invalidate a Data cache line.

The cacheline is cleaned and invalidated.

Note: In Cortex-A53, functionality to simply invalid 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.

Prototype

```
void Xil_DCacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 143: **Xil_DCacheInvalidateLine Arguments**

Name	Description
adr	32 bit address of the data to be invalidated.

Returns

None.

Xil_DCacheFlushLine

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.

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

Prototype

```
void Xil_DCacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 144: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_ICacheInvalidateLine

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.

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

Prototype

```
void Xil_ICacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 145: **Xil_ICacheInvalidateLine Arguments**

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

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

@notice None.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 146: **Xil_ICacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 147: **Xil_ICacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Note: None.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 148: **Xil_ICacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

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.

@notice None.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 149: **Xil_ICacheInvalidateRange Arguments**

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

Returns

None.

Cortex A53 32-bit Processor MMU Handling

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

None.

***Note*:**

Table 150: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	UINTPTR Addr u32 attrib
void	Xil_EnableMMU	None.
void	Xil_DisableMMU	None.

Functions

Xil_SetTlbAttributes

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

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

Prototype

```
void Xil_SetTlbAttributes(UINTPTR Addr, u32 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 151: **Xil_SetTlbAttributes Arguments**

Name	Description
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.

Xil_EnableMMU

Enable MMU for Cortex-A53 processor in 32bit mode.

This function invalidates the instruction and data caches before enabling MMU.

Prototype

```
void Xil_EnableMMU(void);
```

Parameters

The following table lists the `Xil_EnableMMU` function arguments.

Table 152: **Xil_EnableMMU Arguments**

Name	Description
None.	

Returns

None.

Xil_DisableMMU

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.

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

Prototype

```
void Xil_DisableMMU(void);
```

Parameters

The following table lists the `Xil_DisableMMU` function arguments.

Table 153: **Xil_DisableMMU Arguments**

Name	Description
None.	

Returns

None.

Cortex A53 32-bit Mode Time Functions

xtime_l.h provides access to the 64-bit physical timer counter.

Table 154: **Quick Function Reference**

Type	Name	Arguments
void	XTime_StartTimer	None.
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_StartTimer

Start the 64-bit physical timer counter.

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

Prototype

```
void XTime_StartTimer(void);
```

Parameters

The following table lists the `XTime_StartTimer` function arguments.

Table 155: **XTime_StartTimer Arguments**

Name	Description
None.	

Returns

None.

XTime_SetTime

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.

Note: None.

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 156: **XTime_SetTime Arguments**

Name	Description
Xtime_Global	64bit Value to be written to the Global Timer Counter Register. But since the function does not contain anything, the value is not used for anything.

Returns

None.

XTime_GetTime

Get the time from the physical timer counter register.

Note: None.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 157: **XTime_GetTime Arguments**

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

Returns

None.

Cortex A53 32-bit Processor Specific Include Files

The xpseudo_asm.h includes xreg_cortexa53.h and xpseudo_asm_gcc.h. The xreg_cortexa53.h file contains definitions for inline assembler code. It provides inline definitions for Cortex A53 GPRs, SPRs, co-processor registers 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 etc. These inline assembler instructions can be used from drivers and user applications written in C.

Chapter 7

Cortex A53 64-bit Processor Boot Code

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.

Cortex A53 64-bit Processor Cache Functions

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.

Table 158: **Quick Function Reference**

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr INTPTR len
void	Xil_DCacheInvalidateLine	INTPTR adr
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushLine	INTPTR adr
void	Xil_ICacheEnable	None.

Table 158: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr INTPTR len
void	Xil_ICacheInvalidateLine	INTPTR adr
void	Xil_ConfigureL1Prefetch	u8 num

Functions

Xil_DCacheEnable

Enable the Data cache.

***Note*:** None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 159: **Xil_DCacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

***Note*:** None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 160: **Xil_DCacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the Data cache.

The contents present in the cache are cleaned and invalidated.

Note: In Cortex-A53, functionality to simply invalid 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.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 161: **Xil_DCacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

Invalidate the Data cache for the given address range.

The cachelines present in the adderss range are cleaned and invalidated

Note: In Cortex-A53, functionality to simply invalid 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.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, INTPTR len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 162: **Xil_DCacheInvalidateRange Arguments**

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

Returns

None.

Xil_DCacheInvalidateLine

Invalidate a Data cache line.

The cacheline is cleaned and invalidated.

Note: In Cortex-A53, functionality to simply invalid 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.

Prototype

```
void Xil_DCacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 163: **Xil_DCacheInvalidateLine Arguments**

Name	Description
adr	64bit address of the data to be flushed.

Returns

None.

Xil_DCacheFlush

Flush the Data cache.

Note: None.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 164: **Xil_DCacheFlush Arguments**

Name	Description
None.	

Returns

None.

Xil_DCacheFlushLine

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.

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

Prototype

```
void Xil_DCacheFlushLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 165: **Xil_DCacheFlushLine Arguments**

Name	Description
adr	64bit address of the data to be flushed.

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 166: **Xil_ICacheEnable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 167: **Xil_ICacheDisable Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Note: None.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 168: **Xil_ICacheInvalidate Arguments**

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

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.

Note: None.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, INTPTR len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 169: **Xil_ICacheInvalidateRange Arguments**

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

Returns

None.

Xil_ICacheInvalidateLine

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.

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

Prototype

```
void Xil_ICacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 170: **Xil_ICacheInvalidateLine Arguments**

Name	Description
adr	64bit address of the instruction to be invalidated.

Returns

None.

Xil_ConfigureL1Prefetch

Configure the maximum number of outstanding data prefetches allowed in L1 cache.

Note: This function is implemented only for EL3 privilege level.

Prototype

```
void Xil_ConfigureL1Prefetch(u8 num);
```

Parameters

The following table lists the `Xil_ConfigureL1Prefetch` function arguments.

Table 171: **Xil_ConfigureL1Prefetch Arguments**

Name	Description
num	maximum number of outstanding data prefetches allowed, valid values are 0-7.

Returns

None.

Cortex A53 64-bit Processor MMU Handling

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

None.

Note:

Table 172: **Quick Function Reference**

Type	Name	Arguments
void	Xil_SetTlbAttributes	UINTPTR Addr u64 attrib

Functions

Xil_SetTlbAttributes

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.

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

Prototype

```
void Xil_SetTlbAttributes(UINTPTR Addr, u64 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 173: **Xil_SetTlbAttributes Arguments**

Name	Description
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.

Cortex A53 64-bit Mode Time Functions

xtime_l.h provides access to the 64-bit physical timer counter.

Table 174: **Quick Function Reference**

Type	Name	Arguments
void	XTime_StartTimer	None.
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_StartTimer

Start the 64-bit physical timer counter.

Note: The timer is initialized only if it is disabled. If the timer is already running this function does not perform any operation. This API is effective only if BSP is built for EL3. For EL1 Non-secure, it simply exits.

Prototype

```
void XTime_StartTimer(void);
```

Parameters

The following table lists the `XTime_StartTimer` function arguments.

Table 175: **XTime_StartTimer Arguments**

Name	Description
None.	

Returns

None.

XTime_SetTime

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.

***Note*:** None.

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 176: **XTime_SetTime Arguments**

Name	Description
Xtime_Global	64bit value to be written to the physical timer counter register. Since API does not do anything, the value is not utilized.

Returns

None.

XTime_GetTime

Get the time from the physical timer counter register.

***Note*:** None.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 177: **XTime_GetTime Arguments**

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

Returns

None.

Cortex A53 64-bit Processor Specific Include Files

The xpseudo_asm.h includes xreg_cortexa53.h and xpseudo_asm_gcc.h. 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 etc. 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.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright