LatticeMico Mutex

The LatticeMico Mutex allows Multi-master environments to coordinate accesses to a shared resource using WISHBONE or JTAG interface. The LatticeMico Mutex provides a protocol to ensure mutually exclusive ownership of a shared resource. This removes the burden on the multiple masters sharing the same resource to implement logic to arbitrate ownership or shared resource between each other.

Version

This document describes the 1.0 version of the LatticeMico Mutex.

Features

The Mutex core has the following features:

- A master can access the Mutex core via the WISHBONE Slave or JTAG interfaces.
- Provides a hardware-based atomic test-and-set operation.
- Ability to be configured without the JTAG interface.
- Ability to recognize up to 16 masters.
- Ability to be configured to maintain ownership information for up to 256 shared resources.

The Mutex core can be instantiated in all types of designs generated by LatticeMico System (i.e., with and without Lattice CPU cores). The number of instantiations in a design will be limited to one.

Functional Description

The Mutex core consists of 8-bit Mutex registers that can be accessed via two interfaces: WISHBONE and JTAG. The core can support up to 16 Mutex registers, one for each shared resource in the system that requires access control. The data written to and read from the Mutex core is the 8-bit value of the Mutex register.

Basic Operation

The basic operation of the core per Mutex (i.e., shared resource or Mutex register) is:

1. Every master has a unique identifier. IDs 0 through 3 are reserved. IDs 4 through 15 are available for assignment. The current owner of the Mutex can be identified by the 4-bit ID field in the Mutex register.
2. The Mutex is unlocked and available if the VALUE field in the Mutex register are zero. Otherwise the Mutex is locked and unavailable. All masters, regardless of the current ownership, can poll the Mutex register to check whether it is currently available.
3. A master can write to the Mutex register only if one of the following conditions is true.
    a. The ID field in the Mutex register matches the ID of the master (encoded within the data being written) performing the write.
    b. The VALUE field in the Mutex register is zero.
4. A master that wants to take ownership of the Mutex should perform a write to the Mutex register with its ID and a non-zero VALUE. It should follow up with a read of the Mutex register to check if it was able to successfully take ownership of the Mutex; the master has taken ownership if the value read from the Mutex register is the same value that was written to by the master.

From an implementation perspective, a ‘same-cycle’ ownership request from JTAG has higher priority than an ownership request from WISHBONE. For more information regarding Mutex Register, see “Register Descriptions” on page 5.

JTAG Interface

The Mutex core uses Lattice's JTAG Controller FPGA Fabric Interface to enable masters take control of a Mutex via JTAG. The Mutex registers can be read from (or written to) via a JTAG-accessible 18-bit register shown in Figure 1. The Mutex register is visible via Lattice's JTAG Controller FPGA Fabric Interface when the design contains the Mutex core. The process of accessing this interface is beyond the scope of this document. The master should adhere to the aforementioned basic operation when it wants to take ownership of the Mutex via JTAG. The examples show how a JTAG-based

access to the 16-bit register in Figure 1 translates to a read from (or write to) a Mutex register.

Figure 1: Register accessible via JTAG

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Read	Reserved				Mutex Register #				ID				VALUE			

Example 1: Write Mutex 2 - ID = 0x1, VALUE = 0xF

1. Write 0x2021F to JTAG register

Example 2: Read Mutex 15

1. Write 0x30F00 to JTAG register (note that this will initiate a read process in the Mutex core and the JTAG register will be updated with the value in the requested Mutex register)
2. Read JTAG register (this command will return the contents of the requested Mutex register)

WISHBONE Interface

All the Mutex registers are visible via the WISHBONE slave interface of the Mutex core at offset from base address 0x0 through 0xF. The WISHBONE slave interface has an 8-bit data bus. The Mutex core only accepts WISHBONE Classic read and write cycles and all other types of accesses are ignored.

Configuration

The following sections describe the graphical user interface (UI) parameters, the hardware description language (HDL) parameters, and the I/O ports that user can use to configure and operate the LatticeMico Mutex.

UI Parameters

Table 1 shows the UI parameters available for configuring the LatticeMico Mutex through the Mico System Builder (MSB) interface.

Table 1: Mutex UI Parameters

Dialog Box Options	Description	Allowable Values	Default Value
Instance Name	Specifies the name of the Mutex instance.	Alphanumeric and underscores	mutex
Base Address	Specifies the base address for configuring the Mutex. The minimum boundary alignment is 0x80.	0X80000000-0XFFFFFFFF	0X80000000
Mutex Core Configuration			
JTAG Access	Indicates if the JTAG Interface is enabled.	selected | not selected	not selected
Mutex Count	Specifies the number of ASC Log	0-16	1

HDL Parameters

Table 2 lists the parameters that appear in the HDL.

Table 2: Mutex HDL Parameter

Parameter Name	Description	Allowable Values
MUTEX_COUNT	Indicates the number of Mutexes (i.e., Mutex registers) in the core.	0 to 15
MUTEX_JTAG	Indicates that the JTAG Interface is enabled.	0 | 1
MUTEX_INIT_OWNER_N	Indicates the owner of the Mutex N at power-up or assertion of Mutex core reset.	0 to 15
MUTEX_INIT_VALUE_N	Indicates the contents of VALUE field of Mutex register N at power-up or assertion of Mutex core reset.	0 to 15

I/O Ports

Table 3 describes the input and output ports of the LatticeMico Mutex.

Table 3: Mutex I/O Ports

I/O Port	Direction	Active	Description
System Clock and Reset			
CLK	I	—	WISHBONE System Clock
RESET	I	Low	System Reset
WISHBONE Slave Signal			
S_CYC_I	I	High	Indicates a valid bus cycle is present on the bus.
S_STB_I	I	High	Asserts an acknowledgment in response to the assertion of the WISHBONE Master strobe.
S_WE_I	I	—	Level sensitive Write/Read control signal. Low - Read operation, High - Write operation
S_ADR_I	I	—	32-bit wide address used to select a specific register
S_DAT_I	I	—	8-bit data used to read a byte of data from a specific register
S_CTI_I	I	—	Not used, always tied to 0
S_BTE_I	I	—	Not used, always tied to 0
S_LOCK_I	I	—	Not used, always tied to 0
S_SEL_I	I	—	Not used, always tied to 0
S_DAT_O	O	—	8-bit data used to read a byte of data from a specific register
S_ACK_O	O	High	Indicates the requested transfer is acknowledged.
S_ERR_O	O	—	Indicates the address is incorrect
S_RTY_O	O	—	Not used, always tied to 0

Register Descriptions

The LatticeMico Mutex WISHBONE module has a register map to allow the service of the hardened functions through the WISHBONE bus interface read/write operations. Table 4 describe the register map of the Mutex module.

Table 4: WISHBONE Addressable Registers for Mutex Module

Register Name	Register Function	Address	Access
Mutex_N	Holds the info of the Mutex of Register N	0x0 – 0xF	Read/Write

Mutex Register Definition – Mutex_N

The WISHBONE host has Read and Write access to these registers.

Table 5: Control Register Bit Definition

Bit	Field	Description
3:0	Mutex Value	Identify if this Mutex is locked by a Master Component
7:4	Mutex ID	Identify the ownership of this Mutex Register

LatticeMico8 Microprocessor Software Support

This section describes the LatticeMico8 microcontroller software support provided for the LatticeMico Mutex component.

Device Driver

The Mutex device driver interacts directly with the Mutex instance. This section describes the limitations, type definitions, structure, and functions of the Mutex device driver.

Type Definitions

This section describes the type definitions for the Mutex device context structure. This structure, shown in Figure 2, contains the Mutex component instance-specific information and is dynamically generated in the DDStructs.h header file. This information is largely filled in by the managed build process by extracting the Mutex component-specific information from the platform specification file. As part of the managed build process, designers can choose to control the size of the generated structure, and hence the software executable, by selectively enabling some of the elements in this structure via C preprocessor macro definitions. These C preprocessor macro definitions are explained later in this document. You should not manipulate the members directly, because this structure is for exclusive use by the device driver. Figure 6 describes the parameters of the Mutex device context structure shown in Figure 2.

Device Context Structure

Figure 2 shows the Mutex device context structure.

Figure 2: Mutex Device Context Structure

```
struct st_MicoMutexCtx_t{
    const char *   name;
    size_t   base;
    unsigned char   mutex_count;
} MicoMutexCtx_t;
```

Table 6 describes the Mutex device context parameters.

Table 6: Mutex Device Context Parameters

Parameter	Data Type	Description
name	const char*	Mutex instance name (entered in MSB)
base	size_t	MSB-assigned base address for this instance
mutex_count	unsigned char	Processor interrupt line to which this instance is connected

C Preprocessor Macro Definitions

This section describes the C preprocessor macro definitions that are available to the software developer. There are two types of macro definitions: 'object-like' and 'function-like'.

The 'object-like' macro definitions do not take any arguments and are used to control the size of the generated application executable. There are three ways an 'object-like' macro definition can be used by the software developer.

1. Manually adding the -D<macro name> option to the compiler's command line in the application's 'Build Properties'. Refer to the *LatticeMico8 Developer User Guide* for more information on how to manually add the macro definition in the application's 'Build Properties' GUI.
2. Automatically adding the -D<macro name> option to the compiler's command-line in the application's 'Build Properties' by enabling the 'check-box' associated with the macro definition. Refer to the *LatticeMico8 Developer User Guide* for more information on how to set up the check/uncheck the macro definitions in the application's 'Build Properties' GUI.
3. Manually adding the macro definition to the C code using the following syntax:

   ```
   #define <macro name>
   ```

Table 7: C Preprocessor Function-like Macros For Mutex

Macro Name	Second Argument to Macro / Third Argument to Macro (if exist.	Description
MICO_MUTEX_READ_REGISTER	The 8-bit value reads from the Mutex Register content / address offset	This macro reads a character from the Mutex content register with a specific address offset
MICO_MUTEX_WRITE_REGISTER	The 8-bit value write to the Mutex Register content / address offset	This macro writes a character to the Mutex content register with a specific address offset

Note: The first argument to the macro is the Mutex address.

Functions

This section describes the implemented device-driver-specific functions.

MicoMutexInit Function

```
void MicoMutexInit (MicoMutexCtx_t *ctx);
```

This is the Mutex initialization function. Table 8 describes the parameter in the MicoMutexInit function syntax.

Table 8: MicoMutexInit Function Parameter

Parameter	Description
MicoMutexCtx_t	Pointer to a valid MicoMutexCtx _t structure representing a valid Mutex instance.

MicoMutex_Lock Function

```
void MicoMutex_Lock (MicoMutexCtx_t *ctx,
                     unsigned char mutex_number,
                     unsigned char mutex_owner,
                     unsigned char mutex_value);
```

This function allows the calling function to lock, and take ownership of, a Mutex. The Mutex will be locked only if the current VALUE field is zero, or the OWNER ID is the same as the one provided as an argument to the function. Once the Mutex is locked, the corresponding Mutex register will contain the OWNER ID and VALUE provided to the function. This function will not return control to the calling function until it has locked the Mutex.

Table 9 describes the parameter in the MicoMutex_Lock function syntax.

Table 9: MicoMutex_Lock Function Parameter

Parameter	Description
MicoMutexCtx	Pointer to a valid MicoMutexCtx_t structure representing a valid Mutex instance.
unsigned char	Specific which Mutex Register to be locked
unsigned char	Specific the Mutex OWNER ID
unsigned char	Specific Mutex VALUE

MicoMutex_Unlock Function

```
void MicoMutex_Unlock (MicoMutexCtx_t *ctx,
                       unsigned char mutex_number,
                       unsigned char mutex_owner);
```

The Mutex unlock function that allows the calling function to unlock, release ownership of, a Mutex. The Mutex will be unlocked only if the OWNER ID is the same as the one provided as an argument to the function. Once the Mutex is locked, the corresponding Mutex register will reset the VALUE to 0.

Table 10 describes the parameter in the MicoMutex_Unlock function syntax.

Table 10: MicoMutex_Unlock Function Parameter

Parameter	Description
MicoMutexCtx	Pointer to a valid MicoMutexCtx_t structure representing a valid Mutex instance.
unsigned char	Specific which Mutex Register to be unlocked
unsigned char	Specific the Mutex OWNER ID

MicoMutex_GetOwner Function

```
unsigned char MicoMutex_GetOwner (MicoMutexCtx_t *ctx,
                       unsigned char mutex_number);
```

The Mutex GetOwner function that allows the calling function to get the OWNER ID of the corresponding Mutex register number provided to this function.

Table 11 describes the parameter in the MicoMutex_GetOwner function syntax.

Table 11: MicoMutex_GetOwner Function Parameter

Parameter	Description
MicoMutexCtx	Pointer to a valid MicoMutexCtx_t structure representing a valid Mutex instance.
unsigned char	Specific which Mutex Register number

Table 12 describes the values returned by the MicoMutex_GetOwner Function

Table 12: Values Returned by the MicoMutex_GetOwner Function

Return Value	Description
0 - 15	OWNER ID of the given Mutex Register

MicoMutex_GetValue Function

```
unsigned char MicoMutex_GetValue (MicoMutexCtx_t *ctx,
                                  unsigned char mutex_number);
```

The Mutex GetValue function that allows the calling function to get the Mutex VALUE of the corresponding Mutex register number provided to this function.

Table 13 describes the parameter in the MicoMutex_GetValue function syntax.

Table 13: MicoMutex_GetValue Function Parameter

Parameter	Description
MicoMutexCtx	Pointer to a valid MicoMutexCtx_t structure representing a valid Mutex instance.
unsigned char	Specific which Mutex Register number

Table 14 describes the values returned by the MicoMutex_GetValue Function.

Table 14: Values Returned by the MicoMutex_GetValue Function

Return Value	Description
0 - 15	Mutex VALUE of the given Mutex Register

Software Usage Example

This section provides an example of using the Mutex. The example is shown in Figure 3 and assumes the presence of a Mutex component named “mutex”, and a EFB component named “efb”.

Figure 3: Mutex Software Example

```
#include "MicoUtils.h"
#include "DDStructs.h"
#include "MicoEFB.h"
#include "MicoMutex.h"

int main(void){
  MicoMutexCtx _t * mutex = & mutex_mutex;
  MicoEFBCtx_t *efb = &efb_efb;

unsigned char tx_data[3];
unsigned char i2c_slave_addr = 0x60;

  unsigned char i2c_mutex = 0x0;
unsigned char owner_ID = 0x5; // Default Owner ID for LM8
unsigned char mutex_VALUE = 0x1;

// Lock the Mutex for the shared resource - I2C
MicoMutex_Lock (mutex, i2c_mutex, owner_ID, mutex_VALUE);

// Perform the I2C communication via EFB
MicoEFB_I2CWrite (efb, 0x1, 0x0, 0x2, (unsigned char *)
&tx_data, 0x1, 0x0, 0x1, i2c_slave_addr);

// Unlock the Mutex
  MicoMutex_Unlock (mutex, i2c_mutex, owner_ID);
  return(0);


}
```

Revision History

Component Version	Description
1.0	Initial Release.

.

Trademarks