OR1K support

Generated by Doxygen 1.8.6

Tue Nov 18 2014 11:03:05

Contents

Chapter 1

Module Index

1.1 Modules

Here is a list of all modules:

Chapter 2

Module Documentation

2.1 OR1K macros

Macros

- #define REG8(add) ∗((volatile unsigned char ∗) (add))
- #define REG16(add) ∗((volatile unsigned short ∗) (add))
- #define REG32(add) ∗((volatile unsigned long ∗) (add))

2.1.1 Detailed Description

2.1.2 Macro Definition Documentation

2.1.2.1 #define REG16(*add*) ∗((volatile unsigned short ∗) (add))

Access halfword-sized memory mapped register

Used to access a 16 byte-sized memory mapped register. It avoids usage errors when not defining register addresses volatile and handles casting correctly.

See REG8() for an example.

Parameters

add	Register address

2.1.2.2 #define REG32(*add*) ∗((volatile unsigned long ∗) (add))

Access word-sized memory mapped register

Used to access a word-sized memory mapped register. It avoids usage errors when not defining register addresses volatile and handles casting correctly.

See REG8() for an example.

Parameters

add	Register address

2.1.2.3 #define REG8(*add*) ∗((volatile unsigned char ∗) (add))

Access byte-sized memory mapped register

Used to access a byte-sized memory mapped register. It avoids usage errors when not defining register addresses volatile and handles casting correctly.

Example for both read and write:

```
uint8_t status = REG8(IPBLOCK_STATUS_REG_ADDR);
REG8(IPBLOCK_ENABLE) = 1;
```

Parameters

add	Register address

2.2 OR1K interrupt control

Typedefs

- typedef void(∗ or1k_interrupt_handler_fptr)(uint32_t data)

Functions

- void or1k_interrupt_handler_add (int line, or1k_interrupt_handler_fptr handler, uint32_t data)
- void or1k_interrupt_enable (int line)
- void or1k_interrupt_disable (int line)
- uint32_t or1k_interrupts_disable (void)
- void or1k_interrupts_enable (void)
- void or1k_interrupts_restore (uint32_t status)

2.2.1 Detailed Description

Interrupt control function prototypes

2.2.2 Typedef Documentation

2.2.2.1 typedef void(∗ or1k_interrupt_handler_fptr)(uint32_t data)

Function pointer to interrupt handler functions

2.2.3 Function Documentation

2.2.3.1 void or1k_interrupt_disable (int *line*)

Disable interrupts from a given line

Mask given interrupt line. It can be unmasked using or1k_interrupt_enable().

Parameters

line	Interrupt line to disable

2.2.3.2 void or1k_interrupt_enable (int *line*)

Enable interrupts from a given line

Unmask the given interrupt line. It is also important to enable interrupts in general, e.g., using or1k_interrupts_-enable().

Parameters

line	Interrupt line to enable

2.2.3.3 void or1k_interrupt_handler_add (int *line,* or1k_interrupt_handler_fptr *handler,* uint32_t *data*)

Add interrupt handler for interrupt line

Registers a callback function for a certain interrupt line.

Parameters

line	Interrupt line/id to register a handler for
handler	Handler to register
data	Data value passed to the handler

2.2.3.4 uint32_t or1k_interrupts_disable (void)

Disable interrupts

This disables the interrupt exception. This is sufficient to disable all interrupts. It does not change the mask register (which is modified using or1k_interrupt_enable() and or1k_interrupt_disable()).

The interrupt exception can be enabled using or1k_interrupts_enable().

Finally, the status of the interrupt exception enable flag is returned by this function. That allows to call this function even if interrupts are already disabled. To restore the value of the interrupt exception enable flag, use the or1k_-interrupts_restore() function. That way you avoid to accidentally enable interrupts. Example:

```
void f() {
uint32_t interrupt_status = or1k_interrupts_disable();
// do something
or1k_interrupts_restore(status);
}
```

This code will preserve the original status of the interrupt enable flag.

Returns

Interrupt exception enable flag before call

2.2.3.5 void or1k_interrupts_enable (void)

Enable interrupt exception

Enable the interrupt exception. Beside the interrupt exception, it is also necessary to enable the individual interrupt lines using or1k_interrupt_enable().

You should avoid using this function together with or1k_interrupts_disable() to guard atomic blocks as it unconditionally enables the interrupt exception (see documentation of or1k_interrupts_disable()).

2.2.3.6 void or1k_interrupts_restore (uint32_t *status*)

Restore interrupt exception enable flag

This function restores the given status to the processor. or1k_interrupts_restore(0) is identical to or1k_interrupts_-disable() and or1k_interrupts_restore(SPR_SR_IEE) is identical to or1k_interrupts_enable().

It is for example used to guard an atomic block and restore the original status of the interrupt exception enable flag as returned by or1k_interrupts_disable(). See the documentation of or1k_interrupts_disable() for a usage example.

Parameters

status	Status of the flag to restore

2.3 Exception handling

Typedefs

- typedef void(∗ or1k_exception_handler_fptr)(void)

Functions

- void or1k_exception_handler_add (int id, or1k_exception_handler_fptr handler)

2.3.1 Detailed Description

2.3.2 Typedef Documentation

2.3.2.1 typedef void(∗ or1k_exception_handler_fptr)(void)

Function pointer to an exception handler function

2.3.3 Function Documentation

2.3.3.1 void or1k_exception_handler_add (int *id,* or1k_exception_handler_fptr *handler*)

Register exception handler

Register an exception handler for the given exception id. This handler is in the following then called when the exception occurs. You can thereby individually handle those exceptions.

Parameters

id	Exception id
handler	Handler callback

2.4 SPR access

2.5 Miscellaneous utility functions

Functions

- void or1k_report (unsigned long int value)
- unsigned long int or1k_rand (void)

2.5.1 Detailed Description

2.5.2 Function Documentation

2.5.2.1 unsigned long int or1k_rand (void)

Get (pseudo) random number

This should return pseudo-random numbers, based on a Galois LFSR.

Returns

(Pseudo) Random number

2.5.2.2 void or1k_report (unsigned long int *value*)

Report value to simulator

Uses the built-in simulator functionality.

Parameters

value	Value to report

2.6 Cache control

Functions

- void or1k_icache_enable (void)
- void or1k_icache_disable (void)
- void or1k_icache_flush (uint32_t entry)
- void or1k_dcache_enable (void)
- void or1k_dcache_disable (void)
- void or1k_dcache_flush (unsigned long entry)

2.6.1 Detailed Description

2.6.2 Function Documentation

2.6.2.1 void or1k_dcache_disable (void)

Disable data cache

2.6.2.2 void or1k_dcache_enable (void)

Enable data cache

2.6.2.3 void or1k_dcache_flush (unsigned long *entry*)

Flush data cache

Invalidate data cache entry

Parameters

entry	Entry to invalidate

2.6.2.4 void or1k_icache_disable (void)

Disable instruction cache

2.6.2.5 void or1k_icache_enable (void)

Enable instruction cache

2.6.2.6 void or1k_icache_flush (uint32_t *entry*)

Flush instruction cache

Invalidate instruction cache entry

Parameters

entry	Entry to invalidate

2.7 MMU control

Functions

- void or1k_immu_enable (void)
- void or1k_immu_disable (void)
- void or1k_dmmu_enable (void)
- void or1k_dmmu_disable (void)

2.7.1 Detailed Description

2.7.2 Function Documentation

2.7.2.1 void or1k_dmmu_disable (void)

Disable data MMU

2.7.2.2 void or1k_dmmu_enable (void)

Enable data MMU

2.7.2.3 void or1k_immu_disable (void)

Disable instruction MMU

2.7.2.4 void or1k_immu_enable (void)

Enable instruction MMU

2.8 Timer control

Modules

- Multicore and Synchronization Support

Functions

- int or1k_timer_init (unsigned int hz)
- void or1k_timer_set_period (uint32_t hz)
- void or1k_timer_set_handler (void(∗handler)(void))
- void or1k_timer_set_mode (uint32_t mode)
- void or1k_timer_enable (void)
- uint32_t or1k_timer_disable (void)
- void or1k_timer_restore (uint32_t sr_tee)
- void or1k_timer_pause (void)
- void or1k_timer_reset (void)
- unsigned long or1k_timer_get_ticks (void)
- void or1k_timer_reset_ticks (void)
- uint32_t **or1k_critical_start** ()
- void **or1k_critical_end** (uint32_t restore)

2.8.1 Detailed Description

The tick timer can be used for time measurement, operating system scheduling etc. By default it is initialized to continuously count the ticks of a certain period after calling or1k_timer_init(). The period can later be changed using or1k_timer_set_period().

The timer is controlled using or1k_timer_enable(), or1k_timer_disable(), or1k_timer_restore(), or1k_timer_pause(). After initialization it is required to enable the timer the first time using or1k_timer_enable(). or1k_timer_disable() only disables the tick timer interrupts, it does not disable the timer counting. If you plan to use a pair of or1k_timer_disable() and or1k_timer_enable() to protect sections of your code against interrupts you should use or1k_timer_disable() and or1k_timer_restore(), as it may be possible that the timer interrupt was not enabled before disabling it, enable would then start it unconditionally. or1k_timer_pause() pauses the counting.

In the default mode you can get the tick value using or1k_timer_get_ticks() and reset this value using or1k_timer_reset_ticks().

Example for using the default mode:

```
int main() {
uint32_t ticks = 0;
uint32_t timerstate;
or1k_timer_init(100);
or1k_timer_enable();
while (1) {
while (ticks == or1k_timer_get_ticks()) { }
timerstate = or1k_timer_disable();
// do something atomar
or1k_timer_restore(timerstate);
if (ticks == 100) {
printf("A second elapsed\n");
or1k_timer_reset_ticks();
ticks = 0;
}
}
}
```

It is possible to change the mode of the tick timer using or1k_timer_set_mode(). Allowed values are the correct bit pattern (including the bit positions) for the TTMR register, it is recommended to use the macros defined in spr-defs.h. For example, implementing an operating system with scheduling decisions of varying duration favors the implementation of single run tick timer. Here, each quantum is started before leaving the operating system kernel. The counter can be restarted with or1k_timer_reset(). Example:

```
void tick_handler(void) {
// Make schedule decision
// and set new thread
or1k_timer_reset();
// End of exception, new thread will run
}

int main() {
// Configure operating system and start threads..

// Configure timer
or1k_timer_init(50);
or1k_timer_set_handler(&tick_handler);
or1k_timer_set_mode(SPR_TTMR_SR);
or1k_timer_enable();

// Schedule first thread and die..
}
```

2.8.2 Function Documentation

2.8.2.1 uint32_t or1k_timer_disable (void)

Disable timer interrupt

This disables the timer interrupt exception and returns the state of the interrupt exception enable flag before the call. This can be used with or1k_timer_restore() to implement sequences of code that are not allowed to be interrupted. Using or1k_timer_enable() will unconditionally enable the interrupt independent of the state before calling or1k_-timer_disable(). For an example see Timer control.

Returns

Status of timer interrupt before call

2.8.2.2 void or1k_timer_enable (void)

Enable timer interrupt

Enable the timer interrupt exception, independent of the status before. If you want to enable the timer conditionally, for example to implement a non-interruptible sequence of code, you should use or1k_timer_restore(). See the description of or1k_timer_disable() for more details.

The enable will also restore the mode if the timer was paused previously.

2.8.2.3 unsigned long or1k_timer_get_ticks (void)

Get timer ticks

Get the global ticks of the default configuration. This will increment the tick counter according to the preconfigured period.

Returns

Current value of ticks

2.8.2.4 int or1k_timer_init (unsigned int *hz*)

Initialize tick timer

This initializes the tick timer in default mode (see Timer control for details).

Parameters

hz	Initial period of the tick timer

Returns

0 if successful, -1 if timer not present

2.8.2.5 void or1k_timer_pause (void)

Pause timer counter

Pauses the counter of the tick timer. The counter will hold its current value and it can be started again with or1k_-timer_enable() which will restore the configured mode.

2.8.2.6 void or1k_timer_reset (void)

Reset timer counter

2.8.2.7 void or1k_timer_reset_ticks (void)

Reset timer ticks

Resets the timer ticks in default configuration to 0.

2.8.2.8 void or1k_timer_restore (uint32_t *sr_tee*)

Restore timer interrupt exception flag

Restores the timer interrupt exception flag as returned by or1k_timer_disable(). See the description of or1k_timer-_disable() and Timer control for details and an example.

Parameters

sr_tee	Status of timer interrupt

2.8.2.9 void or1k_timer_set_handler (void(∗)(void) *handler*)

Replace the timer interrupt handler

By default the tick timer is used to handle timer ticks. The user can replace this with an own handler for example when implementing an operating system.

Parameters

handler	The callback function pointer to the handler

2.8.2.10 void or1k_timer_set_mode (uint32_t *mode*)

Set timer mode

The timer has different modes (see architecture manual). The default is to automatically restart counting (SPR_TT-MR_RT), others are single run (SPR_TTMR_SR) and continuous run (SPR_TTMR_CR).

Parameters

mode	a valid mode (use definitions from spr-defs.h as it is important that those are also at the correct position in the bit field!)

2.8.2.11 void or1k_timer_set_period (uint32_t *hz*)

Set period of timer

Set the period of the timer to a value in Hz. The frequency from the board support package is used to determine the match value.

2.9 Multicore and Synchronization Support

Functions

- uint32_t or1k_coreid (void)
- uint32_t or1k_numcores (void)
- uint32_t or1k_sync_ll (void ∗address)
- int or1k_sync_sc (void ∗address, uint32_t value)
- uint32_t or1k_sync_cas (void ∗address, uint32_t compare, uint32_t swap)
- int or1k_sync_tsl (void ∗address)
- void **or1k_uart_set_read_cb** (void(∗cb)(char c))

2.9.1 Detailed Description

2.9.2 Function Documentation

2.9.2.1 uint32_t or1k_coreid (void)

Read core identifier

Returns

Core identifier

2.9.2.2 uint32_t or1k_numcores (void)

Read number of cores

Returns

Total number of cores

2.9.2.3 uint32_t or1k_sync_cas (void ∗ *address,* uint32_t *compare,* uint32_t *swap*)

Compare and Swap

Loads a data item from the memory and compares a given value to it. If the values match, a new value is written to the memory, if they mismatch, the operation is aborted. The whole operation is atomic, i.e., it is guaranteed that no other core changes the value between the read and the write.

Parameters

address	Address to operate on
compare	Compare value
swap	New value to write

Returns

The value read from memory (can be used to check for success)

2.9.2.4 uint32_t or1k_sync_ll (void ∗ *address*)

Load linked

Load a value from the given address and link it. If the following or1k_sync_sc() goes to the same address and there was no conflicting access between loading and storing, the value is written back, else the write fails.

Parameters

address	Address to load value from

Returns

Value read from the address

2.9.2.5 int or1k_sync_sc (void ∗ *address,* uint32_t *value*)

Store conditional

Conditionally store a value to the address. The address must have been read before using or1k_sync_ll() and there must be no other load link after that, otherwise this will always fail. In case there was no other write to the same address in between the load link and the store conditional, the store is successful, otherwise it will also fail.

Parameters

address	Address to conditionally store to
value	Value to write to address

Returns

1 if success, 0 if fail

2.9.2.6 int or1k_sync_tsl (void ∗ *address*)

Test and Set Lock

Check for a lock on an address. This means, if there is 0 at an address it will overwrite it with one and return 0. If the lock was already set (value 1 read from address), the function returns 1. The operation is atomic.

Parameters

address	Address of the lock

Returns

0 if success, 1 if failed

Index