Thread Manager Access Registers

The following registers are used to access a variety of functions built in to the FPGA. All registers are accessed as having a 32 bit width even though the implementation may only use a subset of the full 32 bits. In these cases, the least significant bits are utilized, and the upper bits are padded with zeros for read operations and ignored for write operations. Write operations performed on read only registers generate an exception to the CPU.

The depth of a given register specifies the number of successive, 32 bit locations that are utilized by this register. Registers having a depth greater than one are utilizing the least significant address lines to specify a parameter to be passed to a hardware function associated with this register. To keep each access on an even 32 bit boundary, the parameter to be passed in must be multiplied by 4 before adding to the base address of the register.

Example: A particular register accepts an 8 bit parameter encoded into the address used to access the register (depth=256). The address corresponding to a parameter of zero equals the base address of the register. The address corresponding to a parameter of one equals the base address of the register plus four.

References in the following descriptions to a thread’s status refer to that thread’s entry in the Thread ID Table and utilize the format specified in the Thread ID Table Encoding on page 4.

References in the following descriptions to “queue” refer to the Ready to Run Queue, implemented as a linked list within the Thread ID Table, where each thread has a pointer to the next thread in the queue.

Note: This document is a modification of revision two of the Software Thread Manager API. It was modified to be a preliminary design document for a Thread Manager.

que_length (REMOVE)

This call is being removed from the thread manager.

add_thread

Read only

Depth = 256

Overview

Reading this register adds the encoded thread ID to the queue. If the thread happens to be a SW thread, the thread ID is added to the queue. If the thread happens to be a HW thread, the thread manager sends a “RUN” command to the command register of the thread.

Error Checks

The thread’s status must be: USED, NOT_EXITED, and NOT_QUEUED

Calls to Scheduler

is_queued(threadID)
Scheduler will return a 0 on the “data” line if the thread ID is queued, else will return a 1.

enqueue(threadID)

Scheduler will return 0x00000000 if the thread is a SW thread, will return the address of the HW thread, if it is a HW thread. Thread Manager to send a “RUN” to the command register if this is the case.

**Pseudo-Code**

```plaintext
q = is_queued(threadID)
attr = threads[threadID]
if ( attr.used and !attr.exited and !q )
    enqueue(threadID)
    return 0
else
    return error
```

**Return Value on Success**

0

**Return Value on Error**

If the error check fails, the value returned is the thread’s current status with the ERR_BIT set to indicate that the operation was unsuccessful.

**set_idle_thread**

Read only.

Depth = 256

**Overview**

Sets the value of the idle thread ID.

**Error Checks**

The encoded thread ID must be USED and NOT_EXITED and NOT_QUEUED.

**Calls to Scheduler**

is_queued(threadID)

Scheduler will return a 0 on the “data” line if the thread ID is queued, else will return a 1.

set_idle_thread(threadID)

On write, set_idle_thread is called. The scheduler maintains the value of the idle thread.

**Pseudo-Code**

```plaintext
attr = threads[threadID]
q = is_queued(threadID)
if ( attr.used and !attr.exited and !q )
    set_idle_thread(threadID)
```
return 0
else
  return error

Return Value on Success
0

Return Value on Error
If the error check fails, the value returned is the thread’s current status with the ERR_BIT set to indicate that the operation was unsuccessful.

get_idle_thread
Read only.
Depth = 1

Overview
Returns the value of the idle thread ID.

Error Checks
Check to make sure the idle thread has previously been set.

Calls to Scheduler
get_idle_thread()
Scheduler returns the value of the idle thread. If the idle thread has not previously been sit, an error status is set.

Pseudo-Code
threadID = get_idle_thread()
if isError(threadID)
  return error
else
  return threadID

Return Value on Success
0

Return Value on Error
If no thread ID is available the read operation returns zero + the ERR_BIT set.

next_thread
Read only.
Depth = 1

**Overview**

This register is read to retrieve the ID of the next thread in the queue. The thread manager calls the scheduler's next_thread. The scheduler removes and returns the thread’s ID from the queue. If the queue is empty, the scheduler returns the contents of the idle_thread register.

By a matter of principal, a hardware thread should never call next_thread.

**Error Checks**

**Calls to Scheduler**

dequeue()

Scheduler returns the value of the next thread to receive CPU time. Scheduler will return the value of the idle thread if there are no other threads scheduled to run.

**Pseudo-Code**

```plaintext
threadID = valueOf(nextThreadRegister)
dequeue()
return threadID
```

**Return Value on Success**

The ID of the thread to run next. The thread manager returns the thread ID in bits 23..30 with bit 31 set for all error conditions.

**Return Value on Error**

current_cpu_thread (DEPRICATED)

Read only.

Depth = 1

**Overview**

Returns the thread ID of the thread currently running on the CPU.

Assumes a single CPU system.

For a hardware thread, the HW should read its own THREADID register, and should not make a call to the thread manager to retrieve this value.

**Error Checks**

**Calls to Scheduler**
**Pseudo-Code**

```
return current_cpu_thread_id
```

**Return Value on Success**

The ID is returned in bits 23..30, with bit 31 set for all error conditions, else cleared.

**Return Value on Error**

**create_thread_detached**

Read only.

Depth = 1

**Overview**

This register is read to retrieve an unused thread ID in preparation for creating a new, detached thread.

If successful, the status of the new thread is updated to be; USED, NOT_EXITED, NOT_JOINED, DETACHED, and the thread’s parent ID field(PID) is set to zero.

**Error Checks**

Make sure that all thread IDs are not already used.

**Calls to Scheduler**

**Pseudo-Code**

```
if ( parentID == current_cpu_thread )
    if ( newThreadIDAval )
        return new threadID
        attr = used, !exited, !joined, detached
        attr.parent = 0
        threads[threadID] = attr
    else
        return error
else
    return error
```

**Return Value on Success**

The thread ID for the new thread. The ID is returned on bits 23..30, all other bits are 0.

**Return Value on Error**

If no thread ID is available the read operation returns zero + the ERR_BIT set.
**create_thread_joinable**

Read only.

Depth = 256

**Overview**

This register is read to retrieve an unused thread ID number in preparation for creating a new joinable thread. The encoded thread ID, is the calling thread's ID. The new thread must explicitly be joined later to prevent memory leaks.

If successful, the status of the new thread is updated to be; USED, NOT_EXITED, NOT_JOINED, NOT_DETACHED, the thread's parent ID field(PID) is set to the value of the calling thread.

**Error Checks**

Make sure that all thread IDs are not already used.

Make sure the calling thread ID matches the thread ID in the current_cpu_thread.

**Calls to Scheduler**

**Pseudo-Code**

```c
if ( parentID == current_cpu_thread )
    if ( newThreadIDAvail )
        return new threadID
        attr = used, !exited, !joined, !detached
        attr.parent = current_cpu_thread
        threads[threadID] = attr
    else
        return error
else
    return error
```

**Return Value on Success**

The thread ID for the new thread. The ID is returned on bits 23..30, all other bits are 0.

**Return Value on Error**

If no thread ID is available the read operation returns zero + the ERR_BIT set.

If the parent thread ID does not match the current_cpu_thread, return ???

**clear_thread**

Read only.

Depth = 256
Overview

Reading this register deallocates the encoded thread ID by setting the thread’s status to, NOT_USED, NOT_EXITED, NOT_JOINED, NOT_DETACHED, and the thread’s PID field is set to zero.

Error Checks

The calling thread must match the current_cpu_thread

Calls to Scheduler

Pseudo-Code

if ( parentID == current_cpu_thread )
    attr = !used, !exited, !joined, !detached
    attr.parent = 0
    threads[threadID] = attr
else
    return error

Return Value on Success

0

Return Value on Error

If the error check fails, the value returned is the thread’s current status with the ERR_BIT set to indicate that the operation was unsuccessful.

yield_thread

Read only.

Depth = 256

Overview

This register is read to place the current SW thread back on the queue and then return the ID of the next thread in the queue. If the queue is empty, the current SW thread is not re-added to the queue, and its ID is returned instead.

By a matter of principal, this call should only be made by a SW thread. A check is made to make sure the encoded thread ID is from a SW thread. If the thread ID is from a HW thread, the call has no meaning.

Error Checks

Make sure the thread is a SW thread. This is done by checking that the calling thread ID matches the current_cpu_thread.

Calls to Scheduler

dehqueue()
Scheduler returns the value of the next thread ID to run.

enqueue(threadID)

Scheduler adds the passed in thread ID to the queue to eventually run.

**Pseudo-Code**

```plaintext
if ( callingID == current_cpu_thread )
    threadID = valueOf(nextThreadRegister)
    dequeue()
    enqueue(callingID)
    return threadID
else
    return error
```

**Return Value on Success**

The ID of the thread to run next. The thread manager returns the thread ID in bits 23..30 with bit 31 set for all error conditions.

**Return Value on Error**

If the error check fails, the value returned is the thread’s current status with the ERR_BIT set to indicate that the operation was unsuccessful.

**exit_thread**

read only, depth = 256

**Overview**

This register is read to update the status of the calling thread to show that it has terminated. The calling thread should encode its ID. There are no calls to the scheduler. The effects of this operation vary depending on the current status of the thread and are summarized below.

If the thread’s status shows it to be detached, the thread ID is deallocated, changing its status to, NOT_USED, NOT_EXITED, NOT_JOINED, NOT_DETACHED, and its PID field to zero.

If the thread’s status is NOT_DETACHED, the thread’s status is updated to USED, EXITED. Then, if the thread’s PID ! = 0, and the thread status is furthermore JOINED, check if the parent thread’s status = USED, NOT_EXITED, NOT_QUEUED. If so, add the parent thread’s ID to the queue, and return zero. If the parent happens to be HW, a “RUN” command is sent to the hardware thread.

**Error Checks**

If the calling thread is JOINED, check that the parent thread is not queued. This indicates that the parent did not block until the calling thread was finished.

**Calls to Scheduler**

isQueued(parentID)
Scheduler will return a 0 on the “data” line if the thread ID is queued, else will return a 1.

**Pseduo-Code**

```python
attr = threads[threadID]
if attr.detached
    set attributes of thread to unused
else //joined or joinable
    set attributes to used & exited
    if attr.joined
        parentID = attr.parent
        if ( isQueued(parentID) )
            return error
    else
        enqueue(parentID)
```

**Return Value on Success**

0

**Return Value on Error**

Return THREAD_ALLREADY_QUEUED

join_thread

read only, depth = 256 * 256

**Overview**

This register is read to join the encoded thread ID(child) to the encoded current thread (parent). The child thread’s status is first checked to verify that it is, USED, NOT_JOINED, and NOT_DETACHED. If the test passes, the child’s status is checked to see if it has already exited. If it has, the value 0 + THREAD_ALREADY_TERMINATED is returned, else the child’s status is changed to joined, and a value of zero is returned. The calling thread, is removed from the queue, via the scheduler's dequeue call. If the calling thread is HW, it is the HW thread responsibility to wait until it is set to RUN.

**Error Checks**

The child thread status must be USED, NOT_JOINED, NOT_DETACHED

The child thread has the passed in parentID listed as its parent.

**Calls to Scheduler**

decqueue(threadID)

removes the threadID from the queue.

**Pseduo-Code**

```python
attrChild = threads[childID]
attrParent = threads[parentID]
```
if attrChild.USED and attrChild.NOT_JOINED, and
attrChild.NOT_DETACHED
    if attrChild.parent == parentID
        if attrChild.EXITED
            return THREAD_ALREADY_TERMINATED
        else
            attrChild.JOINED = true
            dequeue(parentID)
            return 0
    else
        return error
else
    return error

**Return Value on Success**

0 or THREAD_ALREADY_TERMINATED

**Return Value on Error**

stuff

detach_thread (REMOVE)
read only, depth = 256 * 256

**Overview**

**Error Checks**

**Calls to Scheduler**

**Return Value on Success**

**Return Value on Error**

This register is read to detach the encoded thread ID(child) from the encoded current thread (parent). The child thread’s status is first checked to verify that it is, used, ~exited, ~joined, ~detached. If this test fails, the child’s status + ERR_BIT is returned. If the test passes, the child’s status is changed to detached, the child’s PID is set to zero, and a value of zero is returned.
read_thread

read (write if in debug “stop” mode), depth = 256

Overview

Reading this register returns the encoded thread IDs row from the Thread ID Table without producing any side effects. The ERR_BIT and auxiliary status bits (E1, E2, E3) are returned as zeros. If the design is in debug, stopped mode, writing to register sets this row value to the data written.

Error Checks

Calls to Scheduler

Pseduo-Code

attr = threads(threadID)
return attr

Return Value on Success

Thread attributes for passed in thread ID.

Return Value on Error

Troubleshooting Registers

The following registers are included to enhance troubleshooting the system’s operation.

exception_cause, exception_address read only, depth = 1

Certain events cause a critical exception to be raised. These events should not occur during normal operation and are indicative of a failure within the system. When one of these events is detected, a code representing the particular type of event is stored in the exception_cause register and the associated address that was being accessed is stored in the exception_address register. The system SW can then read these registers to determine the reason for the interrupt. The causes and codes are listed below.

<table>
<thead>
<tr>
<th>Cause</th>
<th>Code returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write to Read Only Register</td>
<td>1</td>
</tr>
<tr>
<td>Undefined Address</td>
<td>2</td>
</tr>
</tbody>
</table>
Soft Reset Failure

soft_stop  read, write, depth = 1

Writing any value to this register asserts the Soft_Stop signal, used by all system IP’s to halt operation. Reading this register returns the value of the Soft_Stop signal in the LSB, all other bits = zero.

soft_start  read, write, depth = 1

Writing any value to this register de-asserts the Soft_Stop and all Soft_Reset signals. Reading this register returns all zeros.

soft_reset  read, write, depth = 1

Writing to this register selectively asserts a number of soft_reset signals depending on the data written. Each bit corresponds to a particular IP; User IP(27), SpinLock(28), Semaphores(29), Scheduler(30), SWTM(31=LSB).

Reading this register returns all zeros with a one in any position(s) corresponding to an IP that failed to signal completion before an encoded time delay. (default delay = 4096 clock cycles)

Scheduler Interface

The Thread Manager and the Scheduler communicate through a dedicated interface. This interface is detailed below.

There are eight signals between the Thread Manager and the Scheduler. Four signals going in each direction.

Thread Manager to Scheduler Signals

opcode (6 bits): The opcode tells the scheduler what function to perform. There are six valid values.

• no_op (0b00 0000): Thread Manager is not asking for any information.
• is_queued (0b00 0001): Thread Manager is asking if the thread id set on the data line is currently queued by the scheduler. Scheduler returns “0x0000 0001” on the data_return line if the thread is queued, else scheduler will return “0x0000 0000.”
• enqueue (0b00 0010): Thread Manager is asking for the thread id set on the data line to be added to the queue.

• dequeue (0b00 0011): Thread Manager is asking the scheduler to remove the top thread id from the queue.

• set_idle_thread (0b00 0100): Thread Manager is asking the scheduler to set the thread id on the data line to be the idle thread.

• get_idle_thread (0b00 0101): Thread Manager is asking for the thread id of the idle thread. The Scheduler will return this id on the data_return line.

**data (32 bits):** Certain opcodes require additional information to be passed to the scheduler. This information is passed on the data signal. The Thread Manager will place this data on the data line prior to setting request high. The Thread Manager will keep data set, until the Scheduler transitions the busy signal from high to low (indicated an end of the request).

**request (1 bit):** When the Thread Manager is making a request to the scheduler, the request line will go high (read 1). This is for all opcodes except no_op. The Thread Manager will keep request high until the scheduler acknowledges by setting the busy signal high. At all other times, Thread Manager will keep request low (read 0).

**current_cpu_thread_id (8 bits):** The Thread Manager will keep this signal set to the value of its current_cpu_thread_id register. This value will be the thread id that is currently running on the cpu. It will not contain any information regarding hardware threads.

### Scheduler to Thread Manager Signals

**busy (1 bit):** The scheduler will keep the busy signal low (read 0), until it responds to a request from the Thread Manager. The Thread Manager sets request high when it is requesting information from the Scheduler through an opcode. Scheduler will set busy high (read 1) to acknowledge the request. It will keep busy high until it is done performing the request. If information is passed back to the thread manager, the scheduler will set this information on the data_return lines prior to setting busy low.

**data_return (32 bits):** Certain opcodes require the scheduler to return information to the Thread Manager. For these opcodes, the scheduler returns the needed information on the data_return signal. The Scheduler will place the data on this signal prior to setting the busy signal low. The Scheduler will keep the data on the signal until the next request by the Thread Manager.

**next_id (8 bits):** The scheduler will set the value of the thread id that is to run next (after a yield call). The value is only valid when next_thread_id_valid is high.

**next_thread_id_valid (1 bit):** When this signal is high, the Scheduler is indicating that the value on next_id is valid. A low value indicates that the value on next_id can not be trusted.
Thread ID Table Encoding : (each row)

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>Next</th>
<th>PID</th>
<th>D</th>
<th>J</th>
<th>S1</th>
<th>S0</th>
<th>E3</th>
<th>E2</th>
<th>E1</th>
<th>ERR_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>8</td>
<td>15</td>
<td>16</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Need a bit to designate HW or SW]

Note : bits 0..7 store a thread ID#, all rows forming the next available Thread ID stack. The value stored here is not related to the thread whose status is stored in this row.

Thread Status :          Auxiliary Status Codes :

<table>
<thead>
<tr>
<th>S1_S0</th>
<th>E3_E2_E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>unused</td>
<td>error in thread PID,status, or no</td>
</tr>
</tbody>
</table>

IDs available

| 0 1   | 0 0 1 |
| used, exited | THREAD_ALREADY_TERMINATED |

| 1 0   | 0 1 0 |
| used, not exited, not queued | THREAD_ALREADY_QUEUED |

| 1 1   | 0 1 1 |
| used, not exited, queued | |

| J = 0 | 1 0 1 |
| this thread has not been joined |

| J = 1 | 1 1 0 |
| this thread has been joined, |

| 1 1 1 |
| this thread is detached |

| D = 0 | 1 0 0 |
| this thread is not detached |

| D = 1 | 1 1 1 |
| this thread is detached |

ERR_BIT = 0 = no error occurred

ERR_BIT = 1 = set for all errors

<table>
<thead>
<tr>
<th>PID</th>
<th>ERR_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>this thread's Parent ID</td>
<td></td>
</tr>
</tbody>
</table>

Next = next thread in queue
FPGA Core : Block Diagram