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 scheduler sends a “RUN” command to the command register of the thread.

Error Checks

The thread’s status must be: USED, NOT_EXITED, and NOT_QUEUE
**Calls to Scheduler**

is_queue(threadID)

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

enqueue(threadID)

Scheduler will add the given thread ID to the queue, if the thread ID happens to be a SW thread. If the thread ID is a HW thread, a “RUN” command is sent to the command register of the HW thread.

**Pseudo-Code**

```pseudo
attr = threads[threadID]
if ( attr.used and !attr.exited )
    q = is_queue(threadID)
    if ( !q )
        retVal = enqueue(threadID)
        if ( retVal != error )
            return 0
        else
            return error
    else
        return error
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. The error value will be 0b000.

**set_idle_thread (REMOVE)**

Read only.

Depth = 256

**Overview**

Sets the value of the idle thread ID.

**Error Checks**

The encoded thread ID must be USED, NOT_EXITED and NOT_QUEUEED.

The scheduler had to of set the idle thread successfully.
**Calls to Scheduler**

is_queued(threadID)

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

set_idle_thread(threadID)

The scheduler will set the idle thread to threadID. If threadID can not be set as the idle thread (most likely because threadID is a HW thread) scheduler returns “1,” else it returns “0.”

**Pseudo-Code**

```plaintext
attr = threads[threadID]
q = is_queued(threadID)
if ( attr.used and !attr.exited and !q )
    retVal = set_idle_thread(threadID)
    if retVal == 1
        return error
    else
        return 0
else
    return error
```

**Return Value on Success**

0

**Return Value on Error**

If the USED, NOT_EXITED and NOT_QUEUED error check fails, the value returned is the thread’s current status with the ERR_BIT set to indicate that the operation was unsuccessful. The error value is 0x000.

If the scheduler could not thread ID as the idle thread, SCHEDULER_COUlD_NOT_COMPLETE is returned.

**get_idle_thread (REMOVE)**

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 set, a value of 512 is returned.

**Pseudo-Code**

```python
threadID = get_idle_thread()
if threadID > 256
   return error
else
   return threadID
```

**Return Value on Success**

0

**Return Value on Error**

If no thread ID is available the read operation returns SCHEDULER_ERROR, with 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 reads the ID off of the next_thread_id line and returns it immediately. The thread manager then calls the scheduler's dequeue. The scheduler removes the top thread ID from the queue. If the queue is empty, the scheduler makes no changes to the queue.

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**

```python
wait for next_thread_value == 1
threadID = next_thread_id
attr = threads[threadID]
if attr.used and !attr.exited
```


```c
return threadID
else
    return error
dequeue()
```

**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**

Returns an exception if the thread manager returns a value of a thread id that is either not used or exited.

---

**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**

```c
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

```c
if ( newThreadIDAvail )
    return new threadID
    attr = used, !exited, !joined, detached
    attr.parent = 0
    threads[threadID] = attr
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 ERROR_IN_STATUS + the ERR_BIT set.

create_thread_joinable
Read only.
Depth = 1

Overview
This register is read to retrieve an unused thread ID number in preparation for creating a new joinable thread. 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**

```plaintext
if ( newThreadIDAvail )
    return new threadID
    attr = used, !exited, !joined, !detached
    attr.parent = current_cpu_thread
    threads[threadID] = attr
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 a thread ID is not available the operation returns ERROR_IN_STATUS plus the ERR_BIT set.

**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**

```plaintext
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**

dequeue()

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, in particular if the thread is joinable or detached 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_QUEUEED. 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.

enqueue(parentID)

Scheduler will add the parentID to the queue.

**Pseudo-Code**

```plaintext
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 plus the error bit is set.

join_thread

read only, depth = 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_ALLREADY_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

dequeue(threadID)

removes the threadID from the queue.

Pseduo-Code

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_ALLREADY_TERMINATED
    else
      attrChild.JOINED = true
decompute(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

**Overview**

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.

**Error Checks**

**Calls to Scheduler**

**Return Value on Success**

**Return Value on Error**

**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**

**Psuedo-Code**

```plaintext
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.

```plaintext
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>
<tr>
<td>Soft Reset Failure</td>
<td>3</td>
</tr>
</tbody>
</table>

```plaintext
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

tm2sch_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.

**tm2sch_data (32 bits)**: Certain opcodes require additional information to be passed to the scheduler. This information is passed on the tm2sch_data signal. The Thread Manager will place this data on the tm2sch_data lines prior to setting tm2sch_request high. The Thread Manager will keep tm2sch_data set, until the Scheduler transitions the sch2tm_busy signal from high to low (indicated an end of the request).

**tm2sch_request (1 bit)**: When the Thread Manager is making a request to the scheduler, the tm2sch_request line will go high (read 1). This is for all opcodes except `no_op`. The Thread Manager will keep tm2sch_request high until the scheduler acknowledges by setting the sch2tm_busy signal high. At all other times, Thread Manager will keep tm2sch_request low (read 0).

**tm2sch_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**

**sch2tm_busy (1 bit)**: The scheduler will keep the sch2tm_busy signal low (read 0), until it responds to a request from the Thread Manager. The Thread Manager sets tm2sch_request high when it is requesting information from the Scheduler through an opcode. Scheduler will set sch2tm_busy high (read 1) to acknowledge the request. It will keep sch2tm_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 sch2tm_data lines prior to setting busy low.

**sch2tm_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 sch2tm_data signal. The Scheduler will place the data on this signal prior to setting the sch2tm_busy signal low. The Scheduler will keep the data on the signal until the next request by the Thread Manager.

**sch2tm_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 sch2tm_next_id_valid is high.

**sch2tm_next_id_valid (1 bit)**: When this signal is high, the Scheduler is indicating that the value on sch2tm_next_id is valid. A low value indicates that the value on sch2tm_next_id can not be trusted.
Thread ID Table Encoding

The Thread Manager uses 256 rows of BRAM. Each row is split between the used thread id stack (bits 0 through 7) and the state of each thread (bits 16 through 31). There is also 8 unused bits (bits 8 through 15). As a note, for any given row, the value of the thread id stack is not related to the value of the state of the thread.

The general encoding for the Thread ID Table is as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread ID</td>
<td>(not used)</td>
<td>PID</td>
<td>D0</td>
<td>J0</td>
<td>U0</td>
<td>X0</td>
<td>E3</td>
<td>E2</td>
<td>E1</td>
<td>ERR_BIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thread ID Stack

The thread manager maintains a stack of available thread ids. These are the thread Ids that could be returned to the CPU from a create_thread call. The first eight bits of the thread ID table is used to keep the stack.

Thread State

The state of the thread is maintained in bits 16 through 31 of the thread ID table. The index into the table is the thread id. For example, index 0 is for thread id 0, index 1 is for thread id 1, and so on. The meaning of each bit of a row is listed below:

*Parent ID (PID)*

Bits 16 through 23

This is the ID of the parent of this thread.

*Detached*

Bit 24

A “1” indicates this thread was created as a detached thread., a “0” indicates this thread is not detached.

*Joinable*

Bit 25

If the detached bit is 1, a “0” indicates this thread is joinable, a “1” indicates this thread is joined. If the detached bit is 0, the joinable bit has no meaning.

*Used*

Bit 26

A “1” indicates this thread is used, a “0” indicates this thread is not used.
**Exited**

Bit 27

A “1” indicates this thread is not exited, a “0” indicates this thread is exited.

**Error Value**

Bits 28 through 30

The meaning of the error bits is as follows. Some error values are call dependent.

- **0b000**: Error in thread PID, error in thread status, or no Ids available. ERROR_IN_STATUS
- **0b001**: Thread is already terminated. THREAD_ALREADY_TERMINATED
- **0b010**: Thread already queued. THREAD_ALREADY_QUEUED
- **0b011**: The scheduler could not successfully complete a requested operation. ERROR_FROM_SCHEDULER
- **0b100**: Not used.
- **0b101**: Not used.
- **0b110**: Not used.
- **0b111**: Not used.

**Error Condition**

Bit 31

A “1” indicates there was an error during the method call, check the error value for the specific error. A “0” indicates the method call completed successfully.

**Thread Manager Block Diagram**