Scheduler Design Document

Scheduler Access Registers (Bus Operations)

The following “virtual” registers are used to access a variety of functions built in to the Scheduler Module. All registers are accessed by bus transactions in the form of memory reads and writes. Each register has a 32 bit width even though the implementation may only use a subset of the full 32 bits. The registers are considered to be “virtual” because they are implemented by encoding an opcode in the address of a bus transaction and then each FPGA core (IP) decodes the opcode and additional parameters that are needed for the specific operation.

References in the following descriptions to a thread’s status refer to that thread’s entry in the scheduler attribute table and utilize the format specified in the encoding of scheduler attributes.

References in the following descriptions to “queue” refer to the ready-to-run queue, implemented using multiple linked lists (one for each priority level). This information is stored in both the scheduler attribute table and the priority data table.

References to ERROR_BITS refer to the least significant bit (LSB) of a return value that is able to have an error condition. Scheduling parameters, in the system, are 32-bit values, in which values from 0 to 127 are in the SW-valid range (i.e. can be used as priority-levels), and values greater than this are in the HW-valid range.

SET_IDLE_THREAD(tid)

• Read only

• Return values:
  • NO_ERROR = Idle thread was successfully set, no ERROR_BIT set.
  • ERROR = Idle thread was not successfully set, ERROR_BIT is set.

• Operation:
  • Idle thread can only be set to a TID that is:
    • Queued, Created, Used, and has a SW scheduling parameter.
    • If the conditions pass, then return with NO_ERROR, otherwise return with ERROR.

GET_IDLE_THREAD()

• Read only

• Return values:
• NO_ERROR = Idle thread's TID is returned with no ERROR_BIT set.
• ERROR = Invalid entry is returned with an ERROR_BIT set.

• Operation:
  • If an idle thread exists then return the idle thread's TID with no ERROR_BIT set.
  • Otherwise ERROR_BIT will be set.

GET_ENTRY(tid)
• Read only
• Return values:
  • NO_ERROR = A thread's scheduler attribute entry is returned.

• Operation:
  • The scheduler attribute entry for a thread is returned to the caller.

GET_ENCODER_OUTPUT()
• Read only
• Return values:
  • NO_ERROR = Output of the priority encoder is returned.

• Operation:
  • The output of the priority encoder is returned to the caller.

GET_SCHEDPARAM(tid, sched_param)
• Read only
• Return values:
  • NO_ERROR = A thread's scheduling parameter is returned.

• Operation:
  • The scheduling parameter for the thread is returned to the caller.

CHECK_SCHEDPARAM(tid, sched_param)
• Read only
• Return values:
  • ERROR = ERROR_BIT is not set, scheduling parameter is valid for given thread.
• NO_ERROR = ERROR_BIT is set, scheduling parameter is invalid for given thread.

• Operation:
  • The scheduling parameter for a thread is examined as well as the thread's attributes:
    • If the thread is queued, then the scheduling parameter must be in the SW-valid range.
      • If not in SW-valid range, then the ERROR_BIT will be set.
      • If the thread is not queued, then the scheduling parameter can have any value.

**SET_SCHEDPARAM(tid, sched_param)**

• Write only

• Return values:
  • NONE, write operations have no return values.

• Operation:
  • The scheduling parameter for a thread is set by the caller
    • If the scheduling parameter is in the HW-valid range, then the scheduling parameter is stored (no queue manipulations are needed).
    • If the scheduling parameter is in the SW-valid range, then queue manipulations are required because the priority of the thread is going to be changed:
      • If the thread is not queued, then only the new priority is set.
      • If the thread is queued, then it is unlinked from its old priority queue, its priority is changed, and then it is relinked into its new priority queue.
      • Additional error-checking is done to ensure that the thread is created, used, and that the calling tid is either the thread itself or the thread's parent.

**TOGGLE_PREEMPTION(on_or_off)**

• Write only

• Return values:
  • NONE, write operations have no return values.

• Operation:
  • The preemption enable/disable bit is toggled based on the calling parameter.
Thread Manager Interface (TM Operations)

The Thread Manager (TM) interacts with the Scheduler through a dedicated interface so as to provide a way for thread management operations to have scheduling side effects. The interface is composed of the following set of signals:

- **Access to B-Port of TM's BRAM.**
  - Allows the Scheduler to read TM's BRAM for error checking purposes.
  - Composed of address, data_in, data_out, enable, and write-enable signals.

- **Reset signals.**
  - Allows the TM to reset the scheduler during run-time.
  - Composed of reset and reset_done signals.

- **TID signals.**
  - Contains information as to what thread is currently running on the CPU, what thread is going to run on the CPU next.
  - Composed of current_cpu_tid, next_cpu_tid, next_tid_valid signals.

- **Operation Interface.**
  - Signals to allow the TM to request scheduling operations.
  - Composed of opcode, data_in, request, busy, and data_out signals.

The operations available through the TM Interface include:

**ENQUEUE(tid)**
- Operation: Introduce a thread to the Scheduler.
  - SW threads: added into the ready-to-run queue based on their priority-levels.
  - HW threads: started by the Scheduler.

**DEQUEUE()**
- Operation: Remove the next thread to run on the CPU from the ready-to-run queue and calculate a new next thread to run.
  - Unlinks the next thread to run from the ready-to-run queue and then finds the thread in the system with the best priority-level and assigns it to be the next thread to run.

**IS_QUEUED(tid)**
• Operation: Returns the queue_bit for a given thread.
  • Reads the queue_bit out of the scheduler attribute table for the given thread and returns it to the TM.

**IS_EMTPY()**

• Operation: Returns whether the ready-to-run queue is empty or not
  • Reads the priority encoder input and returns 1 (true) if it is zero, otherwise returns 0 (false).

**SOFT_RESET()**

• Operation: Returns the Scheduler back to it's initial (boot-up) state.
• Resets all signals, registers, and BRAM contents to initial state.

**Scheduler Attribute Table Format**

The image below contains the format of the scheduler attribute table (THREAD_DATA) that the Scheduler manages. This table is implemented using BRAM and is indexed by thread ID.

<table>
<thead>
<tr>
<th>Field</th>
<th>Width</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q?</td>
<td>(1-bit)</td>
<td>1=Queued, 0=Dequeued</td>
</tr>
<tr>
<td>N0:N7</td>
<td>(8-bit)</td>
<td>Next Pointer, ThreadID index into BRAM</td>
</tr>
<tr>
<td>L0:L6</td>
<td>(7-bit)</td>
<td>Scheduling Priority Level</td>
</tr>
<tr>
<td>P0:P7</td>
<td>(8-bit)</td>
<td>Previous Pointer, ThreadID index into BRAM</td>
</tr>
</tbody>
</table>

The scheduling parameter is a 32-bit value and is stored in an adjacent BRAM (PARAM_DATA) that is also indexed by thread ID. An additional BRAM (PRIORITY_DATA) is used to store information pertaining to the linked lists associated with each priority-level. This BRAM is indexed by priority-level and contains the head and tail pointer information for each priority-levels queue.

**Priority Encoder**

A simple priority encoder is used to select the best priority-level in the system without having to traverse a linked-list stored in BRAM. The priority encoder's input has a single bit per priority level: if the bit is 1 then there are threads resident in that priority-levels queue, if the bit is 0 then there are no threads in that priority-levels queue. The input to the priority encoder is managed during all scheduling operations so that it's input always reflects the current state of the ready-to-run queue. The encoder, when enabled, finds the lowest (i.e. best) priority level that has a 1 as its encoder input bit. The encoder is implemented as a finite-state machine (FSM) that can deterministically and predictably complete in 4 clock cycles. This constant-time execution time
of the priority encoder along with ready-to-run queues partitioned by priority-levels allows for constant-time “searching” of the best priority-level thread in the system. This means that enqueue and dequeue operations are constant, and do not vary with queue length or any other factors except clock speed.

**Scheduler Block Diagram**

The following image is a block diagram of the Scheduler Module. The block diagram also includes the TM Interface signals.