Hardware Thread Interface Specification and Implementation Document

Purpose and Overview

The purpose of the hardware thread interface (HWTI) is to provide an abstraction layer between the Hybridthreads System and a local thread running in the FPGA fabric. A thread that is running in the FPGA fabric is collectively referred to as a Hardware Thread (HWT).

The abstraction layer of the HWTI exists in two directions. The hthreads system must call the HWTI to interact with the HWT. Alternatively, the HWT must use the HWTI to interact with the system, including using the bus for reads and writes. From a HWT's perspective, the HWTI is the mechanism for using a system call, much like a software thread calls the hthread library to perform system level functions.

This paper will first describe the specifications of the HWTI, detailing the calls the system may make, and the calls the thread may make to the system. Then, a description of an implementation of the specification will be given. This implementation has been fully tested as a hardware thread interacting with the hthreads system. Finally, details of the implementations performance, including slice count and speed are given.

The HWT is physically composed of three parts. First is the connection to the bus, specifically the IPIF. The HWT may be connected to either the OPB or PLB. The Second portion of the HWT is the HWTI. There is only one implementation of the HWTI, it may be used for either bus or any user logic. The user logic, formally the Hardware Thread User Logic (HWTUL), is the final piece. The HWTUL contains the “program” the HWT runs.

System Level Application Programming Interface

The system level API consists of a set of five memory mapped registers. The register names are thread_id, verify, command, status, argument, and result. The specification of each of these registers, plus the protocol on how they should be use, are below.

All registers, since they are accessed directly off of the OPB or PLB, are 32 bits.

thread_id

Overview

The thread_id register tells the HWTI what its thread id is. The thread id is assigned by the system at runtime. Specifically, when a thread is created, the system asks the Thread Manager for a thread ID, the system then assigns the thread ID to this register.

The thread_id register is
both readable and writable.

**Protocol**

On system start up, and after a reset, the thread_id is set to 0. When a write occurs to the thread_id register, the status changes from NOT_USED to USED. The thread_id may be written to only when the status register reads NOT_USED. With all other statuses, writing to this register has no effect. Bits 24 to 31, of the system bus data lines, are used to set the thread_id of the HWT. The thread id must be non-zero, consequently the minimum thread id is 1. The maximum thread id is 255.

The thread_id register may be read from at anytime. The read operation does not have any side effects.

The thread_id must be set prior to RUN being issued to the command register.

**verify**

**Overview**

The verify register is read by the system to indicate that a hardware thread exists at this address location.

The verify register is read only.

**Protocol**

Each HWTI should be assigned a unique verify number. As a note, this should be the only difference between two HWTIs. This unique number, when read by the system, tells the system that there is a HWTI at the expected address, and that the HWTI, along with the corresponding HWTUL, is also correct.

The value of the verify register is a generic, and may be set at synthesis time.

**command**

**Overview**

The command register is written to by the system to tell the HWT to RUN, RESET, STEP, or IDLE. A RUN command serves two purposes. First to tell the HWT to start executing, second, if the HWT is waiting on a mutex, to wake up and check the mutex. The RESET command tells the HWT to reset all variables (both in the HWTI and HWTUL) and return to a NOT_USED status (see status register).

The STEP and IDLE command are for debugging. The IDLE command tells the HWT to stop what it is doing and remain in its current state without changing any of its internal state. The STEP command tells the HWT (specifically the HWTUL) to proceed to the next state and then IDLE.

The command register may be read or written to. However, reading this register is implementation specific. In general reading this register will return the last command issued.
Protocol

A RUN may be issued to the HWTI only if the status register is either USED, BLOCK, or IDLE. Issuing a RUN at any other time has no effect on the HWT.

Issuing a RUN while the status is USED changes the status to RUNNING. This will also change the user_status from RESET to RUN, triggering the HWTUL to start executing its state machine.

Issuing a RUN while the status is BLOCK, tells the HWTI to recheck the operation causing the block (mutex lock, or interrupt associate), if successful change the user_status from ACK to RUN, triggering the HWTUL to continue on with its state machine.

Issuing a RESET at anytime sets the status register to NOT_USED, the thread_id register to zero, and the user_status register to RESET. The HWTUL is responsible for resetting any variables it may use. To insure the HWT is in an initialized state, the system should RESET at start up. The system must also issue a RESET if, after the HWT exits, the system wants to reuse the HWT component as a new thread.

The IDLE command is used to temporarily halt the HWTUL's state machine. This should only be used by the system for debug purposes. To restart the HWTUL's state machine, the system must issue a RUN or STEP command. IDLE will only have an effect on the HWT if the status is either RUNNING or BLOCKED. When issued, the IDLE command changes the status to IDLING.

The STEP command is used to advance the HWT to the next state (assuming the conditions exist to advance to the next state) and then return to IDLE. STEP may only be issued when the status register is IDLING. After STEP, the status remains at IDLING.

When an IDLE command is issued, the HWTI sets the user_status to PAUSE, and the status register to IDLING. The HWTUL's state machine must remain in its current state until the HWTI changes the user_status back to RUN or ACK (the user_status prior to idling the HWT). When STEP is issued, the HWTI changes the user_status to either RUN or ACK for one clock cycle, and then returns the user_status to PAUSE.

The command register may be read from at any time, in general returning the last command the HWT received. However, the implementation of the HWTI is free to return any value during a read. This call has no side effect.

The binary values of each command are as follows:

- RUN (0001)
- RESET (0010)
- IDLE (0100)
- STEP (1000)

Bits 28 to 31, of the system bus data lines are read to determine the value of the command.
**status**

*Overview*

The status register is a read only register, indicating to the system the state the HWT is in. It should only be used for debugging purposes. The possible states the HWT may be in are RUNNING, BLOCKED, IDLING, EXITED, EXITED_WITH_ERROR, USED, NOT_USED.

*Protocol*

The HWTI will report each state for the following conditions. Binary values are in parenthesis.

- **NOT_USED (0000 0000):** This is the state of the HWT on system start up and after a RESET command. No other commands have been issued.

- **USED (0000 0001):** This is the state after the thread_id register has been written to, but before a RUN command has been issued.

- **RUNNING (0000 0010):** The thread_id register has been populated, the system issued a RUN command, the HWT is not waiting on a mutex or other blocking type of operations, and the HWT has not exited. Generally means that the HWTUL is executing its state machine (doing useful work).

- **BLOCKED (0000 0100):** May transition to a BLOCKED state from a RUNNING state. Occurs when the HWTUL issues a REQUEST_LOCK operation, and the HWTI is waiting to obtain the lock. Once the lock is obtained, status transitions back to RUNNING. Generally means the HWT is waiting to obtain a mutex.

- **EXITED (0000 1000):** The HWT will transition to this state after the HWTUL is done executing. It indicates that the value in the result register is valid (specific to the meaning of the thread).

- **EXITED_WITH_ERROR (0020 0000):** The HWT will transition to the state, upon command from the HWTUL. This state indicates that the HWTUL could not complete its execution as expected, due to an error (for example, divide by zero).

- **IDLING (0001 0000):** If the system issues an IDLE command, the status transitions to IDLING. HWT will return to its previous status, either RUNNING or BLOCKED, after a RUN command.

The argument register may be read from at any time without side effect. Writing to this register has no effect.

**argument**

*Overview*

Consistent with the pthreads protocol, when a thread is created by the system, the system may pass one argument into the thread. The argument register is used to allow the system to pass in this argument. If used, the system must set the argument after setting the thread_id register and prior to issuing the RUN command.

The meaning of the value of the argument register is thread specific. Generally it is an address pointer to data the thread is to operate on. Setting the argument register is not required.
The argument register is readable at any time, and writable only when the status is USED.

**Protocol**

The system may write to the argument register only if the status register is USED. This means that the system, when it wants to start the HWT must first issue a RESET command, set the thread_id register, set the argument register (if used), and then issue a RUN command.

Upon a RUN command, the HWTI stores the argument register into the user_result register. It does this prior to changing the user_status from RESET to RUN. The HWTI will maintain this value in the user_result register until the HWTUL issues its first non-NOOP opcode.

**result**

**Overview**

When a thread is created, runs, and then exits, the thread has the option of passing results back to the parent thread. As a note, this is only for joinable threads, results have no meaning in detached threads. To pass back results to the parent, the HWT places the value in the result register. For consistency with the pthreads interface, the result value should be a pointer, although this is not required.

The system may read the result register at any time, although, it only has meaning when the status register reads EXITED.

**Protocol**

When the HWTUL stops, and is ready to issue an EXIT command in the user_opcode register, the HWTUL must place any results it wants to report in the user_argument_data register prior to issuing the EXIT opcode. Once the HWTI sees the EXIT command, it will copy the value of the user_argument_address register into the result register.

The system may read from this register at anytime without side effect. Writing to this register has no effect.

**User Logic Application Programming Interface**

Each of the registers in the User Logic Application Programming Interface (the interface between the HWTI and HWTUL layers) may only be accessed by the HWTUL later. They system layer has no direct access to their values.

The width (number of bits) of each register is given in the Protocol section.

**user_status**

**Overview**

It is used by the HWTI layer to give instructions to the HWTUL layer, as well as provide a handshake mechanism when the HWTUL layer issues commands to the user_opcode register. There are four possible values to the user_status register, they are RESET, RUN, ACK, PAUSE.
**Protocol**

This register is 4 bits. Each bit represents a status. The status, their binary value, and their meaning are as follows:

- **RESET (0001):** This is the initial value of user_status on power up. HWTI will also change to this state if the system issues a RESET to the command register. HWTI will remain in this state until the system issues a RUN to the command register, at which time HWTI will change the user_status register to RUN.

- **RUN (0010):** This value has two meanings, depending on the previous status. If the previous status was RESET, RUN tells the HWTUL to start executing from the top of its state machine. If the previous status was ACK, it signifies to the HWTUL layer that the requesting opcode is complete, and any result is stored in the user_result register.

- **ACK (0100):** If the HWTUL layer issues a command to the user_opcode register, the HWTI layer responds by changing the user_status to ACK. The HWTI will also read the user_argument registers for any data and addresses it is to work with. At this point, the operation has started, but not yet complete, the values in the user_result register are not valid. The HWTUL layer will know that the command is complete, and the user_result value valid, when the user_status register changes back to RUN.

- **PAUSE (1000):** If the system issues an IDLE to the command register, the user_status register changes to PAUSE. It signifies to the HWTUL layer to remain in its current state, until the status register changes back to RUN or ACK (the system is prohibited from issuing an IDLE while the HWT is not running). The HWTI later will always return the user_status register to the status it was prior to changing it to PAUSE. The HWTUL should not change its internal state while the user_status is PAUSE.

Writing to this register has no effect. Reading from this register has no side effect.

**user_argument_one / user_argument_two**

**Overview**

Some of the operation codes the HWTUL layer may issue to the HWTI layer require additional information to complete. The user_argument_one, and user_argument_two registers allow for the passing of this data. They are writable by the HWTUL layer when the user_status is RUN. Writing to these registers is prohibited at other times. This means that the arguments must be set prior to the HWTUL layer issuing a command to the user_opcode register.

In general, knowing what argument to place in which register, is equivalent to knowing the parameter order for the equivalent hthread call. For example, hthread_join(hthread_t, void**), takes two arguments (note that hthread_join is not currently supported). The hthread_t argument is placed in the user_argument_one register. The void** argument is placed in the user_argument_two register.

It is expected, as the HWTI grows to support more hthread system calls, additional registers will be needed. These future registers will be named user_argument_three through six.

**Protocol**

The user_argument_data register is 32 bits wide.
The HWTUL layer may write to this register when the user_status is RUN.

The HWTI layer will read the address, and take appropriate action, on the opcodes READ, WRITE, EXIT.

The HWTUL layer may read from this register to view the last value is stored in it. Reading this register has no side effect.

**user_opcode**

**Overview**

When the HWT is RUNNING, the HWTUL layer may request services via the HWTI layer. The user_opcode register is the mechanism allowing the HWTUL layer to request these services.

The goal is to provide all services to a hardware thread that a software thread has. These services will be performed via the user_opcode. The Protocol section lists the services currently supported. A NOOP operation was added to assist in the handshaking protocol to request a service.

**Protocol**

The user_opcode register is 8 bits.

Only one user_opcode may be issued at a time.

The HWTUL layer may issue a new opcode (request a service) only when the user_status register reads RUN. The one exception is when the HWTUL layer changes the opcode to NOOP, which can only be done when the user_status is ACK. At all other times the opcode is ignored.

The HWTUL layer must maintain the value of the opcode (service request), and any user_argument registers, until the user_status value changes from RUN to ACK. Once the status is changed to ACK, the HWTUL must change the opcode to NOOP, the user_argument registers do not have to be changed. The HWTI layer will indicate that the operation is complete by changing the user_status back to RUN. At this time, any value in the user_result register is valid.

The opcodes, their binary value, and their implementation meaning are as follows:

- **NOOP (0000 0000):** The HWTUL is not requesting any service from the HWTI layer.
- **HTHREAD_EXIT (0000 0001):** The HWTUL has finished executing and is requesting to call exit_thread (on the thread manager), any results are placed in user_argument_address.
- **HTHREAD_EXIT_ERROR (0000 1001):** The HWTUL can not continue executing, due to an unchecked error (for example divide by zero). A call to exit_thread (on the thread manager) is still made. The status of the HWT is changed to EXITED_WITH_ERROR.
- **READ (0000 0010):** The HWTUL is requesting to read the value of the memory addressed in user_argument_address.
- **WRITE (0000 0011):** The HWTUL is requesting to write the data in user_argument_data to the address user_argument_address.
- **HTHREAD_SELF (0000 0100):** Returns the value of the thread_id register to the user_result register.
• HTHREAD_YIELD (0000 0101): The hthread_yield call normally allows a CPU bound thread to give up the CPU. Since the HWT is not running on the CPU, the hardware implementation immediately ACKs and returns to RUN. This call is implemented for completeness only.

• HTHREAD_MUTEX_LOCK (0000 0110): The HWTUL is requesting to lock the mutex indicated in user_argument_data.

• HTHREAD_MUTEX_UNLOCK (0000 0111): The HWTUL is requesting to unlock the mutex indicated in user_argument_data.

• HTHREAD_INTRASSOC (0000 1000): Allows the HWT to associated with an interrupt specified in the user_argument_data register.

**user_result**

**Overview**

This register has two uses, first when the HWT is created, and initial arguments are passed in, and second, when the HWTUL layer requests certain services, and the HWI layer passes values back.

When the HWTUL layer makes a service request, if the particular operation returns a value (READ for example), the result is stored in this register prior to the user_status changing back to RUN.

**Protocol**

The user_result register is 32 bits.

The HWI layer will place the value of the service request (opcode) in user_request, and then transition from ACK to RUN. The HWI layer will keep this value constant until the next opcode request.

The HWTUL layer may read this register without side effects. Writing to this register is not permitted.

**State Machine Implementations**

The following two sections detail the implementation of the HWI. The two sections above, give the specifications of the HWI, without regards to how it should be implemented. With all such designs, the implementation is independent of the requirements. And thus, the details that follow are only one, of an infinite set of possible, implementations. Could I have used the word “implement” more times in this paragraph?

The HWI runs off of three state machine. Loosely speaking, the first state machine controls and monitors the registers attached to the bus, the second state machine controls and monitors the registers associated with the user logic API, and the third state machine supervises, assures communication between the first two, and fulfills all the system calls from the HWTUL.

**Process Descriptions**

The VHDL code for HWI is divided up into six processes. These processes, and their states (if applicable) are detailed below.
CYCLE_PROC
The purpose of the CYCLE_PROC process is to count the number of clock cycles during a slave bus transaction. This count is used by the CYCLE_CONTROL process.

CYCLE_CONTROL
The CYCLE_CONTROL process has two purposes. First to suppress the IP2Bus_ToutSup line if the bus transaction takes longer than 4 clock cycles. The second is to maintain the value of the IP2Bus_MstBE, Retry, Error, and PostedWrInh lines. These lines are either not used, or have a constant value (from the point of view of the HWT).

HWTI_STATE_PROCESS
The primary purpose of the HWTI_STATE_PROCESS process is to physically assign the values to each and all of the registers in the HWTI. Since all of the registers get updated at the same time, the chances of a timing error is greatly minimized. Furthermore, this type of process is needed by the Xilinx synthesis tools to recognize the state machines in the VHDL entity.

The second purpose is to reinitialize the state machines when either the Bus2IP_Reset line is raised on the bus, or the system writes a RESET command to the HWTI. The details of the reset process is described in the System State Machine sub-section.

HWTI_SYSTEM_STATE_MACHINE
The states of the System State Machine are listed below. Along with their description.

- START: The initial state after power up and reset. Initializes the system level registers. Transitions to the IDLE state.
- IDLE: Responds to all reads and writes from the system bus, as well as requests from the Controller State Machine. Transitions to the remaining states if operation requires more than a single clock cycle.
- COMMAND_RESET_INIT: On a RESET command, acknowledges the bus transaction. Transitions to the COMMAND_RESET_END_BUS_TRANSACTION_WAIT state.
- COMMAND_RESET_END_BUS_TRANSACTION_WAIT: Once the chip enable goes low, changes the system_command to RESET, which starts the reset process throughout the HWT.
- COMMAND_RUN_INIT: Checks to make sure the system may issue a RUN command. If allowed, the command register is updated.
- END_BUS_TRANSACTION: Performs the acknowledge to the bus. Transitions to the END_BUS_TRANSACTION_WAIT state.
- END_BUS_TRANSACTION_WAIT: Waits for the bus to lower the read or write chip enable line. Transitions to the IDLE state.

HWTI_USER_STATE_MACHINE
The states of the User State Machine are listed below. Along with their description.

- START: The initial state after power up and reset. Initializes the user level registers. Transitions to the IDLE state.
• IDLE: General state that waits for requests from the HWTUL. When a request is read from
the user_request register and the user_status is RUN, Transitions to the ACK_REQUEST
state. State also waits for communication from the Controller State Machine. From a
practical point of view the IDLE state is either waiting for a request from the HWTUL or the
Controller State Machine, never both.

• ACK_REQUEST: Changes the user_status register to ACK. Transitions to IDLE.
• RUN: Updates the user_status to RUN. Transitions to WAIT_TWO_CYCLES.
• WAIT_TWO_CYCLES: Transitions to WAIT_ONE_CYCLE.
• WAIT_ONE_CYCLES: Transitions to IDLE.

**HWTI_CONTROLLER_STATE_MACHINE**

The states of the Controller State Machine are listed below. Along with their description.

• START: The initial state after power up and reset. Initializes the user_result, user_request,
  system_result, system_request registers. Transitions to the NOT_USED state.

• NOT_USED: Waits in this state until the system sets the thread id. Requests the system status
  be updated to USED. Transitions to the NOT_USED_WAIT state.

• NOT_USED_WAIT: Waits until the System State Machine updates the system status to
  USED. Transitions to the USED state.

• USED: Waits until the system issues a RUN command. Requests the System and User State
  Machine change their status registers to RUN. Sets the user_result register to the value of the
  system's argument register. Transitions to the USED_WAIT state.

• USED_WAIT: Waits until both the System and User State Machine update their status
  register. Transitions to the RUNNING state.

• RUNNING: Monitors the user_status register for a change to ACK. This implies the
  HWTUL made a system call. If so, determine the call and transition to the appropriate state.

• RUNNING_WAIT: Waits until the the system status is RUNNING and the user_status is
  RUN. Transitions to RUNNING.

• HTHREAD_EXIT_INIT: Sets the appropriate IP2Bus signals to make the call to the Thread
  Manager. Transitions to the HTHREAD_EXIT_WAIT_FOR_ACK state.

• HTHREAD_EXIT_WAIT_FOR_ACK: Maintains the appropriate IP2Bus signals until the
  bus acknowledges the request. Transitions to the HTHREAD_EXIT_WAIT state.

• HTHREAD_EXIT_WAIT: Waits in this state forever, or until a RESET command.

• READ_INIT: Sets the appropriate IP2Bus signals to do a bus master read. Transitions to the
  READ_WAIT_FOR_ACK state.

• READ_WAIT_FOR_ACK: Maintains the appropriate IP2Bus signals until the bus
  acknowledges the request. Requests the User State Machine update the status to RUN.
  Transitions to the READ_FINISH state.

• READ_FINISH: Deasserts the IP2Bus signals. Transitions to the RUNNING_WAIT state.
• WRITE_INIT: Sets the appropriate IP2Bus signals to do a bus master write. Transitions to the WRITE_WAIT_FOR_ACK state.

• WRITE_WAIT_FOR_ACK: Maintains the appropriate IP2Bus signals until the bus acknowledges the request. Requests the User State Machine update the status to RUN. Transitions to the RUNNING_WAIT state.

• MUTEX_LOCK_REQUEST: Sets the appropriate IP2Bus signals to do a bus master read to the Mutex Manager. Transitions to the MUTEX_LOCK_WAIT_FOR_ACK state.

• MUTEX_LOCK_WAIT_FOR_ACK: Maintains the appropriate IP2Bus signals until the bus acknowledges the request. Checks the Bus2IP_Data lines for a successful lock. Either Requests the User State Machine update the status to RUN or BLOCKED. Transitions to either the RUNNING_WAIT or MUTEX_LOCK_REQUEST_WAIT state.

• MUTEX_LOCK_REQUEST_WAIT: Waits until the system status is changed to BLOCKED. Transitions to the MUTEX_LOCK_CHECK_WAIT_FOR_RUN state.

• MUTEX_LOCK_CHECK_WAIT_FOR_RUN: Waits in this state until a RUN command is issued. Transitions to the MUTEX_LOCK_CHECK_WAIT_FOR_RUN_WAIT state.

• MUTEX_LOCK_CHECK_WAIT_FOR_RUN_WAIT: Waits until the the system status is RUNNING and the user_status is RUN. Transitions to RUNNING.

• MUTEX_UNLOCK_REQUEST: Sets the appropriate IP2Bus signals to do a bus master read to the Mutex Manager. Transitions to the MUTEX_LOCK_WAIT_FOR_ACK state.

• MUTEX_UNLOCK_WAIT_FOR_ACK: Maintains the appropriate IP2Bus signals until the bus acknowledges the request. Requests the User State Machine update the status to RUN. Transitions to the RUNNING_WAIT state.

**System State Machine**

The System State Machine has three general purposes. The first is to control all interaction with the OPB or PLB bus. Second, to maintain the values of the System Level API registers (thread_id, command, argument, status, and result). The final purpose is to act upon changes given to the HWTI by the system.

The states of the System State Machine are detailed in the Process Description sub section (above), and will not be repeated. The following is discussion of the design decisions used to implement the System State Machine.

**Communication Between System and Controller State Machines**

In VHDL, only one process may write to a register. This presents a problem in a multi-state machine entity, like the HWTI. To overcome this problem, each of the state machines “owns” a subset of all the registers in the HWTI.

In the case of the System State Machine, it owns the thread_id, verify, status, command, and argument registers. Or rather all of the system level API registers except one, the result register. The system result register is owned by the Controller state machine.

Depending on the current status and interaction with either the system or the HWTUL, either the System State Machine or the Controller state machine may have a need to change the value of
the status register. The System State Machine needs to initialize the status register to NOT_USED. The Controller State Machine, needs to be able to set the status register to any of the other possible states, USED, RUN, EXITED, EXITED_WITH_ERROR, BLOCKED. In order to facilitate this inherit violation of VHDL, the Controller State Machine communicate to the System State Machine via a system_request register. This register conveys commands for the System State Machine to follow.

The system_request register may take on one of six values, relating to the five status the Controller State Machine wants to change the status to. The sixth value is a no operation request. These are detailed below.

- CHANGE_STATUS_TO_USED: The Controller State Machine is asking the System State Machine to change the status register to USED.
- CHANGE_STATUS_TO_RUN: The Controller State Machine is asking the System State Machine to change the status register to RUN.
- CHANGE_STATUS_TO_EXIT: The Controller State Machine is asking the System State Machine to change the status register to EXITED.
- CHANGE_STATUS_TO.EXIT_ERROR: The Controller State Machine is asking the System State Machine to change the status register to EXITED_WITH_ERROR.
- CHANGE_STATUS_TO_BLOCK: The Controller State Machine is asking the System State Machine to change the status register to BLOCKED.
- NOOP: The Controller State Machine is not requesting a status change from the System State Machine at this time.

The Controller State Machine, which owns the system_request register will maintain one of the CHANGE_STATUS values until the System State Machine complies. The Controller State Machine then changes the system_request register to NOOP.

**HWTI Reset**

The process of resetting the HWTI involves the reinitializing of three state machines, plus the communication of the reset to the HWTUL. The immediate problem of this process is that if the state machine resets too soon, either the HWTUL will not get the signal, or the bus transaction (initiated by system via a write to the command register) will end abruptly.

To overcome this problem, handles the bus transaction or a reset write differently that other bus transactions. Specifically the state machine will acknowledge and end the bus transaction, prior to resetting. Upon completion of the bus transaction, the HWTI resets itself. The HWTUL is reset at the same time as the User State Machine.

**Additional Memory Mapped Registers**

During the writing of the HWTI, a number of additional memory mapped registers were identified as being needed for this implementation. These fall into three groups. Registers needed for debugging, registers needed for being a bus master, and registers needed for responding to the system for unknown addresses. They are as follows:

- DEBUG_SYSTEM: This register is read by the system to learn the state of the System State Machine.
- **DEBUG_USER**: This register is read by the system to learn the state of the User State Machine.

- **DEBUG_CONTROL**: This register is read by the system to learn the state of the Control State Machine.

- **MASTER_READ**: This register is used when the HWT is doing a read operation on the bus. Specifically the IPIF writes to this register with the data from the read. The IPIF will continue to write from this register, until the HWTI acknowledges (as with any other write operation).

- **MASTER_WRITE**: This register is used when the HWT is doing a write operation on the bus. Specifically the IPIF reads this register to know what data to write. The IPIF will continue to read from this register, until the HWTI acknowledge (as with any other read operation).

- **OTHERS**: This is a virtual register of sorts. When ever a read or write operation comes into the HWTI, that is not addressed to one of the known register addresses, the OTHERS register responds. In the case of a read, it returns all zeros. In the case of a write, it does nothing.

### Unimplemented Specifications

During the implementation of the HWTI, it was decided that including the STEP and IDLE features would be cumbersome. To implement these features would require additional states in each of the state machine, as well as the necessary logic that follows. Furthermore, from a system's point of view, these features seem difficult to use, since it is impossible for the system to IDLE the HWT during a specific state. Given all of these reasons, the STEP and IDLE commands were not implemented.

Also, the command register, when read, may not return the last command given to the HWTI. There are two cases of this. First, after a reset, and before any command is given, the command register would return a INIT command. This is to prevent the HWTI from continuously resetting itself (if it maintained the RESET command). Second, if the HWT is waiting on a mutex, the command register will read INIT as well. This is because the HWTI is waiting for a RUN command from the Thread Manager. If the command register is not changed, on the next state the HWTI will think that a RUN command came in, and start executing again.

### User State Machine

The User State Machine has two general purposes. The first is to control all interaction with the HWTUL. Second, to maintain the values of the User Level API registers (user_status, user_argument_one, user_argument_two, user_opcode, and user_result). The fulfillment of the system call requests is left to the Controller State Machine.

The states of the User State Machine are detailed in the Process Description subsection (above), and will not be repeated. The following is discussion of the design decisions that went into the implementation of the User State Machine.

### Communication Between User and Controller State Machines

As mentioned in the System State Machine subsection, each of the state machines owns a subset of all the registers in the HWTI. The User State Machine specifically owns the user_status,
user_argument_one, user_argument_two, and user_opcode registers. The Controller State Machine owns the user_result register.

In a close analogy to the status register with the System State Machine, both the User and Controller State Machine have a need to update the value of the user_status register. The User State Machine needs to update the user_status register during a reset, and to acknowledge a request by the HWTUL. The Controller State Machine needs to update the user_status register, when it has fulfilled the system call made by the HWTUL. To enable this, the Controller State Machine will request and update of the user_status register via the user_request register.

The user_request register may take on one of two values. Either update the user_status register to run, or a no operation request. The details of the user_request values are below.

- CHANGE_STATUS_TO_RUN: The Controller State Machine is asking the User State Machine to change the status register to RUN.
- NOOP: The Controller State Machine is not requesting a status change from the User State Machine at this time.

The Controller State Machine, which owns the system_request register will maintain the CHANGE_STATUS_TO_RUN value until the User State Machine complies. The Controller State Machine then changes the system_request register to NOOP.

**HWTUL Reset**

During a reset, the User State Machine is responsible for not only resetting itself, but also signaling to the HWTUL to reset. Upon a reset, the User State Machine enters the START state, which resets this state machine. Also in this state the user_status register is changed to RESET. This signals to the HWTUL to reset itself. The user_status register will remain in this state until a new RUN command is issued from the system.

**Two Cycle Wait After Run**

When the User State Machine, via a request from the Controller State Machine, changes the user_status to RUN, the User State Machine will wait two additional clock cycles before accepting a new request from the HWTUL. This is to ensure that there are no timing issues between the HWTUL and HWTI.

**Controller State Machine**

The Controller State Machine is the largest and most complicated of the three state machines in the HWTI. Its primary purpose is to fulfill the system calls from the HWTUL. In order to achieve this it needs to be able to read and write to the bus. The Controller State Machine therefor controls the bus master signals. Finally the Controller State Machine is in charge of the inter state machine communication.

The states of the Controller State Machine are detailed in the Process Description sub section (above), and will not be repeated. The following is discussion of the design decisions that went into the implementation of the System State Machine.

**Controlling System Status**

The Controller State Machine, in conjunction with the System State Machine is responsible for maintaining the system status register. Most of the changes to the status register are requested by
the Controller State Machine. This is because the Controller State Machine was tasked with enforcing the rules of when the status may change (see the command and status subsections in the System Level Application Programming Interface section).

The Controller State Machine enforces these rules, and subsequently updates the status register, by monitoring the thread_id, and command system registers, as well user_request register. By monitoring the thread_id register, the Controller State Machine knows when to change the status to USED. By monitoring the command register, it knows when to change the status to RUNNING. When the HWTUL performs a hthread_exit or hthread_mutex_lock request, the Controller State Machine knows when to change the status to either EXITED or BLOCKED respectively.

**Controlling User Status**

The Controller State Machine, in conjunction with the User State Machine is responsible for maintaining the user_status register. The User State Machine updates the user_status after receiving a request from the HWTUL. Specifically, when a non-NOOP request is placed in the user_request register, the User State Machine changes the status to ACK. At this point, the Controller State Machine fulfills the request. Once the specified system call is fulfilled, the Controller State Machine requests to the User State Machine to change the status back to RUN.

**hthread_exit Implementation**

The hthread_exit opcode is implemented by first making an exit_thread call to the Thread Manager. This call requires a bus master read to the exit_thread register on the Thread Manager, with the HWI's thread id embedded in the address. Upon acknowledgment from the Thread Manager the HWTI status changes to EXITED.

The Controller State Machine does not check the return status of the Thread Manager for either a success or failure signal. It assumes that the call was successful.

The only difference between the EXIT and EXIT_WITH_ERROR implementation is the value of the status register. Both calls by the HWTUL result in a call to the Thread Manager to exit.

**Read Implementation**

A READ opcode by the HWTUL is implemented by performing a read master bus transaction. Specifically the Controller State Machine sets the IP2Bus_addr lines to the value of the user_argument_one register. Upon completion of the bus transaction, the user_result register is updated to the value of the Bus2IP_data lines.

**Write Implementation**

A WRITE opcode by the HWTUL is implemented by performing a write master bus transaction. Specifically the Controller State Machine sets the IP2Bus_addr lines to the value of the user_argument_one register, and the IP2Bus_data lines to the value of the user_argument_two register. The user_result register is not updated upon completion of the bus transaction.

**hthread_self Implementation**

The hthread_self opcode is implemented by setting the value of the user_result register to the value of the thread_id register. Because the thread_id register is 8 bits, and the user_result register is 32 bits, the thread id is padded with 0's. The user_status register is returned to RUN on the clock cycle after the user_result register is updated.
**hthread_yield Implementation**

The hthread_yield opcode is, from a hardware thread point of view, useless. The hthread_yield system call is meant for a software thread to temporarily give up the CPU. Since a hardware thread does not run on the CPU, it can not give it up. Therefore, the implementation of this call is simply to ACK the request from the HWTUL, and then to set the user_status back to RUN.

**hthread_mutex_lock Implementation**

The hthread_mutex_lock opcode is implemented by making a mutex_lock request to the Mutex Manager. This call requires a bus master read, to the Mutex Manager's mutex_lock register, with the mutex number and the HWT's thread id embedded in the address lines. The Controller State Machine pulls the mutex number from user_argument_one, line 26 to 31. The thread id is read from the thread_id register.

Upon completion of the bus transaction, the Controller State Machine reads the Bus2IP_Data lines to determine if the HWT has the lock. If the HWT has the lock, the user_status is changed to RUN, and the system's status is not updated. However, if the HWT did not get the lock, the user_status is not updated (it remains at ACK), and the system's status is changed to BLOCKED.

The HWT will remain in a BLOCKED state until a new RUN command is issued to it. When a RUN command is received, the Controller State Machine currently assumes that it is a result of obtaining the lock. The Controller State Machine then updates both the system's status and user_status register to RUNNING and RUN respectively.

**hthread_mutex_unlock Implementation**

The hthread_mutex_unlock opcode is implemented by making a mutex_unlock request to the Mutex Manager. This call requires a bus master read, to the Mutex Manager's mutex_unlock register, with the mutex number and the HWT's thread id embedded in the address lines. The Controller State Machine pulls the mutex number from user_argument_one, line 26 to 31. The thread id is read from the thread_id register.

Upon completion of the bus transaction, the mutex is unlocked and no longer owned by the HWT. The Controller State Machine updates the user_status to RUN.

**hthread_intrassoc Implementation**

The hthread_intrassoc opcode is currently not implemented.

**Implementation Performance**

In this section, the performance results of the HWTI will be given. These numbers are from the implemented version of the HWTI described in the State Machine Implementations section.

**Slice Count**

The HWTI uses 404 slices of the FPGA.

This number includes the slices for the IPIF and a simple HWTUL. The IPIF was the master slave implementation from Xilinx for the OPB. The HWTUL was a thread that exits immediately following a RUN command.

The 404 slices represent 2% of all slices on the Virtex 2 Pro 30.
Timing

The following results were taken from a ModelSim simulation, and not directly from the VHDL implemented version. They are however assumed to be accurate. Timings that include bus transactions were taken when the bus was not used, or rather the HWTI did not have to wait to use the bus.

System Level API Commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Clock Cycles</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write to thread_id register.</td>
<td>5</td>
<td>Time from receiving the thread_id to the time the system status changes to USED.</td>
</tr>
<tr>
<td>Write a RUN to the command register.</td>
<td>5</td>
<td>Time from receiving the RUN command to the time the user_status register changes to RUN.</td>
</tr>
<tr>
<td>Write a RESET to the command register.</td>
<td>4</td>
<td>Time from receiving the RESET command to the time the user_status register changes to UNUSED.</td>
</tr>
</tbody>
</table>

User Level API Commands:

<table>
<thead>
<tr>
<th>opcode</th>
<th>Clock Cycles</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>14</td>
<td>Time from the HWTUL issuing the opcode, to the time the HWTI returns the user_status to RUN, including bus transaction time.</td>
</tr>
<tr>
<td>write</td>
<td>33</td>
<td>Time from the HWTUL issuing the opcode, to the time the HWTI returns the user_status to RUN, including bus transaction time.</td>
</tr>
<tr>
<td>hthread_yield</td>
<td>5</td>
<td>Time from the HWTUL issuing the opcode, to the time the HWTI returns the user_status to RUN.</td>
</tr>
<tr>
<td>hthread_self</td>
<td>5</td>
<td>Time from the HWTUL issuing the opcode, to the time the HWTI returns the user_status to RUN.</td>
</tr>
<tr>
<td>hthread_mutex_lock</td>
<td>20</td>
<td>Time from the HWTUL issuing the opcode, to the time the HWTI returns the user_status to RUN, including bus transaction time, and Mutex Manager time.</td>
</tr>
<tr>
<td>hthread_mutex_unlock</td>
<td>20</td>
<td>Time from the HWTUL issuing the opcode, to the time the HWTI returns the user_status to RUN, including bus transaction time, and Mutex Manager time.</td>
</tr>
<tr>
<td>hthread_exit</td>
<td>20</td>
<td>Time from the HWTUL issuing the opcode, to the time the HWTI ends the bus transaction with the Thread</td>
</tr>
</tbody>
</table>
Address Map

The following table is the address map for the system level registers. To determine the exact address of the register, for a particular HWT, add the base address of the HWT to the offset. For example, the address of the command register, for a HWT with base address 0x6300 0000, is 0x6300 00C0.

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread_id</td>
<td>0x0000 0000</td>
</tr>
<tr>
<td>verify</td>
<td>0x0000 0004</td>
</tr>
<tr>
<td>status</td>
<td>0x0000 0008</td>
</tr>
<tr>
<td>command</td>
<td>0x0000 000C</td>
</tr>
<tr>
<td>argument</td>
<td>0x0000 0010</td>
</tr>
<tr>
<td>result</td>
<td>0x0000 0018</td>
</tr>
<tr>
<td>master_read</td>
<td>0x0000 0020</td>
</tr>
<tr>
<td>master_write</td>
<td>0x0000 0024</td>
</tr>
<tr>
<td>debug_system</td>
<td>0x0000 0028</td>
</tr>
<tr>
<td>debug_user</td>
<td>0x0000 002C</td>
</tr>
<tr>
<td>debug_control</td>
<td>0x0000 0030</td>
</tr>
</tbody>
</table>

C, HIF, and VHDL Comparison and Example

In this section, a line by line comparison between threads written in pthread C code, hthread C code, hthread’s Hardware Intermediate Form (HIF), and VHDL will be given. The purpose is to show a concrete example of a thread for each of the four representations. It is hoped that the reader can gain an understanding that the four forms are functionally equivalent. Furthermore, it is hoped that a developer can use this example to either write his or her own hand written hardware thread, or develop a mechanism to translate one form to the other.

Pthread Example

Many software developers are already familiar with pthreads, the programming model hthreads is derived from. Given this, the comparisons in this section will start with the following pthread
threaded function as its base example. Note that in this example, only the thread function is given, the main function to create a thread is not shown. Also, it is assumed that fooMutex, is a global variable, previously initialized. This example code, when translated to hthreads, includes each of the system calls supported by the HWTI.

```c
void * basicThread( void * argument ) {
    int * fooAddr = (int *) argument;

    pthread_mutex_lock( &fooMutex );
    int fooValue = *fooAddr;
    fooValue += pthread_self();
    pthread_yield();
    *fooAddr = fooValue;
    pthread_mutex_unlock( &fooMutex );

    return fooAddr;
}
```

Each of the ten lines will now be broken down and compared between hthreads, HIF, and VHDL. The VHDL version uses a two process state machine programming model. Where a state, or in cases of function calls, a set of states, represent the equivalent code. There are other VHDL styles that could be used for the HWTUL code, each producing functionally equivalent code. The important factor is not the specific style of code, but how the HWTUL interacts with the HWTI to start, stop, and make system calls. In particular, note the hand shake protocol the HWTI expects, described above in the user_opcode and user_status sections.

### Function Initialization

The function declaration and initialization is the largest difference between the four forms, at least in consideration of the amount of source code used in the VHDL version. Where as in pthread, hthread, and HIF, the function and initial arguments must be declared, in VHDL the interconnect between the HWTI and HWTUL must explicitly be stated. Note that the name of the function is not given in the VHDL form. The equivalent of the function name, is the address of the HWT. The address of the HWT is set by the vendor provided system builder program. The function initialization VHDL code may be seen as having eight sections, library declarations, entity declaration (for the HWTUL), state declaration, signal declaration (that map to the variables used in the thread), constant declaration (that are specific for the information passed to and from the HWTI), next state process (where register values are latched for the next clock cycle), and the opening lines of the state machine process. Obviously, if a two process state machine coding style is not used, this section of code will look quite different.

<table>
<thead>
<tr>
<th></th>
<th>pthread</th>
<th>hthread</th>
<th>HIF</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>void * basicThread( void * argument ) {</td>
<td>void * basicThread( void * argument ) {</td>
<td>function basicThread 1</td>
<td>library IEEE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>use IEEE.std_logic_1164.all;</td>
</tr>
<tr>
<td></td>
<td>pthread_mutex_lock( &amp;fooMutex );</td>
<td></td>
<td></td>
<td>use IEEE.std_logic_arith.all;</td>
</tr>
<tr>
<td></td>
<td>int fooValue = *fooAddr;</td>
<td></td>
<td></td>
<td>use IEEE.std_logic_unsigned.all;</td>
</tr>
<tr>
<td></td>
<td>fooValue += pthread_self();</td>
<td></td>
<td></td>
<td>use IEEE.std_logic_misc.all;</td>
</tr>
<tr>
<td></td>
<td>pthread_yield();</td>
<td></td>
<td></td>
<td>library Unisim;</td>
</tr>
<tr>
<td></td>
<td>*fooAddr = fooValue;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pthread_mutex_unlock( &amp;fooMutex );</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
use Unisim.all;

-- Port declarations
-- Definition of Ports:
--
-- Misc. Signals
--  clock
--
-- HWTI to HWTUL interconnect
-- intrfc2thrd_status
-- intrfc2thrd_result
--
-- HWTUL to HWTI interconnect
-- thrd2intrfc_opcode
-- thrd2intrfc_argument_one
-- thrd2intrfc_argument_two
--

-- Thread Manager Entity section

entity user_logic_hwtul is
  port (  
    clock : in std_logic;
    intrfc2thrd_status : in std_logic_vector(0 to 3);
    intrfc2thrd_result : in std_logic_vector(0 to 31);
    thrd2intrfc_opcode : out std_logic_vector(0 to 7);
    thrd2intrfc_argument_one : out std_logic_vector(0 to 31);
    thrd2intrfc_argument_two : out std_logic_vector(0 to 31)
  );
end entity user_logic_hwtul;

-- Architecture section

architecture IMP of user_logic_hwtul is
-- Signal declarations

type hwtul_state is (START, IDLE, MUTEX_LOCK_1, MUTEX_LOCK_2, MUTEX_LOCK_3, READ_MEMORY_1, READ_MEMORY_2, READ_MEMORY_3, SELF_1, SELF_2, SELF_3, YIELD_1, YIELD_2, YIELD_3, WRITE_MEMORY_1, WRITE_MEMORY_2, WRITE_MEMORY_3, MUTEX_UNLOCK_1,
MUTEX_UNLOCK_2,
MUTEX_UNLOCK_3,
EXIT_INIT,
EXIT_WAIT_ACK,
EXIT_WAIT
);

signal current_state, next_state : hwtul_state := START;
signal fooAddr, fooAddr_next : std_logic_vector(0 to 31);
signal fooVal, fooVal_next : std_logic_vector(0 to 31);

signal opcode, opcode_next : std_logic_vector(0 to 7);
signal argOne, argOne_next : std_logic_vector(0 to 31);
signal argTwo, argTwo_next : std_logic_vector(0 to 31);

-- user_status Constants
constant USER_STATUS_RESET : std_logic_vector(0 to 3) := x"1";
constant USER_STATUS_RUN : std_logic_vector(0 to 3) := x"2";
constant USER_STATUS_ACK : std_logic_vector(0 to 3) := x"4";
constant USER_STATUS_PAUSE : std_logic_vector(0 to 3) := x"8";

-- user_opcode Constants
constant OPCODE_NOOP : std_logic_vector(0 to 7) := x"00";
constant OPCODE_HTHREAD_EXIT : std_logic_vector(0 to 7) := x"01";
constant OPCODE_HTHREAD_EXIT_ERROR : std_logic_vector(0 to 7) := x"09";
constant OPCODE_READ : std_logic_vector(0 to 7) := x"02";
constant OPCODE_WRITE : std_logic_vector(0 to 7) := x"03";
constant OPCODE_HTHREAD_SELF : std_logic_vector(0 to 7) := x"04";
constant OPCODE_HTHREAD_YIELD : std_logic_vector(0 to 7) := x"05";
constant OPCODE_HTHREAD_MUTEX_LOCK : std_logic_vector(0 to 7) := x"06";
constant OPCODE_HTHREAD_MUTEX_UNLOCK : std_logic_vector(0 to 7) := x"07";

-- misc constants
constant Z32 : std_logic_vector(0 to 31) := (others => '0');
constant H32 : std_logic_vector(0 to 31) := (others => '1');

-- Begin architecture
begin -- architecture IMP

HWTUL_STATE_PROCESS : process (clock, fooVal_next, fooAddr_next,
opcode_next, argOne_next, argTwo_next) is
begin

if (clock'event and (clock = '1')) then
  fooAddr <= fooAddr_next;
  fooVal <= fooVal_next;
  opcode <= opcode_next;
  argOne <= argOne_next;
  argTwo <= argTwo_next;
  thrd2intrfc_opcode <= opcode_next;
  thrd2intrfc_argument_one <= argOne_next;
  thrd2intrfc_argument_two <= argTwo_next;

end if;
end process HWTUL_STATE_PROCESS;
end -- architecture IMP;
if ( intrfc2thrd_status = USER_STATUS_RESET ) then
    current_state <= IDLE;
else
    current_state <= next_state;
end if;

end if;
end process HWTUL_STATE_PROCESS;

HWTUL_STATE_MACHINE : process (clock) is
begin

    -- Default register assignments
    next_state <= current_state;
    opcode_next <= opcode;
    argOne_next <= argOne;
    argTwo_next <= argTwo;
    fooAddr_next <= fooAddr;
    fooVal_next <= fooVal;

    -- The state machine
    case current_state is
    when START =>
        -- Set default values
        opcode_next <= OPCODE_NOOP;
        argOne_next <= Z32;
        argTwo_next <= Z32;
        next_state <= IDLE;

    Reading Function Argument

    The first line of the thread is to create a local pointer to an integer. In HIF, an unlimited set of
    registers already exists, so variable declaration is not needed. The argument is pulled from the
    virtual HWTI using the readarg (read argument) command. In VHDL, the declaration of the
    signal to represent the pointer was done in the initialization. However, the assignment is not
    made until after the HWTUL is told to run by the HWTI. Also note, that in the state machine
    style, the next state has to be explicitly declared.

| pthread | int * fooAddr = (int *) argument; |
| hthread | int * fooAddr = (int *) argument; |
| HIF     | readarg R1 0                       |
| VHDL    | when IDLE =>
          |     case intrfc2thrd_status is
          |       when USER_STATUS_RUN =>
          |           fooAddr_next <= intrfc2thrd_result;
          |           next_state <= MUTEX_LOCK_1;
          |       when others =>
          |           next_state <= IDLE;
          |     end case; |
Locking a Mutex

The next step in the thread’s process is to lock the global variable fooMutex, which again was assumed to be initialized outside of this thread. In all forms, once the function is complete, the thread may assume it has the mutex lock. In the HIF version, two explicit steps are made. First to determine the mutex number of fooMutex, and then to lock it. In VHDL, we see three state are needed to lock the mutex, furthermore, a notion of the mutex number is already known (in this case the mutex number is 0). The first state, the HWTUL sets the arguments and sets the syscall number (or opcode). In the second state, the HWTUL waits for the HWTI to acknowledge the request. Finally, the third state, the HWTUL waits for the HWTI to finish the request. It is these three basic states, that are used in each system call to the HWTI. The difference between system calls are the arguments, the opcode, and the result, if any.

<table>
<thead>
<tr>
<th>pthread</th>
<th>pthread_mutex_lock( &amp;fooMutex );</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread</td>
<td>hthread_mutex_lock( &amp;fooMutex );</td>
</tr>
</tbody>
</table>
| HIF     | addressof R2 fooMutex
       | syscall mutex lock R2 |
| VHDL    | -- Lock mutex zero
       | when MUTEX_LOCK_1 =>
       |   -- Tell the HWTI, which mutex to lock
       |     argOne_next <= Z32;
       |     opcode_next <= OPCODE_HTHREAD_MUTEX_LOCK;
       |     next_state <= MUTEX_LOCK_2; |
       | when MUTEX_LOCK_2 =>
       |     -- Wait for the HWTI to ack
       |       if ( intrfc2thrd_status = USER_STATUS_ACK ) then
       |         opcode_next <= OPCODE_NOOP;
       |         next_state <= MUTEX_LOCK_3;
       |       else
       |         next_state <= MUTEX_LOCK_2;
       |       end if;
       | when MUTEX_LOCK_3 =>
       |     -- Wait for the HWTI to tell us to start running again
       |     -- When we start running again, we know we have the lock
       |     if ( intrfc2thrd_status = USER_STATUS_RUN ) then
       |       next_state <= READ_MEMORY_1;
       |     else
       |       next_state <= MUTEX_LOCK_3;
       |     end if;

Reading a Value from Memory

The thread now reads the value of fooAddr from memory. In HIF, this is performed by the gread (or global read) function. In the VHDL version we see a similar set of states seen with the locking of a mutex. The important difference is when the HWTI changes the status back to RUN, the value of the requested read address is placed in the intrfc2thrd_result register. The VHDL state machine may read this register anytime after the status returns to RUN and before the next system call request.

| pthread | int fooValue = *fooAddr; |
Obtaining the Thread ID

The next line of code is actually two operations in one. First the determination of the thread’s ID, second the adding of the ID to fooValue. Because there are two operations, the HIF example must separate the operations into two lines. In the VHDL version, the three states are again seen, however, the operation of adding the thread ID to the existing value of fooValue can be performed internal to the third state. The other languages require multiple clock cycles to perform the same operation. This illustrates, although briefly, instruction level parallelism within hardware.
next_state <= SELF_3;
else
    next_state <= SELF_2;
end if;

when SELF_3 =>
    -- Wait for the HWTI to tell us to start running again
    if (intrfc2thrd_status = USER_STATUS_RUN) then
        next_state <= SELF_1;
        fooVal_next <= fooVal + intrfc2thrd_result;
    else
        next_state <= SELF_3;
    end if;

Yielding the Processor

Yielding the processor is next for the thread. For a hardware thread, yielding the processors is currently irrelevant. A good HLL to HDL compiler will remove hthread_yield references. The practicalness of a yield syscall, for a hardware thread, is a brief, 5 clock cycle, wait statement. However, for completeness, the examples are given.

<table>
<thead>
<tr>
<th>pthread</th>
<th>pthread_yield();</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread</td>
<td>hthread_yield();</td>
</tr>
<tr>
<td>HIF</td>
<td>syscall yield</td>
</tr>
<tr>
<td>VHDL</td>
<td>-- Call hthread_yield</td>
</tr>
<tr>
<td></td>
<td>when YIELD_1 =&gt;</td>
</tr>
<tr>
<td></td>
<td>-- Yield the CPU, in HW, returns immediately to resume execution.</td>
</tr>
<tr>
<td></td>
<td>opcode_next &lt;= OPCODE_HTHREAD_YIELD;</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= YIELD_2;</td>
</tr>
<tr>
<td></td>
<td>when YIELD_2 =&gt;</td>
</tr>
<tr>
<td></td>
<td>-- Wait for the HWTI to ack</td>
</tr>
<tr>
<td></td>
<td>if (intrfc2thrd_status = USER_STATUS_ACK) then</td>
</tr>
<tr>
<td></td>
<td>opcode_next &lt;= OPCODE_NOOP;</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= YIELD_3;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= YIELD_2;</td>
</tr>
<tr>
<td></td>
<td>end if;</td>
</tr>
<tr>
<td></td>
<td>when YIELD_3 =&gt;</td>
</tr>
<tr>
<td></td>
<td>-- Wait for the HWTI to tell us to start running again</td>
</tr>
<tr>
<td></td>
<td>if (intrfc2thrd_status = USER_STATUS_RUN) then</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= WRITE_MEMORY_1;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= YIELD_3;</td>
</tr>
<tr>
<td></td>
<td>end if;</td>
</tr>
</tbody>
</table>

Writing a Value to Memory

The updated value of fooValue is now written back to main memory. For HIF, the gwrite (global write) function is used. In VHDL, the same three step process is used. Note thought that
the once the HWTUL has initiated the write, it only has to wait until the HWTI allows it to run again, since there are no return values the HWTUL is concerned with for a write.

<table>
<thead>
<tr>
<th>pthread</th>
<th>*fooAddr = fooValue;</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread</td>
<td>*fooAddr = fooValue;</td>
</tr>
<tr>
<td>HIF</td>
<td>gwrite R1 R3</td>
</tr>
</tbody>
</table>
| VHDL             | when WRITE_MEMORY_1 =>  
|                  |   -- Tell the HWTI to write foo out to memory  
|                  |     argTwo_next <= fooVal;  
|                  |     argOne_next <= fooAddr;  
|                  |     opcode_next <= OPCODE_WRITE;  
|                  |     next_state <= WRITE_MEMORY_2;  
|                  | when WRITE_MEMORY_2 =>  
|                  |   -- Wait for the HWTI to ACK the write request  
|                  |     if ( intrfc2thrd_status = USER_STATUS_ACK ) then  
|                  |       opcode_next <= OPCODE_NOOP;  
|                  |       next_state <= WRITE_MEMORY_3;  
|                  |     else  
|                  |       next_state <= WRITE_MEMORY_2;  
|                  |     end if;  
|                  | when WRITE_MEMORY_3 =>  
|                  |   -- Wait for the HWTI to tell us to RUN again  
|                  |     if ( intrfc2thrd_status = USER_STATUS_RUN ) then  
|                  |       next_state <= MUTEX_UNLOCK_1;  
|                  |     else  
|                  |       next_state <= WRITE_MEMORY_3;  
|                  |     end if;  

Unlocking a Mutex

Unlocking a mutex is almost identical to locking a mutex, with the obvious exception of unlock verses lock system call. In all cases, once the function is complete, the thread may assume it no longer owns the mutex.

<table>
<thead>
<tr>
<th>pthread</th>
<th>pthread_mutex_unlock( &amp;fooMutex );</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread</td>
<td>hthread_mutex_unlock( &amp;fooMutex );</td>
</tr>
<tr>
<td>HIF</td>
<td>syscall mutex_unlock R2</td>
</tr>
</tbody>
</table>
| VHDL             | -- Unlock mutex zero  
|                  | when MUTEX_UNLOCK_1 =>  
|                  |   -- Tell the HWTI what mutex to lock  
|                  |     argOne_next <= Z32;  
|                  |     opcode_next <= OPCODE_HTHREAD_MUTEX_UNLOCK;  
|                  |     next_state <= MUTEX_UNLOCK_2;  
|                  | when MUTEX_UNLOCK_2 =>  
|                  |   -- Wait for the HWTI to ack  
|                  |     if ( intrfc2thrd_status = USER_STATUS_ACK ) then  
|                  |       opcode_next <= OPCODE_NOOP;  
|                  |       next_state <= MUTEX_UNLOCK_3;  
|
Exiting the Thread

The last statement is to return fooAddr to the parent thread. An important difference is seen with the VHDL version. The actually returning of fooAddr to the parent is handled by the HWTI, however the state machine needs to stop executing or at least maintain itself in a single state. This is because, when a hardware thread exists, the physical logic, in the FPGA fabric still exists, and is not removed (as a software thread stack is cleaned up from memory). Therefore, to be analogous with software, it is important the HWTUL continuously waits until it is reset by the system.

Closeout

The final lines of code are meant to close out the function, providing whatever syntax is needed.
when others =>
next_state <= IDLE;
end case;
end process HWTUL_STATE_MACHINE;
end architecture IMP;