Hardware Thread Interface Specification and Implementation Document

Purpose and Overview

The primary purpose of the hardware thread interface (HWTI) is to provide an abstraction layer between the Hybridthreads System (hthreads) 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. Additionally, the HWTI is extended to provide a quick memory for the hardware thread as well as a function call stack.

The abstraction layer of the HWTI exists in two directions. One interface exists for allowing the hthread system to interact with the thread, a second interface exists allowing the thread to interact with the hthread system. From a hardware thread'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.

The HWTI is instantiated with a small memory. This memory is accessible both by other system cores, and the thread’s user logic. From a system perspective, the memory is an extension of global memory, addressable in the same fashion that standard global memory is accessible. By design, accessing this local memory, for the user logic thread, is also done exactly equivalently to accessing standard global memory. The instantiated local memory allows the HWTI to implement a function call stack in support of a generic function call mechanism. Consequently, the HWTI provides a standard interface to support recursion.

A hardware thread is physically composed of three parts, the bus connection, the HWTI, and finally the user logic circuit representing the functionality of the thread. Typically the bus protocol is abstracted with a vendor supplied bus interface, for example the IBM CoreConnect IPIF. In theory, the HWTI may be connected to any bus. The register set representing the system interface will remain the same, while the protocol to communicate with the bus may have to be modified. In practice, the HWTI has only been synthesized with the IBM CoreConnect OPB buss. Regardless of the bus, the interface between the HWTI and the user logic is constant. The user logic, formally the Hardware Thread User Logic (HWTUL), thus has a consistent interface to communicate with, promoting portability.

This paper is organized in three sections. The first section will first describe the specifications of the HWTI, detailing the protocol allowing the system to initialize and run a hardware thread, and the protocols allowing the hardware thread to invoke the resources of the HWTI. In particular the protocols to invoke functions, including system functions. The second section will be a description of an implementation for the specifications. 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 provided.

System Level Application Programming Interface

The system level API consists of a set of five memory mapped registers for controlling the interface and an address space for reading and writing the local memory within the HWTI. The register names are thread_id, 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 are 32 bits wide.
The local HWTI memory may be accessed through the standard bus operations, using the HWTI address space. The HWTI may reserves a small amount of memory, near the bottom of its address space for maintaining state information. If so, values read from this range, either from the system or the user thread, will be zero, writing to this range will have no effect.

**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.

**command**

*Overview*

The `command` register is written to by the system to tell the hardware thread to RUN, RESET, or COLDBOOT. A RUN command serves two purposes. First to tell the hardware thread to start executing, second, if the hardware thread is waiting on another hthread core, to wake up and check the status of the core, and resume running. The RESET and COLDBOOT are very similar. The RESET command tells the hardware thread to reset all variables and registers specific to the control and execution of the thread’s user logic. This applies both to the registers
in the HWTI and the user logic. After issuing a RESET, the status is returned to NOT_USED (see status register). The COLDBOOT command will reset all variables and registers that the RESET command does and reset any state information that is preserved between a hardware thread’s execution. In effect, it returns the hardware thread to a state consistent to a power on situation.

The RESET command should be given prior to a hardware thread being created by the system. In a well functioning system, the COLDBOOT command should never have to be given. In a test application, the COLDBOOT should be given to restore the HWTI to a power on state (ie resetting all state internal state information).

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 or BLOCKED. 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. More importantly a RUN command results in the HWTI telling the user logic to start executing. On the user logic interface, the go/wait register is updated to a GO, and the function register is updated to the FUNCTION_START value.

Issuing a RUN while the status is BLOCK, tells the HWTI to recheck the operation causing the block (typically a mutex or condition variable lock). If successful the HWTI updates the user control sub-interface allowing the user logic to resume execution.

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

The command register may be read from at any time, in general returning the last command the hardware thread 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)
- **COLDBOOT** (0100)

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 hardware thread is in. It is generally used for debugging purposes. The possible states the hardware thread may be in are RUNNING, BLOCKED, 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 hardware thread on system start up and after a RESET or COLDBOOT 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 hardware thread is not waiting on a mutex or other blocking type of operations, and the hardware thread 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 hardware thread is waiting to obtain a mutex.
- **EXITED (0000 1000):** The hardware thread 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 (0010 0000):** The hardware thread 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).

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 hardware thread must first issue a RESET command, set the thread_id register, set the argument register (if used), and then issue a RUN command.

The user logic is allowed to read the argument value in the same way it reads any passed in arguments to a user defined function. That is, the user logic issues a POP command, indicating the zero indexed parameter, to the HWTI. The HWTI will respond by placing the value of the argument in the value register.
result

Overview

When a thread is created as joinable, runs, and then exits, the thread has the option of passing results back to the parent thread. To pass back results to the parent, the hardware thread 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 user logic calls the hthread_exit() function, it may pass one argument. This argument is passed in the same manner as any other function call, that is, the user logic will PUSH the result to the HWTI prior to calling hthread_exit(). Once the HWTI receives the hthread_exit call it will copy the value of the parameter into the result register.

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

timer

Overview

The timer register reports the number of clock cycles the HWTI has been running for, if still running, or the number of clock cycles it ran for, if it has exited.

Protocol

The timer register begins counting when the initial RUN command is issued, and stops counting when the user logic issues a hthread_exit.

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 user logic), or simply the user interface, may only be accessed by the user logic. The hthread system has no direct access to their values.

The user interface has three sub-interfaces: memory, function, and control. Each sub-interface will be explained as if it were physically seperated from the other two. However, an implementation of the user interface must merge the sub-interfaces into a single interface. The HWTI therefor has four registers that the HWTI uses to signal the user logic. These are the address, value, function, and go/wait. Alternatively, the user logic has four registers it has to signal the HWTI. These are address, value, function, and opcode.

The address and value registers are 32 bits. The function register is 16 bits. The opcode register is 6 bits. The go/wait register is 1 bit.
Memory Sub-Interface

Overview

The memory sub-interface is composed of three registers, opcode, address, and value. The opcode register is writable by the user logic and enables the user logic to request operations from the HWTI. When requesting an action from the HWTI, the user logic must set the address and value registers (as appropriate) in the same clock cycle. The address register is both readable and writable and will be used to indicate memory addresses. The value register is both readable and writable and will be used to indicate data values.

There are six operations (opcodes) permitted, load from address, store to address, declare local variables, read local variable, and address of local variable. These operations are intentionally similar to high level language operations. Depending on the operation, either the value, address, or both register will be used.

When issuing an opcode, the user logic must wait the clock cycle the opcode is first read by the HWTI. This is because the HWTI will keep the goWait register high (a GO signal) during the initial request clock cycle. On the clock cycle following the request, the user logic must wait only if the goWait signal is low (a WAIT signal). The user logic must continue to wait until the goWait signal returns high. Any appropriate response will be available, in either the address or value registers, to the user logic when the goWait signal returns to high. Lastly the opcode register, when the user logic issues an operation, must only be set for one clock cycle. In other words, the opcode register must be set at NOOP the clock cycle following the request.

Protocol

Each of the six permitted operations, their protocols, and high level language equivalent are listed below.

- **LOAD:** Performs a load operation to memory at the address specified in the address register. LOAD may be used for accessing standard global memory, the local instantiated memory, or any other location in the hthread address space. Once the operation is complete, the value register will hold the value of the address. This operation is equivalent to pointer dereferencing, for example *ptr.

- **STORE:** Performs a store operation to memory at the address specified in the address register with the value given in the value register. STORE may be used for saving information to standard global memory, the local instantiated memory, or any other location in the hthread address space. The HWTI does not return any information to the user logic with this operation. This operation is equivalent to storing a value to a dereferenced pointer, for example *ptr = 4.

- **DECLARE:** Allows the user logic to request space in the HWTI’s function stack, for local variables used by the user logic. Each declared variable will have an address, and thus could be accessed by other hthread cores (if a pointer is passed to that core). When requesting a DECLARE operation, the user logic specifies the number of words it wants to reserve space for in the value register. The HWTI does not return any information to the user logic with this operation. The user logic is allowed to issue multiple declare statements within a function. However, requesting a DECLARE after a PUSH request is prohibited. The allocated space is accessible by the user logic using READ and WRITE
operations. The HWTI will maintain space for the declared variable until the function returns. This operation is equivalent to declaring a integer, for example int x, y, z.

- **READ**: Allows the user logic to read the value of an declared variable. The variable to read, is specified as an zero based index, in the address register. For example, if a DECLARE 4 was issued previously, to read the second declared variable the user logic would issue a READ 1. The HWTI responds by placing the value of the variable, in the value register. This operation is equivalent to reading a variable.

- **WRITE**: Allows the user logic to write a value to a declared variable. The variable to write, is specified as a zero based index, in the address register. The value to write is specified in the value register. For example, if a DECLARE 4 was issued previously, to write a 1234 to the second declared variable the user logic would issue a READ 1 1234. The HWTI does not return any information to the user logic with this operation. This operation is equivalent to writing to a variable, for example int x = 1234.

- **ADDRESSOF**: Allows the user logic to request the address of a declared variable. The variable to learn the address of is specified, as a zero based index, in the address register. For example, if a DECLARE 4 was issued previously, to learn the address of the second declared variable the user logic would issue a ADDRESSOF 1. The HWTI responds by returning the address of the variable in the address register. This operation is equivalent to the address of opperator, for example &x.

## Memory Sub-Interface

### Overview

The function call sub-interface is composed of three registers; opcode, function, and value. The opcode register, similar to the opcode register in the memory sub-interface, allows the user logic to request operations from the HWTI. There are four non-noop operations. These operations are pushing function parameters onto the stack, calling a function, popping function parameters from the stack, and returning from a function. By passing parameters via the stack, this enables a consistent function call protocol regardless of the number of parameters.

The function register tells the HWTI which function the user logic wants to call. The HWTI reserves a number of values, x8000 to x8FFF, for system calls it supports, and x9000 to xFFFF, for future library calls. The value x0000 is a signal to the user logic to reset itself, x0001 signals the user logic to execute any state it wants, and x0002 signals the user logic to execute its start function. Values x0003 to x7FFF are reserved for user logic defined functions. The user logic defined function values are completely analogous to starting addresses for functions in software. In hardware, these values may be implemented as states in a state machine. The HWTI will pass control to these states, through the control sub-interface, explained later.

A brief pseudo-code example of how the function sub-interface for calling mutex lock is given in Figure 5. In state x0101, the user logic pushes the address of the mutex it wants to lock onto the stack, in this case the mutex is at x0023 8F20. In state x0102, the user logic calls hthread_mutex_lock() (the hthread function codes are listed in the Control Sub-Interface section).

<table>
<thead>
<tr>
<th>State</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0101</td>
<td>push x0023 8F20</td>
</tr>
<tr>
<td>x0102</td>
<td>call x8032, x0103</td>
</tr>
<tr>
<td>x0103</td>
<td>...</td>
</tr>
</tbody>
</table>

*Function Call Example*
while specifying that once the mutex lock function is completed, the HWTI should return control to the user logic in state x0103.

As is the case in the memory sub-interface, when issuing an opcode, the user logic must wait the clock cycle the opcode is first read by the HWTI. This is because the HWTI will keep the goWait register high (a GO signal) during the initial request clock cycle. On the clock cycle following the request, the user logic must wait only if the goWait signal is low (a WAIT signal). The user logic must continue to wait until the goWait signal returns high. Any appropriate response will be available to the user logic when the goWait signal returns to high.

**Protocol**

Each of the four permitted operations and their protocols are listed below.

- **PUSH:** Prior to calling a function, the user logic may pass parameters to the, soon to be called, function using the PUSH construct. Each PUSH places the parameter specified in the value register onto the HWTI’s stack. The user logic may push as many parameters as needed for a function. Each parameter is 32 bits. The HWTI does not return any value to the user logic for the PUSH operation. When pushing parameters onto the stack, the user logic should push the last parameter first. For example, if the user logic is calling foo(a, b, c), the user logic should push the value of c first, then b, and finally a.

- **POP:** Once the HWTI transfers control to a new set of states in the user logic, representing an user defined called function, the user logic may use the POP operation to retrieve the values of the parameters. To allow the function to read any of the parameters any time prior to a RETURN, the user logic specifies the parameter it wants to read in the value register. For example, if the function foo(a, b, c) was called, to read the parameter a, the user logic would request a POP 0, to read the parameter b, the user logic would request a POP 1, and so on. The HWTI responds with the value of the parameter in the value register.

- **CALL:** After all parameters are PUSHed to the HWTI, the user logic may use CALL to invoke a function. Once the called function finishes, the HWTI returns control to the user logic where the call was made. The CALL operation may be used for either a system or library function, or transfer control to a function defined locally within the user logic. When using the CALL operation, the user logic must specify the function it wants to invoke, and the state to return control to after the call is complete. The function to invoke is specified in the function register. The return state is specified in bits 16 to 31 of the value register (bits 0 through 15 are ignored by the HWTI). When the called function issues a RETURN, the HWTI returns control to the specified return state, with the any return value set in the value register.

- **RETURN:** When a user defined function is ready to return, it issues a RETURN operation. The user logic may also pass back one 32 bit value to the caller function. The value to return is specified in the value register. Any declared variables (from the memory sub-interface) are deallocated from the function stack on a RETURN.
Control Sub-Interface

Overview

The control sub-interface details how the HWTU manages the execution and delays of the HWTUL. This sub-interface has two registers, a goWait register, and a function register.

Protocol

The goWait register tells the user logic to continue execution, or to temporarily halt execution. The HWTI halts the user logic’s execution to give it time to fulfill a request. For example, during a load to global memory, the HWTI may require up to 60 clock cycles to finish the request. It is necessary for the user logic to stop execution until the load is fulfilled and the HWTI can report the value back to the user logic. The single goWait register creates a simple hand shaking protocol between the HWTI and user logic. The user logic may only request a service from the HWTI when the goWait register reads ‘1’ (a go), and must halt execution (or at least not request anything further) when it reads ‘0’ (a wait). The user logic, must also halt execution on the same clock cycle a request to the memory or function sub-interface is made.

The function register enables the HWTI to tell the user logic which logic to execute, or which state (for a state machine implementation) to be in. It is analogous to the program counter in a CPU. Intentionally like the function register in the function call sub-interface, certain values have reserved meaning. A value of x0000 tells the user logic to reset itself, it will be the default value on power up and HWTI reset. A x0001, tells the user logic to control its own execution. A x0002, tells the user logic to execute its first instruction. Values x80000 to xFFFF will not be used (since they were reserved for system call or library functions implemented outside the user logic). Values x0003 to x7FFF will tell the user logic to execute specific states or logic within its implementation.

User Logic State Machine Example

The following code is an example state machine that uses many of the supported opcodes to calculate the factorial of a number. The number to calculate the factorial of is passed to the hardware thread as an argument. The user thread calculates the factorial using a recursive algorithm. Finally the user logic returns the calculated factorial as the threads result.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_misc.all;
library Unisim;
use Unisim.all;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_misc.all;

library Unisim;
use Unisim.all;

-- Title: Hardware Thread User Logic Recursive Factorial Example
---
-- Port declarations
---
-- Definition of Ports:
--
-- Misc. Signals
--    clock
```
```
-- HWTI to HWTUL interconnect
-- instrc2thrd_address 32 bits memory function
-- instrc2thrd_value 32 bits memory
-- instrc2thrd_function 16 bits control
-- thrd2intrfc_goWait 1 bit control

-- HWTUL to HWTI interconnect
-- thrd2intrfc_address 32 bits memory
-- thrd2intrfc_value 32 bits memory function
-- thrd2intrfc_function 16 bits function
-- thrd2intrfc_opcode 6 bits memory function

-- Thread Manager Entity section
entity user_logic_hwtul is
  port (
    clock : in std_logic;
    instrc2thrd_address : in std_logic_vector(0 to 31);
    instrc2thrd_value : in std_logic_vector(0 to 31);
    instrc2thrd_function : in std_logic_vector(0 to 15);
    instrc2thrd_goWait : in std_logic;
    thrd2intrfc_address : out std_logic_vector(0 to 31);
    thrd2intrfc_value : out std_logic_vector(0 to 31);
    thrd2intrfc_function : out std_logic_vector(0 to 15);
    thrd2intrfc_opcode : out std_logic_vector(0 to 5)
  );
end entity user_logic_hwtul;

-- Architecture section
architecture IMP of user_logic_hwtul is
  -- Signal declarations
  constant FUNCTION_RESET                      : std_logic_vector(0 to 15) := x"0000";
  constant FUNCTION_USER_SELECT                : std_logic_vector(0 to 15) := x"0001";
  constant FUNCTION_START                      : std_logic_vector(0 to 15) := x"0002";
  constant POP_ARGUMENT                        : std_logic_vector(0 to 15) := x"0003";
  constant READ_ARGUMENT                       : std_logic_vector(0 to 15) := x"0004";
  constant PUSH_ARGUMENT                       : std_logic_vector(0 to 15) := x"0005";
  constant CALL_FACTORIAL                      : std_logic_vector(0 to 15) := x"0006";
  constant READ_FACTORIAL_RETURN               : std_logic_vector(0 to 15) := x"0007";
  constant PUSH_RETURN                         : std_logic_vector(0 to 15) := x"0008";
  constant CALL_EXIT                           : std_logic_vector(0 to 15) := x"0009";
  constant FUNCTION_EXIT                       : std_logic_vector(0 to 15) := x"000A";
  constant FACTORIAL_1                         : std_logic_vector(0 to 15) := x"0101";
  constant FACTORIAL_2                         : std_logic_vector(0 to 15) := x"0102";
  constant FACTORIAL_3                         : std_logic_vector(0 to 15) := x"0103";
  constant FACTORIAL_4                         : std_logic_vector(0 to 15) := x"0104";
  constant FACTORIAL_5                         : std_logic_vector(0 to 15) := x"0105";
  constant FACTORIAL_6                         : std_logic_vector(0 to 15) := x"0106";
  constant FACTORIAL_7                         : std_logic_vector(0 to 15) := x"0107";
  constant FACTORIAL_8                         : std_logic_vector(0 to 15) := x"0108";
  constant FACTORIAL_9                         : std_logic_vector(0 to 15) := x"0109";
  constant FACTORIAL_END                       : std_logic_vector(0 to 15) := x"010A";
  constant ERROR_STATE                         : std_logic_vector(0 to 15) := x"7999";
  -- Range 0003 to 7999 reserved for user logic's state machine
  -- Range 8000 to 9999 reserved for system calls
  constant FUNCTION_HTHREAD_ATTR_INIT          : std_logic_vector(0 to 15) := x"8000";
  constant FUNCTION_HTHREAD_ATTR_DESTROY       : std_logic_vector(0 to 15) := x"8001";
  constant FUNCTION_HTHREAD_CREATE            : std_logic_vector(0 to 15) := x"8010";
```
constant FUNCTION_HTHREAD_JOIN               : std_logic_vector(0 to 15) := x"8011";
constant FUNCTION_HTHREAD_YIELD               : std_logic_vector(0 to 15) := x"8013";
constant FUNCTION_HTHREAD_EQUAL               : std_logic_vector(0 to 15) := x"8014";
constant FUNCTION_HTHREAD_EXIT               : std_logic_vector(0 to 15) := x"8015";
constant FUNCTION_HTHREAD_EXIT_ERROR         : std_logic_vector(0 to 15) := x"8016";
constant FUNCTION_HTHREAD_MUTEXATTR_INIT     : std_logic_vector(0 to 15) := x"8020";
constant FUNCTION_HTHREAD_MUTEXATTR_DESTROY  : std_logic_vector(0 to 15) := x"8021";
constant FUNCTION_HTHREAD_MUTEXATTR_SETNUM   : std_logic_vector(0 to 15) := x"8022";
constant FUNCTION_HTHREAD_MUTEXATTR_GETNUM   : std_logic_vector(0 to 15) := x"8023";
constant FUNCTION_HTHREAD_MUTEX_INIT         : std_logic_vector(0 to 15) := x"8030";
constant FUNCTION_HTHREAD_MUTEX_DESTROY      : std_logic_vector(0 to 15) := x"8031";
constant FUNCTION_HTHREAD_MUTEX_LOCK         : std_logic_vector(0 to 15) := x"8032";
constant FUNCTION_HTHREAD_MUTEX_UNLOCK       : std_logic_vector(0 to 15) := x"8033";
constant FUNCTION_HTHREAD_MUTEX_TRYLOCK      : std_logic_vector(0 to 15) := x"8034";
constant FUNCTION_HTHREAD_CONDATTR_INIT      : std_logic_vector(0 to 15) := x"8040";
constant FUNCTION_HTHREAD_CONDATTR_DESTROY   : std_logic_vector(0 to 15) := x"8041";
constant FUNCTION_HTHREAD_CONDATTR_SETNUM    : std_logic_vector(0 to 15) := x"8042";
constant FUNCTION_HTHREAD_CONDATTR_GETNUM    : std_logic_vector(0 to 15) := x"8043";
constant FUNCTION_HTHREAD_COND_INIT          : std_logic_vector(0 to 15) := x"8050";
constant FUNCTION_HTHREAD_COND_DESTROY       : std_logic_vector(0 to 15) := x"8051";
constant FUNCTION_HTHREAD_COND_SIGNAL        : std_logic_vector(0 to 15) := x"8052";
constant FUNCTION_HTHREAD_COND_BROADCAST     : std_logic_vector(0 to 15) := x"8053";
constant FUNCTION_HTHREAD_COND_WAIT          : std_logic_vector(0 to 15) := x"8054";
constant FUNC_MALLOC                     : std_logic_vector(0 to 15) := x"A000";
constant FUNCTION_CALLOC                     : std_logic_vector(0 to 15) := x"A001";
constant FUNCTION_FREE                       : std_logic_vector(0 to 15) := x"A002";

constant OPCODE_NOOP                         : std_logic_vector(0 to 5) := "000000";
constant OPCODE_LOAD                         : std_logic_vector(0 to 5) := "000001";
constant OPCODE_DECLARE                      : std_logic_vector(0 to 5) := "000011";
constant OPCODE_READ                         : std_logic_vector(0 to 5) := "000100";
constant OPCODE_WRITE                        : std_logic_vector(0 to 5) := "000101";
constant OPCODE_ADDRESS                      : std_logic_vector(0 to 5) := "000110";
constant OPCODE_PUSH : std_logic_vector(0 to 5) := "010000";
constant OPCODE_POP                          : std_logic_vector(0 to 5) := "010001";
constant OPCODE_CALL                         : std_logic_vector(0 to 5) := "010010";
constant OPCODE_RETURN                      : std_logic_vector(0 to 5) := "010011";
constant Z32 : std_logic_vector(0 to 31) := (others => '0');

-- Begin architecture
begin -- architecture IMP

HWTUL_STATE_PROCESS : process (clock) is
begin
if (clock'event and (clock = '1')) then
  fromUser_address <= fromUser_address_next;
  fromUser_value <= fromUser_value_next;
  fromUser_function <= fromUser_function_next;
end if;

end -- architecture IMP;
fromUser_opcode <= fromUser_opcode_next;

thrd2intrfc_address <= fromUser_address_next;
thrd2intrfc_value <= fromUser_value_next;
thrd2intrfc_function <= fromUser_function_next;
thrd2intrfc_opcode <= fromUser_opcode_next;

argument <= argument_next;
varOne <= varOne_next;
varTwo <= varTwo_next;
result <= result_next;

case fromUser_opcode_next is
when OPCODE_NOOP =>
   -- No new requests from thread to HWTI
case intrfc2thrd_goWait is
when '1' =>
   -- HWTI is allowing us to go
   case intrfc2thrd_function is
   when FUNCTION_USER_SELECT =>
      -- Run the state we want to run
      current_state <= next_state;
      run <= '1';
   when OTHERS =>
      -- Run the state the HWTI wants us to run
      current_state <= intrfc2thrd_function;
      run <= '1';
   end case;
when OTHERS =>
   -- HWTI is telling us to wait
   current_state <= next_state;
   run <= '0';
end case;
end if;
end case;
when OTHERS =>
   -- Clock cycle after a request to HWTI, thread must wait
   current_state <= next_state;
   run <= '0';
end case;
end if;
end process HWTUL_STATE_PROCESS;

HWTUL_STATE_MACHINE : process (clock) is
begin
   -- Default register assignments
   -- next_state <= current_state;
   fromUser_opcode_next <= OPCODE_NOOP;  -- When issuing an OPCODE, must be a pulse
   fromUser_address_next <= fromUser_address;
   fromUser_value_next <= fromUser_value;
   fromUser_function_next <= fromUser_function;
   argument_next <= argument;
   varOne_next <= varOne;
   varTwo_next <= varTwo;
   result_next <= result;

   -- The state machine
   if (run = '1') then
      case current_state is
      when FUNCTION_RESET =>
         -- Set default values
         result_next <= Z32;
         argument_next <= Z32;
         varOne_next <= Z32;
         varTwo_next <= Z32;
         fromUser_opcode_next <= OPCODE_NOOP;
         fromUser_address_next <= Z32;
         fromUser_value_next <= Z32;
         fromUser_function_next <= FUNCTION_START;
      end case;
   end if;
end process HWTUL_STATE_PROCESS;
-- TODO add in a push statement for passing one argument for hthread exit
when FUNCTION_START =>
  next_state <= POP_ARGUMENT;

when POP_ARGUMENT => --0003
  -- Ask the HWTI the value of the passed in argument
  fromUser_value_next <= Z32;
  fromUser_opcode_next <= OPCODE_POP;
  next_state <= READ_ARGUMENT;

when READ_ARGUMENT => --0004
  -- read the value of the passed in argument
  argument_next <= intrfc2thrd_value;
  next_state <= PUSH_ARGUMENT;
when PUSH_ARGUMENT => --0005
  -- push the argument, for the factorial function, on the stack
  fromUser_value_next <= argument;
  fromUser_opcode_next <= OPCODE_PUSH;
  next_state <= CALL_FACTORIAL;

when CALL_FACTORIAL => -- 0006
  -- make a call to the factorial function
  fromUser_function_next <= FACTORIAL_1;
  fromUser_value_next <= Z32(0 to 15) & READ_FACTORIAL_RETURN;
  fromUser_opcode_next <= OPCODE_CALL;
  next_state <= READ_FACTORIAL_RETURN;

when READ_FACTORIAL_RETURN => -- 0007
  -- read the return value
  result_next <= intrfc2thrd_value;
  next_state <= PUSH_RETURN;

when PUSH_RETURN => -- 0008
  -- Push a return value
  fromUser_value_next <= result;
  fromUser_opcode_next <= OPCODE_PUSH;
  next_state <= CALL_EXIT;

when CALL_EXIT => -- 000A
  -- Immediately exit
  fromUser_function_next <= FUNCTION_HTHREAD_EXIT;
  fromUser_value_next <= Z32(0 to 15) & FUNCTION_EXIT;
  fromUser_opcode_next <= OPCODE_CALL;
  next_state <= FUNCTION_EXIT;

when FUNCTION_EXIT => -- 000B
  next_state <= FUNCTION_EXIT;
when ERROR_STATE =>
  next_state <= ERROR_STATE;

=======================================================================
-- Factorial function
-- computes the factorial of the input parameter using recursion
=======================================================================

when FACTORIAL_1 => -- 0101
  -- Read the passed in parameter
  fromUser_value_next <= Z32;
  fromUser_opcode_next <= OPCODE_POP;
  next_state <= FACTORIAL_2;

when FACTORIAL_2 => -- 0102
  -- store the passed in parameter in a register
  varOne_next <= intrfc2thrd_value;
  next_state <= FACTORIAL_3;

when FACTORIAL_3 => -- 0103
  -- Declare one variable
  fromUser_value_next <= x"00000001";
fromUser_opcode_next <= OPCODE_DECLARE;
next_state <= FACTORIAL_4;

when FACTORIAL_4 => -- 0104
-- store the register as a saved variable
fromUser_value_next <= varOne;
fromUser_address_next <= Z32;
fromUser_opcode_next <= OPCODE_WRITE;
next_state <= FACTORIAL_5;

when FACTORIAL_5 => -- 0105
-- check if param <= 1
case varOne is
when x"00000001" =>
-- return a 1
fromUser_value_next <= x"00000001";
fromUser_opcode_next <= OPCODE_RETURN;
next_state <= FACTORIAL_END;
when Z32 =>
-- return a 1
fromUser_value_next <= x"00000000";
fromUser_opcode_next <= OPCODE_RETURN;
next_state <= FACTORIAL_END;
when others =>
-- recursively call factorial, prepare by pushing param-1
fromUser_value_next <= (varOne - 1);
fromUser_opcode_next <= OPCODE_PUSH;
next_state <= FACTORIAL_6;
end case;

when FACTORIAL_6 => -- 0106
-- recursively call factorial
fromUser_value_next <= Z32(0 to 15) & FACTORIAL_7;
fromUser_function_next <= FACTORIAL_1;
fromUser_opcode_next <= OPCODE_CALL;
next_state <= FACTORIAL_7;

when FACTORIAL_7 => -- 0107
-- read the return value, save to a register
-- TODO, change this to multiplication
varTwo_next <= intrfc2thrd_value;
-- read the save variable
fromUser_address_next <= Z32;
fromUser_opcode_next <= OPCODE_READ;
next_state <= FACTORIAL_8;

when FACTORIAL_8 => -- 0108
-- store the variable back to varOne
varOne_next <= intrfc2thrd_value;
next_state <= FACTORIAL_9;

when FACTORIAL_9 => -- 0109
-- return
fromUser_value_next <= varTwo + varOne;
fromUser_opcode_next <= OPCODE_RETURN;
next_state <= FACTORIAL_END;

when FACTORIAL_END => -- 010A
-- if everything is working, should never reach this state
next_state <= FACTORIAL_END;

when others =>
next_state <= ERROR_STATE;
end case;
end if;
end process HWTUL_STATE_MACHINE;
end architecture IMP;
State Machine Implementations

The following two sections detail the implementation of the HWTI. The two sections above, give the specifications of the HWTI, 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.

The HWTI runs off of two state machines. Loosely speaking, the system state machine controls and monitors the registers attached to the bus, the user state machine controls and monitors the registers associated with the user logic and implements the function call stack, system calls, and master bus transactions. A block diagram of the state machine implementation is to the right.

Process Descriptions

The VHDL code for HWTI 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 HWTI).

**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_TIMER**

The purpose of the HWTI_TIMER process is to count the number of clock cycles the HW TI runs for. The timer starts on a RUN command, and stops when an hthread_exit is called. The timer may be viewed by the system via the system’s interface timer register.

**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 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 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_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 than 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 hardware thread 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 hardware thread 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 hardware thread 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 hardware thread 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 therefore 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 hardware thread has the lock. If the hardware thread has the lock, the user_status is changed to RUN, and the system's status is not updated. However, if the hardware thread 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 hardware thread 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, with the exception of read and write, 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. Timings for read and write, both global and local access, were taken from on the board execution.

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>Load (global)</td>
<td>60</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>Load (local)</td>
<td>5</td>
<td>Time from the HWTUL issuing the opcode, to the time the HWTI returns the user_status to RUN, including BRAM transaction time.</td>
</tr>
<tr>
<td>opcode</td>
<td>Clock Cycles</td>
<td>Comment</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Store (global)</td>
<td>32</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>Store (local)</td>
<td>4</td>
<td>Time from the HWTUL issuing the opcode, to the time the HWTI returns the user_status to RUN, including BRAM 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 Manager and the system status changes to EXIT.</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 hardware thread to the offset. For example, the address of the command register, for a hardware thread 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>
</tbody>
</table>
### Register Offset Address

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset Address</th>
</tr>
</thead>
<tbody>
<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 hardware thread 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>pthread</th>
<th>void * basicThread( void * argument )</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread</td>
<td>void * basicThread( void * argument )</td>
</tr>
<tr>
<td>HIF</td>
<td>function basicThread 1</td>
</tr>
<tr>
<td>VHDL</td>
<td>library IEEE;</td>
</tr>
<tr>
<td></td>
<td>use IEEE.std_logic_1164.all;</td>
</tr>
<tr>
<td></td>
<td>use IEEE.std_logic_arith.all;</td>
</tr>
<tr>
<td></td>
<td>use IEEE.std_logic_unsigned.all;</td>
</tr>
<tr>
<td></td>
<td>use IEEE.std_logic_misc.all;</td>
</tr>
<tr>
<td></td>
<td>library Unisim;</td>
</tr>
<tr>
<td></td>
<td>use Unisim.all;</td>
</tr>
<tr>
<td></td>
<td>-- Port declarations</td>
</tr>
<tr>
<td></td>
<td>-- Definition of Ports:</td>
</tr>
<tr>
<td></td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>-- Misc. Signals</td>
</tr>
<tr>
<td></td>
<td>-- clock</td>
</tr>
<tr>
<td></td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>-- HWTI to HWTUL interconnect</td>
</tr>
<tr>
<td></td>
<td>-- intrfc2thrd_status</td>
</tr>
<tr>
<td></td>
<td>-- intrfc2thrd_result</td>
</tr>
<tr>
<td></td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>-- HWTUL to HWTI interconnect</td>
</tr>
<tr>
<td></td>
<td>-- thrd2intrfc_opcode</td>
</tr>
<tr>
<td></td>
<td>-- thrd2intrfc_argument_one</td>
</tr>
<tr>
<td></td>
<td>-- thrd2intrfc_argument_two</td>
</tr>
<tr>
<td></td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>-- Thread Manager Entity section</td>
</tr>
<tr>
<td></td>
<td>entity user_logic_hwtul is</td>
</tr>
<tr>
<td></td>
<td>port (</td>
</tr>
<tr>
<td></td>
<td>clock : in std_logic;</td>
</tr>
</tbody>
</table>
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";
custom OPCODE_READ : std_logic_vector(0 to 7) := x"02";
custom OPCODE_WRITE : std_logic_vector(0 to 7) := x"03";
custom OPCODE_HTHREAD_SELF : std_logic_vector(0 to 7) := x"04";
custom OPCODE_HTHREAD_YIELD : std_logic_vector(0 to 7) := x"05";
custom OPCODE_HTHREAD_MUTEX_LOCK : std_logic_vector(0 to 7) := x"06";
custom OPCODE_HTHREAD_MUTEX_UNLOCK : std_logic_vector(0 to 7) := x"07";

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

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 =>
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.

<table>
<thead>
<tr>
<th>pthread</th>
<th>int * fooAddr = (int *) argument;</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread</td>
<td>int * fooAddr = (int *) argument;</td>
</tr>
<tr>
<td>HIF</td>
<td>readarg R1 0</td>
</tr>
<tr>
<td>VHDL</td>
<td>when IDLE =&gt;</td>
</tr>
<tr>
<td></td>
<td>case intrfc2thrd_status is</td>
</tr>
<tr>
<td></td>
<td>when USER_STATUS_RUN =&gt;</td>
</tr>
<tr>
<td></td>
<td>fooAddr_next &lt;= intrfc2thrd_result;</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= MUTEX_LOCK_1;</td>
</tr>
<tr>
<td></td>
<td>when others =&gt;</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= IDLE;</td>
</tr>
<tr>
<td></td>
<td>end case;</td>
</tr>
</tbody>
</table>

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>
<tr>
<td>HIF</td>
<td>addressof R2 fooMutex</td>
</tr>
<tr>
<td></td>
<td>syscall mutex lock R2</td>
</tr>
<tr>
<td>VHDL</td>
<td>-- Lock mutex zero</td>
</tr>
<tr>
<td></td>
<td>when MUTEX_LOCK_1 =&gt;</td>
</tr>
<tr>
<td></td>
<td>-- Tell the HWTI, which mutex to lock</td>
</tr>
<tr>
<td></td>
<td>argOne_next &lt;= Z32;</td>
</tr>
<tr>
<td></td>
<td>opcode_next &lt;= OPCODE_HTHREAD_MUTEX_LOCK;</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>pthread</th>
<th>int fooValue = *fooAddr;</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread</td>
<td>int fooValue = *fooAddr;</td>
</tr>
<tr>
<td>HIF</td>
<td>gread R3 R1</td>
</tr>
<tr>
<td>VHDL</td>
<td>-- Read the value at fooAddr</td>
</tr>
<tr>
<td></td>
<td>when READ_MEMORY_1 =&gt;</td>
</tr>
<tr>
<td></td>
<td>-- Tell the HWTI what address to read</td>
</tr>
<tr>
<td></td>
<td>argOne_next &lt;= fooAddr;</td>
</tr>
<tr>
<td></td>
<td>opcode_next &lt;= OPCODE_READ;</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= READ_MEMORY_2;</td>
</tr>
<tr>
<td></td>
<td>when READ_MEMORY_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;= READ_MEMORY_3;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= READ_MEMORY_2;</td>
</tr>
<tr>
<td></td>
<td>end if;</td>
</tr>
<tr>
<td></td>
<td>when READ_MEMORY_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;= SELF_1;</td>
</tr>
<tr>
<td></td>
<td>fooVal_next &lt;= intrfc2thrd_result;</td>
</tr>
</tbody>
</table>
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.

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.
hthread  hthread_yield();

HIF  syscall yield

VHDL

-- Call hthread_yield
when YIELD_1 =>
  -- Yield the CPU, in HW, returns immediately to resume execution.
  opcode_next <= OPCODE_HTHREAD_YIELD;
  next_state <= YIELD_2;

when YIELD_2 =>
  -- Wait for the HWTI to ack
  if ( intrfc2thrd_status = USER_STATUS_ACK ) then
    opcode_next <= OPCODE_NOOP;
    next_state <= YIELD_3;
  else
    next_state <= YIELD_2;
  end if;

when YIELD_3 =>
  -- Wait for the HWTI to tell us to start running again
  if ( intrfc2thrd_status = USER_STATUS_RUN ) then
    next_state <= WRITE_MEMORY_1;
  else
    next_state <= YIELD_3;
  end if;

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.

pthread  *fooAddr = fooValue;

hthread  *fooAddr = fooValue;

HIF  gwrite R1 R3

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;
Unlocking a Mutex

Unlocking a mutex is almost identical to locking a mutex, with the obvious exception of unlock versus 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>
<tr>
<td>VHDL</td>
<td>-- Unlock mutex zero</td>
</tr>
<tr>
<td></td>
<td>when MUTEX_UNLOCK_1 =&gt;</td>
</tr>
<tr>
<td></td>
<td>-- Tell the HWTI what mutex to lock</td>
</tr>
<tr>
<td></td>
<td>argOne_next &lt;= Z32;</td>
</tr>
<tr>
<td></td>
<td>opcode_next &lt;= OPCODE_HTHREAD_MUTEX_UNLOCK;</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= MUTEX_UNLOCK_2;</td>
</tr>
<tr>
<td></td>
<td>when MUTEX_UNLOCK_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;= MUTEX_UNLOCK_3;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= MUTEX_UNLOCK_2;</td>
</tr>
<tr>
<td></td>
<td>end if;</td>
</tr>
<tr>
<td></td>
<td>when MUTEX_UNLOCK_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>--When we start running again, we know we gave up the lock</td>
</tr>
<tr>
<td></td>
<td>if ( intrfc2thrd_status = USER_STATUS_RUN ) then</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= EXIT_INIT;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= MUTEX_UNLOCK_3;</td>
</tr>
<tr>
<td></td>
<td>end if;</td>
</tr>
</tbody>
</table>

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.

```plaintext
pthread
}  

hthread
}  

HIF

VHDL
when others =>
  next_state <= IDLE;
end case;
end process HWTUL_STATE_MACHINE;
end architecture IMP;
```