HWTI Introduction

The Hybridthreads Computational Platform (Hthreads) is an integrated tool suite and hardware/software co-design runtime kernel that allows programmers to drive the generation of custom multi-core systems on programmable chip architectures from a high level multi-threaded programming model. With Hthreads, programmers design, implement, and test their application with a pthreads compatible API. Their code can then be compiled to run on either a general purpose CPU or synthesized into a thread specific hardware core, known as a hardware thread. Communication and synchronization between software and hardware threads, as specified by the programmer’s code in the form of Hthread API calls, are implemented for the programmer in either library routines for software threads, that are linked in during the build process, or by the Hardware Thread Interface (HWTI), that is synthesized with each hardware thread. The purpose of this document is to describe the protocol, features, and design of the HWTI version three. Details of the Hthread system in general may be found at [http://www.ittc.ku.edu/hybridthreads](http://www.ittc.ku.edu/hybridthreads).

The HWTI features two interfaces allowing accessibility to and from the Hthread system and the hardware thread. The first interface, known formally as the System Level Application Programming Interface, or system interface for short, is a set of memory mapped registers that allows the system kernel to interact with the hardware thread in accordance to the Hthread system calls. The second interface, known as the User Logic Application Programming Interface, or user interface for short, is a set of control, request, and data registers. The hardware thread’s user logic interacts with the user interface in a protocol adopted from function calling in high level languages. Additionally, the HWTI features a local memory, accessible through both interfaces. The local memory not only provides a fast global memory accessible by the hardware thread, but also allows the HWTI to support a traditional function call stack, variable declaration, and dynamic memory allocation. The HWTI is designed to be system independent, allowing programmers to develop code to interact with the HWTI without knowledge of the exact implementation of the HWTI or the low level details of the target FPGA.

The following table lists the supported features of the HWTI and when they were made available. Version one of the HWTI was a proof of concept that extended the concept of threads, including thread communication and synchronization, to include threads that ran exclusively in hardware. Version two modified the system and user interface to more closely match the supported Hthread library calls. Version three, modified the user interface to include generalized support for language translation and user defined functions, in addition to providing a more complete set of Hthread APIs.

<table>
<thead>
<tr>
<th>HWTI Feature</th>
<th>HWTI V1</th>
<th>HWTI V2</th>
<th>HWTI V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Independent Hardware Thread</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Standardized System Independent Interface</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Hthread Compatable System Interface</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Locally Accessable Global Memory</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
The remainder of this document is organized in four sections. The first section describes the theoretical and research aspects behind the HWTI. The second 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 third section will be a implementation description. 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.

### Feature Description Goals

#### FPGA Accessability

Reconfigurable computing has been a discipline of computer engineering for nearly a decade. Initially two problems where identified that needed to be solved before reconfigurable computing could become mainstread. The first problem, the size and speed of reconfigurable chips, is largely solved thanks to advancements in fabrication techniques. The second problem, easy accessability to the FPGA fabric has largely gone unsolved. The goal of this problem is to allow engineers to implement an application design within the FPGA, and allow communication to and from it without an undue burdon on the engineer’s time. Historically this is addressed by introducing custom register sets in the FPGA core that is accessable as memory mapped FIFO queues. Although this technique allows access to the FPGA from the CPU, it forces excessive design and evaluation time of the engineer who must custom build each core and each software driver to the core. The HWTI takes a different approach to solving the accessability problem,

<table>
<thead>
<tr>
<th>Feature Description</th>
<th>Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Timer</td>
<td>X</td>
</tr>
<tr>
<td>Mutex Synchronization Controls</td>
<td>X</td>
</tr>
<tr>
<td>Condition Variable Synchronization Controls</td>
<td>X</td>
</tr>
<tr>
<td>Load and Store Access to Global Memory</td>
<td>X</td>
</tr>
<tr>
<td>Function Call Stack</td>
<td>X</td>
</tr>
<tr>
<td>Support for User Defined Functions, with Arbitrary Parameter List</td>
<td>X</td>
</tr>
<tr>
<td>Recursive User Defined Function Calls</td>
<td>X</td>
</tr>
<tr>
<td>Mutex/Mutex Attribute Initiation/Destruction</td>
<td>X</td>
</tr>
<tr>
<td>Condition Variable/Condition Variable Attribute Initition/Destruction</td>
<td>X</td>
</tr>
<tr>
<td>Declaration of Local Variables</td>
<td>X</td>
</tr>
<tr>
<td>Addressof Operation (for local variables)</td>
<td>X</td>
</tr>
<tr>
<td>Dynamic Memory Allocation/Deallocation</td>
<td>X</td>
</tr>
<tr>
<td>Thread Creation and Join Controls</td>
<td>Planned</td>
</tr>
</tbody>
</table>
namely it provides a standard register set the CPU may use to access an FPGA core, while also providing a standard port map and protocol to allow hardware cores to communicate with the CPU without needing knowledge of the underlying FPGA or bus connect.

The system interface is designed to work with the semantics of the hthread_create and hthread_join system calls. When a thread is created through the hthread_create system call, four parameters are passed to the system: a pointer to a thread structure, a pointer to thread attribute structure, pointer to the function to run, and a pointer to the thread’s arguments. The pointer to the thread attribute is for the callee function, and has no meaning for the thread being created. The thread attribute structure contains the address of the hardware thread to start. The system uses this information, instead of the pointer to the function, to know which hardware thread to start. The thread’s argument is passed to the hardware thread through a memory mapped register. When a hardware thread exits, any return value is stored in a separate memory mapped register that is passed back to the parent thread through the semantics of the hthread_join system call.

The user interface is designed to give the hardware thread’s user logic access to operations very similar to common operations from ANSI C. These operations are pointer referencing, address of operation, variable declaration, and function calls, which include parameter passing.

**Pointer Support**

In order to be independent from the CPU, a hardware thread must not rely on the CPU to feed or retrieve information for it. The hardware thread therefore must be able to retrieve data from memory. To achieve this, the HWTI supports the notion of pointer referencing and dereferencing. The read a memory location, the hardware thread passes the address of the data to read to the HWTI via the user interface. The HWTI in turn performs a standard read bus operation, the memory location responds, and the HWTI passes the value back to the user logic. To perform a write, the user logic passes both the address and the value to the HWTI, which in turn initiates the bus write operation. The bus protocol is abstracted from the user by the HWTI. In this manner, a hardware thread has access to global memory without having to know any low level system details. The user logic only has to know how to interact with the HWTI user interface.

**Distributed Global Memory**

The memory latency problem, the delay between a load or store request from the CPU to memory, is a well studied and understood problem. Hardware threads, who operate similarly to a CPU suffer from the same problem. If a hardware thread could only operate on data that is physically located off chip, its performance would be greatly degraded. Caches are added to CPUs to solve this problem, however for hardware threads that are resource constrained, caches are too expensive to create. The problem is then, how can hardware threads be given a fast memory to access, without instantiating caches.

The HWTI, to solve this problem, instantiates a fast local memory that is globally accessible. This memory is accessible by other Hthread cores, including the CPU, through standard bus read and write protocols. Conversely, the memory is accessible to the hardware thread’s user logic through HWTI user interface using the same protocol as would be used for accessing traditional global memory. On each load or store request, the HWTI checks to see if the address range requested is local (within the HWTI) or global (outside the current HWTI). The memory is instantiated using the on chip memory embedded within the FPGA fabric.
Function Call Stack, Function Calls, and Local Variable Declaration

One of the objectives of the Hthread Computational Platform is to create an unmodified ANSI C to VHDL translator. Each thread specified in the C language can be automatically and correctly synthesized into a custom hardware core, in the form of a hardware thread. To achieve this, the C code is translated into an equivalent VHDL state machine using the Hthread Compiler (HTC). However, a state machine, that can maintain control flow and arithmetic operations, can not support memory or function calling operations without assistance. These include, pointer support, variable declaration, function calls, and address of operations. The HWTI provides the abstraction for these semantics.

The semantics of pointer operations were provided earlier in this document. Function calls are achieved by the HWTI using the hardware thread’s local memory to create a function call stack. The HWTI’s function call stack works analogously to software function call stacks. There are two key differences. First, the stack and frame pointer are maintained as registers within the HWTI, pointing to local memory instead of traditional global memory. The second difference is that instead of pushing the contents of the program counter during a function call, as done in software, the HWTI pushes the user logic’s state machine’s return state. This mechanism consequently allows the HWTI to support recursive function calls in hardware. The recursive depth of a function is only limited to the availability of local memory.

To declare local variables, the user logic issues a “declare” request, with the number of bytes in memory it wants to set aside for local variables. The HWTI then increments its stack pointer the specified number of bytes. Instead of a variable name, as done in software, the user logic uses index numbers that coorespond to the declared variables. Since the variables are declared and granted space with the HWTI’s local memory, they all have memory addresses. The address of operator works by converting the index number into its equivalent memory address, taking into account the HWTI’s base address and current frame pointer.

Dynamic Memory Allocation

Dynamic memory allocation in hardware has proven to be a difficult task. Dynamic memory allocation is achievable only if the hardware can access an allocator. Since the standard allocator is implemented through C’s standard library, hardware access can only be given indirectly. To address this problem, the HWTI implements a light-weight version of malloc and free. To implement malloc and free the HWTI adds two limitations. First the same thread that allocates memory must deallocate it. Second, since the dynamic memory is allocated within the local memory, there is a limit to the size and number of memory segments than can be allocated. The memory the HWTI allocates is pre-allocated in 8B, 32B, and 1024B segments at the top of the local address range. These sizes were selected to assist with the dynamic creation of mutexes, condition variables, and threads. Because the memory is pre-allocated, it avoids requiring an implementation of defraging routines. When the user logic calls malloc, the HWTI selects the smallest appropriate preallocated memory space, and returns its address to the user. The HWTI marks the memory used in a malloc state table. If the requested memory size is larger than 1024B, the HWTI allocates this space by decrementing a heap pointer the specified amount and returning the appropriate address to the user. Currently the user logic may request only a single segment of memory larger than 1024B. When the user logic calls free, the HWTI marks the appropriate malloc state table entry as free.
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 (i.e. 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).

• EXITED_WITH_OVERFLOW (0100 0000): During operation of the HWT, if the stack space and heap space collide, an overflow will occur. This may occur either because the function call stack grows to large, or the user requests more dynamic memory than available. The HWT will terminate, call exit on the Thread Manager, and set its status to OVERFLOW.

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 separated from the other two. However, an implementation of the user interface must merge the sub-interfaces into a single interface. The HWTI therefore 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 go/wait 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 go/wait signal is low (a WAIT signal). The user logic must continue to wait until the go/wait signal returns high. Any appropriate response will be available, in either the **address** or **value** registers, to the user logic when the go/wait 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, there 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 an 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 operator, for example &x.

### Function 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 supported system and library calls are listed in a section of their own below. 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

<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>
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), 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 there 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 HWTI manages the execution and delays of the HWTUL. This sub-interface has two registers, a goWait register, and an 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 necessarily 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.

Supported System and Library Function Calls

The HWTI supports a key subject of the hthread system calls. In general, the HWTI supports default behaviour of all the hthread system calls. This subset will most software threads to be translated into hardware threads without loss of functionality. The HWTI also supports dynamic memory allocation, in the form of malloc, calloc, and free, as well as memory transfers, in the form of memcpy.

In the table below, each supported system and library call is demonstrated with an example. The vhdl code is an example of how the user logic thread would make the call. To limit the space required to show each example, only the states where the call is set up and made are listed. It is assumed all constants are defined, registers have appropriate value, and the user logic is using a two process state machine similar to the one given in User Logic State Machine Example.

hthread Supported System Calls

<table>
<thead>
<tr>
<th>Library Call</th>
<th>Function Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread_attr_init()</td>
<td>0x8000</td>
<td>Not yet supported.</td>
</tr>
<tr>
<td>hthread_attr_destroy()</td>
<td>0x8001</td>
<td>Not yet supported.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread_create()</td>
<td>0x8010</td>
<td>Not yet supported.</td>
</tr>
<tr>
<td>hthread_join()</td>
<td>0x8011</td>
<td>Not yet supported.</td>
</tr>
<tr>
<td>hthread_self()</td>
<td>0x8012</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>when STATE 1 =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>make the system call</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_function &lt;= FUNCTION_HTHREAD_SELF;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_value &lt;= Z32(0 to 15) &amp; U_STATE_2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_opcode &lt;= OPCODE_CALL;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_state &lt;= WAIT_STATE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when STATE 2 =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>returns the thread id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg1.next &lt;= toUser_value;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_state &lt;= STATE_3;</td>
</tr>
<tr>
<td>hthread_equal()</td>
<td>0x8014</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>when STATE 1 =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push the first parameter, the thread ID to compare against</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_value &lt;= reg1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_opcode &lt;= OPCODE_PUSH;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_state &lt;= WAIT_STATE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return_state_next &lt;= STATE_2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when STATE 2 =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push the first parameter, the thread ID to compare against</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_function &lt;= FUNCTION_HTHREAD_EQUAL;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_value &lt;= Z32(0 to 15) &amp; U_STATE_3;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_opcode &lt;= OPCODE_CALL;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_state &lt;= WAIT_STATE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when STATE 3 =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>returns SUCCESS (0) or FAILURE (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reg2.next &lt;= toUser_value;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_state &lt;= STATE_4;</td>
</tr>
<tr>
<td>hthread_exit()</td>
<td>0x8015</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>when STATE 1 =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push the first parameter, the return value, typically a pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_value &lt;= reg1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_opcode &lt;= OPCODE_PUSH;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_state &lt;= WAIT_STATE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return_state_next &lt;= STATE_2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when STATE 2 =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push the first parameter, the return value, typically a pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_value &lt;= Z32;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_opcode &lt;= OPCODE_CALL;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_state &lt;= WAIT_STATE;</td>
</tr>
<tr>
<td>hthread_exit_error()</td>
<td>0x8016</td>
<td>Not part of hthread or pthread standard.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added for debug purposes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when STATE 1 =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push the first parameter, the return value, typically a pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_value &lt;= reg1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thrd2intrfc_opcode &lt;= OPCODE_PUSH;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next_state &lt;= WAIT_STATE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return_state_next &lt;= STATE_2;</td>
</tr>
</tbody>
</table>
when STATE_2 =>
   -- make the system call, note that the exit thread never returns
   thrd2intrfc_function <= FUNCTION_HTHREAD_EXIT_ERROR;
   thrd2intrfc_value <= Z32;
   thrd2intrfc_opcode <= OPCODE_CALL;
   next_state <= WAIT_STATE;

hthread_mutexattr_init() 0x8020
hthread_mutexattr_destroy() 0x8021
hthread_mutexattr_getnum() 0x8023
hthread_mutexattr_setnum() 0x8022
hthread_mutex_init() 0x8030
hthread_mutex_destroy() 0x8031
hthread_mutex_lock() 0x8032

when STATE_1 =>
   -- push the first parameter, address of the mutex to lock
   thrd2intrfc_value <= reg1;
   thrd2intrfc_opcode <= OPCODE_PUSH;
   next_state <= WAIT_STATE;
   return_state_next <= STATE_2;
when STATE_2 =>
   -- make the system call
   thrd2intrfc_function <= FUNCTION_HTHREAD_MUTEX_LOCK;
   thrd2intrfc_value <= Z32(0 to 15) & U_STATE_3;
   thrd2intrfc_opcode <= OPCODE_CALL;
   next_state <= WAIT_STATE;
when STATE_3 =>
   -- returns SUCCESS (0) if the lock is granted, FAILURE (1) is locked a
   mutex the current hardware thread already locked.
   reg2_next <= toUser_value;
   next_state <= STATE_4;

hthread_mutex_unlock() 0x8033

when STATE_1 =>
   -- push the first parameter, address of the mutex to unlock
   thrd2intrfc_value <= reg1;
   thrd2intrfc_opcode <= OPCODE_PUSH;
   next_state <= WAIT_STATE;
   return_state_next <= STATE_2;
when STATE_2 =>
   -- make the system call
   thrd2intrfc_function <= FUNCTION_HTHREAD_MUTEX_UNLOCK;
   thrd2intrfc_value <= Z32(0 to 15) & U_STATE_3;
   thrd2intrfc_opcode <= OPCODE_CALL;
   next_state <= WAIT_STATE;

hthread_mutex_trylock() 0x8034

when STATE_1 =>
   -- push the first parameter, address of the mutex to lock
   thrd2intrfc_value <= reg1;
   thrd2intrfc_opcode <= OPCODE_PUSH;
   next_state <= WAIT_STATE;
return_state_next <= STATE_2;

when STATE_2 =>
  -- make the system call
  thrd2intrfc_function <= FUNCTION_HTHREAD_MUTEX_TRYLOCK;
  thrd2intrfc_value <= Z32(0 to 15) & U_STATE_3;
  thrd2intrfc_opcode <= OPCODE_CALL;
  next_state <= WAIT_STATE;

when STATE_3 =>
  -- returns SUCCESS (0) if the lock is granted, FAILURE (1) if the mutex
  is already locked
  reg2.next <= toUser_value;
  next_state <= STATE_4;

hthread_condattr_init() 0x8040 Not yet supported.

hthread_condattr_destroy() 0x8041 Not yet supported.

hthread_condattr_setnum() 0x8042 Not yet supported.

hthread_condattr_getnum() 0x8043 Not yet supported.

hthread_cond_init() 0x8050 Not yet supported.

hthread_cond_destroy() 0x8051 Not yet supported.

hthread_cond_signal() 0x8052

when STATE_1 =>
  -- push the first parameter, address of the cond var to signal
  thrd2intrfc_value <= reg1;
  thrd2intrfc_opcode <= OPCODE_PUSH;
  next_state <= WAIT_STATE;
  return_state_next <= STATE_2;

when STATE_2 =>
  -- make the system call
  thrd2intrfc_function <= FUNCTION_HTHREAD_COND_SIGNAL;
  thrd2intrfc_value <= Z32(0 to 15) & U_STATE_3;
  thrd2intrfc_opcode <= OPCODE_CALL;
  next_state <= WAIT_STATE;

hthread_cond_broadcast() 0x8053

when STATE_1 =>
  -- push the first parameter, address of the cond var to broadcast
  thrd2intrfc_value <= reg1;
  thrd2intrfc_opcode <= OPCODE_PUSH;
  next_state <= WAIT_STATE;
  return_state_next <= STATE_2;

when STATE_2 =>
  -- make the system call
  thrd2intrfc_function <= FUNCTION_HTHREAD_COND_BROADCAST;
  thrd2intrfc_value <= Z32(0 to 15) & U_STATE_3;
  thrd2intrfc_opcode <= OPCODE_CALL;
  next_state <= WAIT_STATE;

hthread_cond_wait() 0x8054

when STATE_1 =>
  -- push the second parameter, address of the mutex
  thrd2intrfc_value <= reg1;
  thrd2intrfc_opcode <= OPCODE_PUSH;
### HWTI Documentation

**Version 3.0** of

**Documentation Revision 3.0.b**

**Date: 20 September 2006**

---

<table>
<thead>
<tr>
<th>Function</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc()</td>
<td>0xA000</td>
<td>Limited support for allocating large chunks of memory. Works best when allocating chunks smaller than 1024B. Returns 0, when can not allocate requested space.</td>
</tr>
<tr>
<td>calloc()</td>
<td>0xA0001</td>
<td>Limited support for allocating large chunks of memory. Works best when allocating chunks smaller than 1024B. Returns 0, when can not allocate requested space. Does not initialize data with zeros.</td>
</tr>
</tbody>
</table>

```vhdl
next_state <= WAIT_STATE;
return_state_next <= STATE_2;
when STATE_2 =>
  -- push the first parameter, address of the condition variable
  thrd2intrfc_value <= reg2;
  thrd2intrfc_opcode <= OPCODE_PUSH;
  next_state <= WAIT_STATE;
  return_state_next <= STATE_3;
when STATE_3 =>
  -- make the system call
  thrd2intrfc_function <= FUNCTION_HTHREAD_COND_WAIT;
  thrd2intrfc_value <= Z32(0 to 15) & U_STATE_4;
  thrd2intrfc_opcode <= OPCODE_CALL;
  next_state <= WAIT_STATE;

malloc() 0xA000

when STATE_1 =>
  -- push the number of bytes to allocate
  thrd2intrfc_value <= reg1;
  thrd2intrfc_opcode <= OPCODE_PUSH;
  next_state <= WAIT_STATE;
  return_state_next <= STATE_2;
when STATE_2 =>
  -- make the system call
  thrd2intrfc_function <= FUNCTION_MALLOC;
  thrd2intrfc_value <= Z32(0 to 15) & U_STATE_3;
  thrd2intrfc_opcode <= OPCODE_CALL;
  next_state <= WAIT_STATE;
when STATE_3 =>
  -- returns the address of the allocated memory
  reg2_next <= toUser_value;
  next_state <= STATE_4;

calloc() 0xA0001

when STATE_1 =>
  -- push the second parameter, the number of bytes of the structure
  thrd2intrfc_value <= reg1;
  thrd2intrfc_opcode <= OPCODE_PUSH;
  next_state <= WAIT_STATE;
  return_state_next <= STATE_2;
when STATE_2 =>
  -- push the first parameter, the number of structures to allocate
  thrd2intrfc_value <= reg2;
  thrd2intrfc_opcode <= OPCODE_PUSH;
  next_state <= WAIT_STATE;
  return_state_next <= STATE_3;
when STATE_3 =>
  -- make the system call
  thrd2intrfc_function <= FUNCTION_CALLOC;
```
thrd2intrfc_value <= Z32(0 to 15) & U_STATE_4;
thrd2intrfc_opcode <= OPCODE_CALL;
next_state <= WAIT_STATE;
when STATE_4 =>
  -- returns the address of the allocated memory
  reg2_next <= toUser_value;
  next_state <= STATE_4;

free() 0xA002 May only deallocate data allocated by the current hardware thread. A software thread may not deallocate data allocated by a hardware thread.
when STATE_1 =>
  -- push the first parameter, address of the memory to de-allocate
  thrd2intrfc_value <= reg1;
  thrd2intrfc_opcode <= OPCODE_PUSH;
  next_state <= WAIT_STATE;
  return_state_next <= STATE_2;
when STATE_2 =>
  -- make the system call
  thrd2intrfc_function <= FUNCTION_FREE;
  thrd2intrfc_value <= Z32(0 to 15) & U_STATE_3;
  thrd2intrfc_opcode <= OPCODE_CALL;
  next_state <= WAIT_STATE;

memcpy() 0xA1000 Does not check for pointer overlap. Number of bytes must be an even interval of 4.
when STATE_1 =>
  -- push the third parameter, the number of bytes to copy
  thrd2intrfc_value <= reg1;
  thrd2intrfc_opcode <= OPCODE_PUSH;
  next_state <= WAIT_STATE;
  return_state_next <= STATE_2;
when STATE_2 =>
  -- push the second parameter, the address to copy to
  thrd2intrfc_value <= reg2;
  thrd2intrfc_opcode <= OPCODE_PUSH;
  next_state <= WAIT_STATE;
  return_state_next <= STATE_3;
when STATE_3 =>
  -- push the first parameter, the address to copy from
  thrd2intrfc_value <= reg3;
  thrd2intrfc_opcode <= OPCODE_PUSH;
  next_state <= WAIT_STATE;
  return_state_next <= STATE_4;
when STATE_4 =>
  -- make the system call
  thrd2intrfc_function <= FUNCTION_MEMCPY;
  thrd2intrfc_value <= Z32(0 to 15) & U_STATE_5;
  thrd2intrfc_opcode <= OPCODE_CALL;
  next_state <= WAIT_STATE;
User Logic State Machine Example

The following code is an example state machine that uses many of the supported opcodes and system calls 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;

-- Port declarations
-- Definition of Ports:
-- Misc. Signals
--   clock
-- HWTI to HWTUL interconnect
--   intrfc2thrd_address 32 bits memory function
--   intrfc2thrd_value 32 bits memory
--   intrfc2thrd_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

entity user_logic_hwtul is
  port (
    clock : in std_logic;
    intrfc2thrd_address : in std_logic_vector(0 to 31);
    intrfc2thrd_value : in std_logic_vector(0 to 31);
    intrfc2thrd_function : in std_logic_vector(0 to 15);
    intrfc2thrd_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

-- type state_machine is (  
FUNCTION_RESET,  
FUNCTION_START,  
READ_ARGUMENT,  
PUSH_ARGUMENT,  
call_factorial,  
READ_FACTORIAL_RETURN,  
PUSH_RETURN,  
call_exit,  
factorial_1,  
factorial_2,  
factorial_3,  
factorial_4,  
factorial_5,  
factorial_6,  
factorial_7,  
factorial_8,  
factorial_9,  
factorial_end,  
error_state,  
wait_state );

-- Function definitions
constant U_FUNCTION_RESET : std_logic_vector(0 to 15) := x"0000";
constant U_FUNCTION_WAIT : std_logic_vector(0 to 15) := x"0001";
constant U_FUNCTION_USER_SELECT : std_logic_vector(0 to 15) := x"0002";
constant U_FUNCTION_START : std_logic_vector(0 to 15) := x"0003";
constant U_READ_FACTORIAL_RETURN : std_logic_vector(0 to 15) := x"0007";
constant U_FACTORIAL_1 : std_logic_vector(0 to 15) := x"0101";
constant U_FACTORIAL_7 : std_logic_vector(0 to 15) := x"0107";

-- 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_DESTROY : std_logic_vector(0 to 15) := x"8011";
-- constant FUNCTION_HTHREAD_EXIT : std_logic_vector(0 to 15) := x"8012";
-- constant FUNCTION_HTHREAD!important : std_logic_vector(0 to 15) := x"8013";
-- constant FUNCTION_HTHREAD_EQUAL : std_logic_vector(0 to 15) := x"8014";
-- constant FUNCTION_HTHREAD_YIELD : std_logic_vector(0 to 15) := x"8015";
constant FUNCTION_HTHREAD_MUTEX_INIT : std_logic_vector(0 to 15) := x"8020";
constant FUNCTION_HTHREAD_MUTEX_DESTROY : std_logic_vector(0 to 15) := x"8021";
constant FUNCTION_HTHREAD_MUTEX_SETNUM : std_logic_vector(0 to 15) := x"8022";
constant FUNCTION_HTHREAD_MUTEX_GETNUM : std_logic_vector(0 to 15) := x"8023";
constant FUNCTION_HTHREAD_MUTEX_LOCK : std_logic_vector(0 to 15) := x"8024";
constant FUNCTION_HTHREAD_MUTEX_UNLOCK : std_logic_vector(0 to 15) := x"8025";
constant FUNCTION_HTHREAD_MUTEX_TRYLOCK : std_logic_vector(0 to 15) := x"8026";
constant FUNCTION_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 FUNCTION_MEMCPY : std_logic_vector(0 to 15) := x"A003";
-- user_opcode Constants
constant OPCODE_NOOP : std_logic_vector(0 to 5) := "000000";

-- Memory sub-interface specific opcodes
constant OPCODE_LOAD : std_logic_vector(0 to 5) := "000001";
constant OPCODE_STORE : std_logic_vector(0 to 5) := "000010";
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";

-- Function sub-interface specific opcodes
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');

signal current_state, next_state : state_machine := FUNCTION_RESET;
signal return_state, return_state_next : state_machine := FUNCTION_RESET;

signal argument, argument_next : std_logic_vector(0 to 31);
signal varOne, varOne_next : std_logic_vector(0 to 31);
signal varTwo, varTwo_next : std_logic_vector(0 to 31);
signal result, result_next : std_logic_vector(0 to 31);
signal toUser_address : std_logic_vector(0 to 31);
signal toUser_value : std_logic_vector(0 to 31);
signal toUser_function : std_logic_vector(0 to 15);
signal toUser_goWait : std_logic;

-- misc constants
---------------------------------------------------------------------------

begin -- architecture IMP

HWTUL_STATE_PROCESS : process (clock) is
begin

-- Find out if the HWTI is tell us what to do
if (intrfc2thrd_goWait = '1') then
  case intrfc2thrd_function is
    when U_FUNCTION_USER_SELECT =>
      current_state <= next_state;
    when U_FUNCTION_RESET =>
      current_state <= FUNCTION_RESET;
    when U_FUNCTION_START =>
      current_state <= FUNCTION_START;
    when U_FACTORIAL_1 =>
      current_state <= FACTORIAL_1;
    when U_READ_FACTORIAL_RETURN =>
      current_state <= READ_FACTORIAL_RETURN;
    when U_FACTORIAL_7 =>
      current_state <= FACTORIAL_7;
    -- List all the functions the HWTI could tell us to run
    when U_FUNCTION_USER_SELECT =>
      current_state <= next_state;
    when U_FUNCTION_RESET =>
      current_state <= FUNCTION_RESET;
    when U_FUNCTION_START =>
      current_state <= FUNCTION_START;
    when U_FACTORIAL_1 =>
      current_state <= FACTORIAL_1;
    when U_READ_FACTORIAL_RETURN =>
      current_state <= READ_FACTORIAL_RETURN;
    when U_FACTORIAL_7 =>
      current_state <= FACTORIAL_7;
  end case;
end if;

if (clock'event and (clock = '1')) then
  toUser_address <= intrfc2thrd_address;
  toUser_value <= intrfc2thrd_value;
  toUser_function <= intrfc2thrd_function;
  toUser_goWait <= intrfc2thrd_goWait;

  argument <= argument_next;
  varOne <= varOne_next;
  varTwo <= varTwo_next;
  result <= result_next;
  return_state <= return_state_next;
end if;

begin -- architecture IMP
current_state <= FACTORIAL_7;
-- If the HWTI tells us to do something we don't know, error
when OTHERS =>
current_state <= ERROR_STATE;
end case;
else
  current_state <= WAIT_STATE;
end if;
end if;
end process HWTUL_STATE_PROCESS;

HWTUL_STATE_MACHINE : process (clock) is
begin
  -- Default register assignments
  -- next_state <= current_state;
thrd2intrfc_opcode <= OPCODE_NOOP; -- When issuing an OPCODE, must be a pulse
thrd2intrfc_address <= Z32;
thrd2intrfc_value <= Z32;
thrd2intrfc_function <= U_FUNCTION_USER_SELECT;
next_state <= current_state;
return_state_next <= return_state;
argument_next <= argument;
varOne_next <= varOne;
varTwo_next <= varTwo;
result_next <= result;

  -- The state machine case current_state is
  case current_state is
    when FUNCTION_RESET =>
      -- Set default values
      result_next <= Z32;
      argument_next <= Z32;
      varOne_next <= Z32;
      varTwo_next <= Z32;
      thrd2intrfc_opcode <= OPCODE_NOOP;
      thrd2intrfc_address <= Z32;
      thrd2intrfc_value <= Z32;
      thrd2intrfc_function <= U_FUNCTION_START;
    when FUNCTION_START =>
      -- Ask the HWTI the value of the passed in argument
      thrd2intrfc_value <= Z32;
      thrd2intrfc_opcode <= OPCODE_POP;
      return_state_next <= READ_ARGUMENT;
      next_state <= WAIT_STATE;
    when READ_ARGUMENT => --0004
      -- read the value of the passed in argument
      argument_next <= toUser_value;
      next_state <= PUSH_ARGUMENT;
    when PUSH_ARGUMENT => --0005
      -- push the argument, for the factorial function, on the stack
      thrd2intrfc_value <= argument;
      thrd2intrfc_opcode <= OPCODE_PUSH;
      return_state_next <= CALL_FACTORIAL;
      next_state <= WAIT_STATE;
    when CALL_FACTORIAL => -- 0006
      -- make a call to the factorial function
      thrd2intrfc_function <= U_FACTORIAL_1;
      thrd2intrfc_value <= Z32(0 to 15) & U_READ_FACTORIAL_RETURN;
      thrd2intrfc_opcode <= OPCODE_CALL;
      next_state <= WAIT_STATE;
    when READ_FACTORIAL_RETURN => -- 0007
when PUSH_RETURN => -- 0008
-- Push a return value
thrd2intrfc_value <= result;
thrd2intrfc_opcode <= OPCODE_PUSH;
return_state_next <= CALL_EXIT;
next_state <= WAIT_STATE;
when CALL_EXIT => -- 000A
-- Immediately exit
thrd2intrfc_function <= FUNCTION_HTHREAD_EXIT;
thrd2intrfc_value <= Z32;
thrd2intrfc_opcode <= OPCODE_CALL;
next_state <= WAIT_STATE;
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
thrd2intrfc_value <= Z32;
thrd2intrfc_opcode <= OPCODE_POP;
return_state_next <= FACTORIAL_2;
next_state <= WAIT_STATE;
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
thrd2intrfc_value <= x"00000001";
thrd2intrfc_opcode <= OPCODE_DECLARE;
return_state_next <= FACTORIAL_4;
next_state <= WAIT_STATE;
when FACTORIAL_4 => -- 0104
-- store the register as a saved variable
thrd2intrfc_value <= varOne;
thrd2intrfc_address <= Z32;
thrd2intrfc_opcode <= OPCODE_WRITE;
return_state_next <= FACTORIAL_5;
next_state <= WAIT_STATE;
when FACTORIAL_5 => -- 0105
-- check if param <= 1
case varOne is
when x"00000001" =>
-- return a 1
thrd2intrfc_value <= x"00000001";
thrd2intrfc_opcode <= OPCODE_RETURN;
next_state <= WAIT_STATE;
when 32 =>
-- return a 1
thrd2intrfc_value <= x"00000000"
thrd2intrfc_opcode <= OPCODE_RETURN;
next_state <= WAIT_STATE;
when others =>
-- recursively call factorial, prepare by pushing param-1
thrd2intrfc_value <= (varOne - 1);
thrd2intrfc_opcode <= OPCODE_PUSH;
return_state_next <= FACTORIAL_6;
next_state <= WAIT_STATE;
end case;

when FACTORIAL_6 => -- 0106
  -- recursively call factorial
  thrd2intrfc_value <= Z32(0 to 15) & U_FACTORIAL_7;
  thrd2intrfc_function <= U_FACTORIAL_1;
  thrd2intrfc_opcode <= OPCODE_CALL;
  next_state <= WAIT_STATE;

when FACTORIAL_7 => -- 0107
  -- read the return value, save to a register
  -- TODO, change this to multiplication
  varTwo_next <= toUser_value;
  -- read the save variable
  thrd2intrfc_address <= Z32;
  thrd2intrfc_opcode <= OPCODE_READ;
  return_state_next <= FACTORIAL_8;
  next_state <= WAIT_STATE;

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

when FACTORIAL_9 => -- 0109
  -- return
  thrd2intrfc_value <= varTwo + varOne;
  thrd2intrfc_opcode <= OPCODE_RETURN;
  next_state <= FACTORIAL_END;

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

when WAIT_STATE =>
  -- wait state is used for one of two purposes.
  -- 1. the HWTI is telling us to wait, ie go_wait=0
  -- 2. the HWTUL in previous state made a call to the HWTI and
  -- needs to wait one clock cycle.
  next_state <= return_state;
end case;
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 below.

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 HWTI 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.

**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.

● **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).

### 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 HWTI'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>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</td>
</tr>
<tr>
<td>opcode</td>
<td>Clock Cycles</td>
<td>Comment</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------</td>
<td>-------------------------------------------------------------------------</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>
<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 mutex variable, previously initialized.

```c
void * basicThr
     read( 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>
</tbody>
</table>
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;

-- 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

        if (clock'event and (clock = '1')) then
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>
</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>
</tbody>
</table>
| HIF           | addressof R2 fooMutex
    syscall mutex lock R2 |
| VHDL          | -- Lock mutex zero
    when MUXTEX LOCK_1 =>
        -- Tell the HWTI, which mutex to lock
        argOne_next <= Z32;
        opcode_next <= OPCODE_HTHREAD_MUTEX_LOCK;
        next_state <= MUXTEX_LOCK_2; |
    when MUXTEX LOCK_2 =>
        -- Wait for the HWTI to ack
        if ( intrfc2thrd_status = USER_STATUS_ACK ) then
            opcode_next <= OPCODE_NOOP;
            next_state <= MUXTEX_LOCK_3;
        else
            next_state <= MUXTEX_LOCK_2;
        end if;
    when MUXTEX 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 <= MUXTEX_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.

<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>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>next_state &lt;= READ_MEMORY_3;</td>
</tr>
<tr>
<td></td>
<td>end if;</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.

<table>
<thead>
<tr>
<th>pthread</th>
<th>fooValue += pthread_self();</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread</td>
<td>fooValue += hthread_self();</td>
</tr>
</tbody>
</table>
| HIF     | syscall self
|         | add R3 R3 returnVal
### Yielding the Processor

Yielding the processor is next for the thread. For a hardware thread, yielding the processor is currently irrelevant. A good HLL to HDL compiler will remove hthread_yield references. The practical-ness 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>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread</td>
<td><code>pthread_yield();</code></td>
</tr>
<tr>
<td>hthread</td>
<td><code>hthread_yield();</code></td>
</tr>
<tr>
<td>HIF</td>
<td><code>syscall yield</code></td>
</tr>
</tbody>
</table>
| 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.

<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 MUX_UNLOCK_1 =>
    -- Tell the HWTI what mutex to lock
    argOne_next <= Z32;
    opcode_next <= OPCODE_HTHREAD_MUX_UNLOCK;
    next_state <= MUX_UNLOCK_2;

when MUX_UNLOCK_2 =>
    -- Wait for the HWTI to acknowledge
    if (intrfc2thrd_status = USER_STATUS_ACK) then
        opcode_next <= OPCODE_NOOP;
        next_state <= MUX_UNLOCK_3;
    else
        next_state <= MUX_UNLOCK_2;
    end if;

when MUX_UNLOCK_3 =>
    -- Wait for the HWTI to tell us to start running again
    -- When we start running again, we know we gave up the lock
    if (intrfc2thrd_status = USER_STATUS_RUN) then
        next_state <= EXIT_INIT;
    else
        next_state <= MUX_UNLOCK_3;
    end if;

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.

pthread
return fooAddr;

hthread
return fooAddr;

HIF
return R1

VHDL

when EXIT_INIT =>
    argOne_next <= fooAddr;
    opcode_next <= OPCODE_HTHREAD_EXIT;
    next_state <= EXIT_WAIT_ACK;

when EXIT_WAIT_ACK =>
    case intrfc2thrd_status is
    when USER_STATUS_ACK =>
        opcode_next <= OPCODE_NOOP;
        next_state <= EXIT_WAIT;
    when others =>
        next_state <= EXIT_WAIT_ACK;
    end case;

when EXIT_WAIT =>
Closeout

The final lines of code are meant to close out the function, providing whatever syntax is needed.

<table>
<thead>
<tr>
<th>pthread</th>
<th>}</th>
</tr>
</thead>
<tbody>
<tr>
<td>hthread</td>
<td>}</td>
</tr>
<tr>
<td>HIF</td>
<td>when others =&gt;</td>
</tr>
<tr>
<td>VHDL</td>
<td>next_state &lt;= IDLE;</td>
</tr>
<tr>
<td></td>
<td>end case;</td>
</tr>
<tr>
<td></td>
<td>end process HWTUL_STATE_MACHINE;</td>
</tr>
<tr>
<td></td>
<td>end architecture IMP;</td>
</tr>
</tbody>
</table>