HT Quick Start Guide

February 16, 2015

Version 1.0

901-000015-000

Trademarks

The following are trademarks of Convey Computer Corporation:

The Convey Computer Logo: 

Convey Computer

HC-1

HC-1^{ex}

HC-2

HC-2^{ex}

Trademarks of other companies

Intel is a registered trademark of Intel Corporation

Adobe and Adobe Reader are registered trademarks of Adobe Systems Incorporated

Linux is a registered trademark of Linus Torvalds

Revisions

Version	Description
1.0	February 16, 2015. Initial Release

Table of Contents

1 Overview 4
1.1 Introduction 4
1.2 Document Content 4
1.2.1 Intended Audience 4
1.2.2 Required Software 4
2 Example Project 5
2.1 Overview 5
2.2 Source Files 5
2.2.1 Host Application Source 5
2.2.2 Personality Source 6
2.2.3 Functional Model Source 7
3 Build Steps 9
3.1 Functional Model 9
3.2 SystemC Simulation 9
3.3 Verilog Simulation 9
3.4 Personality FPGA 10
3.5 Run on Convey System 10
3.6 Generated Files 10
3.7 Reports 10
3.7.1 HT Design Report 11
3.7.2 HT Performance Monitor Report 11
4 Other Examples 12
4.1 Streaming Interface 12
4.1.1 Overview 12
4.1.2 Source Files 12
4.2 Incorporate Existing Verilog Module 13
4.2.1 Overview 13
4.2.2 Source Files 13
5 Customer Support Procedures 17

1 Overview

1.1 Introduction

The Convey Hybrid Threading (HT) development environment enables developers of Convey personalities to compile applications written in C/C++ to target both the host processor (x86) and the Convey coprocessor (FPGA). By allowing the developer to focus on the core functionality of the application instead of low-level hardware details, it provides a significant increase in developer productivity.

1.2 Document Content

This document provides example based instructions for using the HT Tools to develop a coprocessor personality for a Convey coprocessor.

A simple vector addition example is used to show basic HT functionality and design flow. Alternate implementations of the vector addition example are also provided illustrating additional features of the tool set.

1.2.1 Intended Audience

This document is intended for users interested in developing custom personalities for the Convey family of products. While it does raise the level of abstraction involved in programming the personality, it is recommended that the user have some experience with FPGA development. The user should also be capable of writing and debugging applications in C/C++.

1.2.2 Required Software

The HT environment requires the following components:

- Convey Personality Development Kit (PDK)
- SystemC libraries
- C/C++ compiler
- Xilinx ISE Design Suite14
- An HDL simulator for Verilog/VHDL simulation.
    - Mentor ModelSim
    - VCS

2 Example Project

2.1 Overview

The Vector Add design is a simple example that illustrates some of the basic capabilities of the HT toolset. This example is found in the **ex_vadd/** project directory. The design implements the following function using HT instructions running on the Convey coprocessor:

```
uint64_t vadd(uint64_t *a1, uint64_t *a2, uint64_t *a3, uint64_t
vecLen)
    {
    uint64_t sum = 0;
    for (i=0; i<vecLen; i++) {
          a3[i] = a1[i] + a2[i];
          sum += a3[i];
    }
    return sum;
    }
```

While this is a simple example, the basic concepts used in an HT design and the design flow are explained.

2.2 Source Files

The following are the source files for the ex_vadd HT project:

```
ex_vadd/
    Makefile
    src/                          (host application source)
          Main.cpp
    src_model/                    (Personality functional model source)
          Model.cpp
          model.htd
    src_pers/                     (Personality source)
          PersAdd_src.cpp
          PersCtl_src.cpp
          vadd.htd
```

2.2.1 Host Application Source

The host source contains the code that runs on the host processor. It is located in the **ex_vadd/src/** directory. In the example the host source (**Main.cpp**) does the following

- Allocates arrays in host memory
- Fills the host memory arrays

- Constructs the HtHif class (Ht Host Interface)
- Constructs a CHtAuUnit class for each unit in the design
- Allocates arrays in coprocessor memory
- Copies the arrays from host to coprocessor memory
- Sends message to all units, containing pointers to the arrays and the vector length
- Calls the coprocessor to perform the operation
- Waits for the coprocessor to complete
- Checks results

2.2.2 Personality Source

The personality source contains the code that runs on the coprocessor. It is located in the **`ex_vadd/src_pers/`** directory. The personality source consists of the HT description file and source for the custom instructions contained in each module of the design.

2.2.2.1 HT Description

The HT description file (**`vadd.htd`**) for the example contains the following

- Module/function definitions
  - Module entry point
  - Module return
  - Instructions included in the module
- Definition of variables
- Define interfaces needed
  - Memory Interface
  - Host Message
  - Call Interface

2.2.2.2 Custom Instructions

There is a custom instruction file for each module of the design. The vector add example has two modules, Ctl and Add. The Ctl module is the top level entry point into the personality. The Ctl module starts a thread in the Add module for each array element to be processed. The Add module adds an element from each operand array and returns the result. As each spawned thread returns from the Add module, the Ctl module accumulates the returned sum. Once all elements have been processed the Ctl module waits for all threads to return, then returns the sum.

Note: Pointers to the vectors are sent to the personality using host messages. Data from host messages is stored in shared variables.

2.2.2.2.1 Ctl Module

The Ctl module has four instructions.

- *CTL_ENTRY*
- *CTL_ADD*

- *CTL_JOIN*
- *CTL_RTN*

The functionality of these instructions is defined in the **`PersCtl_src.cpp`** file

The *CTL_ENTRY* instruction is the entry point for the personality. It initializes variables then sets the next instruction to execute to *CTL_ADD*. The *CTL_ADD* instruction begins a thread in the Add module for each element in the array. When each thread returns from the Add module the *CTL_JOIN* instruction is executed, updating the result. After a thread is started for each element the thread in the Ctl module is paused until all threads return and the *CTL_RTN* instruction is performed, terminating the thread in the Ctl module and returning to the host application.

2.2.2.2.2 Add Module

The Add module has four instructions

- *ADD_LD1*
- *ADD_LD2*
- *ADD_ST*
- *ADD_RTN*

The functionality of these instructions is defined in the **`PersAdd_src.cpp`** file.

The *ADD_LD1* instruction is the entry point. This instruction loads an element from operand array 1, and then sets the *ADD_LD2* instruction as the next instruction to execute. The *ADD_LD2* instruction loads an element from operand array 2 and waits for the reads of elements 1 and 2 to complete before setting the *ADD_ST* instruction as the next instruction to execute. The *ADD_ST* instruction adds the elements, stores them and waits for the store to complete before setting the next instruction to *ADD_RTN*. The *ADD_RTN* instruction terminates the thread in the Add module and returns to the Ctl module.

2.2.3 Functional Model Source

The model source emulates the behavior of the coprocessor personality, implementing the interface between the host and the coprocessor and performing the functionality of the coprocessor. It is located in the **`ex_vadd/src_model/`** directory. The model source consists of the HT description file and source for the model of the personality.

2.2.3.1 HT Description

The HT description file (**`model.htd`**) for the example contains the following

- Module definitions
  - Single module is required
  - Module entry point
  - Module return
  - Single instruction is required
- Define host interfaces needed
- Definition of variables
  - Shared variables for host messages

- Private variables for call parameters

2.2.3.2 Model Source

In the example the model source (**`Model.cpp`**) does the following

- Receives host messages
- Receives call
- Performs vector add
- Returns to host application

3 Build Steps

The build environment provided by Convey uses the Makefile at the top level of the project. The HT Makefile supports targets for all steps of the development flow as shown in Table 1.

Target	*Description*
model	Builds functional model application (*app_model*)
sysc	Builds SystemC simulation application (*app_sysc*)
vsim	Builds Verilog simulation application (*app_vsim*)
app	Builds coprocessor application (*app*)
pers	Implements coprocessor FPGA bitfile

Table 1 – Make Targets

3.1 Functional Model

The functional model allows the programmer to develop the host application, including the runtime interfaces to the coprocessor, before developing the HT code for the coprocessor. The functional model is a purely behavioral model of the personality, implemented in C++. The application compiled using the functional model using

```
make model
```

This compiles the host source (in **src/**) and the model source (in **src_model/**) into an executable called *app_model* located in the top level of the project directory.

3.2 SystemC Simulation

After developing the HT code, cycle accurate simulation of the design can be performed using SystemC simulation. The HT application is compiled using

```
make sysc
```

This compiles the host source (in **src/**) and the personality source (in **src_pers/**) into an executable called *app_sysc*. Most of the development and debug time is spent in running SystemC simulation, which is significantly faster than Verilog simulation, as well as easier to debug.

3.3 Verilog Simulation

Once the design is verified using SystemC simulation, Verilog can be generated and Verilog simulation performed. Verilog files are generated using

```
make vsim
```

This generates Verilog files from the personality source (in src_pers/) and builds an application called *app_vsim*, which runs a Verilog simulation of the generated code. The application runs a Verilog simulation of the design using either Mentor ModelSim or

Synopsys VCS. The Verilog simulator is selected by the CNY_PDK_HDLSIM environment variable (options are Mentor or Synopsys).

3.4 Personality FPGA

The personality FPGA is built from the Verilog files using the Xilinx development tools using

```
make pers
```

This step can take several hours and is typically done after the design has been fully debugged in SystemC simulation.

3.5 Run on Convey System

The application is compiled for the Convey system using:

```
make app
```

This builds and application called *app* which is run on the Convey system after the personality has been generated.

3.6 Generated Files

This section outlines the files generated by the HT tools.

```
ex_vadd/
      ht/
            model/        (model simulation directory)
            phys/         (physical build directory)
            sim/          (Verilog simulation directory)
            sysc/         (SystemC simulation directory)
            verilog/      (Generated Verilog files)
            vsim/         (Verilog simulation directory)
      HtDsnRpt.html       (Design report)
      App                 (application for Convey system)
      app_model           (application using functional model)
      app_sysc            (application for SystemC simulation)
      app_vsim            (Script to run Verilog simulation)
      app_vsim.exe        (application for Verilog simulation)
      personalities/      (personality directory for FPGA)
```

3.7 Reports

The HT tools generate two reports

- Design report (**`HtDsnRpt.html`**)
- Performance monitor report (**`HtMonRpt.txt`**)

3.7.1 HT Design Report

The design report is generated when the design is compiled. It contains information about the design including

- call graph
- generated APIs
- generated RAMs

3.7.2 HT Performance Monitor Report

The performance monitor report is generated when the SystemC simulation is run (`./app_sysc`). The report contains information including

- cycle count
- memory request counts
- thread activity

4 Other Examples

This section contains alternative implementations of the vector add coprocessor functionality. These implementations demonstrate other functionality available using the HT Toolset.

4.1 Streaming Interface

4.1.1 Overview

The Vector Add design with streaming interfaces is a simple example that illustrates the streaming capability of the HT toolset. This example is found in the **`ex_vadd_stream/`** project directory.

4.1.2 Source Files

The following are the source files for the ex_vadd HT project:

```
ex_vadd/
    Makefile
    src/                        (host application source)
          Main.cpp
    src_model/                  (Personality functional model source)
          Model.cpp
          model.htd
    src_pers/                   (Personality source)
          PersAdd_src.cpp
          vadd.htd
```

4.1.2.1 Personality Source

The personality source contains the code that runs on the coprocessor. It is located in the **`ex_vadd_stream/src_pers/`** directory. The personality source consists of the HT description file and source for the custom instructions contained in each module of the design.

4.1.2.1.1 HT Description

The HT description file (**`vadd.htd`**) for the example contains the following

- Module/function definitions
    - Module entry point
    - Module return
    - Instructions included in the module
- Definition of variables
- Define interfaces needed
    - Streaming Interfaces

- Host Message
- Call Interface

4.1.2.1.2 Custom Instructions

The streaming vector add example has one module, Vadd. The Vadd module is the top level entry point into the personality. In addition to the instructions, this design has non-instruction functionality that is executed every clock.

4.1.2.1.2.1 Vadd Module

The Vadd module has two instructions.

- *VADD_ENTER*
- *VADD_RETURN*

The functionality of these instructions along with non-instruction functionality is defined in the **`PersVadd_src.cpp`** file.

The *VADD_ENTER* instruction is the entry point for the personality. This instruction checks the availability of read streams A and B, which contain the operands and write stream C, where the sum of the operands will be written. It initializes variables then determines the address of the starting address of the first element of each stream. The streams are opened and the thread is paused until writes to the write stream are complete. When the thread resumes the next instruction executed is *VADD_RETURN.*

The *VADD_RETURN* instruction terminates the thread in the Vadd module and returns to the host application.

The streams are read and written in non-instruction functionality. If all three streams are ready the operands are read from streams A and B, the operands are summed. The result is written to stream C and

The Vadd function also contains non-instruction functionality that is performed every clock cycle. The non-instruction functionality checks if the both read streams have and element ready and if the write stream can accept and element. If so the non-instruction functionality does the following:

- reads an element from read streams A and B
- adds the elements to get the element sum
- adds the element sum to the vector sum
- writes the element sum to write stream C

4.2 Incorporate Existing Verilog Module

4.2.1 Overview

This example incorporates the add function, implemented as a Verilog primitive into and HT design. This example is found in the **`ex_vadd_prim/`** project directory

4.2.2 Source Files

The following are the source files for the vadd HT example using a Verilog module:

```
ex_vadd_prim/
    Makefile
    src/                          (host application source)
          Main.cpp
    src_pers/                     (Personality source)
          PersAdd_prim.cpp
          PersAdd_prim.h
          PersAdd_src.cpp
          PersCtl_src.cpp
          prims.v
          vadd.htd
```

4.2.2.1 Personality Source

The personality source contains the code that runs on the coprocessor. It is located in the **ex_vadd_prim/src_pers/** directory. The personality source consists of the

- HT description file (vadd.htd)
- Source for the custom instructions (PersCtl_src.cpp and PersAdd_src.cpp)
- Verilog module (prims.v)
- contained in each module of the design.

4.2.2.1.1 HT Description

The HT description file (**vadd.htd**) for the example contains the following

- Module/function definitions
    - Module entry point
    - Module return
    - Instructions included in the module
- Definition of variables
- Define interfaces needed
    - Host Message
    - Call Interface
- Define Verilog primitive

4.2.2.1.2 Custom Instructions

There is a custom instruction file for each module of the design. The vector add example, using a Verilog module has two modules, Ctl and Add. The Ctl module is the top level entry point into the personality. The Ctl module starts a thread in the Add module for each array element to be processed. The Add module adds an element from each operand array and returns the result. As each spawned thread returns from the Add module, the Ctl module accumulates the returned sum. Once all elements have been processed the Ctl module waits for all threads to return, then returns the sum.

Note: Pointers to the vectors are sent to the personality using host messages. Data from host messages is stored in shared variables.

4.2.2.1.2.1 Ctl Module (PersCtl_src.cpp)

The Ctl module has four instructions.

- *CTL_ENTRY*
- *CTL_ADD*
- *CTL_JOIN*
- *CTL_RTN*

The functionality of these instructions is defined in the **`PersCtl_src.cpp`** file

The *CTL_ENTRY* instruction is the entry point for the personality. It initializes variables then sets the next instruction to execute to *CTL_ADD*. The *CTL_ADD* instruction begins a thread in the Add module for each element in the array. When each thread returns from the Add module the *CTL_JOIN* instruction is executed, updating the result. After a thread is started for each element the thread in the Ctl module is paused until all threads return and the *CTL_RTN* instruction is performed, terminating the thread in the Ctl module and returning to the host application.

4.2.2.1.2.2 Add Module (PersAdd_src.cpp)

The Add module has four instructions

- *ADD_LD1*
- *ADD_LD2*
- *ADD_PAUSE*
- *ADD_ST*
- *ADD_RTN*

The functionality of these instructions is defined in the **`PersAdd_src.cpp`** file.

The *ADD_LD1* instruction is the entry point. This instruction loads an element from operand array 1, and then sets the *ADD_LD2* instruction as the next instruction to execute. The *ADD_LD2* instruction loads an element from operand array 2 and waits for the reads of elements 1 and 2 to complete before setting the *ADD_PAUSE* as the next instruction to execute. The *ADD_PAUSE* instruction stores the operands in private variables and sets and input valid indicator. The thread is then paused until the Verilog module completes at which time the thread is resumed and the next instruction is set to *ADD_ST*. The *ADD_ST* instruction stores the result and waits for the store to complete before setting the next instruction to *ADD_RTN*. The *ADD_RTN* instruction terminates the thread in the Add module and returns to the Ctl module.

The Add module contains non-instruction functionality that is performed every clock cycle. The non-instruction functionality declares temporary variables (not saved between cycles) for outputs of the Verilog module. The add_5stage Verilog module is called as a function with inputs and outputs of the Verilog module as arguments. If the output valid output from the Verilog module is active the results are stored and the thread is resumed and will execute the *ADD_RTN* instruction.

4.2.2.1.2.3 Verilog Primitive Declaration (PersAdd_prim.h)

The **PersAdd_prim.h** file contains the declarations of the primitive state structure and the primitive function.

The primitive state structure contains internal state in the Verilog module that is needed to simulate the HT design.

The primitive function declaration associates the Verilog module to the function and maps the clock and I/O ports of the Verilog module to the arguments of the function.

4.2.2.1.2.4 Verilog Primitive Functionality (PersAdd_prim.cpp)

The **PersAdd_prim.cpp** file contains a description of the functionality of the Verilog module. This description is used in systemC simulation.

5 Customer Support Procedures

E-mail	support@conveycomputer.com
Web	Go to Customer Support at www.conveycomputer.com