XAPP1209 (v1.0.1) August 8, 2014

Designing Protocol Processing Systems with Vivado High-Level Synthesis

Author: Kimon Karras, James Hrica

Summary

This application note describes how the Vivado® High-Level Synthesis (HLS) tool enables higher productivity in protocol processing designs by providing abstractions in critical areas. This simplifies designs and makes them less error-prone. While the basics of implementing protocol processing designs using Vivado HLS are fairly straightforward, there are some subtle aspects that warrant a detailed explanation. This application note includes:

- Basic Concepts and Code Examples for building a packet processing system in Vivado HLS.
- Advanced Concepts and Code Examples.
- Included with this application note is an example design that demonstrates how a protocol processing sub system can be built using Vivado HLS. This document includes information about how to use the example design, along with Example Design File Location and Details and an Example Design Description. The example design implements a basic set of networking modules that implement Address Resolution Protocol (ARP) and ping functionality. The required Vivado HLS files are provided, as well as a Vivado Design Suite project that you can use to implement the design on a Xilinx® VC709 development board.

Introduction

Protocol Processing

Protocol processing on different levels is present in any modern communication system because any exchange of information requires the use of a communication protocol. The protocol typically contains packets. The packets must be created by the sender and reassembled at the receiver, while ensuring adherence to protocol specifications. This makes protocol processing ubiquitous. Consequently, protocol processing, and implementing protocol processing functionality efficiently, is important for FPGA design.

This application note explains how to address key challenges encountered when processing protocols using Vivado HLS.

Raising the Level of Abstraction and Related Benefits

Vivado HLS raises the level of abstraction in system design by:

- Using C/C++ as a programming language and leveraging the high-level constructs it offers.
- Providing additional data primitives that allow you to easily use basic hardware building blocks (bit vectors, queues, etc.)

These characteristics allow you to use Vivado HLS to address common protocol system design challenges more easily than when using RTL, with the following benefits:

System assembly

Vivado HLS modules are treated as functions, with the function definition being equivalent to an RTL description of the module and a function call being equivalent to a module instantiation. This simplifies the structural code describing the system by reducing the amount of code that has to be written.

Simple FIFO/memory access

Accessing a memory or a FIFO in Vivado HLS is done in one of two ways: through methods of an appropriate object (for example, the read and write methods of a stream object) or by accessing a standard C array, which synthesis then implements as block RAM or distributed RAM. Vivado HLS takes care of the additional signaling, synchronization, and/or addressing as required.

Abstraction of control flow

Vivado HLS provides a set of flow-control aware interfaces ranging from simple FIFO interfaces to AXI4-Stream. In all of these interfaces, you access the data without having to check for back pressure or data availability. Vivado HLS schedules execution appropriately to take care of all contingencies, while ensuring correct execution.

Word realignment

The abstraction of flow control enables you to use Vivado HLS to perform core protocol processing tasks, such as word realignment easily. Data access aided by the abstraction of flow control eliminates the need for error prone reading and writing from FIFO/memories, and thus allows you to write simpler code.

Easy architecture exploration

In Vivado HLS, you can insert pragma directives in the code to communicate the features of your design to Vivado HLS. These can range from fundamental issues, such as the pipelining of a module, to more mundane ones, such as the depth of a FIFO queue. In any case, pragma directives provide you with the ability to explore a wide range of architectural alternatives without requiring changes to the implementation code itself.

C and C/RTL simulation

Vivado HLS designs can be verified using a two-step simulation process.

1. C simulation, in which the C/C++ is compiled and executed like a normal C/C++ program. While this simulation is not cycle-accurate, it mirrors the functionality of the auto-generated RTL code very well. This enables functional verification of the design by using C/C++ test benches at C/C++ execution speeds, thus enabling very long simulations, which are not possible in RTL.
2. Verification with C/RTL co-simulation. Vivado HLS automatically generates an RTL test bench from the C/C++ test bench, then implements and executes an RTL simulation that can be used to check the accuracy of the implementation.

Understanding Directives

Because the C++ code used in Vivado HLS is compact in nature, you can leverage its features to realize development time and productivity benefits as well as improvements in code maintainability and readability. Furthermore, Vivado HLS allows you to maintain control over the architecture and its features. To take full advantage of its capabilities, correct understanding and use of Vivado HLS directives is fundamental.

SDNet

HLS occupies an intermediate slot in the hierarchy of Xilinx-provided packet processing solutions. It is complemented by Vivado Design Suite SDNet [Ref 1], which uses:

- A domain-specific language to offer a simpler, if more constrained, way of expressing protocol processing systems
- RTL, which allows for the implementation of a considerably wider breadth of systems that Vivado HLS is not able to express (for example, systems requiring detailed clock management using DCMs or differential signaling).

You can, however, use HLS to implement the vast majority of protocol processing solutions efficiently, without compromising the quality of results or design flexibility.

Basic Concepts and Code Examples

This section provides guidelines and code examples for building a simple protocol processing system with Vivado HLS.

When starting a new design, the most basic tasks to be accomplished are:

- Determining the design structure. An example is provided in the section Setting Up a Simple System.
- Implementing the design in Vivado HLS. An example is provided in the section Implementing a State Machine with Vivado HLS.

Setting Up a Simple System

In Vivado HLS, the basic building block of a system is a C/C++ function. Building a system consisting of modules and submodules essentially means that a top-level function calls lower level functions. Figure 1 illustrates a simple three-stage pipeline example to introduce the basic concepts for system building in Vivado HLS. Protocol processing is typically performed in pipelined designs, with each stage addressing a specific part of the processing.

Code Example 1 - Creating a Simple System in Vivado HLS

```
void topLevelModule(stream<axiWord> &inData, stream<axiWord> &outData) {
    #pragma VHLS dataflow interval=1

    #pragma HLS INTERFACE port=inData axis
    #pragma HLS INTERFACE port=outData axis

    static stream<ap_uint<64> > modOne2modTwo
    static stream<ap_uint<64> > modTwo2modThree;

    #pragma HLS STREAM variable = modOne2modTwo depth = 4;
    #pragma HLS STREAM variable = modTwo2modThree depth = 4;

    moduleOne(inData, modOne2modTwo);
    moduleTwo(modOne2modTwo, modTwo2modThree);
    moduleThree(modTwo2modThree, outData);
}
```

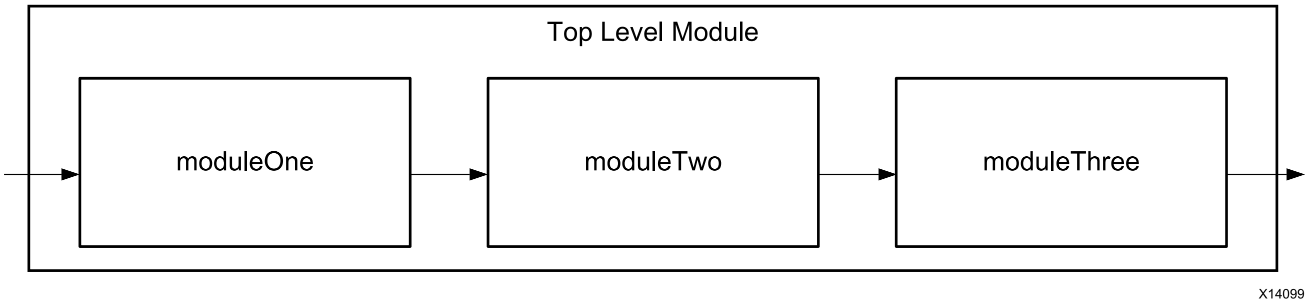


Figure 1: **Simple Three-Stage Pipeline**

Code Example 1 - Creating a Simple System in Vivado HLS creates the top module function that calls the other sub-functions. The top module function uses two parameters, both of which are objects of class `stream`, which is one of the template classes provided by the Vivado HLS libraries. A stream is a Vivado HLS modeling construct that represents an interface over which data is to be exchanged in a streaming manner. A stream can be implemented as a FIFO queue or shift register, as detailed in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2]. A stream is a template class that can be used with any C++ construct. In this case, a defined data structure (`struct`) called `axiWord` is used. This is shown in Code Example 2 - Definition of a C++ struct for Use in a Stream Interface.

Code Example 2 - Definition of a C++ struct for Use in a Stream Interface

```
struct axiWord {
   ap_uint<64>      data;
   ap_uint<8>       strb;
   ap_uint<1>       last;
};
```

This `struct` defines part of the fields for an AXI4-Stream interface. This kind of interface is automatically supported in Vivado HLS and can be specified using a pragma statement. Pragmas are directives to the Vivado synthesis tool that help guide the tool to reach the required results. The pragmas in lines 4 and 5 of Code Example 1 - Creating a Simple System in Vivado HLS tell Vivado HLS that both parameters (essentially the input and output ports of the top module) are to use AXI4-Stream interfaces and provide a name for the resulting interface. The AXI4-Stream interface includes two mandatory signals, the valid and ready signals, which were not included in the declared `struct`. This is because the Vivado HLS AXI4 interface manages these signals internally, which means that they are transparent to user logic. As mentioned earlier, Vivado HLS completely abstracts flow control when using AXI4-Stream interfaces.

An interface does not have to use AXI4-Stream. Vivado HLS provides a rich set of bus interfaces, which are listed in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2]. AXI4-Stream is used here as an example of a popular, standardized interface that can be used for packet processing.

The next task in implementing your design is to ensure that our three modules are connected to each other. This is also accomplished through streams, but in this case, the streams are internal to the top module. Lines 7 and 8 of Code Example 1 - Creating a Simple System in Vivado HLS declare two streams for this purpose.

These streams:

- Make use of another Vivado HLS construct, `ap_uint`.
    - The `ap_unit` construct consists of bit-accurate unsigned integers and can be thought of and manipulated as a bit.

  - Because it is a template class, the width of this array must also be specified. In this case, 64 bits are used, matching the width of the data members of the input and output interfaces of the top module.
- Are declared as static variables. A static variable maintains its value over multiple function calls. The top-level module (and thus all its submodules) is called once in every clock cycle when executed as a sequential C/C++ program, so any variables that must maintain their values from the one cycle to the next intact, must be declared as static.

As mentioned, a stream interface can be implemented as a FIFO queue or as a memory, which means that it can also have a specific depth to act as a buffer for data traffic. The depth of the FIFO queue can be set for each stream using the stream pragma, as shown in lines 10 and 11 of Code Example 1 - Creating a Simple System in Vivado HLS. For typical feed-forward designs, buffering might not be required. Omitting the pragma causes Vivado HLS to automatically use a FIFO with a depth of one, which allows Vivado HLS to efficiently implement small FIFOs using flip-flops, and thus save block RAMs and LUTs.

Creating Pipelined Designs

The last pragma to discuss is perhaps the most important one. The `dataflow` pragma in line 2 of Code Example 1 - Creating a Simple System in Vivado HLS instructs Vivado HLS to attempt to schedule the execution of all the sub-functions in this function in parallel. It is important to note that the effect of the `dataflow` pragma does not propagate down the hierarchy of the design. Thus, if a lower level function contains sub-functions whose execution has to be scheduled in parallel, then the `dataflow` pragma must be specified in that function separately. The parameter `interval` defines the Initiation Interval (`II`) for this module. `II` defines the throughput of the design by telling Vivado HLS how often this module has to be able to process a new input data word. This does not preclude the module being internally pipelined and having a latency greater than 1. An `II = 2` means that the module has 2 cycles to complete the processing of a data word before having to read in a new one. This can allow Vivado HLS to simplify the resulting RTL for a module. That being said, in a typical protocol processing application the design has to be able to process one data word in each clock cycle, thus from now on an `II = 1` is used.

Note: The parameter `interval` is a deprecated feature and is subject to change.

Finally, you call the functions themselves. In Vivado HLS this also corresponds with the instantiations of the modules. The parameters that are passed to each module essentially define the module communication port. In this case, you create a chain of the three modules by connecting the input to the first module, then the first module to the second over stream `modOne2modTwo`, and so on.

Implementing a State Machine with Vivado HLS

Protocol processing is inherently stateful. You are required to read in successive packet words arriving onto a bus over many clock cycles and decide on further operations according to some field of the packet. The common way to handle this type of processing is by using a state machine, which iterates over the packet and performs the necessary processing. Code Example 3 - Finite State Machine using Vivado HLS shows a simple state machine, which either drops or forwards a packet, depending on an input from a previous stage.

The function receives three arguments: the input packet data over the `inData` stream, a one-bit flag that shows if a packet is valid or not over the `validBuffer` stream, and the output packet data stream, called `outData`.

Note: Parameters in the Vivado HLS functions are passed by reference. This is necessary when using Vivado HLS streams, which are complex classes. Simpler data types like the `ap_uint` can also be passed by value.

The pipeline pragma in line 2 of Code Example 3 - Finite State Machine using Vivado HLS instructs Vivado HLS to pipeline this function to achieve an initiation interval of 1 (`II = 1`), meaning that it is able to process one new input data word every clock cycle. Vivado HLS examines the design and determines how many pipeline stages it needs to introduce to the design to meet the required scheduling restrictions. To describe this pragma a bit further, assume you are performing a read-modify-write operation. If it is not pipelined, an `II=1` cannot be met because scheduling dictates that the read occur in clock cycle T and the write in clock cycle T+1. This is the default behavior in Vivado HLS without the pipeline pragma. Inserting the pragma causes Vivado HLS to schedule the access in a way that the target `II` value can be reached.

Caution: If accesses are interdependent, reaching the target `II` value might be impossible. Additional explanation on this topic can be found in Advanced Concepts and Code Examples.

Code Example 3 - Finite State Machine using Vivado HLS

```
1    void dropper(stream<axiWord>& inData, stream<ap_uint<1> >&
     validBuffer, stream<axiWord>& outData) {
2    #pragma HLS pipeline II=1 enable_flush
3
4    static enum dState {D_IDLE = 0, D_STREAM, D_DROP} dropState;
5    axiWord currWord = {0, 0, 0, 0};
6
7    switch(dropState) {
8    case D_IDLE:
9           if (!validBuffer.empty() && !inData.empty()) {
10              ap_uint<1> valid = validBuffer.read();
11              inData.read(currWord);
12              if (valid) {
13                 outData.write(currWord);
14                 dropState = D_STREAM;
15               }
16}
17          else
18                 dropState = D_DROP;
19          break;
20   case D_STREAM:
21          if (!inData.empty()) {
22             inData.read(currWord);
23             outData.write(currWord);
24             if (currWord.last)
25                dropState = D_IDLE;
26           }
27           break;
28   case D_DROP:
29         if (!inData.empty()) {
30            inData.read(currWord);
31            if (currWord.last)
32               dropState = D_IDLE;
33         break;
34         }
35   }
36 }
```

Line 4 declares a static enumeration variable that expresses state in this FSM. Using an enumeration is optional but allows for more legible code because states can be given proper names. However, any integer or `ap_uint` variable can also be used with similar results. Line 5 declares a variable of type `axiWord`, in which packet data to be read from the input is stored.

The switch statement in line 7 represents the actual state machine. Using a switch is recommended but not mandatory. An if-else decision tree would also perform the same functionality. The switch statement allows the tool to enumerate all the states and optimize the resulting state machine RTL code efficiently.

Execution starts at the `D_IDLE` state where the FSM reads from the two input streams in lines 10 and 11. These two lines demonstrate both uses of the read method of the stream object. Both methods read from the specified stream and store the result into the given variable. This method performs a blocking read, which means that if the method call is not successfully executed, the execution of the remaining code in this function call is blocked. This happens when trying to read from an empty stream.

You can use the method described above to describe an explicit state machine in HLS. In many cases (such as when partially unrolling a loop) HLS also creates a state machine to orchestrate the control flow required.

Stream Splitting and Merging

In the following code example, two pragmas are used at the top of the function to indicate how Vivado HLS must handle this function. The `inline` pragma instructs Vivado HLS not to dissolve and absorb this function into its top level. Using the `dataflow` pragma in a function causes it to respect the boundaries of any functions called from it. In this case, therefore, the `inline` pragma is not required.

However, this is only valid for the immediate lower level from the one in which `dataflow` was used. If a sub-function contains more nested sub-functions itself, these have to be inlined (or not) manually. If no inline directive is used, VHLS determines whether or not to inline, based on the size and complexity of each function. For example, if you have a three layer design and specify `dataflow` on layer 0 (the lowest one), the boundaries of the functions in layer 1 are preserved automatically because of the `dataflow` pragma. This does not, however, apply to the boundaries of the functions in layer 2.

The ability to forward packets to different modules according to some field in the protocol stack, and then to recombine these streams before transmission, is a critical functionality in protocol processing. Vivado HLS allows for the use of high-level constructs to facilitate this, as Code Example 4 - Simple Stream Merge illustrates for the case of a stream merging.

Code Example 4 - Simple Stream Merge

```
void merge(stream<axiWord> inData[NUM_MERGE_STREAMS], stream<axiWord> &outData) {
#pragma HLS INLINE off
#pragma HLS pipeline II=1 enable_flush

static enum mState{M_IDLE = 0, M_STREAM}   mergeState;
static ap_uint<LOG2CEIL_NUM_MERGE_STREAMS> rrCtr           = 0;
static ap_uint<LOG2CEIL_NUM_MERGE_STREAMS> streamSource    = 0;
axiWord                 inputWord       = {0, 0, 0, 0};

 switch(mergeState) {
       case M_IDLE:
            bool streamEmpty[NUM_MERGE_STREAMS];
 #pragma HLS ARRAY_PARTITION variable=streamEmpty complete
                for (uint8_t i=0;i<NUM_MERGE_STREAMS;++i)
                streamEmpty[i] = inData[i].empty();
            for (uint8_t i=0;i<NUM_MERGE_STREAMS;++i) {
                uint8_t tempCtr = streamSource + 1 + i;
                if (tempCtr >= NUM_MERGE_STREAMS)
                    tempCtr -= NUM_MERGE_STREAMS;
                if(!streamEmpty[tempCtr]) {
                  streamSource = tempCtr;
                  inputWord = inData[streamSource].read();
                  outData.write(inputWord);
                  if (inputWord.last == 0)
                  mergeState = M_STREAM;
                break;
27                 }
28           }
29           break;
30       case M_STREAM:
31             if (!inData[streamSource].empty()) {
32                   inData[streamSource].read(inputWord);
33                   outData.write(inputWord);
34                   if (inputWord.last == 1)
35                        mergeState = M_IDLE;
36             }
37             break;
38       }
39  }
```

In this example, a module merge is used, which has a stream array as input (`inData`) and a single stream (`outData`) as output. The purpose of this module is to read from the input streams in a fair manner and output the read data to the output stream. The module is implemented as a two-state FSM, which is described using the same constructs that were previously introduced. The focus of the example is on how the merge functionality over the multiple streams is implemented.

The first state in the FSM ensures fairness when choosing the input stream. This is done using a round-robin algorithm to go over the queues. The algorithm starts looking for new data from the queue after the one that was accessed previously. Thus, for example, if in a four queue system, queue 2 was accessed in clock cycle T, in cycle T+1 the search for data to output starts with queue 3 and then goes on to 0, 1, and, finally, 2. The code in lines 17-20 implements the round-robin algorithm. The constant `NUM_MERGE_STREAMS` specifies the number of streams that are to be merged. Subsequently, line 20 tests the current stream, which is identified by the `tempCntr` variable for content. If it is not empty:

- The current stream identified by `tempCntr` is set to be the active stream (line 21).
- Data is read from that stream (line 22).
- If the data word currently read in the input is not the last (checked in line 24), the state machine moves to the `M_STREAM` state, where it outputs the remaining data word from the selected stream identified by `tempCntr`.
- When the last data word is processed, the FSM reverts to state `M_IDLE`, where it repeats the previous process.

Splitting an incoming stream would be a similar process. Data words coming from one stream would be routed appropriately to a stream array.

Extracting and Realigning Fields

Extracting and realigning fields is one of the most fundamental operations in packet processing. Because packets typically arrive in a module through a bus over multiple clock cycles, it is common that fields of interest are not aligned properly in the data word in which they arrive and/or these fields spawn multiple data words.

To process the fields, they must be extricated from the data stream, buffered, and realigned for processing.

Code Example 5 - Source MAC Address Extraction

```
if (!inData.empty()) {
  inData.read(currWord);
  switch(wordCount) {
     case 0:
        MAC_DST = currWord.data.range(47, 0);
        MAC_SRC.range(15, 0) = currWord.data.range(63, 48);
        break;
     case 1:
```

```
9         MAC_SRC.range(47 ,16) = currWord.data.range(31, 0);
10          break;
11      case 2:
12 ……
```

Code Example 5 - Source MAC Address Extraction illustrates a simple field extraction and realignment case, in which the source MAC address is extracted from an Ethernet header. The data arrives over a 64-bit stream called `inData`. In each clock cycle the data is read in (line 2) and, depending on the data word read, the appropriate statement is executed. Thus, in line 6 the first 16 bits of the source MAC address are extracted and shifted to the beginning of the `MAC_SRC` variable. In the next clock cycle, the remaining 32 bits of the MAC address arrive on the bus and are placed in the 32 higher bits of the `MAC_SRC` variable.

Advanced Concepts and Code Examples

In the previous section, the description of a simple three-stage pipeline using Vivado HLS was introduced. However, typical packet processing systems might encompass many modules distributed into several layers of hierarchy.

Creating Systems with Multiple Levels of Hierarchy

Figure 2 shows an example of such a system. The first level of hierarchy consists of two modules, one of which includes three submodules of its own. The top level module looks like the one described in the section above, Setting Up a Simple System. However, the lower level module containing the three submodules uses the `INLINE` pragma to dissolve this function and raise its submodules to the top level, as shown in Code Example 6 - Intermediate Module in Vivado HLS.

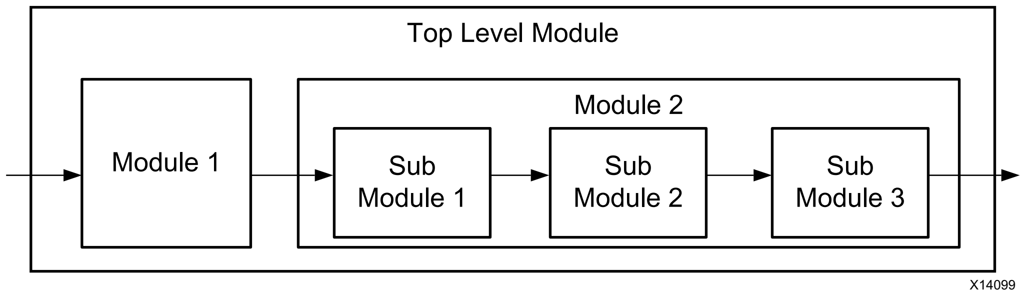


Figure 2: **Example Design with Two Levels of Hierarchy**

Code Example 6 - Intermediate Module in Vivado HLS

```
void module2(stream<axiWord> &inData, stream<axiWord> &outData) {
#pragma HLS INLINE

.........
```

With the inlining of the function, the system resembles Figure 3 after Vivado HLS synthesis. This allows Vivado HLS to create a data flow architecture out of the modules correctly, pipelining and executing all of them concurrently. Module and signal names are maintained as they were after the `inlining` of the function.

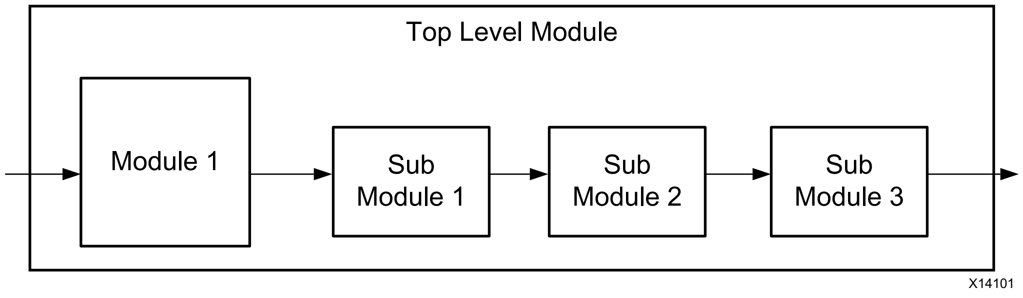


Figure 3: **Intermediate Hierarchy Level Dissolved in a Pipelined Vivado HLS Design**

Using High-Level Language Constructs

One major advantage of Vivado HLS is that it allows you to use high-level language constructs to express complex objects, thus raising the level of abstraction considerably over traditional RTL design. One example of this is the description of a small look-up table.

Code Example 7 - CAM Class Declaration uses a class object to create a table that stores and retrieves the ARP protocol data. The class has one private member, which is an array of `noOfArpTableEntries` number of entries of `arpTableEntry` type. This type is a `struct`, which consists of the MAC address, the corresponding IP address, and a bit that indicates whether this entry contains valid data or not.

Code Example 7 - CAM Class Declaration

```
class cam {
private:
    arpTableEntry filterEntries[noOfArpTableEntries];
public:
    cam();
    bool write(arpTableEntry writeEntry);
    bool clear(ap_uint<32> clearAddress);
    arpTableEntry compare(ap_uint<32> searchAddress);
};
```

The class also includes four methods that operate on this table:

- The Write Method
- The Clear Method
- The Compare Method
- The Constructor Method (shown in Code Example 12 - CAM Class Constructor with Pragma Directive to Partition the Array)

The Write Method

The write method, illustrated in Code Example 8 - Write Method for the CAM Class, takes a new entry as a parameter and stores it in an empty location in the table. Initially, it goes through all the entries in the table and selects the first one that does not contain valid data. This process involves a for loop that goes through all array elements.

For the design to reach the target `II=1`, the for loop must be unrolled completely. Vivado HLS does this automatically if the loop is in a pipelined region (a part of the code in which the pipeline pragma was applied). To unroll the for loops throughout an entire function, apply the pipeline pragma to it.

Alternatively, if the function in which this method is used is pipelined, and the method is inlined into that function upon Vivado synthesis, the method essentially inherits the pipeline property applied to the function through the pragma. This is the approach used in the example design accompanying this application note. In other cases, the unrolling behavior is determined by Vivado HLS, depending on various criteria (presence of variable bounds, number of iterations, etc.). You can explicitly use a pragma unroll directive to instruct Vivado HLS on how to treat the loop. This can be done with the loop unroll pragma. The method returns TRUE if the entry was stored successfully and FALSE when no empty entry was found.

Code Example 8 - Write Method for the CAM Class

```
bool cam::write(arpTableEntry writeEntry) {
  for (uint8_t i=0;i<noOfArpTableEntries;++i) {
     if (this->filterEntries[i].valid == 0) {
        this->filterEntries[i] = writeEntry;
        return true;
             }
       }
  return false;
}
```

The Clear Method

The clear method, shown in Code Example 9 - Clear Method for the CAM Class, allows for the deletion of the entry, which contains the IP address provided as a parameter. The implementation is similar to the write method, with a for loop going through all the entries and comparing the IP addresses of the valid ones with the one provided, and then erasing the first matching entry in the table. Again, TRUE is returned upon success and FALSE if no entry to delete was found.

Code Example 9 - Clear Method for the CAM Class

```
bool cam::clear(ap_uint<32> clearAddress) {
  for (uint8_t i=0;i<noOfArpTableEntries;++i){
     if (this->filterEntries[i].valid == 1 && clearAddress ==
this->filterEntries[i].ipAddress) {
        this->filterEntries[i].valid = 0;
        return true;
            }
       }
  return false;
}
```

The Compare Method

The final method is the compare method, shown in Code Example 10 - Compare Method for the CAM Class. It implements the actual look-up functionality. In this case, an IP address is provided, for which the corresponding MAC address has to be returned. This is accomplished by going through all the entries in the table with a for loop and searching for a valid entry with the same IP address. This entry is then returned in its entirety. An invalid entry is returned if nothing is found.

Code Example 10 - Compare Method for the CAM Class

```
arpTableEntry cam::compare(ap_uint<32> searchAddress) {
  for (uint8_t i=0;i<noOfArpTableEntries;++i){
     if (this->filterEntries[i].valid == 1 && searchAddress ==
this->filterEntries[i].ipAddress)
        return this->filterEntries[i];
       }
  arpTableEntry temp = {0, 0, 0};
  return temp;
}
```

This description demonstrates how Vivado HLS can be used to leverage high-level programming constructs and describe packet processing systems in a software-like manner. This is not possible in RTL.

Ensuring Design Throughput

The previous section introduced the use of a class to describe a self-contained look-up object, which is subsequently synthesized and integrated into modules. While this solution is functionally correct, it does not necessarily ensure that the design reaches the desired throughput target. In a typical protocol processing design, packets arrive over a bus over multiple clock cycles. A maximum of one new data word per clock cycle might have to be processed. For example, in a 10 Gb/s design, processing packets at line rate requires that the design can consume one 64-bit data word every clock cycle at 156 MHz. Widening the bus results in a reduced frequency requirement or in increased headroom while processing the data (thus an `II = 2` might be possible). For the purposes of this discussion, assume that the target `II` is always 1. Similar methodologies can be followed to design systems with different `II` targets.

To attain the target `II` goal, it is important to ensure that Vivado HLS can access the required streams and variables in a timely fashion. A straightforward example of a code snippet that violates this principle is shown below in Code Example 11 - Where the II = 1 Constraint Cannot be Met. This example modifies the code from Code Example 5 - Source MAC Address Extraction and attempts to complete the realignment of the MAC source address in one state (state 0).

Code Example 11 - Where the II = 1 Constraint Cannot be Met

```
1  switch(wordCount) {
2    case 0:
3       if (!inData.empty()) {
4          inData.read(currWord);
5          MAC_DST = currWord.data.range(47, 0);
6          MAC_SRC.range(15, 0) = currWord.data.range(63, 48);
7           outData.write(currWord);
8       }
9       if (!inData.empty()) {
10          inData.read(currWord);
11          MAC_SRC.range(47 ,16) = currWord.data.range(31, 0);
12          outData.write(currWord);
13 }
14              break;
15   default:
16      if (inData.read_nb(currWord))
17          outData.write(currWord);
18      break;
19 }
```

Synthesizing this code in Vivado HLS results in an `II = 2`. The Vivado synthesis output at the console window contains the following message:

```
@W [SCHED-68] Unable to enforce a carried dependency constraint (II = 1,
distance = 1)
   between fifo read on port 'inData_V_data_V'
(hlsProtocolProcessing/sources/iiExample.cpp:8) and fifo read on port
'inData_V_data_V' (hlsProtocolProcessing/sources/iiExample.cpp:3).
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

This message informs you that a carried dependency between the two reads inhibits the synthesis process from scheduling the accesses to achieve an `II = 1`. The issue: in the code lines indicated, the module attempts to access the stream `inData` twice in the same clock cycle. Because this is impossible in a stream (which represents a single port of a memory/FIFO construct), synthesis fails to meet the set constraints. Similar limitations apply when accessing static variables used to maintain information between states. You must therefore use caution when scheduling accesses in states to reach the desired throughput goal.

More complex access issues can arise in complex designs, such the look-up table, which was introduced in the previous section. If you synthesize the code provided there as-is, Vivado HLS cannot meet a target `II = 1`. This is because of access congestion due to the limited number of memory ports. By default, Vivado HLS uses a block RAM to store the table entries, and a block RAM contains two access ports. To parse all the entries of an eight-entry table, the module requires at least four clock cycles, which means it cannot attain the target `II = 1`. To address this issue, you must instruct Vivado HLS to partition the array in which the table entries are stored. Partitioning the array essentially breaks it down to multiple, smaller arrays and allows the use of more memories and, therefore, more access ports. In the most extreme case, which is the one used in Code Example 12 - CAM Class Constructor with Pragma Directive to Partition the Array, you can partition an array completely, which essentially creates a register array in which the individual elements are stored.

Code Example 12 - CAM Class Constructor with Pragma Directive to Partition the Array

```
cam::cam(){
  #pragma VHLS array_partition variable=filterEntries complete
  for (uint8_t i=0;i<noOfArpTableEntries;++i)
     this->filterEntries[i].valid = 0;
}
```

This code snippet shows the constructor for the table class described in the previous section. The constructor sets all entries to invalid and also includes the pragma that partitions the array. Alternatively, the pragma could be specified in the function in which the object of the class was instantiated. In that case, the pragma would apply only to the specific object; used as shown in the example, however, it applies to all objects of this class.

Example Design

Example Design File Location and Details

You can download the design files for this application note from the following location:

https://secure.xilinx.com/webreg/clickthrough.do?cid=361691

The following table provides details about the example design.

Table 1: **Example Design Details**

Parameter	Description
General	
Developer Name	Xilinx
Target Devices	• HLS Design: All Xilinx devices • Vivado Design Suite Design: Virtex®-7 XC7VX690T-2FFG1761C
Source code provided?	Y
Source code format (if provided)	C++, Verilog, VHDL
Design uses code or IP from existing reference design, application note, 3rd party or Vivado Design Suite software	• Ten Gigabit Ethernet PCS/PMA (10GBASE-R/KR) v4.1 • Ten Gigabit Ethernet MAC v13.0 • FIFO Generator v11.0 • AXI4-Stream Register Slice v1.1
FIFO Generator v11.0	
AXI4-Stream Register Slice v1.1	

Table 1: **Example Design Details *(Cont'd)***

Parameter	Description
Simulation	
Functional simulation performed	Y
Timing simulation performed?	Y
Testbench provided for functional and timing simulation?	Y
Testbench format	C++
Simulator software and version	Any supported by HLS (the Vivado simulator is the default)
SPICE/IBIS simulations	N
Implementation software tool(s) and version	Vivado HLS 2014.1 and later, Vivado 2013.4 and later
Hardware Verification	
Hardware verified?	Y
Platform used for verification	Xilinx VC709 Development Board

Example Design Description

To better describe the concepts introduced in this application note, a simple system implementing basic ping and ARP functionality is provided. In a typical network processing subsystem, this system would:

- Reside between the Ethernet MAC and a user application.
- Respond to ping requests as well as provide support for MAC address resolution, while looping back all other packets. (For use in a real system, the loopback module would have to be replaced with your application.)

Figure 4 shows the structure of the example design system. It consists of a parser module that identifies the packet type for incoming packets and forwards the packets to one of the following modules:

- ARP Server: The ARP server responds to ARP requests directed at this system and handles MAC address resolution requests instigated by your application.
- Internet Control Message Protocol (ICMP) Server: The ICMP server replies to ping requests sent to this system.
- Loopback: The loopback sends packets back through the MAC and into the network without change.

At the system output, the merge module recombines traffic streams from the three other modules and produces one output stream to send to the network.

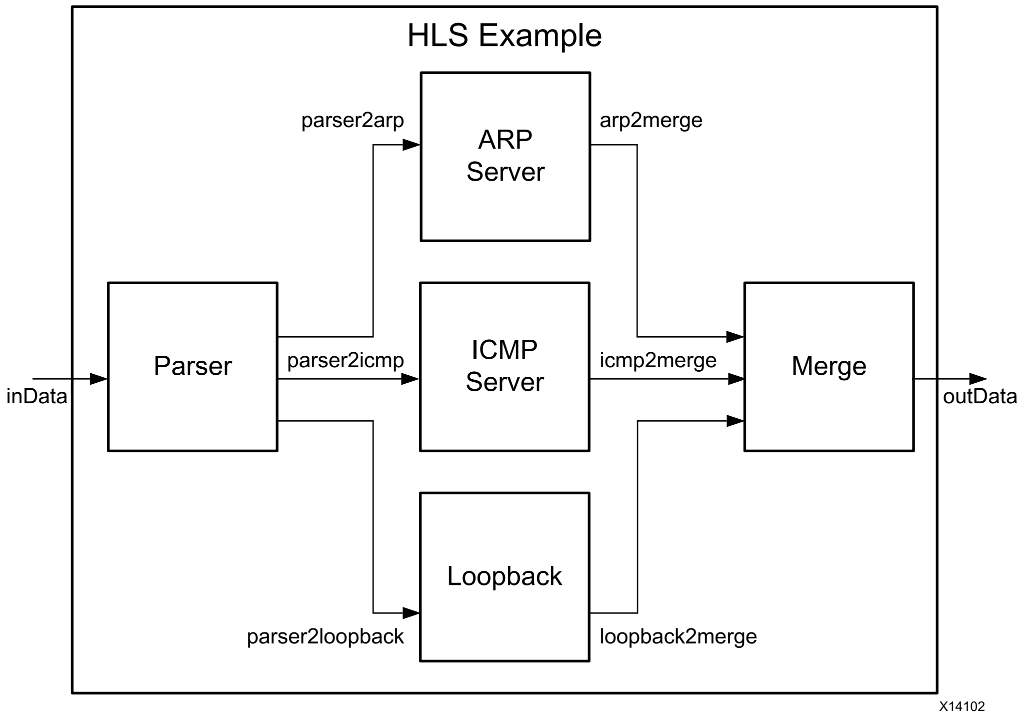


Figure 4: **Structure of the Example Design System**

The description of the system follows the same principles introduced in the first part of this application note. The top level function `VHLSExample` consists of five sub-functions, each corresponding to one of the submodules shown in Figure 4. All modules are connected using streams. The external ports of the system are configured to use AXI4-Stream interfaces. The typical set of pragmas is applied. In this case, the depth property of the `STREAM` pragma is set to 1, but larger values might be necessary to address transient effects in the system. This is not correct. The external port streams use AXI4S. The internal ones use the ap_fifo I/F, which is what HLS streams typically use by default.

Increasing the value for the depth of a stream naturally increases resource usage as well. This increase happens step-wise because the basic buildings blocks used for a stream (either LUTs and flip-flops or block RAMs) can accommodate a specific number of entries before having to add more resources to store additional entries. To illustrate this, Table 2 shows the resources used for various stream depth values. For values 1 and 4, the amount of resources used is identical because no additional resources had to be used to fit the extra entries; whereas, when the number of entries is increased to 8, resource consumption also increases commensurately.

Table 2: **Resource Use for Different Stream Depth Values**

Stream Depth	LUTs	Flip-Flops	Block RAMs
1	2564	210	0
4	2564	210	0
8	2456	1592	40

Moving down one hierarchy level, the Parser and the ICMP server consist of multiple submodules joined together in a pipelined fashion. Figure 5 illustrates this for the Parser. There are three submodules:

- Ethernet Detection: Checks the `EtherType` field in the Ethernet frame header and determines the lower layer protocol. It then forwards the packet either to the ARP server or to the Length Adjust module.

- Length Adjust: Readjusts the packet by stripping away any padding added by the Ethernet layer to meet the minimum packet size requirements, making the packet length in the IP header equal to the actual packet length.
- ICMP Detection: Uses the `protocol` field in the IPv4 header to detect any ICMP packet and forward it to the ICMP server or Loopback module.

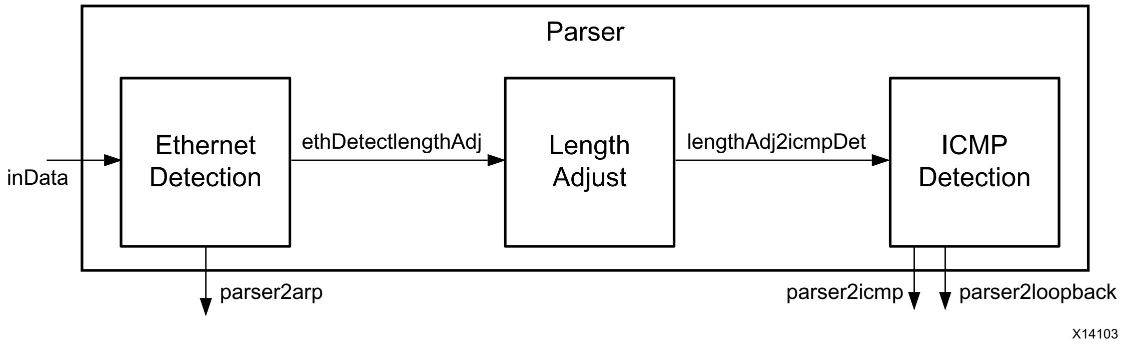


Figure 5: **Parser Block Diagram**

All the submodules are state machines that adhere to the FSM description methodology (introduced in the section Implementing a State Machine with Vivado HLS, page 5).

The ICMP server receives ICMP packets from the parser, processes them, and creates ping replies for valid packets. Its structure is shown in Figure 6. It consists of three stages arranged in a pipeline manner with an additional signal that jumps over the dropper and forwards the IP checksum directly to the IP checksum module. The three modules are:

- Create Reply: Parses the ICMP header, determines if the packet is a valid ICMP packet, creates a reply, and calculates the IP checksum for the newly created reply packet. This checksum is then forwarded to the Insert Checksum module. The packet status (valid or invalid) is signaled to the Dropper over the `validBuffer`.
- The Dropper allows valid packets to stream through it, while filtering invalid packets and removing them from the packet stream.
- The Insert Checksum module receives the checksum for the newly created ICMP reply packet over the `cr2checksum` stream and reinserts it into valid packets, which it receives from the Dropper.

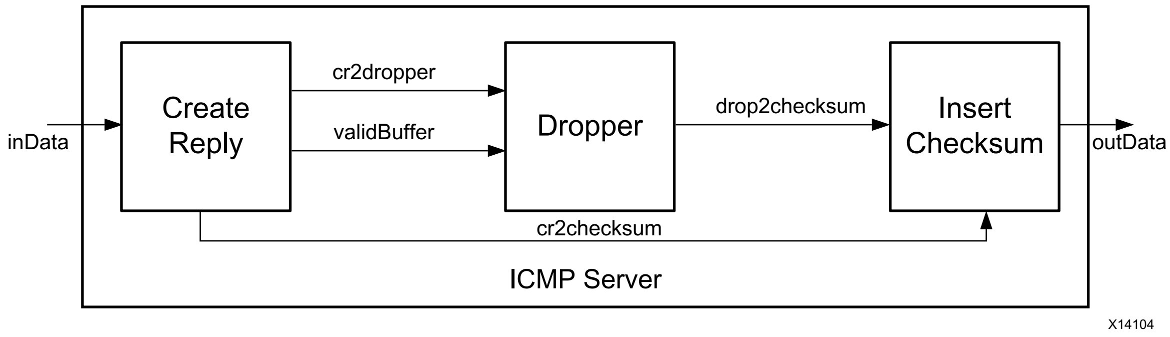


Figure 6: **ICMP Server Block Diagram**

Both the Parser and the ICMP server are inlined into the top level functions and are dissolved upon Vivado synthesis. This allows HLS to better optimize the scheduling of the design in the top level. This results in a final system architecture that resembles Figure 7. The dashed lines demarcate the location of the parser and ICMP modules in the source code. In the synthesized system the mid-level modules have been removed and their submodules brought to the top level. Stream names have been omitted for the sake of clarity.

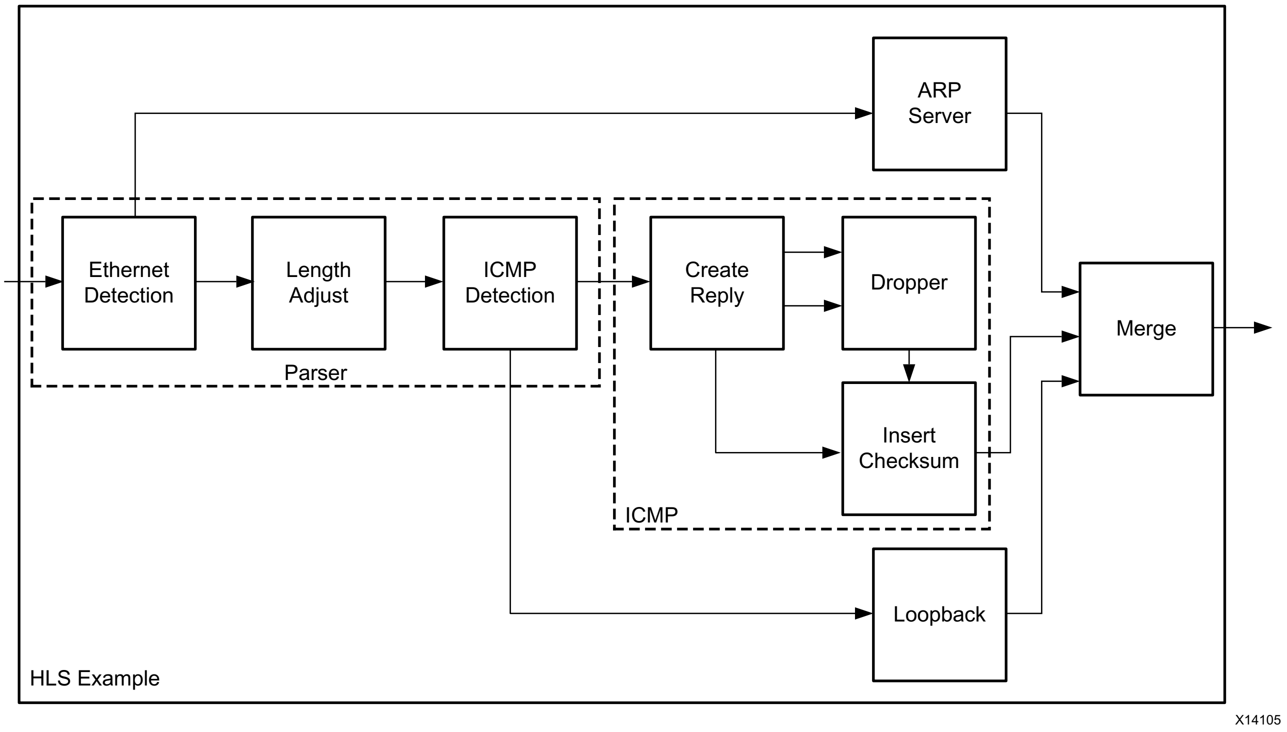


Figure 7: **System Resulting from the Vivado Synthesis**

Table 3 and Table 4 contain excerpts from an expanded Vivado synthesis report that shows the submodules of the `VHLSExample` module. The complete Vivado HLS report is explained in more detail in the subsequent sections.

Table 3: **Post Vivado Synthesis Sub-module List**

Instance	Module	Latency		Interval		Type
		Minimum	Maximum	Minimum	Maximum	
`grp_detect_mac_protocol_fu_799`	`detect_mac_protocol`	1	1	1	1	Function
`grp_cut_length_fu_839`	`cut_length`	1	1	1	1	Function
`grp_detect_ip_protocol_fu_731`	`detect_ip_protocol`	1	1	1	1	Function
`grp_arp_server_fu_547`	`arp_server`	2	2	1	1	Function
`grp_createReply_fu_689`	`createReply`	1	1	1	1	Function
`grp_dropper_fu_863`	`dropper`	1	1	1	1	Function
`grp_insertChecksum_fu_775`	`insertChecksum`	1	1	1	1	Function
`grp_loopback_fu_887`	`loopback`	1	1	1	1	Function
`grp_merge_fu_649`	`merge`	1	1	1	1	Function

Table 4: **Post Vivado Synthesis Sub-Module Resource Utilization List**

Instance	Module	BRAM_18K	DSP48E	Flip-Flops	LUTs
`arp_server_U0`	`arp_server`	0	0	1355	1718
`createReply_U0`	`createReply`	0	0	553	555
`dropper_U0`	`dropper`	0	0	213	12
`ethernetDetection_U0`	`ethernetDetection`	0	0	433	95
`icmpDetection_U0`	`icmpDetection`	0	0	225	841
`insertChecksum_U0`	`insertChecksum`	0	0	369	120
`lengthAdjust_U0`	`lengthAdjust`	0	0	236	67
`loopback_U0`	`loopback`	0	0	205	4
`merge_U0`	`merge`	0	0	414	673
Total	**9**	**0**	**0**	**4003**	**4085**

The final module of interest in the design is the ARP server, which implements two functions:

- Receives ARP requests over the network, determines whether or not these requests are destined for this node, and if so, sends a reply to the requesting node containing the MAC address of this node. The MAC address is hard coded in the source code.
- Receives external requests to resolve the MAC of an IP address. This is received over the `queryIP` stream shown in Figure 8. The ARP server then looks up the IP address in an internal table, which it maintains. If a match is found, the ARP server reads the entry from the table and returns the value for the MAC address over the `returnMAC` stream. If no match is found, the module sends an ARP Request for this IP address to the network broadcast address and waits for a reply. If nothing is received, the operation times out and the module returns to its idle state. If a reply is received, the MAC address corresponding to the requested IP address is stored in the internal table for future use and then sent back over the `returnMAC` stream.

The `queryIP` and `returnMAC` streams use the normal `ap_fifo` interface, which is the native interface of an Vivado HLS stream. It resembles a typical FIFO interface, with read, write, full and empty signals.

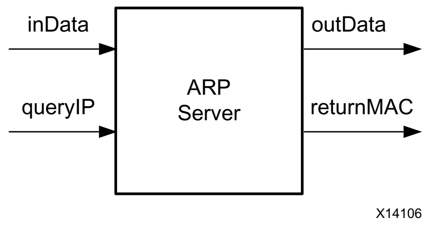


Figure 8: **ARP Server Block Diagram**

Example Design Contents and Descriptions

The example design accompanying this application note consists of:

- A Vivado HLS project, that contains all the modules described previously. This is described in the section Protocol Processing Example: Vivado HLS Project, below.
- A Vivado Design Suite project that integrates the Vivado HLS modules with the necessary companion modules (for example, an Ethernet MAC core) to produce a functional example design targeting the VC709 evaluation board. This is described in the sections Using Vivado Design Suite to Implement the Design, page 21 and Testing the Example Design on the VC709 Evaluation Board, page 22

Protocol Processing Example: Vivado HLS Project

The example design provided contains six C++ source code files, one for each module and one accompanying header file for each. An additional header file (`globals.hpp`) contains declarations pertinent to the entire project. Finally, a test bench file (`VHLSExample_tb.cpp`) is provided to facilitate simulation and verification of the design from within the Vivado HLS environment. All the files are located in the sources subfolder in the project directory.

The project is pre-configured to be built for a clock period of 6.66 ns and targets an XCVX690T device by default. You can change these settings through the solution and project settings.

Building the project generates a detailed report, an example of which is shown in Figure 9. This report contains critical information about the generated design.

Important: Examine the report to determine whether or not Vivado HLS can produce a design that meets all the set constraints.

Performance Reporting

The first part of the report provides performance estimates. This includes information on the throughput and frequency of the design. In this case, the design meets timing (achieved clock period 6.38 ns).

Important: Keep in mind that timing value is a Vivado HLS estimate and might vary from the post synthesis or post place and route value. This is because Vivado HLS estimates timing by using fixed timing values for different operation types and devices. This results in two sources of discrepancy:

- Vivado HLS does not perform full logic synthesis, thus cannot take advantage of simplifications of logic that might result from it.
- Vivado HLS does not take into account detailed placement and routing information, which cannot be available at the time.

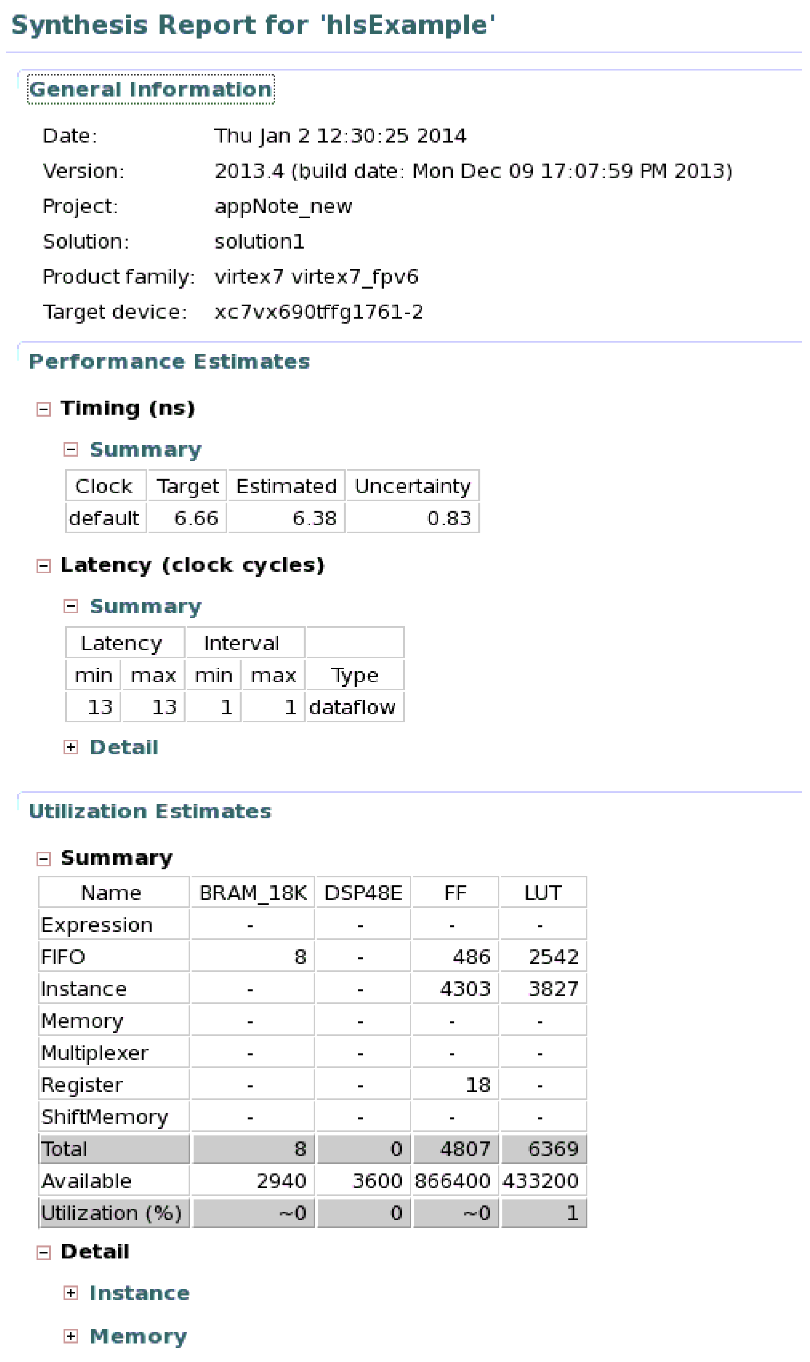
A more precise estimate can be obtained by evaluating the design through the export menu (click the **Solution** menu, then choose **Export RTL**, and on the pop-up window check the **Evaluate** check box. Click **OK** to run the evaluation). This runs logic synthesis, place, and then route on the design. Again, the resulting timing might vary from what is achieved in the final design, in which placement and routing is changed to accommodate any additional logic any additional logic found in the non-HLS portion of your design, though this estimation is inherently more precise than the one generated by Vivado HLS synthesis. Typically, state machine-based designs exhibit better timing after place and route compared to Vivado HLS synthesis reports, so utilizing this additional step occasionally to glimpse the post synthesis design performance is recommended.

Other performance information that can be obtained from the report is the final latency and `II` value for this design. The example design used in this application note has a total latency of 13 clock cycles and an `II = 1`, as requested. If any of the constraints are not met, you can find more information about the issue in the console window. You can analyze latency further using the Analysis view. This can provide a detailed overview of the scheduling of the module and be used to account for each cycle of latency reported by Vivado synthesis. Clicking on each module name opens a detailed view specific to that module.

Going back to the main report, you can click **Instance** under Detail to obtain a breakdown of the latency and `II` information per module. Clicking on each module name opens a separate report for that particular module. These reports match the main report in format and content type. You can use this part of the report to identify which module in the design fails the `II` target, or to determine the latency incurred by each module.

Example of Auto-generated Synthesis Report

As mentioned earlier, the Vivado Design Suite automatically generates a report after you build and synthesize your project. An example report is shown in the figure below.



Synthesis Report for 'hlsExample'

General Information

Date:	Thu Jan 2 12:30:25 2014
Version:	2013.4 (build date: Mon Dec 09 17:07:59 PM 2013)
Project:	appNote_new
Solution:	solution1
Product family:	virtex7 virtex7_fpv6
Target device:	xc7vx690tffg1761-2

Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
default	6.66	6.38	0.83

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
13	13	1	1	dataflow

⊞ Detail

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	8	-	486	2542
Instance	-	-	4303	3827
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	18	-
ShiftMemory	-	-	-	-
Total	8	0	4807	6369
Available	2940	3600	866400	433200
Utilization (%)	~0	0	~0	1

⊟ Detail

⊞ Instance

⊞ Memory

Figure 9: **Example Synthesis Report for the Example Design**

Resource Utilization Reporting

The second part of the report contains resource utilization estimates. These estimates might (and usually do) differ from the final values that result after running logic synthesis on the design for reasons similar to those of the performance estimation. Furthermore, the breakdown in the utilization might vary because logic synthesis might decide to use LUTs to implement something for which Vivado HLS estimated that a DSP48 slice will be used, or logic synthesis might use distributed RAM instead of block RAM for the implementation of a queue. Expanding the **Instance** menu lists the resource use for each submodule, as it does in the Peformance section. Expanding the **FIFO** menu provides detailed information on all internal streams in the design and the resources they use.

The final part of the report (not shown here) lists the module interfaces. In this case, there are two AXI4-Stream interfaces, `inData` and `outData`, two `ap_fifo` interfaces, `queryIP` and `returnMAC`, along with a host of control signals used to drive the generated Vivado HLS core.

To facilitate C and C/RTL verification, the example design includes a test scenario that exercises all the four paths present in the design. In this scenario, packets are read from the input file (called `in.dat`) and are injected in the systems input queue. These packets include ARP requests, ICMP requests, and Transmission Control Protocol (TCP) packets. The ARP and ICMP requests are answered by the system, while the TCP packet is not recognized and looped back. At the end of the test an IP address is written into the `queryIP` queue and the ARP server produces and sends an ARP request corresponding to that address. The output from the simulation is compared with a golden output file called `gold.dat`. Thus, running the C simulation involves (after successfully building the project, of course) navigating to the `/project_dir/solution1/csim/build` folder and typing the following commands (when using any Linux system):

```
./vhlsExample
/project_dir/sources/csim/in.dat
/project_dir/sources/csim/queryReply.dat
/project_dir/sources/csim/gold.dat
/project_dir/sources/csim/out.dat
```

The next step following successful C verification is to use the C/RTL co-simulation to ensure correct functionality of the generated RTL. Vivado HLS allows seamless transition from the C to the C/RTL co-simulation. The tool automatically generates an appropriate RTL test bench from the provided C one, executes the RTL simulation, and then compares the output to the golden one, just like in the C simulation.

The final step before integrating the example design in the Vivado Design Suite for logic synthesis is exporting the core. To do this, select **Export RTL** on the **Solution** menu. From the various format options select **IP Catalog**. Click **OK** and Vivado HLS generates the IP core in the standard IP-XACT format. The generated files are located in the `/project_dir/solution1/impl/ip` folder.

To facilitate execution of all the design steps described above, the example design includes a Tcl script that you can use to perform all the above steps in sequence. The script is called `run_hls.tcl`. To execute the script, type `vivado_hls -f run_hls.tcl` in the console.

Using Vivado Design Suite to Implement the Design

After generating the core in Vivado HLS, the core needs to be imported into the provided Vivado Design Suite project. This project was generated using Vivado Design Suite 2014.1, although it should be possible to use newer versions. It targets the Xilinx VC709 development board [Ref 3].

Opening the Vivado Design Suite project accompanying the Vivado HLS project brings up a screen containing the design sources for the project. These include the network interface and its accompanying clock generation signals, the Vivado HLS module, and two modules that facilitate testing of the design.

The Vivado HLS-generated IP core appears in the sources pane marked with a red exclamation mark, which denotes that the IP core cannot be found in any of the IP repositories currently in use with this Vivado Design Suite installation. This can be resolved by adding the IP core generated from Vivado HLS in the previous steps in an existing or new repository. It is not, however, necessary for synthesizing and implementing the design because the design files for the core are already included in the example design project.

The two additional test bench modules used are the debouncer, which eliminates voltage level oscillations from pressing the push buttons on the board, and the `queryGenerator`, which produces an IP address, the MAC address of which is being requested from the ARP server, and reads the reply from the Vivado HLS module. See Figure 10.

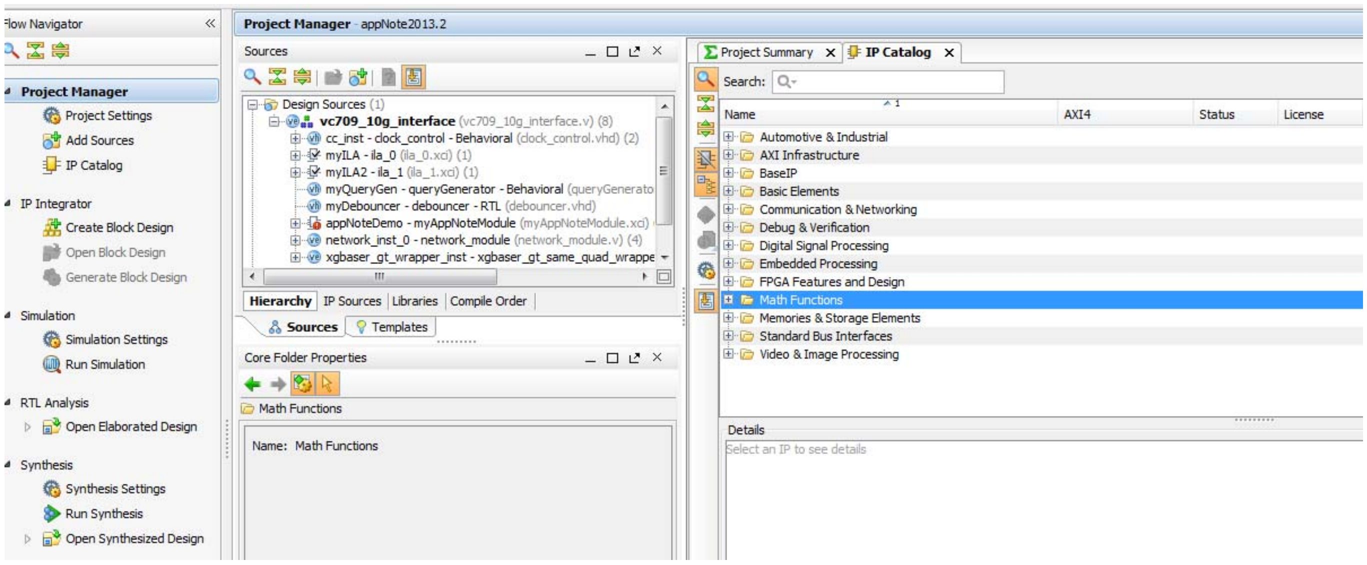


Figure 10: **Overview of the Example Design in the Vivado Design Suite Project**

The example design is readily implemented using Vivado tools. Generating a bitstream and downloading it to a VC709 evaluation board immediately starts the system.

Testing the Example Design on the VC709 Evaluation Board

To test the design on the VC709 evaluation board, you must connect the board to a PC, which serves as the counterpart and produces the test stimuli. This can be done either directly if the PC has an SFP port, or over a switch which contains both SFP and standard Ethernet ports. You can use an open-source program, such as Wireshark, to monitor traffic on any network interface. This allows you to monitor the packets that are exchanged between the PC and the VC709 evaluation board to verify that the correct information exchange takes place.

The simplest scenario that can be used to test the system is to send it a simple ping request by typing:

```
ping 1.1.1.1
```

at a Linux or Windows terminal. This initially sends an ARP request to determine the MAC address belonging to this IP address. The Vivado HLS module responds with an ARP reply containing the MAC corresponding to the VC709 evaluation board. The PC then sends a ping request to the VC709 evaluation board, which produces a ping reply to each ping request until you interrupt the process on the PC side. Note that only the first ping request triggers an ARP request. In all subsequent ping requests, the MAC address for the VC709 evaluation board is found on the PCs ARP table, and thus no additional ARP requests are triggered. Testing isolated ARP requests is possible by using the arping command in Linux:

```
arping -I ethInterfaceName 1.1.1.1
```

Testing the ARP request functionality is done by using the south push button on the right side of the VC709 evaluation board. Every time the button is pressed, a query for the IP address `1.1.1.2` is generated. If no additional configuration is performed at the PC, these requests time out and fail because the PC ARP table does not contain a MAC address corresponding to this IP address. The `arp` command has to be used to add this IP address to the PC ARP table:

```
arp -s 1.1.1.2 AB:90:78:56:34:12
```

Alternatively, the PC IP address on that interface can be set to `1.1.1.10`. All other intermittent traffic sent from the PC to the VC709 evaluation board is looped back to the PC without changes.

Conclusions

Vivado HLS allows quick and easy implementation of protocol processing designs on FPGAs using C/C++ and leveraging the productivity increases offered by higher level languages as opposed to traditional RTL. You can take advantage of additional features offered by Vivado HLS to target the desired architecture and to quickly explore design trade-offs without rewriting the source code. Such features include:

- Straightforward system assembly using C functions
- Data exchange over streams (which offer standardized FIFO-like interfaces and free-flow control)
- Vivado HLS pragmas

As a vehicle for explaining the basic concepts of such designs, this application note uses a simple ARP/ICMP server that replies to ping and ARP requests and resolves IP address queries. You can synthesize the example design with Vivado HLS and integrate the design into the accompanying infrastructure, which allows it to be tested using a Xilinx VC709 evaluation board. This demonstrates that Vivado HLS designed modules can perform protocol processing at line rates of 10 Gb/s and higher.

References

The following references are used in this application note:

1. Software Defined Specification Environment for Networking (SDNet)
2. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
3. *VC709 Evaluation Board for the Virtex-7 FPGA: User Guide* (UG887)

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
08/08/2014	1.0.1	Corrected design files link.
05/30/2014	1.0	Initial Xilinx release.

Notice of Disclaimer