XMOS Timing Analyzer Whitepaper

Version 1.0

This document explains how *static timing analysis* can be used to determine whether timing-critical sections of code when compiled and run on XMOS devices are guaranteed to complete within their deadlines. It introduces the XMOS Timing Analyzer tool, which can help you automate this task.

Publication Date: 2010/05/18

Copyright © 2010 XMOS Ltd. All Rights Reserved.

1 Introduction

To guarantee the correct behavior of real-time software running on embedded processors can pose a significant challenge. Data-dependent control flow, where execution times of many functions are dependent on the data inputs, means that instruction sequences are hard to predict. In many systems, the presence of a memory hierarchy complicates this problem further, requiring the state of all caches and the system bus to be accurately modeled in order to predict the time of load and store operations. Interrupt-driven I/O processing, in which external events can alter execution flow at any time, can make it impossible to predict all possible states of the machine and thus accurately time a section of code.

The traditional approach to verifying timing involves writing test benches to exericse each function, for example by observing pin activity or counting instructions to determine execution time. However, creating stimuli that cover all timing corner cases may not be possible until the whole of the system is built, or even after the product is deployed. This can significantly increase time-to-market and in the worst case lead to a product recall.

XMOS devices can be programmed to respond to events instead of interrupts: event handlers exist in the current context, which means no time is spent context switching. As a result, response times are virtually zero, enabling many real-time interfaces to be implemented. Events steer the execution through the code along well-defined paths, and no uncertainty is introduced due to caches, buses or interrupts.

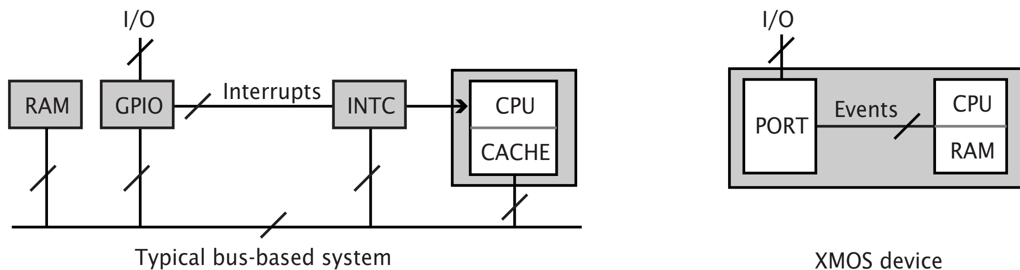

Figure 1: A typical bus-based system compared to an XMOS device.

The XMOS Timing Analyzer (XTA) allows you to determine the performance of code compiled for any XMOS device. Using either the interactive GUI or a script, you can specify the time in which sections of source code must be executed, for example the time taken to handle an event. The tool identifies all corresponding paths through the object and times them, using the worst-case time to determine a pass or fail result. If the tool detects a timing failure, it produces feedback that helps you optimize parts of your program until timing closure is achieved.

2 Ethernet MII Receive Specification

The Ethernet MII interface is an example of where it is advantageous to implement a real-time interface in software. Using software allows early adoption of new hardware standards and allows custom protocols to be implemented.

The waveform diagram below illustrates the operation of the MII receive protocol.

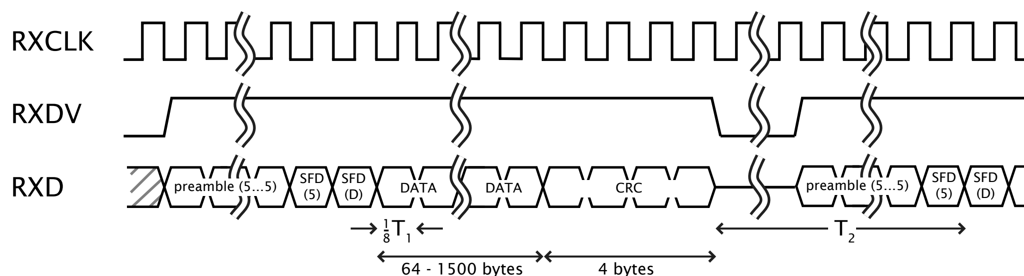

Figure 2: Ethernet MII Receive Waveform Diagram

The signals are as follows:

- RXCLK is a free running clock generated by the Ethernet PHY.
- RXDV is a data valid signal driven high by the PHY during frame transmission.
- RXD carries a nibble of data per clock period from the PHY to the receiver.

The receiver is required to wait for a preamble of nibbles of values 0x5, followed by two nibbles with values 0x5 and 0xD. It then inputs the actual data, which is in the range of 64 to 1500 bytes, followed by four bytes containing a CRC.

When run at a rate of 100Mbps:

- The value of T1 is 320ns.
- The value of T2 is 1520ns.

3 Ethernet MII Receive XC Implementation

The XC language [1] provides extensions to C that simplify control over I/O and the processing of events. An XC program that implements the MII receive interface is shown below.

Program 1 XC Implementation of MII Receive Protocol

```
buffered in port:32 RXD  = XS1_PORT_4A;
         in port    RXDV = XS1_PORT_1I;

void miiRec() {

  RXDV when pinseq(0) :> void;
  while (1) {
    int eop = 0;

#pragma xta label "wait_for_sfd"
    RXD when pinseq(0xD) :> void;
    ...
    do {
      select {
#pragma xta label "word_receive"
        case RXD :> word :
          // process word
          break;
#pragma xta label "rx_dv_low"
        case RXDV when pinseq(0) :> void :
          eop = 1;
          // input and process part-word
          switch (part word size) { compute crc and error condition }
          switch (error condition) { handle error }
          break;
        }
    } while (!eop);
  }
}
```

The main operations performed by this program are described by their line numbers in the paragraphs below:

1, 2 The pins are mapped to the ports `RXD` and `RXDV`, and the data port `RXD` is configured to convert a stream of data nibbles from the PHY into a stream of words for input by the processor.

In `main` (not shown), the port `RXD` is synchronized to the clock signal RXCLK and is configured to use a 1-bit port `RXDV` as a ready-in strobe signal that causes data to be sampled only when the signal is high.

6 The program initializes itself by performing a *conditional input* that waits for the signal RXDV to be low.

11 The program waits for the start of the next frame by conditionally inputting the last nibble of the preamble (0xD).

14 The `select` statement is used to wait for an event on one of a set of ports and respond to it. In this example, the processor waits for either the next word of data from RXD (line 16) or for the data valid signal RXDV to go low (line 20).

4 Timing the Ethernet MII Implementation

When compiled and run on an XMOS device, the Ethernet MII implementation must execute fast enough to meet the MII timing specification. For 100Mbs Ethernet, the following two timing requirements must be met:

- The processor must always be ready to input a word of data every 320ns (T1 in Figure 2), which means that the time to execute the innermost loop (lines 16→18, 27, 13→14) must be executed within this time.
- After detecting the data-valid signal going low, the processor must process the last packet of data and be ready to detect the next packet's SFD nibble with value 0xD (lines 20→25, 27→28, 7→11) within 1520ns (T2 in Figure 2).

Using the XTA GUI, shown on the following page, you can view the source code and select which endpoints to time between. The XTA analyzes all paths between the endpoints in the object code and displays them visually.

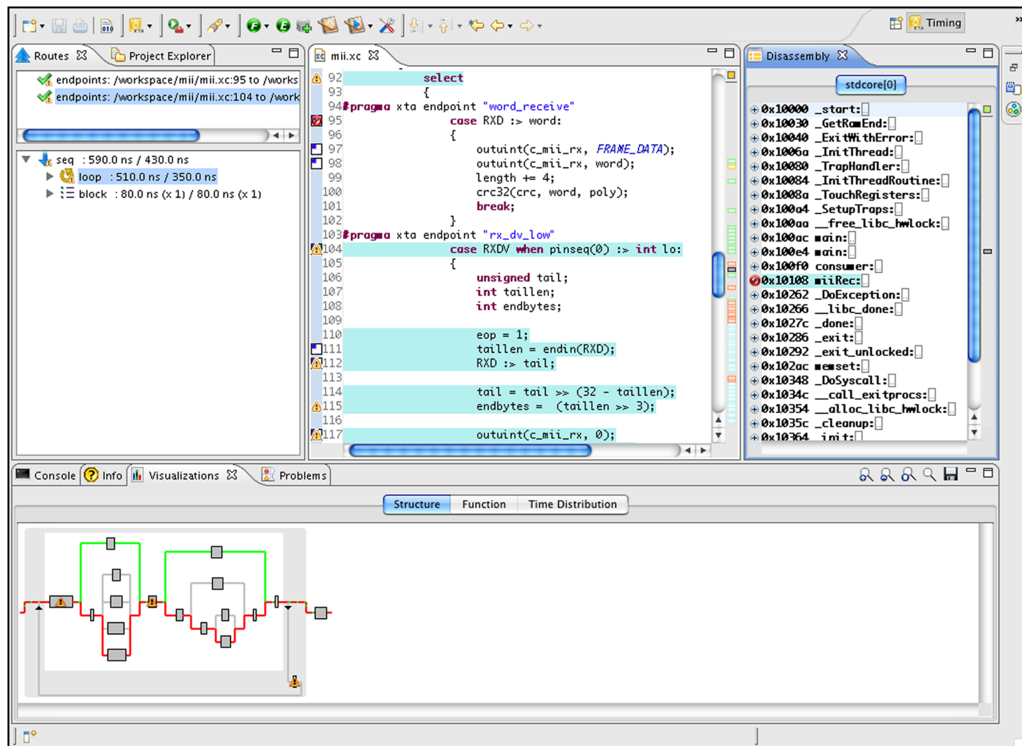

Figure 3: The XTA GUI

Analyzing the first route bewteen the endpoint labeled `word_receive` (line 15 in Program 1) and itself identifies a single path. In the GUI, you can specify a timing requirement of 320ns for this route, which the tool indicates is met with a green tick next the the route name.

Analyzing the second route between the endpoints labeled `rd_dv_low` (line 19 in Program 1) and `wait_for_sfd` (line 11) identifies more than one path. These paths are due to branching within the body of the case statements (lines 23-24) that deal with the CRC calculation and error cases. In Figure 3, these multiple paths are shown graphically in the bottom panel of the GUI. The path with the best-case time is shown in green, the worst-case time in red. In the GUI, you can specify a timing requirement of 1520ns for this route, which the tool indicates is met with a green tick. Note that the tool finds *all* possible paths and times them, providing 100% functional coverage on timing.

The XTA tool can be used for more than just pass/fail testing. Structural code views highlight timing hot-spots, and instruction-level views and traces can highlight

hardware resource contention. This information helps you focus on optimizing code where it has the greatest impact. If a degree of slack is present when meeting the timing requirements, you can use the XTA to determine how much the processor frequency can be reduced, thereby saving power.

5 Closing Timing At Compile-Time

Having interactively specified the timing requirements for pairs of endpoints in the program that need to execute in real-time, you can instruct the tool to generate a script that automatically checks these requirements are met at compile-time. The script generated for the MII code looks as follows:

```
analyze endpoints word_receive word_receive
set required - 320 ns
analyze endpoints rx_dv_low wait_for_sfd
set required - 1520 ns
print summary
```

The XTA reports the worst-case time for all routes as well as the slack or violation. In the MII example, the output at compile-time is as follows:

```
PASS (required 320.0ns, worst-case 180.0ns, slack 140.0ns)
PASS (required 1520.0ns, worst-case 1200.0ns, slack 320.0ns)
```

6 Conclusion

This document has shown how static timing analysis can be used to time sections of code that have real-time requirements. It introduced the XMOS Timing Analyzer, which identifies the execution paths through a program and times them for a target XMOS device. Timing requirements are specified either interactively using a GUI or as a script. The XTA determines worst-case execution time, reporting timing failures as compilation errors, or providing a guarantee that all requirements are met.

7 What to Read Next

This document provides only an introduction to the concept of static timing analysis and the XTA tool. The following documents provide more in-depth information:

- **Validating Timing Constraints with the XMOS Tools** [2]: Provides a “hands-on” tutorial that involves timing a UART implementation using the XTA.

An interactive version of this tutorial is also available in the XMOS Development Environment (Version 10.4+).

- **The XMOS Tools User Guide** [3]: Includes a chapter on how to use many features of the XTA GUI, which is released as part of the XMOS Development Environment.
- **Taking the guesswork out of timing in real-time software systems** [4]: Provides background information on static timing analysis, an overview of the XTA tool, and outlines future possibilities for the XTA technology.

You may also find information from the following online resources useful:

- The XMOS Corporate Website: http://www.xmos.com/
- The XMOS Community Website: http://www.xcore.com/

Bibliography

[1] Douglas Watt. *Programming XC on XMOS Devices*. XMOS Limited, Sep 2009. http://www.xmos.com/published/xc_en.

[2] Douglas Watt. Validating Timing Constraints with the XMOS Tools. Website, 2010. http://www.xmos.com/published/timingtut.

[3] Douglas Watt and Huw Geddes. *The XMOS Tools User Guide*. XMOS Limited, 2009. http://www.xmos.com/published/xtools_en.

[4] Peter Hedinger and Edward Clarke. Taking the guesswork out of timing in real-time software systems. Website, 2010. http://www.xmos.com/published/xta-article.

Disclaimer

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

Copyright © 2010 XMOS Ltd. All Rights Reserved. XMOS and the XMOS logo are registered trademarks of XMOS Ltd in the United Kingdom and other countries, and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners. Where those designations appear in this document, and XMOS was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.