Take the guesswork out of timing in real-time software systems*

To guarantee the correct operation of real-time software running on an embedded processor is a significant challenge. Data-dependent control flow, where execution times of many functions are dependent on the data inputs, means that instruction sequences are difficult to time accurately. This problem is further complicated by the use of interrupts and complex memory hierarchies, which can make timing near-impossible to accurately predict.

To address this problem so far, the test bench has been relied upon to exercise the timing corner cases of each function. In complex systems, this often means developing much of the rest of the application before suitable stimulus can be created. This has added significant effort and delay to certifying software system timing.

New processor architectures with predictable timing enable new approaches to timing certification. This article describes a new technique, *static timing analysis*, which can be used to formally prove that real-time constraints in the source code are always met. In short, a static timing tool analyzes object code and determines worst-case timing paths. Paths are analyzed interactively or against assertions in the source code to produce a pass or fail result, allowing designers to refine timing-critical sections of their code until timing closure is achieved.

The article shows how this approach can be used to develop a software implementation of a 100Mbps Ethernet MII interface.

Introduction

As processor architectures improve in speed and responsiveness, it becomes increasingly attractive to perform real-time functions in software instead of hardware. An interface such as an IIC master that defines timing is simple to implement in software. In contrast, an IIC slave must respond to events within strict time limits and is therefore traditionally performed in hardware. Developing real-time interfaces in software represents a real-time programming challenge.

Verifying software functionality is a well-practiced task using software test benches. Closing timing may require a significant amount of additional effort. The standard approaches to closing timing are usually limited to testing in-circuit whilst observing pin activity, simulating the software using a cycle-accurate simulator or counting the instructions for the paths of interest to determine the execution time. All of these approaches share a common requirement: to provide suitable stimulus to fully exercise the software under test ensuring that corner cases are adequately covered. This increases the verification effort due to extending test benches to include timing. In many cases, obtaining suitable stimulus may not be possible until much later in the project when the rest of the system is available to generate it. Static timing analysis removes this dependency and allows timing closure of each function individually.

*Updated version of article *Taking the guesswork out of timing in real-time systems* by Peter Hedinger & Edward Clarke, Embedded.com 06/04/2009.

Predictable processor architecture

Accurate instruction timing predictions are a fundamental requirement for a static timing analysis tool. There are many reasons why a sequence of instructions may not always take the same time to execute.

Interrupts by nature alter the execution flow dramatically by forcing a change of the CPU context for an amount of time. Further, all RTOSs have critical sections of code where interrupts are disabled. This means only one task can truly be real-time in a single threaded system and this is likely to be the scheduler in systems utilizing an RTOS.

Architectures that include memory hierarchy and cache memory in particular are known to exhibit less predictable execution times. The previous contents of the cache very strongly influence timing unless cache lines can be locked.

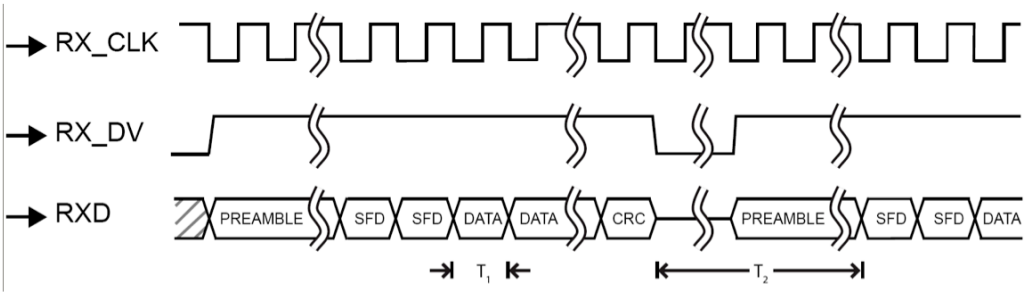
Resource conflict also can also increase execution time. Being denied a hardware resource will naturally block execution and adversely affect timing.

By avoiding these architectural issues, and employing simple round-robin scheduling, event driven processors provide highly predictable maximum execution timing. Further, the ability to pre-load output ports with a valid time allows execution to continue whilst the port autonomously handles the timed output.

Only data dependent execution flow may impede accurate timing prediction. A static timing tool can determine all possible paths through the software allowing for the variability due to data dependencies, providing guaranteed timing.

Case Study Introduction

The Ethernet MII interface is an example of where it is advantageous to replace hardware with a software implementation. Using software allows early adoption of new hardware standards and implementation of custom protocols. The interface between the MAC and PHY layers is shown below.



The source code to implement this interface is shown below. It is written in XC, which has support for direct control of physical pins through port inputs (`:>`) and outputs (`<:`). In addition, the `select` statement provides a way to respond to multiple events in a single thread concurrently and effectively provides a hardware switch statement. In this example, the pins are mapped to the ports `port_rxd` and `port_rx_dv`, and the data port `port_rxd` is configured to convert the stream of data nibbles from the PHY into a series of words.

Before the receiver starts, it ensures that the signal RX_DV is low. It then waits for the start of a packet, identified by the signal RX_DV going high and the Start Frame Delimited (SFD) being received. The inner loop waits for the availability of the next data word or the signal RX_DV going low. When RX_DV goes low, the program inputs any remaining data nibbles and checks for errors before starting to receive the next packet (not shown).

```
buffered in port:32 port_rxd = XS1_PORT_4A;
in port port_rx_dv = XS1_PORT_1I;
void mii_rx_pins() {
      port_rx_dv when pinseq(0) :> int lo;
      do {
            int eop = 0;
            ...

            #pragma xta label "wait_for_sfd"
            port_rx_dv when pinseq(1) :> int hi;
            port_rxd when pinseq(0xD) :> int sfd;
            ...
            do {
                  select {
                        #pragma xta label "word_receive"
                        case port_rxd :> word:
                              ...
                              break;

                        #pragma xta label "rx_dv_low"
                        case port_rx_dv when pinseq(0) :> int lo:
                              eop = 1;
                              ...
                              break;
                  }
            } while (!eop);
      } while (1);
}
```

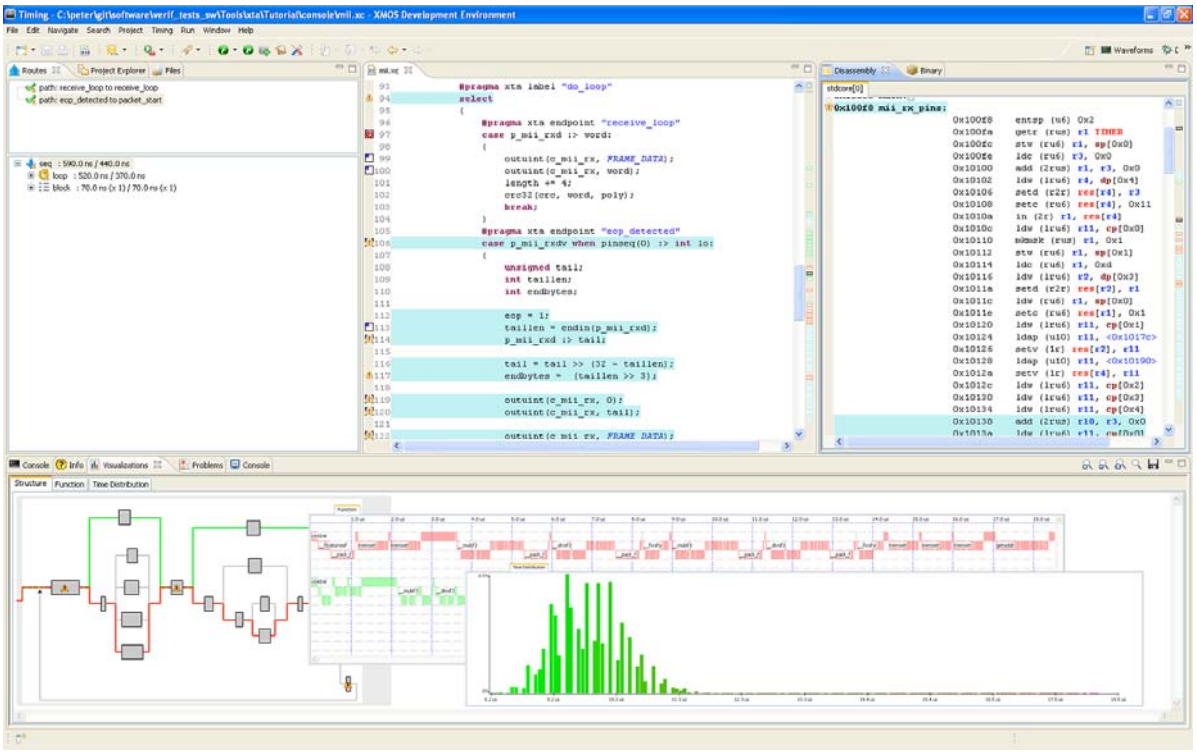
This program must run fast enough to meet the Ethernet MII timing specification. For 100Mbs Ethernet, a word of data is received every 320ns (T1), which means the innermost loop must be executed within this time. The inter-frame gap is a minimum of 96 bit times, and the preamble is 56 bit times. The time in which the inter-frame code must then complete (T2) is 1520ns.

The challenge is to guarantee that when the source code is compiled and run on a target device, it meets both of these timing constraints. There is only one valid path through the function for receiving data to consider, but the inter-frame code has a number of paths that corresponding to inputting end-packets with different sizes and handling error cases. Manually ensuring that all of these paths execute within the allowed time is time-consuming and prone to error, especially when the process must be repeated every time the code is modified.

Case Study Worked Through

Static timing analysis automates the task of verifying that a program meets its timing constraints. Given an architecture with predictable timing on which to run the program, it is possible for a tool to ensure that worst-case execution time is fast enough to meet the constraints. The tool operates on binary executable files to ensure the accuracy of the timing analysis. In addition it makes the tool language-independent, supporting code generated from any compiled source, be it assembly, C, C++ or XC. The user can use the GUI to view the code at source level.

The user starts by specifying a route between two endpoints in the code that needs to be executed within a specific time period. The MII code in the previous example specifies three endpoint labels as source-level pragmas. The timing tool then applies a combination of compiler, simulation and search techniques to ensure that all valid paths through the route are identified and accurately timed. Only paths explicitly excluded by the user (referred to as false paths) are ignored.



An interactive GUI or console can be used to visualize the execution paths through the program and to identify which of these are false paths. In the MII code, the first route that needs to be timed is from the endpoint label `word_receive` around the loop and back to itself. Two paths are found: one around the innermost loop and one that exits this loop and passes through the code waiting for the next SFD. The second route that needs to be timed is from the endpoint label `rx_dv_low` to the endpoint label `wait_for_sfd`. The tool finds all possible paths between these two endpoints, including false paths that pass through the word receive loop. Armed with

this information, the user can generate a script to run during compilation that checks the code meets the MII timing constraints.

For the MII code, the script looks like:

```
add exclusion wait_for_sfd
analyze path word_receive word_receive
set required – 320 ns
remove exclusion *
add exclusion word_receive
analyze path rx_dv_low wait_for_sfd
set required – 1520 ns
```

The assertions check the slack or violation for all possible paths of execution that have not been excluded and report the worst-case. In this example, the output is:

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

The tool can be used for more than just pass/fail testing. Structural code views can highlight timing hot-spots, and instruction-level views and traces can highlight hardware resource contention. This information helps the user focus on optimizing code where it has the greatest impact.

In some cases the user may have to provide additional information before the tool can time the code. Data-dependent execution flow, such as an unknown loop count, requires worst-case values in order to accurately time the code. These unknown values are highlighted to the user and can be specified through either the GUI or console.

Summary and further work

This article shows how a practical real-time software implementation can be proved to be timing-safe through the use of a static timing analysis tool. Whilst this is the main goal of such a tool, the technology opens up further possibilities.

XC includes direct support for timed input/output operations allowing the tool to identify these instructions and automatically generate the appropriate timing constraints. For example, the tool can check that the code between two timed inputs is executed fast enough.

The static timing tool identifies all paths through the code, including the worst-case. By analyzing the data that causes the worst-case timing, it is possible to reverse engineer the stimulus that caused this case. The worst-case test stimulus can then be added to the test bench used at a later stage, such as when testing in hardware. This gives the designer confidence that the test stimulus accurately covers all corner cases.

One final possibility is for power optimization. Power consumption is closely matched to instruction execution, particularly for event-driven processors that can enter a low power state when paused. Analysis and optimization of timing paths between pausing instructions allows the energy consumed per loop iteration to be determined and consequently reduced through optimization.

Designing real-time software capable of performing hardware functions is an attractive prospect, but until now the flow has been missing a tool to accurately certify timing. The introduction of static timing analysis makes it possible to guarantee the correct behavior of real-time software running on a processor with predictable instruction timing.

Static timing analysis also brings the advantage that timing closure of individual functions can be achieved well in advance of a full test bench or the rest of the system being available. By performing a formal analysis of the code, the static timing analyzer identifies all execution paths through the program, ensuring that no corner cases are missed. Adding timing assertions to the source code means that the source code not only describes the functionality, but also defines the required timing. The correct behavior of the code is validated for the target device at compile-time.

Static timing analysis takes the guesswork out of timing real-time software systems.