Time and Performance Measurement Instrumentation Functional Specification

Authors:
John Miller, Mercury Federal Systems, Inc.

Revision History

Revision	Description of Change	By	Date
1.01	Creation from previous internal documents	jmiller	2010-04-01

Table of Contents

1 Introduction

1.1 Purpose

The Time and Performance Measurement Instrumentation package (TPMI) is used in OpenCPI to profile system timing. This component is used to measure the time intervals of important signals emitted by software and is compatible with events and signals generated in the IP. TPMI also provides the capability to emit timing information for system resource's to provide capacity tracking along the event timeline.

TPMI is also responsible for managing the system time source. In order to view signals and events generated from different nodes and IP blocks in a single time synchronized view, all of the signals must be generated or calculated from the same time base. There are several supported time source options available. Note that time

1.2 Overview

TPMI provides a developer with a set of tools that capture signal timing information while introducing minimal overhead. Timing information is captured in the runtime code and is placed in one or more circular queues owned by the process. Signal information such as the name, type, width, and format attributes are captured at runtime through a registration API. The registered signal can then be used to emit time tagged information at runtime.

TPMI provides several layers of options to the developer to accommodate a wide range of time analysis requirements. If a user decides that retaining the class hierarchy is desirable then each of the hierarchy classes must inherit from the "Time::Emit" class. This allows the output formatter to traverse the classes and provide the hierarchical information in the output format. Alternatively a user may choose to emit timing information in a less intrusive manner by adding a single line of code to emit a time tagged signal.

Example:

```
CPI_EMIT("Start of DMA transfer")
```

In the example above, the single line macro performs two functions, signal registration and emit. There is a side effect of using the single line macro that may not be apparent, the registration occurs on first use. This causes a 1-time latency overhead the first time that the code is executed. If the first use overhead is not desirable another API is provided to allow the signal to be registered prior to use.

Example:

```
static CPI::Time::Emit::RegisterSignal ev0( "my signal" );
int main( int argc, char** argv )
{
      CPI::Time::Emit::getSTime::Emit().emit(ev0);
}
```

Once the timing information has been captured the developer can use one of the formatting options provided in the TPMI package. The formatting class retrieves the available process timing signals and streams the selected formatted output to the output stream provided by the user. The first format is named “CPIReadable” and provides the timing signal information in a user readable format that is easily parsed.

Example:

Signal Id	Relative time	Definition	Time class hierarchy	Value
4	0	"Object Exists	" Global:Instance:1::	0
1	5	"my signal	" Global:Instance:1::	0
1	7	"my signal	" Global:Instance:1::	89
5	36	"uint64	"QLengthTestClass:Instance:1::	0

The second format named “VCDFormat” provides an output that is compatible with the GTKWave viewer. This is the common format that is also used by most IP development tools to view signal information.

Example:

```
$date
         Wednesday, October 14 2009 14:17:42
$end
$version
            CPI VCD Software Signal Dumper V1.0
$end
$timescale
          1 us
$end
$scope module Software $end
$scope module Global:Instance:1:: $end
$var reg 1 !! my_signal $end
$var reg 64 #! plonglong_type $end
$var reg 1 $! Object_Exists $end
$var reg 1 '! Time::EmitTest_main.cxx_line_130 $end
$var reg 1 (! state_signal $end
$var reg 64 @! plonglong_type $end
$var reg 64 A! pulonglong_type $end
$var reg 64 B! pdouble_type $end
$var reg 1 H! Time::EmitTest_main.cxx_line_117 $end
$upscope $end
$scope module QLengthTestClass:Instance:1:: $end
$var reg 32 %" uint64 $end
```

```
$var reg 1 &" Object_Terminated $end
$upscope $end
$scope module QLengthTestClass:Instance:2:: $end
$var reg 1 $# Object_Exists $end
$var reg 32 %# uint64 $end
$upscope $end
$scope module SimpleSignalTestClass:Instance:1:: $end
$var reg 1 $$ Object_Exists $end
$var reg 1 &$ Object_Terminated $end
$var reg 1 ($ state_signal $end
$var reg 8 )$ uint8 $end
$var reg 16 *$ uint16 $end
$var reg 32 +$ uint32 $end
$var reg 1 -$ Time::EmitTest_main.cxx_line_157 $end
$var reg 1 .$ A_signal_1 $end
$upscope $end
$scope module PropertySignalTestClass:Instance:1:: $end
$var reg 1 $% Object_Exists $end
$var reg 1 &% Object_Terminated $end
$upscope $end
$scope module Class_A:Instance:1:: $end
$var reg 1 $& Object_Exists $end
$var reg 1 .& A_signal_1 $end
$scope module Class_B:Instance:1:: $end
$var reg 1 $( Object_Exists $end
$var reg 1 D( B_signal_1 $end
$var reg 1 E( B_signal_2 $end
$scope module Class_C:Instance:1:: $end
$var reg 1 $* Object_Exists $end
$var reg 32 %* uint64 $end
$var reg 1 &* Object_Terminated $end
$var reg 1 F* State_Signal1 $end
$var reg 1 G* State_Signal2 $end
$upscope $end
$upscope $end
$upscope $end
$scope module Class_A:A1_instance:2:: $end
$var reg 1 $' Object_Exists $end
$upscope $end
$scope module Time::EmitResource:Instance:1:: $end
$var reg 1 $) Object_Exists $end
$var reg 1 &) Object_Terminated $end
$var reg 32 C) Container_Memory $end
$upscope $end
$upscope $end
$enddefinitions $end
$dumpvars
0!!
0#!
0$&
0.&
0$(
0D(
0E(
0$*
0%*
0&*
0F*
0G*
0$'
0$)
0&)
0C)
$end

#0
1$!
#1
0$!

#5
1!!
#6
0!!

#7
1!!
#8
0!!
#9
```

```
b1111111111111111111111111111111111111111111111111111111111111101 #!
#967
b0000000000000000000000000000000000000000000000000000000000000001 %"
#969
b0000000000000000000000000000000000000000000000000000000000000010 %"
#971
b0000000000000000000000000000000000000000000000000000000000000011 %"
#973
b0000000000000000000000000000000000000000000000000000000000000100 %"
b0000000000000000000000000000000000000000000000000000000000000011 %#
#2811
b0000000000000000000000000000000000000000000000000000000000000100 %#
#47386
1-$
#47387
0-$
#124068
b0000000000000000000000000000000000000000000000000010010000000000 C)
#124076
1E(
#124077
0E(
#124082
b0000000000000000000000000000000000000000000000000001010001111000 C)
#124084
b0000000000000000000000000000000000000000000000000010100000000000 C)
#124086
1&)
#124087
0&)
#124126
1&*
#124127
0&*
#124134
1H!
#124135
0H!
$dumpoff
$end
```

A raw format is also available which is used as a common storage format that can later be converted for use in a viewer.

2 Signal types

The TPMI API allows the user to emit several types of signals; this section describes them in detail. Each signal has a width to indicate the number of bits that are required to capture the signal value range. Simple signals have a default width of 1.

2.1 Transient

A transient signal has a width of 1 and is simply a signal whose leading edge is tagged with a time value. This is most commonly used to indicate when a stateless signal occurs. When this signal is formatted with the VCD format option, it is described in the VCD file as a signal whose leading edge is tagged with the time value and has its falling edge occur on the next time tick.

2.2 State

A state signal has a width of 1 and allows the user to define the state value, either 0 or 1. So it is the user that controls the duration that this signal remains in the on or off state.

2.3 Value

A value signal can have a width from 1 to 64. This signal is used to emit a “value” with the selected capacity. This signal is typically used to indicate when a resource or property value has been modified. There are several macros that are provided to allow the user to emit values with a defined size and also to emit PValue properties. The PValue Macro registers the property using the property name and extracts the width information based on the property type. Each time that the property signal gets emitted, the property value is captured.

PValue Example:

```
PVLongLong plonglong("plonglong type", -3 );
CPI_EMIT_P( plonglong );
```

Unsigned 64 Example:

```
CPI_EMIT_UINT64( “My Resource”, 1024);
```

2.4 Code line and file Signal

This is a special signal that can be used to emit the name of the file and the line number where the signal was emitted.

Example:

```
CPI_EMIT_HERE;
```

3 Getting Started

3.1 Simple 1 line emit case with first use overhead

```
#include <CpiTimeEmit.h>

int main( int argc, char** argv)
{
     CPI_EMIT( “Entering main()”);
}
```

3.2 Simple 2 line emit case with no first use overhead

```
#include <CpiTimeEmit.h>

static CPI::Time::Emit::RegisterSignal ev0( "my signal" );

int main( int argc, char** argv)
{
      CPI::Time::Emit::getSTime::Emit().emit(ev0);
}
```

3.3 Emitting PValue property signals

```
#include <CpiTimeEmit.h>

int main( int argc, char** argv)
{
     PVLongLong plonglong("plonglong type", -3 );
    CPI_EMIT_P( plonglong );
}
```

3.4 Emitting sized value signals

```
#include <CpiTimeEmit.h>

int main( int argc, char** argv)
{
     CPI_EMIT_UINT64( “My Resource”, 1024);
}
```

3.5 Emitting state signals

```
#include <CpiTimeEmit.h>

int main( int argc, char** argv)
```

```
{
   CPI_EMIT_STATE( “Worker Enabled”, 1 );
   // User code
   CPI_EMIT_STATE( “Worker Enabled”, 0 );
}
```

3.6 Emitting file line number signals

```
#include <CpiTimeEmit.h>
int main( int argc, char** argv)
{
   CPI_EMIT_HERE;
   // User code
   CPI_EMIT_HERE;
}
```

3.7 Capturing class hierarchy information

All of the previous examples were simple case examples that do not capture class hierarchy information. This example demonstrates how to use the TPMI API in a class structure to capture class and instance information with each signal. This example shows how a parent/child relationship forms a runtime class hierarchy.

Note that when a class inherits from the Time::Emit class, a signal can be automatically generated when the class constructor and destructor get called. This behavior can be controlled with the “CPI_TIME_EMIT_TRACE_CD” environment variable.

```
#include <CpiTimeEmit.h>
using namespace CPI::Util;
class A : public Time::Emit {
public:
  A(const char* instance_name=0):
      Time::Emit( "Class A", instance_name){}
  void doStuff() {
    CPI_EMIT_( "A signal 1");
  }
};
class B : public Time::Emit {
public:
  B( Time::Emit* p ):Time::Emit( p, "Class B"){}
  void doStuff() {
    CPI_EMIT_( "B signal 1");
    CPI_EMIT_( "B signal 2");
  }
};
```

```
class C : public Time::Emit {
public:
  C( Time::Emit* p ):Time::Emit( p, "Class C"){}
  void setStates(int v) {
    switch ( v ) {
    case 1:
      CPI_EMIT_STATE_( "State Signal1", 1 );
      CPI_EMIT_STATE_( "State Signal2", 1 );
      break;
    case 2:
      CPI_EMIT_STATE_( "State Signal1", 0 );
      CPI_EMIT_UINT64_( "uint64", 56789 );
      break;
    case 3:
      CPI_EMIT_STATE_( "State Signal2", 0 );
      break;
    }
  }
};
class Resource : public Time::Emit {
public:
  Resource(const char* res_inst_name=0):Time::Emit("Resource",
res_inst_name),
    m_memory(1024*10)
  {
    CPI_EMIT_UINT32_( "Container Memory", m_memory );
  }
  void allocate( int amount ) {
    m_memory -= amount;
    CPI_EMIT_UINT32_( "Container Memory", m_memory );
  }
  void free( int amount ) {
    m_memory += amount;
    CPI_EMIT_UINT32_( "Container Memory", m_memory );
  }
private:
  CPI::OS::uint64_t m_memory;
};
int main( int argc, char** argv)
{
  // Time::Emit class, instance 1,
  A a;
  // Time::Emit class, instance 2, w/ instance def.
  A a1("A1 instance");
  // Runtime parentable time class
  B b( &a);
  {
    // Traceble Resource object
    Resource tr;

    // Generate signals
    a.doStuff();
    b.doStuff();

    // Now do some resource stuff
    tr.allocate( 1024 );
    a.doStuff();
    tr.allocate( 5000 );
    b.doStuff();
    tr.free( 1024);
    tr.free( 5000);

    // allow the resource object to be removed to capture that on the
trace
  }

{
    C c( &b );

    // inherited object using state trace
    c.setStates(1);
    a.doStuff();
    c.setStates(2);
    a.doStuff();
    c.setStates(3);

    // allow c to be removed to capture that on the trace
  }

  CPI_EMIT_HERE;
```

```
}
```

3.8 Emitting signals in “C” code

TPMI also provides “C” bindings to allow code written in C to emit signals. The C compatible header file named ”CpiTimeEmitC.h” contains the required prototypes and macros. A macro named CPI_EMIT_C is provided as a 1 line “emit” capability however it incurs the first use overhead. As an alternative the registration and emit functions can be called separately.

```
#include <CpiTimeEmitC.h>

int main( int argc, char** argv)
{
      int eid = CpiTimeARegsiter(“time sensitive signal”);

      /* user code */

     CpiEmit( eid );

    CPI_EMIT(“don’t care about first use overhead”);

}
```

4 Formatting the TPMI signals for output

Once an application has captured TPMI signals, they can be formatted and sent to an output stream for analysis. There are three formats that are currently supported, CPIRaw, CPIReadable and VCDFormat. The formatter class is instantiated with the selected format and then redirected to an output stream.

Example;

```
// VCD format
CPI::Time::Emit::Formatter
 tfa(CPI::Time::Emit::Formatter::VCDFormat);
  std::cout << tfa;

// Readable format
CPI::Time::Emit::Formatter
tfb(CPI::Time::Emit::Formatter::CPIReadable);
 std::cout << tfb;
```

5 System time source Options

5.1.1 Time Source 1: Linux absolute time

This time source uses the Linux “gettimeofday” function. In a multi-node system, NTP can be used to synchronize time. It is a portable “hardware independent” time source available on any Linux GPP. This is a time source for applications that do not require accurate time stamps. The inherit accuracy is limited to the accuracy of NTP and the significant and unpredictable overhead of the Linux system call which also affects the application timing at the emit call site.

5.1.2 Time Source 2: Best Available with no additional time base hardware

This time source uses the available resources on a GPP to achieve the best possible time. This approach utilizes the CPU free running clock in conjunction with the systems absolute time to generate time stamps. Absolute time is calculated from the free running CPU clock by associating it with Linux absolute time. Through integration we profile the clock access latencies and “dial” them out within the transfer function used to convert the free running time to absolute time. A servo is used to correct clock drift. The resolution of this approach is defined by the frequency of the free running clock. The inherit accuracy is limited to the accuracy of NTP, our ability to profile and dial out latency and the ability to compensate for clock drift.

5.1.3 Time Source 3: GPS time source

There are many GPS time sources available on the market. When a specific GPS time source is selected this section will be filled in.

6 Advanced features

6.1 Using trigger events

Event triggers allow the user to start collecting time data when a user event is detected. There are several event trigger roles that can be used all of which are described below. Each of these triggers affects the collection of signals by flushing the queue or queues and restarting data collection.

LocalQGroupTrigger allows any emitter with this role to trigger the collection of events in the local queue. The first event with this role starts the event collection and all others are ignored.

GlobalQGroupTrigger has the same effect as LocalQGroupTrigger except that it affects all of the time emit queues in the process.

LocalQMasterTrigger allows an emitter to start collecting time data in the local queue. If more than 1 emitter use this trigger, the last emitter with this role flushes the local queue and restarts event collection.

GlobalQMasterTrigger has the same affect as LocalQMasterTrigger except that it affects all of the time emit queues in the process.

6.2 Configuring the signal Queues

The signal queue configuration can be changed either programmatically or by setting environment variables. The QConfig structure can be used to pass configuration information to classes that inherit from Time::Emit. If this parameter is NULL in the constructor the inherited class uses the “global” signal Queue and does not create its own. However if a QConfig parameter is passed into the constructor, its values are uses to create an signal Q that can also be used by its children classes.

```
      struct QConfig {
            unsigned int qByteCount;
            bool         stopWhenFull;
      };
```

The ***qByteCount*** sets the size of the signal queue in bytes. The default value is 4096.

stopWhenFull is used to control the queue behavior. By default this value is false and causes the queue to free run by overwriting old signals with new signals once it becomes full. If this is set to true, signals are ignored once the queue is full.

Having the ability to configure and utilize multiple queues allows a user to customize signal collection within a system.

The environment variables can also be used to set the default behavior of the signal queues, refer to the “Compile and Runtime Options” section of this document for more information.

6.3 Customizing Signal Registration

Although there are several macros that are intended to simplify the process of registering and emitting signals, the actual API calls can be made directly by the application.

6.3.1 Signal Registration

The class “CPI::Time::Emit::RegisterSignal” is used to register signals. The constructor for this class takes four parameters.

```
RegisterSignal( const char* signal_name,
                int width=1,
                SignalType type=CPI::Time::Emit::Transient,
                DataType dt=CPI::Time::Emit::u
               );
```

Where:

- **signal_name** is the name of the registered signal.
- **width** is the number of bits that will be needed to hold the signals value range.
- **type** is the signal type described earlier in this document.
- **dt** is the “DataType” that is only used by the formatter to interpret the value for output.

Example:

This example registers a “value” signal named “my signal” with a width of 8 bits whose value when formatted will be interpreted as an integer.

```
static CPI::Time::Emit::RegisterSignal ev1(
   "my signal", 8,
   CPI::Time::Emit::Value,
   CPI::Time::Emit::i
 );
```

6.3.2 Emit Signal

Once registered the allocated id can be used to emit the signal with no runtime registration overhead.

```
CPI::Time::Emit::getSTime::Emit().emit(ev1);
```

Note the RegisterSignal class has an SignalId conversion operator that allows it to be used directly in the emit method.

7 Compile and Runtime Options

TPMI can be configured at compile time as well as run time to suit the requirements of the user. Compile time options can be added to the users makefile during the build process and the runtime options are pulled from the environment.

7.1 Compile Time Options

CPI_TIME_EMIT_SUPPORT

If defined this option turns on the emit macros in the user code. If this option is not defined the macros are effectively turned off and compiled out of the code.

CPI_TIME_EMIT_MULTI_THREADED

If defined this option allows the time emit macros to be safely used in a multi-threaded environment/

7.2 Run Time Options

CPI_TIME_EMIT_Q_SIZE

This environment variable allows the user to override the default size of the time emit queues. The size is defined in bytes.

CPI_TIME_EMIT_Q_SWF

If this environment variable is set to “true” it caused the time emit module to stop collecting time signals when the queue is full. The default behavior is to continue collecting signals replacing older signals in the queue.

CPI_TIME_EMIT_TRACE_CD

This switch allows the user to emit signals when a time emit super class enters its constructor or destructor. Setting its value to “true” will cause the signals to be recorded.

CPI_TIME_EMIT_DUMP_ON_EXIT

When set to “true” this environment variable causes the time emit module to dump all of the process signal queues to disk when the process exits.

CPI_TIME_EMIT_DUMP_FILENAME

Defines the filename that is used when the time emit module dumps the signal queues to disk on exit.

CPI_TIME_EMIT_DUMP_FORMAT

Defined the signal dump format. The following are valid selections:

- "RAW”
- "READABLE"
- "VCD"

8 Requirements on which TPMI is based:

- Shall introduce minimal runtime overhead
- Shall provide a “C” binder
- Shall provide a “1 line” API to define and generate a time signal
- Shall be capable of providing signal descriptions in the 1 line API so that post processing is not required. (i.e. captured signal information is self contained and complete)
- Shall provide the ability to record software signals
  - *Shall be capable of recording transient signals*
  - *Shall be capable of recording state signals*
  - *Shall be capable of recording signals with value of specified data widths*
- Shall be capable of recording signals from 1 or more defined trigger signals
- Shall be capable of turning the signal recording on or off from a compile time switch
- Shall provide signal recording buffer configuration
  - *Shall be capable of allowing the user to control the signal buffer size*
  - *Shall be capable of allowing the user to control the signal buffer full behavior, either stop when full or overwrite buffer.*
- Shall support signal hierarchy
- Shall provide a post signal collection formatting option that supports visualization in GTKWave
- Shall utilize a time stamp mechanism
- Time stamp shall support the highest resolution available on the system
- Time skew from compute node to compute node shall not exceed XX nS
- Time skew from compute node to FPGA subsystem shall not exceed XX nS