Introduction to MatchLib Connections Library

The MatchLib Connections library is part of the open source Matchlib library, which was originally developed by NVidia and is available here: <https://github.com/NVlabs/matchlib>

The Connections Library provides the basic transaction modeling and communication mechanisms that are used in the Matchlib library and in models that use Matchlib.

Some of the key features of the Connections library are:

- Models are synthesizeable using HLS
- Throughput accurate modeling is enabled in SystemC prior to HLS
- Features are provided to support verification and debug in SystemC prior to HLS

Include files

The following header files should be included in all model files using the Connections library:

```
#include <nvhls_connections.h>
#include <mc_connections.h>
#include <mc_trace.h>
```

Compiler Directives

It is recommended that you use the following g++ compiler flags when compiling models using the Connections library:

-std=c++11 -DHLS_CATAPULT -DSC_INCLUDE_DYNAMIC_PROCESSES -DCONNECTIONS_ACCURATE_SIM

The Catapult compiler flags should include the same flags, however the C++ 2011 standard is instead set using the following option in your Catapult directives file:

options set Input/CppStandard c++11

Connections Library Reference Documentation

This section documents the recommended (and Mentor-supported) public APIs from the Connections library. Note that there are additional APIs in Connections library that are not yet actively supported by Mentor.

Out

```
namespace Connections {
   template <typename Message> class Out {
   public:
   Out(const char* name);
   void Reset ();
```

```
   void Push(const Message& m);
   bool PushNB(const Message& m);
 };
};
```

The Out port is used to output transactions from a module. The Push method is blocking and will not complete until the transaction has been transmitted successfully. The PushNB is non-blocking and will return immediately with a bool indicating if the transaction was transmitted successfully on that call. The Reset method must be called in the reset state of the process that calls Push or PushNB.

In

```
namespace Connections {
   template <typename Message> class In {
   public:
   In(const char* name);
   void Reset ();
   Message Pop();
   bool PopNB(Message& m);
 };
};
```

The In port is used to input transactions into a module. The Pop method is blocking and will not complete until the transaction has been received successfully. The PopNB is non-blocking and will return immediately with a bool indicating if the transaction was received successfully on that call. The Reset method must be called in the reset state of the process that calls Pop or PopNB.

Example

```
#include <nvhls_connections.h>
#include <mc_trace.h>

#pragma hls_design top
class dut : public sc_module {
public:
  sc_in<bool> INIT_S1(clk);
  sc_in<bool> INIT_S1(rst_bar);

  Connections::Out<NVUINTW(32)> INIT_S1(out1);
  Connections::In <NVUINTW(32)> INIT_S1(in1);

  SC_CTOR(dut)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
    async_reset_signal_is(rst_bar, false);
  }

private:
```

```
  void main() {
    out1.Reset();
    in1.Reset();

    wait();

    #pragma hls_pipeline_init_interval 1
    #pragma pipeline_stall_mode flush
    while(1) {
      uint32_t t = in1.Pop();
      out1.Push(t + 0x100);
    }
  }
};
```

Combinational

```
namespace Connections {
   template <typename Message> class Combinational {
   public:
   Combinational(const char* name);
   void ResetRead();
   void ResetWrite();
   Message Pop();
   bool PopNB(Message& data);
   void Push(const Message& m);
   bool PushNB(const Message& m);
 };
};
```

The Combinational class is the most common channel used for transaction communication. It synthesizes to rdy, vld, and msg wires, where the “msg” carries the transaction data. The methods available in the Combinational class are the same as those available in the In<> and Out<> ports. Normally, if you are using ports, you should not directly call these methods within the Combinational class. However, if you have two SystemC processes within a module that need to communicate via Combinational channel, then those processes can directly call these methods without needing to use ports. Note that in this case the process calling “Pop” methods will need to call ResetRead in its reset state, while the process calling “Push” methods will need to call ResetWrite in its reset state.

set_sim_clk

```
namespace Connections {
   void set_sim_clk(sc_clock* clk_ptr);
};
```

The set_sim_clk function should be called in your sc_main function before any other Connections APIs are used. It has a single argument which is a pointer to the global clock that is used for all Matchlib components. (Currently Matchlib only supports a single global clock). If the set_sim_clk function is not called in your models, then it defaults to a 1 ns clock.

Example

```
#include <mc_scverify.h>
#include <mc_trace.h>
#include "dut.h"
#define NVHLS_VERIFY_BLOCKS (dut)
#include <nvhls_verify.h>


class Top : public sc_module {
public:
  NVHLS_DESIGN(dut) INIT_S1(dut1);

  sc_clock clk;
  SC_SIG(bool, rst_bar);

  Connections::Combinational<NVUINTW(32)>        INIT_S1(out1);
  Connections::Combinational<NVUINTW(32)>        INIT_S1(in1);

  SC_CTOR(Top)
     :   clk("clk", 1, SC_NS, 0.5,0,SC_NS,true)
  {
    Connections::set_sim_clk(&clk);
    sc_object_tracer<sc_clock> trace_clk(clk);

    dut1.clk(clk);
    dut1.rst_bar(rst_bar);
    dut1.out1(out1);
    dut1.in1(in1);

    SC_CTHREAD(reset, clk);

    SC_THREAD(stim);
    sensitive << clk.posedge_event();
    async_reset_signal_is(rst_bar, false);

    SC_THREAD(resp);
    sensitive << clk.posedge_event();
    async_reset_signal_is(rst_bar, false);
  }


  void stim() {
    LOG("Stimulus started");
    in1.ResetWrite();
    wait();

    for (int i = 0; i < 10; i++)
      in1.Push(i);

    sc_stop();
    wait();
  }
```

```
   void resp() {
    out1.ResetRead();
    wait();

    while (1)
      LOG("TB resp sees: " << std::hex << out1.Pop());
  }

  void reset() {
    rst_bar.write(0);
    wait(5);
    rst_bar.write(1);
    wait();
  }
};


sc_trace_file* trace_file_ptr;

int sc_main(int argc, char** argv) {
  trace_file_ptr = sc_create_vcd_trace_file("trace");

  Top top("top");
  trace_hierarchy(&top, trace_file_ptr);
  sc_start();
  return 0;
}
```

Fifo

```
namespace Connections {
  template <typename Message, unsigned int NumEntries>
  class Fifo: public sc_module {
  public:
    sc_in<bool> clk;
    sc_in<bool> rst;
    Connections::In<T>  enq;
    Connections::Out<T> deq;

    SC_CTOR(Fifo);
 };
}
```

Connections::Fifo implements a FIFO in the Connections library where the type of data stored in the FIFO is specified by the template parameter Message while the size of the FIFO is specified by the template parameter NumEntries. The Connections::Fifo has four ports – clk and rst boolean input ports, and enq and deq Connections ports.
Internal processes use async_reset_signal_is(rst, false) by default.
Define CONNECTIONS_SYNC_RESET for sync reset.
Define CONNECTIONS_POS_RESET for active high reset.

The origin of the Connections::Fifo is the Connections::Buffer class provided with Nvidia's Matchlib library.

Example

```
#pragma once
#include <mc_connections.h>
typedef sc_uint<32> T;

SC_MODULE(dut) {
public:
  sc_in<bool> CCS_INIT_S1(clk);
  sc_in<bool> CCS_INIT_S1(rst_bar);
  Connections::In<T>  CCS_INIT_S1(in);
  Connections::Out<T> CCS_INIT_S1(out);

  Connections::Fifo<T, 8> CCS_INIT_S1(fifo);
  Connections::Combinational<T> CCS_INIT_S1(fifo_in);
  Connections::Combinational<T> CCS_INIT_S1(fifo_out);

  SC_CTOR(dut) {
    fifo.clk(clk);
    fifo.rst(rst_bar);
    fifo.enq(fifo_in);
    fifo.deq(fifo_out);

    SC_THREAD(in_stream);
    sensitive << clk.pos();
    async_reset_signal_is(rst_bar, false);

    SC_THREAD(out_stream);
    sensitive << clk.pos();
    async_reset_signal_is(rst_bar, false);
  }

  void in_stream() {
    fifo_in.ResetWrite();
    in.Reset();
    wait();
    while (1) { fifo_in.Push(in.Pop()); }
  }
  void out_stream() {
    fifo_out.ResetRead();
    out.Reset();
    wait();
    while (1) { out.Push(fifo_out.Pop()); }
  }
};
```

SyncIn, SyncOut, SyncChannel

```
class SyncIn: public sc_port {
public:
  SC_CTOR(SyncIn);
  void sync_in();
  void reset_sync_in();
};

class SyncOut: public sc_port {
public:
  SC_CTOR(SyncIn);
  void sync_out();
  void reset_sync_out();
};

class SyncChannel: public sc_module {
public:
  SC_CTOR(SyncChannel);
  void sync_out();
  void reset_sync_out();
  void sync_in();
  void reset_sync_in();
};
```

These classes are used to implement a two wire synchronization handshake between two different processes. The SyncChannel synthesizes to vld and rdy signals. The sync_out() method drives the vld signal, and the sync_in() method drives the rdy signal. Both of these methods will block until the handshake completes (ie both vld and rdy are true), and then both will return. The reset methods need to be called in the reset states of their respective processes. The methods within SyncChannel should only be used if ports are not being used, for example if you have two processes in the same module that need to be synchronized using SyncChannel.

Required Methods for User-Defined Transaction Classes

The Matchlib Connections library automatically supports the following features on all transactions used with Connections:

- Ability to trace transactions in a SystemC simulation to a ".vcd" file for viewing in a waveform viewer.
- Ability to log transactions in a SystemC simulation to a text log file.
- Ability to convert (or "Marshall") a transaction to/from a bitstream.

All of the C++ built in datatypes (such as int, uint32, etc) as well as the AC datatypes (such as ac_int) and SC datatypes (such as sc_uint), as well as transaction types provided in the Matchlib library such as the AXI4 tranaction types have built-in support to enable the above features to work with the Connections library. However, if you define your own transaction types in your models, there are a few member functions you need to define to enable the above features to work. Below we show an example of a

user-defined transaction type to model a car engine, and the methods that are added to support these capabilities.

```
struct engine_t
{
  static const int plugs = 4;
  sc_uint<16> engine;
  spark_plug_t spark_plugs[plugs];

  static const unsigned int width = 16 + (spark_plug_t::width * plugs);
  template <unsigned int Size> void Marshall(Marshaller<Size> &m) {
    m &engine;
    for (int i=0; i<plugs; i++)
      m &spark_plugs[i];
  }
  inline friend void sc_trace(sc_trace_file *tf, const engine_t& v, const
std::string& NAME ) {
      sc_trace(tf,v.engine,  NAME + ".engine");
      for (int i=0; i<plugs; i++)
        sc_trace(tf,v.spark_plugs[i],  NAME + ".spark_plug" +
std::to_string(i));
  }

  inline friend std::ostream& operator<<(ostream& os, const engine_t& rhs)
  {
    os << rhs.engine << " ";
    for (int i=0; i<plugs; i++)
      os << rhs.spark_plugs[i] << " ";
    return os;
  }
};
```

The first item you need to provide is “static const unsigned int width”. The value of this must be the bitwidth of the transaction. This bitwidth represents the exact bitwidth of the transaction that will be used during synthesis, and it is not (necessarily) the same as the bitwidth that the C++ compiler may use to represent the transaction during SystemC simulation.

The second item you need is the “Marshall” method. This method is used to convert the transaction to/from a bitstream. The bitstream is specified via the Marshaller argument, in this example it is named “m”. Each field in the transaction is streamed to/from the Marshaller object using the “&” operator, which is overloaded to perform the conversion operation.

The third item you need is the “sc_trace” method, which is the SystemC standard tracing method (and is documented in the SystemC LRM). This method enables the transaction to be traced into a “.vcd” file during SystemC simulation. Each field of the transaction is included in the trace file by calling sc_trace on them within the implementation of this method.

The fourth item is the output streaming operator “<<”. This method prints the transaction contents to a text file. It is recommended that you not include any newlines when printing the transaction, but that you do separate subfields with a space.

Connections Library Tracing and Logging Features

You can enable tracing and logging in your sc_main function, as shown in the example below.

```
int sc_main(int argc, char** argv) {
  sc_trace_file* trace_file_ptr = sc_create_vcd_trace_file("trace");

  Top top("top");
  trace_hierarchy(&top, trace_file_ptr);

  channel_logs logs;
  logs.enable("log_dir");
  logs.log_hierarchy(top);

  sc_start();
  return 0;
}
```

The trace_hierarchy call must be made after the top level module has been instantiated. If desired, you can trace only specific subhierarchies, and you can call the trace_hierarchy function multiple times to specify exactly what should be traced.

Channel logs are enabled by first instantiating a channel_logs object within your sc_main function. The “enable” method specifies the directory to store the logs, and the “log_hierarchy” call specifies a hierarchy to be logged. The destructor for the “logs” object causes the log files to be fully written out and closed at the end of simulation.