Matchlib Reference Manual

23 August 2024

Contents

The MatchLib Connections Library

The MatchLib Connections library is part of the open source Matchlib library, which was originally developed by Nvidia Research.

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 synthesizable 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

The Matchlib Connections library is built on top of SystemC, and supports standard SystemC modeling constructs such as modules, ports, clocks, resets, signals, etc. The Connections library automatically detects all SystemC clocks (sc_clock objects) used in your design. Designs with multiple clocks and multiple clock frequencies are supported.

Include files

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

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

Compiler Directives

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

-std=c++11 -DSC_INCLUDE_DYNAMIC_PROCESSES

When SystemC models are used within Catapult HLS these compiler flags are automatically set.

Terminology

This documentation uses standard SystemC terminology, summarized here:

- Port (sc_port) : a port is a communication interface on a module that can be bound to a channel.
- Module (sc_module): a module is a component in a design hierarchy, like a Verilog module.
- Channel (sc_channel) : a channel is a communication pathway between two modules or processes.
- Process: A process is declared using SC_THREAD or SC_METHOD in SystemC and represents an independent thread of control within the SystemC model.
- Message Passing Interface: A message passing interface reliably delivers messages (or transactions) from one process to another. This document uses this term to denote the type of

communication found in Kahn Process Networks. See https://en.wikipedia.org/wiki/Kahn_process_networks

- Synchronization Interface: A synchronization interface synchronizes one process with another and/or with a global clock. For an example of a synchronization interface, see https://en.wikipedia.org/wiki/Barrier_(computer_science)
- Signal IO: In digital HW design, signals are the fundamental communication mechanism. Signals enable communication between two HW blocks/processes, but communication with signals in real HW always incurs at least some delay because communication cannot be faster than the speed of light. In HDLs and in SystemC, signal delays are modeled with the "delayed update" semantics.
- “blocking” / “non-blocking”: A blocking message passing interface suspends the execution of the calling process until the message is either sent or received. A non-blocking message passing interface never suspends execution of the calling process: instead, a return code is provided to indicate whether the operation completed or not.

Connections::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.

An example is provided on the next page.

Connections::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 <mc_connections.h>

#pragma hls_design top
class dut : public sc_module {
public:
  sc_in<bool> CCS_INIT_S1(clk);
  sc_in<bool> CCS_INIT_S1(rst_bar);

  Connections::Out<uint32_t> CCS_INIT_S1(out1);
  Connections::In <uint32_t> CCS_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);
    }
  }
};
```

Connections::Combinational<>

```
namespace Connections {
   template <typename Message> class Combinational {
   public:
   Combinational(sc_module_name 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 dat wires, where the "dat" 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.

Example

```
#include <mc_scverify.h>
#include "dut.h"


class Top : public sc_module {
public:
  CCS_DESIGN(dut) CCS_INIT_S1(dut1);

  sc_clock clk;
  SC_SIG(bool, rst_bar);

  Connections::Combinational<uint32>        CCS_INIT_S1(out1);
  Connections::Combinational<uint32>        CCS_INIT_S1(in1);

  SC_CTOR(Top)
     :   clk("clk", 1, SC_NS, 0.5,0,SC_NS,true)
  {
    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() {
    CCS_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)
      CCS_LOG("TB resp sees: " << std::hex << out1.Pop());
  }

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


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

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

Connections::Fifo<>

```
template <class T, int N>
class Fifo: public sc_module {
public:
  sc_in<bool> CCS_INIT_S1(clk);
  sc_in<bool> CCS_INIT_S1(rst);

  Connections::In <T> CCS_INIT_S1(enq);
  Connections::Out<T> CCS_INIT_S1(deq);

  Fifo(sc_module_name name);
 };
```

Connections::Fifo<> implements a FIFO using the Connections library where the type of data stored in the FIFO is specified by the template parameter T while the size of the FIFO is specified by the template parameter N. The Fifo has four ports – clk and rst boolean input ports, and enq and deq Connections ports. The Fifo reads input transactions from the enq port and writes transactions to the deq port.

Connections::SyncIn<>, Connections::SyncOut<>, Connections::SyncChannel<>

```
namespace Connections {

class SyncIn: public sc_port {
public:
  SyncIn(const char* name);
  void sync_in();
  void reset_sync_in();
};

class SyncOut: public sc_port {
public:
  SyncOut(const char* name);
  void sync_out();
  void reset_sync_out();
};

class SyncChannel: public sc_channel {
public:
  SyncChannel(sc_module_name name);
  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 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.

The SyncChannel is what is called a “barrier” in the SW domain, see:

https://en.wikipedia.org/wiki/Barrier_(computer_science)

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 transaction types have built-in support to enable the above features to work with the Connections library. However, if you define your own transaction classes in your models, there are a few member functions you need to define to enable the above features to work.

The standard SystemC methods that are required for transaction classes are:

- `inline friend void sc_trace(sc_trace_file *tf, const this_type &v, const std::string &NAME );`
  - This function is needed for SystemC standard waveform tracing support.
- `inline friend std::ostream &operator<<(ostream &os, const this_type &rhs);`
  - This function is needed for SystemC standard transaction streaming/printing support.
- `bool operator==(const this_type & rhs) const ;`
  - This function is needed for transactions that are used with SystemC sc_signal<T>

The additional methods and constants that are needed if a transaction is using the Matchlib Connections Marshaller capability are:

- `static const unsigned int width = ... ;`

- This constant is needed for the Matchlib Connections Marshaller. It specifies the overall bitwidth of the type.

- `template <unsigned int Size> void Marshall(Marshaller<Size>& m);`

- This function is needed for the Matchlib Connections Marshaller. This function packs or unpacks the type to or from a bitstream.

It is not recommended to write any of these methods explicitly. Instead, a macro is provided that will automatically create these methods for your transaction classes. Here is an example:

```
#include "auto_gen_fields.h"
#include "ac_enum.h"

enum ColorBase { red, green, blue};  // C-style enum

typedef ac_enum<ColorBase, 2> Color_t;

template <int X, int Y>
struct sub_trans_t {
  uint16_t field1;
```

```
  ac_int<8, false> field2;
  Color_t enum3;
  uint16_t field4[X][Y];

  AUTO_GEN_FIELD_METHODS(sub_trans_t, ( \
     field1 \
   , field2 \
   , enum3 \
   , field4 \
  ) )
  //
};
```

The AUTO_GEN_FIELD_METHODS macro automatically generates all five transaction methods described above. It takes as its first argument the name of the transaction class. The second argument is a comma separated list of all the transaction fields. Note that there are two parentheses at the beginning and end of the macro.

If your transaction class does not need the Matchlib Marshaller capability, then you can use the AUTO_GEN_FIELD_METHOD_V2 macro instead. This macro has the same arguments, and it will not generate the two methods related to the Marshaller.

If your transaction class has fields which are themselves user-defined transaction classes, then each of these classes also needs to use these macros as shown above.

Connections Library Tracing and Logging Features

You can enable tracing and/or 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_trace_static::setup_trace_file("trace");

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

  Connections::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. An optional "true" second argument to the enable method will cause the output logs to use unbuffered IO. This may slow down generation of the log files somewhat but can be useful in debugging cases where the simulation may be hanging or crashing. The "log_hierarchy" call specifies a hierarchy to be logged. The destructor for the channel_logs object causes the log files to be fully written out and closed at the end of simulation.

Connections Channel Random Stall Injection

During pre-HLS simulation, you can enable random stall injection to stress test your design. When this feature is enabled, the rdy signal within all Connections::Combinational<> channels within both the design and the testbench will be forced low for randomly varying amounts of time. This stresses the system, ensuring that synchronization throughout the system is properly implemented.

The random stall injection feature is enabled by enabling the following compile flag during pre-HLS compilation:

```
-DCONN_RAND_STALL
```

See example 60* within the Matchlib examples kit for example usage.

Connections Fast Simulation Mode

For higher simulation performance during pre-HLS simulation, you can enable the Matchlib Connections library fast simulation mode. This mode is enabled with the following compile flag during pre-HLS compilation:

```
-DCONNECTIONS_FAST_SIM
```

When this mode is enabled, all Connections::Combinational<> channels are simulated with a TLM fifo rather than using the three rdy/vld/dat signals. The TLM fifo is an event driven TLM model with a default storage capacity of 16 items. The effect of using the TLM fifo is that in general each process can make more progress each time they run, resulting in less context switching within the overall simulation, thus resulting in higher simulation performance. When this option is used, the performance accuracy compared to the actual RTL simulation is greatly reduced, since multiple transactions over the same channel can be sent or received in a single clock cycle.

See example 20* within the Matchlib examples kit for example usage.

Connections Channel Latency and Capacity Back Annotation

The Connections library allows you to back annotate latency and capacity values for Connections::Combinational<> channels within your design and testbench. This feature can enable more accurate pre-HLS simulation. The feature is enabled with:

```
#ifndef __SYNTHESIS__
#include <connections/annotate.h>
#endif

int sc_main(int argc, char **argv)
{

  Top top("top");
#ifndef __SYNTHESIS__
  Connections::annotate_design(top);
#endif
  sc_start();
  return 0;
}
```

For further information, refer to Matchlib toolkit example 72*.

Connections Reset Signal Definition

Within the Connections library header files, Connections modules use asynchronous, active-low reset by default. If you want modules within the Connections header files to use a synchronous, active-low reset instead, define CONNECTIONS_SYNC_RESET within your model prior to the #include statement for the Connections header files.

Within your own models, the macros CONNECTIONS_NEG_RESET_SIGNAL_IS() and CONNECTIONS_POS_RESET_SIGNAL_IS() can be used in place of SystemC's reset_signal_is() and async_reset_signal_is() so that the macro CONNECTIONS_SYNC_RESET can select the appropriate type.

Accessing Connections rdy/vld/dat Signals Directly

In some cases, you may need to access the rdy/vld/dat signals within Connections channels or ports directly within your models. Typically, this is only necessary when you are writing low-level code which is effectively "RTL in SystemC". The Connections library supports this with the disable_spawn() method, shown in the example below. This method must be called in the module constructor for any Connections::In<> or Connections::Out<> ports for which direct access to the rdy/vld/dat signals is needed. When this feature is used, you should not use any of the Reset(), Push(), or Pop() methods on the port. Instead, all access should use the rdy/vld/dat signals.

An example of the disable_spawn() method is provided in Matchlib example 64*, also shown below.

```
class arbiter_comb : public sc_module {
public:
  Connections::In <packet> CCS_INIT_S1(in1);
  Connections::In <packet> CCS_INIT_S1(in2);
  Connections::Out<packet> CCS_INIT_S1(out1);

  sc_signal<ac_int<2, false>> CCS_INIT_S1(winner);

  SC_CTOR(arbiter_comb) {
    SC_METHOD(out1_vld_method);
    dont_initialize();
    sensitive << in1.vld << in2.vld;

    SC_METHOD(winner_method);
    dont_initialize();
    sensitive << in1.vld << in2.vld << in1.dat << in2.dat;

    SC_METHOD(in1_rdy_method);
    dont_initialize();
    sensitive << winner << out1.rdy;

    SC_METHOD(in2_rdy_method);
    dont_initialize();
    sensitive << winner << out1.rdy;

#ifdef CONNECTIONS_SIM_ONLY
    in1.disable_spawn();
    in2.disable_spawn();
    out1.disable_spawn();
#endif
  }

  void in1_rdy_method() {
    in1.rdy = (winner.read() == 1) && out1.rdy;
  }

  void in2_rdy_method() {
    in2.rdy = (winner.read() == 2) && out1.rdy;
  }
```

Connections Utility Macros – CCS_INIT_S1, CCS_LOG

The CCS_INIT_S1 macro can be used to automatically provide a string argument to the constructor for objects which require such an argument. Examples of such objects are sc_modules, sc_signals, sc_ports, etc.

Example:

```
Connections::Out<uint32_t> CCS_INIT_S1(out1);
Connections::In <uint32_t> CCS_INIT_S1(in1);
```

The CCS_LOG macro can be used to print text output to std::cout, along with the hierarchical path to the module instance and the SystemC current simulation time. This macro can only be used within sc_modules since it needs to be able to determine the hierarchical path.

Example:

```
while (1) {
  uint32_t t = in1.Pop();
  CCS_LOG("dut input is: " << t);
  out1.Push(t + 0x100);
}
```

This will output:

```
   7 ns top.dut1 dut input is: 0
   8 ns top.dut1 dut input is: 1
   9 ns top.dut1 dut input is: 2
```

Wrapper Generation for SystemC DUT insertion into SystemVerilog Testbench

Matchlib models can automatically generate a wrapper to enable the pre-HLS SystemC models to be instantiated in a SystemVerilog testbench. This facilitates early verification and stress testing of the SystemC DUT and enables the same SystemVerilog testbench to be reused on the post-HLS RTL.

This feature is demonstrated in examples 08* and 45* in the Matchlib examples kit, also shown below.

First the DUT needs to use the AUTO_GEN_PORT_INFO macro to list the ports:

```
#include "auto_gen_port_info.h"

class dma : public sc_module, public local_axi
{
public:
  sc_in<bool> CCS_INIT_S1(clk);
  sc_in<bool> CCS_INIT_S1(rst_bar);

  r_master<> CCS_INIT_S1(r_master0);
  w_master<> CCS_INIT_S1(w_master0);
  r_slave<>  CCS_INIT_S1(r_slave0);
  w_slave<>  CCS_INIT_S1(w_slave0);
  Connections::Out<bool> CCS_INIT_S1(dma_done);
  Connections::Out<sc_uint<32>> CCS_INIT_S1(dma_dbg);

  AUTO_GEN_PORT_INFO(dma, ( \
    clk \
  , rst_bar \
  , r_master0 \
  , w_master0 \
  , r_slave0 \
  , w_slave0 \
  , dma_done \
  , dma_dbg \
  ) )
  //
```

Next, the SystemC testbench constructor needs to invoke a call to cause the wrapper to be generated:

```
  SC_CTOR(Top)
    :   clk("clk", 1, SC_NS, 0.5,0,SC_NS,true) {

#ifndef CCS_SYSC
    auto_gen_wrapper dma_wrap("dma");
    dma1.gen_port_info_vec(dma_wrap.port_info_vec);
    dma_wrap.gen_wrappers(10, true);
#endif
```

This call will cause four files to be generated:

```
Generating dma_wrap.cpp
Generating dma_wrap.h
Generating dma_wrap.v
Generating dma_wrap.sv
```

These four files can be used to easily instantiate the SystemC DUT into a SystemVerilog testbench as shown in example 45*.

Declaring Clock Aliases

If your design uses a clock that is not explicitly declared somewhere in your testbench or DUT as an sc_clock, and if the clock is used by processes that also use Connections ports, then the clock declaration needs to be made available to the Connections library. The add_clock_alias() function is used to do this. The following code from Matchlib toolkit example 45*/dma_wrap.h shows its usage.

```
sc_clock connections_clk;

virtual void start_of_simulation() {
  Connections::get_sim_clk().add_clock_alias(
    connections_clk.posedge_event(), clk.posedge_event());
}

SC_CTOR(dma_wrap)
: connections_clk("connections_clk", 10, SC_NS, 0.5,0,SC_NS,true)
{ }
```

In this example the clk is an sc_in<bool> that is the actual clock signal that is generated in the SystemVerilog testbench and passed into the DUT. The Connections library needs to know the clock frequency, so an alias clock is declared named connections_clk. This clock has the same frequency as the clk signal in the SystemVerilog testbench. The add_clock_alias() call is performed once at the start of the simulation to inform the Connections library about the alias.

Matchlib Memory Model Classes

Introduction

In pre-HLS models, memories appear as normal C/C++ arrays, and are often mapped to RAMs during HLS. To meet performance and area goals, RAMs and their associated logic must often be carefully constructed. For example, to meet design throughput requirements, RAMs may often need to have multiple banks so that multiple accesses to the banks can occur concurrently.

This section describes the classes that are used to model RAMs and ROMs within Matchlib models. For a detailed discussion of the Matchlib memory modeling methodology, see:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/blob/master/matchlib_examples/doc/matchlib_memory_modeling_methodology.pdf

ac_array_1D<class T, int N>

Summary:

- Simple non-shared 1D array implemented as a RAM/ROM.
- Provides built-in checking for index violations for all array dimensions in both pre-HLS model and in RTL (via PSL or SVA assertions in RTL).
- Usage is same as normal C array, aside from declaration.
- See Matchlib toolkit example 35*.

ac_shared_array_1D<class T, int N>

Summary:

- Simple shared 1D array implemented as a RAM/ROM.
- Provides built-in checking for index violations for all array dimensions in both pre-HLS model and in RTL (via PSL or SVA assertions in RTL).
- Usage is same as normal C array, aside from declaration.
- See Matchlib toolkit example 12*.

ac_bank_array_2D<class T, int dim1, int dim0> , ac_bank_array_3D, ac_bank_array_vary

Summary:

- Banked memory where each bank is a separate RAM instance.
- During HLS, right-most index implemented as RAM port, all other indexes become either constants after loop unrolling or a mux tree.
- Should only be used in situations where bank conflicts are clearly not possible.
- Provides best achievable QOR in all cases, including for "non-power of 2" arrays.
- Provides built-in checking for index violations for all array dimensions in both pre-HLS model and in RTL (via PSL or SVA assertions in RTL).
- Usage is same as normal C array, aside from declaration.
- See Matchlib toolkit example 36*.

ac_shared_bank_array_2D<class T, int dim1, int dim0>, ac_shared_bank_array_3D, ac_shared_bank_array_vary

Summary:

- Shared banked memory where each bank is a separate RAM instance.
- During HLS, right-most index implemented as RAM port, all other indexes become either constants after loop unrolling or a mux tree.
- Should only be used in situations where bank conflicts are clearly not possible.
- Provides best achievable QOR in all cases, including for "non-power of 2" arrays.
- Provides built-in checking for index violations for all array dimensions in both pre-HLS model and in RTL (via PSL or SVA assertions in RTL).
- Usage is same as normal C array, aside from declaration.
- See Matchlib toolkit example 38*.

Scratchpad <typename T, int N, int CAPACITY_IN_WORDS>

Summary:

- Banked memory where low order bits of address select bank (like Catapult interleave directive).
- Requests routed thru crossbar to proper bank, responses routed thru crossbar from proper bank.
- Emits error message on any bank conflicts.
- No arbitration or queueing, backpressure is not possible.
- Supports memory transaction log file generation.
- See Matchlib toolkit example 41*.

ArbitratedScratchpad

Summary:

- Banked memory where low order bits of address select bank (like Catapult interleave directive).
- Requests routed thru crossbar to proper bank, responses routed thru crossbar from proper bank.
- Bank conflicts are allowed.
- Has arbitration and queues to handle potential bank conflicts.
- Incoming requests subject to backpressure due to potential bank conflicts.
- See Matchlib toolkit example 42*.

```
template <typename DataType, unsigned int CapacityInWords,
          unsigned int NumInputs, unsigned int NumBanks,
          unsigned int InputQueueLen>
class ArbitratedScratchpad {};
```

extended_array<class T, int N>

Summary:

- Provides memory transaction log file generation for debugging.
- Inherits from ac_array_1D, and can be used in place of it.
- Can also be used in place of ac_shared_array_1D for pre-HLS sim only.
- Usage is same as normal C array, aside from declaration.
- Emits error message on uninitialized memory reads (UMR) in pre-HLS simulation only
- See Matchlib example 12* and “memory_logging_and_debug.pdf” in the Matchlib toolkit doc directory.

Matchlib AXI4 Classes

Introduction

Matchlib contains several configurable AXI4 classes to enable a wide variety of AXI4 interfaces and on-chip networks to be constructed. To use these classes, you will need a basic understanding of the AXI4 protocol. An introduction to the AXI4 protocol and Matchlib classes is provided in the section titled "Matchlib AXI4 Interfaces" in the following document:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/blob/master/matchlib_examples/doc/matchlib_training.pdf

AXI4 Configuration Classes

An AXI config class consists of a struct with an enum defining the following constants:

- dataWidth: The bitwidth of an AXI data word.
- addrWidth: The bitwidth of the AXI address.
- idWidth: The bitwidth of the AXI ID field.
- useVariableBeatSize: Set to 1 if the interface supports beats that are less than one data word, 0 otherwise
- useMisalignedAddresses: Set to 1 if addresses not aligned to the data word boundary are supported, 0 otherwise.
- useLast: Set to 1 if the last bit in write data and read requests is used, 0 otherwise.
- useWriteStrobes: Set to 1 if write strobes are supported, 0 otherwise.
- useBurst: Set to 1 if bursts are supported, 0 otherwise
- useFixedBurst: Set to 1 if fixed-type bursts are supported, 0 otherwise.
- useWrapBurst: Set to 1 if wrap-type bursts are supported, 0 otherwise.
- maxBurstSize: The maximum burst length.
- useQoS: Set to 1 if the QoS field is supported, 0 otherwise.
- useLock: Set to 1 if the Lock field is supported, 0 otherwise.
- useProt: Set to 1 if the Prot field is supported, 0 otherwise.
- useCache: Set to 1 if the Cache field is supported, 0 otherwise.
- useRegion: Set to 1 if the Region field is supported, 0 otherwise.
- useWriteResponses: Set to 1 if write responses are supported, 0 otherwise.
- aUserWidth: The bitwidth of the AxUser fields.
- wUserWidth: The bitwidth of the WUser field.
- bUserWidth: The bitwidth of the BUser field.
- rUserWidth: The bitwidth of the RUser field.

You can define your own custom configuration class, or you can use any of the following pre-defined configuration classes defined in the library, in the axi/axi4_configs.h file:

```
namespace axi {
namespace cfg {

/**
 * \brief A standard AXI configuration.
 */
struct standard {
  enum {
    dataWidth = 64,
    useVariableBeatSize = 0,
    useMisalignedAddresses = 0,
    useLast = 1,
    useWriteStrobes = 1,
    useBurst = 1, useFixedBurst = 0, useWrapBurst = 0, maxBurstSize = 256,
    useQoS = 0, useLock = 0, useProt = 0, useCache = 0, useRegion = 0,
    aUserWidth = 0, wUserWidth = 0, bUserWidth = 0, rUserWidth = 0,
    addrWidth = 32,
    idWidth = 4,
    useWriteResponses = 1,
  };
};
```

```
/**
 * \brief An AXI configuration with no write responses.
 */
struct no_wresp {
  enum {
    dataWidth = 64,
    useVariableBeatSize = 0,
    useMisalignedAddresses = 0,
    useLast = 1,
    useWriteStrobes = 1,
    useBurst = 1, useFixedBurst = 0, useWrapBurst = 0, maxBurstSize = 256,
    useQoS = 0, useLock = 0, useProt = 0, useCache = 0, useRegion = 0,
    aUserWidth = 0, wUserWidth = 0, bUserWidth = 0, rUserWidth = 0,
    addrWidth = 32,
    idWidth = 4,
    useWriteResponses = 0,
  };
};

/**
 * \brief An AXI configuration with no write strobes.
 */
struct no_wstrb {
  enum {
    dataWidth = 64,
    useVariableBeatSize = 0,
    useMisalignedAddresses = 0,
    useLast = 1,
    useWriteStrobes = 0,
    useBurst = 1, useFixedBurst = 0, useWrapBurst = 0, maxBurstSize = 256,
    useQoS = 0, useLock = 0, useProt = 0, useCache = 0, useRegion = 0,
    aUserWidth = 0, wUserWidth = 0, bUserWidth = 0, rUserWidth = 0,
    addrWidth = 32,
    idWidth = 4,
    useWriteResponses = 1,
  };
};
/**
 * \brief An AXI configuration corresponding to the AXI4-Lite standard.
 */
struct lite {
  enum {
    dataWidth = 32,
    useVariableBeatSize = 0,
    useMisalignedAddresses = 0,
    useLast = 0,
    useWriteStrobes = 1,
    useBurst = 0, useFixedBurst = 0, useWrapBurst = 0, maxBurstSize = 1,
    useQoS = 0, useLock = 0, useProt = 0, useCache = 0, useRegion = 0,
    aUserWidth = 0, wUserWidth = 0, bUserWidth = 0, rUserWidth = 0,
    addrWidth = 32,
    idWidth = 0,
    useWriteResponses = 1,
  };
};

/**
 * \brief A configuration like AXI4-Lite, but without write strobes.
 */
struct lite_nowstrb {
  enum {
    dataWidth = 32,
    useVariableBeatSize = 0,
```

```
    useMisalignedAddresses = 0,
    useLast = 0,
    useWriteStrobes = 0,
    useBurst = 0, useFixedBurst = 0, useWrapBurst = 0, maxBurstSize = 1,
    useQoS = 0, useLock = 0, useProt = 0, useCache = 0, useRegion = 0,
    aUserWidth = 0, wUserWidth = 0, bUserWidth = 0, rUserWidth = 0,
    addrWidth = 32,
    idWidth = 0,
    useWriteResponses = 1,
  };
};

}; // namespace axi
}; // namespace cfg
```

AXI4 Transaction Payloads and Master/Slave Ports

The AXI4 transaction payloads and master and slave ports are defined in the axi/axi4.h file. They have a template argument which is the AXI4 configuration class. Based on the provided configuration class, the bitwidths of the various fields of the AXI specification are defined, and classes and convenience functions can be used to instantiate AXI Connections, wire them together, and use them to implement the AXI protocol. Each AXI signal is defined as a UIntOrEmpty of an appropriate width, allowing for the presence of 0-width fields when they can be elided entirely. If useWriteResponses = 0, the B channel is not removed entirely (for implementation convenience) but is reduced to minimum width. All AW and AR fields are identical and are combined into a common AddrPayload type.

The following shows the key classes and methods within the AXI4 payload class.

```
namespace axi {
template <typename Cfg>
class axi4 {
 public:
  typedef AXI4_Encoding Enc;

  enum {
    DATA_WIDTH = Cfg::dataWidth,
    ADDR_WIDTH = Cfg::addrWidth,
    ID_WIDTH = Cfg::idWidth,
    BID_WIDTH = (Cfg::useWriteResponses == 0 ? 0 : Cfg::idWidth),
    ALEN_WIDTH =
        (Cfg::useBurst != 0 ? nvhls::log2_ceil<Cfg::maxBurstSize>::val : 0),
    ASIZE_WIDTH = (Cfg::useVariableBeatSize != 0 ? 3 : 0),
    LAST_WIDTH = (Cfg::useLast != 0 ? 1 : 0),
    CACHE_WIDTH = (Cfg::useCache != 0 ? Enc::ARCACHE::_WIDTH : 0),
    BURST_WIDTH = ((Cfg::useBurst != 0 &&
                    (Cfg::useFixedBurst != 0 || Cfg::useWrapBurst != 0))
                       ? Enc::AXBURST::_WIDTH
                       : 0),
    WSTRB_WIDTH = (Cfg::useWriteStrobes != 0 ? (DATA_WIDTH >> 3) : 0),
    RESP_WIDTH = Enc::XRESP::_WIDTH,
    AUSER_WIDTH = Cfg::aUserWidth,
    WUSER_WIDTH = Cfg::wUserWidth,
    BUSER_WIDTH = (Cfg::useWriteResponses == 0 ? 0 : Cfg::bUserWidth),
    RUSER_WIDTH = Cfg::rUserWidth,
  };

  typedef NVUINTW(ADDR_WIDTH) Addr;
  typedef NVUINTW(DATA_WIDTH) Data;
  typedef typename nvhls::UIntOrEmpty<ID_WIDTH>::T Id;
  typedef typename nvhls::UIntOrEmpty<BID_WIDTH>::T BId;
  typedef typename nvhls::UIntOrEmpty<ALEN_WIDTH>::T BeatNum;
  typedef typename nvhls::UIntOrEmpty<ASIZE_WIDTH>::T BeatSize;
  typedef typename nvhls::UIntOrEmpty<LAST_WIDTH>::T Last;
  typedef typename nvhls::UIntOrEmpty<WSTRB_WIDTH>::T Wstrb;
  typedef typename nvhls::UIntOrEmpty<CACHE_WIDTH>::T Cache;
  typedef typename nvhls::UIntOrEmpty<BURST_WIDTH>::T Burst;
  typedef NVUINTW(RESP_WIDTH) Resp;

  typedef typename nvhls::UIntOrEmpty<AUSER_WIDTH>::T AUser;
  typedef typename nvhls::UIntOrEmpty<WUSER_WIDTH>::T WUser;
  typedef typename nvhls::UIntOrEmpty<BUSER_WIDTH>::T BUser;
  typedef typename nvhls::UIntOrEmpty<RUSER_WIDTH>::T RUser;
```

```
  /**
   * \brief A struct composed of the signals associated with AXI read and write
requests.
   */
  struct AddrPayload : public nvhls_message {
    Id id;
    Addr addr;
    Burst burst;
    BeatNum len;    // A*LEN
    BeatSize size;  // A*SIZE
    Cache cache;
    AUser auser;


   AddrPayload() {  }

  };

  /**
   * \brief A struct composed of the signals associated with an AXI read
   * response.
   */
  struct ReadPayload : public nvhls_message {
    Id id;
    Data data;
    Resp resp;
    Last last;
    RUser ruser;


   ReadPayload() { }
  };

  /**
   * \brief A struct composed of the signals associated with an AXI write
   * response.
   */
  struct WRespPayload : public nvhls_message {
    BId id;
    Resp resp;
    BUser buser;


   WRespPayload() {}

  };

  /**
   * \brief A struct composed of the signals associated with AXI write data.
   */
  struct WritePayload : public nvhls_message {
    Data data;
    Last last;
    Wstrb wstrb;
    WUser wuser;

    WritePayload() {
    }
  };

 /**
   * \brief The AXI read class.
   *
```

```
   * Each Connections implementation contains two ready-valid interfaces, AR for
   * read requests and R for read responses.
   */
  class read {
   public:
    /**
     * \brief The AXI read channel, used for connecting an AXI master and AXI slave.
     */
    template <Connections::connections_port_t PortType = AUTO_PORT>
    class chan {
     public:
     typedef Connections::Combinational<AddrPayload, PortType> ARChan;
     typedef Connections::Combinational<ReadPayload, PortType> RChan;

      ARChan ar;  // master to slave
      RChan r;    // slave to master

      chan(const char *name);
    }; // read::chan

   /**
     * \brief The AXI read master port.  This port has an AR request channel as output
and an R response channel as input.
     */
    template <Connections::connections_port_t PortType = AUTO_PORT>
    class master {
     public:
      typedef Connections::Out<AddrPayload, PortType> ARPort;
      typedef Connections::In<ReadPayload, PortType> RPort;

      ARPort ar;
      RPort r;

      master(const char *name);

      void reset() {
        ar.Reset();
        r.Reset();
      }

      template <class C>
      void operator()(C &c) {
        ar(c.ar);
        r(c.r);
      }
    }; // read::master

    /**
     * \brief The AXI read slave port.  This port has an AR request channel as input
and an R response channel as output.
     */
    template <Connections::connections_port_t PortType = AUTO_PORT>
    class slave {
     public:
      typedef Connections::In<AddrPayload, PortType> ARPort;
      typedef Connections::Out<ReadPayload, PortType> RPort;

      ARPort ar;
      RPort r;

      slave(const char *name);

      void reset() {
```

```
        ar.Reset();
        r.Reset();
      }

      template <class C>
      void operator()(C &c) {
        ar(c.ar);
        r(c.r);
      }
    }; // read::slave
  }; // read

 /**
   * \brief The AXI write class.
   *
   * Each Connections implementation contains three ready-valid interfaces: AW
   * for write requests, W for write data, and B for write responses.
   */
  class write {
   public:
    /**
     * \brief The AXI write channel, used for connecting an AXI master and AXI slave.
     */
    template <Connections::connections_port_t PortType = AUTO_PORT>
    class chan {
     public:
     typedef Connections::Combinational<AddrPayload, PortType> AWChan;
     typedef Connections::Combinational<WritePayload, PortType> WChan;
     typedef Connections::Combinational<WRespPayload, PortType> BChan;

      AWChan aw;  // master to slave
      WChan w;    // master to slave
      BChan b;    // slave to master

      chan(const char *name);

    };  // write::chan

   /**
     * \brief The AXI write master port.  This port has AW and W request channels as
outputs and a B response channel as input.
     */
    template <Connections::connections_port_t PortType = AUTO_PORT>
    class master {
     public:
      typedef Connections::Out<AddrPayload, PortType> AWPort;
      typedef Connections::Out<WritePayload, PortType> WPort;
      typedef Connections::In<WRespPayload, PortType> BPort;

      AWPort aw;
      WPort w;
      BPort b;

      master(const char *name);

      void reset() {
        aw.Reset();
        w.Reset();
        b.Reset();
      }


      template <class C>
```

```
      void operator()(C &c) {
        aw(c.aw);
        w(c.w);
        b(c.b);
      }
    };  // write::master

   /**
     * \brief The AXI write slave port.  This port has AW and W request channels as
inputs and a B response channel as output.
     */
    template <Connections::connections_port_t PortType = AUTO_PORT>
    class slave {
     public:
      typedef Connections::In<AddrPayload, PortType> AWPort;
      typedef Connections::In<WritePayload, PortType> WPort;
      typedef Connections::Out<WRespPayload, PortType> BPort;

      AWPort aw;
      WPort w;
      BPort b;


      slave(const char *name);

      void reset() {
        aw.Reset();
        w.Reset();
        b.Reset();
      }


      template <class C>
      void operator()(C &c) {
        aw(c.aw);
        w(c.w);
        b(c.b);
      }
    }; // write::slave
  }; // write
}; // axi4

}; // namespace axi
```

AXI4 Burst Segmentation

Matchlib provides classes to perform automatic burst segmentation. To use these classes, you will need a basic understanding of the AXI4 protocol. An introduction to the AXI4 protocol and Matchlib classes is provided in the section titled “Matchlib AXI4 Interfaces” in the following document:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/blob/master/matchlib_examples/doc/matchlib_training.pdf

When automatic burst segmentation is used, models using AXI4 master interfaces do not need to deal with the details of adhering to AXI4 burst length limits, since the automatic burst segmenter supports unlimited length bursts.

The segment class is configured with an AXI4 config class and provides the following general typedefs and classes:

```
namespace axi
{

  template <typename Cfg>
  class axi4_segment : public axi::axi4<Cfg>
  {
  public:

    typedef Cfg axi_cfg;
    typedef AXI4_Encoding Enc;

    typedef typename axi::axi4<Cfg>::AddrPayload ar_payload;
    typedef typename axi::axi4<Cfg>::AddrPayload aw_payload;
    typedef typename axi::axi4<Cfg>::ReadPayload r_payload;
    typedef typename axi::axi4<Cfg>::WritePayload w_payload;
    typedef typename axi::axi4<Cfg>::WRespPayload b_payload;

    template <Connections::connections_port_t PortType = AUTO_PORT>
    using r_chan = typename axi::axi4<Cfg>::read::template chan<PortType>;
    template <Connections::connections_port_t PortType = AUTO_PORT>
    using w_chan = typename axi::axi4<Cfg>::write::template chan<PortType>;

    static const int bytesPerBeat = axi::axi4<Cfg>::DATA_WIDTH >> 3;

    struct ex_ar_payload : public ar_payload {
      // this payload enables burst reads of unlimited length - it extends from
ar_payload
      NVUINTW(32) ex_len{0};
    };

    struct ex_aw_payload : public aw_payload {
      // this payload enables burst writes of unlimited length - it extends from
aw_payload
      NVUINTW(32) ex_len{0};
    };

    template <Connections::connections_port_t PortType = AUTO_PORT>
    struct w_master: public axi::axi4<Cfg>::write::template master<PortType>;

    template <Connections::connections_port_t PortType = AUTO_PORT>
    struct r_master: public axi::axi4<Cfg>::read::template master<PortType>;
```

```
    template <Connections::connections_port_t PortType = AUTO_PORT>
    struct r_slave : public axi::axi4<Cfg>::read::template slave<PortType>;

   template <Connections::connections_port_t PortType = AUTO_PORT>
    struct w_slave : public axi::axi4<Cfg>::write::template slave<PortType>;

  }
}; // namespace axi
```

It is important to understand that if a module uses the automatic burst segmenter, this is not visible on the external interface of the module. Instead, the external interface of the module always uses the native AXI4 protocol. (This is true both in the pre-HLS simulation and in the post-HLS RTL.) The implementation details of the segmenter are internal to the module and are associated only with the specific write master or read master ports that are using the segmenter.

The axi4_segment.h file defines the following macros:

```
AXI4_W_SEGMENT_BIND
AXI4_R_SEGMENT_BIND
AXI4_W_SEGMENT_RESET
AXI4_R_SEGMENT_RESET
AXI4_W_SEGMENT_CFG
AXI4_R_SEGMENT_CFG
```

The use of these macros is shown in the code within the dma.h file from the Matchlib toolkit include directory, copied below. Note the text highlighted in bold.

```
typedef axi::axi4_segment<axi::cfg::standard> local_axi;

#pragma hls_design top
class dma : public sc_module, public local_axi
{
public:
  sc_in<bool> CCS_INIT_S1(clk);
  sc_in<bool> CCS_INIT_S1(rst_bar);

  // Declare Native AXI4 r/w master/slave ports
  r_master<> CCS_INIT_S1(r_master0);
  w_master<> CCS_INIT_S1(w_master0);
  r_slave<>  CCS_INIT_S1(r_slave0);
  w_slave<>  CCS_INIT_S1(w_slave0);
  Connections::Out<bool> CCS_INIT_S1(dma_done);

  SC_CTOR(dma) {
    // Do port bindings for segmenter modules
    AXI4_W_SEGMENT_BIND(w_segment0, clk, rst_bar, w_master0);
    AXI4_R_SEGMENT_BIND(r_segment0, clk, rst_bar, r_master0);
  }

private:

  Connections::Combinational<dma_cmd> CCS_INIT_S1(dma_cmd_chan);

  // Declare segmenter modules.
  AXI4_W_SEGMENT_CFG(local_axi, w_segment0)
  AXI4_R_SEGMENT_CFG(local_axi, r_segment0)

  // master_process recieves dma_cmd transactions from the slave_process.
  // the master_process performs the dma operations via the master0 axi port,
```

```
  // and then sends a done signal to the requester via the dma_done transaction.

  void master_process() {
    // Reset segmenters in the Reset state:
    AXI4_W_SEGMENT_RESET(w_segment0, w_master0);
    AXI4_R_SEGMENT_RESET(r_segment0, r_master0);

    dma_cmd_chan.ResetRead();
    dma_done.Reset();

    wait();

    while (1) {
      // Declare AR and AW payloads with extended (unlimited) burst lengths
      ex_ar_payload ar;
      ex_aw_payload aw;

      dma_cmd cmd = dma_cmd_chan.Pop();
      // Set the extended burst size
      ar.ex_len = cmd.len;
      aw.ex_len = cmd.len;
      ar.addr = cmd.ar_addr;
      aw.addr = cmd.aw_addr;

      // Push the extended AR and AW payloads to the segmenters
      r_segment0_ex_ar_chan.Push(ar);
      w_segment0_ex_aw_chan.Push(aw);

#pragma hls_pipeline_init_interval 1
#pragma pipeline_stall_mode flush
      while (1) {
        // get the read response
        r_payload r = r_master0.r.Pop();
        w_payload w;
        w.data = r.data;
        // Push the write data to the write segmenter
        w_segment0_w_chan.Push(w);

        if (ar.ex_len-- == 0) { break; }
      }

      // get the write response from the write segmenter
      b_payload b = w_segment0_b_chan.Pop();
      dma_done.Push(b.resp == Enc::XRESP::OKAY);
    }
  }

  // slave_process accepts incoming axi4 requests from slave0 and
  // programs the dma registers.
  // when the start register is written to,
  // a dma_cmd transaction is sent to the dma master_process

  void slave_process() {
    // Reset the slave ports in the reset state
    r_slave0.reset();
    w_slave0.reset();
    dma_cmd_chan.ResetWrite();

    wait();

    dma_cmd cmd1;

    while (1) {
```

```
      aw_payload aw;
      w_payload w;
      b_payload b;

      // Ask for a single write item that master sent to the slave port.
      // If master sent a burst, the get_single_write() function will automatically
      // handle it as an AXI4 slave error and return false.
      if (w_slave0.get_single_write(aw, w, b)) {
        b.resp = Enc::XRESP::SLVERR;
        switch (aw.addr) {
          case offsetof(dma_address_map, ar_addr):
            cmd1.ar_addr = w.data;
            b.resp = Enc::XRESP::OKAY;
            break;
          case offsetof(dma_address_map, aw_addr):
            cmd1.aw_addr = w.data;
            b.resp = Enc::XRESP::OKAY;
            break;
          case offsetof(dma_address_map, len):
            cmd1.len = w.data;
            b.resp = Enc::XRESP::OKAY;
            break;
          case offsetof(dma_address_map, start):
            dma_cmd_chan.Push(cmd1);
            b.resp = Enc::XRESP::OKAY;
            break;
        }
        // Push the write response to the master
        w_slave0.b.Push(b);
      }
    }
  }
};
```

Slave Port Convenience Functions

There are several convenience functions available when using the w_slave and r_slave classes within the axi4_segment class.

The first function is w_slave::get_single_write(), shown in the example directly above. This function has the following declaration:

```
bool get_single_write(aw_payload &ret_aw, w_payload &ret_w, b_payload &ret_b);
```

This function is designed to be used in AXI4 slaves that expect only a single beat at any write address. If the upstream write master sends a burst write of greater than 1 beat, the get_single_write() function will automatically return the appropriate AXI4 error transactions upstream, and then the function itself will return false. Otherwise, the function will return true and the aw and w payloads will contain the incoming transactions, and the b payload will be preset to contain the appropriate transaction ID. It is the responsibility of the write slave model to push the actual b transaction back upstream to the master if the function returned true.

The next function is r_slave::single_read(), which has the signature:

```
bool single_read(ar_payload &ret_ar, r_payload &ret_r);
```

This function can be used with a model using the r_slave class when the model only expects burst reads that contain 1 beat. Similar to the get_single_write() function above, the single_read() function will automatically handle returning proper error transactions upstream for any read bursts greater than 1 beat, and the function itself will return false in this case. When the burst length is 1 beat, the function returns true, and the ar payload will contain the read address transaction, and the r payload will be preset with the transaction ID and last bit. It is the responsibility of the model to fill in the read data and then push the read transaction back upstream to the read master.

When you want to handle bursts of greater than 1 beat within an AXI4 slave, the following convenience functions enable this. The first set of functions is for read bursts, and includes the r_slave::start_multi_read() and r_slave::next_multi_read() functions. The second set of functions is for write bursts, and includes the r_slave::start_multi_write() and r_slave::next_multi_write() functions. These functions are used in Matchlib toolkit example 55*/mixed_ram.h and are shown below.

```
template <class cfg>
class ram : public sc_module
{
public:
  sc_in<bool> CCS_INIT_S1(clk);
  sc_in<bool> CCS_INIT_S1(rst_bar);
  typename cfg::template r_slave<AUTO_PORT>     CCS_INIT_S1(r_slave0);
  typename cfg::template w_slave<AUTO_PORT>     CCS_INIT_S1(w_slave0);

  static const int sz = 0x10000; // size in cfg::DATA_WIDTH words

  typedef ac_int<cfg::DATA_WIDTH, false> arr_t;
  arr_t *array {0};

  SC_CTOR(ram) {
    array = new arr_t[sz];

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

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

  void slave_r_process() {
    r_slave0.reset();

    wait();

    while (1) {
      typename cfg::ar_payload ar;

      // start an axi4 slave read burst of 1 or more beats
      r_slave0.start_multi_read(ar);

      while (1) {
        typename cfg::r_payload r;

        if (ar.addr >= (sz * cfg::bytesPerBeat)) {
          SC_REPORT_ERROR("ram", "invalid addr");
          r.resp = cfg::Enc::XRESP::SLVERR;
```

```
        } else {
          r.data = array[ar.addr / cfg::bytesPerBeat];
        }

        // send read data upstream to AXI4 master,
        // loop if more beats to process, otherwise we are done

        if (!r_slave0.next_multi_read(ar, r)) { break; }
      }
    }
  }

  void slave_w_process() {
    w_slave0.reset();
    wait();

    while (1) {
      typename cfg::aw_payload aw;
      typename cfg::b_payload b;

      // start an AXI4 slave write burst of 1 or more beats
      w_slave0.start_multi_write(aw, b);


      while (1) {
        // Pop the write data beat
        typename cfg::w_payload w = w_slave0.w.Pop();

        if (aw.addr >= (sz * cfg::bytesPerBeat)) {
          SC_REPORT_ERROR("ram", "invalid addr");
          b.resp = cfg::Enc::XRESP::SLVERR;
        } else {
          decltype(w.wstrb) all_on{~0};

          if (w.wstrb == all_on)
          { array[aw.addr / cfg::bytesPerBeat] = w.data.to_uint64(); }
          else {
            // code here omitted…
          }
        }

        // if there are more write beats to process then loop, otherwise we are done.
        if (!w_slave0.next_multi_write(aw)) { break; }
      }

      // the write response (b) payload must be explicitly sent upstream to master.
      w_slave0.b.Push(b);
    }
  }
};
```

AxiSplitter<>

The AxiSplitter<> module performs routing of AXI4 requests based on a configurable address map.

Refer to Matchlib toolkit example 09* for example usage of this model.

AxiArbiter<>

The AxiArbiter<> module performs round-robin arbitration of AXI4 requests.

Refer to Matchlib toolkit example 09* for example usage of this model.

AXI4-Lite

To model the AXI4-Lite protocol using the Matchlib AXI4 classes, select one of the AXI4-Lite configuration classes provided in the library. The two provided in axi/axi4_configs.h are:

```
/**
 * \brief An AXI configuration corresponding to the AXI4-Lite standard.
 */
struct lite {
  enum {
    dataWidth = 32,
    useVariableBeatSize = 0,
    useMisalignedAddresses = 0,
    useLast = 0,
    useWriteStrobes = 1,
    useBurst = 0, useFixedBurst = 0, useWrapBurst = 0, maxBurstSize = 1,
    useQoS = 0, useLock = 0, useProt = 0, useCache = 0, useRegion = 0,
    aUserWidth = 0, wUserWidth = 0, bUserWidth = 0, rUserWidth = 0,
    addrWidth = 32,
    idWidth = 0,
    useWriteResponses = 1,
  };
};
/**
 * \brief A configuration like AXI4-Lite, but without write strobes.
 */
struct lite_nowstrb {
  enum {
    dataWidth = 32,
    useVariableBeatSize = 0,
    useMisalignedAddresses = 0,
    useLast = 0,
    useWriteStrobes = 0,
    useBurst = 0, useFixedBurst = 0, useWrapBurst = 0, maxBurstSize = 1,
    useQoS = 0, useLock = 0, useProt = 0, useCache = 0, useRegion = 0,
    aUserWidth = 0, wUserWidth = 0, bUserWidth = 0, rUserWidth = 0,
    addrWidth = 32,
    idWidth = 0,
    useWriteResponses = 1,
  };
};
```

In your model, use the AXI4 payload classes and master and slave ports. Note that the AXI4-Lite protocol does not support bursts, so all read and write transactions have a single data beat. Thus, it is very easy to model AXI4-Lite protocol directly in your model. There is an example of the AXI4-Lite protocol in Matchlib toolkit example 87*.

Matchlib APB Classes

The content of this section is still under development.
Refer to the APB example in Matchlib toolkit example 52*.

Matchlib Utility Functions

BitsToType

The BitsToType function converts an sc_lv<width> to the specified type. To use this function, the type T must have a Marshaller function defined, which is typically done via the AUTO_GEN_FIELDS macro for user defined types.

The declaration is:

```
template <typename T>
T BitsToType(sc_lv<Wrapped<T>::width> mbits);
```

Example usage:

```
#include "TypeToBits.h"
template <class L, int R>
void bits_to_type_if_needed(L& left, sc_lv<R>& right)
{
   left = BitsToType<L>(right);
}
```

TypeToBits

The TypeToBits function converts the specified type to an sc_lv<width>. To use this function, the type T must have a Marshaller function defined, which is typically done via the AUTO_GEN_FIELDS macro for user defined types.

The declaration is:

```
template <typename T>
sc_lv<Wrapped<T>::width> TypeToBits(T in) {}
```

Example usage:

```
#include "TypeToBits.h"
template <int L, class R>
void type_to_bits_if_needed(sc_lv<L>& left, R& right)
{
   left = TypeToBits(right);
}
```