System Verification with MatchLib

Russell Klein, Program Director
Siemens EDA
Agenda

• Motivation
• MatchLib
• AXI Bus Modeled in MatchLib
• Processor Models
  • Host Code
  • Fast ISS
  • RTL
• A Simple Design Example
Motivation

• System Level Verification needs to be done early
  • RTL is available too late, and runs to slowly
  • Emulation and FPGA prototypes are available after RTL
• Virtual prototypes can be available early
  • But may not match final design
  • Verify what you build
• MatchLib is a synthesizable (thru HLS) communication framework
  • Enables early and practical system level verification
  • Higher level than RTL, much higher performance
  • Throughput accurate with the implementation
Path from Abstract to Detailed

• With HLS functional blocks can be modeled and simulated at the high level
• With MatchLib, communication elements can now be modeled and simulated abstractly

• We need to also bring in IP like processors
What is MatchLib?

- **Modular Approach To Circuits and Hardware Library**
- Developed by NVIDIA Labs while creating a machine learning accelerator
  - Needed a more abstract method for simulating system behavior
  - Needed to be able to closely (but not exactly) model performance
- Needed to evaluate many different architectures for performance and power
  - Could not afford to design them all in RTL
  - Could not afford to be significantly wrong
What is MatchLib?

• Library of reusable communication models and functions
  • Encapsulate verified functionality
  • Encapsulate QoR optimized implementation
  • Heavy use of templates and parameterization

• Common HW communication components modeled as
  • C++ functions: datapath description
  • C++ classes: state updating methods
  • SystemC modules: self contained modules

• Testbench components
MatchLib Addresses Complexity and Risk

• The complexity/risk in many of today’s advanced HW designs has shifted from the past.
• Today’s HW designs often process huge sets of data, with large intermediate results.
  • Machine Learning
  • Computer Vision
  • 5G Wireless
• The design of the memory/interconnect architecture and the management of data movement in the system often has more impact on power/performance than the design of the computation units themselves.
• Evaluating and verifying memory/interconnect architecture at RTL level is not feasible:
  • Too late in design cycle
  • Too much work to evaluate multiple candidate architectures.
• The most difficult/costly HW (& HW/SW) problems are found during system integration.
  • If integration first occurs in RTL, it is very late and problems are very costly.
  • MatchLib lets integration occur early when fixing problems is much cheaper.
NVIDIA Matchlib vs RTL Results

**RC17 SYSTEMC-BASED VERIFICATION**
Functional and Performance Verification on SystemC models

**FUNCTIONAL VERIFICATION**
- Most verification run on SystemC/C++, signed off using C++ coverage tools
- Reuse of SystemC testbenches on HLS-generated RTL DUTs
- Automated stall injection and in-design assertions for improved coverage

**PERFORMANCE VERIFICATION**
- Sim-accurate SystemC models for Latency-Insensitive Channels
- Up to 30x speedup vs. RTL
- Less than 2.6% error in cycle count

**RC17 SOC PHYSICAL DESIGN**
87M Transistor SoC in TSMC 16nm FinFET

<table>
<thead>
<tr>
<th>RC17 Stats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Die Size</td>
</tr>
<tr>
<td>Partitions</td>
</tr>
<tr>
<td>Frequency range</td>
</tr>
<tr>
<td>Voltage range</td>
</tr>
<tr>
<td>Performance (16b GMACS)</td>
</tr>
<tr>
<td>Max GMACS/W</td>
</tr>
<tr>
<td>Programmability</td>
</tr>
</tbody>
</table>
MatchLib is Open Source on Github

- https://github.com/NVlabs/matchlib

Or search for “matchlib github”

MatchLib

MatchLib is a SystemC/C++ library of commonly-used hardware functions and components that can be synthesized by most commercially-available HLS tools into RTL.

Doxxygen-generated documentation can be found here.

MatchLib is based on the Connections latency-insensitive channel implementation. Connections is included with the Catapult HLS tool and is available open-source on HLSlibs. Additional documentation on the Connections latency-insensitive channel implementation can be found in the Connections Guide.

Getting Started

Tool versions

MatchLib is regressed against the following tool/dependency versions:
More Details

• Good 30 minute intro video here:
  • or Google “nvidia machine learning mentor events”
Key Parts of MatchLib

• “Connections”
  • Synthesizable (HLS) Message Passing Framework
  • SystemC/C++ used to accurately model concurrent IO that synthesized HW will have
  • Automatic stall injection enables interconnect to be stress tested at C++ level

• Parameterized AXI4 Fabric Components
  • Router/Splitter
  • Arbiter
  • AXI4 <-> AXI4Lite
    • Automatic burst segmentation and last bit generation

• Parameterized Banked Memories, Crossbar, Reorder Buffer, Cache
• Parameterized NOC components
MatchLib AXI4

- Class that models the AXI-4 protocol using a combinatorial channel
- Configurable for
  - Width of address, data, ID, and user fields
  - Optional read response and “last” signal
- Access classes
  - `axi::axi4<Cfg>::read::master` and `axi::axi4<Cfg>::read::slave`
  - `axi::axi4<Cfg>::write::master` and `axi::axi4<Cfg>::write::slave`
- Current version only performs full bus-width accesses
  - We extended these class with `read_xx` and `write_xx` methods for partial bus width accesses
#ifndef __INCLUDED_SYS_AXI_STRUCT_H__
define __INCLUDED_SYS_AXI_STRUCT_H__

struct sysbus_axi4_config {
    enum {
        dataWidth = 64,
        addrWidth = 44,
        
        useVariableBeatSize = 0, useLast = 1,
        useMisalignedAddresses = 0, useBurst = 1,
        useWriteStrobes = 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, idWidth = 4,
        useWriteResponses = 1
    };
};

typedef typename axi::axi4_segment<sysbus_axi4_config> sysbus_axi;
typedef typename axi::axi4_segment<axi::cfg::standard> local_axi64;
typedef typename axi::axi4<axi::cfg::lite_nowstrb> local_axi4_lite;
typedef typename axi::axi4_segment<axi::cfg::lite_nowstrb> local_axi4_lite_seg;

#endif

• Driven off a set of emuns in a configuration struct
• Used by classes and functions to implement a specific AXI
• Common configurations are provided with the library
AXI Bus Segments

- From configuration struct, bus segments and ports are defined
- `r_master, w_master` are ports with the sc_in/sc_out to attach to a slave
- `r_slave, w_slave` are ports with the sc_in/sc_out to attach to a master
- `r_chan, w_chan` are signal bundles used in a module
- All these have methods that are used to affect traffic on the bus

```cpp
class my_hw_module : public sc_module, public sysbus_axi
{
public:
    //== Ports
    sc_in<bool>  clk;
    sc_in<bool>  reset_bar;
    r_master     read_master;
    w_master     write_master;
    r_slave     read_slave;
    w_slave     write_slave;
    //== Local signals
    r_chan     read_bus_signals;
    w_chan     write_bus_signals;
};
```
Payload Definitions

//== Local Signals
aw_payload  aw;
w_payload    w;
b_payload    b;
ar_payload   ar;
r_payload    r;

//== send data method
void send_it(sc_int addr, sc_int *data, sc_int count, sc_int)
{
    aw.addr = addr;
    aw.len = count;
    aw.id = master_id;
    w_master.aw.Push(aw);
    w_strb = 0xF;
    for (int i=0; i<count; i++) {
        w.data = data[i];
        w.last = (i == count-1) ? 0 : 1;
        w_master.w.Push(w);
    }
    b = w_master.b.Pop();
    if (b.resp != Enc::XRESP::OKAY) printf("something bad happened! \n");
}
Creating an AXI Fabric

- RISC-V Rocket Core
- I/O AXI
- Memory AXI
- Code & Data Memory
- AXI Fabric
- UART
- Shared Memory
- Inference Accel
Creating an AXI Fabric

RISC-V Rocket Core

I/O AXI

Memory AXI

Code & Data Memory

Splitters

Arbiters

UART

Shared Memory

Inference Accel
Fabric - Splitters

- Splitters fan out the AXI signals based on the “addr_bounds” array
- This code fragment shows items relevant to splitters
  - Omits other details

```cpp
class fabric : public sc_module, public sysbus_axi
{
  //== Ports
  sc_in<bool> clk;
  sc_in<bool> reset_bar;
  r_slave    r_cpu, r_acc;
  w_slave    w_cpu, w_acc;

  //== Local signals
  r_chan     r_cpu2mem, r_cpu2uart, r_cpu2acc;
  w_chan     w_cpu2mem, w_cpu2uart, w_cpu2acc;
  r_chan     r_acc2mem, r_acc2uart;
  w_chan     w_acc2mem, w_acc2uart;

  sc_signal<32> addr_bounds[3][2];

  //== Instances
  axi_splitter<sysbus_axi4_config, 3, 32> cpu_router;
  axi_splitter<sysbus_axi4_config, 2, 32> acc_router;

  SC_CTOR(fabric) {
    addr_bounds[0][0] = 0x70000000; addr_bounds[0][1] = 0x7FFFFFFF; // shared mem
    addr_bounds[1][0] = 0x60000000; addr_bounds[1][1] = 0x6000FFFF; // UART
    addr_bounds[2][0] = 0x60010000; addr_bounds[2][1] = 0x6001FFFF; // Accelerator

    // attach signals to routers
```
Fabric - Arbiters

- Arbiters act as a multiplexer selecting a transaction from the splitters
- This code fragment shows items relevant to arbiters
  - Omits other details

```cpp
class fabric : public sc_module, public sysbus_axi
{
    //== Ports
    sc_in<bool> clk;
    sc_in<bool> reset_bar;

    r_master r_mem, r_uart, r_acc;
    w_master w_mem, w_uart, w_acc;

    //== Local signals
    r_chan r_cpu2mem, r_cpu2uart, r_cpu2acc;
    w_chan w_cpu2mem, w_cpu2uart, w_cpu2acc;

    r_chan r_acc2mem, r_acc2uart;
    w_chan w_acc2mem, w_acc2uart;

    //== Instances
    axi_arbiter<sysbus_axi4_config, 2, 4> mem_arbiter;
    axi_arbiter<sysbus_axi4_config, 2, 4> uart_arbiter;
    axi_arbiter<sysbus_axi4_config, 1, 1> acc_arbiter;

    SC_CTOR(fabric) {
        // attach signals to routers
    }
};
```
Including a CPU in the Simulation

• No CPU models are available with MatchLib interfaces
  • Some type of “wrapper” is required to interface CPU model to the design

• CPU Models
  • Host Code Execution (HCE)
    • Very fast, functionally accurate
  • Fast Instruction Set Simulators (Spike, QEMU, AFM, OVPSim)
    • Runs target instruction set, functionally accurate
  • RTL
    • Slow, but clock cycle accurate
Host Code Execution

- Code is compiled for the host (simulation computer)
- Annotations are made for bus cycles to be sent to hardware
- Code activity is not included in the simulation
  - Instruction fetches and stack/data references are omitted
  - This may or may not impact your verification goals
HCE Example

CPU (Host Program)

AXI Fabric

UART

Shared Memory

Inference Accel

AXI segment

SystemC/MatchLib

C/C++ (Host code)
HCE Example – SystemC wrapper

- Wrapper for HCE function
- Defines clk, reset, and bus connection
- Calls "cpu_thread()" which can perform any function

```cpp
class host_code_tb : public sc_module,
    public sysbus_axi
{
    public:
        sc_in<bool>    clk;
        sc_in<bool>    reset_n;
        r_master       read_master;
        w_master       write_master;

        void sw_thread()
        {
            write_master.reset();
            read_master.reset();
            wait();
            cpu_thread();
            sc_stop();
        }

    SC_CTOR(host_code_tb)
    {
```
HCE Example – cpu_thread

```c
void cpu_thread()
{
    int x;
    int errors;
    TB_WRITE(0x1234, 0xA5A5);
    x = TB_READ(0x1234);
    if (x != 0xA5A5) {
        errors++;
        report_error(MISMATCH, 0x1234);
    }
    return;
}
```

- `cpu_thread()` can perform read and write operations on the AXI bus
  - Typically, through macros so code can be consistent through different stages of verification
- `cpu_thread` is a member function of class `host_code_tb`
HCE Example – Bus Interface Macros

- Allows for common code for HCE and cross compiled
- Macros can be defined for different sized accesses and named peripheral registers
  - Can drive burst cycles (approximating cache line accesses)
Fast Processor Simulator

• Code is cross compiled for the target processor
  • Running actual ARM or RISC-V instructions
• Processor simulator interprets instructions and emulates behavior of the program on the target processor
• Bus cycles are generated using address dereferences
  • works the same as on a real processor
• Like HCE, code activity is (typically) not included in the simulation
  • Local memory is used for code and data storage
  • Instruction fetches and stack/data references are omitted from simulation
  • This may or may not impact your verification goals
Fast Processor Example

IPC Channels

CPU (ISS)
Code & data memory
ISS Process

SystemC Process
AXI Fabric
UART
Shared Memory
Inference Accel

AXI Xactor

SystemC/MatchLib
Fast Processor Model
Fast Processor Simulator

- There are many fast processor models
  - QEMU, Spike, OVPSim, AFM, etc.
- All run in the 100s - 1,000s of millions of instructions per second
- Not clock cycle accurate, usually do not model caches

- Each will have different methods for capturing bus cycles
  - We used QEMU (Quick EMUlator) [http://www.qemu.org](http://www.qemu.org)
QEMU – co-simulation basics, SystemC side

- AXI transactor thread launches QEMU process, with executable image
  - Can be done with threads, more complex but faster
- Thread creates sockets for IPC
- Waits on reset()
- Then waits on bus cycle or advance command from socket
- Get bus cycle or advance command
  - Run bus cycle and return result or advance a number of wait() operations
QEMU – co-simulation basics, ISS side

- Defines I/O memory region to trap bus cycles
  - `memory_region_init_io()`
- Connects to sockets for IPC
- Using TCG plugin, set up instruction count limit
  - QEMU advances only a certain amount of time, then communicates with HW
  - Run to next I/O cycle (bus operation) or instruction count limit
    - If instruction count limit hit, send “advance” command
    - If I/O cycle hit, send cycle and get result
RTL Processor Models

• Neither HCE nor ISS models of software behavior are timing accurate
  • No model of code and data accesses
  • No model of the impact of caches
  • No model of the computation time

• **IF** the processor and software materially impact the performance of the system, then a realistic model is needed
  • Usually, this is RTL

• An RTL processor can be combined with MatchLib and SystemC for higher performance, but throughput accurate verification

• RTL + SystemC is well understood
  • But there are some quirks with MatchLib
RTL Example

Logic Simulation Process

- CPU (RTL)
- SystemC
- Verilog
- Code & data memory
- AXI Fabric
- UART
- Shared Memory
- Inference Accel

- SystemC/MatchLib
- Verilog RTL
module top (input clk, input reset_bar);

wire aw_ready;
wire aw_valid;
wire [75:0] aw_msg; // repeat for all segments

systemc_subsystem_wrapper scsw( // SystemC subsystem
  .clk(clk), .reset_bar(reset_bar),
    .aw_ready_port (aw_ready),
    .aw_valid_port (aw_valid),
    .aw_msg_port (aw_msg), // repeat for all segments
);

rocket_subsystem risc_v( // RTL processor
  .clk(clk), .reset_bar(reset_bar),
    .aw_ready_port (aw_ready),
    .aw_valid_port (aw_valid),
    .aw_msg_port (aw_msg), // repeat for all segments

• All AXI segments declared as ready/valid/msg triplet
• Size would be a summation of field widths
SystemC Sub-system Wrapper

SC_MODULE(systemc_subsystem_wrapper)
{
    sc_in<bool> clk;
    sc_in<bool> reset_bar;
    sc_in<bool> aw_ready_port;
    sc_in<bool> aw_valid_port;
    sc_in<sc_lv<76>> aw_msg_port;
    // repeat for all segments
    systemc_sub_system scs;
    SCCTOR(systemc_subsystem_wrapper)
    {
        scs.clk(clk);
        scs.reset_bar(reset_bar);
        scs.w_cpu.aw.rdy(aw_ready_port);
        scs.w_cpu.aw.val(aw_valid_port);
        scs.w_cpu.aw.msg(aw_msg_port);
        // repeat for all segments
    }
}

• “msg” bundles passed to SystemC as sc_lv logic vectors

• ready/valid/msg triplet is mapped to <port_name>.<seg>.[rdy|val|msg]
Rocket Subsystem Verilog

module rocket_subsystem (input clk, input reset_bar,
                         input aw_ready,
                         output aw_valid,
                         output [75:0] aw_msg,
                         // repeat for all segments
                         );

// declarations and code here...

// assignments to/from msg to local signals

assign aw_msg = {aw_len, aw_addr, aw_id};
assign w_msg = {w_strb, w_last, w_data};
assign {b_resp, b_id} = b_msg;
assign ar_msg = {ar_len, ar_addr, ar_id};
assign {r_last, r_resp, r_data, r_id} = r_msg;

• “msg” bundles are passed into System Verilog module
• Break out into individual signals using the Verilog concatenate operator

• For ordering, you need to dig through matchlib/axi code to find declaration.
• Pro Tip: or compile and bring up in a waveform viewer or debugger
Example

• Yolo object recognition algorithm
• Characterize function/performance of AI accelerator with MatchLib and RISC-V processor

Code available at: https://github.com/hlslibs/ac_ml/tree/master/designs/HLS_SEMINAR_2021/system_design
Example Design

- RISC-V Rocket Core
- I/O AXI
- Memory AXI
- Code & Data Memory
- AXI Fabric
- UART
- Shared Memory
- Inference Accel
## Results

<table>
<thead>
<tr>
<th>Processor</th>
<th>Interconnect</th>
<th>Accelerator</th>
<th>Run time for 1 inference (seconds)</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Code</td>
<td>C++ (N/A)</td>
<td>C++</td>
<td>12</td>
<td>n/a</td>
</tr>
<tr>
<td>Host Code</td>
<td>CONNECTIONS_FAST_SIM</td>
<td>C++</td>
<td>1,027</td>
<td>n/a</td>
</tr>
<tr>
<td>Host Code</td>
<td>CONNECTIONS.Accurate_sim</td>
<td>C++</td>
<td>6,455</td>
<td>n/a</td>
</tr>
<tr>
<td>Fast ISS (QEMU)</td>
<td>CONNECTIONS.Accurate_sim</td>
<td>C++</td>
<td>7,272</td>
<td>n/a</td>
</tr>
<tr>
<td>RTL</td>
<td>CONNECTIONS.Accurate_sim</td>
<td>C++</td>
<td>97,329</td>
<td>+/- 1%</td>
</tr>
<tr>
<td>RTL</td>
<td>RTL</td>
<td>RTL</td>
<td>(est) 2,600,000</td>
<td></td>
</tr>
</tbody>
</table>
System Verification With MatchLib

- MatchLib enables earlier verification at the system level
  - Verify what you build
- A processor can be brought into the simulation in several forms
  - Host Code Execution
  - Fast Instruction Set Simulator
  - RTL
- Enables fast functional verification, and slower throughput accurate verifications
  - Much faster and earlier than possible with traditional design cycles

Code available at: [https://github.com/hlslibs/ac_ml/tree/master/designs/HLS_SEMINAR_2021/system_design](https://github.com/hlslibs/ac_ml/tree/master/designs/HLS_SEMINAR_2021/system_design)
Thank You

Russell.Klein@Siemens.com
https://github.com/russ-klein
Questions or Comments?

?? || //