SoC Chapter 5: ESL: Electronic System Level Modelling

Modern SoC Design on Arm
End-of-Chapter Exercises

Q1 TLM Speedup

Estimate the number of CPU instructions executed by a modelling workstation when net-level and TLM models alternatively simulate the transfer of a frame over a LocalLink interface.

Answer:

A lot of assumptions are needed to answer this question, but the key aspect is to compare the differences.

We'll assume an 8-bit wide bus carrying a frame of 512 bytes. The TLM approach becomes even better with larger words or lengths.

For the TLM call, we further assume that a pointer is passed to an array buffer already containing the data. Blocking TLM coding involves a function call and return with several arguments. This will take no more than 100 instructions. An increment of a loosely timed runahead value and a quantum keeper check might add a few more instructions in the normal case. At the end-of-quantum time, the scheduler has some non-trivial work to do, but this is comparable in complexity to handling just one net for one byte of the frame in an EDS (event-driven simulation) approach.

Blocking TLM requires two phases and may have to make several attempts (e.g., 10) as it implements the flow control, so could be 20 times slower than non-blocking TLM. If it has to try an enormous number of times, it could become inefficient.

A net-level simulation could be a fast, cycle-accurate C model, as generated by VTOC or Verilator. Or it could have event-level timing for each net of the bus. In Verilog, a bus is typically a 'vector', with all bits being handled in parallel, so bus width has no impact in either case (C model or RTL simulation). Nonetheless, the net-level simulator has to model the transfer of each byte/word of the frame and so will have a factor of x512 slow down arising from that. Moreover, it must also model the handshake nets, which, with the standard synchronous interface of LocalLink, multiplies the number of signals involved by 3. Moreover, the behaviour of the serialiser and deserialiser logic at each end of the link must also be simulated, whereas this can be implicit in TLM modelling. This adds perhaps another factor of 3 to 10 in complexity.

Totting all this up, the net-level model could have 3x512x3 more events to consider than a TLM model. i.e. at least 5000 times more work. If this work is scheduled at compile time by Verilator, the instructions used per event might be about 5, making 25000 times slower. If simulated in an RTL simulator with a 4-value logic system, we can estimate a further factor of 5 to 100x slow down.

When some of the further exercises are complete, timing the resulting simulations will give a firm idea of relative instruction count.

Q2 Net-level and TLM SystemC coding

Using the additional materials from the `four-phase` folder, perform a net-level simulation of the source and sink. Then code, in SystemC, a net-level FIFO to go between them and modify the test bench to include it. Finally, write and deploy a transactor to the TLM-1 style sink that is also provided.

Note: You may want to look at the Toy ESL exercises in parallel with this exercise. Also, the keyword virtual was missing in the original definition of the TLM interface in this question. It should read:

```
class  simple_tlm1_blocking_sink_if
{
public:
  virtual void putbyte(sc_uint<8> data) = 0;
};
```

Answer:

The provided files `fourphase_netlevel_src` and `fourphase_netlevel_sink` implement a net-level direct connection between the source and sink. The Makefile may need adjusting according to where SystemC has been installed.

Twice as many top-level nets are needed when the FIFO is present. I've duplicated the existing ones, with suffix 1. The significant lines needed in the testbench to instantiate a FIFO are:

```
  sc_signal<bool> strobe1, ack1;
  sc_signal<sc_uint<8> > data1;
  FOURPHASE_NETLEVEL_FIFO *fifo;
```

and

```
  fifo = new  FOURPHASE_NETLEVEL_FIFO("fifo");
  fifo->strobe_in(strobe);      fifo->ack_in(ack);       fifo->data_in(data);
  fifo->strobe_out(strobe1);    fifo->ack_out(ack1);  fifo->data_out(data1);
```

Then add `#include "fourphase_netlevel_fifo.h"` and change the sink bindings to use the FIFO outputs.

A synchronous FIFO is certainly a bit easier to code than the required asynchronous solution. A low-level implementation using C-elements may be appropriate, but verbose. Here is the code for a higher-level implementation:

```
#ifndef  FOURPHASE_NETLEVEL_FIFO_H
#define  FOURPHASE_NETLEVEL_FIFO_H

// Four-Phase Asynchronous Handshake using SystemC

// This folder contents (C) DJ Greaves 2021. You may use them as you
// wish, without waranty or license from the author, provided this
// copyright message is preserved.

#include <systemc.h>

// Four-phase data fifo
SC_MODULE(FOURPHASE_NETLEVEL_FIFO)
{
  sc_in<bool>  strobe_in;    sc_out<bool>  ack_in;    sc_in<sc_uint<8> >  data_in;
  sc_out<bool> strobe_out;   sc_in<bool>  ack_out;    sc_out<sc_uint<8> >  data_out;
# define   FIFO_SIZE   (4)
  int in_ptr = 0, out_ptr = 0, n_items = 0;
  sc_uint <8 > data [ FIFO_SIZE ];

  // These two events model the forward and reverse combinational paths through the component.
  // They would not be needed in a synchronous FIFO, since everything can be operated off the clock method.
  sc_event fwd_prodder , rev_prodder;

  void  enqueue ()
  {
    ack_in = false;
    while ( true )
      {
        while  (true)
          {
            if ( strobe_in && n_items < FIFO_SIZE) break;
            wait ();
          }
        std::cout << "enqueue:_" << n_items << "_in=" << in_ptr << "_out=" << out_ptr << "\n";
        wait(2 , SC_NS );
        data[in_ptr] = data_in;
        in_ptr = (in_ptr + 1) % FIFO_SIZE;
        n_items += 1;
        fwd_prodder. notify ( SC_ZERO_TIME );
        ack_in = true;
        wait(2 , SC_NS );
        while ( strobe_in ) wait ();
        wait(2 , SC_NS );
        ack_in = false;
      }
  }


  void  dequeue ()
  {
    strobe_out = false;
    while ( true )
      {
        while (n_items == 0) wait ();
        wait(2 , SC_NS );
        data_out = data [ out_ptr ];
        out_ptr = (out_ptr + 1) % FIFO_SIZE;
        n_items -= 1;
        rev_prodder. notify ( SC_ZERO_TIME );
        strobe_out = true;
        wait(2 , SC_NS );
        while (! ack_out) wait ();
        wait(2 , SC_NS );
        strobe_out = false;
        while (ack_out) wait ();
      }
  }

  SC_CTOR ( FOURPHASE_NETLEVEL_FIFO )
    {
      SC_THREAD ( enqueue );
      sensitive << strobe_in << rev_prodder;

      SC_THREAD ( dequeue );
      sensitive << ack_out << fwd_prodder;
    }
};

// eof
#endif
```

To make the receiving sink a little less eager, so that the FIFO sometimes has some occupancy, modify its body with something like:

```
int random_wait_time = 1 + (data.read() % 10);
wait(random_wait_time, SC_NS);
```

A transactor to the TLM sink is included in the following testbench:

```
// Four-Phase Asynchronous Handshake using SystemC

// This folder contents (C) DJ Greaves 2021. You may use them as you
// wish, without waranty or license from the author, provided this
// copyright message is preserved.

// https://www.cl.cam.ac.uk/~djg11/wwwhpr/fourphase/fourphase.html

// Here we solve the second part of the exercise from Chapter 5,
// where we add a transactor and use the TLM-style sink.

#include <systemc.h>
# include "fourphase_netlevel_src.h"

// # include "fourphase_netlevel_sink.h"

#include "fourphase_netlevel_fifo.h"

#include "fourphase_tlm1_sink.h"


SC_MODULE(XACTOR_SIMPLEX_4PT_TO_BTLM1) // This is the transactor.
{
  sc_in<bool> strobe_in; sc_out<bool> ack_in;   sc_in<sc_uint<8> > data_in;
  simple_tlm1_blocking_sink_if *tlm_out;

  void operate()
  {
    ack_in = false;
    while (true)
      {
        while (!strobe_in) wait();
        wait(2, SC_NS);
        ack_in = true;
        wait(2, SC_NS);
        tlm_out->putbyte(data_in);
        while (strobe_in) wait();
        wait(2, SC_NS);
        ack_in = false;
      }
  }

  SC_CTOR(XACTOR_SIMPLEX_4PT_TO_BTLM1)
    {
      SC_THREAD(operate);
      sensitive << strobe_in;
      // Null
    }

  void bind_out(simple_tlm1_blocking_sink_if *p)
  {
    tlm_out = p;
  }
};

SC_MODULE(BASIC_NETLEVEL_TESTBENCH)
{
  sc_signal<bool> strobe, ack;
  sc_signal<sc_uint<8> > data;

  FOURPHASE_NETLEVEL_SRC *src0;

  sc_signal<bool> strobe1, ack1;
  sc_signal<sc_uint<8> > data1;
  FOURPHASE_NETLEVEL_FIFO *fifo;
  XACTOR_SIMPLEX_4PT_TO_BTLM1 *xactor0;
  FOURPHASE_TLM_SINK *sink0;

  SC_CTOR(BASIC_NETLEVEL_TESTBENCH): strobe("strobe"), ack("ack"), data("data")
    {
      src0 = new FOURPHASE_NETLEVEL_SRC("src0");
      fifo = new FOURPHASE_NETLEVEL_FIFO("fifo");
      xactor0 = new XACTOR_SIMPLEX_4PT_TO_BTLM1("xactor0");
      sink0 = new FOURPHASE_TLM_SINK("tlm_sink0");

      src0->strobe(strobe);        src0->ack(ack);        src0->data(data);
      fifo->strobe_in(strobe);     fifo->ack_in(ack);     fifo->data_in(data);
```

```
        fifo -> strobe_out( strobe1 );      fifo -> ack_out( ack1 );     fifo -> data_out( data1 );
        xactor0 -> strobe_in ( strobe1 );   xactor0 -> ack_in ( ack1 );   xactor0 -> data_in ( data1 );
        xactor0 -> bind_out( sink0 );
      }
};


int sc_main(int argc, char *argv[])
{
  BASIC_NETLEVEL_TESTBENCH *toplevel = new BASIC_NETLEVEL_TESTBENCH ("toplevel");
  sc_trace_file *wf = sc_create_vcd_trace_file ("tracefile");
  sc_trace (wf, toplevel->strobe, "strobe");
  sc_trace (wf, toplevel->ack, "ack");
  sc_trace (wf, toplevel->data, "data");

  sc_start (100.0, SC_MS);
  sc_close_vcd_trace_file (wf); // wf.close ();
std::cout << "Testbench finished at " << sc_time_stamp () << "\n";
  return 0;
}
// eof
```

Q3 Virtual Platforms

If access to the real hardware is not yet possible, discuss how development, debugging and performance analysis of device driver code can be facilitated by a virtual platform. What might be the same and what might be different?

Answer:

Given that simulation of the RTL for the real hardware is too slow, a virtual platform is needed to run and develop the low-level software. Three main forms of virtual platform are:

1. An instruction set simulator, using Gem5, QEMU, SystemC or similar.
2. A fast, cycle accurate C++ model extracted from the RTL using Verilator, VTOC, Carbon or similar tool.
3. An FPGA implementation of the (lightly modified) design RTL.

Sometimes hybrids of these are used. Speed of execution will be the principal variation, but debuggability, protection of IP and response time after RTL changes also vary.

Certain devices and hence device drivers need to operate in real-time for proper development, such as when controlling a thermal printer.

Others can work with a synthetically slowed down model of the device. For instance, a file of ADC samples from the IF (intermediate frequency) ADC of a digital radio receiver can be read by the model at any speed. Radio reception is simplex, which makes this easy. But reactive systems, such as those involving duplex communication or sensor/actuator plant control cannot always be realistically slowed down.

Debugging accessibility is often much easier on a virtual platform. For instance, adding profiling code to a real system normally changes its behaviour slightly and sometimes this can upset the metric or artefact (e.g., a race condition) of interest.

Q4 Toy ESL Demos

In the additional materials toy–esl folder, work through the four SystemC TLM coding examples in which processors access memory. (Note, for ease of getting started and debugging, this material does not use TLM 2.0 sockets. It essentially does the same thing as the Prazor system, but at a much more basic level.)

Answer:

Emphasize to the students the equivalences and differences between the different models of the same sort of system. An interesting aspect is how well arbitration is modelled, once there is more than one initiating processor. Does the system properly model the contention delay arising from shared use of a resource? Is the average throughput correct even if the delay distribution is unmodelled? Is this a problem for real-time systems (yes — developed in the next question).

Q5 Queueing Delay Modelling

A NoC switching element is modelled using SystemC TLM. What mechanisms exist for capturing the queuing delay if passthrough TLM sockets are to be used in the NoC element model?

Answer:

A SystemC passthrough TLM socket enables the same thread that is invoking the transport() method of the model of the element to invoke the transport() method of the next component the flit/packet is being passed to.

The thread could use a succession of wait calls, inside the element, looping until the detailed condition for it to be dequeued is satisfied. The thread timing and ordering would be completely accurate, but the initiator must be aware that the thread it sends is likely to be yielding to the SystemC scheduler. Similarly, if an estimator is modelling the local queuing delay at the element, the thread could yield for that number of clock cycles. This would be more efficient since the 'detailed condition' does not have to be periodically checked.

Using loosely timed TLM, the thread will be passed on immediately to its next destination, but with the estimate of how long it is supposed to have been waiting added to its delay parameter. This is the situation that really requires passthrough sockets. It is efficient since there is no interaction with the scheduler (unless the accumulated delay has exceeded the LT quantum).

Approaches that simply add the expected average delay to each transaction will clearly not give any insight into the distribution of delays. Generally, the average overall performance should be modelled correctly. But systems that have to meet hard real-time performance targets cannot be modelled this way since these targets are often expressed in terms of the tail of the distribution.

A well-known example is that for real-time audio over a computer network, it is the delay of the 99th percentile of traffic that is important: the occupancy and size of the playout buffer must be dimensioned accordingly. But this is not highly relevant to TLM modelling of SoCs since the timescales are orders of magnitude different: audio and network delays are tens of milliseconds whereas TLM quanta are microseconds. However, simple Markovian queuing models (exponentially spaced arrivals) give the wrong answer when traffic is correlated or less random in other ways (e.g., convoy effect). A detailed queuing model should be run occasionally to pro- vide confidence that the simplified models are behaving properly.

Q6 Rentian Wire Length Estimation

Assuming a typical Rent value, using a spreadsheet or simple program, tabulate the average wiring length versus number of hierarchy levels crossed for a transactional interface in a typical SoC.

Which of the following do you need to assume: total number of hierarchy levels, average number of child components to a component, variation in area of a component, Rentian exponent, average number of connections to a component and percentage of local nets to a component? Obtain a numerical figure for the partial derivatives of the result with respect to each of your assumptions. Which is the most important?

Answer:

This has been the topic of hundreds of papers since Rent's first observations, so only a broad-brush answer can be expected here. There are numerous synthetic circuit generators listed on the relevant Wikipedia page for Rent's Rule, but probably it is more informative to write one from scratch.

Let us assume a module structure, of $N = 8$ levels, where each level of the hierarchy is similar in structure to its children, except for the bottom level. At the bottom level of this tree we have logic gates, interconnected at the net level. A program in FSharp is available on www.cl.cam.ac.uk/users/djg11/rents-rule-explorer

Note that a naive fractal model, with homogeneous wiring and regular structure at each level with give a Rent exponent of unity, since it is self-similar between levels. The key thing to include, to achieve the lower exponents encountered in real-world designs (that have some design/engineering behind them) is the ratio of terminal contacts to gates needs to get smaller as the number of gates gets larger (see *The Interpretation and Application of Rent's Rule* Christie& Stroobandt 2000). This can be programmed and the exponent will emerge, but in the example code, the Rent formula is used explicitly with parameters `rent_alpha` and `rent_exponent`. Hence the average number of contacts to a component and the number of nets is emergent (given an average fan out figure).

The model averages over 20 random runs for each test and the normalised standard deviation (i.e. divided by the mean) experienced across runs is reported as one confidence indication. These figures are all in the range 10 to 20 percent, which is find for this exercise.

The program then perturbs each of its numeric assumptions by 5 percent to give the following derivative estimates.

Parameter (p)	$\frac{\partial}{\partial p}$ Silicon Area	$\frac{\partial}{\partial p}$ Wiring Length	Comment
Rent_alpha	0	**0.81**	Greater inter-component wiring increased overall net length.
Rent_exponent	0	**5.8**	The same effect, exponentially magnified!
Rent_netbeta	0	1.0	Completely as expected.
Average_net_fanout	0	-0.47	As expected: increas$_p$ed fanout is more contacts per net.
Area_swell	1.1	0.34	Expected unity and 2, so ok.
Leaf_area_variation	0.00014	-0.0024	Leaf cells already varied 3 to 1 in size, so additional perturbation is irrelevant.
Relative_deviation	0.89	1.1	Large absolute value here reflects model non-linearities!
Average_no_children	10	12	Correct, approximately N.

1 It assumes there are sufficient metal layers for the quantity of wiring to not affect area (i.e. area is determined only by the active silicon layer) (but there is an Area_swell. per hierarchic layer), hence the first four partials for area come out as zero.

Further, assume that interconnections at all higher levels are transactional busses whose length we must measure. Let the average number of children to a component be C, then the system size is C^N. We are asked to tabulate the average bus length as we change the number of hierarchy levels.

For a robust estimation and modelling system, we desire low partials for each assumed parameter, since we typically won't have much confidence in them. Most in the table are small, which is good.

The variation in size with the average number of children to a component looks undesirably massive at 10 or 12: but this is only to be expected: we have an exponential response to this parameter and behaviour at each level is correlated, giving a systematic increase in its number of children. That parameter was deliberately modelled that way to show how the derivatives can demonstrate a poor modelling approach. If an uncorrelated approach were used, as with the leaf area variation, a close to zero derivative should be reported.

The significant results in the table are the top two partials, highlighted in bold font. This demonstrates the sensitivity of wiring length to the Rent exponent. Both parameters in the formula alter the ratio of local nets to external contacts for a given component: when there are more contacts, more of the nets go on "foreign trips" outside the current component. Hence, we conclude synthetic models are overly sensitive to such parameters and are unlikely to reflect reality. On the other hand, wiring length prediction based on unplaced, real component hierarchies can be far more accurate, since they do not explicitly evaluate the Rent formula.

The wiring at the bottom of the model between individual gates is generally not transactional. By transactional, in this book, we mean interconnect that conveys data packets using handshake. We assume perhaps 75 per cent of the nets in the top half of the design hierarchy are accordingly transactional. The remainder are for power, interrupts, clocks and so on. A table of their average length versus number of levels crossed is included in the online folder, with the program.

Q7 Static vs Dynamic Power Analysis

To what extent can a simple spreadsheet or static analysis determine the average activity ratio for a net or subsystem? What further information is needed? Given activity numbers, what further information may be needed to generate an idealised mapping of subsystems to power domains? What other considerations should be applied to determine a practical power domain mapping?

Answer:

We should first be careful to disambiguate two meanings of the word 'dynamic' here. Dynamic analysis generally refers to instrumentation of a running system or simulation, whereas static analysis generally refers to looking at the system structure. This is the distinction used in the world of software and compilers. For digital hardware, the term dynamic power refers to electricity used as nets switch logic values (charging capacitors and short-circuit currents) and static power refers to the power used while no transitions are occurring (arising from leakage and resistive pull-ups).

Static power use can be estimated from the post-synthesis inventory of components and their data sheets (although power gating also has to be factored in). This is relatively straightforward.

For dynamic power, the number of transitions on a net need to be known, along with its capacitance. The precise number of nets is only known post RTL synthesis, but pre-synthesis estimation techniques could also be applied. Sticking with post synthesis approaches, the number of transitions can either be estimated or measured in simulation. The precise net capacitances are only known after place and route.

For ESL models that do not have any RTL, the approaches used by the TLM POWER3 library to estimate bus length and measure transitions on bust nets provide a reasonable approximate approach, especially taking into account that it is the longer nets between components that have the most capacitance.

With RTL, a reasonable compromise is to synthesize the RTL to the net level, use relevant simulation testbenches to exercise the design to give SAIF-style activity and estimate net capacitances based on a Rent-style analysis. A large spreadsheet listing all the nets and their number of transitions and estimated capacitance can then give energy and power results.

For other approaches, look up vectorless power estimation.

Q8 Transactional Order

Give simple examples where out-of-order transaction processing arising from the loosely timed modelling approach causes and does not cause functional accuracy errors. Are transaction counts likely to be wrong under loose timing?

Answer:

The main sequential consistency problem arises with low-level access to a shared register or RAM. For instance, on hardware that is supposed to support sequential consistency, a WaW situation is properly resolved using the last-writer-wins rule, but with out-of-order transactions, the last writer might be different from reality, leading to a functional accuracy problem. But if the hardware were not guaranteeing this property, there is no problem. Equally, operations on different registers or FUs that have no shared state will follow an accurate state trajectory for any interleaving that occurs.

When the hardware requires a fence instruction to enforce sequential consistency, this can be modelled in the LT model by making all LT threads perform a re-synch. A rendezvous barrier in the thread library can be invoked as needed (although the author has no first-hand experience of a multicore SystemC implementation).

In all cases, the transaction count should be accurate, in that it models a feasible execution schedule, but control flow divergence arising from the sequential consistency changes (e.g., additional retries when a modelled program claims a mutex lock) might occur.