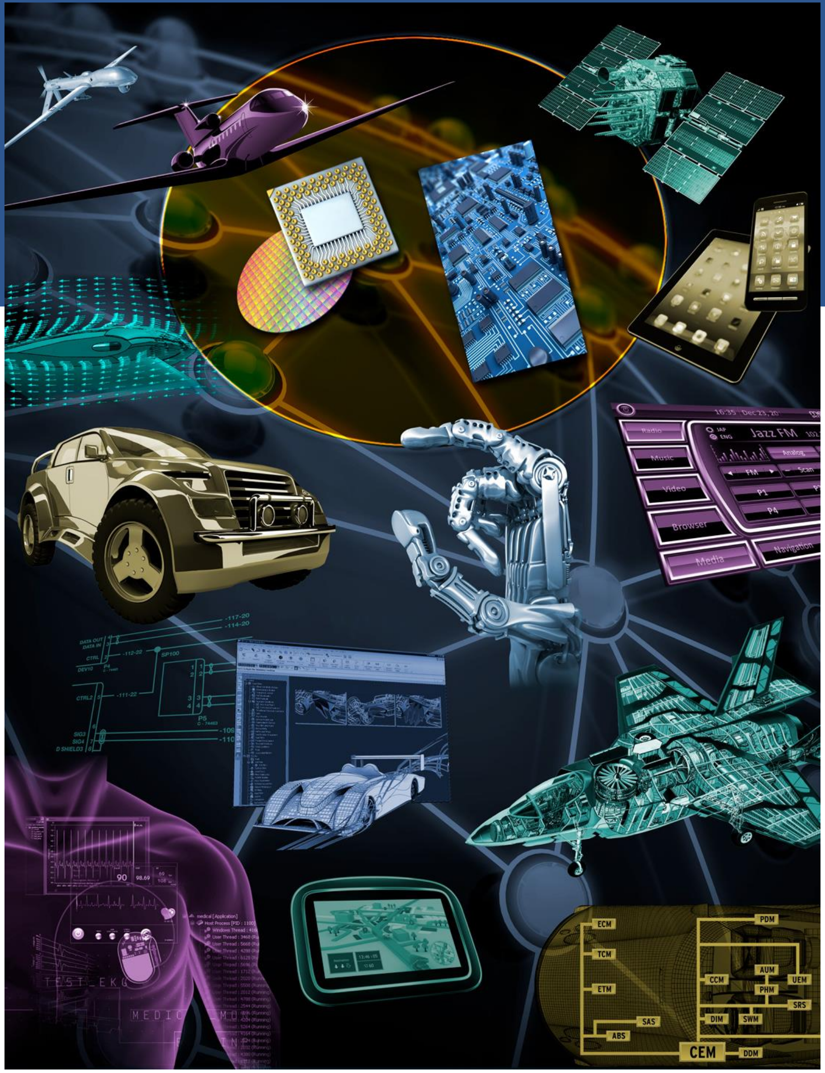


Catapult HLS Modeling using SystemC and Matchlib

Stuart Swan

HLS IP/Platform Architect

11 September 2019

Mentor®

A Siemens Business

Why use HLS?

- Automatic scheduling, resource sharing, pipelining
- Easier retargeting to different silicon technologies
- More abstract design models
- Enable more DV and debugging to occur at higher abstraction level
- Make architectural exploration feasible

Why use C++ for HLS?

- Templates, classes, inheritance, …
- Easy integration of existing C/C++ models
- Simulation speed
- …

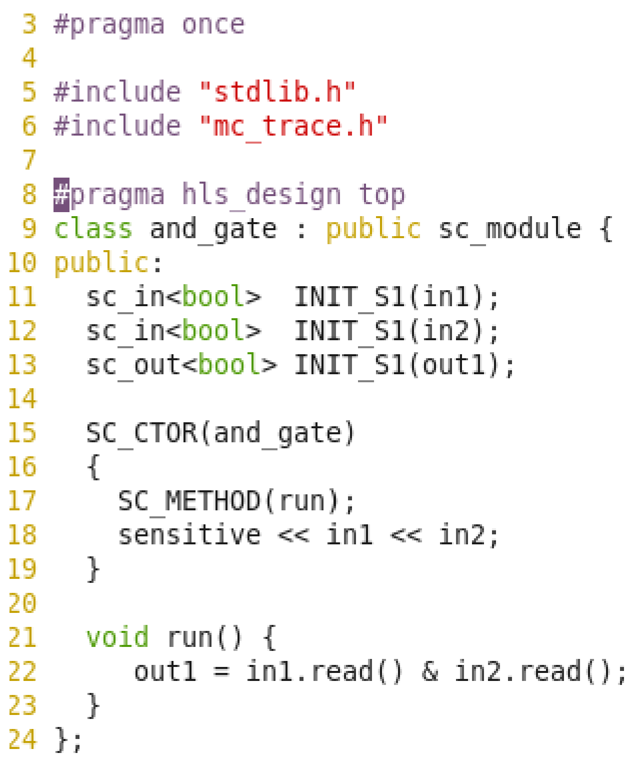
Why use SystemC + Matchlib for HLS?

- SystemC provides HDL semantics on top of C++ language
  - time, module structure, hierarchy, channels, HW semantics, resets, signals
- SC enables time based behaviors to be cleanly modeled and verified prior to synthesis (as compared to pure C++ HLS)
- SC integrates nicely with HDL simulators
- Matchlib enables models to be “throughput accurate” pre-HLS

How does Catapult SC synthesis differ from C++ synthesis ?

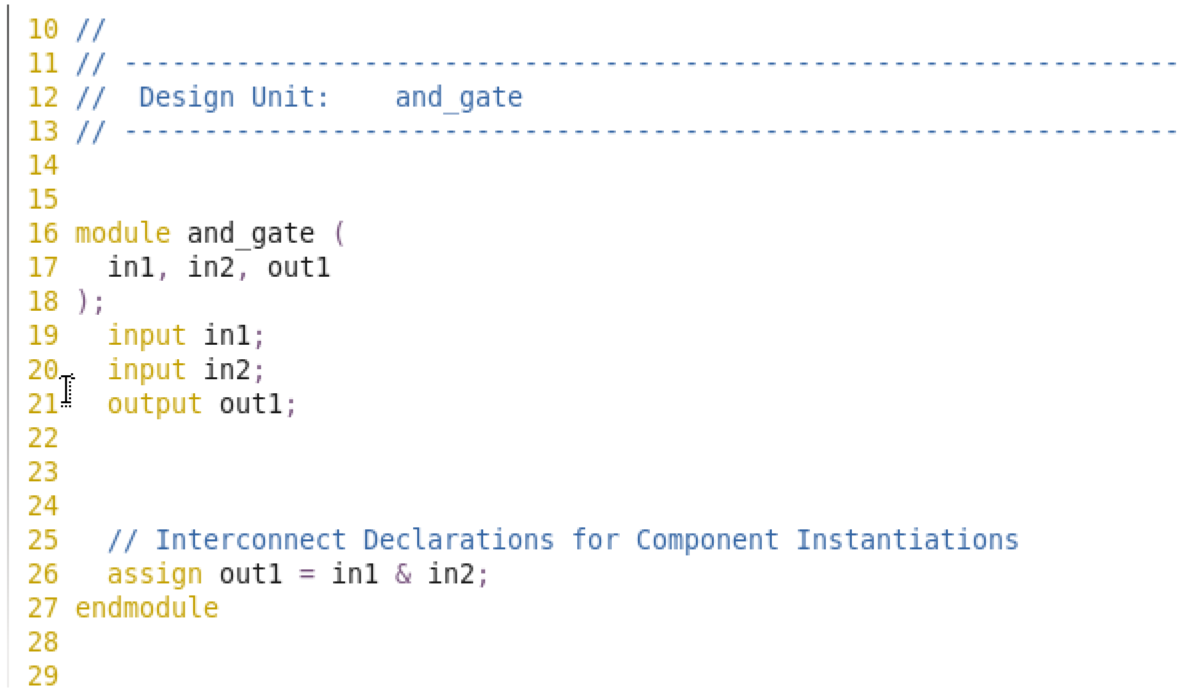
- Primary difference is in input language
- Once code is read in by Catapult, 90% of flow and usage is same.

SC Combinational Process



```
#pragma once

#include "stdlib.h"
#include "mc_trace.h"

#pragma hls_design top
class and_gate : public sc_module {
public:
  sc_in<bool>  INIT_S1(in1);
  sc_in<bool>  INIT_S1(in2);
  sc_out<bool> INIT_S1(out1);

  SC_CTOR(and_gate)
  {
    SC_METHOD(run);
    sensitive << in1 << in2;
  }

  void run() {
     out1 = in1.read() & in2.read();
  }
};
```

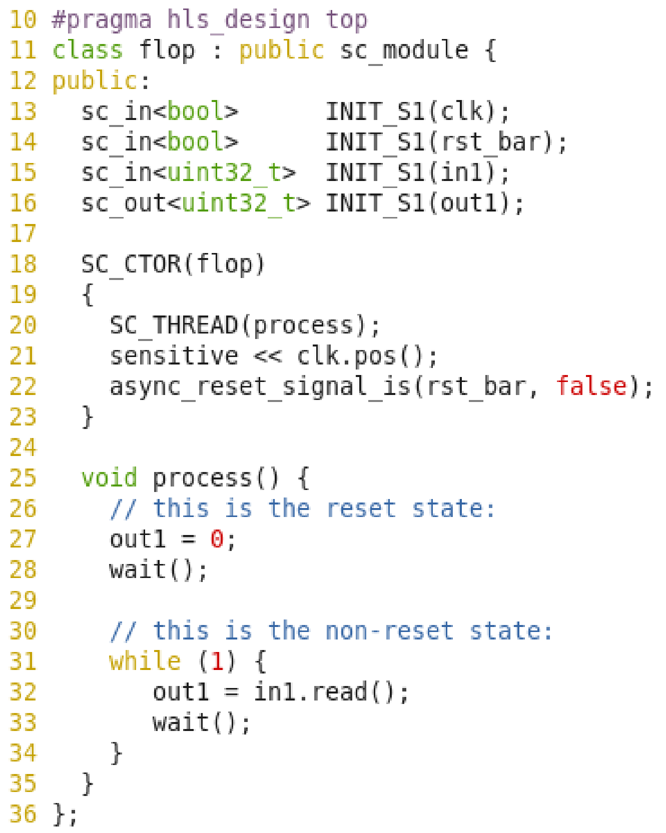
HLS Input



```
//
// ------------------------------------------------------------------
//  Design Unit:    and_gate
// ------------------------------------------------------------------


module and_gate (
  in1, in2, out1
);
  input in1;
  input in2;
  output out1;



  // Interconnect Declarations for Component Instantiations
  assign out1 = in1 & in2;
endmodule


```

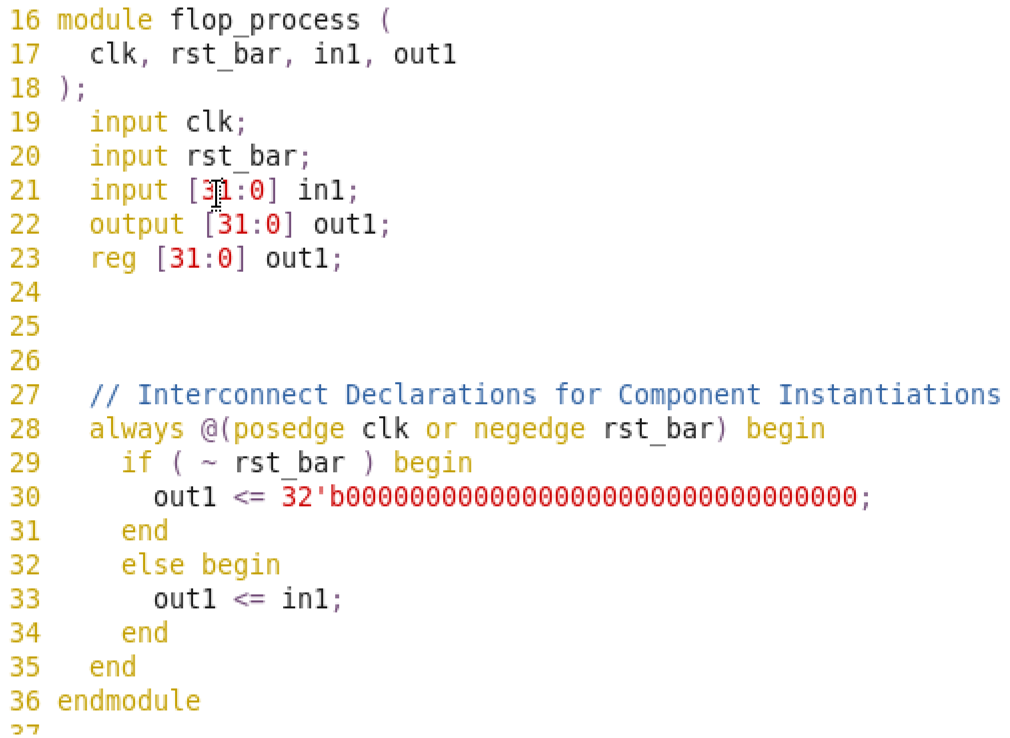
HLS Output

SC Sequential Process



```
#pragma hls_design top
class flop : public sc_module {
public:
  sc_in<bool>       INIT_S1(clk);
  sc_in<bool>       INIT_S1(rst_bar);
  sc_in<uint32_t>  INIT_S1(in1);
  sc_out<uint32_t> INIT_S1(out1);

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

  void process() {
    // this is the reset state:
    out1 = 0;
    wait();

    // this is the non-reset state:
    while (1) {
       out1 = in1.read();
       wait();
    }
  }
};
```

HLS Input



```
module flop_process (
  clk, rst_bar, in1, out1
);
  input clk;
  input rst_bar;
  input [31:0] in1;
  output [31:0] out1;
  reg [31:0] out1;



  // Interconnect Declarations for Component Instantiations
  always @(posedge clk or negedge rst_bar) begin
    if ( ~ rst_bar ) begin
      out1 <= 32'b00000000000000000000000000000000;
    end
    else begin
      out1 <= in1;
    end
  end
endmodule
```

HLS Output

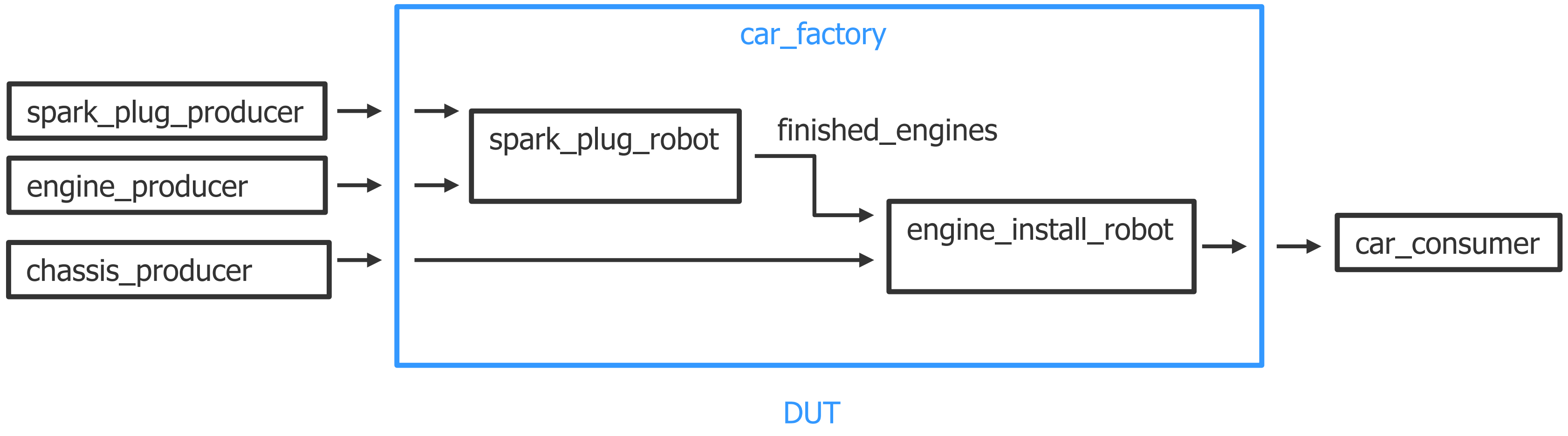
Important to understand how HLS sees your SC model:

- Set of processes communicating **only** thru signals (sc_signal<>)
- For synthesis purposes, hierarchy is irrelevant (it is preserved in the RTL however)
- HLS synthesis occurs on each process one at a time, in complete isolation
  - No analysis/optimizations across processes
  - Huge designs can be synthesized thru HLS, as long as each process is not too large
- Each combinational process (SC_METHOD) becomes combinational logic in RTL
- Each sequential process becomes exactly one FSM + Datapath in RTL

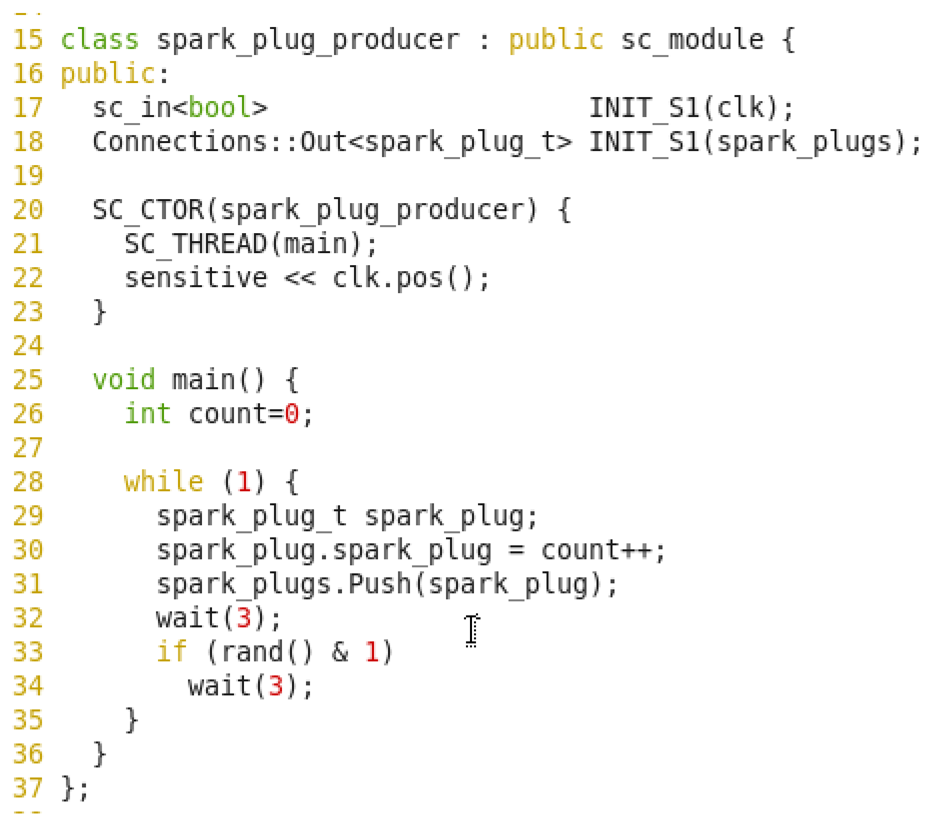
How HLS sees your SC model (continued)

- Primary optimizations done by HLS are in construction of the FSM and Datapath (e.g. resource sharing).
- Unlike RTL synthesis, HLS is (usually) allowed to add clock cycles (latency) into design to improve QOR
  - e.g. often loops are pipelined to maintain thruput while latency is added by HLS
- Properly constructed models and proper usage of HLS should show no functional differences (aside from latency differences) pre and post HLS.
- For SC HLS, reset behaviors and IO protocols are present in pre-HLS model and synthesized into RTL together with rest of model

Simple Example of HW Architectural Model using SC + Matchlib



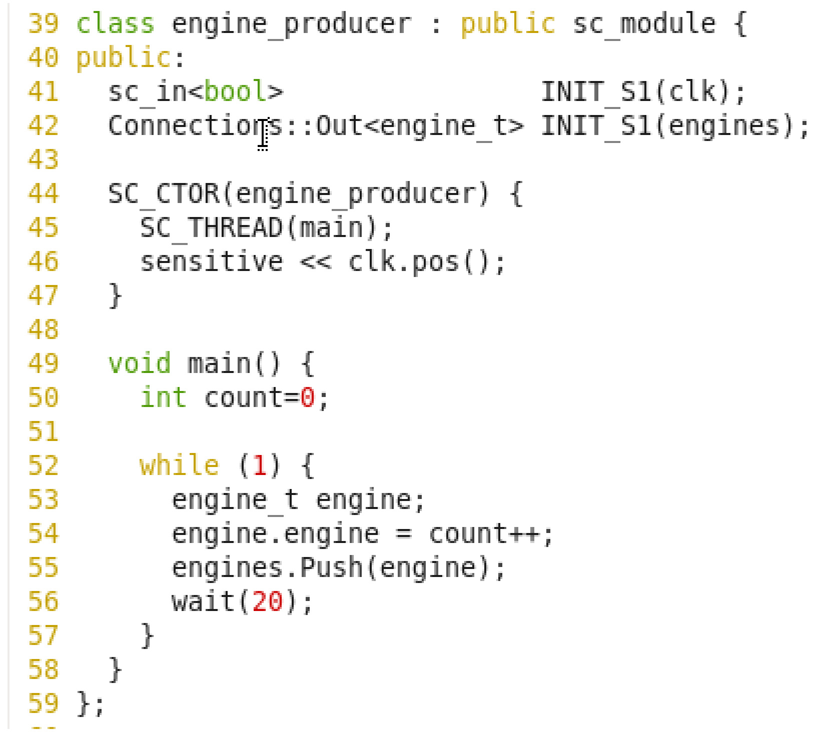
spark_plug_producer



```
class spark_plug_producer : public sc_module {
public:
  sc_in<bool>                    INIT_S1(clk);
  Connections::Out<spark_plug_t> INIT_S1(spark_plugs);

  SC_CTOR(spark_plug_producer) {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    int count=0;

    while (1) {
      spark_plug_t spark_plug;
      spark_plug.spark_plug = count++;
      spark_plugs.Push(spark_plug);
      wait(3);
      if (rand() & 1)
        wait(3);
    }
  }
};
```

produces new spark_plug every 3-6 seconds

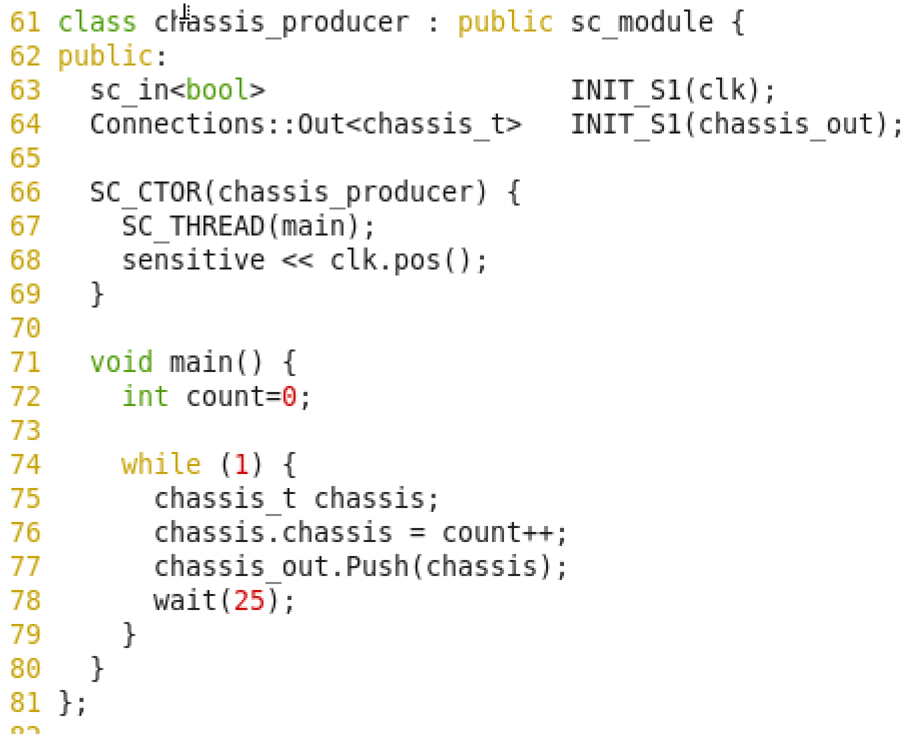
engine_producer



```
class engine_producer : public sc_module {
public:
  sc_in<bool>                INIT_S1(clk);
  Connections::Out<engine_t> INIT_S1(engines);

  SC_CTOR(engine_producer) {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    int count=0;

    while (1) {
      engine_t engine;
      engine.engine = count++;
      engines.Push(engine);
      wait(20);
    }
  }
};
```

produces new engine every 20 seconds

chassis_producer



```
class chassis_producer : public sc_module {
public:
  sc_in<bool>                   INIT_S1(clk);
  Connections::Out<chassis_t>   INIT_S1(chassis_out);

  SC_CTOR(chassis_producer) {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    int count=0;

    while (1) {
      chassis_t chassis;
      chassis.chassis = count++;
      chassis_out.Push(chassis);
      wait(25);
    }
  }
};
```

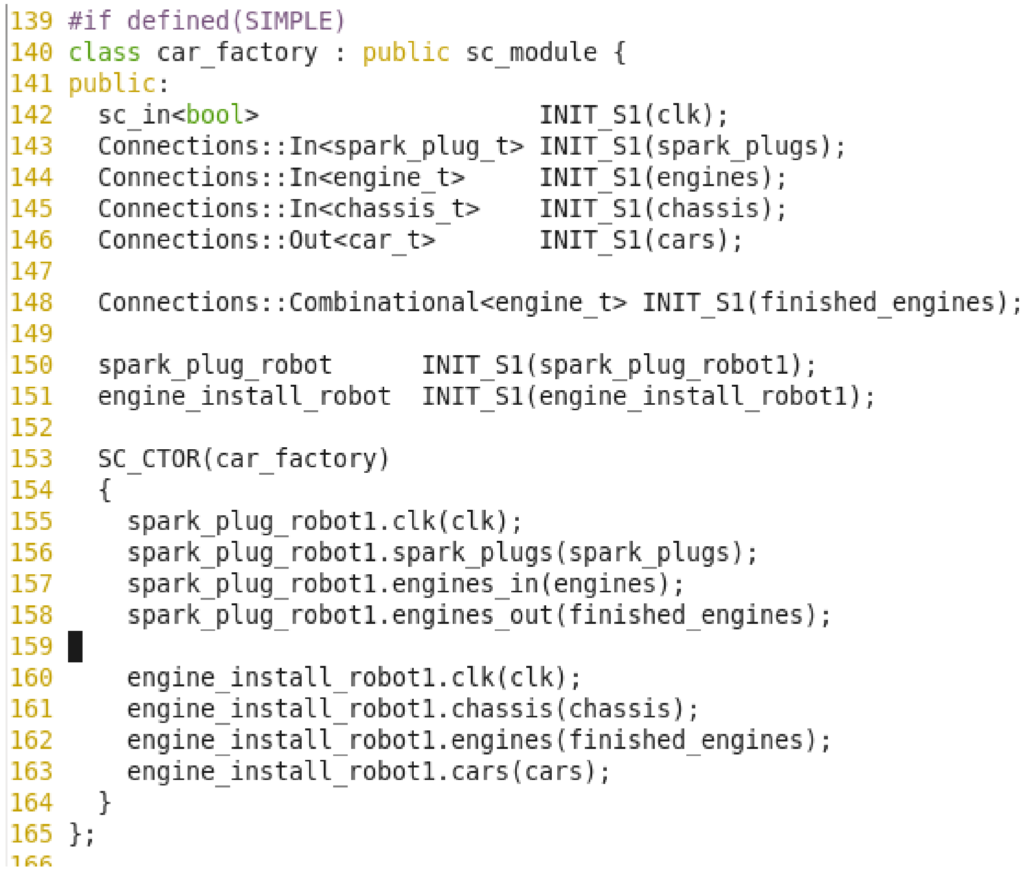
produces new chassis every 25 seconds

car_consumer

```
class car_consumer : public sc_module {
public:
  sc_in<bool>             INIT_S1(clk);
  Connections::In<car_t>  INIT_S1(cars);

  SC_CTOR(car_consumer) {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    int count = 0;
    while (1) {
      cars.Pop();
      ++count;
      LOG("got car # " << count);
      if (count == 10)
      {
        LOG("total cars produced: " << count);
        LOG("time per car: " << sc_time_stamp() / count);
        sc_stop();
      }
    }
  }
};
```

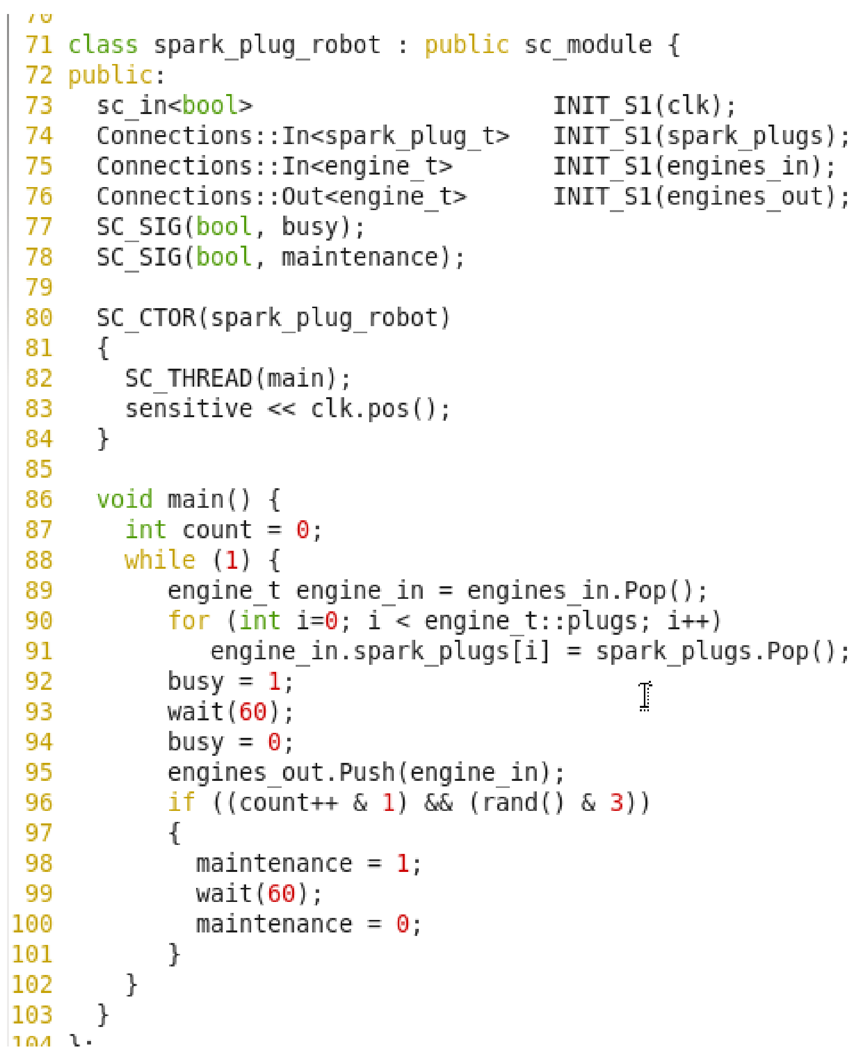
consumes cars as quickly as possible

Simple car_factory



```
#if defined(SIMPLE)
class car_factory : public sc_module {
public:
  sc_in<bool>                   INIT_S1(clk);
  Connections::In<spark_plug_t> INIT_S1(spark_plugs);
  Connections::In<engine_t>     INIT_S1(engines);
  Connections::In<chassis_t>    INIT_S1(chassis);
  Connections::Out<car_t>       INIT_S1(cars);

  Connections::Combinational<engine_t> INIT_S1(finished_engines);

  spark_plug_robot      INIT_S1(spark_plug_robot1);
  engine_install_robot  INIT_S1(engine_install_robot1);

  SC_CTOR(car_factory)
  {
    spark_plug_robot1.clk(clk);
    spark_plug_robot1.spark_plugs(spark_plugs);
    spark_plug_robot1.engines_in(engines);
    spark_plug_robot1.engines_out(finished_engines);

    engine_install_robot1.clk(clk);
    engine_install_robot1.chassis(chassis);
    engine_install_robot1.engines(finished_engines);
    engine_install_robot1.cars(cars);
  }
};
```

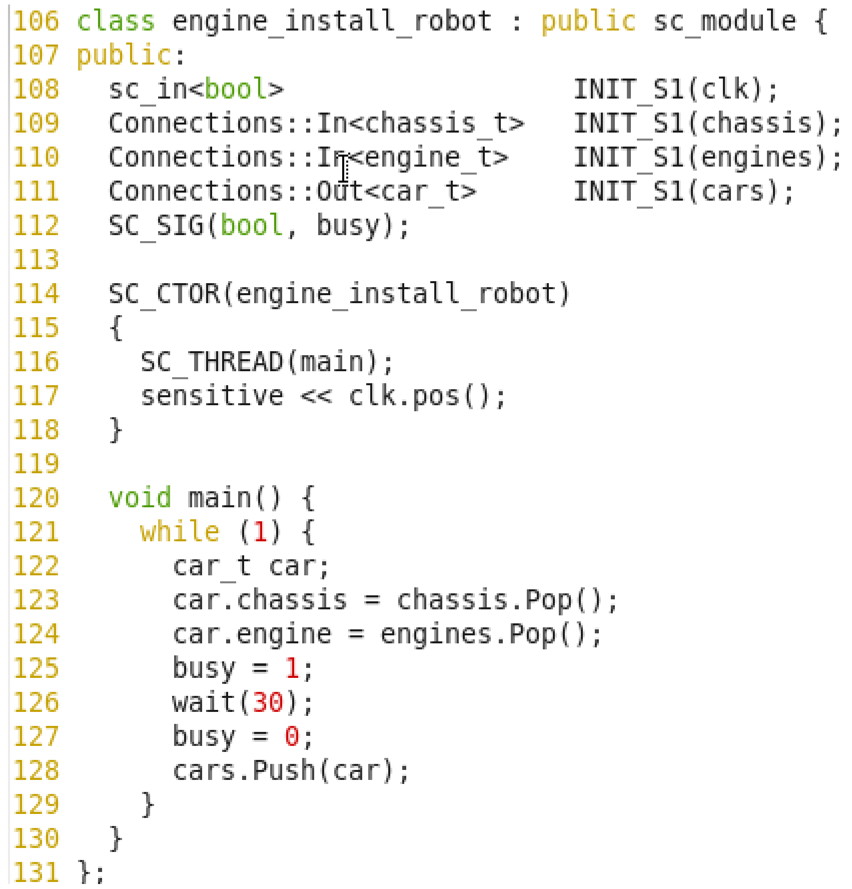
spark_plug_robot



```
class spark_plug_robot : public sc_module {
public:
  sc_in<bool>                       INIT_S1(clk);
  Connections::In<spark_plug_t>     INIT_S1(spark_plugs);
  Connections::In<engine_t>         INIT_S1(engines_in);
  Connections::Out<engine_t>        INIT_S1(engines_out);
  SC_SIG(bool, busy);
  SC_SIG(bool, maintenance);

  SC_CTOR(spark_plug_robot)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    int count = 0;
    while (1) {
       engine_t engine_in = engines_in.Pop();
       for (int i=0; i < engine_t::plugs; i++)
          engine_in.spark_plugs[i] = spark_plugs.Pop();
       busy = 1;
       wait(60);
       busy = 0;
       engines_out.Push(engine_in);
       if ((count++ & 1) && (rand() & 3))
       {
         maintenance = 1;
         wait(60);
         maintenance = 0;
       }
    }
  }
};
```

Consumes 4 spark_plugs and 1 unfinished engine
Produces finished_engine after 60 seconds
After every other engine, 75% of time
needs 60 seconds of maintenance (ie idle time)

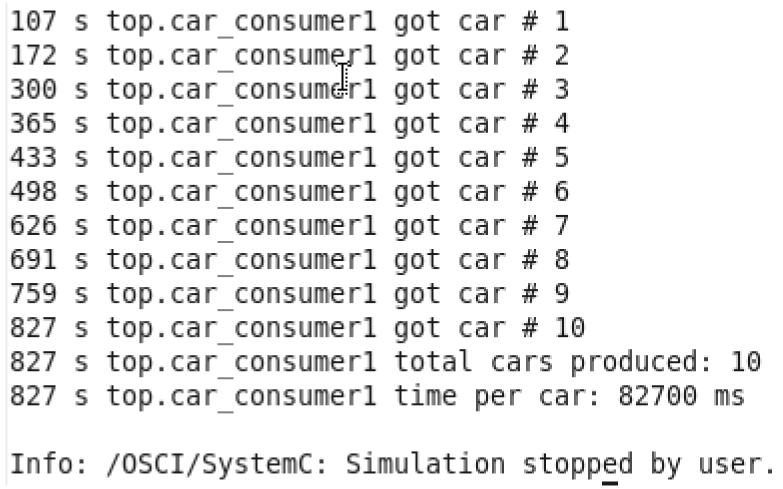
engine_install_robot



```
class engine_install_robot : public sc_module {
public:
  sc_in<bool>                  INIT_S1(clk);
  Connections::In<chassis_t>   INIT_S1(chassis);
  Connections::In<engine_t>    INIT_S1(engines);
  Connections::Out<car_t>      INIT_S1(cars);
  SC_SIG(bool, busy);

  SC_CTOR(engine_install_robot)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    while (1) {
      car_t car;
      car.chassis = chassis.Pop();
      car.engine = engines.Pop();
      busy = 1;
      wait(30);
      busy = 0;
      cars.Push(car);
    }
  }
};
```

Consumes 1 chassis and 1 finished_engine
Produces car after 30 seconds

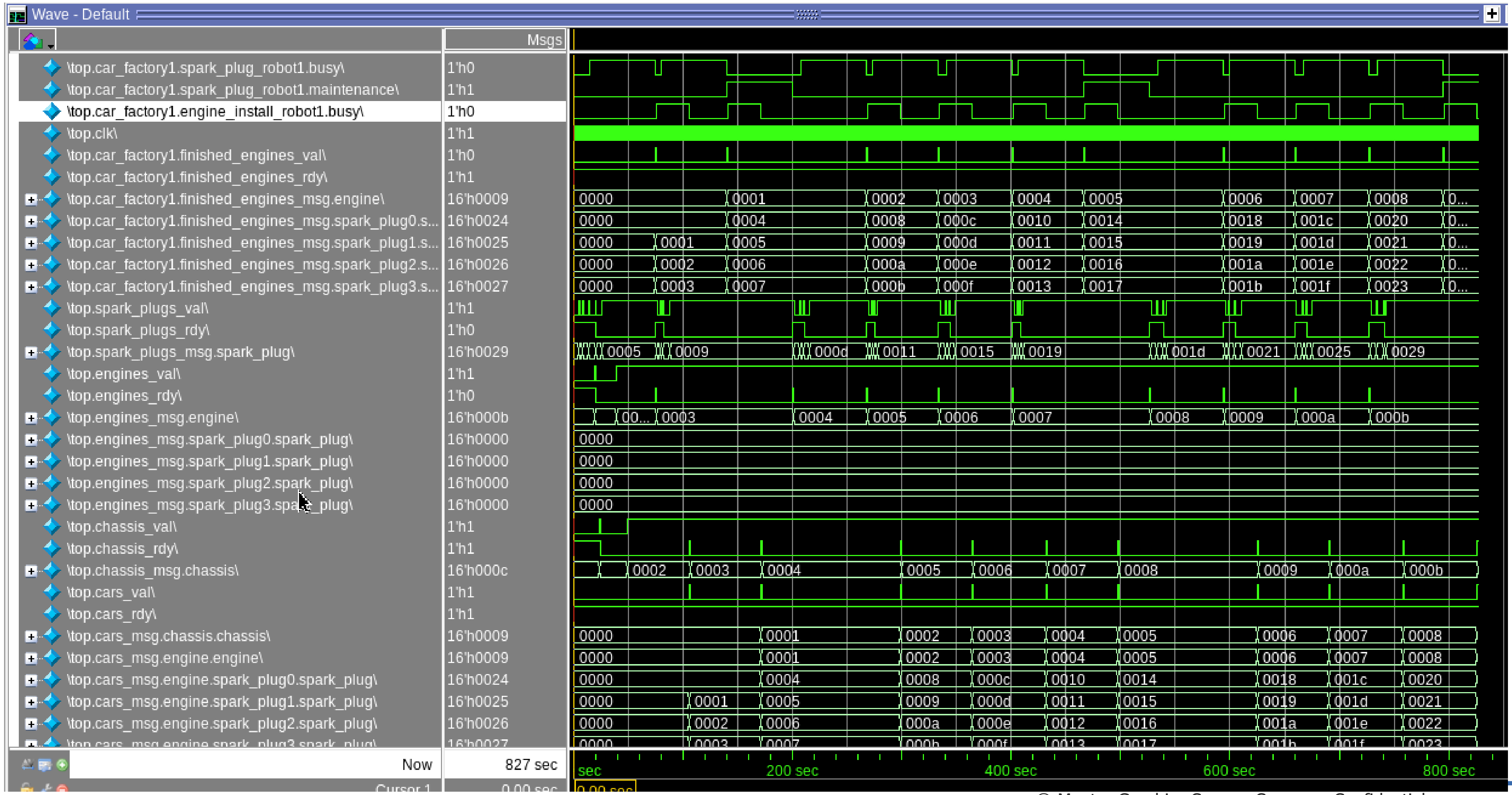
Running simple car_factory



```
107 s top.car_consumer1 got car # 1
172 s top.car_consumer1 got car # 2
300 s top.car_consumer1 got car # 3
365 s top.car_consumer1 got car # 4
433 s top.car_consumer1 got car # 5
498 s top.car_consumer1 got car # 6
626 s top.car_consumer1 got car # 7
691 s top.car_consumer1 got car # 8
759 s top.car_consumer1 got car # 9
827 s top.car_consumer1 got car # 10
827 s top.car_consumer1 total cars produced: 10
827 s top.car_consumer1 time per car: 82700 ms

Info: /OSCI/SystemC: Simulation stopped by user.
```

Goal is to produce each car in smallest amount of time

Running simple car_factory



```
Wave - Default
                                                              Msgs
\top.car_factory1.spark_plug_robot1.busy\                     1'h0
\top.car_factory1.spark_plug_robot1.maintenance\              1'h1
\top.car_factory1.engine_install_robot1.busy\                 1'h0
\top.clk\                                                     1'h1
\top.car_factory1.finished_engines_val\                       1'h0
\top.car_factory1.finished_engines_rdy\                       1'h1
\top.car_factory1.finished_engines_msg.engine\                16'h0009
\top.car_factory1.finished_engines_msg.spark_plug0.s...       16'h0024
\top.car_factory1.finished_engines_msg.spark_plug1.s...       16'h0025
\top.car_factory1.finished_engines_msg.spark_plug2.s...       16'h0026
\top.car_factory1.finished_engines_msg.spark_plug3.s...       16'h0027
\top.spark_plugs_val\                                         1'h1
\top.spark_plugs_rdy\                                         1'h0
\top.spark_plugs_msg.spark_plug\                              16'h0029
\top.engines_val\                                             1'h1
\top.engines_rdy\                                             1'h0
\top.engines_msg.engine\                                      16'h000b
\top.engines_msg.spark_plug0.spark_plug\                      16'h0000
\top.engines_msg.spark_plug1.spark_plug\                      16'h0000
\top.engines_msg.spark_plug2.spark_plug\                      16'h0000
\top.engines_msg.spark_plug3.spark_plug\                      16'h0000
\top.chassis_val\                                             1'h1
\top.chassis_rdy\                                             1'h1
\top.chassis_msg.chassis\                                     16'h000c
\top.cars_val\                                                1'h1
\top.cars_rdy\                                                1'h1
\top.cars_msg.chassis.chassis\                                16'h0009
\top.cars_msg.engine.engine\                                  16'h0009
\top.cars_msg.engine.spark_plug0.spark_plug\                  16'h0024
\top.cars_msg.engine.spark_plug1.spark_plug\                  16'h0025
\top.cars_msg.engine.spark_plug2.spark_plug\                  16'h0026
\top.cars_msg.engine.spark_plug3.spark_plug\                  16'h0027
Now                                                           827 sec
Cursor 1                                                      0.00 sec
```

Overutilized

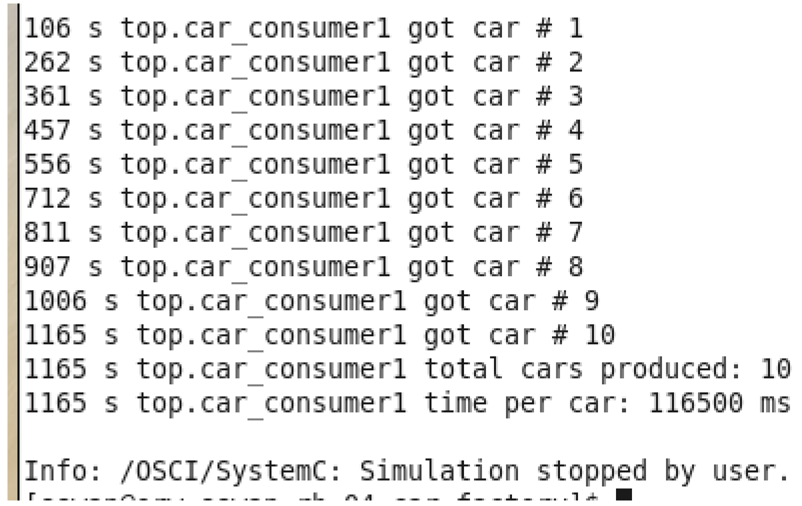
Underutilized

Sequential car_factory

```
#elif defined(SEQUENTIAL)
class car_factory : public sc_module {
public:
  sc_in<bool>                    INIT_S1(clk);
  Connections::In<spark_plug_t> INIT_S1(spark_plugs);
  Connections::In<engine_t>     INIT_S1(engines);
  Connections::In<chassis_t>    INIT_S1(chassis);
  Connections::Out<car_t>       INIT_S1(cars);

  SC_CTOR(car_factory)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  SC_SIG(bool, spark_plug_robot_busy);
  SC_SIG(bool, spark_plug_robot_maintenance);
  SC_SIG(bool, engine_install_robot_busy);

  void main() {
    spark_plug_t plugs[engine_t::plugs];
    int plug_count = 0;

    engine_t unfinished_engine;
    int unfinished_engine_count = 0;
    engine_t finished_engine;
    int finished_engine_count = 0;
    chassis_t chassis_inst;
    int chassis_count = 0;
    int spark_plug_robot_count = 0;

    while (1) {
      if (plug_count < engine_t::plugs)
        if (spark_plugs.PopNB(plugs[plug_count]))
          ++plug_count;

      if (unfinished_engine_count == 0)
        if (engines.PopNB(unfinished_engine))
          ++unfinished_engine_count;

      if (chassis_count == 0)
        if (chassis.PopNB(chassis_inst))
          ++chassis_count;
```

```
      if ((unfinished_engine_count == 1) && (plug_count == engine_t::plugs)
          && (finished_engine_count == 0))
      {
       finished_engine = unfinished_engine;
       for (int i=0; i < engine_t::plugs; i++)
          finished_engine.spark_plugs[i] = plugs[i];
       spark_plug_robot_busy = 1;
       wait(60);
       spark_plug_robot_busy = 0;
       if ((spark_plug_robot_count++ & 1) && (rand() & 3))
       {
         spark_plug_robot_maintenance = 1;
         wait(60);
         spark_plug_robot_maintenance = 0;
       }
       finished_engine_count = 1;
       plug_count = 0;
       unfinished_engine_count = 0;
      }

      if ((finished_engine_count == 1) && (chassis_count == 1))
      {
        car_t car;
        car.chassis = chassis_inst;
        car.engine = finished_engine;
        engine_install_robot_busy = 1;
        wait(30);
        engine_install_robot_busy = 0;
        cars.Push(car);
        finished_engine_count = 0;
        chassis_count = 0;
      }

      wait();
    }
  }
};
```

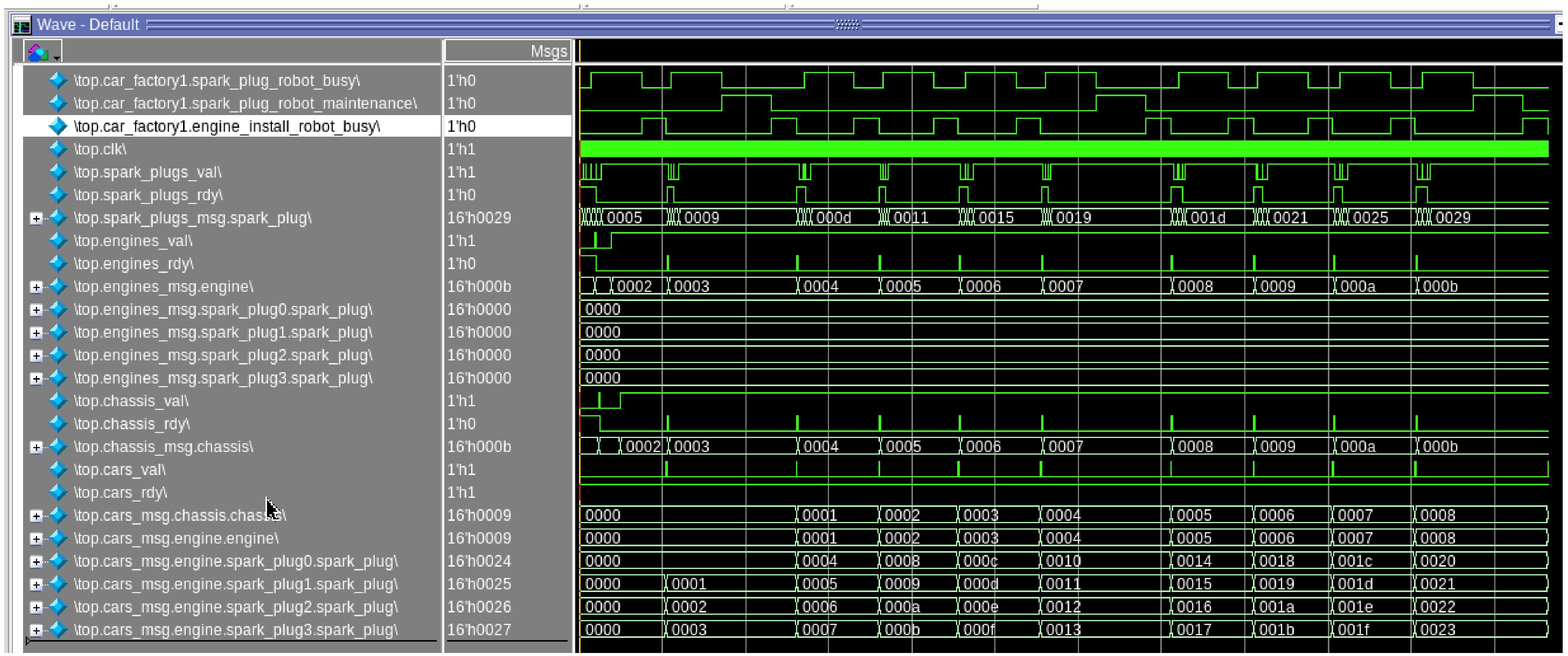
Running sequential car_factory



```
106 s top.car_consumer1 got car # 1
262 s top.car_consumer1 got car # 2
361 s top.car_consumer1 got car # 3
457 s top.car_consumer1 got car # 4
556 s top.car_consumer1 got car # 5
712 s top.car_consumer1 got car # 6
811 s top.car_consumer1 got car # 7
907 s top.car_consumer1 got car # 8
1006 s top.car_consumer1 got car # 9
1165 s top.car_consumer1 got car # 10
1165 s top.car_consumer1 total cars produced: 10
1165 s top.car_consumer1 time per car: 116500 ms

Info: /OSCI/SystemC: Simulation stopped by user.
[illegible]
```

Car production time got worse!

Running sequential car_factory



	Msgs
\top.car_factory1.spark_plug_robot_busy\	1'h0
\top.car_factory1.spark_plug_robot_maintenance\	1'h0
\top.car_factory1.engine_install_robot_busy\	1'h0
\top.clk\	1'h1
\top.spark_plugs_val\	1'h1
\top.spark_plugs_rdy\	1'h0
\top.spark_plugs_msg.spark_plug\	16'h0029
\top.engines_val\	1'h1
\top.engines_rdy\	1'h0
\top.engines_msg.engine\	16'h000b
\top.engines_msg.spark_plug0.spark_plug\	16'h0000
\top.engines_msg.spark_plug1.spark_plug\	16'h0000
\top.engines_msg.spark_plug2.spark_plug\	16'h0000
\top.engines_msg.spark_plug3.spark_plug\	16'h0000
\top.chassis_val\	1'h1
\top.chassis_rdy\	1'h0
\top.chassis_msg.chassis\	16'h000b
\top.cars_val\	1'h1
\top.cars_rdy\	1'h1
\top.cars_msg.chassis.chassis\	16'h0009
\top.cars_msg.engine.engine\	16'h0009
\top.cars_msg.engine.spark_plug0.spark_plug\	16'h0024
\top.cars_msg.engine.spark_plug1.spark_plug\	16'h0025
\top.cars_msg.engine.spark_plug2.spark_plug\	16'h0026
\top.cars_msg.engine.spark_plug3.spark_plug\	16'h0027

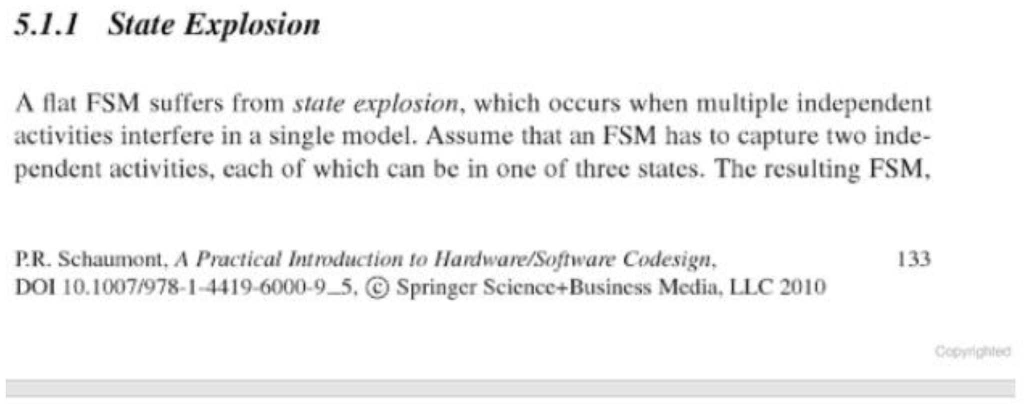
Worse!

Worse!

Will HLS fix sequential car_factory?

- Can HLS split the single sequential process into 2 different processes to improve the utilization / QOR?
  - — No. HLS always generates a single FSM+Datapath per SC process in the input model.
- OK, then can HLS generate single FSM that is the product of the 2 simpler state machines?
  - — No. Even if it did, you would have "state explosion", leading to bad QOR due to a huge FSM.

The “state explosion” problem



5.1.1 State Explosion

A flat FSM suffers from *state explosion*, which occurs when multiple independent activities interfere in a single model. Assume that an FSM has to capture two independent activities, each of which can be in one of three states. The resulting FSM,

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*, 133
DOI 10.1007/978-1-4419-6000-9_5, © Springer Science+Business Media, LLC 2010

Copyrighted

134 5 Microprogrammed Architectures

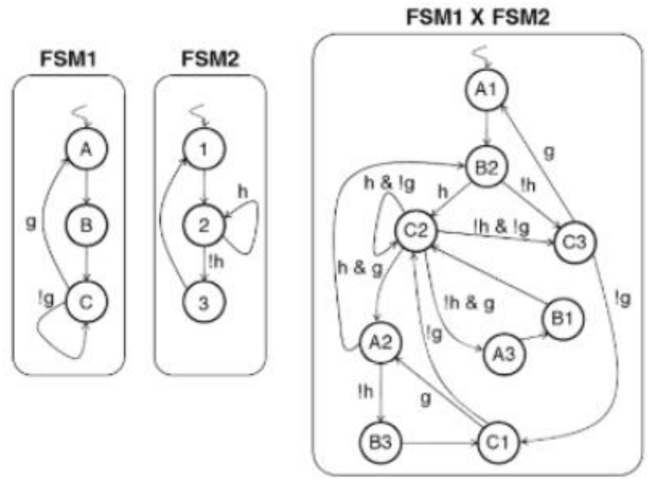
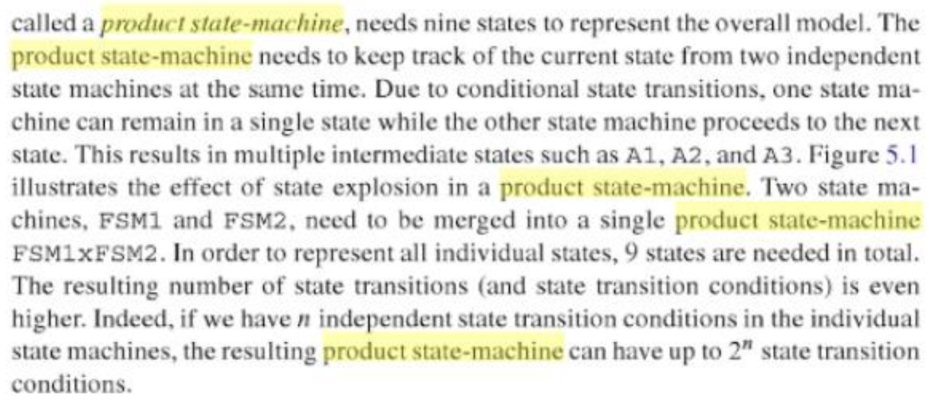


Fig. 5.1 State explosion in FSM when creating a product state-machine



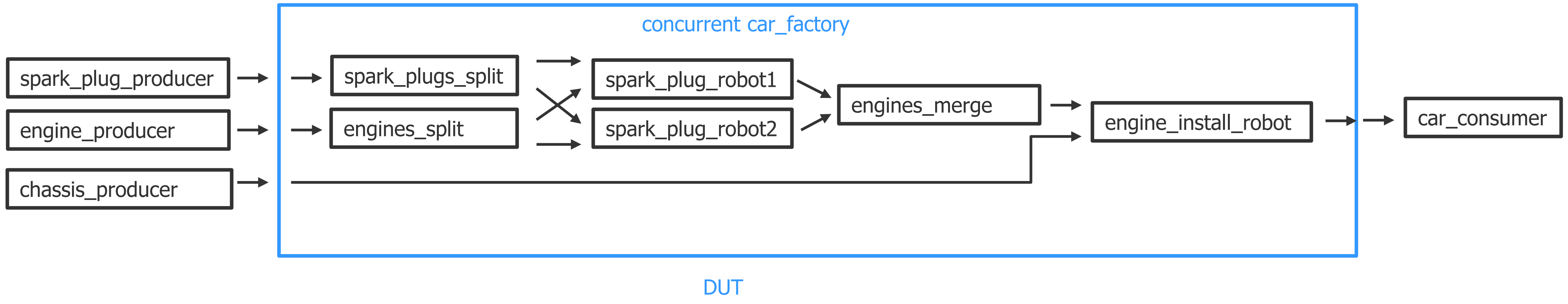
called a *product state-machine*, needs nine states to represent the overall model. The product state-machine needs to keep track of the current state from two independent state machines at the same time. Due to conditional state transitions, one state machine can remain in a single state while the other state machine proceeds to the next state. This results in multiple intermediate states such as `A1`, `A2`, and `A3`. Figure 5.1 illustrates the effect of state explosion in a product state-machine. Two state machines, `FSM1` and `FSM2`, need to be merged into a single product state-machine `FSM1xFSM2`. In order to represent all individual states, 9 states are needed in total. The resulting number of state transitions (and state transition conditions) is even higher. Indeed, if we have n independent state transition conditions in the individual state machines, the resulting product state-machine can have up to 2^n state transition conditions.

Mentor A Siemens Business

How do we fix the car_factory architecture?

- Primary problem in “simple” car_factory is overutilization of spark_plug_robot
- Obvious solution: add another spark_plug_robot

concurrent car_factory

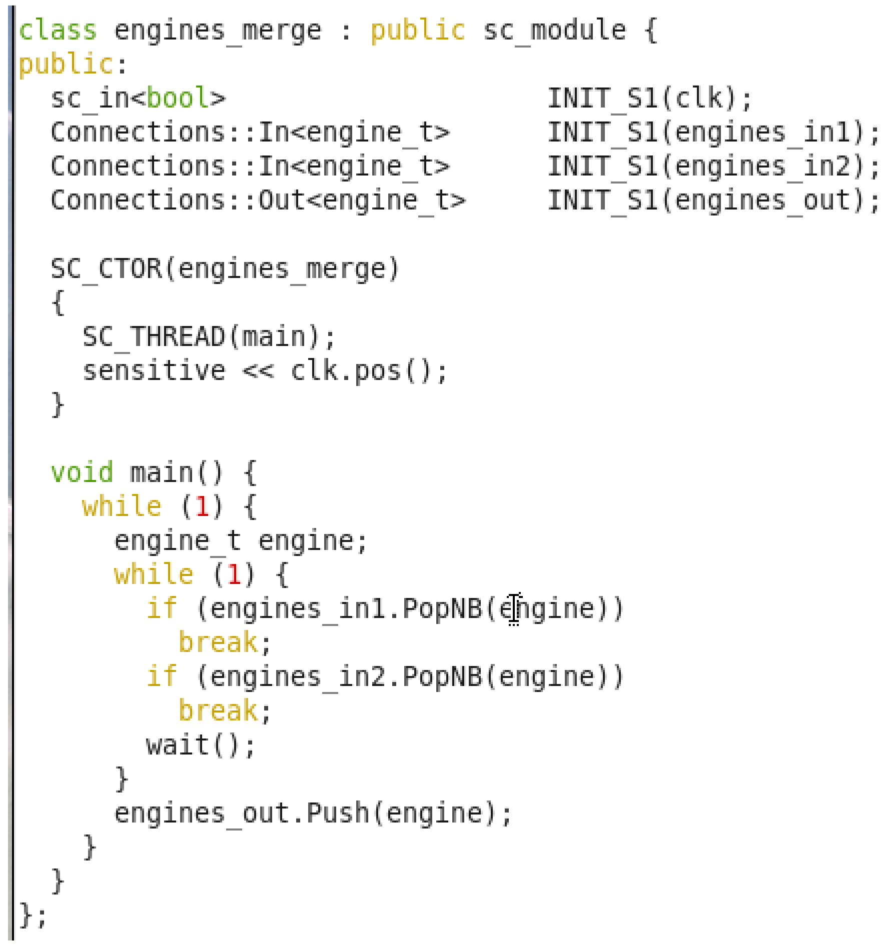


spark_plugs_split and engines_split

```
class spark_plugs_split : public sc_module {
public:
  sc_in<bool>                        INIT_S1(clk);
  Connections::In<spark_plug_t>      INIT_S1(spark_plugs_in);
  Connections::Out<spark_plug_t>     INIT_S1(spark_plugs_out1);
  Connections::Out<spark_plug_t>     INIT_S1(spark_plugs_out2);

  SC_CTOR(spark_plugs_split)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    while (1) {
      spark_plug_t spark_plug = spark_plugs_in.Pop();
      while (1) {
        if (spark_plugs_out1.PushNB(spark_plug))
          break;
        if (spark_plugs_out2.PushNB(spark_plug))
          break;
        wait();
      }
    }
  }
};
```

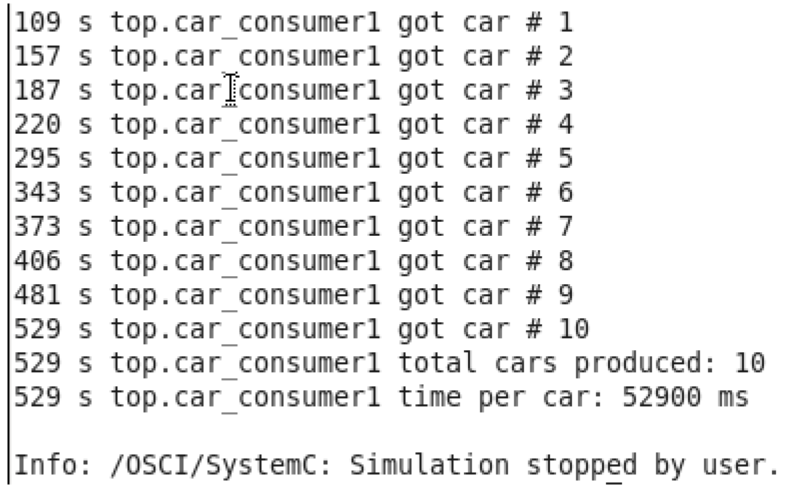
```
class engines_split : public sc_module {
public:
  sc_in<bool>                     INIT_S1(clk);
  Connections::In<engine_t>       INIT_S1(engines_in);
  Connections::Out<engine_t>      INIT_S1(engines_out1);
  Connections::Out<engine_t>      INIT_S1(engines_out2);

  SC_CTOR(engines_split)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    while (1) {
      engine_t engine = engines_in.Pop();
      while (1) {
        if (engines_out1.PushNB(engine))
          break;
        if (engines_out2.PushNB(engine))
          break;
        wait();
      }
    }
  }
};
```

engines_merge



```
class engines_merge : public sc_module {
public:
  sc_in<bool>                     INIT_S1(clk);
  Connections::In<engine_t>       INIT_S1(engines_in1);
  Connections::In<engine_t>       INIT_S1(engines_in2);
  Connections::Out<engine_t>      INIT_S1(engines_out);

  SC_CTOR(engines_merge)
  {
    SC_THREAD(main);
    sensitive << clk.pos();
  }

  void main() {
    while (1) {
      engine_t engine;
      while (1) {
        if (engines_in1.PopNB(engine))
          break;
        if (engines_in2.PopNB(engine))
          break;
        wait();
      }
      engines_out.Push(engine);
    }
  }
};
```

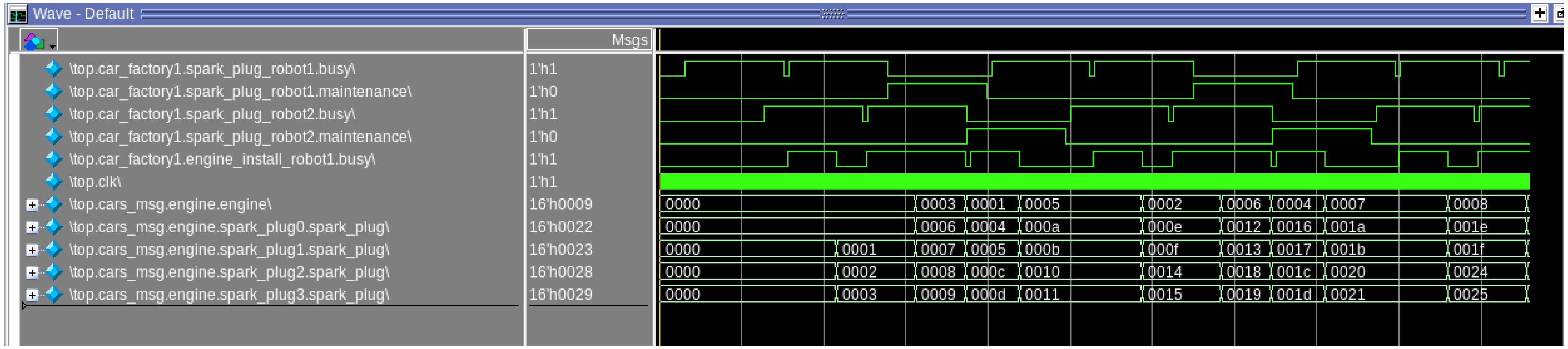
Running concurrent car_factory



```
109 s top.car_consumer1 got car # 1
157 s top.car_consumer1 got car # 2
187 s top.car_consumer1 got car # 3
220 s top.car_consumer1 got car # 4
295 s top.car_consumer1 got car # 5
343 s top.car_consumer1 got car # 6
373 s top.car_consumer1 got car # 7
406 s top.car_consumer1 got car # 8
481 s top.car_consumer1 got car # 9
529 s top.car_consumer1 got car # 10
529 s top.car_consumer1 total cars produced: 10
529 s top.car_consumer1 time per car: 52900 ms

Info: /OSCI/SystemC: Simulation stopped by user.
```

Big improvement in car production time!

Running concurrent car_factory



```
Wave - Default
                                                     Msgs
\top.car_factory1.spark_plug_robot1.busy\            1'h1
\top.car_factory1.spark_plug_robot1.maintenance\     1'h0
\top.car_factory1.spark_plug_robot2.busy\            1'h1
\top.car_factory1.spark_plug_robot2.maintenance\     1'h0
\top.car_factory1.engine_install_robot1.busy\        1'h1
\top.clk\                                            1'h1
\top.cars_msg.engine.engine\                         16'h0009   0000       0003 0001 0005  0002 0006 0004 0007  0008
\top.cars_msg.engine.spark_plug0.spark_plug\         16'h0022   0000       0006 0004 000a  000e 0012 0016 001a  001e
\top.cars_msg.engine.spark_plug1.spark_plug\         16'h0023   0000  0001 0007 0005 000b  000f 0013 0017 001b  001f
\top.cars_msg.engine.spark_plug2.spark_plug\         16'h0028   0000  0002 0008 000c 0010  0014 0018 001c 0020  0024
\top.cars_msg.engine.spark_plug3.spark_plug\         16'h0029   0000  0003 0009 000d 0011  0015 0019 001d 0021  0025
```

Both spark_plug_robots busy at same time

Better utilization

Output order of engines and plugs no longer matches input order

Summing it up…

- In practice, SC modeling for HLS is a series of stepwise refinements that get to a good architectural model.
- You can do a lot of refinement & architectural analysis in pure SC model by analyzing the pre-HLS simulations.
- Use HLS to find latency/area/power issues and iterate with changes to the SC model source code.

SystemC HLS Process Structure and Blocking/Non-Blocking IO

- A process (SC_THREAD) is like an “always” block in Verilog that is sensitive to the clock.
  - Usually it is “bigger” than a Verilog always block because it is an implicit state machine, may be pipelined by catapult, etc.
  - Usually every "block" in the HW architecture diagram will imply at least one independent SC_THREAD in the HLS model.
- Process structure has a huge impact on HLS runtime and QOR
  - Big processes usually mean "big" FSMs with lots of states and transitions **== bad QOR**
  - So, knowing how to arrive at a "good" process structure is extremely important
  - We saw in the car_factory example how breaking the design down into simpler concurrent activities led to a more efficient implementation

Guidelines for arriving at a good process structure

- Usually IO is the biggest consideration in how to structure SC_THREADs.
- The ideal structure is usually an SC_THREAD with at least one blocking input (ie one Pop operation) and/or at least one blocking output (ie Push operation).
- At least one of the blocking inputs or outputs should be called unconditionally on every iteration of the main loop.
- If functionality can easily be expressed as independent SC_THREADs then it is generally better to do so – they will run independently, have smaller control FSMs, etc.
- But, resources cannot be shared by Catapult across different SC_THREADs, and any communication between 2 SC_THREADs will take at least 1 clock cycle since they are clocked processes.

Unconditional blocking inputs or outputs..

- Why is it important to have at least one unconditional blocking input or output?
  - Usually the blocking IO is tied to each iteration of the main loop (which may be pipelined with for example II=1).
  - The blocking IO semantics and the clear II semantics clearly tie down the expected thruput requirements between the user and the HLS tool.
  - In contrast, if the IO is non-blocking, then even if the II=1 then Catapult may generate a state machine where on some iterations of a pipelined loop no actual IO occurs (since it is non-blocking)
  - Also, with blocking IO, the FSM will be in a clear "blocked" state when no IO can occur, and this makes idle state modeling easier (e.g. for power saving)

When do you need to use Non-blocking IO?

- Arbitration requires Peek or PopNB to all arbitrated inputs.
- Time-based splitting and merging of transaction streams requires PushNB and PopNB (respectively)
- There are cases where a process will need ALL non-blocking IO (all PopNB and PushNB), but it should be pretty rare.
  - In this case the process should be kept as small / simple as possible, ideally communicating with other processes that follow the guidelines above.
  - With all non-blocking IO, you will likely be modeling at very close to RTL level, and most likely HLS will just be translating SC RTL into Verilog RTL.

Summary: Prefer to use blocking IO over nonblocking IO.

- Your models will be simpler and more likely to have a good process structure.
- 100% blocking IO is called KPN (Kahn Process Networks)
  - KPN is deterministic
  - easier to verify.
- Non-blocking IO is sometimes needed, but introduces timing dependent behavior, and can make verification more difficult in some cases if the timing dependent behavior is externally visible.

Where to use wait() in HLS models

- Use 1 wait() statement at top of main loop to model the reset state.
- Only use wait() in HLS models within loops that have exclusively non-blocking IO.
- If you are modeling low-level protocols using sc_signal, you may need to use wait() (but you should try to avoid doing this!)
- You should not be using wait() anywhere else.

Always try to think first about the HW architecture of the model

- Put your HW architects in front of a white board and get them to talk!
- What are primary blocks, transaction streams and required thruputs?
- What activities can naturally be expressed as separate concurrent processes?
- Where does arbitration need to occur?
- How can the architecture be modified to reduce undesirable contention?
- How can data storage and lifetime be minimized?
- What are expensive resources that should be shared ?
  - Either add into same process,
  - Or add arbiter to allow multiple processes to access

What should be left for Catapult to do?

- Automated microarchitecture generation:
  - Detailed scheduling
  - resource allocation and sharing
  - loop unrolling
  - loop pipelining
  - memory access scheduling and interfacing
  - FSM generation
  - meeting timing of target silicon

Conclusion

- Modeling a "good" process structure is key to getting good QOR thru HLS
- HLS tools such as Catapult can provide analysis feedback to help you identify "bad" process structures, but they will not automatically find a "good" one for you.
- Properly modeling the HW architecture and a good process structure is the responsibility of the HLS modeling engineers and HW architects

Mentor®

A Siemens Business

www.mentor.com