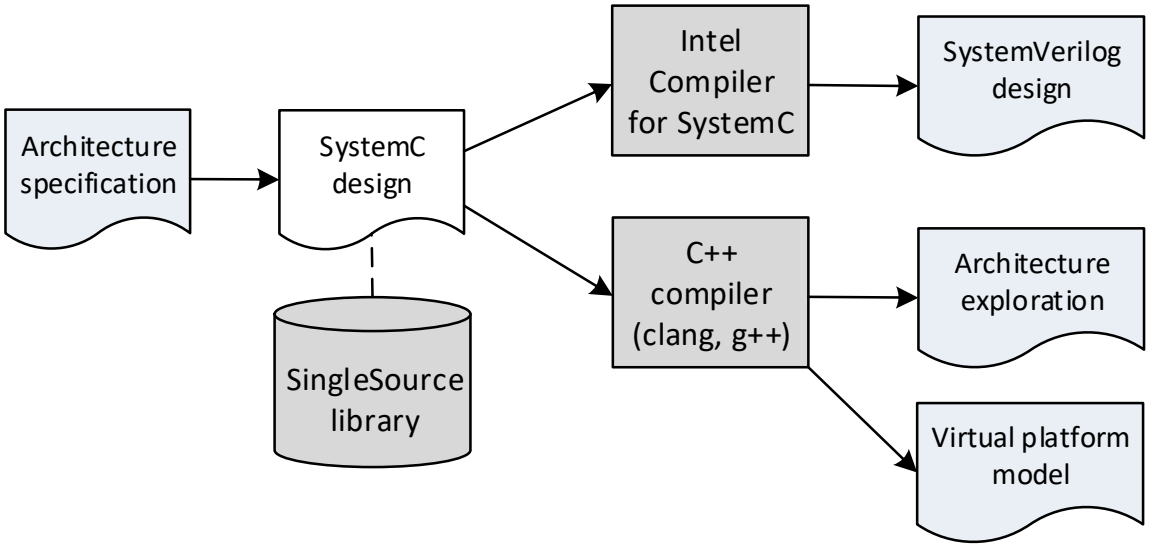
SINGLE SOURCE LIBRARY FOR HIGH LEVEL MODELLING AND DIGITAL DESIGN

Part I. Introduction

Single Source design flow

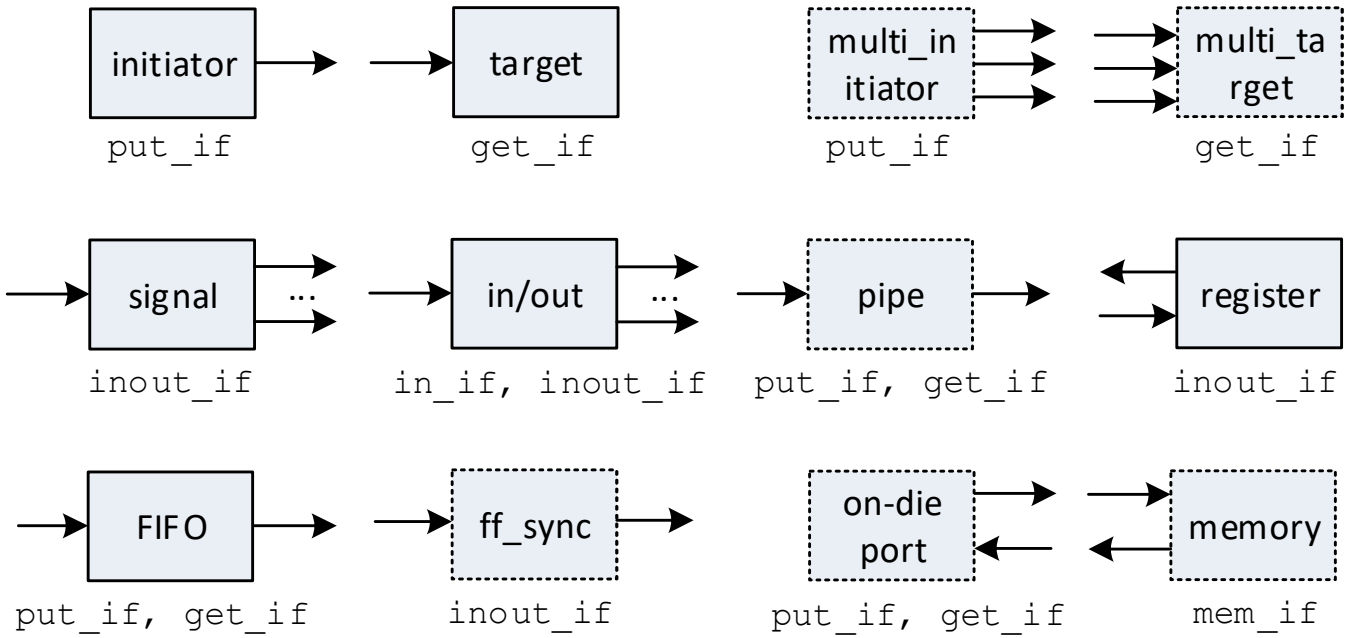
- Single Source design flow uses the same SystemC code for
  - Digital design, SystemVerilog compatible with Intel ASIC/FPGA flow
  - Architecture exploration, performance evaluation
  - Virtual platform, fast SystemC simulation model compatible with Simics



Single Source Library

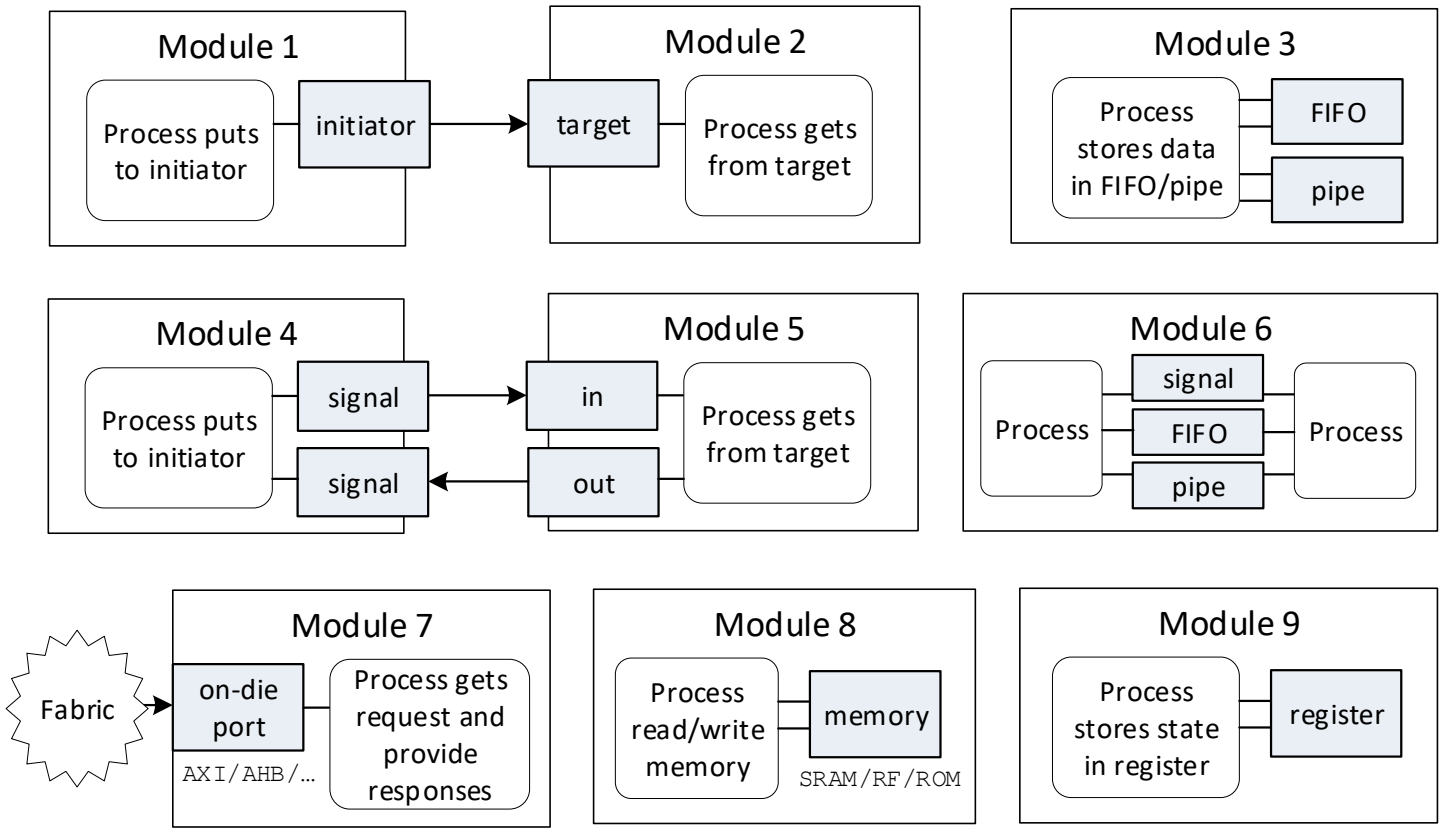
- Single Source Library of high-level channels with two modes
  - Cycle accurate mode
    - Cycle accurate clocked design
    - No extra cost in area, performance, and power
  - Fast simulation mode
    - No clock, request(data)-driven simulation
    - No extra process activation, fast simulation
- New design style forced by the library
  - Simple and error-preventive API causes significant code simplification
  - Reduced number of processes, specially clocked threads

Library Overview



Two implementations inside of the channels: cycle accurate and fast simulation

Use cases


Module 1
Process puts to initiator
initiator
Module 2
target
Process gets from target
Module 3
Process stores data in FIFO/pipe
FIFO
pipe
Module 4
Process puts to initiator
signal
signal
Module 5
in
out
Process gets from target
Module 6
Process
signal
FIFO
pipe
Process
Fabric
Module 7
on-die port
AXI/AHB/...
Process gets request and provide responses
Module 8
Process read/write memory
memory
SRAM/RF/ROM
Module 9
Process stores state in register
register

Library interfaces

Interface	Channels	Comment
`sct_put_if`	`sct_initiator, sct_fifo, sct_base_port`	`bool ready()` `bool put(const T&)`

`sct_get_if`	`sct_target, sct_fifo, sct_base_port`	`bool request()`
		`T get()`
		`T peek()`

`sct_fifo_if`	`sct_fifo inherits sct_put_if and sct_get_if`	`unsigned elem_num()` `bool almost_full(unsigned)`
`sct_in_if`	`sct_in`	`T read()`
`sct_inout_if`	`sct_out, sct_signal, sct_register, ff_sync`	`T read()` `void write(const T& val)`

Library implementation and usage

- Library implemented in headers only
  - Project should include **sct_common.h**
    - All required headers included inside
    - Namespace **sct** using added
- Defines and interfaces in **sct_ipc_if.h**
  - **SCT_TLM_MODE** – if defined fast simulation mode enabled, by default cycle accurate mode
  - **SCT_DEFAULT_TRAITS** – specifies clock and reset levels, default SCT_POSEDGE_NEGRESET
  - **SCT_TLM_MODE** and **SCT_DEFAULT_TRAITS** are usually defined in the project Makefile/CMakeList.txt

SystemC processes

- SC_METHOD – for combinational logic
  - sensitivity list with all read signals/port and accessed target/initiator/fifo/…
  - sensitivity list does not contain reset
- SC_THREAD – for sequential logic
  - sensitivity list with all read signals/port and accessed target/initiator/fifo/…
  - reset specification with **async_reset_signal_is** or/and **sync_reset_signal_is**
- SCT_THREAD – same as SC_THREAD, but clock is explicitly provided
  - `SCT_THREAD(func, clk)`
  - required if no target/initiator/fifo/… are in process sensitivity, only signals/ports

SystemC processes example

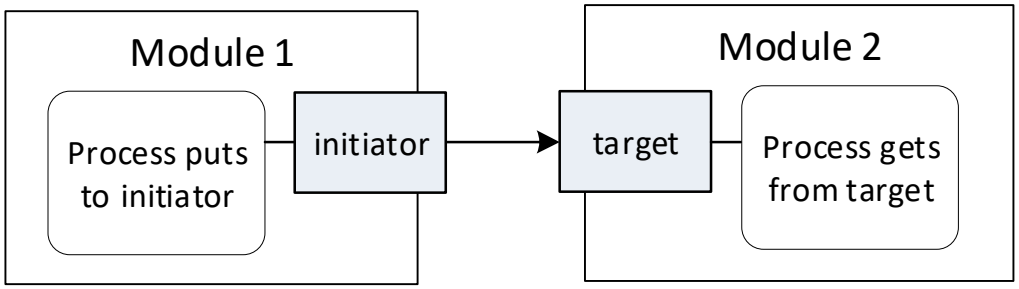
```
class A : public sc_module {
    sc_in<bool>              clk{"clk"};
    sc_in<bool>              nrst{"nrst"};
    sct_target<bool>         targ{"targ"};
    sct_in<bool>             in{"in"};
    sct_signal<sc_uint<8>>   sig{"sig"};
    A (const sc_module_name& name) : sc_module(name) {
        targ.ck_nrst(clk, nrst);
        SC_METHOD(methProc); sensitive << in;
        SC_THREAD(thrdProc1); sensitive << targ;
        async_reset_signal_is(nrst, 0);
        SCT_THREAD(thrdProc2, clk.pos()); sensitive << sig;
        async_reset_signal_is(nrst, 0);
    }
}
```

SINGLE SOURCE LIBRARY FOR HIGH LEVEL MODELLING AND DIGITAL DESIGN

Part II. Initiator and Target

Initiator and target

- Initiator and Target are intended to connect two modules
  - Initiator implements `sct_put_if` and should be used in one process to put requests
  - Target implements `sct_get_if` and should be used in one process to get requests which put by the connected Initiator



Put interface

sct_put_if	`bool ready()`	Return true if it is ready to put request
	`void reset_put()`	Reset this initiator/FIFO
	`void clear_put()`	Clear (remove) request put in this cycle
	`bool put(const T& data)`	Put request into initiator/FIFO if it is ready, return ready to request
	`bool put(const T& data, sc_uint<N> mask)`	Put request into initiator/FIFO if it is ready, `mask` used to enable/disable put or choose targets in multi-cast put, return ready to request
	`void b_put(const T& data)`	May-block put request, could be used in THREAD process only

Get interface

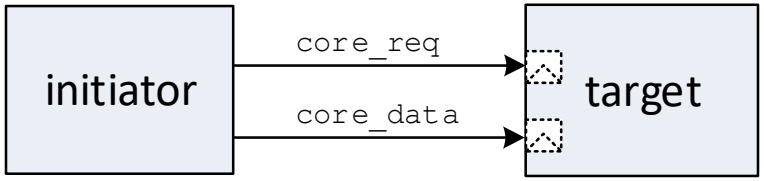
sct_get_if	`bool request()`	Return true if it has request to get
	`void reset_get()`	Reset this target/FIFO
	`void clear_get()`	Clear (return back) request got in this cycle
	`T peek()`	Peek request, return current request data, if no request last data returned
	`T get()`	Get request and remove it from FIFO/target, return current request data, if no request last data returned
	`bool get(T& data, bool enable)`	Get request and remove it from FIFO/target if `enable` is true, return true if there is a request
	`T b_get()`	May-block get request, could be used in THREAD process only

Initiator and target parameters

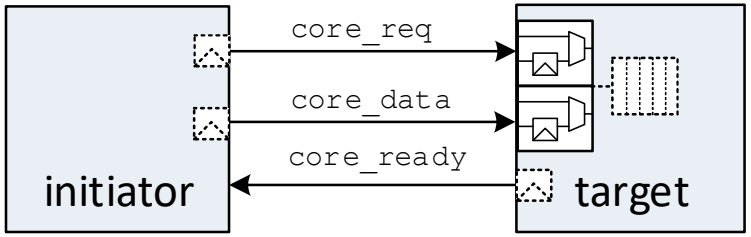
```
template< class T,                          // Payload data type
    class TRAITS = SCT_CMN_TRAITS,          // Clock edge and reset level traits
    bool TLM_MODE = SCT_CMN_TLM_MODE>       // Cycle accurate (0) or fast simulation (1) mode
class sct_initiator {
   sct_initiator(sc_module_name name,       // Module name -- same as instance variable name
                 bool sync = 0);            // Sync register to pipeline request
};
template< class T,                          // Payload data type
    class TRAITS = SCT_CMN_TRAITS,          // Clock edge and reset level traits
    bool TLM_MODE = SCT_CMN_TLM_MODE>       // Cycle accurate (0) or fast simulation (1) mode
class sct_target {
   sct_target(sc_module_name name,          // Module name -- same as instance variable name
              bool sync = 0,                // Is register required to pipeline request
              bool always_ready = 0);       // Is always ready to get request
};
```

Connection types

Combinational connection



Buffered connection



- Combinational connection
  - For target which is always ready
  - Optional registers for request
  - Full throughput
- Buffered connection
  - Optional registers for request
  - Optional register for ready, automatically assigned to protect from combinational loop
  - Optional FIFO
  - Full throughput in every mode

Initiator and target example

```
class A : sc_module {
sct_initiator<T>     SC_NAMED(init);
A(const sc_module_name& name){
    SC_THREAD(randProc); sensitive << init;
}}
class B : sc_module {
sct_target<T>        SC_NAMED(targ);
explicit B(const sc_module_name& name){
    SC_METHOD(checkProc); sensitive << targ;
}}
class Top : sc_module {
   A    SC_NAMED(a);
   B    SC_NAMED(b);
   Top(const sc_module_name& name) {
      a.init.bind(b.targ);
}}
```

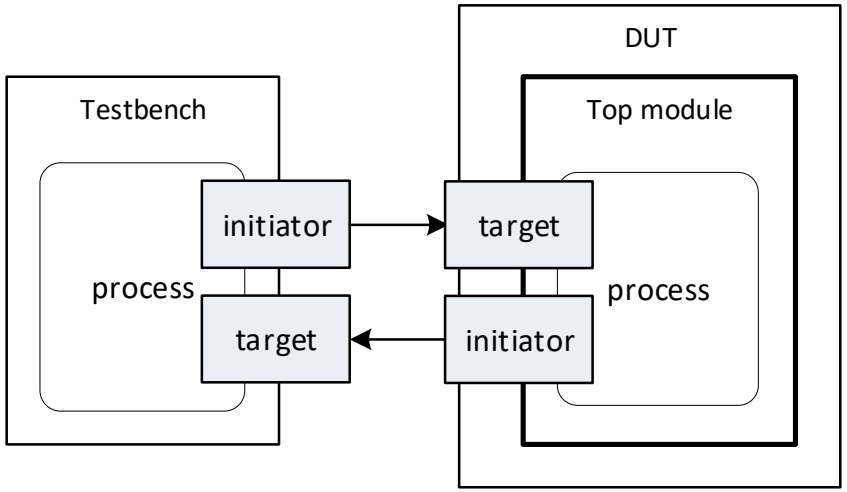
```
void randProc() {
    init.reset_put();
    wait();
    while (true) {
        T val = getRandom();
        init.put(val);        // init.b_put(val);
        wait();
    }
}
void checkProc() {
    targ.reset_get();
    if (targ.request()) {
        intr = targ.get() == 42;
    }
}
```

Target with FIFO example

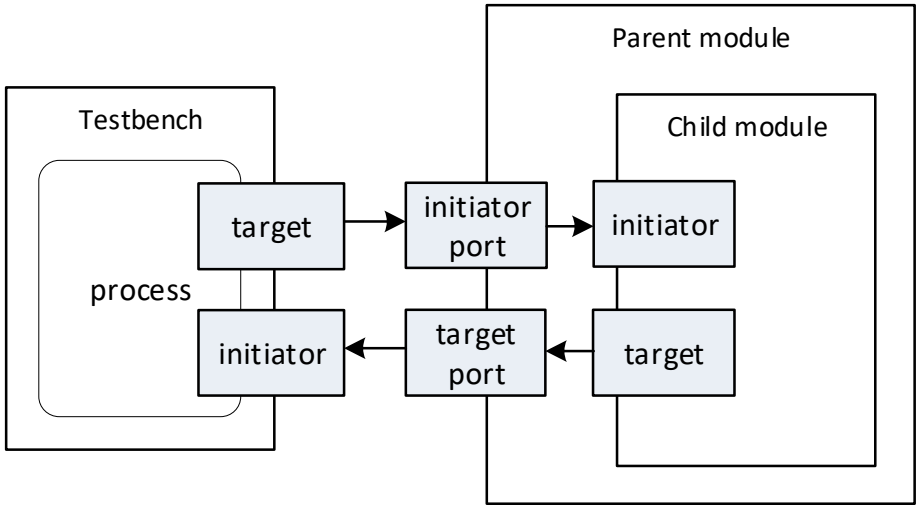
```
template<unsigned LENGTH>                   // FIFO size (maximal number of elements)
void add_fifo(bool sync_valid = 0,          // Is register required to pipeline core_req and core_data
              bool sync_ready = 0,          // Is register required to pipeline core_ready
              bool init_buffer = 0);        // Initialize all the elements with zeros
                                            // First element to get is always initialized to zero


template<class T>
struct A : public sc_module {
    sct_target<T>       run{"run"};
    explicit A(const sc_module_name& name) : sc_module(name) {
        run.clk_nrst(clk, nrst);
        run.template add_fifo<2>(1, 1);  // Add FIFO with 2 element and registers in request/response
}}
```

Initiator and target in top module

- Target and initiator can be instantiated in top module to be connected to testbench
  - no sync register allowed for top module and testbench targets/initiators
- For multi-language simulation BIND_CHANNEL macro
  - BIND_CHANNEL(dut_module, dut_channel, tb_channel) – for individual channel
  - BIND_CHANNEL(dut_module, dut_channel, tb_channel, size) – for channel array/vector



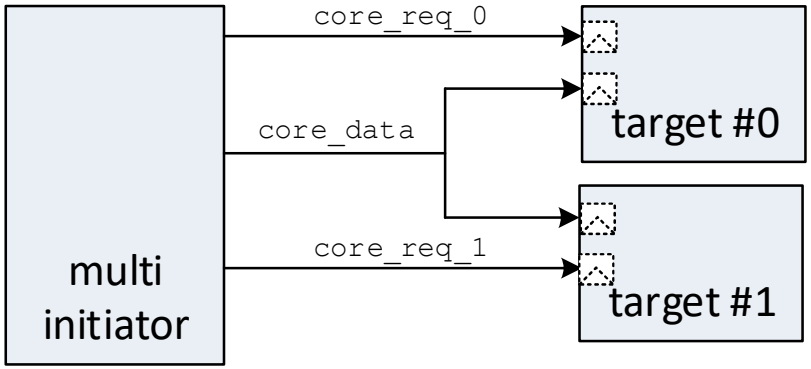
Target/Initiator ports



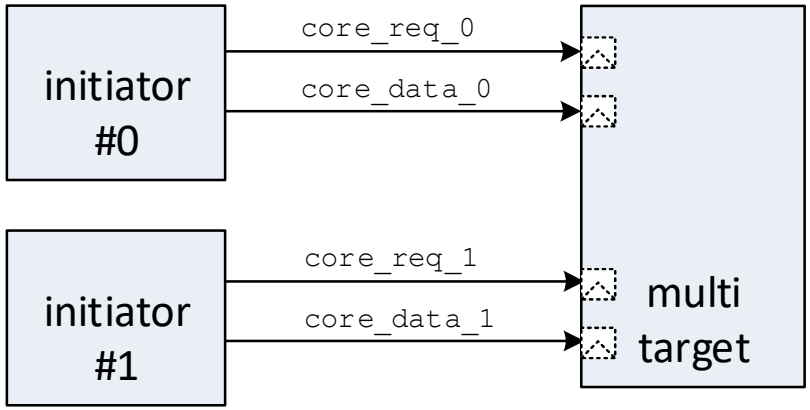

```
template<class T>
struct Child : public sc_module {
    sct_target<T>       run{"run"};
    sct_initiator<T>    resp{"resp"};
};
template<class T>
struct Parent: public sc_module  {
    // Target/initiator ports
    sc_port<sct_target<T>>       run;
    sc_port<sct_initiator<T>>    resp;
    Child<T>                     child{"child"};
    Parent(sc_module_name) : sc_module(name) {
        run(child.run);
        resp(child.resp);
 }}
```

Multi-initiator and Multi-target

Multi-initiator connection



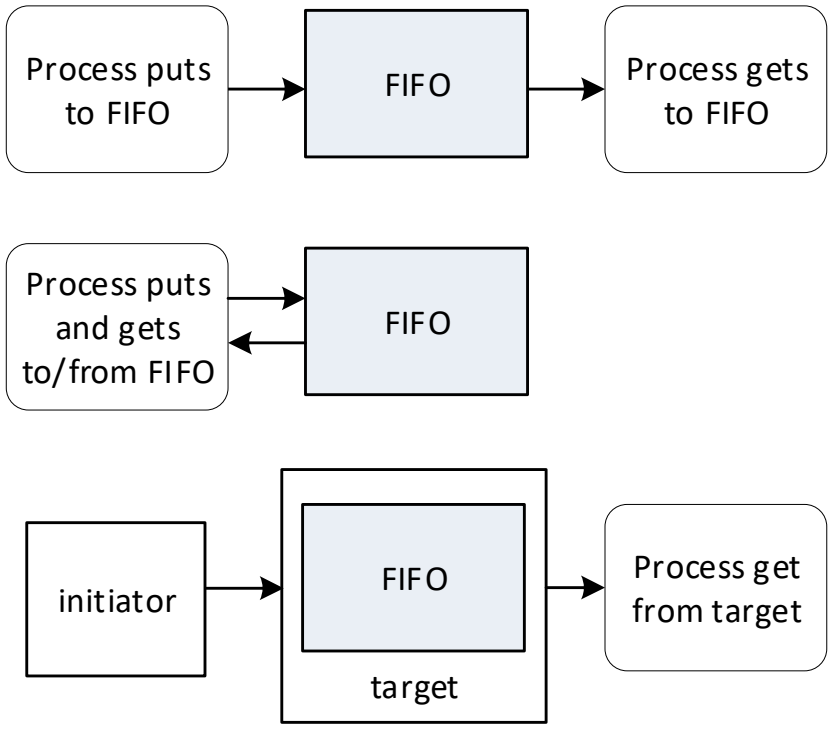
Multi-target connection



SINGLE SOURCE LIBRARY FOR HIGH LEVEL MODELLING AND DIGITAL DESIGN

Part III. FIFO and signals

FIFO



- FIFO usages
  1. Inter-process communication inside of a module
  2. Buffer for a process
  3. Internal buffer for a target
- FIFO parameters
  - Size
  - Synchronous/combinational for request and ready paths

FIFO interface

sct_fifo_if	inherits `sct_put_if<T>` and `sct_get_if<T>`	
	`unsigned size()`	FIFO LENGTH
	`unsigned elem_num()`	Number of elements in FIFO, value updated last clock edge for METHOD, last DC for THREAD
	`bool almost_full(unsigned N)`	Return true if FIFO has (LENGTH-N) elements or more, value updated last clock edge for METHOD, last DC for THREAD
	`bool almost_empty(unsigned N)`	Return true if FIFO has N elements or less, value updated last clock edge for METHOD, last DC for THREAD

FIFO parameters

```
template<
    class T,
    unsigned LENGTH,                    // Size (maximal number of elements)
    class TRAITS = SCT_CMN_TRAITS,      // Clock edge and reset level traits
    bool TLM_MODE = SCT_CMN_TLM_MODE>   // Cycle accurate (0) or fast simulation (1) mode
>
class sct_fifo {
   sct_fifo(const sc_module_name& name,
            bool sync_valid = 0,        // Request path has synchronous register
            bool sync_ready = 0,        // Response path has synchronous register
            bool use_elem_num = 0,      // Element number/Almost full or empty used
            bool init_buffer = 0)       // Initialize all buffer elements with zeros in reset
                                        // First element to get is always initialized to zero
};
```

Minimal FIFO size required

Initiator process	Target process	sync_valid	sync_ready	Minimal FIFO size
method	method	0	0	1
method	method	0	1	1
method	method	1	0	1
method	method	1	1	2
thread	thread	0	0	2
thread	thread	0	1	3
thread	thread	1	0	3
thread	thread	1	1	4

https://github.com/intel/systemc-compiler/wiki/SingleSource-library

Producer/consumer with FIFO in methods

```
sc_in<bool>          clk{"clk"};
sc_in<bool>          nrst{"nrst"};
sct_fifo<T, 2>       fifo{"fifo",0,1};
sct_signal<bool>     enbl{"enbl"};

explicit A(const sc_module_name& name) :
sc_module(name) {
    fifo.clk_nrst(clk, nrst);

    SC_METHOD(producerProc);
    sensitive << fifo.PUT << enbl;

    SC_METHOD(consumerProc);
    sensitive << fifo.GET;
}
```

```
void producerProc(){
    fifo.reset_put();
    if (enbl && fifo.ready()) {
        fifo.put(getSomeVal());
    }
}
void consumerProc(){
    fifo.reset_get();
    if (fifo.request()) {
        T val = fifo.get();
        doSomething(val);
    }
}
```

Producer/consumer with FIFO in threads

```
sc_in<bool>          clk{“clk”};
sc_in<bool>          nrst{“nrst”};
sct_fifo<T, 2>       fifo{"fifo"};
sct_signal<bool>     enbl{“enbl”};

explicit A(const sc_module_name& name) :
sc_module(name) {
    fifo.clk_nrst(clk, nrst);

    SCT_THREAD(producerProc, clk);
    sensitive << fifo.PUT << enbl;
    async_reset_signal_is(nrst, 0);

    SCT_THREAD(consumerProc, clk);
    sensitive << fifo.GET;
    async_reset_signal_is(nrst, 0);
}
```

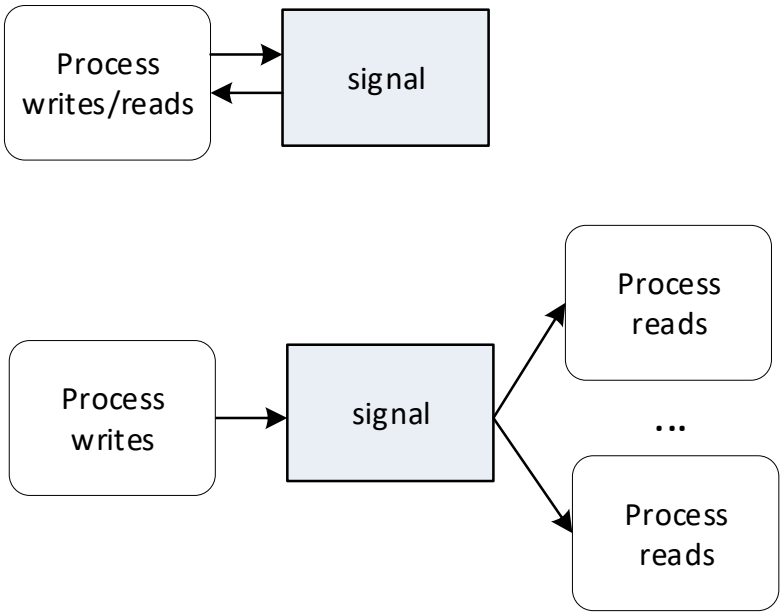
```
void producerProc(){
   fifo.reset_put();
   wait();
   while (true) {
       if (enbl) fifo.b_put(prodValue());
       wait();
}}
void consumerProc(){
   fifo.reset_get();
   wait();
   while (true) {
      consValue(fifo.b_get());
      wait();
}}
```

One process stores requests in FIFO

```
struct Top : public sc_module {
    sc_in<bool>          clk{"clk"};
    sc_in<bool>          nrst{"nrst"};
    sct_fifo<T, 5>       fifo{"fifo"};
    Top(sc_module_name name) : sc_module(name)
    {
        fifo.clk_nrst(clk, nrst);

        SCT_THREAD(storeProc, clk);
        sensitive << fifo;   // fifo.PUT & fifo.GET
        async_reset_signal_is(nrst, 0);
 }}
```

```
void storeProc() {
    fifo.reset();
    wait();
    while (true) {
       if (fifo.ready()) {
          fifo.put(getSomeValue());
       }
       wait();
       if (fifo.request()) {
          doSomething(fifo.get());
       }
    }
}
```

Signals inside module



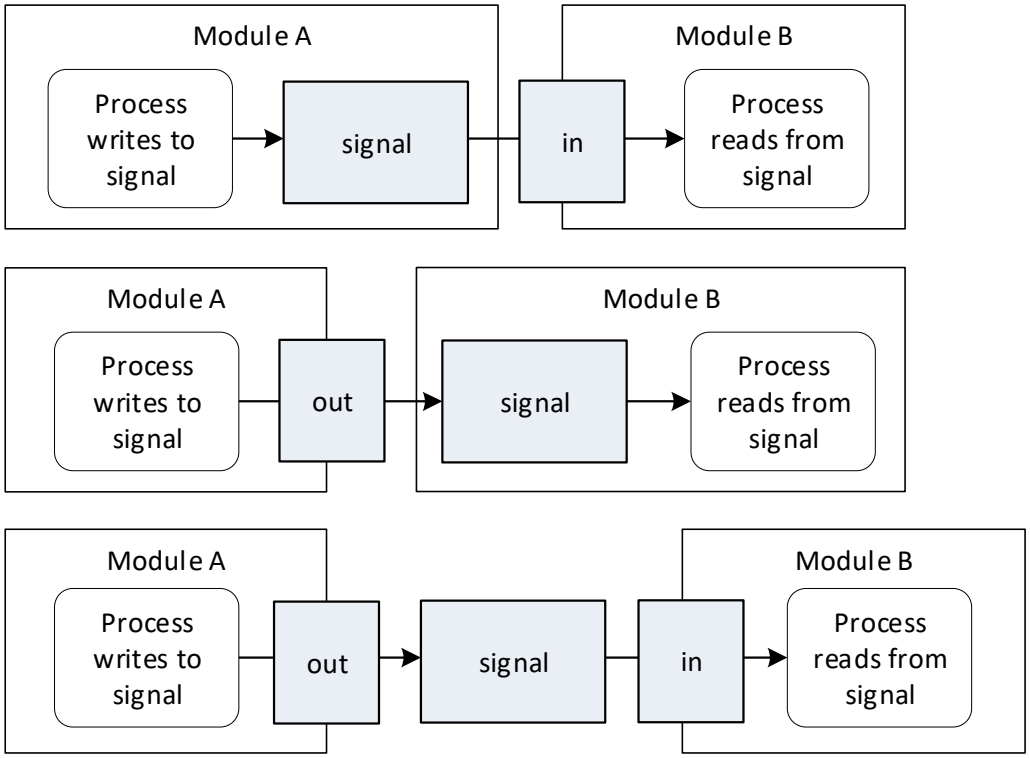
- Signal is wire or registers depends on context
  - sct_signal<T>
  - can be written in 1 process, read in many process
  - process reads signal should include it into sensitivity list

Signals and ports example

```
using T = sc_uint<16>;
sc_in<bool>          clk{"clk"};
sc_in<bool>          nrst{"nrst"};
sct_in<bool>         enbl{"enbl"};
sct_signal<T>        data{"data"};
sct_out<T>           out{"out"};

explicit A(const sc_module_name& name) :
sc_module(name) {
    SCT_THREAD(threadProc, clk);
    sensitive << enbl;
    async_reset_signal_is(nrst, 0);

    SC_METHOD(methProc);
    sensitive << data;
}
```

```
void threadProc(){
    data = 0;          // Reset logic
    wait();
    while (true) {
        if (enbl) data = data.read()+1;
        wait();
    }
}
void methProc(){
    out = data.read()*2;
}
```

Ports and signals between modules

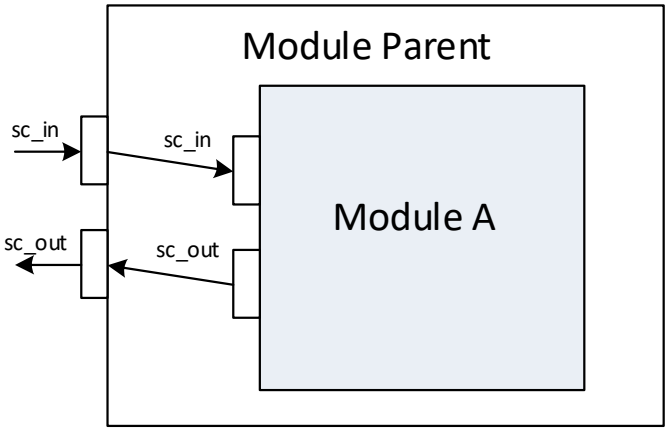


- Reduce some ports with signal to port connection
  - One child module can be bound to another one w/o extra signals in parent
  - Used defined function to bind module ports to signals

Ports bound example

```
class A : public sc_module {
  sct_in<T> in{"in"};
  void proc() {
     T var = in.read();          // Call sig.read()
}}

class B : public sc_module {
  sct_out<T> out{"out"};
  void proc() {
     out.write(val);            // Call sig.write()
}}

class parent : public sc_module {
    A a{"a"};
    B b{"b"};
    sct_signal<T> sig{"sig"};
    parent(…) {
       a.in(sig);
       b.out(sig);
}}
```

Ports bound to parent module

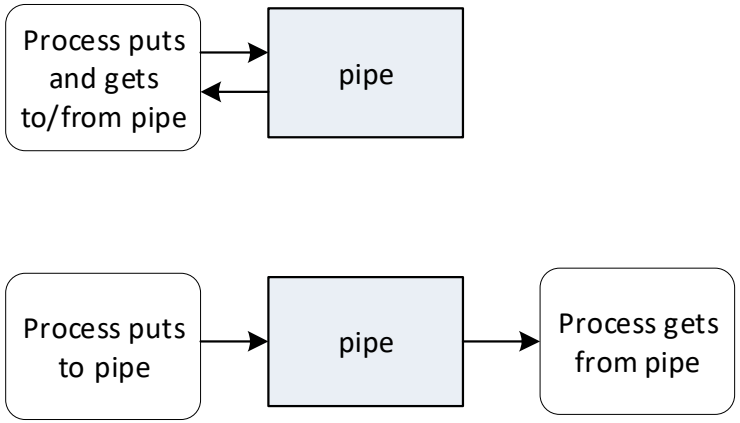



```
class A : public sc_module {
  sct_in<T> in{"in"};
  sct_out<T> out{"out"};
  void proc() {
     T var = in.read(); // Call parent in.read()
}}

class parent : public sc_module {
    A a{"a"};
    sct_in<T> in{"in"};
    sct_out<T> out{"out"};
    parent(…) {
       // Child module port bound to parent one
       a.in(in);
       a.out(out);
}}
```

SINGLE SOURCE LIBRARY FOR HIGH LEVEL MODELLING AND DIGITAL DESIGN

Part IV. Other channels

Pipe



- Intended to pipeline combinational logic and enable re-timing
  - can be used in method/thread
  - supports put bubbles and get backpressure
  - in generated SV replaced with specified component

Pipe parameters

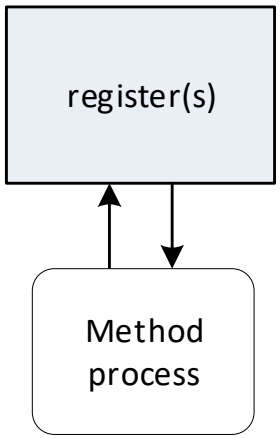
```
template<
    class T,
    unsigned N,                         // Number of pipeline registers, one or more
    class TRAITS = SCT_CMN_TRAITS,      // Clock edge and reset level traits
    bool TLM_MODE = SCT_CMN_TLM_MODE>   // Cycle accurate (0) or fast simulation (1) mode
>
class sct_pipe (const sc_module_name& name,
         bool addInReg = 0,             // Add input register not moved by re-timing
         bool addOutReg = 0)            // Add output register not moved by re-timing
};
```

Pipe example

```
sc_in<bool>           SC_NAMED(clk);
sc_in<bool>           SC_NAMED(nrst);
sct_target<T>         SC_NAMED(run);
sct_initiator<T>      SC_NAMED(resp);
sct_fifo<T, 5>        SC_NAMED(pipe);

explicit A(const sc_module_name& name) :
sc_module(name) {
    run.clk_nrst(clk, nrst);
    resp.clk_nrst(clk, nrst);
    pipe.clk_nrst(clk, nrst);

    SC_METHOD(methProc);
    sensitive << pipe << run << resp;
}
```

```
void methProc() {
    run.reset_get();
    resp.reset_put();
    pipe.reset();

    if (pipe.ready() && run.request()) {
        // Heavy computation to be pipelined
        T data = compute(run.get());
        pipe.put(data);
    }
    if (pipe.request() && resp.ready()) {
        resp.put(pipe.get());
    }
}
```

Register



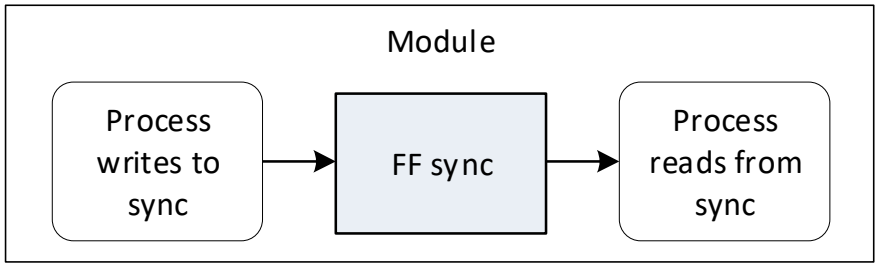
- Register used to have state in method process
  - Register value updated at clock edge
  - Used to avoid extra clocked thread process
- Register is written by one process
- Register is read
  - typically, by the same process as written
  - other method processes
- Register should be added to sensitivity list of process where it is read

Register example

```
sct_signal<bool>    req{"req"};
sct_register<T>     cntr{"cntr"};

A(sc_module_name name) : sc_module(name) {
    cntr.clk_nrst(clk, nrst);

    SC_METHOD(methProc);
    sensitive << req << cntr;
}
```

```
void methProc() {
    cntr.reset();
    // Register counts number of request up to N
    if (cntr.read() > N) {
        cntr.write(0);
    } else
    if (req) {
        cntr.write(cntr.read()+1);
    }
}
```

Flip-Flop Synchronizer



- Synchronizer 1 or 2 FF, 1bit data (bool)
  - Synchronizer can be used in thread and method processes

- Synchronizer implements sct_inout_if
- `reset()` function to be called in thread process reset section only

```
template <unsigned SyncType,     // Number of FF: 1 or 2
          bool RstVal,           // Reset value
          class TRAITS = SCT_DEFAULT_TRAITS>
class sct_ff_synchronizer {};
```

Synchronizer example

```
struct A : public sc_module
{
  sct_target<T> targ{"targ"};
  sct_ff_synchronizer<2> sync{"sync"};
  sct_out<bool> out{"out"};

  A(sc_module_name name) : sc_module(name) {
     sync.clk_nrst(clk, nrst);

     SC_METHOD(writeProc);
     sensitive << targ;

     SC_METHOD(readProc);
     sensitive << sync;
}
```

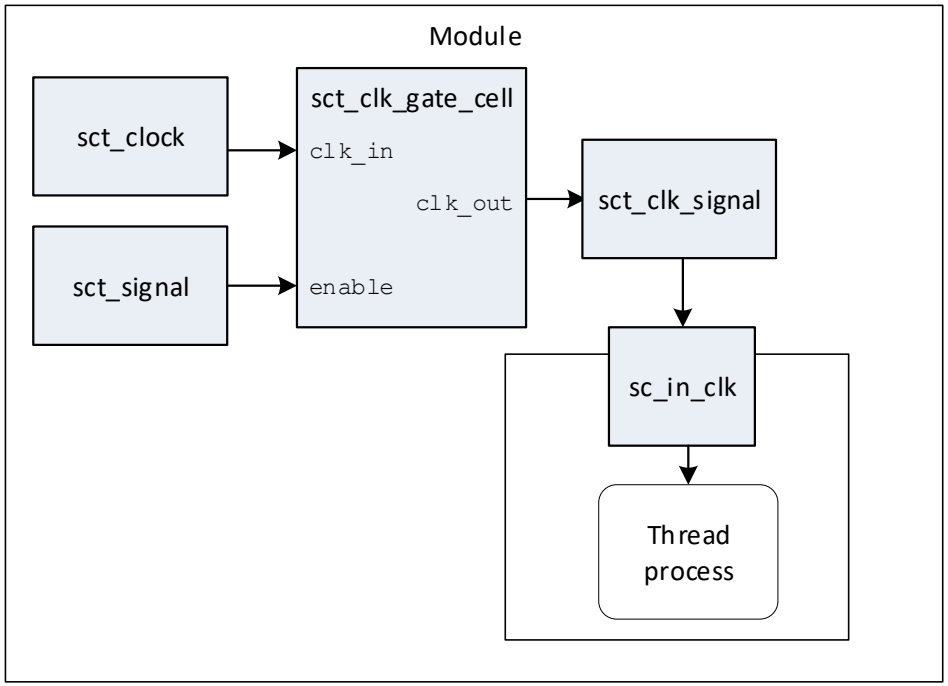
```
void writeProc() {
    sync = 0;
    if (targ.request()) {
        sync = targ.get();
    }
}
void readProc() {
    out = sync.read();
}
```

Clock generator

- sct_clock<> is clock generator like sc_clock with enable/disable control
  - In fast simulation mode only clock period is used

```
// Enable clock activity, clock is enabled after construction
void enable();
// Disable clock activity, can be called at elaboration phase
void disable();
// Register clock gate signals/ports to control clock activity
void register_cg_enable(sc_signal_inout_if<bool>& enable);
// Get clock period
const sc_time& period() const;


sct_clock<>     clk{"clk", 1, SC_NS};
explicit A(const sc_module_name& name) : sc_module(name) {
    if (SCT_CMN_TLM_MODE) {
        clk.disable();
}}
```

Clock gating



- sct_clk_gate_cell and sct_clk_signal used together to provide clock gating
- sct_clk_signal has no delta cycle delay

Clock gating example

```
SC_MODULE(A) {
    sc_in_clk               SC_NAMED(clk);
    sc_in<bool>             SC_NAMED(nrst);
    sc_in<bool>             SC_NAMED(clk_enbl);
    sct_clk_signal          SC_NAMED(clk_out);
    sct_clk_gate_cell       SC_NAMED(clk_gate);
    sc_in<bool>             SC_NAMED(clk_in);

    explicit A(const sc_module_name& name) : sc_module(name) {
        clk_gate.clk_in(clk);           // Clock input
        clk_gate.enable(clk_enbl);      // Gate clock input
        clk_gate.clk_out(clk_out);      // Gated clock output
        clk_in(clk_out);

        SCT_THREAD(thrdProc, clk_in.pos(), nrst);   // Use clock input bound to gated clock
        async_reset_signal_is(nrst, 0);
}};
```

Specify clock edge and reset level

- Clock edge and reset level normally are the same for the design
  - To update them for whole design `SCT_DEFAULT_TRAITS` should be defined
  - To specify clock edge/reset level for individual channels, template parameter should be used

```
sc_in<bool>     nrst{"nrst"};
sct_target<T, SCT_NEGEDGE_POSRESET>        run{"run"};
sct_initiator<T, SCT_POSEDGE_NEGRESET>     resp{"resp"};

A(sc_module_name name) : sc_module(name) {
   SCT_THREAD(thrdProc, clk);  // SCT_DEFAULT_TRAITS::CLOCK used here
   async_reset_signal_is(nrst, SCT_DEFAULT_TRAITS::RESET);
}
```

Reset/initialization for channels

- SS channels need to be reset with specified reset(), reset_get() and reset_put()
  - In thread every channel used in this process should be initialized in the reset section
  - In method every channel used in this process is initialized in the beginning of the process or accessed at all execution path in the process code

```
sct_initiator<T>  init{"init"};
sct_target<T>     targ{"targ"};
sct_fifo<T, 2>    fifo{"fifo"};
sct_synchnoizer<2> sync{"sync"};
void thrdProc() {
    // Mandatory required
    init.reset_put();
    targ.reset_get();
    fifo.reset_put();  // or reset_get()
    sync.reset();
    wait();
    ...}
```

```
void methodProc() {
   // Optional, if accessed not at all paths
   init.reset_put();
   targ.reset_get();
   fifo.reset_put();  // or reset_get()
   sync.reset();
   ...}
```

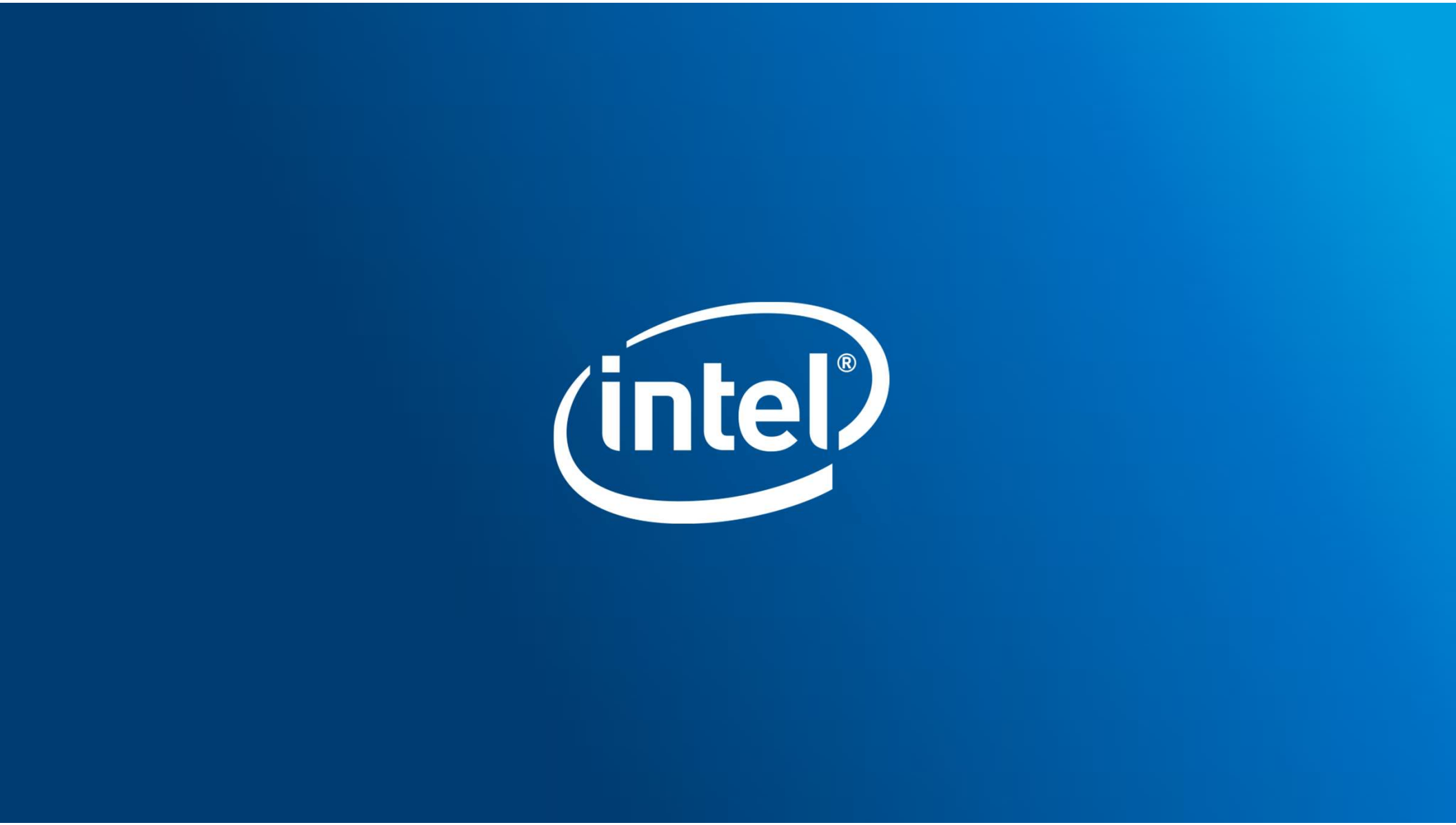
C++ struct as channel payload

```
struct Rec_t {
    bool enable;
    sc_uint<16> addr;
    Rec_t() : enable(false), addr(0) {}
    bool operator == (const Rec_t& r) const
       return (enable==r.enable && addr==r.addr);
}};
namespace std {
inline ::std::ostream& operator << (
    ::std::ostream& os, const Rec_t& r) {
    os << r.enable << r.addr; return os;
}}
namespace sc_core {
  void sc_trace(sc_trace_file*, const Rec_t&,
                const std::string&) {
}}
```

```
sct_target<Rec_t>   run{"run"};

void methProc() {
   run.reset_get();
   if (run.request()) {
      Rec_t data = run.get();
   }
}
```

Contacts

- Wiki: https://github.com/intel/systemc-compiler/wiki/SingleSource-library
- GitHub repo: https://github.com/intel/systemc-compiler
- This training videos are available by request


intel®