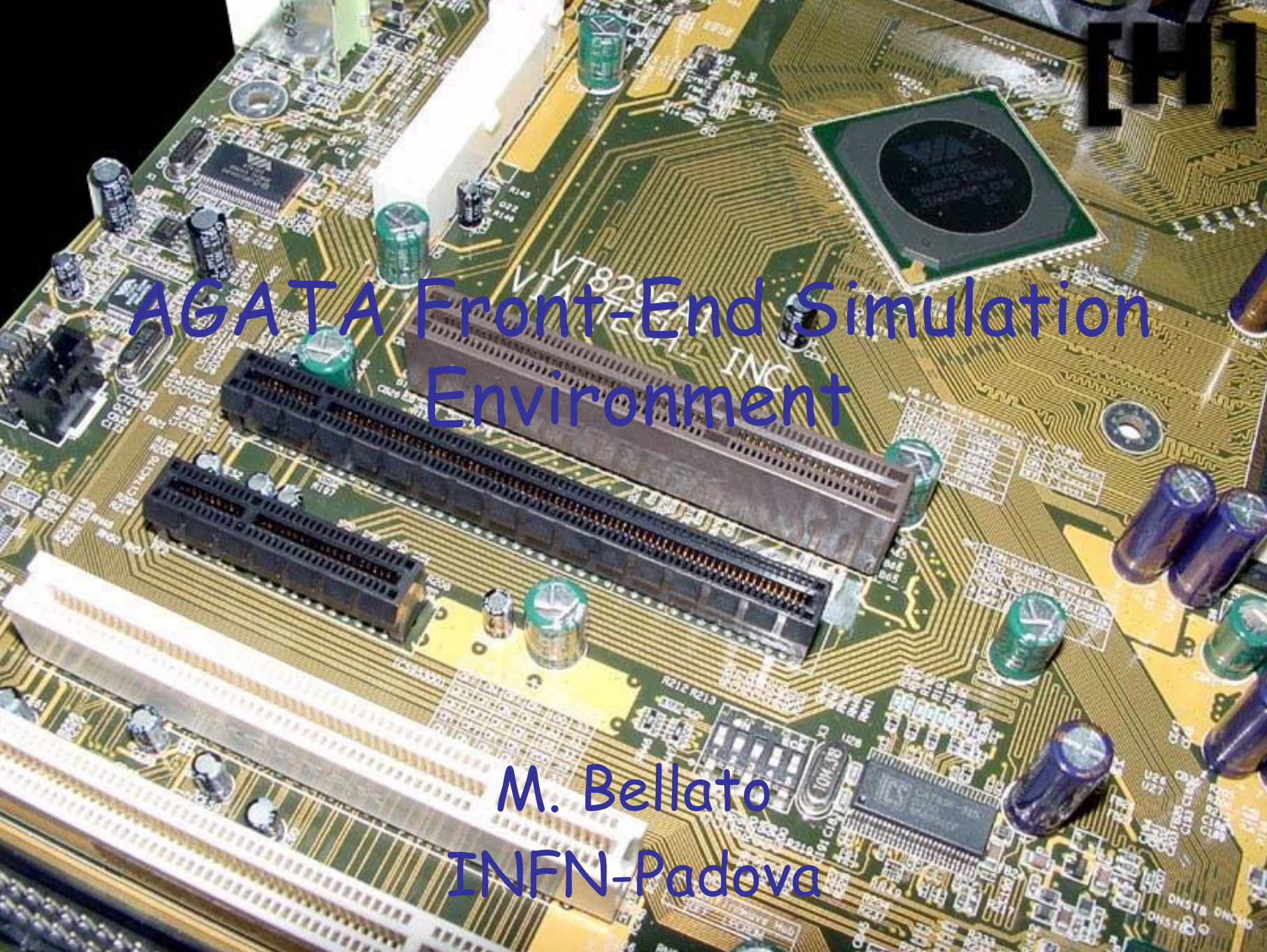
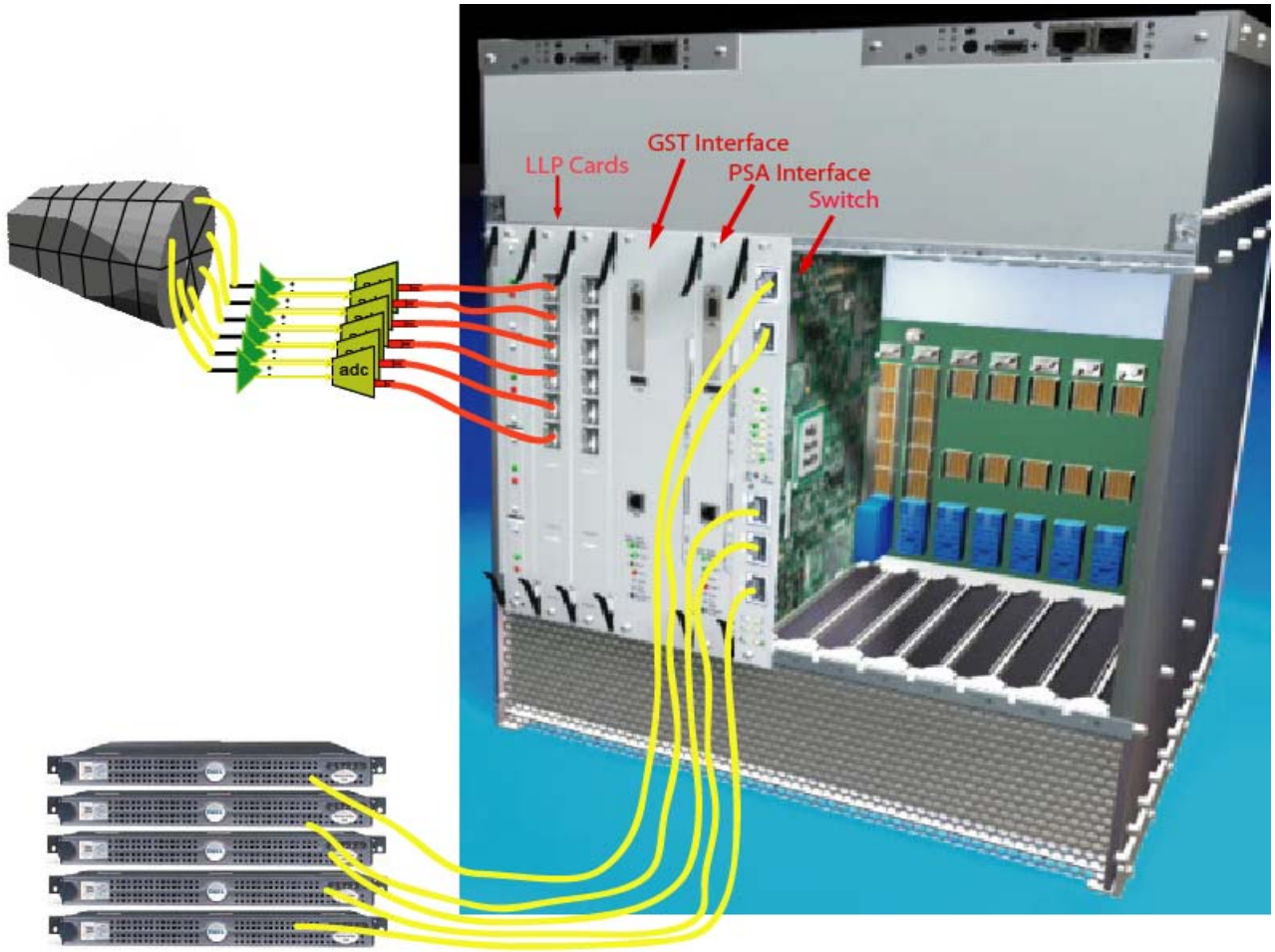


AGATA Front-End Simulation Environment

M. Bellato
INFN-Padova

What are we speaking about



Why do we need it ?

- Agata front-end is complex !
  - The readout must be fully pipelined to cope with high trigger rates
  - Need to gain insight of LLP - PSA Interface- GTS interactions
- Need to investigate behaviour with sampled data
  - We should assess correctness of event building before PSA, at any rate.
  - And dimension fifos and memory depths according to expected trigger rates and service times
- Need a reference model of the front-end hardware to be agreed by different working groups
  - A way to reduce misunderstandings
- Need a way to evaluate quickly the impact of changes
  - Engineering changes are unavoidable, but sometimes underestimated

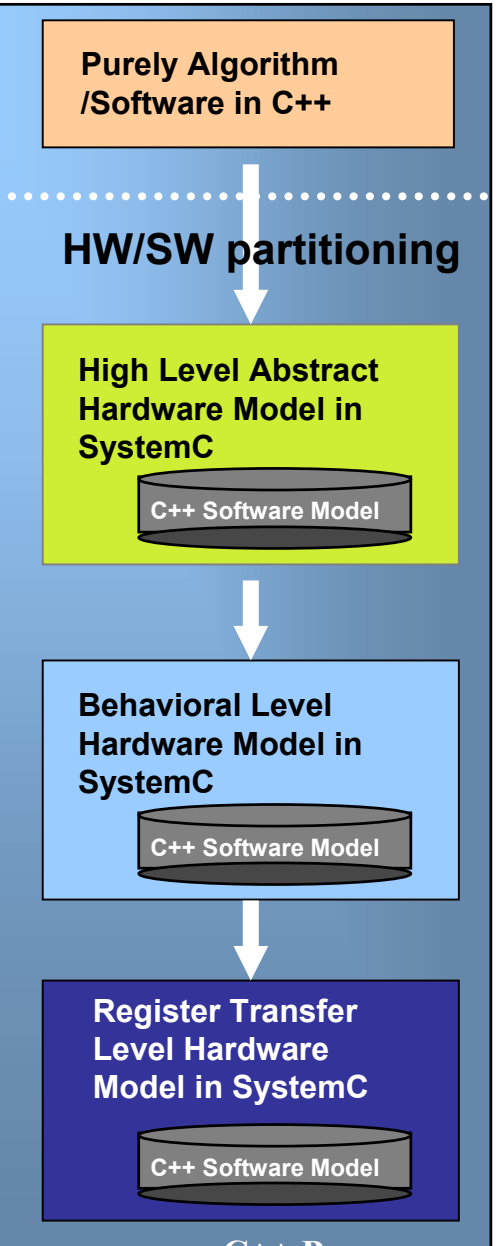
How do we get it ?

- System Level simulation
  - Complete
  - Abstraction is high
  - Lacks timing information
    - Not good for modeling front-end hardware
- Gate Level simulation
  - Accurate but abstraction is poor
    - The description of the model is huge and unmanageable
  - Timing information is complete
  - Simulation is unacceptably slow for complex models
- Cycle accurate simulation
  - Abstraction can be mixed
  - Tming information is accurate at the clock cycle
  - Model is compact
  - Simulation is fast

Tools

- SystemC chosen as simulation environment
  - http://www.systemc.org
  - Open environment for system level design
  - Can mix structural, RTL and behavioural hardware descriptions.
  - Open source, is a C++ library
  - Dumps traces in VCD or WIF formats
  - Can be debugged with any C debugger
  - Is synthesizable
  - Can be simulated from within major EDA tools (Mentor Modelsim, Cadence NCsim, ...)

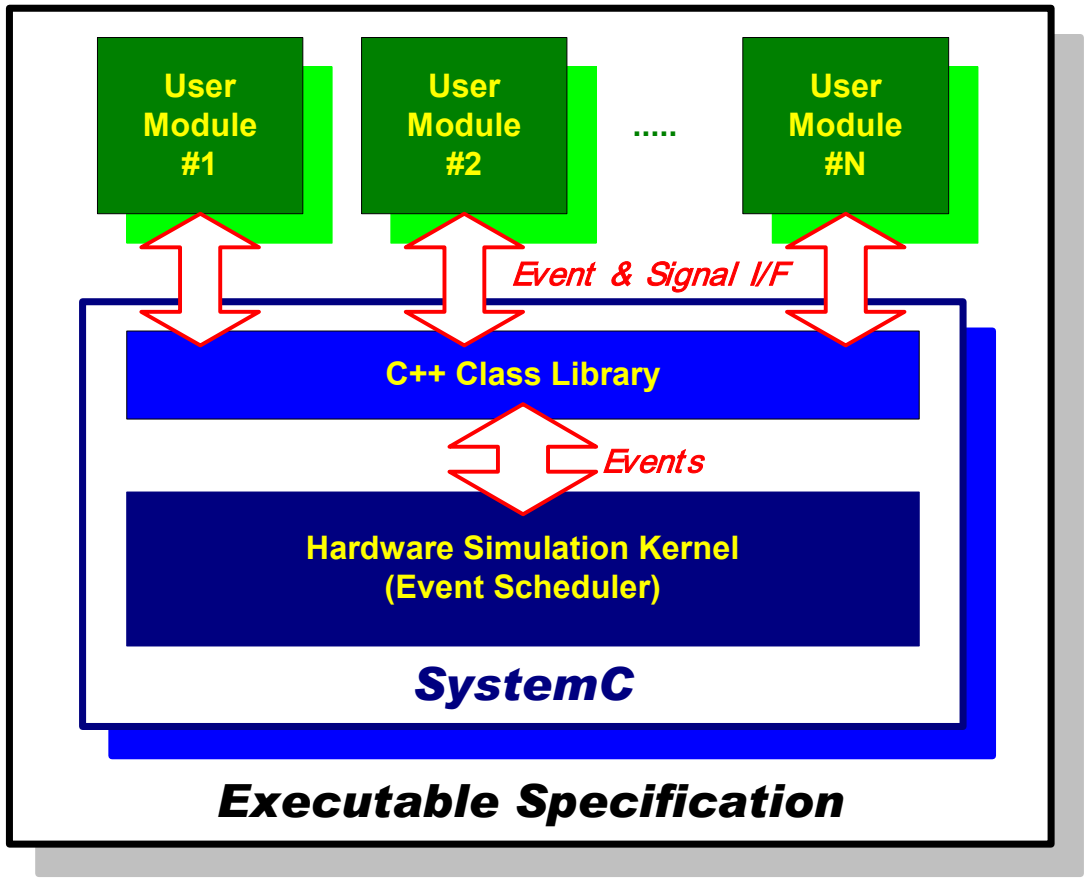
Highlights



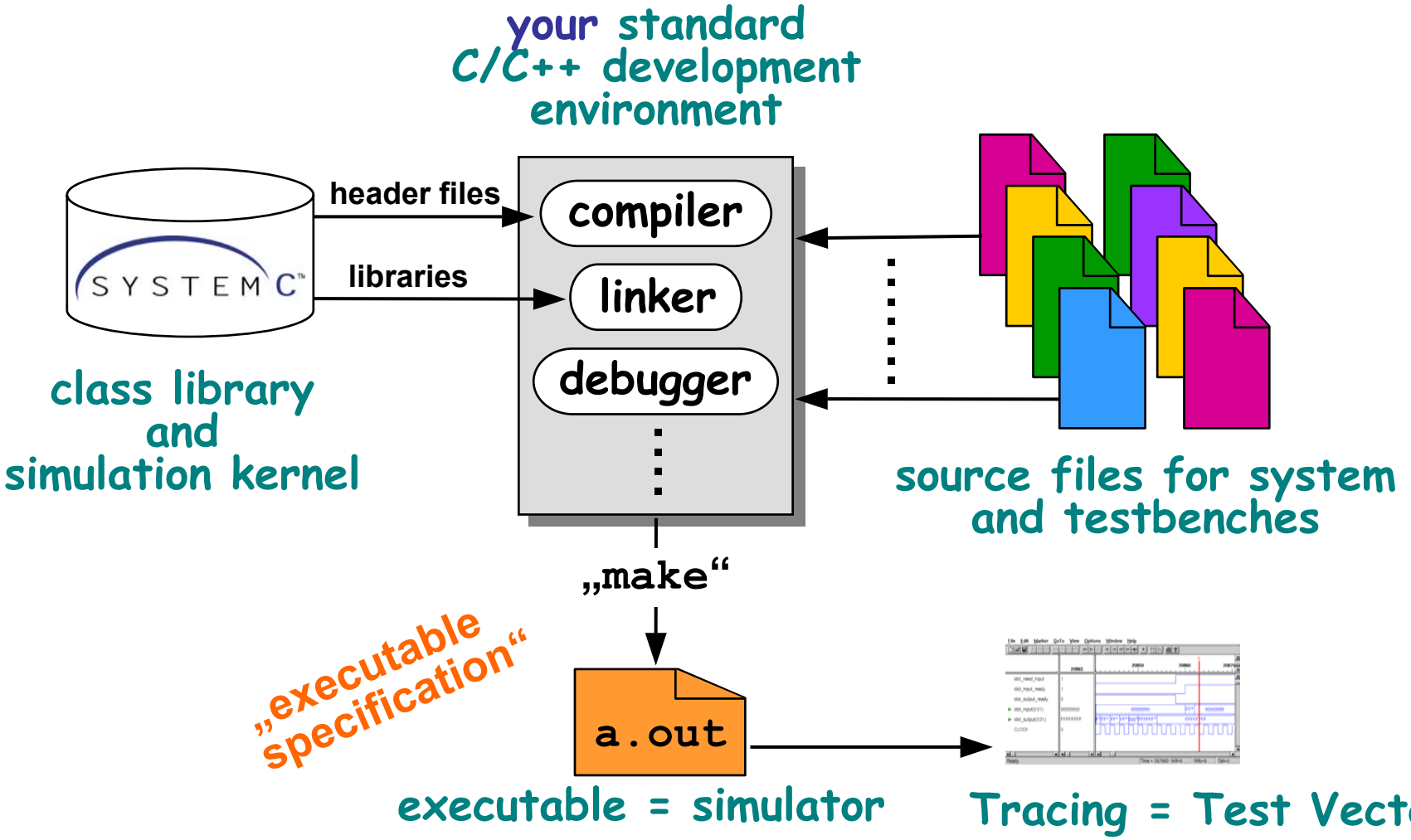
Interface in a C++ environment

- Modules
  - **Container class includes hierarchical Entity and Processes**
- Processes
  - **Describe functionality, Event sensitivity**
- Ports
  - **Single-directional(in, out), Bi-directional(inout) mode**
- Signals
  - **Resolved, Unresolved signals**
- Rich set of port and signal types
- Rich set of data types
  - **All C/C++ types, 32/64-bit signed/unsigned, fixed-points, MVL, user defined**
- Clocks
  - **Special signal, Timekeeper of simulation and Multiple clocks, with arbitrary phase relationship**
- Cycle-based simulation
  - **High-Speed Cycle-Based simulation kernel**
- Multiple abstraction levels
  - **Untimed from high-level functional model to detailed clock cycle accuracy RTL model**
- Communication Protocols
- Debugging Supports
  - **Run-Time error check**
- Waveform Tracing
  - **Supports VCD, WIF, ISBD**

SystemC & Modules



Simulation Flow



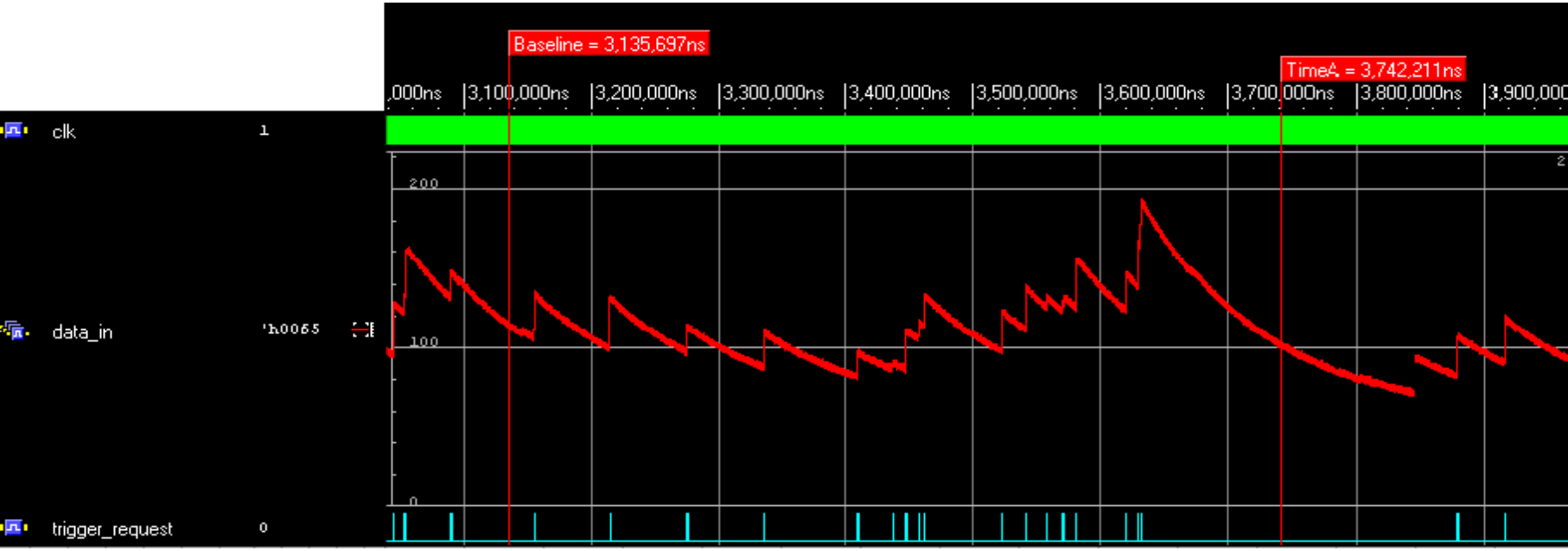
Example - SCC/1

```
SC_MODULE(SCC) {
     sc_in_clk clk;                          // The clock
     sc_in<bool> reset;                      // Reset
     sc_out<bool> trigger_request;           // to GTS
     sc_out<bool> local_trigger;             // to mezzanines
     sc_in<sc_uint<ADC_BITS> > data_in;      // sample input
     sc_in<sc_uint<12> > threshold;          // threshold
     sc_in<sc_uint<10> > hold_time;          // trigger duration

     // local signals
     sc_uint<ADC_BITS> pipe[NSAMPLE];
     sc_signal <sc_uint<ADC_BITS> > result;
     sc_signal <sc_int<ADC_BITS> > average_out;

     ....

     void count();     // core counting process
     void average (); // average process
     void compare() ; // compare to a threshold

     SC_CTOR(SCC) {
        SC_METHOD(count);
        sensitive_pos << clk << reset;
        SC_METHOD(average);
        sensitive_pos << clk << reset;
        SC_METHOD(compare);
        sensitive_pos << clk << reset;
     }};
```

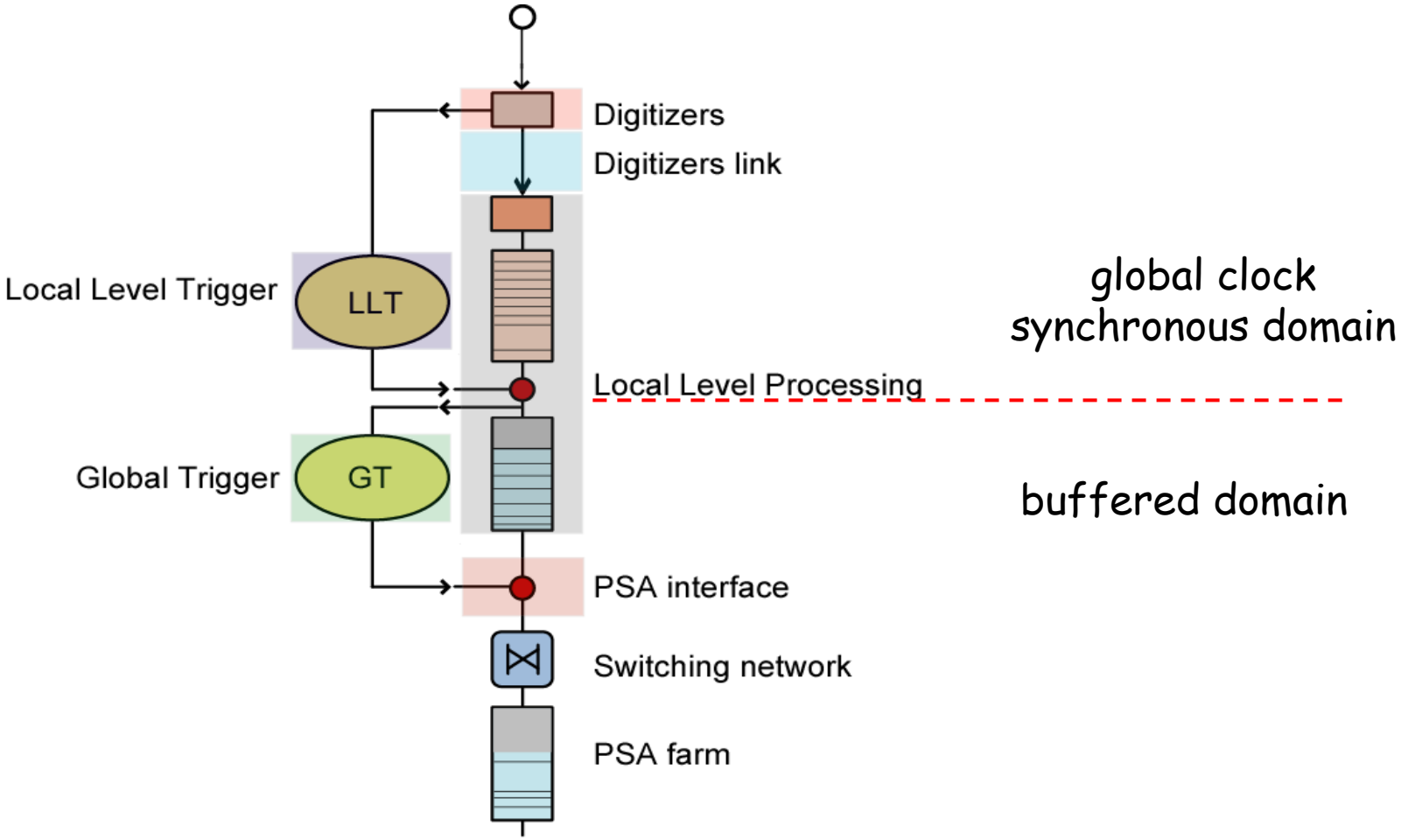
Example - SCC/2

```
//
// SCC accumulator - count the number of samples
// in the pipe that are greater than the midsample
// and smaller than it
//

void SCC::count() {
int i;
   if (reset) { // Reset operations
        for (i=0; i< NSAMPLE; i++) {
                /* synopsys unroll */
                pipe[i] = 0;
                }
   } else {
        tmp = data_in.read();
        pacc=0; nacc=0;
        for (i=0;i<NSAMPLE-1;i++){      // shift
                /* synopsys unroll */
                pipe[i] = pipe[i+1];
                }
        pipe[NSAMPLE-1] = tmp;
        for (i=0; i< MIDSAMPLE; i++) {  // forward count
                /* synopsys unroll */
                if(pipe[i] < pipe[MIDSAMPLE]) nacc++;
                }
        for (i=MIDSAMPLE+1; i< NSAMPLE; i++) {  // backward count
                /* synopsys unroll */
                if(pipe[i] > pipe[MIDSAMPLE]) pacc++;
                }
        result=pacc+nacc;
   } };
```

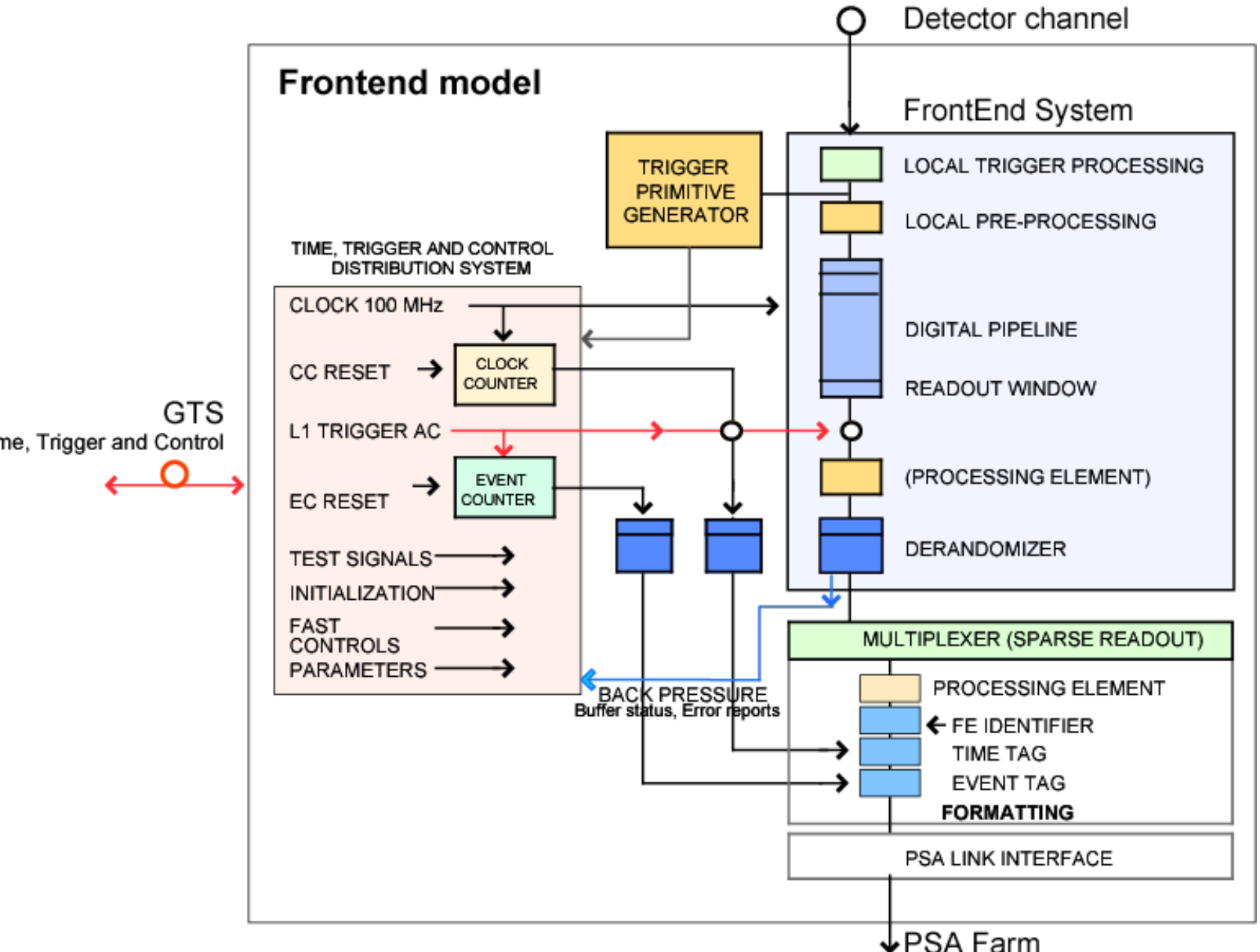
SCC Simulation Result



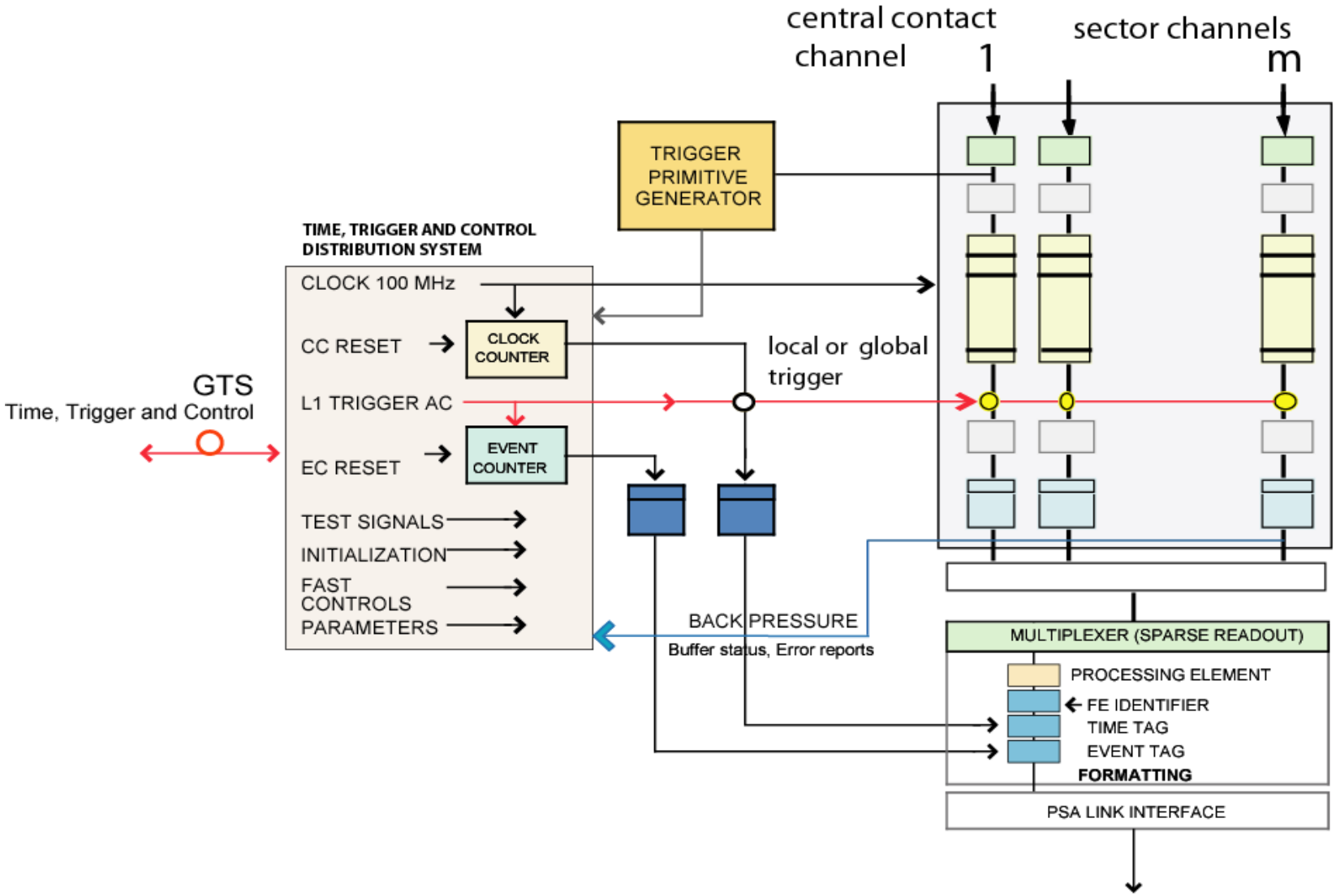
Front-end Data Flow



The Channel model



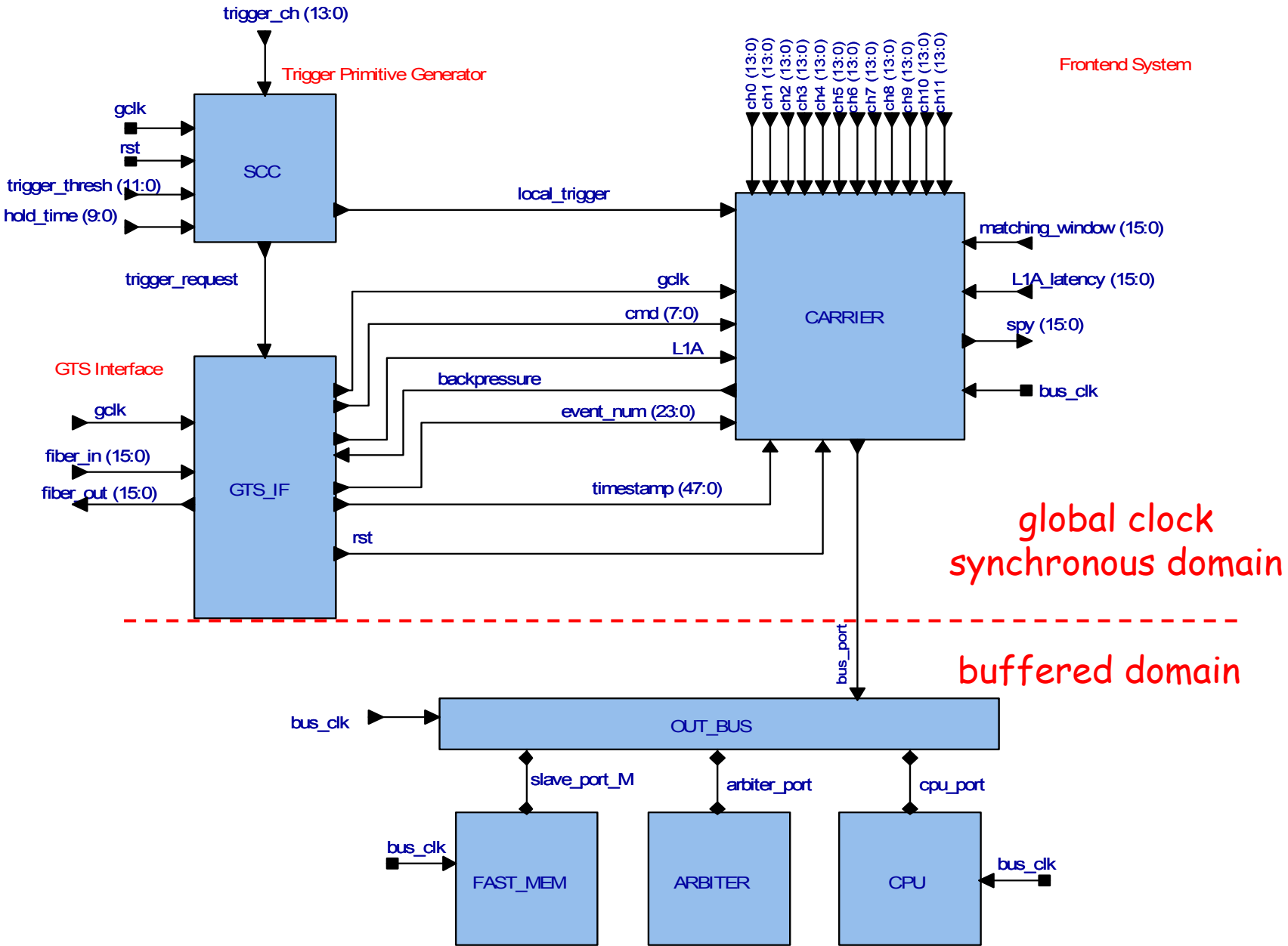
The Crystal model



The simulation hierarchy/1

- agata_fe/
  - bus/ — bus model + cpu
  - carrier/ — carrier = 2 mezzanines+readout
  - channel/ — channel - bottom of hierarchy
  - data/ — sampled waveforms
  - fes/ — fes = GTS + 1 carrier
  - gts/ — GTS master + fanout + GTS interface
    - gts_master/
    - fanin_fanout/
    - gts_if/
  - makefile — global Makefile
  - mezzanine/ — mezzanine = 6 channels
  - mwd_fixed/ — fixed point mwd
  - mwd_float/ — floating point mwd
  - scc/ — local trigger algorithm
  - util/ — fifo, memory, random models, etc

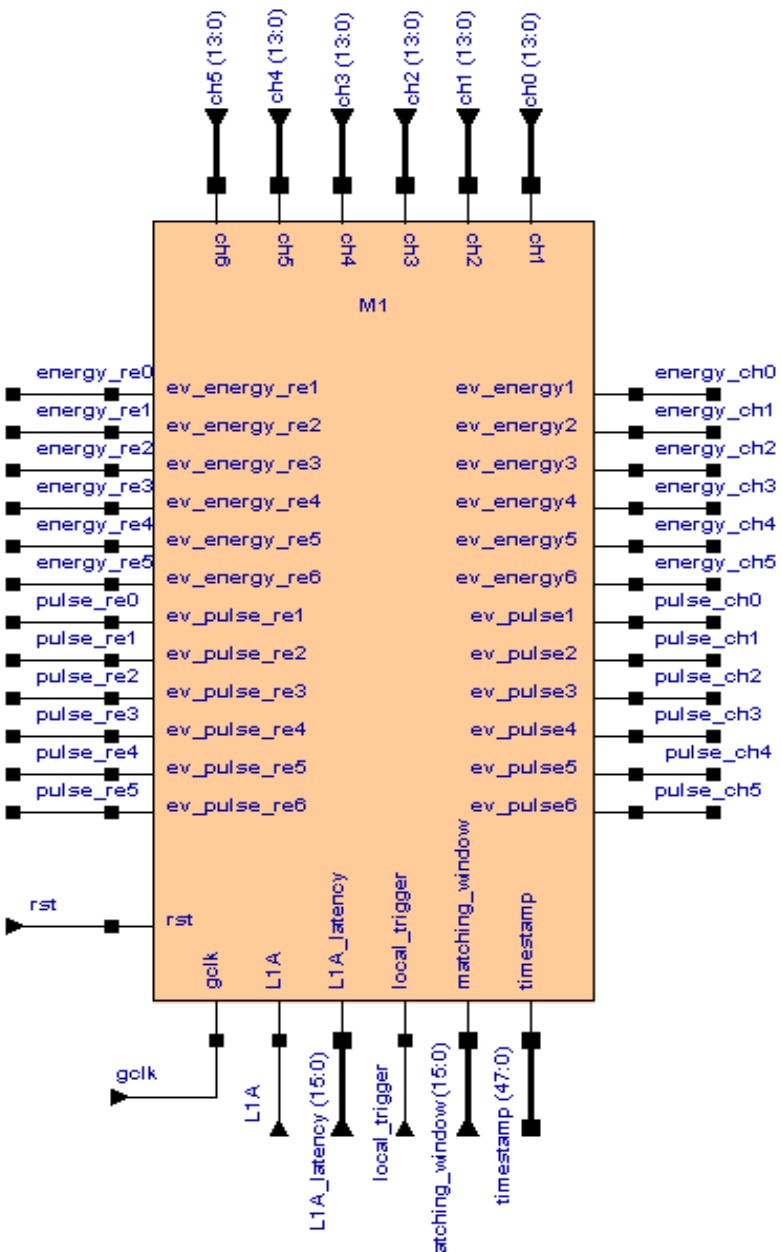
FES Block Diagram

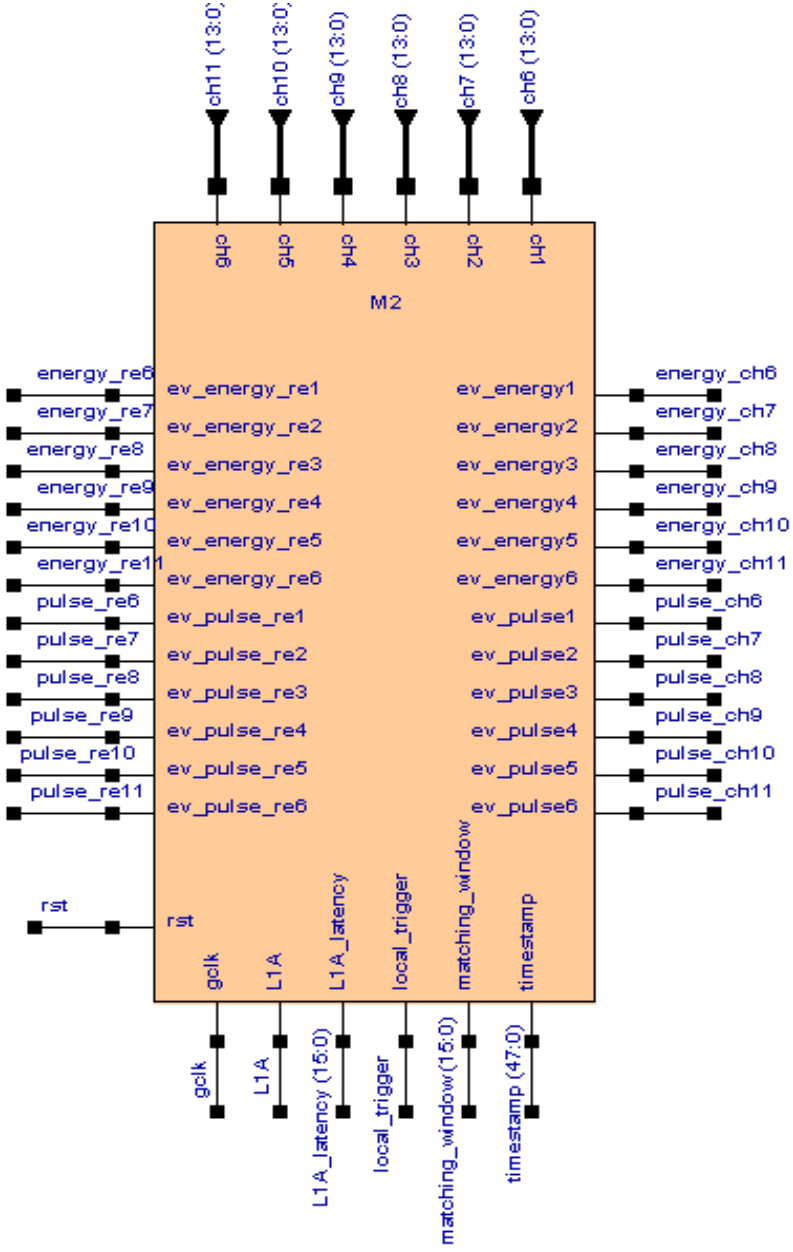


Fes.h

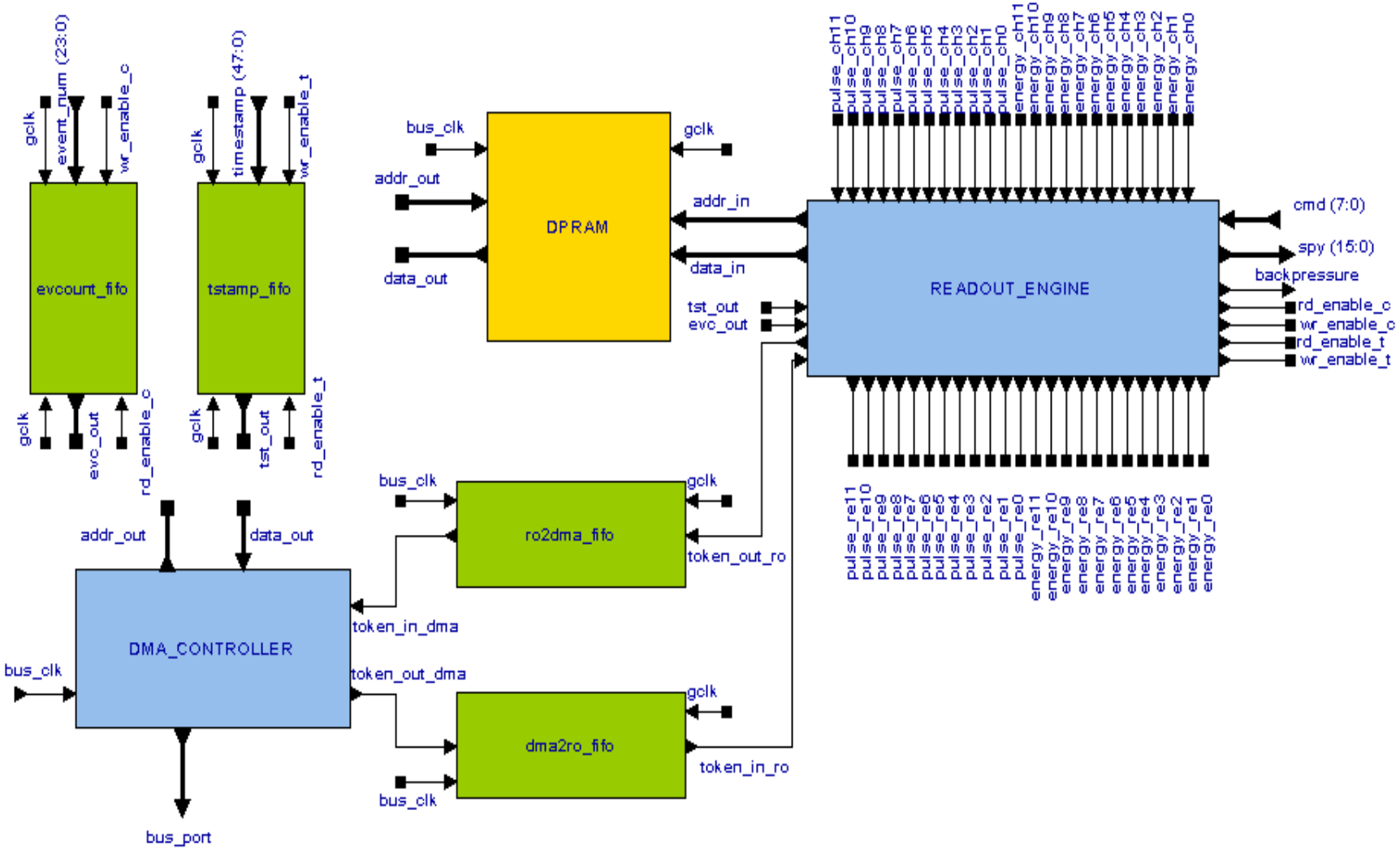
```
class FES: public sc_module
{
public:
  SC_HAS_PROCESS(FES);

  // ports
  sc_in_clk gclk;                                  // GTS clock
  sc_in_clk bus_clk;                               // bus clock
  sc_in<sc_uint<16> > fiber_in;                    // from GTS
  sc_out<sc_uint<16> > fiber_out;                  // to GTS
  sc_in<sc_int<ADC_BITS> > ch[12];                 // 12 data channels
  sc_in<sc_uint<ADC_BITS> > trigger_ch;            // 1 trigger data channel
  sc_in<sc_uint<12> > trigger_thresh;              // threshold for SCC
  sc_in<bool> L1A;                                 // validation from GTS
  sc_in<bool> backpressure;                        // backpressure to GTS
  sc_in<sc_uint<16> > event_count;                 // event counter
  sc_in<sc_uint<16> > matching_window;             // trigger matching window
  sc_in<sc_uint<16> > L1A_latency;                 // latency value for L1A
  sc_in<sc_uint<10> > hold_time;                   // hold time for trigger request
  sc_out<sc_uint<16> > spy;                        // a spy channel
  sc_out<bool> trigger_request;                    // trigger request to GTS
  sc_in<bool> local_rst;                           // local reset

private:
  fast_mem *M9;                                    // a memory bank
  arbiter *arb;                                    // a bus arbiter
  bus *bus1;                                       // a bus object
  CARRIER *c1;                                     // the carrier
  CPU *cpu;                                        // a cpu
  SCC *scc;                                        // a local trigger facility
  GTS_IF *gts_if;                                  // the GTS mezzanine
....
```

Carrier Block Diagram/1





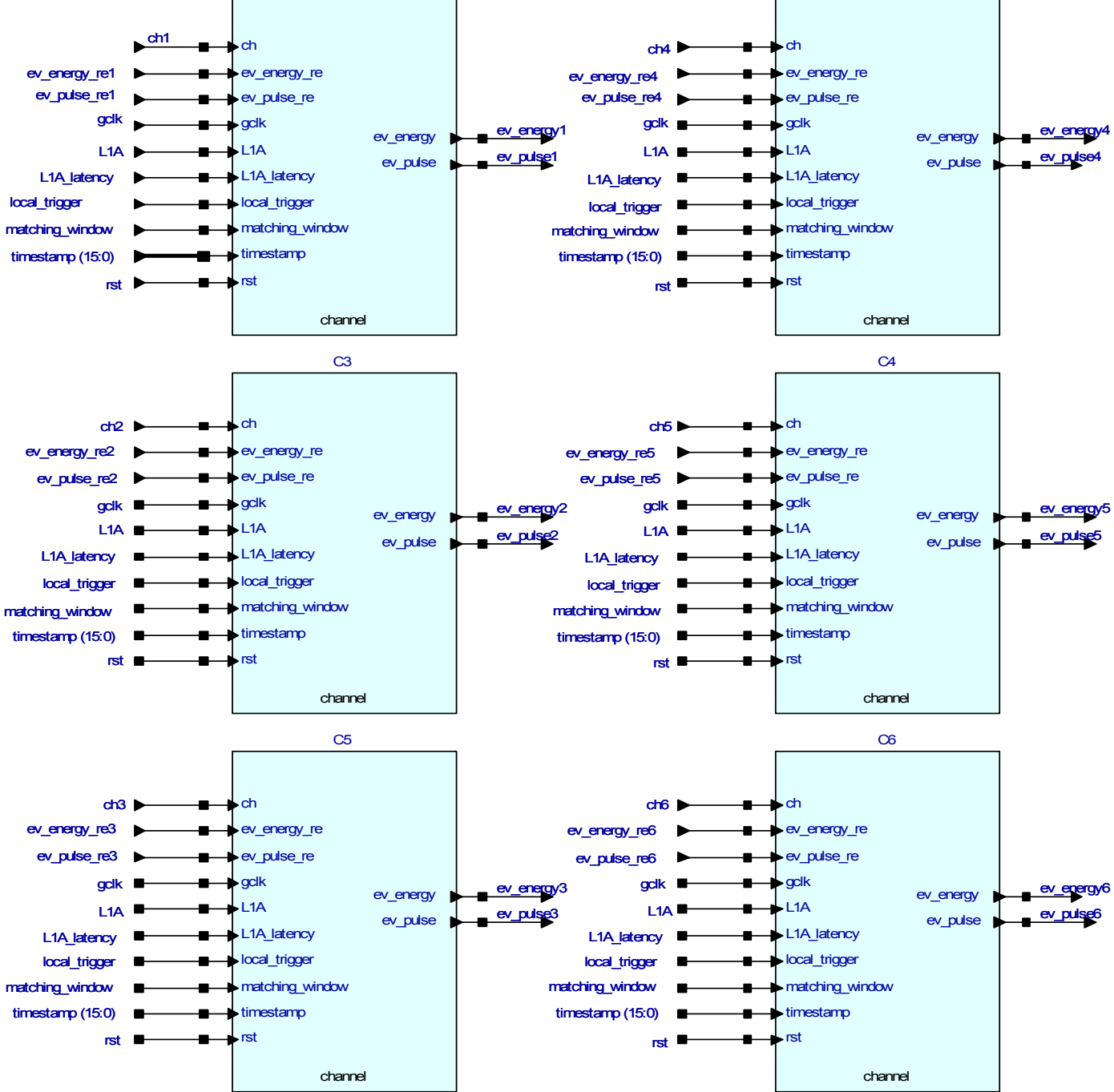
Carrier Block Diagram/2



Carrier.h

```
class CARRIER: public slave_if, public sc_module
{
public:
  SC_HAS_PROCESS(CARRIER);

  // ports
  sc_in_clk gclk;
  sc_in_clk bus_clk;
  sc_in<bool> rst;
  sc_in<bool> L1A;
  sc_in<bool> local_trigger;
  sc_in<sc_uint<48> > timestamp;
  sc_in<sc_uint<16> > event_count;
  sc_in<sc_uint<16> > matching_window;
  sc_in<sc_uint<16> > L1A_latency;
  sc_in<sc_int<DSIZE_P> > ch[12];
  sc_out<sc_int<32> > data;
  sc_out<sc_uint<16> > spy;
  sc_port<blocking_if, 1> bus_port;                // bus port

private:
  MEZZANINE *m1, *m2;                              // the mezzanines
  sc_uint<16> DPRAM[RO_BUFSIZE];                   // the dual port output buffer
  FIFO <sc_uint<48>, MAX_L1A_SERVICE > *tstamp_fifo; // FIFO for timestamps
  FIFO <sc_uint<16>, MAX_L1A_SERVICE > *evcount_fifo; // FIFO for event counter
  FIFO <sc_uint<1>, 2 > *ro2dma, *dma2ro;          // token exchange RO-DMA

  void readout_engine();
  void dma_controller();
  void tags_storage();
  void wait_loop();
  ......
```

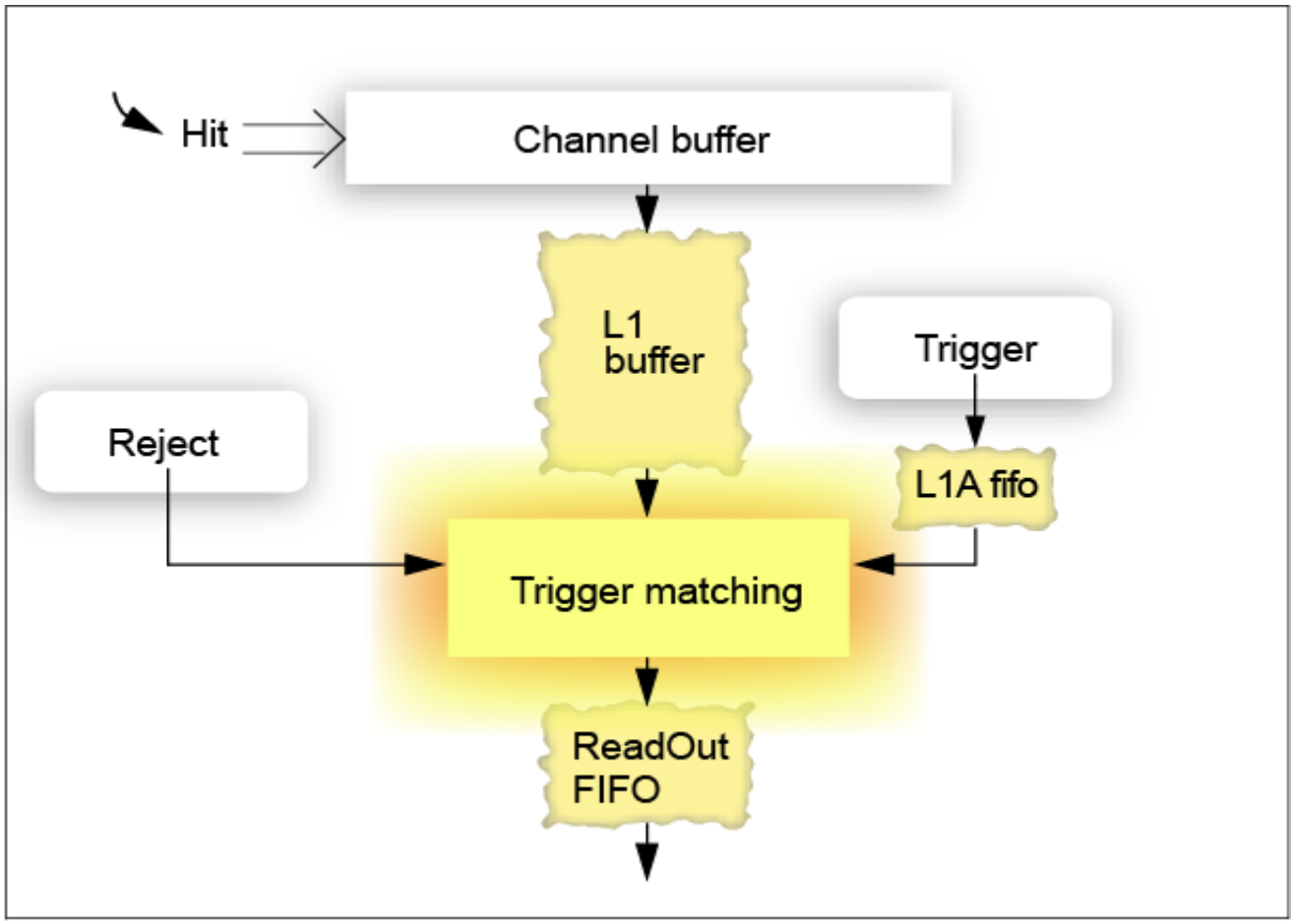
Mezzanine Block Diagram



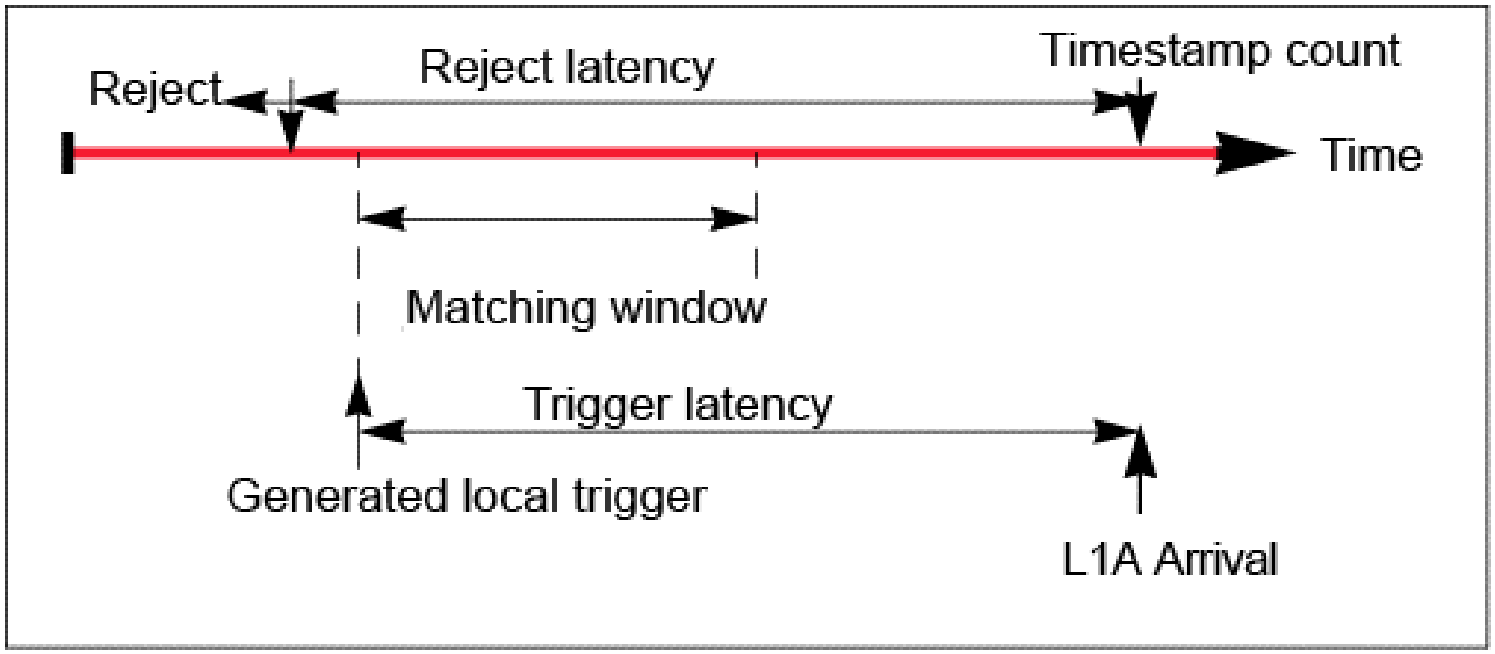
Mezzanine.h

```
SC_MODULE(MEZZANINE)
{
  .....
  sc_in<sc_int<DSIZE_P> > ch1, ch2, ch3, ch4, ch5, ch6;
  sc_out<sc_int<DSIZE_EV> > ev_pulse1,ev_pulse2,ev_pulse3;
  sc_out<sc_int<DSIZE_EV> > ev_pulse4,ev_pulse5,ev_pulse6;
  sc_out<sc_int<DSIZE_EV> > ev_energy1,ev_energy2,ev_energy3;
  sc_out<sc_int<DSIZE_EV> > ev_energy4,ev_energy5,ev_energy6;
  sc_in<bool> ev_pulse_re1, ev_pulse_re2, ev_pulse_re3;
  sc_in<bool> ev_pulse_re4, ev_pulse_re5, ev_pulse_re6;
  sc_in<bool> ev_energy_re1, ev_energy_re2, ev_energy_re3;
  sc_in<bool> ev_energy_re4, ev_energy_re5, ev_energy_re6
  sc_out<bool> ev_empty1, ev_empty2, ev_empty3, ev_empty4, ev_empty5, ev_empty6;
  sc_out<bool> en_empty1, en_empty2, en_empty3, en_empty4, en_empty5, en_empty6;

  CHANNEL *c1, *c2, *c3, *c4, *c5, *c6;

  SC_CTOR (MEZZANINE)
  {
    c1 = new CHANNEL ("c1");
    c2 = new CHANNEL ("c2");
    c3 = new CHANNEL ("c3");
    c4 = new CHANNEL ("c4");
    c5 = new CHANNEL ("c5");
    c6 = new CHANNEL ("c6");
    .....
```

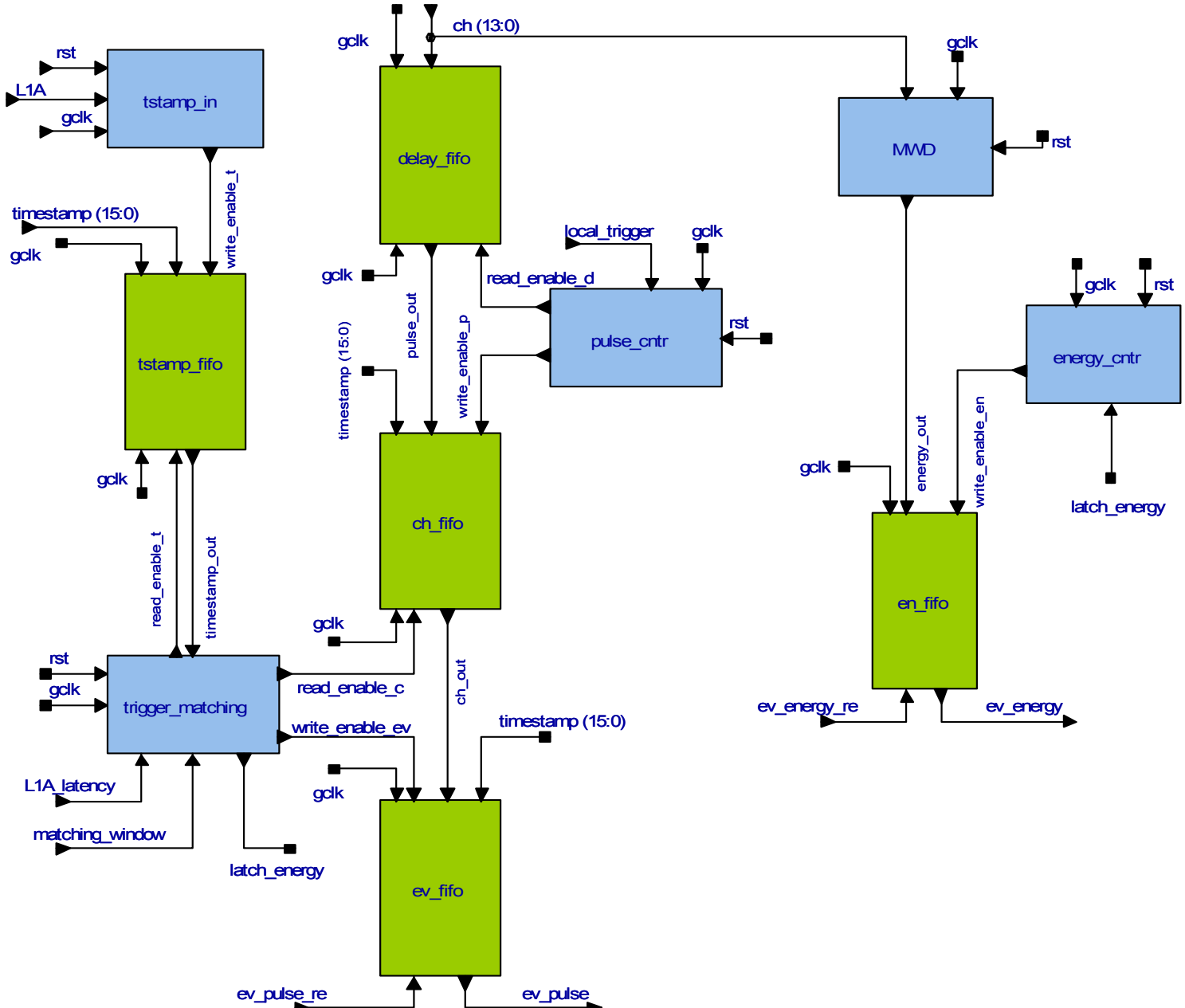
The Channel Architecture



Trigger Matching



Channel Block Diagram



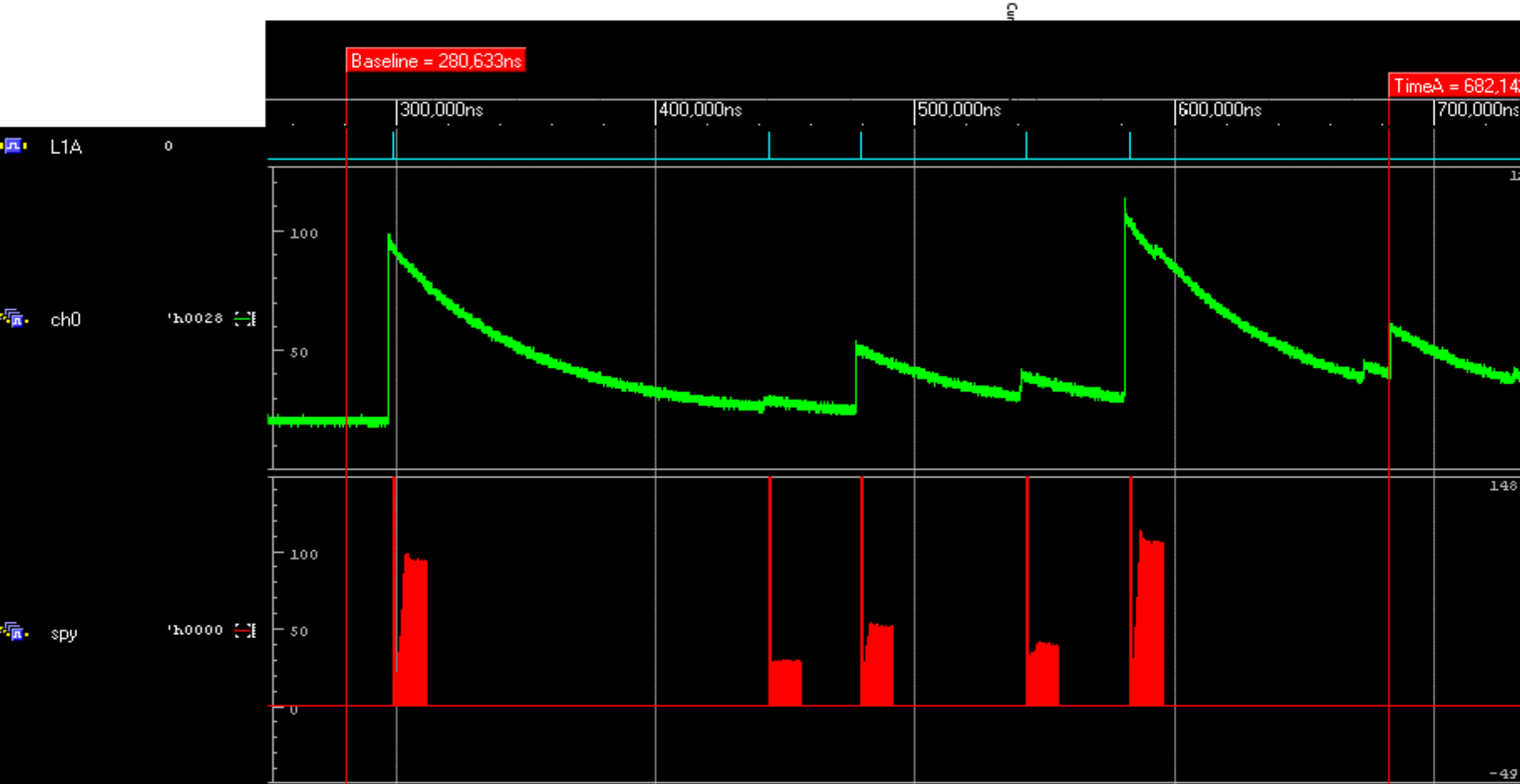
Channel.h

```
SC_MODULE(CHANNEL)
{
  sc_in_clk gclk;
  sc_in<bool> rst;
  sc_in<bool> L1A;
  sc_in<bool> local_trigger;
  sc_in<sc_uint<TSTAMP_SIZE> > timestamp_lsw;     // lower 16 bits of timestamp
  sc_in<sc_int<DSIZE_P> > ch;                     // sampled data in
  sc_in<bool> ev_pulse_re;                        // ev fifo read enable
  sc_in<bool> ev_energy_re;                       // en fifo read enable
  sc_out<bool> ev_empty;                          // empty fifos
  sc_out<bool> en_empty;                          // empty fifos
  sc_out<sc_int<DSIZE_EV> > ev_pulse;             // pulse out
  sc_out<sc_int<DSIZE_EV> > ev_energy;            // energy out
  sc_in<sc_uint<16> > matching_window;
  sc_in<sc_uint<16> > L1A_latency;

  FIFO <sc_uint<TSTAMP_SIZE>, MAX_L1A_SERVICE> * tstamp_fifo;
  FIFO <item_p, FIFOLEN_D> * delay_fifo;
  FIFO <sc_int<TSTAMP_SIZE+1>, FIFOLEN_P> * ch_fifo;
  FIFO <item_ev, FIFOLEN_EV> * ev1_fifo;
  FIFO <item_ev, FIFOLEN_EV> * en1_fifo;
  MWD * mwd_ch;

  void pulse_cntr();                              // pulse controller
  void ener_cntr();                               // energy controller
  void recording_machine();                       // storage controller
  void manage_rollover();                         // time rollover handler
  void trigger_matching();                        // trigger match
  .....
```

Channel.cc sample

```
oid CHANNEL::trigger_matching() {
   ……
   case MATCH_S_START:
        trigger_matched = false;
        ev_discard = false;
       // extract timestamp and computes elapsed time
       // for the oldest event in the fifo
       if(ch_fifo->IsEmpty() == false) {
          // take first fifo val without removing
          event_time= ch_fifo->Read();
          if(event_time & (1<<TSTAMP_SIZE)) {
             // it is really a time stamp !
             event_time &= (1<<TSTAMP_SIZE)-1;
             elapsed = (timestamp_lsw.read()- event_time) % (1<<TSTAMP_SIZE);
             trigger_request_time =(L1A_time - L1A_latency.read()) % (1<<TSTAMP_SIZE);
             upper_limit=(trigger_request_time+matching_window.read())%(1<<TSTAMP_SIZE);
             manage_rollover();
             if(trigger_matched || ev_discard)   {
                dump_count=1;
                if(trigger_matched) {
                   ev1_fifo->Write(SOF);
                   latch_energy = true;
                   }
                next_state = MATCH_S_DUMP_START;
                }
             else next_state = CHECK_L1A_FIFO; //event is newer than L1A arrival time
             }
          else next_state = ERROR; //event is misaligned
          }
        else next_state = CHECK_L1A_FIFO; //event fifo is empty
        break;
```

Trigger Matching run example



Interesting Parameters

- Channel Fifo's occupancy vs trigger rate
  - Overflow probability
  - Fifo's depth
- Global dead time vs trigger rate
- Bus throughput
- Data misalignment vs trigger rate
- Backpressure rate
- Throttling system emulation

Status of simulation code

- Building blocks are ok
  - Except GTS trigger processor (only multiplicty trigger available) & GTS fanin-fanout
- Needs revision for uniform management of parameters
- Needs extensive testbench code for
  - Exercising all interactions between GTS and the frontends
  - Checking the event building before PSA
  - Find out most critical situations
- Needs long (> 1 sec) sampled traces (central contacts + 1 segment) to check functionality with real data

Further Development

- Up to us:
  - Use the model as a proof of concept
    - Must be agreed by everybody or adjusted
    - Leave as it is now, make measures and keep as reference
  - Extend the model to a faithful hardware description
    - Use it to validate and debug hardware
    - Use it online as a diagnostic tool
    - Every group must endorse the competent part and keep the model up-to-date with hardware

Conclusions

- System level simulation is effective because we gain insight of the problem
- The detail of description depends on simulation goal but can be changed and mixed
- SystemC works fine both for high and low level hardware description and is fast
- The channel model is the most crucial part of front-end and has impact on design choices
- GTS will extensively use this model for design investigation
- Do we extend it further ?