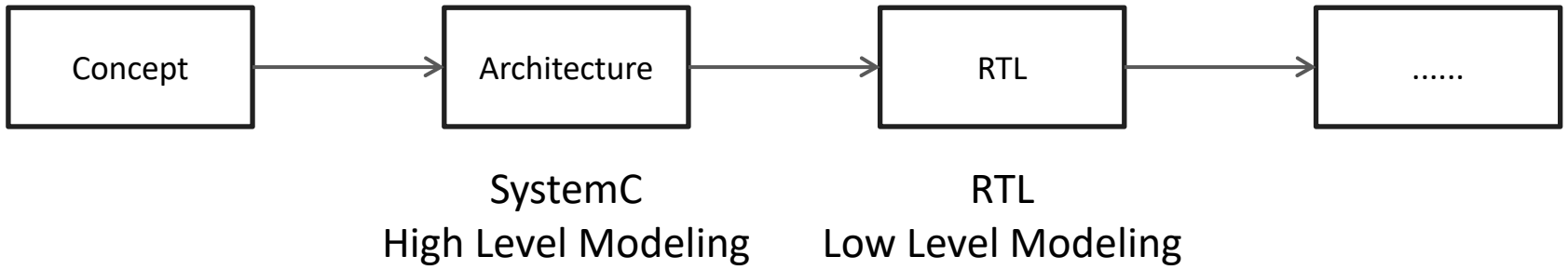
编辑母版文本样式

EVENT DRIVEN SIMULATION USING SYSTEMC

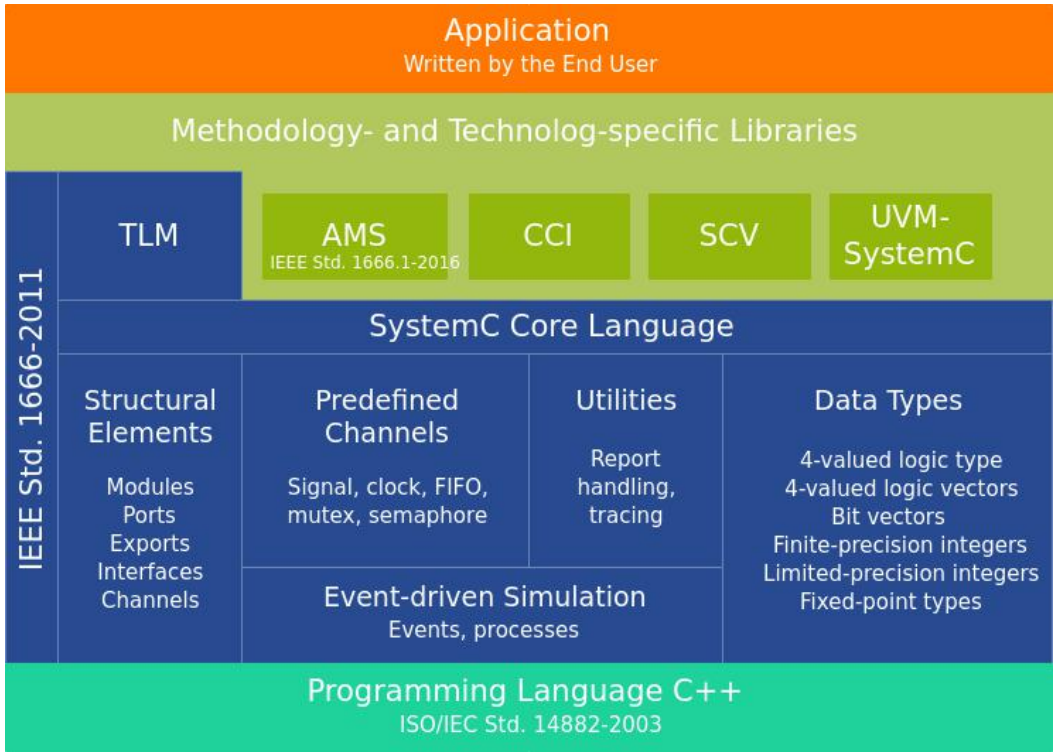
Introduction to SystemC

- SystemC addresses system design and verification
  - It focus on agile development of embedded SoC
  - It's designed to easily model complex circuit functionality
  - But difficult to directly convert to RTL



Introduction to SystemC

- SystemC contains a lot of high level features
  - Embedded in C++, flexible and extensible
  - Builtin event-driven simulation
  - Support AMS for Analog/Mix-Signal modeling
  - Support UVM for system verification



This LAB

- This lab is divided into 4 parts
  - Basic SystemC: a simple adder module
  - Event driven: a adder module with delay
  - High level design: a pipelined multiplier
  - High level design: a UART module

Install depedency

- Install systemc first
  - `apt install libsystemc-dev`

- Gtkwave is needed to show the waveform
  - Please connect server using VNC

Part 1. Basic SystemC

- Basic SC Module
  - `SC_MODULE` to define a module
  - `SC_CTOR` to define construction method
  - `SC_METHOD` to define functionality
  - `sensitive` to define the sensitive list

- When any var in the sensitive list changes, the function in `SC_METHOD` will be called immediately

```
SC_MODULE(adder) {
    sc_in<int> a, b;
    sc_out<int> sum;
    void add() {
        sum = a + b;
    }
    SC_CTOR(adder) {
        SC_METHOD(add);
        sensitive << a << b;
    }
};
```

Part 1. Basic SystemC

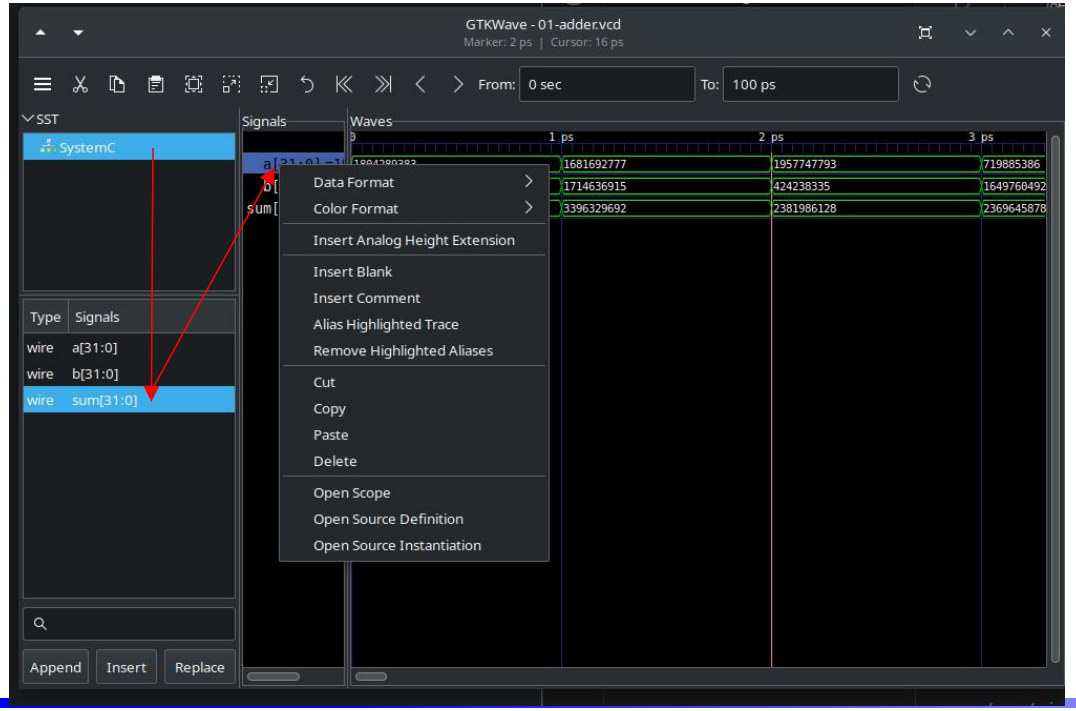
- Basic testbench
  - `sc_signal` to define signal
  - `operator ()` to connect signal
  - create trace vcd file and trace signal
  - run random test

- `make run-01-adder`

```
int sc_main(int argc, char ** argv) {
    // -------- define signal --------
    sc_signal<int> a("a");
    sc_signal<int> b("b");
    sc_signal<int> sum("sum");

    // -------- define module --------
    adder add("adder");
    add.a(a); add.b(b); add.sum(sum);

    // -------- trace signal --------
    auto f = sc_create_vcd_trace_file(argv[1]);
    sc_trace(f, a, a.name());
    sc_trace(f, b, b.name());
    sc_trace(f, sum, sum.name());

    // -------- run random test --------
    for(int i = 0; i < 100; i++) {
        a = rand(); b = rand();
        sc_start(1, SC_PS);
    }

    sc_close_vcd_trace_file(f);
    return 0;
}
```

Usage of GTKWave

- Click “SST” to show signal list
- Drag wire to “Signals” to show the signal
- Ctrl + Scroll to scale time axis
- Right click signal, you can change its format



Part. 2. Event driven

- SC thread wait event explicitly
  - `SC_THREAD` to define a thread
  - `wait()` to wait signal explicitly

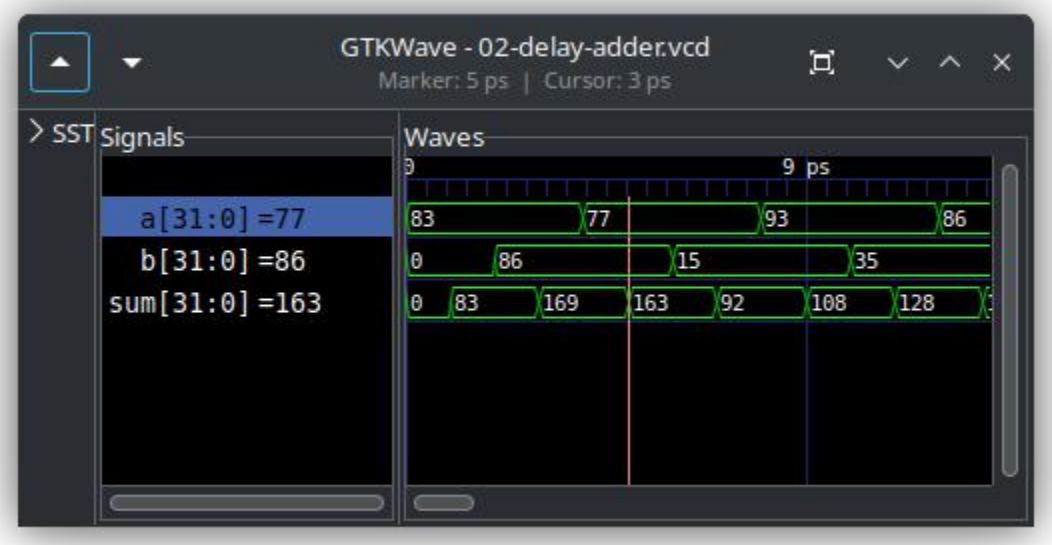
- The add function
  - it is triggered when a or b changes
  - then it delay 1ps and output a + b
  - and wait next event

```
SC_MODULE(adder) {
    sc_in<int> a, b;
    sc_out<int> sum;
    void add() {
        while(true) {
            wait(1, SC_PS);
            sum = a + b;
            wait();
        }
    }
    SC_CTOR(adder) {
        SC_THREAD(add);
        sensitive << a << b;
    }
};
```

Part. 2. Event driven

- Delay is 4ps, signal changes every 2ps
  - Use decimal format

```
// -------- run random test --------
for(int i = 0; i < 100; i++) {
    a = rand();
    sc_start(2, SC_PS);
    b = rand();
    sc_start(2, SC_PS);
}
```



Part. 3. High level design

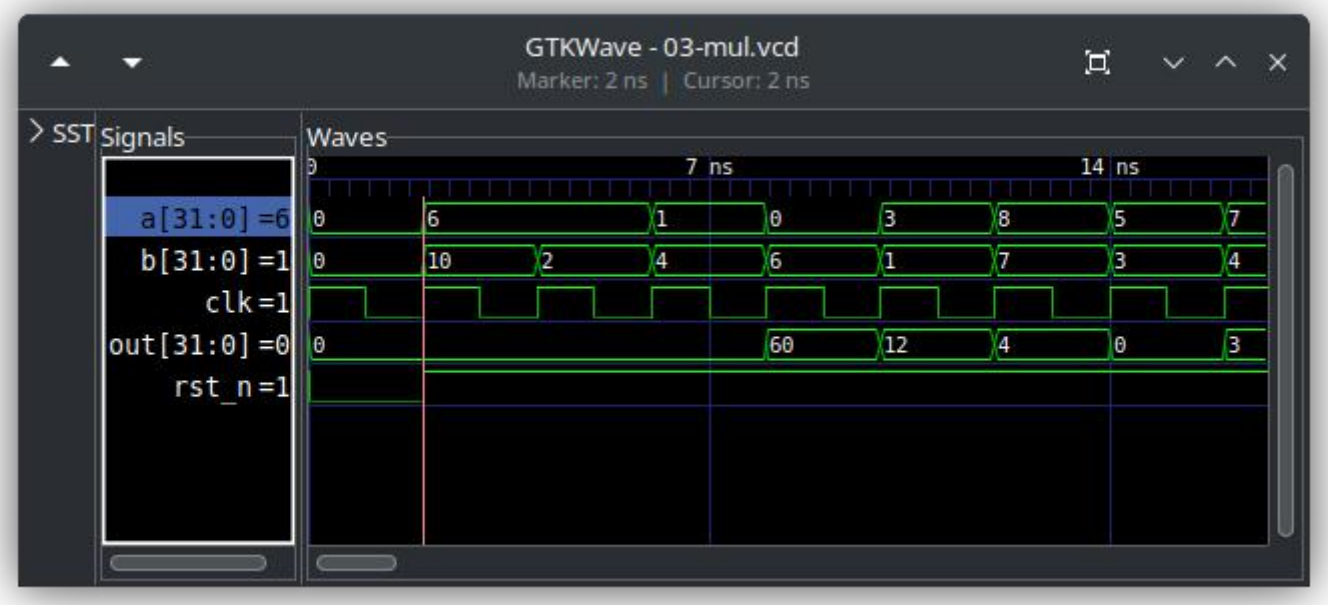
- a pipelined multiplier
  - latency = 3 cycle
  - use `std::deque` to simulate the behavior
  - sensitive to `clk.pos()`
- code before the first `wait()` is reset logic
- `SC_THREAD` active thread immediately
- `reset_signal_is` define reset signal
  - reset signal restart all thread

```
SC_MODULE(Mul) {
    sc_in<bool> clk, rst_n;
    sc_in<int> a, b;
    sc_out<int> out;
    deque<int> queue{0,0,0};
    void mul() {
        out = 0;
        wait();
        while(true) {
            out = queue.front();
            queue.pop_front();
            queue.push_back(a * b);
            wait();
        }
    }
    SC_CTOR(Mul) {
        SC_THREAD(mul);
        sensitive << clk.pos();
        reset_signal_is(rst_n, 0);
    }
};
```

reset logic

Part. 3. High level design

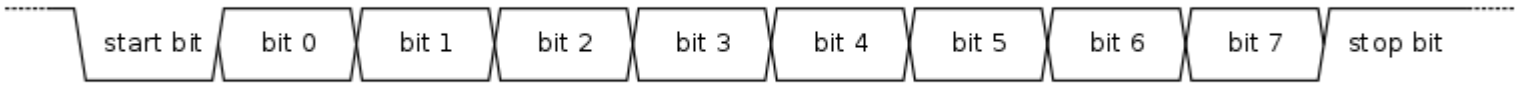
- `make run-03-mul`



Part. 4. High level design

- UART: Universal Asynchronous Receiver/Transmitter

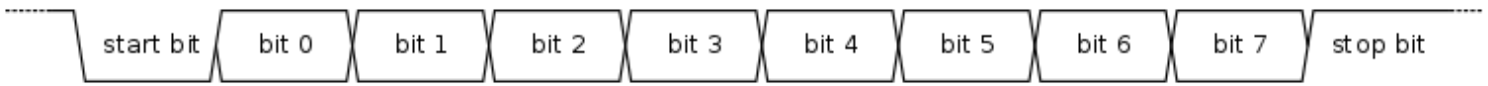
- UART protocol: output 1 bit data in each cycle



  - first output start bit ‘0’
  - then 8 bit to represent the data
  - finally 2 bits ‘11’ (usually 1.5bit, here is a simplified model)

Part. 4. High level design

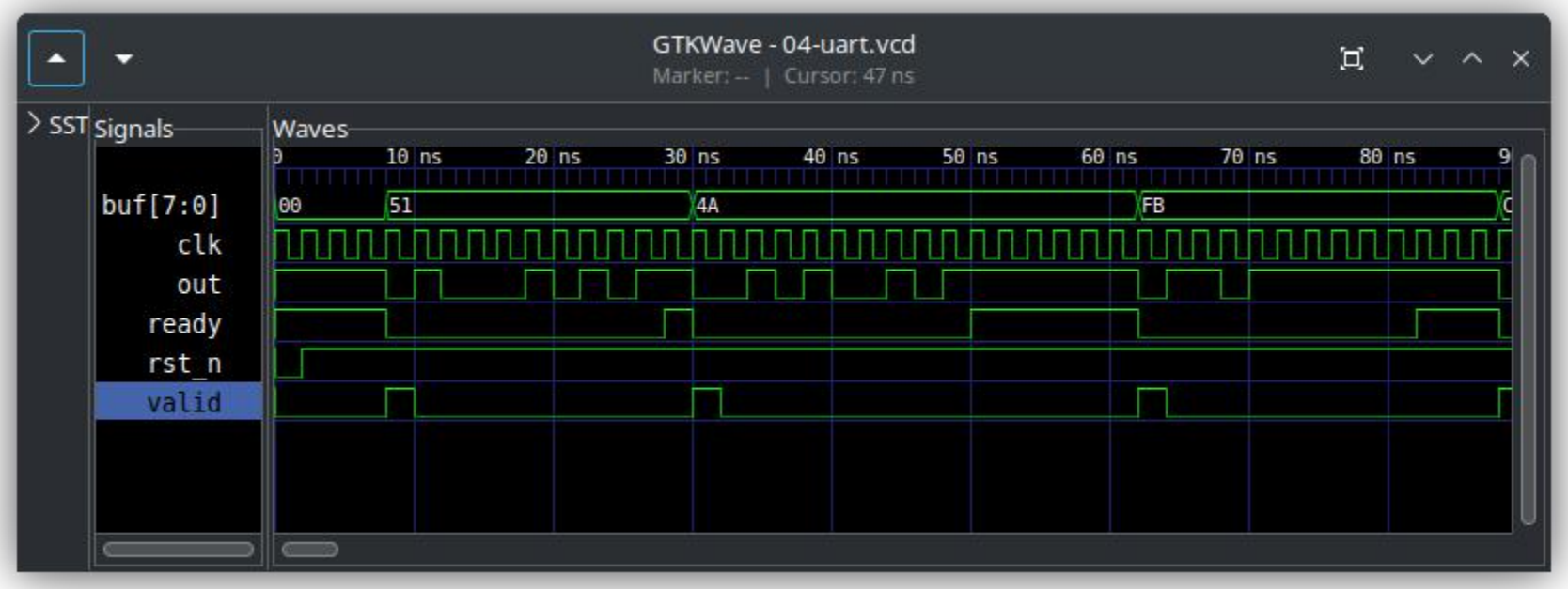
- Try to use `SC_THREAD` to implement it
  - before start, `ready` is 1
  - when `valid`, set `ready` to 0, output start bit
  - send each bit
  - set stop bit, it takes 2 cycle
  - `ready` is reset to 1 in the second stop bit




```
void uart_thread() {
    ready = 1;
    out   = 1;
    wait();
    while(true) {
        if(valid) {
            ready = 0;
            out   = 0;
            sc_int<8> data = buf;
            wait();
            for(int i = 0; i < 8; i++) {
                out = data & 1;
                data = data >> 1;
                wait();
            }
            out = 1;
            wait();
            ready = 1;
            wait();
        }
        else wait();
    }
}
```

Part. 4. High level design

- `make run-04-uart`



Part. 4. High level design

- Imaging how to implement the module by Chisel
  - Check how many states are there
    - `idle -> start -> send -> stop`
  - Check the state transition
  - Implement each state
  - Debug

- It’s difficult to get it work in RTL
  - In early development, high level model language is very important
  - SystemVerilog is similar to SystemC
    - but there is no opensource event-driven simulator