Performance Estimation

Uses

1. System-level
   - based on the system specification
   - guides hw/sw partitioning
   - imprecise, quick

2. Validation
   - based on the implementation
   - confirms non-functional requirements (latency, throughput, power consumption, etc)
   - precise, slow

Methods

1. Measurement
   - for final estimation only

2. Simulation
   - from executable models

challenge: coverage (inputs?)

3. Probabilistic Analysis
   - based on distributions
   e.g. queueing theory

4. Deterministic Analysis
   - determine worst/base-case latencies and bounds
   e.g. task graph scheduling ⇒ makespan
   e.g. real-time schedulability tests

involves abstractions/simplifications

Discrete Event Simulation
- concurrent processes are modelled as hierarchies of modules
- express module behaviour with imperative (e.g. C) or declarative (e.g. logic/algebraic) languages
- state is the collection of variables in each module
- module ports exchange signals
- events are exchanges of signals

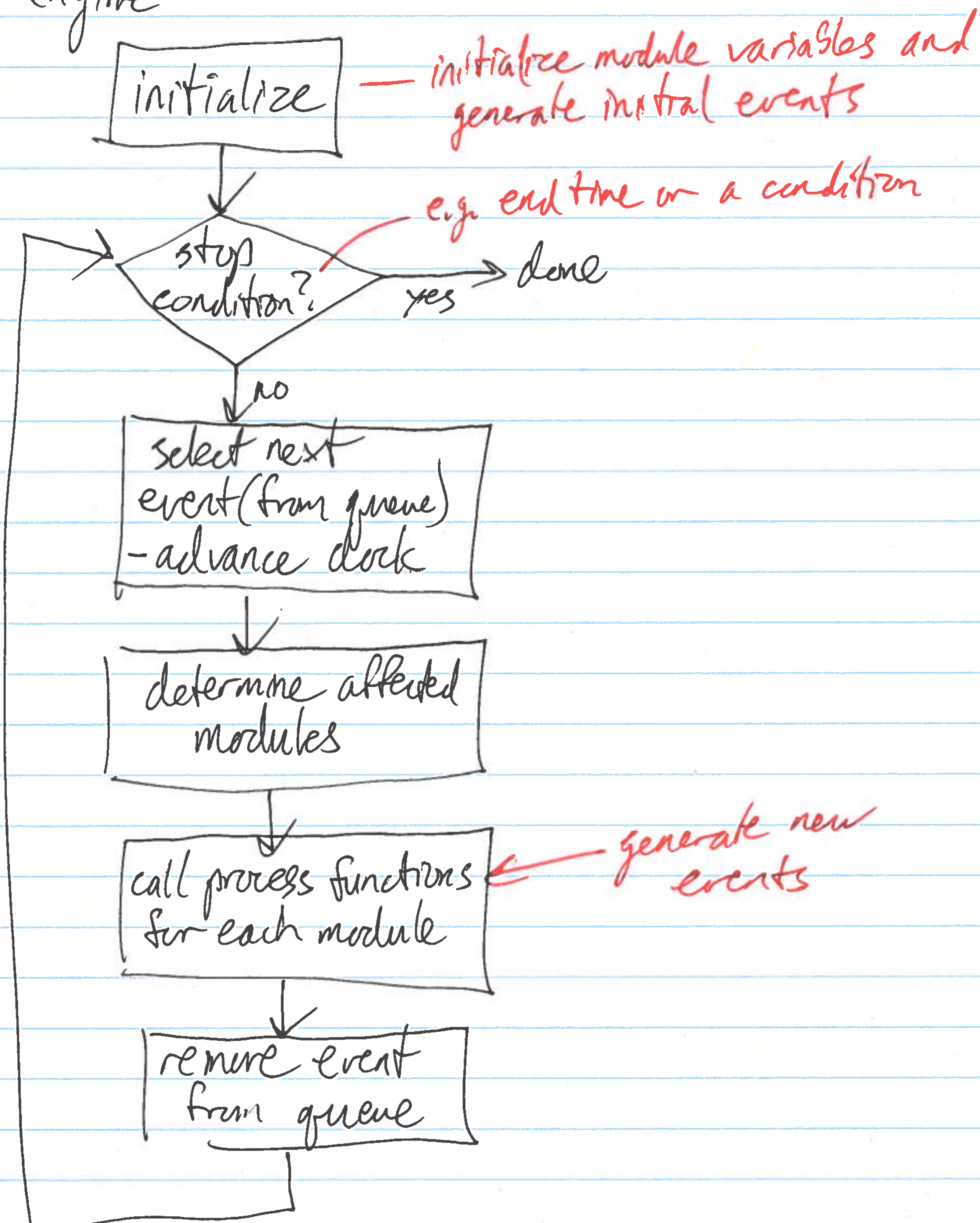
Simulator parts:

1. clock tracks current time

2. event queue - ordered by event times
   - events are executed one at a time

3. modules - process functions are invoked for events that they are sensitive too
   - process functions manipulate variables and generate events

- simulation engine



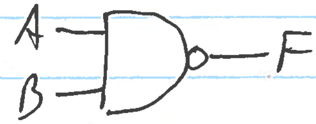
- new events may share the same time
  - they executed in successive delta cycles (zero duration)

System C

- system-level modeling language
- can do functional (no time) to cycle-accurate simulations
- C++ library of templates and classes
- provides:
  - h/w-oriented data types
  - communication mechanisms
  - event-driven simulation kernel

Modules: basic blocks
- I/O done through ports
- have processes that are scheduled by the kernel

Processes:
SC_THREAD: called once, executes forever
SC_METHOD: called whenever inputs change (based on a sensitivity list)




```
#include <systemc>
using namespace sc_core;

SC_MODULE(nand) {
```

declare module

```
    sc_in<bool> A, B;
    sc_out<bool> F;

    void evaluate() {
        F.write(!(A.read() && B.read()));
    }

    SC_CTOR(nand) {
```

constructor

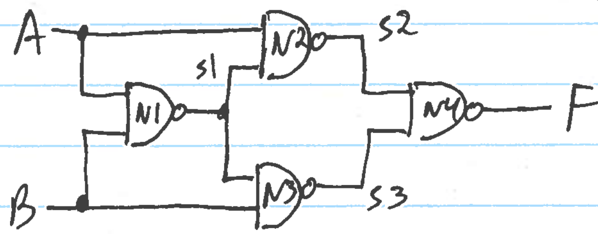
```
        SC_METHOD(evaluate);
```

registers function with the simulation kernel

```
        sensitive << A << B;
```

sensitivity list - for the last registered method

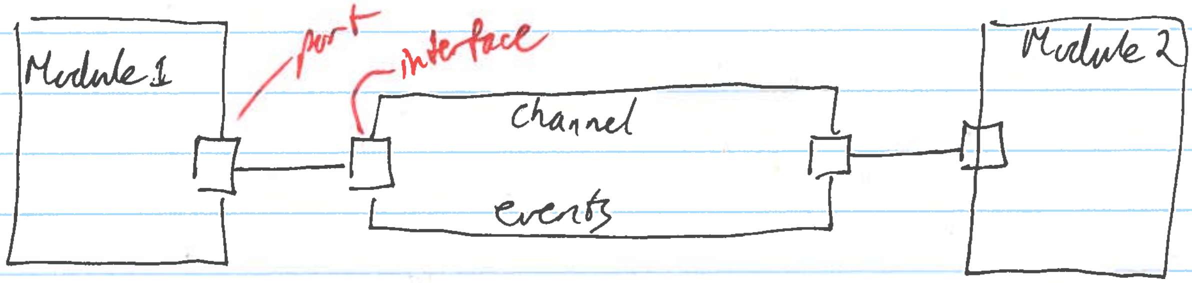
```
    }
};
```




```
#include "nand.h"

SC_MODULE(xor) {
    sc_in<bool> A, B;
    sc_out<bool> F;
    nand n1, n2, n3, n4;

    SC_CTOR: n1("N1"), n2("N2"), n3("N3"), n4("N4") {

        n1.A(A);
        n1.B(B);       <- defining connections    OR
        n1.F(s1);

        n2 << A << s1 << s2;
              A    B     F

        n3(s1);   A
        n3(B);    B
        n3(s3);   F

        n4 << s2 << s3 << F;
    }
};
```

Channels

- communication between modules
- ~~event~~ sync. primitives are used for synchronize
- interfaces define access methods to a channel
- bind module ports to interfaces



Synchronization Primitives

wait - blocks SC_THREAD until desired event occurs

wait(sc_event)
wait(time_out, sc_event)
wait(time)

notify - raise event

event.notify()
? event.notify(time) - schedule notification in future
event.notify(SC_ZERO_TIME) - schedules notification at end of current time delta

- example: Learn > Lecture > Handouts > systemC Handout

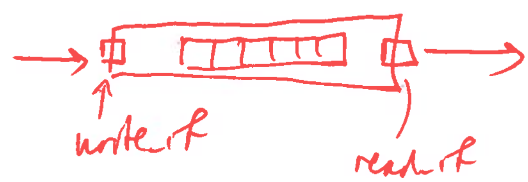
FIFO - Interface

```
class write_if : public sc_interface
{
  public:
    virtual void write(char) = 0;
    virtual void reset() = 0;
};

class read_if : public sc_interface
{
  public:
    virtual void read(char &) = 0;
    virtual int num_available() = 0;
};
```

pure virtual functions
(abstract class)

Stuart Swan, Cadence, 2002

Channel FIFO - Implementation




```
class fifo : public sc_channel, public write_if, public read_if
{
  public:
    fifo() : num_elements(0), first(0) {}

    void write(char c) {
      if (num_elements == max_elements)
        wait(read_event);

      data[ (first + num_elements) % max_elements ] = c;
      ++ num_elements;
      write_event.notify();
    }

    void read(char& c) {
      if (num_elements == 0)
        wait(write_event);

      c = data[first];
      -- num_elements;
      first = (first + 1) % max_elements;
      read_event.notify();
    }

  void reset() { num_elements = first = 0; }

  int num_available() { return num_elements; }

 private:
  enum e { max_elements = 10 };  // just a constant
  char data[max_elements];
  int num_elements, first;
  sc_event write_event, read_event;
};
```

Producer / Consumer

```
class producer : public sc_module
{
 public:
  sc_port<write_if> out;  // the producer's output port

  SC_CTOR(producer)      // the module constructor
  {
   SC_THREAD(main);  // start the producer process
  }

  void main()            // the producer process
  {
     char c;
     while (true) {
       ...
       out->write(c);       // write c into the fifo
       if (...)
         out->reset();      // reset the fifo
     }
  }
};
```

```
class consumer : public sc_module
{
 public:
  sc_port<read_if> in;       // the consumer's input port

  SC_CTOR(consumer)      // the module constructor
  {
   SC_THREAD(main);      // start the consumer process
  }

  void main()            // the consumer process
  {
     char c;
     while (true) {
       in->read(c);            // read c from the fifo
       if (in->num_available() > 5)
         ...;                  // perhaps speed up processing
     }
  }
};
```

Top

```
class top : sc_module
{
 public:
   fifo fifo_inst;                 // a fifo instance
   producer *producer_inst;   // a producer instance
   consumer *consumer_inst; // a consumer instance

   SC_CTOR(top)              // the module constructor
   {
     producer_inst = new producer("Producer1");
     // bind the fifo to the producer's output port
     producer_inst->out(fifo_inst);

     consumer_inst = new consumer("Consumer1");
     // bind the fifo to the consumer's input port
     consumer_inst->in(fifo_inst);
   }
};
```

```
int sc_main(int argc,
  char *argv[]) {
  top t1("top");
  sc_start();
}
```

run simulation