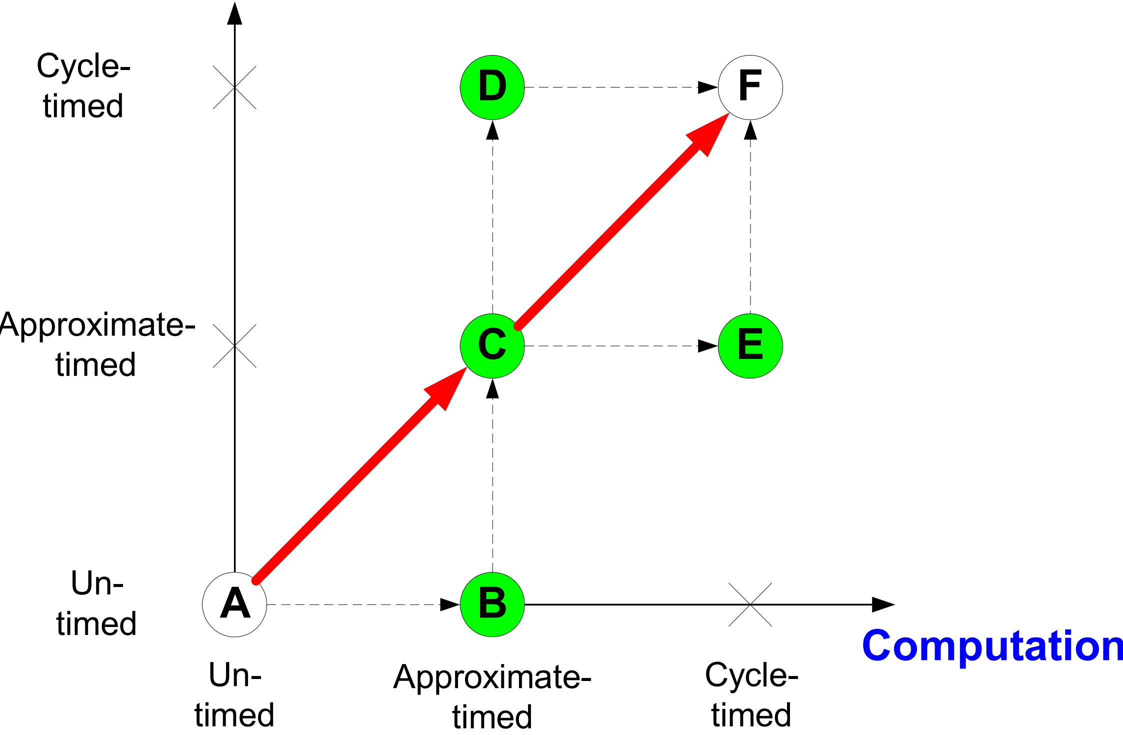
System-on-Chip Design
Transaction-Level Modeling with SystemC

Dr. Hao Zheng
Comp. Sci & Eng.
U of South Florida

Motivation

- Why use transaction-level modeling and ESL languages?
  - Manage growing system complexity
  - Enable HW/SW co-design
  - Speed-up simulation
  - Support system-level design and verification
- Increase designers' productivity
- Reduce development costs and risk
- Accelerate time-to-market & time-to-money

Levels of Abstraction



A. **Specification Model**
   "'Untimed' Functional Models"

B. **Component-Assembly Model**
   "Architecture Model"
   "'Timed' Functional Model"

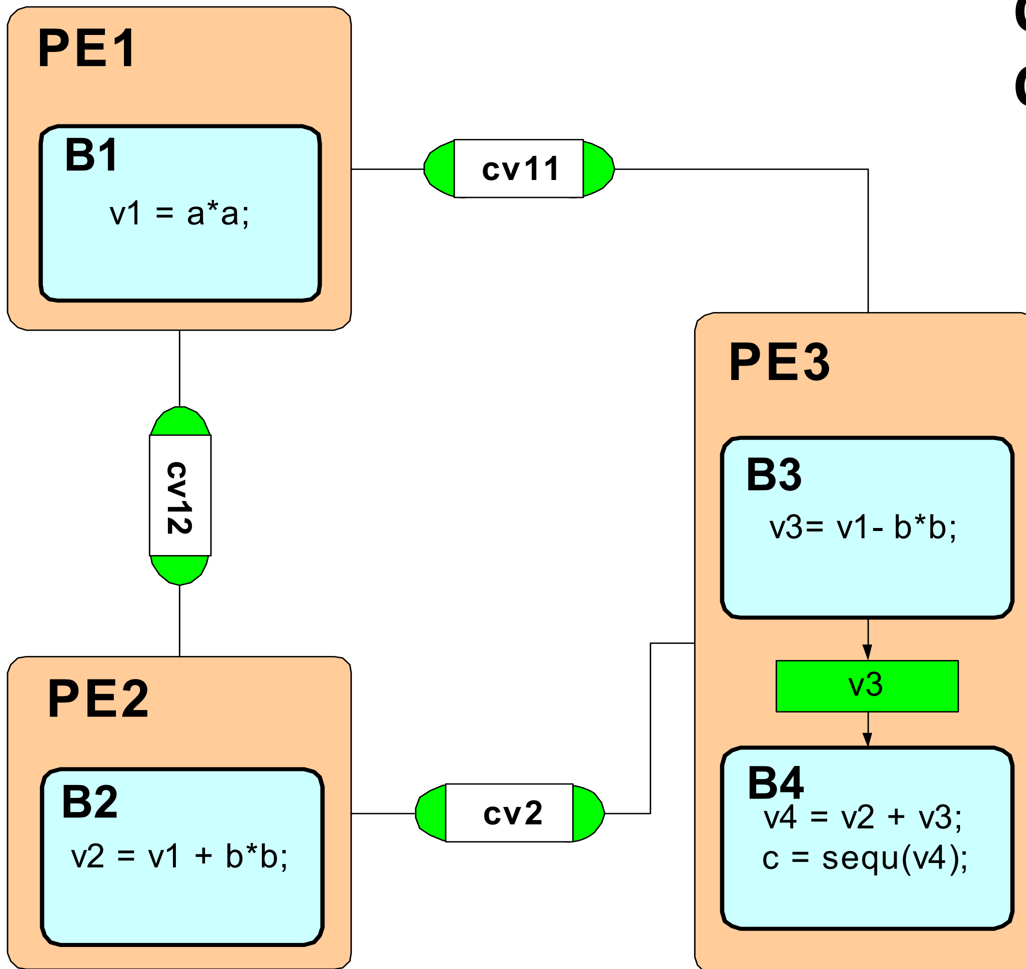
C. **Bus-Arbitration Model**
   "Transaction Model"

D. **Bus-Functional Model**
   "Communication Model"
   "Behavior-Level Model"

E. **Cycle-Accurate Computation Model**

F. **Implementation Model**
   "Register-Transfer Level (RTL) Model"

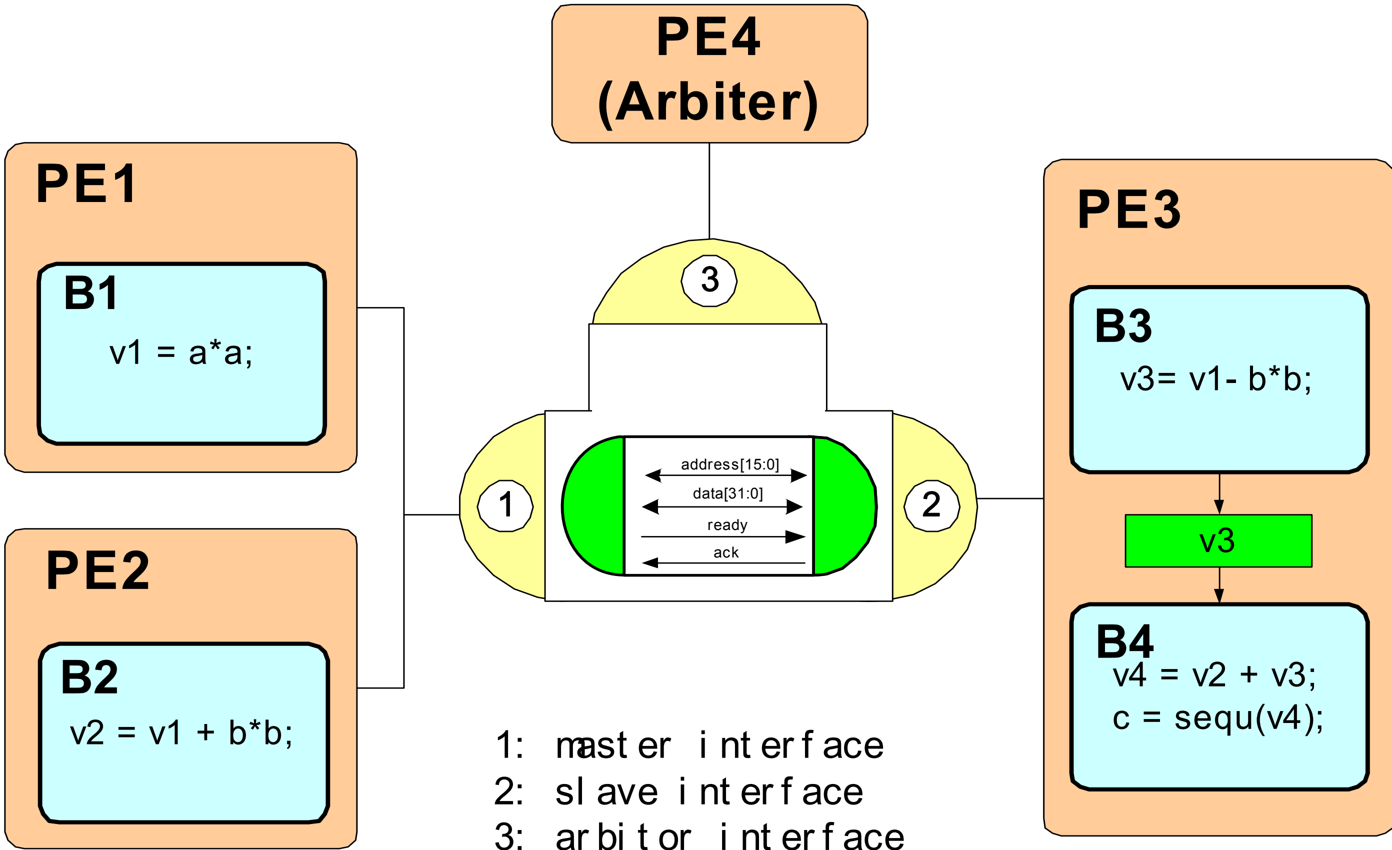
Functional Model: Untimed or Timed



Computation – behavior
Communication – abstract channels

A network of communicating sequential processes connected by abstract channels.

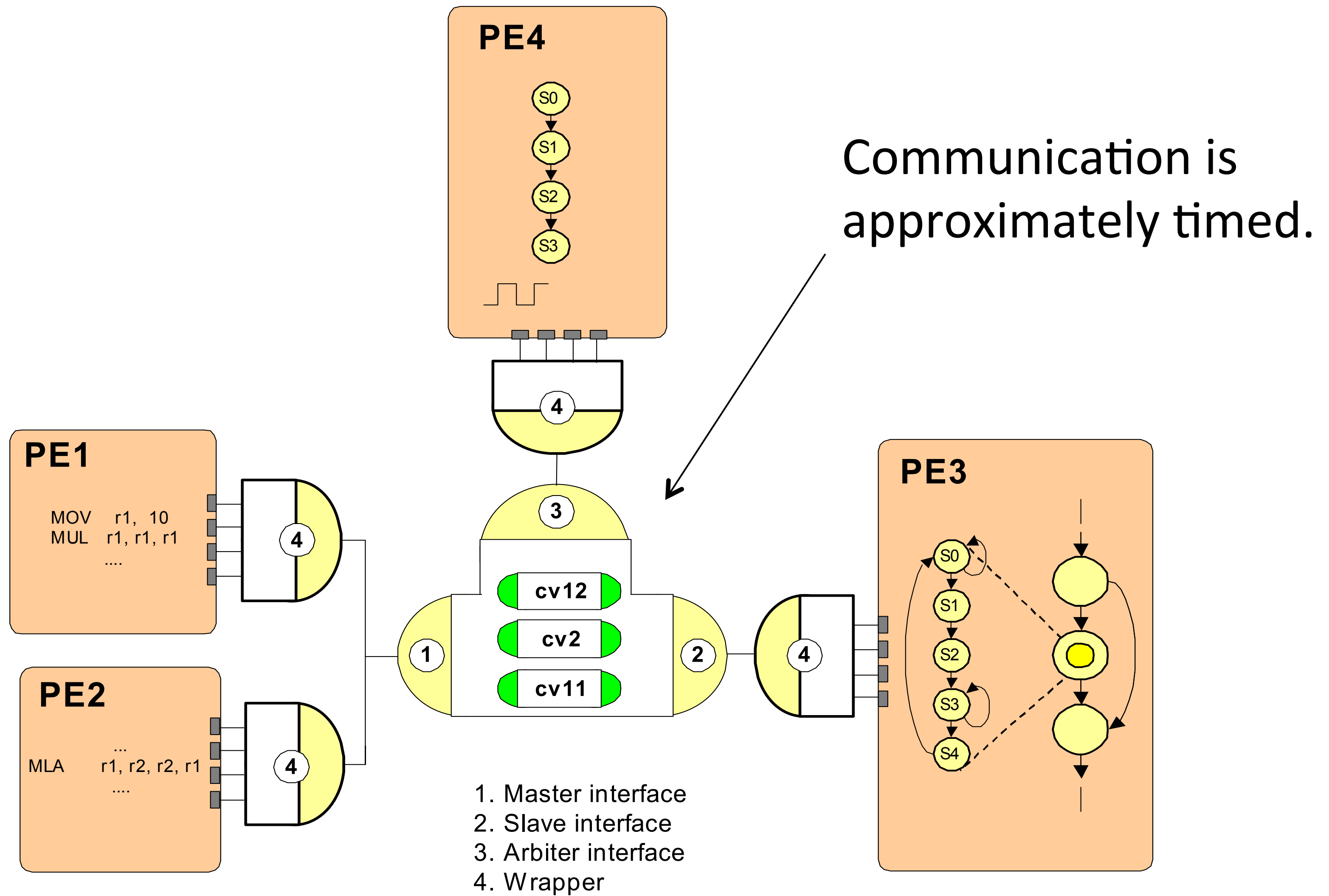
Bus Functional/Arbitration Model



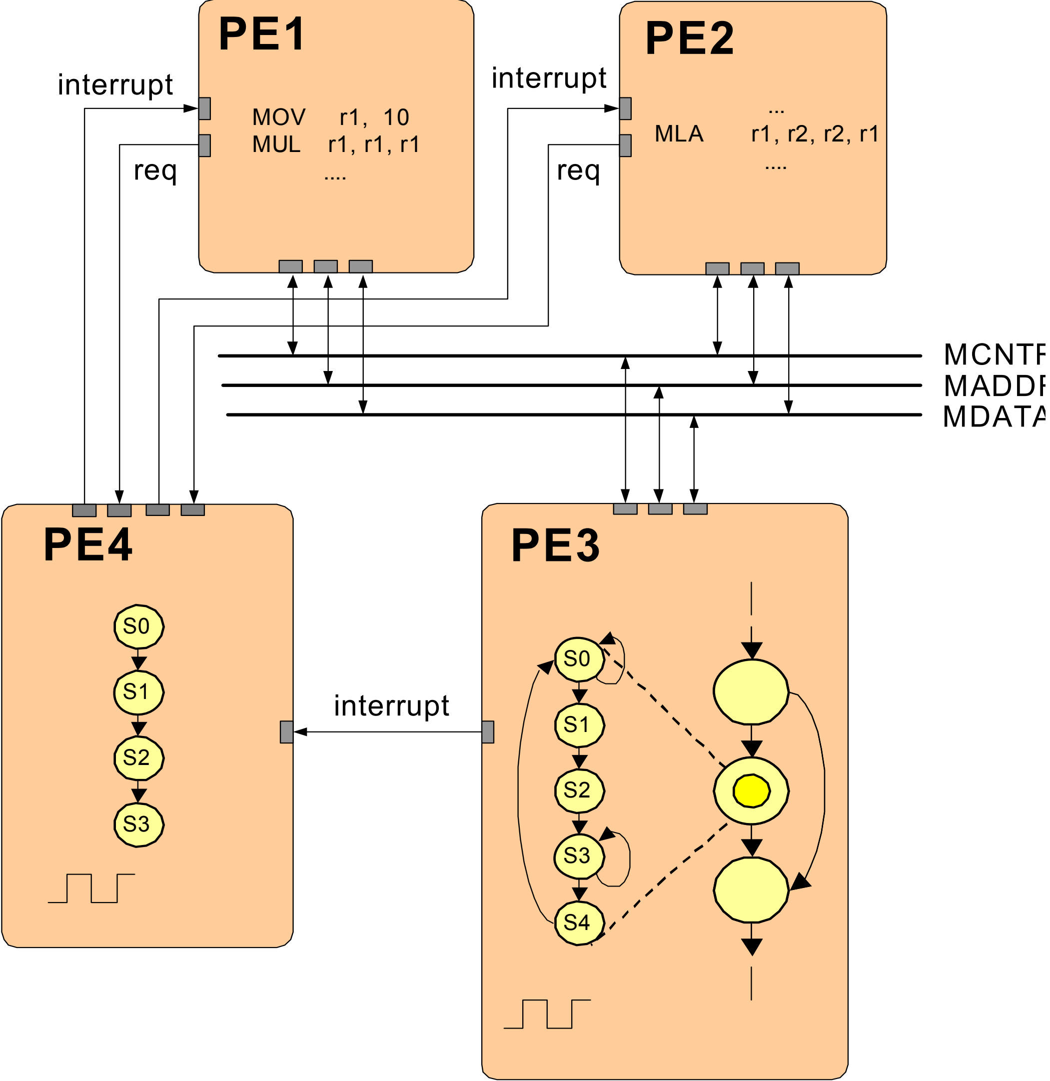
Computation – behavioral, approximately timed

Communication – protocol bus channels

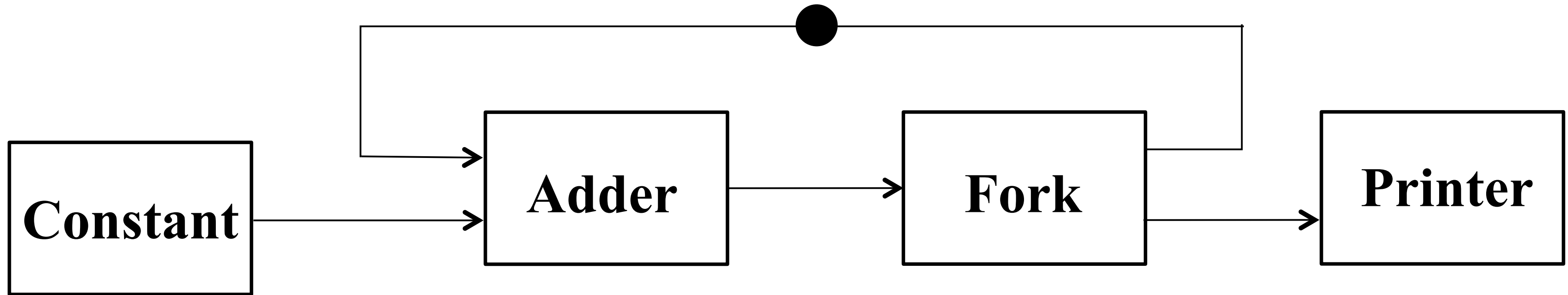
Cycle Accurate Computation Model



Implementation Model



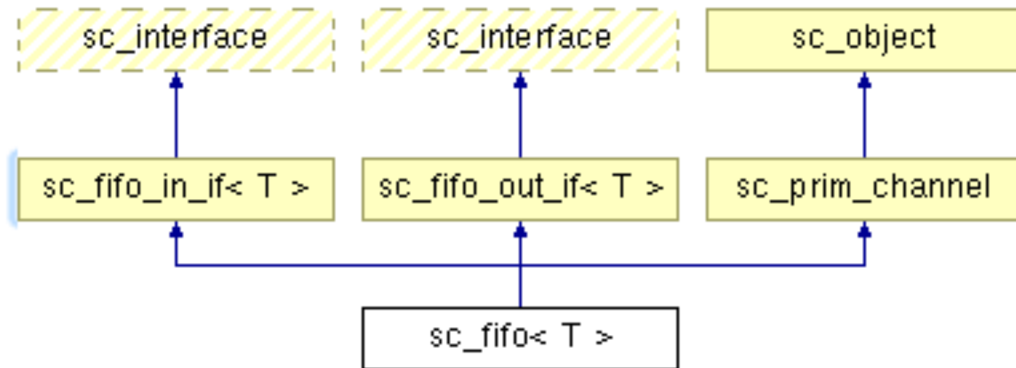
Dataflow Modeling (SystemC-1, Chapter 5)



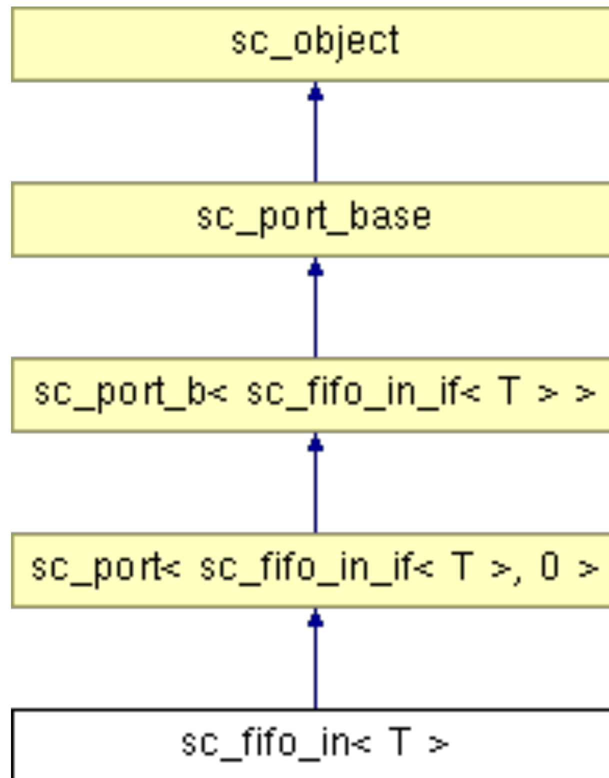
- Actors – processes
- communications – FIFO channels
  - Channels are bounded.
- Comm. channel operations – blocking read & write

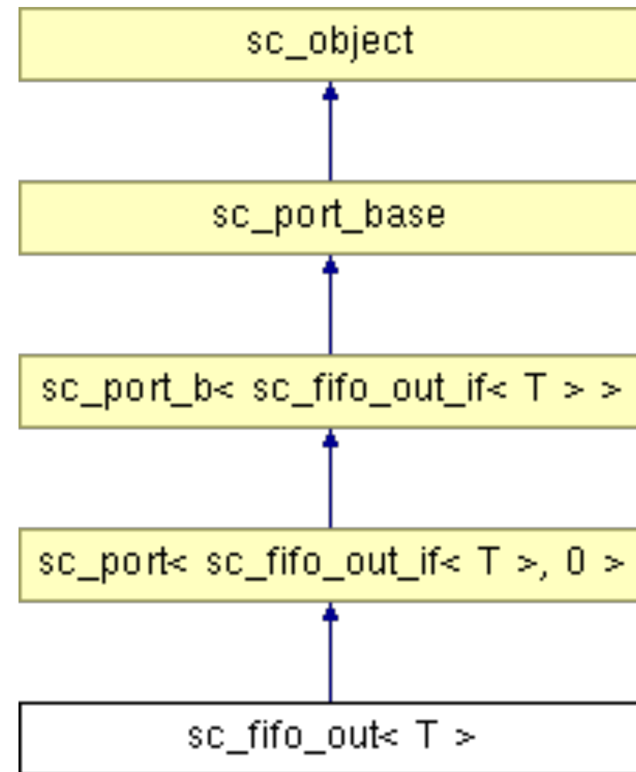
Basic Channel: sc_fifo<T>

```
void read(T&);
T    read();
bool nb_read(T&);
int  num_available();
void write(const T&);
bool nb_write(const T&);
int  num_free();
sc_fifo(int size=16);
sc_fifo(char* name, int size=16);
```



Ports Compatible with sc_fifo<T>





sc_fifo_in<T>: support only read operations.

sc_fifo_out<T>: support only write operations.

Dataflow Modeling: Adder

```
template <class T>
SC_MODULE(DF_Adder) {
   sc_fifo_in<T> din1, din2;
   sc_fifo_out<T> dout;

   void process() {
      while (1)
         dout.write(din1.read() + din2.read());
   }

   SC_CTOR(DF_Adder) { SC_THREAD(process); }
};
```

Dataflow Modeling: Constant Generator

```
template <class T>
SC_MODULE(DF_Const) {
   sc_fifo_out<T> dout;

   void process() {
      while (1) dout.write(constant_); }

   SC_HAS_PROCESS(DF_Const);
   DF_Const(sc_module_name N, const T& C) :
     sc_module(N), constant_(C)
     { SC_THREAD(process); }

   T constant_;
};
```

Dataflow Modeling: Fork

```
template <class T>
SC_MODULE(DF_Fork) {
   sc_fifo_in<T> din;
   sc_fifo_out<T> dout1, dout2;

   void process() {
      while (1) {
         T value = din.read();
         dout1.write(value);
         dout2.write(value);
      }}

   SC_CTOR(DF_Fork) { SC_THREAD(process); }
};
```

Dataflow Modeling: Printer

```
template <class T>
SC_MODULE(DF_Printer) {
   sc_fifo_in<T> din;

   void process() {
      for (int i=0; i < n_iter; i++) {
         T value = din.read();
         cout << name() << " " <<value<<endl;
      }
      done_ = true; return; // terminate
   }

   SC_HAS_PROCESS(DF_Printer);
   DF_Printer(...) ... { SC_THREAD(process); }
};
```

Dataflow Modeling: Top Module

```
sc_main(int argc, char* argv[]) {
   DF_Const<int> constant("constant", 1);
   DF_Adder<int> adder("adder");
   DF_Fork<int>  fork("fork");
   DF_Printer<int> printer("printer", 10);

   sc_fifo<int> const_out("const_out", 5);
   sc_fifo<int> adder_out("adder_out", 1);
   sc_fifo<int> feedback("feedback", 1);
   sc_fifo<int> to_printer("2printer", 1);

   feedback.write(42); // channel init.
   ...
}
```

Dataflow Modeling: Top Module

Port binding

```
sc_main(int argc, char* argv[]) {
    ...
    constant.output(const_out);
    adder.din1(feedback);
    adder.din2(const_out);
    fork.din(adder_out);
    fork.dout1(feedback);
    fork.dout2(to_printer);
    printer.din(to_printer);

    sc_start(); //No sim. time limit

    return 0;
}
```

Timed Models

```
template <class T>
SC_MODULE(DF_Const) {
    sc_fifo_out<T> dout;

    void process() {
        while (1) {
            wait(200, SC_NS);
            dout.write(constant_);
        }
        ...
};
```

Computational delay → `wait(200, SC_NS);`

Timed Models

```
template <class T>
SC_MODULE(DF_Adder) {
    ...

    void process() {
        while (1)
            T data = din1.read() + din2.read();
            wait(200, SC_NS);
            dout.write(data);
    }
    ...
};
```

Stopping Dataflow Simulation

- Execute the model for a fixed number of iterations – then stop simulation.
  - Example: DF_printer
- Or let constant generator produce stop after producing a number of tokens.
- If termination depends on that multiple modules finish, then they output flags to a terminator module to decide whether to stop simulation.
- For timed models, use `sc_start()` with a time limit.

Reading

- ***SystemC-1*** book, Chapter 5.

Concepts of TLM

- Transaction: a single object that includes signals carrying data and protocols to allow data to be transferred.
- TLM separates computation from communication.
  - Separate refinements of comp & comm.
- In a TLM, communication is done via function calls.
  - Ex: burst_read(char* buf, int addr, int len);
  - Focus on what data to transfer from and to locations
  - Not about how data transfer is implemented.

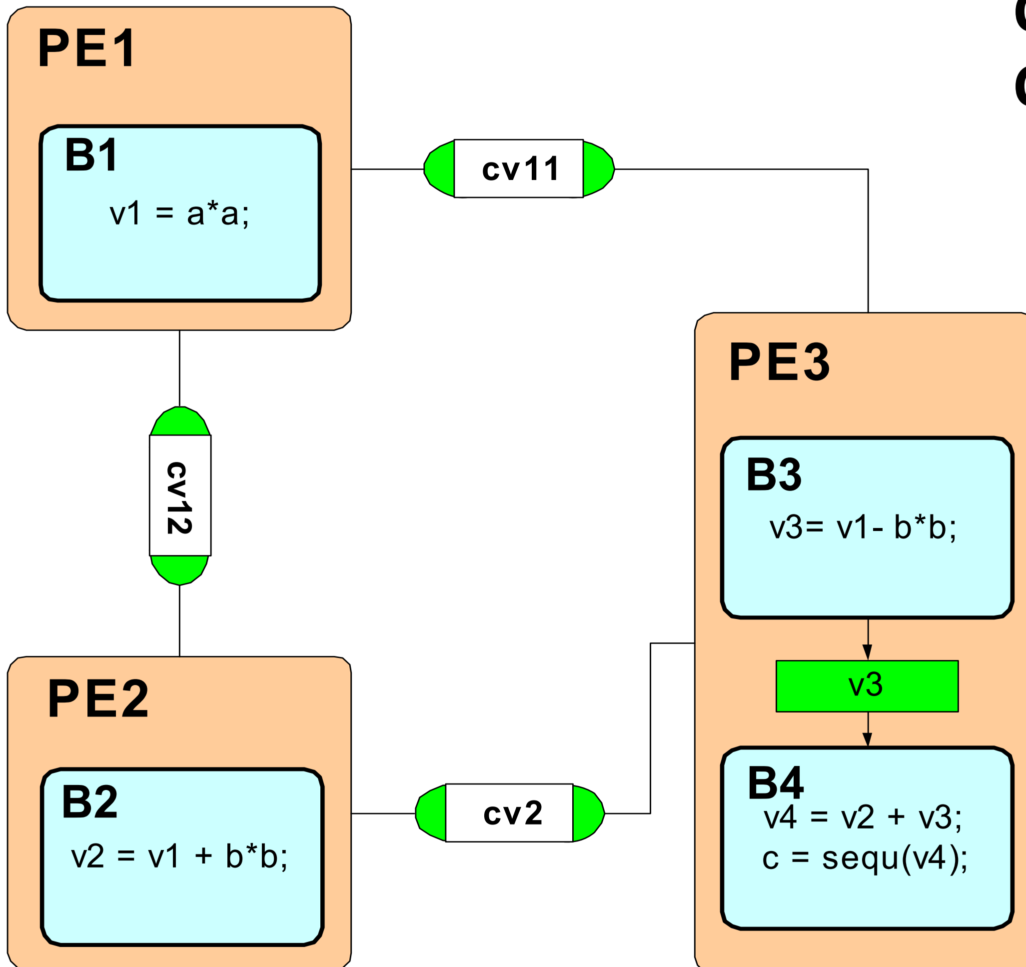
Why TLM

- Higher simulation speed compared to RTL models
- Higher modeling accuracy compared to functional models for evaluating design properties.
- A TLM can integrate both SW and HW models.
  - Provide a platform for early SW development.
  - Early system exploration and verification

Support TLM: Interface and Channel

- An interface declares access methods to channels
- A channel implements access methods declared in the interfaces inherited by the channel.
- A design model can choose from different channels as long as they all implement the same interfaces.
- Separation of interfaces and channels facilitates design space exploration and refinement.

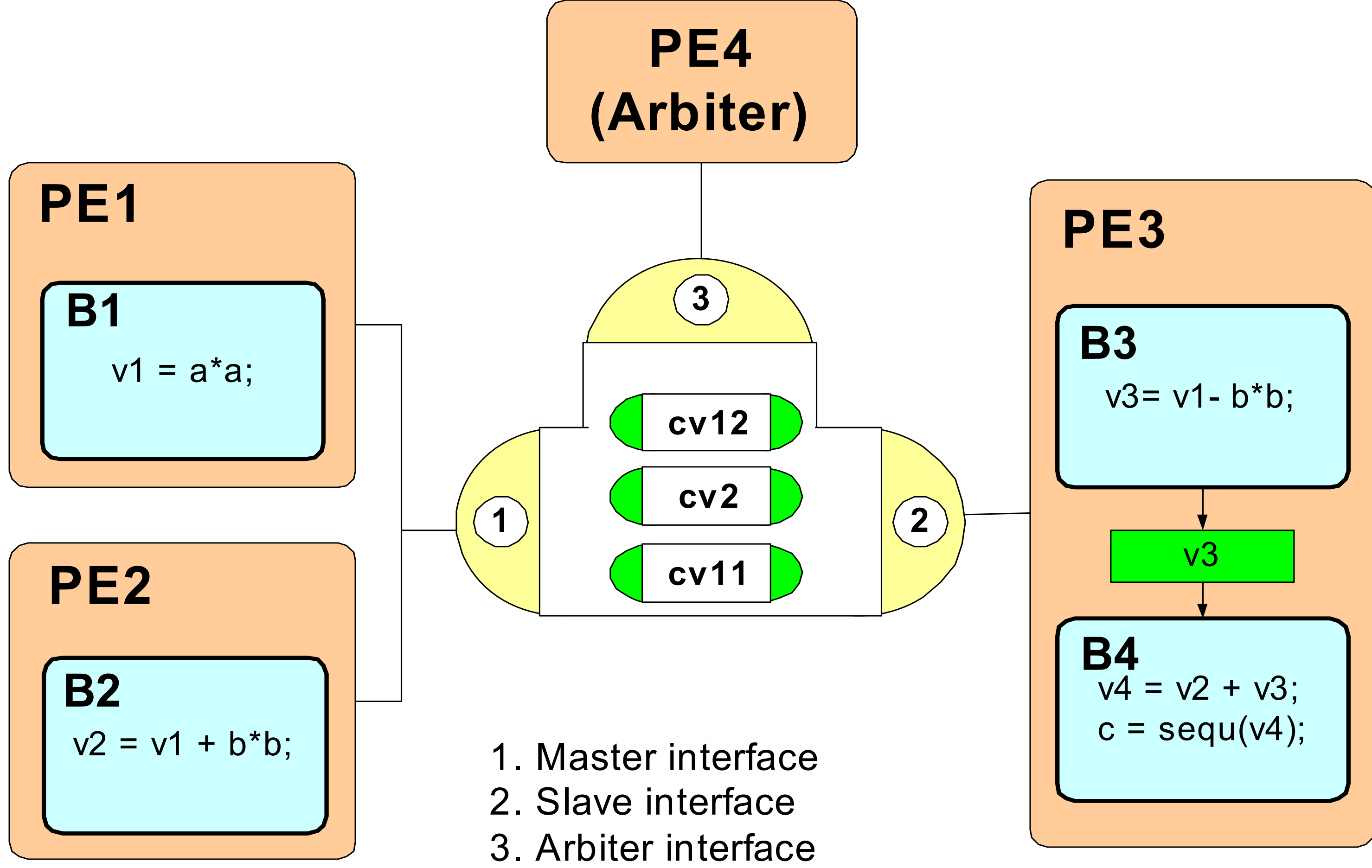
Functional Model: Untimed or Timed



Computation – behavior
Communication – abstract channels

A network of communicating sequential processes connected by abstract channels.

Bus Arbitration Model



Abstract channels are implemented in an abstract communication structure.

A Simple Bus Design

```
class bus_if: virtual public sc_interface
{
public:
  virtual void burst_read (   char* data,
                              unsigned addr,
                              unsigned len) = 0;

  virtual void burst_write (  char* data,
                              unsigned addr,
                              unsigned len) = 0;
}
```

How many cycles would be needed to to complete a burst transaction in a RTL model?

A Simple Bus Design

```
class simple_bus :
           public bus_if, public sc_channel
{
public:
  simple_bus(sc_module_name nm, unsigned mem_size,
                  sc_time cycle_time) :
     sc_channel(nm), _cycle_time(cycle_time) {...}
  ~simple_bus() {...}
  virtual void burst_read(...) {}
  virtual void burst_write(...) {}

protected:
  char* _mem;
  sc_time _cycle_time;
  sc_mutex _bus_mutex; // ensure exclusion access to _mem
}
```

sc_mutex

```
sc_mutex name;

name.lock(); // lock the mutex

int name.trylock(); // non-blocking lock
                    // return 0 for success
                    // return -1 otherwise

name.unlock(); // free a locked mutex.
```

A Simple Bus Design

```
virtual void burst_read(   char* data,
                           unsigned addr
                           unsigned len)
{
  _bus_mutex.lock();

  // Block the caller for the data xfer
  // Modeling mem read delay
  wait (len * _cycle_time);

  // xfer data
  memcpy(data, _mem + addr, len);

  _bus_mutex.unlock();
}
```

A Simple Bus Design

```
virtual void burst_write(  char* data,
                           unsigned addr
                           unsigned len)
{
  _bus_mutex.lock();

  // Block the caller for the data xfer
  // Modeling mem write delay
  wait (len * _cycle_time);

  // xfer data
  memcpy(_mem + addr, data, len);

  _bus_mutex.unlock();
}
```

Arbitration is not supported. Any idea how to do it?

Reading

- ***SystemC-1*** book, section 8.1 – 8.2.

A design Problem: Matrix Multiplication

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,k} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,k} \end{bmatrix}$$

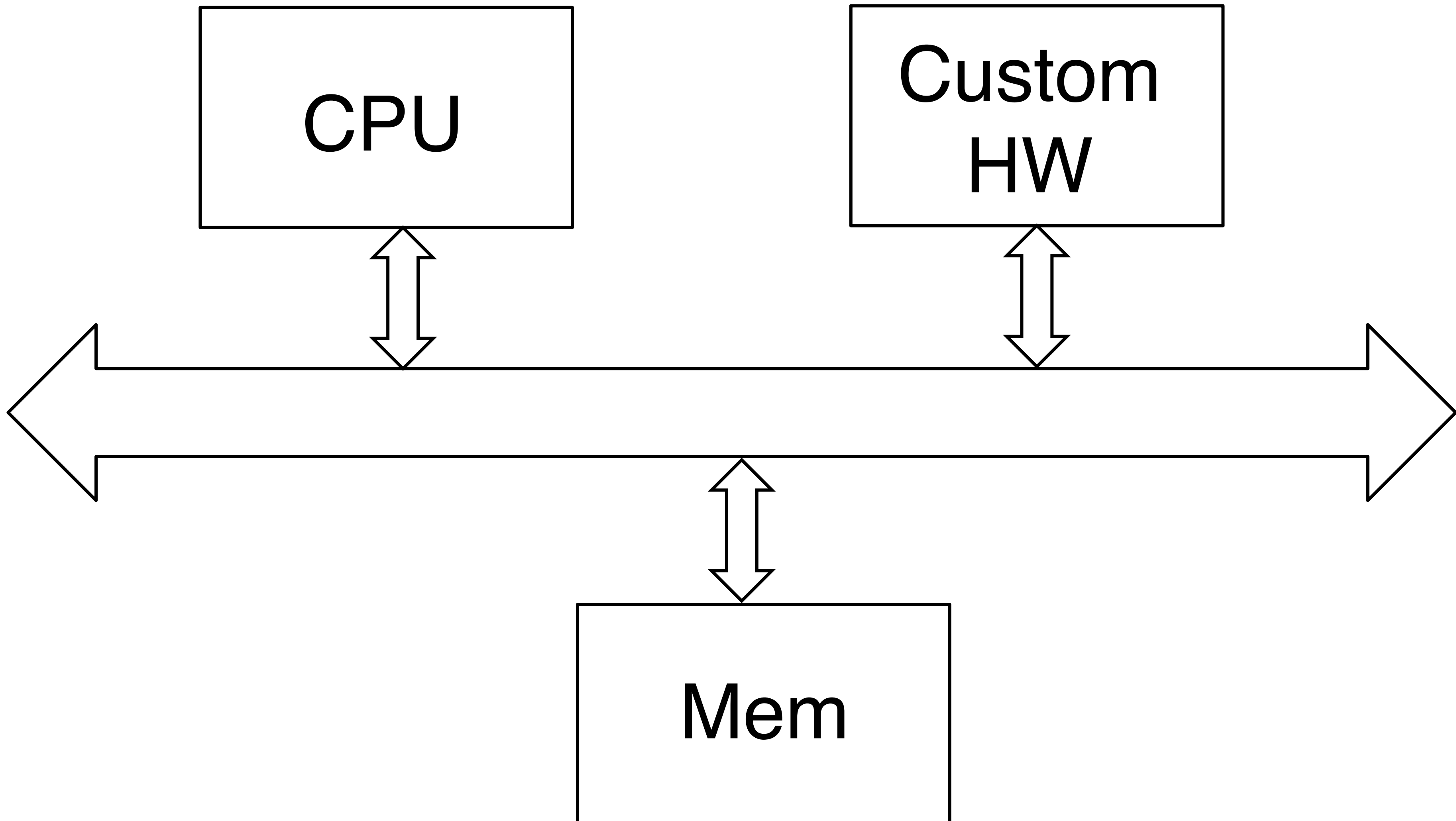
$$= \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,k} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,k} \end{bmatrix}$$

$$\text{where } c_{i,j} = \sum_{x=1}^{n} a_{i,x} b_{x,j}$$

A design Problem: Matrix Multiplication

- *First* step, algorithmic modeling – a C program.
- *Second* step: transaction level modeling
- *Third* step: communication refinement
- *Fourth* step: custom HW implementation
- *Fifth* step: replace CPU with a cycle-accurate instruction set simulator
  - Will skip this step

A design Problem: Matrix Multiplication



A design Problem: Matrix Multiplication

- Words – unsigned 32-bit integers.
- Performance constraints
  - Memory access overhead: 100 cycles/access
  - Memory read/write delay: 10 cycles/word
  - CPU Add: 15 cycles
  - CPU shift: 1 cycle
  - CPU multiply: 500 cycles
  - Bus xfer: 1 cycle/word
  - Custom HW: depend on the implementation

A design Problem: Matrix Multiplication

- Custom HW implementation
  - multiplication – various optimizations for performance
  - multi/accumulation
  - multiple copies of the above for parallelism
- Need to find its average performance for estimation.
  - need to run it for a large set of random inputs.