Using the Liberty Simulation Environment

Manish Vachharajani

http://www.liberty-research.org
Modeling in Liberty

- Functionality through structure
- Connect together module instances
- Modules
  - Black boxes encapsulating computation
  - Communicate via ports
A Simple Example

```
include "preset_data.lss";
include "sink.lss";

instance generator : preset_data;
instance garbage : sink;

generator.out -> garbage.in;
```

Output

```
$ ./Xsim none
Simulator has no more events at time 4/0
Finish Time: 4/0
```
Instrumenting the Simulator

- Separate instrumentation from specification
- Instrument simulation by collecting events
- Events have attached per-event data

Event generation
- Each module emits events during execution
- Framework generates events any time data is sent or received

We define collectors to catch and process these events
- Compute summary statistics
- Display messages for debugging/logging
A Simple Example (continued)

```c
collector in.resolved on "garbage" { 
    record = <<<<
        if (LSE_signal_data_known(status) &&
            !LSE_signal_data_known(prevstatus)) {
            printf(LSE_time_print_args(LSE_time_now));
            if(LSE_signal_data_present(status)) {
                printf(": %d\n", *datap);
            } else {
                printf(": Nothing sent\n");
            }
        } else {
            printf(": Nothing sent\n");
        }
    >>>>>;
}
```
collector in.resolved on “garbage” {
  record = <<<<
    ....
    printf(LSE_time_print_args(LSE`_time_now));
    printf(“: %d\n”, *datap);
  >>>>;
} Output

$ ./Xsim none
0/0: 1
1/0: 2
2/0: 3
Simulator has no more events at time 4/0
Finish Time: 4/0
Data Collector Details

```cpp
collector event : instance {
    decl = <<<< state declaration >>>;
    init = <<<< state initialization >>>;
    record = <<<< data collection >>>;
    report = <<<< data reporting >>>;
};
```

- Collectors can store state
  - Declare state in `decl` section
  - Initialize state in `init` section
- Report statistics at end of simulation
  - Aggregate statistic during simulation in `record` section
  - Display statistic in the `report` section
Communication Data Types

- Port-to-port communication is typed
  - Each module port has a type
  - Compatible types can be connected
  - Modules can have *polymorphic* port types
  - Type Inference resolves polymorphism

- Additionally all communication is tagged with a *DynId* (LSE_dynid_t)
  - Annotate information for statistics collection
  - Store dynamic instruction based information
  - Extensible to store configuration specific information

```
Generator out:int Generator
out:int in:int Garbage
int int
Flip-delay
```
Module Parameters

- Modules are templates for hardware blocks
- Tailor modules for specific purpose by setting parameters

\[ \text{module-name.parameter-name} = \text{value}; \]

Parameters can customize:

- Basic module features
  - Size of a cache
  - Depth of a queue
- Interfaces
  - Existence of ports
  - Existence of other parameters
- Algorithms
Parameters in Action

```java
var i : int;
for(i =0; i < 5; i++) {
    generator.data[i] = 2*i + 1;  // Set a parameter
}
```

Output

```
$ ./Xsim none
0/0: 1
1/0: 3
2/0: 5
3/0: 7
4/0: 9
Simulator has no more events at time 6/0
Finish Time: 6/0
```
Algorithmic Parameters

- In addition to simple values, parameters can be algorithms too
- For example
  - Arbitration logic on an arbiter
  - Branch predictor state machine
  - Cache replacement policy
  - Output data on a reusable data producer
Algorithmic Parameters in Action

Building Fetch Logic

- Customize the combiner module
- The combiner has parameterizable ports
- The combiner defines a user point called combine

```verbatim
instance pc_adder : combiner;
pc_adder.inputs[0] = "pc";
pc_adder.outputs[0] = "npc";
padder_adder.combine = <<<
    *npc_data = *pc_data + 4;
    *npc_id = pc_id;
    *npc_status = LSE_signal_something;
>>>;
```
Control in LSE
Building Fetch Logic

instance pc : delay;
  pc.initial_state =
    delay::init("0");
  pc.out ->:int pc_adder.pc;
  pc_adder.npc -> pc.in;

Output

$ ./Xsim none
Unknown port status at time 0
==== Dumping port status ====

Instance pc:
  Port out:
    global : dYeUaU,
  Port in:
    global : dYeUaU,

Instance pc_addr:
  Port npc:
    global : dYeUaU,
  Port pc:
    global : dYeUaU,
Handshaking in LSE

- Three signals
  - Forward DATA
  - Forward ENABLE
  - Backwards ACK
- Bus inspired handshaking
- Default control is back pressure
**Introduction to the Model of Computation**

- LSE uses a 3-value logic scheme
  - DATA
    - LSE_signal_unknown
    - LSE_signal_something + actual data
    - LSE_signal_nothing
  - ENABLE
    - LSE_signal_unknown
    - LSE_signal_enabled
    - LSE_signal_disabled
  - ACK
    - LSE_signal_unknown
    - LSE_signal_ack
    - LSE_signal_nack

```bash
$ ./Xsim none
Unknown port status at time 0
==== Dumping port status ====
Instance pc:
  Port out:
    global : dYeUaU,
  Port in:
    global : dYeUaU,
Instance pc_addr:
  Port npc:
    global : dYeUaU,
  Port pc:
    global : dYeUaU,
```
Model of Computation (continued)

- At the beginning of each cycle all signals have value `LSE_signal_unknown`
- Instance evaluation causes signals to become more defined
  - Signals transition from unknown to other state
  - No transitions (or data changes) from other states
- Iterative convergence
  - Evaluate until all signals are *not* unknown, then move to next cycle
  - Fixed points with unknown signals are errors
Control in LSE

Modules with State

- Module A outputs **DATA** based on its state
- Module B **ACKs** the data based on internal state
- Module A **ENABLEs** data if a positive **ACK** is received
- Modules A and B update state if **ENABLE** is asserted
Modules with State

- Output **DATA** based on internal state
- Output **ENABLE** calculated from input **ACK**
- Input **ACK** calculated from output **ACK**
Combinational Modules

- Output **DATA** based on its input
- **ENABLE** and **ACK** flow through
Back Pressure Control

- By default **ACKs** flow backwards
  - State elements create boundaries
  - State elements convert **ACK** into **ENABLE**

![Diagram showing state element 1, combinational instance, and state element 2 with arrows indicating flow]

- If State Element 2 is full
  - Negative ACK flows backwards to State Element 1
  - State Element 1 converts negative ACK into negative **ENABLE**
  - State Element 1 preserves its current state
  - State Element 2 does not update its state
Control in LSE
Combinational Loops

1. **pc** sets **out** DATA signal
2. **pc_adder** sets **npc** DATA signal
3. **pc** waits for **out** ACK before setting **in** ACK and **out** ENABLE
4. **pc_adder** waits for **npc** ACK before setting **pc** ACK, and it waits for **pc** ENABLE before setting **npc** ENABLE

---

Output

```
$ ./Xsim none
Unknown port status at time 0
==== Dumping port status ====
Instance pc:
  Port out:
    global : dYeUaU,
  Port in:
    global : dYeUaU,
Instance pc_addr:
  Port npc:
    global : dYeUaU,
  Port pc:
    global : dYeUaU,
```
Control Functions

Breaking Combinational Loops

Output Port

Output Control Function

Local Input

Global Output

Input Port

Input Control Function

Global Input

Local Output

enable

data

ack
Control Functions in Action
Building Fetch Logic

pc_adder.pc.control = <<<<
  return LSE_signal_ack |
  LSE_signal_status_data(istatus) |
  LSE_signal_status_enable(istatus)
;
>>>>;

Output

```
$ ./Xsim none
0/0: PC=0x00000000
0/1: PC=0x00000004
0/2: PC=0x00000008
0/3: PC=0x0000000c
0/4: PC=0x00000010
0/5: PC=0x00000014
0/6: PC=0x00000018
...
```
Not all concurrency is created equal!

- Not all models of concurrency solve the mapping problem
  - All inputs needed to produce any output (EXPRESSION, Asim)
    - No zero-latency loops
- Require repartitioning – merging/splitting blocks
  - Partitioning depends on configuration
  - Repartitioning => Remapping!
LSE Concurrent Semantics

- Zero-latency connections

- Zero-latency loops allowed
  - Modules perform partial evaluations
  - *Don’t allow combinational logic loops*

- Iterate until convergence
  - Guaranteed: output signals change only once
  - Efficient: a schedule can be generated

- There is no need to repartition in LSE
Fan Out and Fan In

- All ports are *arrays* of ports
- Modules support scalable inputs/outputs
- Can be indexed *explicitly* or *implicitly*
- For example
  - *arbiter* arbitrates any # of sources
  - *tee* fans out to any # of destinations

```java
instance
   next_pc_select_arbiter;
   pc_adder.npc ->
   next_pc_select.in[0];
   branch_unit.target ->
   next_pc_select.in[1];
```
Adding State to Instances

- Instance state can be augmented
- Parameter `funcheader` declarations and definitions
- Use `${...}` to get LSS defined entities into userpoints

```plaintext
instance wb_arbiter : arbiter;

var highest_priority = new runtime_var("highest_priority", int) :
 runtime_var ref;

wb_arbiter.funcheader = <<<
 #define WIDTH LSE_signal_width(in)
 >>>;
```
Adding State to Instances

Round Robin Arbiter

```c
wb_arbiter.comparison_func = <<<<
    if(LSE_signal_extract_data(status1) == LSE_signal_something &&
        LSE_signal_extract_data(status2) == LSE_signal_something) {
        int dist1,dist2;
        dist1 =(port1 + WIDTH - ${highest_priority}) % WIDTH;
        dist2 =(port2 + WIDTH - ${highest_priority}) % WIDTH;
        return (dist2 < dist1);
    } else {
        /* Handle Other Cases... */
    }
>>>>;

wb_arbiter.end_of_timestep = <<<<
    ${highest_priority} = (${highest_priority} + 1) % WIDTH;
>>>;
```
LSE Modules

• LSE supports two types of modules
  • Hierarchical
  • Leaf

• Hierarchical Modules
  • Use LSS to define interface
  • Use other modules to define behavior
Module Interfaces

- Interface consists of
  - Ports – runtime communication
  - Parameters – customization options

- Interface is not static!
  - Parameter values can affect interface
    - Create ports
    - Create parameters
  - For example:
    - is_BTB = FALSE
    - is_BTB = TRUE
Module Behavior

- Instantiate
  - Create instances from other leaf or hierarchical modules
  - Use parameters to guide instantiation

- Configure
  - Propagate parameter values to child instances

- Connect
  - Module inputs to child instance inputs
  - Module outputs from child instance outputs
  - Use parameters to guide connectivity
Building a Module: Example

- \( n \) cycle delay element
- Hierarchically compose delay modules
- Use a parameter delay to control cycle delay

```
module delayn {
    inport in : int;
    outport out : int;
    parameter delay : int;

    /* Other LSS code */
}
```
Defining the Behavior

- Instantiate `delay` module
- Connect delay to `in` port and `out` port

```plaintext
module delayn {
    inport in : int;
    outport out : int;
    parameter delay : int;

    instance delay0 : delay;
    in -> delay0.in;
    delay0.out -> out;
}
```
Handling Multiports

• Ports have implicit `width` parameter
• Use `width` to make appropriate number of internal connections

```module delayn {
    ...
    instance delay0 : delay;
    LSS_connect_bus(in, delay0.in, in.width);
    for(i = 0; i < in.width; i++) {
        in[i] -> delay0.in[i];
    }
    LSS_connect_bus(delay0.out, out, out.width);
    for(i = 0; i < out.width; i++) {
        delay0.out[i] -> out[i];
    }
};```
Checking Parameter Validity

- Use control flow statements
  - To verify valid connectivity
  - To verify valid parameterization
- Use `punt` function
  - To report errors
  - To abort simulator generation

```c
if(in.width != out.width) {
    punt("in and out width must be equal");
}
```

```c
if(delay <= 0) {
    punt("delay must be > 0");
}
```
Parameter Dependent Instantiation

- Parameters can control
  - Child instantiation
  - Instance connectivity

```java
var i:int;
var delays : instance ref[];
delays[0] = new instance("delay0", delay);
for(i = 1; i < delay; i++) {
    delays[i] = new instance("<<<delay${i}>>>", delay);
    LSS_connect_bus(delays[i-1].out, delays[i].in
                   in.width);
}
LSS_connect_bus(in, delays[0].in, in.width);
LSS_connect_bus(delays[delay-1].out, out,
                out.width);
```
Default Parameter Values

- Parameters can have default values
- Assignments in `module` body give default values
  - Multiple, conditional, and dependent assignments allowed
  - No assignments after parameter is read
- User assignments override default assignments

```plaintext
module example_good {
  parameter rob_size : int;
  parameter isw_size : int;
  rob_size = 32;
  isw_size = rob_size;
};
instance ex : example_good;
ex.rob_size = 64;
```

```plaintext
module example_bad {
  parameter rob_size : int;
  parameter isw_size : int;
  rob_size = 32;
  if(isw_size > rob_size) {
    isw_size = rob_size;
  }
};
instance ex : example_bad;
ex.rob_size = 64;
```
Using Polymorphism

- Module ports can be polymorphic
- Data type in port declaration can be
  - Actual type (int, bool, aggregate, etc.)
  - List of types (e.g. [int | bool])
  - Unconstrained (*)
  - A type variable (‘a, ‘b, etc.)
- Type Inference resolves polymorphism!

```lss
module delayyn {
  inport in : 'a;
  outport out : 'a;
  parameter delay : int;
  /* Other LSS code */
};
```