T04: Logic Verification Environment
Logic CAD of VLSI (046880)

- Review the *Environment* via an example
- Demo
  - Given our demo system
  - Write a Verification environment
    - Unit level
    - One flow random
Demo System

Floor controllers

- I2C Master
- I2C bus F[1:f]
- Up/Down

- I2C Master
- I2C bus E[1:e]
- Floor<f>
- Cancel<f>

Elevator controllers

An Elevator System “Controller”

- xF
- I2C Slave
- BRAIN
- xE
- I2C Master
- I2C Slave

Motor controllers

- I2C bus M[1:e]
- UP/DN/OFF

- I2C Slave

Doors controllers

- I2C bus D[1:e]
- Open/Close
Verification Environment Components

- Generators
  - Generates the specific test(s)

- Stubs
  - Act as environments hiding internal complexity

- Injectors
  - Feed the test to the system

- Monitors/Watchers/Assertions
  - Inspect the external (internal) model signals
  - Classify what is going on

- Verifiers
  - Make sure expected results are reached

- Coverage Analyzers
  - Tracks what happened at a high level
Full System Testing

An Elevator System “Controller”

Generator

Floor controllers

L2C Master

I2C bus F[1:f]

Up/Down

1

2

3

... F

Elevator controllers

L2C Master

I2C bus E[1:e]

Floor<f>

Cancel<f>

Motor controllers

I2C Master

I2C Slave

L2C bus M[1:e]

UP/DN/OFF

Stub

I2C Master

I2C Slave

BRAIN

I2C bus D[1:e]

Open/Close

Stub/Monitor

Coverage

Verifier

Spring 2014
Example Cluster Level Testing

- More options for which parts to include in simulation exist
Unit Level Testing

- Verify the logic of the “brain” vs. entire system
  - Simpler
  - Faster
  - But not enough
“Brain” Interfaces and Transactions

**Inputs**
- **F** = floor
  - Transactions: UP/DN
  - Signals: F<f>[1:0] 2'b00=None; 2'b01=UP; 2'b10=DN
- **E** = elevator
  - Transactions: SET(floor); CLR(floor); ARRIVE(floor)
  - Signals: E<e>_CMD[1:0] SET=1 CLR=2; ARR=3; E<e>_FLR[3:0] = CMD floor

**Outputs**
- **M** = motor
  - Transactions: UP/DN/OFF
  - Signals: UP<e>; DN<e>
- **D** = door
  - Transactions: OPEN/CLOSE
  - Signals: EO<e>; FEO<f>_<e>
A System View

- What “Objects” are there?

class Elevator {
    int floor;  // what floor is the elevator in
    int requestMask;  // what buttons are pressed in the elevator
}
class Floor {
    bool pendingUp, pendingDn;  // what button is pressed
}
class Building {
    vector<Elevator> elevators;
    vector<Floor> floors;
    Brain dut;
}
TLM example: Elevators Interface

- The hardware (in Verilog)
  - input [1:0] E_0_CMD, E_1_CMD, …;
  - input [3:0] E_0_FLR, E_1_FLR, …;
  - And they are provided each clock cycle

- What we want is
  - ElevatorTrans(int elevator, ElevCmd cmd, int floor)

- Or even better a method of the Elevator object
  - Elevator::Trans (ElevCmd cmd, int floor) // where is “int elevator”? 
  - Floor::Trans (ElevCmd cmd)
A Generator

- A single generator – drive all inputs
- Randomized ALL inputs
- Is it as simple as the below?

In each clock cycle
  For each floor
    Randomize floor command (or none)
  For each elevator
    Randomize elevator command (or none)
Generator – A Constraint

- The “ARRIVE” command adds a constraint
  - Can’t arrive without moving
  - Can’t jump over floors

- Suddenly we need to track movements

```java
class Elevator {
    int floor;
    int requestMask;
    int move; // -1 = dn; 0 = stop; 1 = up
}
```

In each clock cycle
  - For each floor
    Randomize floor command (or none)
  - For each elevator
    Randomize elevator command (or none)
    ```java
    if cmd==arrive && move==1 next_floor=floor+1
    if cmd==arrive && move==-1 next_floor=floor-1
    ```
Generator – Constraint Solving

- Our example is simple
  - Elevators move without coordination
  - Still we need apply “realistic” floor reporting

- Real examples are far more complex
  - Processor’s stream of input commands must make “Sense”
  - Network Switch packets must abide some known standard
  - etc., etc.

- But constraints are useful even for our simple example
  - We will get back to constraints later…
Why do we “Monitor”?

- Tests could be coded as functions on “system state”
  - E.g. for each clock Validate(Building)

- What is the problem with this approach?
  - It couples the tests with the injectors and generator environment
  - For example if we later decide to go for FULL SYSTEM TEST we need to re-write it

- Monitor only relies on the “Brain” IO signals
- Builds the system state from that information only
- A “shadow” system is maintained for tracking state
Given the "Elevator" class

Need to add method to change the "shadow" elevator state

```cpp
Elevator::handleElevatorCmd(ElevCmd cmd, int f) {
    if (cmd == ARR) { floor=f;}
    if (cmd == SET) {requestMask |= f;}
    if (cmd == CLR) {requestMask &= ~f;}
}

Building::handleElevatorCmd(int e, ElevCmd cmd, int f) {
    elevators[e].handleElevatorCmd(cmd, f);
}
```
A Monitor per Interface

- We will need to cover all the interfaces:

  ```
  Building::handleElevatorCmd(int e, ElevCmd cmd, int f) {
    elevators[e].handleElevatorCmd(cmd, f);
  }
  Building::handleFloorCmd(int e, FloorCmd cmd, int f) {
    floors[e].handleFloorCmd(cmd, f);
  }
  Building::handleMotor(int e, bool up, bool dn) {
    ...
  }
  Building::handleDoor(int e, int f, bool open) {
    ...
  }
  ```

- Monitors should be registered and invoked every clock cycle!
  - Best if they are only invoked if change happen on their watched signals
The concept of Aspect coding

- Regular Coding
  ```c
  int factorial(int n)
  
  if n > MAX_ALLOWED
      throw Exception
  
  if n < 0
      throw Exception
  
  if n <= 1
      return 1
  ``
  return n*factorial(n-1)

- Aspect Coding
  ```c
  int factorial(int n) // good case
  
  if n <= 1
      return 1
  ``
  return n*factorial(n-1)

  ```c
  extend factorial(int n) // exception
  
  if n > MAX_ALLOWED
      throw Exception
  
  if n <= 0
      throw Exception
  ``

  ```c
  extend factorial(int n) // profile
  
  factorialCounter++;
  ```
The concept of Aspect coding

- Tests are “layers” on top of the methods handling signal changes
- For example:

```c++
// check that door was opened if previous floor required it. elevators[e].needToOpen track that need.

extend Building::handleDoor(int e, int f, bool open) { // command open a door clears needToOpen
    if (open) elevators[e].needToOpen = false;
}

extend Elevator::handleElevatorCmd(ElevCmd cmd, int f) { // when an elevator arrives to floor it sends
    if (cmd==ARR) { // an ARR command.
        if (neededToOpen) { // needToOpen can’t be true if arriving
            print ERROR “had to open on floor %d” f-1;
            neededToOpen = false;
        }
        
        if ((dir == UP | dir == DN) && (isFloorRequested(requestMask , f)) {
            neededToOpen = true;
        }
        ....
    }
}
```
Scoreboard – a verification template

- Scoreboards are dynamic verification entities act as repository for expected events
- For example:
  - When an elevator “floor” button is pressed it can be stored on the scoreboard
  - A test may check that all such events are eventually removed from the scoreboard

extend Buidling::handleElevatorCmd(int e, ElevCmd cmd, int f) {
    if (cmd == SET)
        scoreboard[e, f] = true;
    else if (cmd == CLR)
        scoreboard[e, f] = true;
}

extend Building::handleDoor(int e, int f, bool open) {
    if (open) elevators[e].needToOpen = false;
}
A word about “events”

- Consider the random generation of events
- “Bugs” normally hide in the corners of our system space
  - E.g. Two elevators are on same floor
    - Going to same direction or the reverse direction
- A good test plan define these cases
- Then we try to “constrain” the generator such that it generates these events
- Tracking which events were actually tested is key
What do we want to test?

For example:

- **Correctness**
  - All calls are eventually treated
  - Elevator stops at requested floors

- **Performance**
  - Elevators do not stop without a reason
  - Elevators do not travel without a reason
Something to think about…

- Suggest more checks
- Suggest events to cover
- How can you program them in C++ ?
- Assume the “main” generates the inputs
- Make the main call the “monitors”