Today’s topics:
- Structural Models
- Connectivity
- And Data Models
We learned some basic rules:

- The 4 principles of dealing with complexity
  - Abstraction – of representation
  - Hierarchy – divide and conquer
  - Regularity – multi-use
  - Design-Methodology – automation relies on selected sub-sets

- The design is refined with information about:
  - Structure
  - Behavior
  - Physical parameters

- In this lecture we focus on the Structure: 
  Hierarchy and Connectivity
Data Explosion and Tool Capacity

<table>
<thead>
<tr>
<th>Representation of a uP</th>
<th># of Entities</th>
<th>Memory for Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>High level model</td>
<td>~50k lines</td>
<td>2-5MB</td>
</tr>
<tr>
<td>RTL - Register Transfer Level</td>
<td>~500k lines</td>
<td>10-50MB</td>
</tr>
<tr>
<td>Gates</td>
<td>~5M</td>
<td>100-400MB</td>
</tr>
<tr>
<td>Transistors</td>
<td>~50M</td>
<td>1-2GB</td>
</tr>
<tr>
<td>Polygons</td>
<td>~500M</td>
<td>200GB</td>
</tr>
</tbody>
</table>

- But flat model runtime is the real issue!
- P&R practical limit is ~2M place-able instances
  - TAT - Turn around time > 3 days
Terminology for Structural Modeling by Hierarchical Organization of *cells*:

- The system structure is a collection of interconnected components, that are recursively divided into sub-components.

- *Cells*, entities, blocks, modules, macros, elements,….. These are all names for the “boxes”.
Definition and Use

- The use of a cell is called an Instance
- The definition of a cell is called a template or prototype or just Cell
- The Design is an instantiation tree
But for most Tools it looks like this:

- What is the difference?
- What’s the point?

Abstraction!
Connectivity: Node/Net

- Instances are interconnected by *nodes* or *nets* *
- A *node* represents an ideal connection (lumped)
- A *net* is a collection of wires, implementing a *node*
  so eventually has the same value at all of its points
- A node/net carries the same signal among all components connected to it

* Terminology may change between publication, book or company
A Programming Allegory

- Cell = Procedure
- Instance = The invocation of it
- Net = a variable, but can’t be “double set”
- Abstraction = API

What are procedure arguments?

What is breaking the allegory?

```c
int factorial(int n) {
    if (n==1) return 1
    return (factorial(n-1)*n)
}

int comb(int n, int k) {
    int x = factorial(n);
    int y = factorial(n-k);
    int z = factorial(k);
    return(x/y/z);
}
```
Focus on

STRUCTURE
Folded model

- Each Cell is defined once
- Cell definition contains
  - Instances of other cells
  - Connections between the instances
- Most memory efficient model
- But most complex to traverse
  - Just remember, will show later…

```text
{Top}
{i3 {A} i1 {B}}
{i4 {C} i2 {D}}
{A} {B} {C} {D} {E} {F} {J}

```
Folded Example

Class Cell(name Top) {
    Instances {
        {Class Instance(name i1, master Inv)}
        {Class Instance(name i2, master Inv)}
    }
}

Class Cell(name Inv) {
    Instances {
        {Class Instance(name n, master NMOS)}
        {Class Instance(name p, master PMOS)}
    }
}

Class Cell(name PMOS) {}  
Class Cell(name NMOS) {}
Flat model

- Top contains
  - Just the Leaf cells instances
  - Connections between them
- Medium memory usage but easy to traverse
- No hierarchy = no re-use = no abstraction!
Flat Example

{Top}

Class Cell(name Top) {
    Instances {
        {Class Instance(name i1/p, master PMOS)}
        {Class Instance(name i2/p, master PMOS)}
        {Class Instance(name i1/n, master NMOS)}
        {Class Instance(name i2/n, master NMOS)}
    }
    Class Cell(name PMOS) {}  
    Class Cell(name NMOS) {}  
}
**Occurrence (Expended) model**

- A *Folded* model + tree of all instances
- Easy to traverse but Worst memory usage
- Hierarchy is maintained
- Both instance and cell based analysis supported
Occurrence Example

Class Cell(name Top) {
    Instances {
        {Class Instance(name i1, master Inv)}
        {Class Instance(name i2, master Inv)}
    }
}

Class Cell(name Inv) {
    Instances {
        {Class Instance(name n, master NMOS)}
        {Class Instance(name p, master PMOS)}
    }
}

Class Cell(name PMOS) {}
Class Cell(name NMOS) {}

Class OccInst (name top) {
    Class OccInst (name i1) {
        Class OccInst (name i1/n) {}
        Class OccInst (name i1/p) {}
    }
    Class OccInst (name i2) {
        Class OccInst (name i2/n) {}
        Class OccInst (name i2/p) {}
    }
}
Hierarchical names in the Occurrence model

- Similar to file names in hierarchical directories
  - e.g. i4/i1/i1
- Can be extended with cell names
  - e.g.{Top}/i4{C}/i1{F}/i1{E}
- What is the top instance?
  - There is none!
Why Occurrence?
For example Parametric Cells

- **Standard Cell methodology**
  - Implement logic gates once
  - Logic function may be attached to each output e.g. \((a + b & c)\)
  - Circuit design usually done a *Folded* model

- On modern process technologies once placed, the standard cell is affected by its environment! (e.g. well proximity effect)
  - Extraction tools produces a *Flat* model

- A timing analysis tool may need to use
  - The exact transistor parameters – Different on each instance
  - The logic function – A cell property
  - So such tools are coded with an *Occurrence* model
Occurrence vs. Runtime Context

- A common tradeoff in algorithm (tool) design:
  
  **Memory vs. Runtime Complexities**

- What if entire occurrence tree does not fit in memory?
- Instead of exploding entire hierarchy into a tree
- Do it on demand!

- Replace pointer to occurrence object by
  - Pointer to Folded model instance
  - A “stack” of parents – also known as context

**Occurrence**

```cpp
class occInst {
    Instance *inst; // the folded inst
    class occInst *parent;
    set<class occInst*> children;
}
```

Tree is pre-built and objects used as needed

**Folded Model Runtime Context**

```cpp
class ctxInst {
    list<Instance *> ctx; // all folded insts chain
}
```

ctxInst is allocated as needed by the algorithm
Choosing the Right Hierarchy
A side note: Need for Automation

- Top-down vs. bottom-up?
  - e.g. by pipeline stages or by bit-slices?
- Organize by flow of data or by flow of control?
- Is there an optimal “branching factor?”
- Practical problems:
  - Mapping between views? (mismatching hierarchies)
  - parameterized cells?

```
+---+----+
<table>
<thead>
<tr>
<th>R</th>
<th>U1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U2</td>
</tr>
<tr>
<td></td>
<td>U3</td>
</tr>
<tr>
<td></td>
<td>c1</td>
</tr>
<tr>
<td></td>
<td>d1</td>
</tr>
<tr>
<td></td>
<td>c2</td>
</tr>
<tr>
<td></td>
<td>d2</td>
</tr>
<tr>
<td></td>
<td>c3</td>
</tr>
<tr>
<td></td>
<td>d3</td>
</tr>
</tbody>
</table>
```

```
+---+----+
<table>
<thead>
<tr>
<th>R</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>c1</td>
</tr>
<tr>
<td></td>
<td>d1</td>
</tr>
<tr>
<td></td>
<td>c2</td>
</tr>
<tr>
<td></td>
<td>d2</td>
</tr>
<tr>
<td></td>
<td>c3</td>
</tr>
<tr>
<td></td>
<td>d3</td>
</tr>
</tbody>
</table>
```
Focus on

CONNECTIVITY
Ports (or pins, or terminals)*

- Ports provide means to pass values into the Cell
  - Similar to parameter/argument relation in software
- Directionality: IN, OUT, or INOUT
  - This is actually behavioral information
  - NOTE: This is the intended logic signal flow direction. Electrical charge flows by the rules of physics!

* Terminology may change between publication, book or company
Instance Ports*

- Represent the instantiation of the Cell Port
- “Connects” the Cell Port, instance and Node
  - In many models Instance Ports only exists when a connection is made

* Terminology may change between publication, book or company
Folded Example with Connectivity

Class Cell(name Top) {
  Instances { … }
  Nodes {
    {Class Node(name I, isPort) {
      InstancePorts { ClassInstPort(inst i1, port I, node I) } }
    {Class Node(name Z) {
      InstancePorts {
        ClassInstPort(inst i1, port Q, node Z)
        ClassInstPort(inst i2, port I, node Z) }
    {Class Node(name O, isPort) {
      InstancePorts { ClassInstPort(inst i2, port Z, node O) }
    }
  Class Cell(name Inv) { … }
  Instances { … }
  Nodes {…}
}
Class Cell(name PMOS) { Nodes {…} }
Class Cell(name NMOS) { Nodes {…} }

{Top}
i1 {Inv} i2 {Inv}

{Inv}
p {PMOS}
I Z O

n {NMOS}
I Q
Connection Across the Hierarchy

- A partial view of Z node objects

```
{Top}
i1 {Inv}  i2 {Inv}
I  Z  O

{Inv}
p {PMOS}
I  Q

{n} {NMOS}
```

```
instPorts

i1 instPorts

Z instPorts

i1%Z inst node port

i2%Z inst node port

Q instPorts

p%D inst node port

p%D inst node port

p%G inst node port

p%G inst node port

I instPorts

p%G inst node port

p%G inst node port
```
Traversals – Getting from A to B...

- For example: **Forward Traversal**
  - Given an out instance port of an instance within the hierarchy (e.g., i1/n/D)
  - Find all the input ports of all **primitive** instances connected to it

- Flat model:
  - Trivial since all primitives appear at the single hierarchy
    - Use “node” pointer to get to the node then loop over all instPorts

- For Folded:
  - Get the top node connected to that instance
    - While the current node has a port change it to the node connected to that port on the parent instance
    - Recurse down from that node through all instance ports collecting all primitive cells instance ports found

- For occurrence:
  - How can we extend the occurrence model to make this process more efficient?
    - Keep occurrence nodes trees
Folded Model Traversal

- Given starting instPort i1/p%D
- Get to curNode = i1/p%D->node = i1/Q
- While curNode has port
  - curInst = prevInst(context)
  - find instPort of curInst that (instPort.port == curNode)
  - curNode = curInstPort.node
- foreach instPort of curNode
  - Recurse down via the instPort
  - Stop when reached primitive cell
Global Nodes?

- Like global (external, static) variables in software
  - You can always assume that they are available
- They implicitly cross each cell interface
- Often used for power supplies, clocks,…
  - To simplify schematics and size of model
- Often a source of trouble
  - The code does not like exceptions
A Net-Cell?

- We said that a net carries the same signal at every point. This is a good abstraction for the logic designer, but not for the electrical circuit designer.....

- In essence, a net can be modeled as a cell, with pins at its connection points to cells in the logic view. This cell may have an internal sub-structure:
Interconnect Trees

- A “well-behaved” net-cell contains a tree
  - No cycles
  - Signal source s0, sinks s1, s2, s3, ....

- But in real life even a single via cause a loop
  - Requires a tool to convert into a DAG
Connectivity at Various Abstractions

Images by Concept's “NLView” and Cadence’s “Virtuoso”
How are these concepts effect and impact implementation of TOOLS, SOFTWARE
Graph-Theory Terms for Circuit Model

- In a Graph $G(V,E)$
  - An edge connects 1 pair of vertices
- In a Hypergraph:
  - Edges may be incident to any number of vertices
- Bipartite graph:
  - A graph with 2 sets of vertices, and each edge has an endpoint in both sets.
- For complete description, we must designate cells by their ports!
  - Circuit is a graph: nodes / cells are vertices; ports are edges
  - Tri- partite model : cells / nodes / ports with connecting edges
Circuit as a tri-partite graph

- Cells, ports, nodes = vertices
- Can be simplified by ignoring the ports (as bi-partite)
- For some purposes you can ignore nodes or ignore cells
The Connectivity as a Directed Graph

- **Graph vertices:**
  - Cells (usually leaf-cells), or Nodes (i.e. signals)
    - depending on what you are trying to do

- **Graph edges:**
  - Each multi-port node is treated as several 2-port edges originating from the source terminal of the node
  - Note that leaf cells (primitives) were modeled here as additional edges to represent their behavior
Combinational Logic Networks (CLN)

- Apply the following methodology:
  - Leaf cells have
    - Multiple inputs and a single output
    - A combinational logic function
  - Ports
    - At all levels are either inputs or outputs
    - Of the root called primary inputs and primary outputs
  - Each node has a single source terminal
  - No “logic loops”

- With these restrictions:
  - A circuit can be represented by a directed graph.
    - What are the vertices? What are the edges?
Removing some CLN Restrictions

- Allow multiple-output cells
  - How would you construct a graph now?
- Allow synchronous delay elements
  - But no combinational loops!
- This is our “RTL abstraction”
Buses*: Special kinds of Nodes

- Name is borrowed from public transportation. The basic idea is to use the same node for multiple “users”, instead of many different point-to-point wires.

- Different possible characteristics of “bus”:
  - Output pins of multiple cells drive the bus
    - Possible contention among drivers?
    - Tri-state logic? ("polite" usage of the bus: one active driver at a time)
    - “wired” logic?
  - INOUT pins of some cells drive the bus
    - Topological order may be lost (Is there a loop somewhere?)

- A special electrical or logic protocol arbitrates this node
  - Precharging? Request/Acknowledge?

* Terminology may change between publication, book or company
Netlists: Textual Representation of Hardware Models

- **netlist** = list all node connected to each cell instance, or all cell instances connected to each node
- A file representation of in memory data model
  - Too many flavors Verilog, VHDL, Spice, SPF, EDIF …
- Can be hierarchical or flat
- At a mathematical form a netlist = *Incidence matrix*
  - rows=cells or pins, cols=nets, 1=in -1=out

```
0 1 0 1 0
-1 1 1 -1 0
1 -1 -1 0 1
0 1 0 0 -1
```

Example incidence matrix
Example netlist (in Verilog syntax)

module rotator(Init, Load, Clk, Rst);
input [7:0] Init;
input Load;
input Clk;
input Rst;
// empty module… for now
endmodule

module rotcomp (Init, Load, Clk, Rst, Test, Limit);
input [7:0] Init;
input Load;
input Clk;
input Rst;
input [7:0] Test;
output Limit;
wire [7:0] A;
  rotator rotate( Init, Load, Clk, Rst, A);
  comparator compare(A, Test, Limit);
endmodule

module comparator(A, B, EQ)
input [7:0] A, B;
output EQ;
  assign EQ = ~| (A ^ B);
endmodule
Internal and External Forms

- Netlists are often used as external/intermediate forms
- Models of the leaf cells (primitives) are either built-in or given as an additional input, such as a ‘cell library file’
- A tool reads a netlist, builds internal model, writes out a new netlist
Folded Hierarchy Data Structure

- As an object oriented model
- UML Class Diagram
- Tutorial and “Wet” homework
Occurrence/Flattened Data Structure

- Can be derived by recursive expansion of Folded model

Keep in mind:

- Prepare for flexibility (additional fields and pointers)
- No fixed-size arrays…. No hard limit is acceptable
- For faster access, consider using Hash-tables
- Have access routines for various traversals. Avoid pointer-chasing in your application code
- Marking objects (“visited” bits…., clearing bits….)
- To reduce paging: place application-specific data outside of the connectivity database, with pointers to the objects…
- Plan for serialization: What happens to all the pointers when you want to store the entire model on disk?
Attributes: What are they? What for?

- Attributes (or properties) are (name,value) pairs, which can be attached to cells, ports, nodes or their instances
  - A mechanism to add some behavioral modeling information and physical parameters into a structural hardware model
  - It can be used for parameterization, by passing symbolic values
  - Bad idea: to infer behavioral/physical information from names of cells/ports/nodes
  - Good idea: use attributes; Names should only be used as identifiers

- Predefined attributes help standardization and efficiency

- By allowing to define new attributes, models become extensible and can be used for new purposes
Attributes: Sparse and Dense

- “sparse” attribute – appear on few objects
  - E.g. mark “power” nodes
- “dense” attribute – appear on most objects of some type
  - E.g. “visited” in some graph traversal code

What are the differences in efficient implementation?

- Sparse attributes:
  - Should be allocated on demand
  - May require hash or sorted container if many are defined
- Dense attribute
  - Best is storage is O(N). How can we use an array?
  - O(1) “cleanup”
Abstractions for functional modeling

- All these levels can be expressed textually in Hardware Description Languages (HDLs)
- The higher levels are sometimes called “behavioral” - but actually they are all mixtures of structure and behavior
- In the tutorial we focus on a logic network model (given to us as a netlist) , and build a tool to validate its behavior…
Summary

- **H/W structure modeling**
  - Hierarchy and connectivity - cells, ports and nodes
  - Occurrence and flattening
  - Hypergraphs and graphs
  - Netlists
  - Internal data structures for hierarchy and connectivity
  - Logic networks: directed acyclic graphs; RTL abstraction