Tutorial 2
Hierarchical Connectivity Model
HCM

Flattener example
Hierarchical Connectivity Model UML

- Design:
  - A container to hold a set of cells

- Cell:
  - A single hierarchy of the design

- Instance:
  - The instantiation of another cell within the cell

- Port:
  - Allows connecting to cell nodes from outside

- Node:
  - A connection of ports and instance ports

- InstPort:
  - The instantiation of the master cell port on its instance
HCM - Notes

- An instance points to both:
  - The containing cell
  - The master cell
  - Both should point to a valid object

- An instPort points to:
  - The instance
  - The node in the containing cell
  - The port in the master cell of the instance
  - All must be valid for the instance port to exist

- A port must have a node

- At the lowest level of hierarchy exist cells that have nodes that all of them have ports but no instance ports. No instances exist in these cells
HCM - Directory Structure

bin/ # holds several utilities for manipulating gate level
src/ # holds the actual HCM code
include/ # the header files HCM applications should include
test/ # some example programs using HCM

Makefile # How to compile an application using HCM
ISCAS-85/ # a set of example Gate Level Verilog circuits
flattener/ # the example code used in this tutorial
An HCM Application

#include “hcm.h”

using namespace std;

// create a container for holding the design
hcmDesign* d = new hcmDesign("MyDesign");

// read in the standard cells definition
d->parseStructuralVerilog("../ISCAS-85/stdcell.v");

// read in the design. Note it should be ordered “Declare before use”
d->parseStructuralVerilog("my_design_modules.v");

// find a cell by name
hcmCell *topCell = d->getCell("myCellName");

// loop over all nodes in the cell
std::map<std::string, hcmNode*>::const_iterator nI;
for (nI = cell->getNodes().begin(); nI != cell->getNodes().end(); nI++) {
    hcmNode *node = (*nI).second;
}
An HCM Application

// Some properties example:

// create a double property
cell->setProp("tcd", 0.5);

double tcd;
cell->getProp("tcd", tcd);

// delete a property
cell->delProp<double>("tcd");

// when the property is not defined
assert(cell->getProp("tcd", tcd) == NOT_FOUND);
module D ( DI , DO );
input DI;
output DO;
endmodule
module C ( CI , CO );
input CI;
output CO;
wire n;
D D1 (.DI(CI), .DO(n)) ;
D D2 (.DI(n), .DO(CO)) ;
endmodule
module C ( CI , CO );
input CI;
output CO;
wire n;
D D1 (.DI(CI), .DO(n)) ;
D D2 (.DI(n), .DO(CO)) ;
endmodule
module B (BI, BO);
input BI;
output BO;

D D3 (.DI(BI), .DO(BO)) ;

endmodule
module A (AI1, AI2, AO1, AO2);
input AI1, AI2;
output AO1, AO2;

B B1 (.BI(AI1), .BO(AO1));
C C1 (.CI(AI2), .CO(AO2));

endmodule
Folded model

Lecture recall

- 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
**Flat model**

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

![Diagram of Flattening](image)
Flattening

- Converting a *folded* model design to a *flat* model design
  - “Break” the abstraction

Motivation

- Easier to traverse
- Ability to have unique properties for “identical instances” in different occurrences of the cell
Flattening

sCell (Folded model topCell)

| Flattening |

| Flattening |

dCell (Flat model cell, Name: topCellName_flat)

hcmCell *flatCell = hcmFlatten(cellName + string("_flat"), topCell, globalNodes);
Flattening

- Flat model cell name:
  - "Folded model top cell name"+_flat"

- Call hcmFlatten function
  - Creates flat model cell

```hcmCell *flatCell = hcmFlatten(cellName + string("_flat"), topCell, globalNodes);```

- Global nodes are usually power nodes
  - VDD, VSS
  - Keep names for all sources
Flattening

**hcmFlatten**

- **Returns:** Flat model cell
  - **Declaration:**
    ```
    hcmCell *hcmFlatten(string flatCellName, hcmCell *sCell, set< string> &globalNodes)
    ```
  - **Step 1:**
    - Creates the cell in the same design
    ```
    hcmCell *dCell = sCell->owner()->createCell( flatCellName );
    if (dCell == NULL) {
        error message
        exit (1)
    }
    ```

La imagen muestra un diagrama que ilustra el proceso de planchar, donde el nodo `sCell` (nodo doblado) se pasa a `dCell` (nodo planificado) mediante la función `hcmFlatten`. El diagrama muestra un esquema de las conexiones entre los nodos antes y después del proceso de planchado.
Flattening
hcmFlatten

- Step 2: Copy all (folded top cell) ports & connected nodes
  - Go over top cell nodes
  - For each node, which is connected to a port (not internal node)
    - Create a new node for dCell
      - Keep original node name
    - Create a new port in dCell
      - Connected to the node
      - Direction as in sCell
      - Always gets the node’s name
Step 2: Copy all (folded top cell) ports & connected nodes

- Go over top cell nodes
- For each node, which is connected to a port (not internal node)
  - Create a new node for dCell
    - Keep original node name
  - Create a new port in dCell
    - Connected to the node
    - Direction as in sCell
    - Always gets the node's name
Step 2: Copy all (folded top cell) ports & connected nodes

- Go over top cell nodes
- For each node, which is connected to a port (not internal node)
  - Create a new node for dCell
    - Keep original node name
  - Create a new port in dCell
    - Connected to the node
    - Direction as in sCell
    - Always gets the node’s name
Step 2: Copy all (folded top cell) ports & connected nodes

- Go over top cell nodes
- For each node, which is connected to a port (not internal node)
  - Create a new node for dCell
    - Keep original node name
  - Create a new port in dCell
    - Connected to the node
    - Direction as in sCell
    - Always gets the node's name

hcmFlatten

Spring 2015
Step 2: Copy all (folded top cell) ports & connected nodes

- Go over top cell nodes
- For each node, which is connected to a port (not internal node)
  - Create a new node for dCell
    - Keep original node name
  - Create a new port in dCell
    - Connected to the node
    - Direction as in sCell
    - Always gets the node's name
**Flattening**

**Flattening**

- **Step 2: Copy all (folded top cell) ports & connected nodes**
  - Go over top cell nodes
  - For each node, which is connected to a port (not internal node)
    - Create a new node for dCell
      - Keep original node name
    - Create a new port in dCell
      - Connected to the node
      - Direction as in sCell
      - Always gets the node's name
Flattening

hcmFlatten

- **Step 2:** Copy all (folded top cell) ports & connected nodes
  - Go over top cell nodes
  - For each node, which is connected to a port (not **internal node**)
    - Create a new node for dCell
      - Keep original node name
    - Create a new port in dCell
      - Connected to the node
      - Direction as in sCell
      - Always gets the node’s name

---

**sCell**  
(Folded model topCell)

**dCell**  
Flat model cell

**Flattening**
**Step 2: Copy all (folded top cell) ports & connected nodes**

- Go over top cell nodes
- For each node, which is connected to a port (not internal node)
  - Create a new node for dCell
    - Keep original node name
  - Create a new port in dCell
    - Connected to the node
    - Direction as in sCell
    - Always gets the node’s name
Flattening

hcmFlatten

\[
\text{std::map< std::string, hcmNode* >::const_iterator nI; for (nI = sCell->getNodes().begin(); nI != sCell->getNodes().end(); nI++) {
    const hcmNode *node = (*nI).second;
    const hcmPort *port = node->getPort();
    if (port == NULL) continue;

    hcmNode *newNode = dCell->createNode(node->getName());
    if (newNode == NULL) { error message & exit }

    hcmPort *newPort = newNode->createPort(port->getDirection());
    if (newPort == NULL) { error message & exit }
}
\]

sCell
(Folded model topCell)

dCell
Flat model cell

Flattening
Flattening

hcmFlatten

- Step 3:
  - Create empty ‘context’ object
  - Call recursive function: “flatten”
Step 3:

- Create empty ‘context’ object
  - Keeps hierarchical context
  - Vector of (pointers to) instances
    - From top level cell down to current cell (bottom: primitive cell)
  - Functions to build “flat” inst/node name that represents the hierarchical structure

- Call recursive function: “flatten”
Flattening

Context Obj

Step 3:

- Create empty ‘context’ object
  - Keeps hierarchical context
  - Vector of (pointers to) instances
    - From top level cell down to current cell (bottom: primitive cell)
  - Functions to build “flat” inst/node name that represents the hierarchical structure

- Call recursive function: “flatten”
Step 3:

- Create empty ‘context’ object
  - Keeps hierarchical context
  - Vector of (pointers to) instances
    - From top level cell down to current cell (bottom: primitive cell)
  - Functions to build “flat” inst/node name that represents the hierarchical structure

- Call recursive function: “flatten”
Step 3:

- Create empty ‘context’ object
  - Keeps hierarchical context
  - Vector of (pointers to) instances
    - From top level cell down to current cell (bottom: primitive cell)
  - Functions to build “flat” inst/node name that represents the hierarchical structure

- Call recursive function: “flatten”
Flattening
Context Obj

Step 3:

- Create empty ‘context’ object
  - Keeps hierarchical context
  - Vector of (pointers to) instances
    - From top level cell down to current cell (bottom: primitive cell)
  - Functions to build “flat” inst/node name that represents the hierarchical structure

- Call recursive function: “flatten”
Flattening
Context Obj

Step 3:
- Create empty ‘context’ object
  - Keeps hierarchical context
  - Vector of (pointers to) instances
    - From top level cell down to current cell (bottom: primitive cell)
  - Functions to build “flat” inst/node name that represents the hierarchical structure
- Call recursive function: “flatten”
Step 3:
- Create empty ‘context’ object for the top cell
- Call recursive function: “flatten”

```cpp
hcmCtx ctx; // empty context for the top cell...
if (flatten(ctx, sCell, dCell, globalNodes)) {
    cerr << "-F- Fatal Error Message";
    exit(1);
}
```
Flattening

Step 4: flatten – Recursive Call

- As long as the current level cell has sub-instances (not primitive) – dive in
  - Push (back) sub-inst to the context
  - Call recursively to flatten
    - With new (static alloc) created context
    - “sCell”: Current instance’s master
Flattening

Step 4: flatten – Recursive Call

- As long as the current level cell has sub-instances (not primitive) – dive in
  - Push (back) sub-inst to the context
  - Call recursively to flatten
    - With new (static alloc) created context
    - “sCell”: Current instance’s master

```c
flatten(<empty-context>, sCell, dCell, globalNodes);
```
Flattening

Step 4: flatten – Recursive Call

- As long as the current level cell has sub-instances (not primitive) – dive in
  - Push (back) sub-inst to the context
  - Call recursively to flatten
    - With new created context
    - "sCell": Current instance’s master

```plaintext
flatten(<empty-context>, sCell, dCell, globalNodes);
flatten(<&a>, A, dCell, gN);
```
Flattening

Step 4: flatten – Recursive Call

- As long as the current level cell has sub-instances (not primitive) – dive in
  - Push (back) sub-inst to the context
  - Call recursively to flatten
    - With new created context
    - “sCell”: Current instance’s master

```
flatten(<empty-context>, sCell, dCell, globalNodes);
flatten(<&a>, A, dCell, gN);
flatten(<&a,&b>, B, dCell, gN);
```
Flattening

Step 4: flatten – Recursive Call

- As long as the current level cell has sub-instances (not primitive) – dive in
  - Push (back) sub-inst to the context
  - Call recursively to flatten
    - With new created context
    - “sCell”: Current instance’s master

```plaintext
flatten(<empty-context>, sCell, dCell, globalNodes);
flatten(<&a>, A, dCell, gN);
flatten(<&a,&b>, B, dCell, gN);
flatten(<&a, &b, &d>, C, dCell, gN);
```
Flattening

Step 4: flatten – Recursive Call

The code:

```cpp
if (sCell->getInstances().size()) {
    int res = 0;
    map< std::string, hcmInstance* >::iterator il;

    for (il = sCell->getInstances().begin(); il != sCell->getInstances().end(); il++) {
        hcmInstance *inst = (*il).second;

        hcmCtx instCtx = ctx; // Got in flatten – from upper level
        instCtx.insts.push_back(inst);

        res += flatten(instCtx, inst->masterCell(), dCell, globalNodes);
    }
    return 0;
}
```
Step 5: flatten – Primitive Cells (Insts)

Create new instance

- Represents the primitive instance in the flat model
- Master cell: Primitive instance’s master cell
- Name: Represents hierarchical structure
  - From context
  - Unique

```
Inst: a {master: A}
Inst: b {master: B}
Inst: c {master: C}
Inst: d {master: C}
```

```
New Instance
Name: a_name/b_name/d_name
Master cell: C
```
Flattening

Step 5: flatten – Primitive Cells (Insts)

string hName = ctx.getHName();

hmInstance *newInstance = dCell->createInst(hName, sCell);

if (newInstance == NULL) {
    cerr << "-F- Could not create new instance: " << hName << " { " << sCell->getName() << " }" << endl;
    exit(1);
}
Flattening

Step 6: flatten – Primitive Cells (Nodes)

- Create / Connect flat model nodes
  - Reminder – Primitive cell nodes:
    - All have ports
      - Port name is node name
      - Node goes from the port into the cell and connects to nothing
    - None have inst-ports

![Diagram of Primitive Cell C]

Node: N1  Port: N1
Flattening

Step 6: flatten – Primitive Cells (Nodes)

- Create / Connect flat model nodes
  - Go over primitive cell nodes (internal, connected to ports only), for each:
  - Get / Build the “wanted node name” in the Flat model top cell
  - Create / Find the node in the Flat model top cell
  - Connect the Flat model new instance (its cell relevant port) with the Flat model node

```
sCell
  Inst: a {master: A}
  Inst: b {master: B}
  Inst: c {master: C}
  Inst: d {master: C}

dCell
  New Instance
  Name: a_name/b_name/d_name
  Master cell: C
```
Flattening

Step 6: flatten – Primitive Cells (Nodes)

- Create / Connect flat model nodes
  - Go over primitive cell nodes (internal, connected to ports only), for each:
  - Get / Build the “wanted node name” in the Flat model top cell
  - Create / Find the node in the Flat model top cell
  - Connect the Flat model new instance (its cell relevant port) with the Flat model node

Cell: C

sCell

Inst: a {master: A}
Inst: b {master: B}
Inst: c {master: C}
Inst: d {master: C}

dCell

New Instance
Name: a_name/b_name/d_name
Master cell: C

L1
Flattening

Step 6: flatten – Primitive Cells (Nodes)

- Create / Connect flat model nodes
  - Go over primitive cell nodes (internal, connected to ports only), for each:
  - Get / Build the “wanted node name” in the Flat model top cell
  - Create / Find the node in the Flat model top cell
  - Connect the Flat model new instance (its cell relevant port) with the Flat model node

![Diagram showing the process of flattening primitive cells (nodes)]
Flattening

Step 6: flatten – Primitive Cells (Nodes)

- Create / Connect flat model nodes
  - Go over primitive cell nodes (internal, connected to ports only), for each:
  - Get / Build the “wanted node name” in the Flat model top cell
  - Create / Find the node in the Flat model top cell
  - Connect the Flat model new instance (its cell relevant port) with the Flat model node

Node name in the Flat model top cell: L1 (next slides)
Flattening

Step 6: flatten – Primitive Cells Nodes

Create / Connect flat model nodes

- Go over primitive cell nodes (internal, connected to ports only), for each:
- Get / Build the “wanted node name” in the Flat model top cell
- Create / Find the node in the Flat model top cell
- Connect the Flat model new instance (its cell relevant port) with the Flat model node

Node name in the Flat model top cell: L1  (next slides)
Flattening

Step 6: flatten – Primitive Cells Nodes

- Create / Connect flat model nodes
  - Go over primitive cell nodes (internal, connected to ports only), for each:
  - Get / Build the “wanted node name” in the Flat model top cell
  - Create / Find the node in the Flat model top cell
  - Connect the Flat model new instance (its cell relevant port) with the Flat model node

Cell: C

sCell

Inst: a {master: A}

Inst: b {master: B}

Inst: c {master: C}

Inst: d {master: C}

L3

dCell

Name: a/b/d
Master cell: C

Name: a/c
Master cell: C

L1

L2

a/L3
Flattening

Step 6: flatten – Primitive Cells Nodes

```c++
std::map< std::string, hcmNode* >::const_iterator nl;
for (nl = sCell->getNodes().begin(); nl != sCell->getNodes().end(); nl++) {
    const hcmNode *node = (*nl).second;

    // May be a global node
    string topNodeName;
    if (ctx.getInstPortTopNodeHName(node->getName(), topNodeName, globalNodes))
        continue;

    // Get the node or create it
    hcmNode *newNode = dCell->getNode(topNodeName);
    if (newNode == NULL) {
        newNode = dCell->createNode(topNodeName);
        if (newNode == NULL) {
            cerr << "-F- Could not create new node: " << topNodeName << endl;
            exit(1);
        }
    }

    // Connect to the new instance
    dCell->connect(newInst, newNode, node->getName());
}
```
Flattening

Step 6: flatten – Primitive Cells Nodes

- Get / Build node name in the Flat model top cell (getInstPortTopNodeHName)
  - The function is called from the primitive instance context object
  - Gets: Primitive cell (inst master) node (N1), global nodes set

- Go over all context instances (bottom to top)
  - d {C} context: d -> b -> a

Top Cell

Inst: a {master: A}
Inst: b {master: B}
Inst: d {master: C}
Inst: N1

→

[Diagram of flattened model]
Flattening

Step 6: flatten – Primitive Cells Nodes – N1 Example

- Instance: d \{C\}
- Build instPortName: d%N1
  - Inst name (in cell B) and instance port – unique name (in the current level)
- Search d instPort: d%N1
  - No
    - Return from the function
    - N1 will not have relevant net in the Flat model top cell
    - Logic: No-one can connect this instance
  - Yes – Next Slide

Top Cell

```
Inst: a \{master: A\}
  Inst: b \{master: B\}
    Inst: d \{master: C\}
```

Name: a/b/d
Master cell: C
Flattening

Step 6: flatten – Primitive Cells Nodes – N1 Example

- d \text{instPort} (d\%N1) was found
- Search \text{instPort} node
  - No
    - Return node name: a/b/d/N1
    - Will be connected to: a/b/d port N1
    - Not connected to other instances
  - Yes – Next Slide

Top Cell

Inst: a \{master: A\}
Inst: b \{master: B\}
Inst: d \{master: C\}

Name: a/b/d
Master cell: C

a/b/d/N1

Spring 2015
Flattening

Step 6: flatten – Primitive Cells Nodes – N1 Example

- `instPort` node – J1
- Is J1 connected to a port
  - Yes
    - Cell B port
    - Move to the next context instance (d -> b -> a)
    - InstPortName: b%J1
    - Continue with the process

---

Top Cell

Inst: a {master: A}

Inst: b {master: B}

Inst: d {master: C}

N1

L1

Spring 2015
Flattening

Step 6: flatten – Primitive Cells Nodes – N1 Example

- instPort node – J1
- Is J1 connected to a port
  - Yes
    - Cell B port
    - Move to the next context instance ( d -> b -> a)
    - InstPortName: b%J1
    - Continue with the process

Top Cell

Inst: a {master: A}
Inst: b {master: B}
Inst: d {master: C}

L1

N1

K1
Flattening
Step 6: flatten – Primitive Cells Nodes – N1 Example

- instPort node – J1
- Is J1 connected to a port
  - Yes
    - Cell B port
    - Move to the next context instance (d -> b -> a)
    - InstPortName: b%J1
    - Continue with the process

Top Cell

Inst: a {master: A}
Inst: b {master: B}
Inst: d {master: C}

Name: a/b/d
Master cell: C
Flattening

Step 6: flatten – Primitive Cells Nodes – N1 Example

- instPort node – J2
- Is J2 connected to a port
  - Yes – As before

Cell: C

sCell

Inst: a {master: A}
Inst: b {master: B}
Inst: c {master: C}
Inst: d {master: C}

J2

dCell

Name: a/b/d

N1

N2

L1
Flattening

Step 6: flatten – Primitive Cells Nodes – **N1 Example**

- instPort node – K2
- Is K2 connected to a **port**
  - No
  - Is K2 Global Node (VDD/VSS/K2)?
  - If yes – Return name (VDD/VSS/K2) and connect in Flat top level

![Diagram of N1 example](image-url)
Flattening

Step 6: flatten – Primitive Cells Nodes – N1 Example

- instPort node – K2
- Is K2 connected to a port
  - No
    - K2 is not Global Node (VDD/VSS)
    - Node name for Flat model: \( a/b/K2 \)
    - Same name when traverse left from c \{C\}
    - Both (c,d) are connected to the same node

---

![Diagram](image-url)
Flattening

Step 6: flatten – Primitive Cells Nodes – N1 Example

- instPort node – K2
- Is K2 connected to a port
  - No
    - K2 is not Global Node (VDD/VSS)
    - Node name for Flat model: \( a/b/K2 \)
    - Same name when traverse left from c \{C\}
    - Both (c,d) are connected to the same node

---

**Diagram:**

- **sCell**
  - Inst: a \{master: A\}
  - Inst: b \{master: B\}
  - K2
  - Inst: c \{master: C\}
  - Inst: d \{master: C\}

- **dCell**
  - L1
  - Name: a/b/d
  - L2
  - Name: a/b/c
  - a/b/K2
Flattening

Step 6: flatten – Primitive Cells Nodes

```cpp
int hcmCtx::getInstPortTopNodeHName(string ipName, string &name, set< string> &globalNodes) { 

    string nodeName = ipName;
    for (int i = insts.size() - 1; i >= 0; i--) {
        const hcmInstance *inst = insts[i];
        string instPortName = inst->getName() + string("%") + nodeName;
        const hcmInstPort *instPort = inst->getInstPort(instPortName);
        if (instPort == NULL) {
            // Some message (not error)
            return(1)
        }
    // we continue up until the node connected to instance is not connected to a port
    const hcmNode *node = instPort->getNode();
    if (node == NULL) {
        // the inst port not connected - use it
        name = getHName(i) + string("/") + nodeName;
        return(0);
    }
    nodeName = node->getName();

    ..........
int hcmCtx::getInstPortTopNodeHName(string ipName, string &name, set<string> &globalNodes) {

    nodeName = node->getName();

    // we might have just stopped at this level
    if (node->getPort() == NULL) {
        // may be a global node so exist
        if (globalNodes.find(nodeName) != globalNodes.end()) {
            name = nodeName;
        } else {
            // the inst port not connected - use it
            name = getHName(i-1) + string("/") + nodeName;
        }
        return(0);
    }
}

// end loop on context insts

name = nodeName;
return(0);
}