EN2911X: Reconfigurable Computing
Topic 03: Reconfigurable Computing Design Methodologies

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Fall 2012
Reconfigurable computing design flow

I. System Specification (C/C++/Java/MATLAB/SystemC)

II. partitioning

HW specification

III. compiling

Verilog

IV. synthesis

V. Mapping & packing

VI. place & VII. route

VIII. configuration

Processor (hard/soft)  HW accelerator

reconfigurable computing system
I. System specification

Use High-Level Languages (HLLs) (C, C++, Java, MATLAB).

Advantages:

- Since systems consist of both SW and HW, then we can describe the entire system with the same specification
- Fast to code, debug and verify the system is working

Disadvantages:

- No concurrent support
- No notion of time (clock or delay)
- Different communication model than HW (uses signals)
- Missing data types (e.g., bit vectors, logic values)
Traditional models for system modeling

- C/C++ system model
  - compilation
  - analysis & results
  - automatic or manual
- Verilog
  - Simulation
  - Synthesis
Using high-level languages (HLL) for HW/SW specification

- Augment the HLL (e.g. C++) with a new data types, classes, library, to support additional hardware-like functionality (e.g. SystemC, ImpulseC)
  - Unified language across all stages of platform design
  - Fast simulation
  - There are many tools for compiling C++
  - HW synthesis from SystemC descriptions are not yet as mature
II. Hardware-Software partitioning

• Given a system specification, decompose or partition the specification into tasks (functional objects) and label each task as HW or SW such that the system cost / performance is optimized and all the constraints on area resources / cost are satisfied.

• The exact performance depends on the computational model in hand
  – Given the same application, a system with an FPGA on a slow bus results in a design with different performance results than a system with a FPGA as a coprocessor.
Task graph and profiling

Each node/vertex is characterized by:
- Hardware area $h_i$
- Hardware execution time $t_i$
- Software memory size $m_i$
- Software execution time $s_i$
- Average number of times the task is executed $n_i$

Each edge is characterized by communication time:
- Transfer time
- Synchronization time
- Average number of times communication take place
Objective of partitioning

Partitioning criteria:

1. Minimize total execution time (latency)
2. Minimize total power and/or HW area
3. Minimize communication between HW and SW
4. Maximize concurrency (SW/HW in parallel) and utilization
Low-level partitioning from software binaries

• Rather than partition from the high-level description, it is possible to compile the program as SW and then partition the resultant executable binary into SW and HW parts.
  – Advantages:
    • No need to worry about which language is being used
    • Can be used to develop dynamic runtime partitioners and synthesizers
  – Main steps:
    • De-compilation of binary to recover high-level information
    • Partitioning and synthesis
    • Binary updating to account for the SW parts that migrated to HW

Extreme solutions

- **All HW solutions:**
  - Minimum design latency (MinT)
  - Maximum design area (MaxA)

- **All SW Solutions:**
  - Maximum design latency (MaxT)
  - Maximum memory space

**Solution of the HW/SW problem is bounded:**

\[
0 \leq \text{area of HW/SW system} \leq \text{MaxA} \\
\text{MinT} \leq \text{Latency of HW/SW system} \leq \text{MaxT} \\
0 \leq \text{memory of HW/SW system} \leq \text{MaxA}
\]
Possible methods to solve the problem

- **Random partitioning**: try a few random partitions and use the best
- **Explicit enumeration**: consider every possible SW/HW partition
  - Optimal but takes exponential time as a function in the number of tasks
- **Greedy approach**: start with SW solution and migrate time-consuming tasks into HW.
  - Not necessarily optimal
  - Limitations from Amdahl’s law: if 80% of the program can be transformed into parallelized HW then the program can be speed up at best by 5x
- **Two possible approaches**: 
  - Optimal technique based on integer linear programming (ILP)
  - Heuristic technique based on Kernighan/Lin partitioning heuristics
1. ILP-based partitioning

Objective:
\[
\text{max } c_1x_1 + c_2x_2 + \cdots + c_nx_n
\]

Constraints:
\[
a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \leq b_1
\]
\[
a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \leq b_2
\]
\[\vdots\]
\[
q_{1,1}x_1 + q_{1,2}x_2 + \cdots + q_{1,n}x_n \leq d_1
\]
\[
q_{2,1}x_1 + q_{2,2}x_2 + \cdots + q_{2,n}x_n = d_2
\]
\[\vdots\]
If \(x_i\) is allowed to take fractional values → linear programming (LP)
else integer linear programming (ILP)

Maximize \[9x_1 + 14x_2 + 20x_3 + 32x_4\]
Subject to \[3x_1 + 5x_2 + 8x_3 + 10x_4 \leq 16\]
\(x_i \in \{0, 1\}\)
Examples of LP

\[
\begin{align*}
\text{max} & \quad x_1 + x_2 \\
\text{subject to:} & \quad 2x_1 + x_2 \leq 4 \\
& \quad 3x_1 + 4x_2 \leq 12 \\
& \quad x_j \geq 0 \ (j = 1, 2)
\end{align*}
\]

- How to solve LP?
- What is the runtime?
- Many toolboxes are available
Examples of ILP

max \ x_1 + x_2
subject to:
2x_1 + x_2 \leq 4
3x_1 + 4x_2 \leq 12
x_j \in \mathbb{Z}^+ \ (j = 1, 2)

2 optimal solutions!

- How to solve ILP?
- What is the runtime?
- Many toolboxes are available
SW/HW Partitioning using ILP

\( x_i \in \{0, 1\} \)  A binary variable that indicate if task i is mapped to HW (0) or SW (1)

\( f_i \)  time when task i is complete

Objective: \( \min f_5 \)

Constraints:

\[ \sum_i h_i (1 - x_i) \leq \text{number of LEs in FPGA} \]
\[ \sum_i m_i x_i \leq \text{memory size} \]

\[ f_1 = t_1 (1 - x_1) + s_1 x_1 \]
\[ f_2 = f_1 + t_2 (1 - x_2) + s_2 x_2 \]
\[ f_3 = f_1 + t_3 (1 - x_3) + s_3 x_3 \]
\[ f_4 = f_4 + t_4 (1 - x_4) + s_4 x_4 \]
\[ f_5 \geq f_2 + t_5 (1 - x_5) + s_5 x_5 \]
\[ f_5 \geq f_4 + t_5 (1 - x_5) + s_5 x_5 \]

\( x_i \in \{0, 1\} \)

ignoring communication overhead
2. Heuristic approach

Total size is constrained by number and size of available FPGA(s)

Kernighan/Lin – Fidducia/Mattheyses algorithm

- Start with all task vertices free to swap/move (*unlocked*)
- Label each possible swap/move with immediate change in execution time that it causes (*gain*)
- Iteratively select and execute a swap/move with highest gain (whether positive or negative); *lock* the moving vertex (i.e., cannot move again during the pass),
- Best solution seen during the pass is adopted as starting solution for next pass
Summary of SW / HW partitioning

• A very important and challenging problem, especially for complex systems. Very relevant for SoCs.
• Metrics for partitioning (e.g., latency, power, area)
• Techniques: exact (ILP) but potentially slow, and heuristics (KL) but potentially not optimal
• Complexity can arise from communication, synchronization, real-time constraints.
III. Compilation

- Reconfigurable configurable has the ability to execute multiple operations in parallel through spatial distribution of the computing resources.
- We have already studied how HDLs (e.g., Verilog) are synthesized to logic.
- High-level algorithmic descriptions lack descriptions of concurrency. When compiling an algorithmic description or SW-based language into HW, it is necessary to expose the concurrency in the application. How can we do that?
Abstract models for circuit computation

- Structural representations (graphs)
- Logic networks
- Finite state machines
- *Data flow graphs*
Data-flow graphs (DFG)

- A data-flow graph (DFG) is a graph which represents a data dependencies between a number of operations.
- Dependencies arise from a various reasons
  - An input to an operation can be the output of another operation
  - Serialization constraints, e.g., loading data on a bus and then raising a flag
  - Sharing of resources
- A *dataflow graph* represents operations and data dependencies
  - *Vertex* set is one-to-one mapping with tasks
  - A *directed edge* is in correspondence with the transfer of data from an operation to another one
Example algorithm

[Giovanni’ 94] Design a circuit to numerically solve the following differential equation in the interval \([0, a]\) with step-size \(dx\)

\[
y'' + 3xy' + 3y = 0
\]

\[
x(0) = x; \ y(0) = y; \ y'(0) = u
\]

```
read (x, y, u, dx, a);
do {
    x1 = x + dx;
    u1 = u - (3*x*u*dx) - (3*y*dx);
    y1 = y + u*dx;
    c = x1 < a;
    x = x1; u = u1; y = y1;
} while (c);
write(y);
```
DFG of algorithm

\[
x_1 = x + \text{dx};
\]
\[
u_1 = u - (3 \times x \times u \times \text{dx}) - (3 \times y \times \text{dx});
\]
\[
y_1 = y + u \times \text{dx};
\]
\[
c = x_1 < a;
\]
Detecting concurrency from DFGs

Extended DFG where vertices can represent links to link graph
DFGs in a hierarchy of graphs

Paths in the graph represent concurrent streams of operations
Control / data-flow graphs (CDFG)

- Control-flow information (branching and iteration) can be also represented graphically.
- Data-flow graphs can be extended by introducing branching vertices that represent operations that evaluate conditional clauses.
- Iteration can be modeled as a branch based on the iteration exit condition.
- Vertices can also represent model calls.
CDFG example

\[
\begin{align*}
  x &= a \times b; \\
  y &= x \times c; \\
  z &= a + b; \\
  \text{if} \ (z \geq 0) \ {\{ \\
    p &= m + n; \\
    q &= m \times n; \\
  \}}
\end{align*}
\]
Code transformations

Data-flow-based transformations:
A. Tree height reduction
B. Constant and variable propagation
C. Common sub-expression elimination
D. Dead code elimination
E. Operator strength reduction
F. Code motion

Control-flow-based transformations:
G. Conditional expansion
H. Loop expansion
A. Tree-height reduction (THR)

- *Tree-height reduction* applies to arithmetic expression trees and strives to achieve the expression split into two-operand expressions to exploit parallelism
- The idea is to attempt to balance the expression tree as much as possible
- If we have \( n \) operations, what is the best height that can be achieved?

\[ x = a \times (b \times c \times d + e) \]

Exploiting the distributive property at the expense of adding an operation.
A. Impact of serializing memory accesses on tree height reduction


- Bandwidth to the memory is critical to realize the speed-up potential of reconfigurable systems
B. Constant and variable propagation

- **Constant propagation** consists of detecting constants operands and pre-computing the value of the operation with that operand. The result might a constant which can be propagated to other operations as input
- **Example:**
  - $a = 0; b = a + 1; c = 2 \times b$
    - Replaced by $→ a = 0; b = 1; c = 2$

- **Variable propagation** consists of detecting the copies of the variable and using the right-hand side in the following references in place of the left-hand side
- **Example:**
  - $a = x; b = a + 1; c = 2 \times a$
    - Replaced by $→ a = x; b = x + 1; c = 2 \times x$
C. Common sub-expression elimination

- *Common Sub-expression Elimination (CSE)* avoids unnecessary computations.
- Example:
  
  ```
  a = x + y;
  b = a + 1;
  c = x + y;
  
  Can be replaced by
  ```
  ```
  a = x + y;
  b = a + 1;
  c  = a;
  ```

  Example:
  
  ```
  t = x*a + x*b;
  ```
  Replace by
  ```
  t = x*(a+b);
  ```
D. Dead code elimination

• *Dead code elimination (DCA).* Dead code consists of all operations that cannot be reached, or whose results is never referenced elsewhere. Example:
  
  ```
  a = x;
  b = x + 1;
  c = 2 * x;
  ```

  The first assignment can be removed if it is never subsequently referenced.
E. Operator strength reduction

- **Operator strength reduction** means reducing the cost of implementing an operator by using a “weaker” one (that uses less hardware / simpler and faster) but still get the same result.

- **Example:**
  - $a = x^2$;
    Replaced by $a = x \times x$;
  - $b = 3 \times x$
    Replaced by $t = x \ll 1; b = x + t$
  - How many adders are needed to implement $231 \times x$? Hint: binary representation of 231 is 011100111.
F. Code motion: hoisting and sinking

- Code motion (up – hoisting or down – sinking) may have the potential benefit of making the two computations independent, thus enabling concurrent execution. Such movement may increase the potential for other optimization techniques.

- **Examples:**
  1. for (i = 1; i < a * b) {...}
     Replaced by → t = a * b; for (i =1; i <= t) {...}
  2. before code hoisting
     ![Diagram](a)
     ![Diagram](b)
     ![Diagram](c)
     ![Diagram](d)

[example from Cardoso ‘10]
G. Conditional expansion

- A conditional construct can be always transformed in a parallel construct with a test in the end.
- Conditional expansion can increase the performance of the circuit when the conditional clause depends on some late-arriving signal.
- However, it can preclude the possibility of hardware sharing.

- If \( (C) \) then \( x = A \) else \( x = B \)
  \( \Rightarrow \) compute \( A \) and \( B \) in parallel, \( x = C \ ?A:B \)
H. Loop unrolling (expansion)

- In *loop expansion*, or unrolling, a loop is replaced by as many instances of the body as the number of operations. The benefits are reducing loop overhead in expanding the scope of other transformations.

- Example:

```c
x = 0;
for (i = 1; i <= 12; i++) {
    x = x + a[i];
}
```

```c
x = 0;
for (i = 1; i <= 12; i = i+3) {
    x = x + a[i] + a[i+1] + a[i+2];
}
```
Data oriented transformations

A. Bit-width narrowing and bit optimizations
B. Data distribution
C. Data replication
D. Scalar replacement
E. Data transformation for floating points
A. Bit width optimizations

- Reconfigurable computing enables the possibility of fine-grain arithmetic. Bit-width inference and analysis can lead to customized arithmetic units that are more efficient than in regular architectures.

```c
int i;
int c[32];
for (i =0 ; i < 32; i++)
    c[i] = i + 5;
```

- Do we really need to have 32 bits for i, c, and the adder?
- Bit-width narrowing can be either static or profile-based dynamic. Static analysis is faster but more conservative.
- Static analysis can be done using interval analysis
- Example: let’s say that a and b are signed 8-bit variables. How many bits do we need for c, d, and e?

```c
    c = a + b;
    d = a-b;
    e = c*d;
```
B. Data distribution

- Partitions a given program array variable into many distinct arrays, each of which holds a disjoint subset of the original array’s data values. Each of these newly created arrays can then be mapped to many distinct internal memory units, but the overall storage capacity used by the arrays is preserved.
- Enables implementations that can access the various sections of the distributed data concurrently (whenever mapped to distinct memories, thus increasing the data bandwidth).

```c
int img[N][N];
...
for(j=0; j < N; j++) {
    ...
    for(i=0; i < N; i++) {
        ...
        = img[j][i];
    }
    ...
}
(a)
```

```c
int imgOdd[N][N/2], imgEven[N][N/2];
...
for(j=0; j < N; j++) {
    ...
    for(i=0; i < N; i+=2) {
        ...
        = imgOdd[j][i/2];
        ...
        = imgEven[j][i/2];
    }
    ...
}
(b)
```
C. Data replication

- Data replication: a data-oriented transformation that increases the available data bandwidth, this time at the expense of an increase in the storage used. In this case, data is copied as many times as desired into distinct array variables, which can then be mapped to distinct memory modules to enable concurrent data access.

```c
int img[N][N];
for(j=0; j < N; j++) {
    for(i=0; i < N; i++) {
        img[j][i] = ...
    }
}

(a)
```

```c
int imgA[N][N], imgB[N][N];
for(j=0; j < N; j++) {
    for(i=0; i < N; i+=2) {
        imgA[j][i] = ...
        imgB[j][i+1] = ...
    }
}

(b)
```
D. Scalar replacement

- Scalar replacement: Identify reusable data items and cache in internal registers for increased performance

```
... int vec[N][4], k[4];
... for(j=0; j < N; j++) {
    ...
    for(i=0; i < 4; i++) {
        ...
        ... = vec[j][i] * k[i];
    }
    ...
}
...

... int vec[N][4], k[4], k0, k1, k2, k3;
... k0 = k[0]; k1 = k[1]; k2 = k[2]; k3 = k[3];
... for(j=0; j < N; j++) {
    ...
    ... = vec[j][0] * k0;
    ...
    ... = vec[j][1] * k1;
    ...
    ... = vec[j][2] * k2;
    ...
    ... = vec[j][3] * k3;
    ...
}
```

[example from Cardoso ‘10]
E. Data transformations for floating points

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
</table>

\[ x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)} \]

S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
Normalize significand: \(1.0 \leq |\text{significand}| < 2.0\)
Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit). Significand is Fraction with the “1.” restored

What is the smallest and largest possible values?
Example: what is the floating point representation of \(-0.75\)?

- \(-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}\)
- \(S = 1, \text{Fraction} = 1000...00_2\)
- \(\text{Exponent} = -1 + \text{Bias} = -1 + 127 = 126 = 01111110_2\)
- Answer: \(1011111101000...00\)

[source: Patterson]
E. Example of floating point addition

• Consider a 4-digit decimal example
  \[ 9.999 \times 10^1 + 1.610 \times 10^{-1} \]

• 1. Align decimal points
  Shift number with smaller exponent
  \[ 9.999 \times 10^1 + 0.016 \times 10^1 \]

• 2. Add significands
  \[ 9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1 \]

• 3. Normalize result & check for over/underflow
  \[ 1.0015 \times 10^2 \]

• 4. Round and renormalize if necessary
  \[ 1.002 \times 10^2 \]

[source: Patterson]
E. Example of floating point addition

• Now consider a 4-digit binary example
  \(1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}\) (0.5 + −0.4375)

• 1. Align binary points
  Shift number with smaller exponent
  \(1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}\)

• 2. Add significands
  \(1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}\)

• 3. Normalize result & check for over/underflow
  \(1.000_2 \times 2^{-4}\), with no over/underflow

• 4. Round and renormalize if necessary
  \(1.000_2 \times 2^{-4}\) (no change) = 0.0625

[source: Patterson]
E. Floating point addition HW

Step 1
- Compare exponents

Step 2
- Shift smaller number right
- Add

Step 3
- Normalize
- Round

Step 4
- Rounding hardware

[source: Patterson]
E. Comparison of HW area requirements

- 32 LEs for integer 32-bit addition
- 840 LEs for floating point single-precision addition
E. Dealing with floating points

- Use of non-standard floating point formats
- Floating point to fixed point conversion

- Inherent trade-off between range and precision. Discretization is inevitable.
- Need to profile floating point variables to analyze the ranges.
- Impact of discretization errors depend on their severity and on the algorithm.
E. Example of conversion errors

→ Average absolute error (%) between floating-point and fixed-point representations for 10 random real numbers between 0 and 128.

→ Average absolute error (%) between floating-point and fixed-point representations for 10 random real numbers between 0 and 128, and 10 random real numbers between 0 and 1.
IV. Behavioral synthesis

Given:

- a sequencing graph (data/control flow graph) that is constructed from the circuit behavioral circuit specification after code optimizations
- a set of functional resources (multipliers, adders, …, etc) each characterized in terms of area, delay and power
- a set of constraint (on circuit delay, area and power)

Synthesizing the output circuit consists of two stages:

1. Place operations in time (scheduling) and space (allocation them to resources)
2. Determine the detailed connection of the data path the control unit
Synthesis questions

- How many resources and what type of resources do we need? Min amount, max amount?
- When (in time) and where (which resource) should operations be scheduled?
- How long will the synthesized algorithm take to finish? How long to take under resource constraints?
Scheduling (temporal assignment)

- **Scheduling** is the task of determining the start times of all operations, subject to the precedence constraints specified by the sequencing graph.

- The *latency* of the sequencing graph is the difference between the start time of the sink and the start time of the source.
Scheduling to minimize the latency

Consider the following differential equation integrator

\[ y'' + 3xy' + 3y = 0 \]
\[ x(0) = x; \ y(0) = y; \ y'(0) = u \]

```plaintext
read (x, y, u, dx, a);
do {
    x1 = x + dx;
    u1 = u - (3*x*u*dx) - (3*y*dx);
    y1 = y + u*dx;
    c = x1 < a;
    x = x1; u = u; y = y1;
} while (c);
write(y);
```
ASAP scheduling for minimum latency

Assuming all operations to have 1 unit delay, what is the latency here?
ASAP scheduling algorithm

ASAP (G_s(V,E)) {
    Schedule v_0 by setting t_0 = 1;
    repeat {
        Select a vertex v_i whose predecessors are all scheduled;
        Schedule v_i by setting t_i = \max_{j:(v_j,v_i) \in E} t_j + d_j;
    }
    until (v_n is scheduled);
    return (t);
}

ALAP scheduling to meet latency constraint
ALAP scheduling algorithm

\[
\text{ALAP} \left( G_s(V,E), \lambda \right) \{
\text{Schedule } v_n \text{ by setting } t_n = \lambda + 1; \\
\text{repeat } \{
\text{Select a vertex } v_i \text{ whose successors are all scheduled; }
\text{Schedule } v_i \text{ by setting } t_i = \min_{j: (v_i,v_j) \in E} t_j - d_i;
\}
\text{until } (v_0 \text{ is scheduled}); \\
\text{return } (t);
\}
\]
Operation mobility

- The *mobility* of an operation corresponds to the difference of the start time computed between the ALAP and ASAP algorithms.
- Mobility measure the freedom we have in scheduling an operation to meet the timing schedule.
Resource allocation

- Bind a resource to two operations as long as they do not execute concurrently and can implemented by a resource of the same type

How many resources needed of ALU ( +, - and < ) and multiplier?
Can we do better? Can we get the same latency with less resources?

Resources sharing the same instance are colored with the same color. How many instances are now needed? How can we find the solution?
Finding the minimal number of resources for a given latency (T) using list scheduling

- Initialize all resource instances to 1.
- for $t = 1$ to $T$: For each resource type:
  - Calculate the slack (ALAP time – $t$) of each candidate operation
  - Schedule candidate operations with zero slack and update the number of resource instances used if needed
  - Schedule any candidate operations requiring no new resource instances

What is the intuition behind this heuristic?
Scheduling under resource constraint

Assume we just one instance of a multiplier and one instance of an ALU (+, - and ==), how can we schedule all operations? What is the latency?
Finding the minimal latency for a given resource constraint (C) using list scheduling

• Label all operations by the length of their longest path to the sink and rank them in decreasing order
• Repeat
  – For each resource type
    • Determine the candidate operations (U) that can be scheduled
    • Select a subset of U by priority such that the resource constraint usage (C) is not exceeded
  – Increment time

What is the intuition behind this heuristic?
There is an inherent tradeoff between area and latency.
Scheduling and allocation necessitates a control unit that orchestrates the sequencing of operations. How to design the control unit?
Control unit example

for(i=0; i<10; i=i+1)
begin
  x = a[i] + b[i];
  z = z + x;
end

How many control signals are produced from the control unit?

How can we design the control unit?
Embedding a digital circuit to FPGA fabric

- **Mapping** decomposes the circuit into logic sections and flip-flops such that each section fits into a K-LUT LE.
- **Packing** groups LEs into clusters so that each cluster fits into a LAB.
- **Placement** determines the position of each cluster into the LABs of the island style FPGA.
- **Routing** determines the exact routes for the communicating LE/LABs.

What are the objectives/metrics that these algorithms should pursue?
V. Mapping finds a covering for a given circuit using K-LUT

[Figure form Cong FPGA’ 01]

Map to a LUT in a LB
A covering example

[From Ling et al. DAC’ 05]

There could be many possible covering? Which one should be picked?
Framework for structural mapping

- **Cut generation:** Identify all the k-cuts for each node such that all nodes are covered.
- **Cut ranking:** Use dynamic programming to identify the best cuts based on area or timing.
- **Cut Selection:** Identify the optimal cuts from the results of dynamic programming.
- **Generation of final mapping solution:** Identify programming bits of the LUT corresponding to each cut.

[Example adapted from J. Zambreno]
Mapping: 1. Cut generation

Goal: generate all cuts with $k$ (input) $\leq 4$

Traversing in topological order from input to output:

$\text{cut}(a) = \{\{a\}\}$
$\text{cut}(b) = \{\{b\}\}$
$\text{cut}(c) = \{\{c\}\}$

$\text{cut}(d) = \{\{d\}, \{d, a\}\}$
$\text{cut}(e) = \{\{e\}, \{e, b\}\}$
$\text{cut}(f) = \{\{f\}, \{f, c\}\}$

$\text{cut}(g) = \{\{g\}, \{g, f\}, \{g, f, c\}\}$

$\text{cut}(h) = \{\{h\}, \{h, g\}, \{h, g, f\}, \{h, e\}, \{h, e, b\}, \{h, g, e\}\}$

$\text{cut}(i) = \{\{i\}, \{i, h\}, \{i, h, g\}, \{i, h, e\}, \{i, d\}, \{i, d, a\}, \{i, h, d\}\}$
Mapping: 2. Cut ranking using dynamic programming (area)

Area of a cut = 1 + sum of area of cut fan-ins

cut(a) = \{\{a\}=1\}
cut(b) = \{\{b\}=1\}
cut(c) = \{\{c\}=1\}

cut(d) = \{\{d\}=2, \{d, a\}=1\}
cut(e) = \{\{e\}=2, \{e, b\}=1\}
cut(f) = \{\{f\}=2, \{f, c\}=1\}

cut(g) = \{\{g\}=2, \{g, f\}=2, \{g, f, c\}=1\}

cut(h) = \{\{h\}=3, \{h, g\}=3, \{h, g, f\}=3, \{h, e\}=3, \{h, e, b\}=2, \{h, g, e\}=3\}

cut(i) = \{\{i\}=4, \{i, h\}=4, \{i, h, g\}=4, \{i, h, e\}=4, \{i, d\}=4, \{i, d, a\}=3, \{i, h, d\}=4\}
Working backwards from primary outputs to primary inputs:

best cut(i) = \{i, d, a\} = 3  
best cut(h) = \{h, e, b\} = 2  
best cut(g) = \{g, f, c\} = 1

We covered all 9 gates in the circuit with this solution.

What is the critical path delay in terms of LEs?
Mapping: 2. Cut ranking using dynamic programming (timing)

Time of a cut = 1 + max time of cut fan-ins

cut(a) = {{a}=1}
cut(b) = {{b}=1}
cut(c) = {{c}=1}

cut(d) = {{d}=2, {d, a}=1}
cut(e) = {{e}=2, {e, b}=1}
cut(f) = {{f}=2, {f, c}=1}

cut(g) = {{g}=2, {g, f}=2, {g, f, c}=1}

cut(h) = {{h}=2, {h, g}=2, {h, g, f}=2, {h, e}=2, {h, e, b}=2, {h, g, e}=2}

cut(i) = {{i}=3, {i, h}=2, {i, h, g}=2, {i, h, e}=2, {i, d}=3, {i, d, a}=3, {i, h, d}=2}
Mapping: 3. Cut selection (timing)

Working backwards from primary outputs to primary inputs:

best cut(i) = \{i, h, g\}=2
best cut(f) = \{f, c\}
best cut(e) = \{e, b\} = 1
best cut(d) = \{a, d\} = 1

We **covered** all 9 gates in the circuit with this solution

What is total area in terms of LEs?
Mapping: 4. final mapping

- For each selected cut:
  - simulate all possible $2^k$ input patterns
  - examine the output
  - assemble the bits for each LUT
Complications from fan outs

- Fan outs $\rightarrow$ logic duplication $\rightarrow$ complicates cut ranking and selection

\[
\begin{align*}
\text{cut}(a) &= \{a\} = 1 \\
\text{cut}(b) &= \{b\} = 2 \\
\text{cut}(c) &= \{c\} = 2
\end{align*}
\]

Total cost = 4 LEs, where in fact we only need 3 LEs $\rightarrow$ need to modify area computations; how?

How to cover with $K = 3$ LEs?
Packing

How can we decide which LEs should go together in the same logic cluster?

Possible method (VPACK): Construct each cluster sequentially

• Start by choosing seed LE for the cluster
• Then greedily selects the LE which shares the most inputs and outputs with the cluster being constructed
• Repeat the procedure until greedily until the cluster is full or the number of inputs exceed the limit I
• Can addition of a LE to a cluster reduces the number of distinct inputs?
VI. Placement

- Placement assigns an exact position or LAB for each cluster in the input netlist

- **Two possible approaches:**
  - Placers based on recursive min-cut algorithms
  - Placers based on simulated annealing

[figure from H. Zhou]
How to measure wire length for nets with fanout > 1?

1. Semi-perimeter method: Half the perimeter of the bounding rectangle that encloses all the pins of the net to be connected. Most widely used approximation!

2. Complete graph: Since #edges in a complete graph \(\frac{n(n-1)}{2}\)

3. Minimum chain: Start from one vertex and connect to the closest one, and then to the next closest, etc.

[slide adapted from H. Zhou]
Dealing with nets with fanout > 1

4. **Star model**: Connect one pin to all other pins of the net.

5. **Minimum spanning tree**: Connect all pins with a tree of a minimum length.

6. **Steiner-tree approximation**: Connect all pins (plus additional intermediate points) with a minimum tree. Best construction, but computationally expensive.

[slide adapted from H. Zhou]
I. Min-cut placement

Two main questions:

- \( Q_1 \): How to partition a hypergraph?
- \( Q_2 \): How to propagate net connectivity information from one block to another?
A_1. Kernighan-Lin method for finding a min-cut partition

- Start with all nodes free to swap (*unlocked*)
- Label each possible swap with immediate change in execution time that it causes (*gain*)
- Iteratively select and execute a swap with highest gain (whether positive or negative); *lock* the moving vertex (i.e., cannot move again during the pass),
- Best solution seen during the pass is adopted as starting solution for next pass
Partitioning subcircuits in isolation does not lead to a global optimal result

Case I: Blocks are partitioned in isolation → optimal local partitioning results but far from optimal global results

Case II: Information about cells in one block are accounted for in the other block → local partitioning results are translated to global wirelength results
A$_2$: Terminal propagation is a good technique to capture global connectivity.

- B$_1$ has been partitioned; B$_2$ is to be partitioned.
- u is propagated as a fixed vertex $u_f$ to the subblock that is closer.
- $u_f$ biases the partitioner to move v upward.
II. Iterative improvement algorithms

Suppose you start with a random placement, how can you improve it?

Possible algorithm:
- Pick a pair of cells and swap their locations if this leads to reduction in WL

What’s wrong with the previous greedy algorithm?
⇒ It can simply get stuck in a local optimal result
Simulated annealing allows us to avoid getting trapped in a local minima

**Modified algorithm**

- Generate a random move (say a swap of two cells)
  - calculate the chance in WL ($\Delta L$) due to the move
  - if the move results in reduction ($\Delta L < 0$) then accept
  - else reject with probability $1 - e^{-\Delta L / T}$

- $T$ (temperature) controls the rejection probability

- Initially, $T$ is high (thus avoiding getting trapped early in a local minima) then the temperature *cools down in a scheduled* manner; at the end, the rejection probability is 1

- With the right “slow-enough” cooling scheduling, simulated annealing is guaranteed to reach the global optimal
How do the cooling scheduling and corresponding cost functions look like?

[source: I. Markov]
Placement before & after simulated annealing

[using VPR tool]
VII. Routing

Assign exact routes for each wire in the given circuit in the FPGA fabric such that no two wires overlap

General idea:
• Order the wires according to some criteria (e.g., timing criticality)
• Sequentially route each wire using shortest path algorithms (after removing the resources consumed from preceding routed wires)
Problem: Find the shortest path for a 2-pin wire from s to t

- grid cell capacity is full
- grid cell still has available tracks

Speed ups are possible using A* search algorithms and other AI search techniques
Impact of Net Ordering

- A bad net ordering
  - may unnecessarily increase the total wirelength
  - or even yield the chip unroutable!

- Example: Two nets A and B

A first then B (Bad order)

B first then A (Good order)

- Length in placement
- Timing criticality
When a route for a net can’t be found then rip up and re-route

Cannot route C

So rip-up B and route C first.

Finally route B.

[Example from Prof. D. Pan Lecture]
VPR. After routing

After placement

After placement and routing

You probably saw similar layouts from the Quartus II tool.
VIII programming the FPGA

Configuration data in
Configuration data out

= I/O pin/pad
= SRAM cell