High Level Synthesis

Design Representation

- Intermediate representation essential for efficient processing.
  - Input HDL behavioral descriptions translated into some canonical intermediate representation.
    - Language independent
    - Uniform view across CAD tools and users
  - Synthesis tools carry out transformations of the intermediate representation.
Scope of High Level Synthesis

Verilog / VHDL Description

Control and Data Flow Graph (CDFG)

Transformation

Scheduling

Allocation

FSM Controller

DataPath Structure

Simple Transformation

A = B + C;
D = A * E;
X = D − A;

Stmt 1
Stmt 2
Stmt 3

Read B Read C

Read A

Read A

Read E

Read D

Read A

+ * −

Write A

Write D

Write X
Transformation with Control/Data Flow

```plaintext
 case (C)
  1: begin
      X = X + 3;
      A = X + 1;
    end
  2: A = X + 5;
    default: A = X + Y;
 endcase
```
X = X + 3;
A = X + 1;
A = X + 5;
A = X + Y;

Data flow graph can be drawn similarly, consisting of “Read” and “Write” boxes, operation nodes, and multiplexers.

Another Example

if (X == 0)
    A = B + C;
    D = B - C;
else
    D = D - 1;
Compiler Transformations

- Set of operations carried out on the intermediate representation.
  - Constant folding
  - Redundant operator elimination
  - Tree height transformation
  - Control flattening
  - Logic level transformation
  - Register-Transfer level transformation
Constant Folding

Constant 4

Constant 12

Constant 16

Write X

Write X

Redundant Operator Elimination

\[ C = A \times B; \]
\[ D = A \times B; \]

Read A

Read B

Read A

Read B

Read A

Read B

Read A

Read B

Write C

Write D

Write C

Write D
Tree Height Transformation

\[ a = b - c + d - e + f + g \]

Control Flattening
Logic Level Transformation

\[ C = A + A'B = A + B \]

High Level Synthesis

PARTITIONING
Why Required?

- Used in various steps of high level synthesis:
  - Scheduling
  - Allocation
  - Unit selection
- The same techniques for partitioning are also used in physical design automation tools.
  - To be discussed later.

Component Partitioning

- Given a netlist, create a partition which satisfies some objective function.
  - Clusters almost of equal sizes.
  - Minimum interconnection strength between clusters.
- An example to illustrate the concept.
Behavioral Partitioning

- With respect to Verilog, can be used when:
  - Multiple modules are instantiated in a top-level module description.
    - Each module becomes a partition.
  - Several concurrent “always” blocks are used.
    - Each “always” block becomes a partition.
Partitioning Techniques

• Broadly two classes of algorithms:
  1. **Constructive**
     • Random selection
     • Cluster growth
     • Hierarchical clustering
  2. **Iterative-improvement**
     • Min-cut
     • Simulated annealing

Random Selection

• Randomly select nodes one at a time and place them into clusters of fixed size, until the proper size is reached.
• Repeat above procedure until all the nodes have been placed.
• Quality/Performance:
  – Fast and easy to implement.
  – Generally produces poor results.
  – Usually used to generate the initial partitions for iterative placement algorithms.
Cluster Growth

\[ m \text{ : size of each cluster, } V \text{ : set of nodes} \]

\[ n = \frac{|V|}{m} \]

for (\(i=1; i<=n; i++\))

\{ \]

\hspace{1cm} seed = vertex in \( V \) with maximum degree;

\hspace{1cm} \( V_i = \{\text{seed}\} \);

\hspace{1cm} \( V = V - \{\text{seed}\} \);

\hspace{1cm} for (\(j=1; j<m; j++\))

\hspace{2cm} \{ \]

\hspace{3cm} t = vertex in \( V \) maximally connected to \( V_i \);

\hspace{3cm} \( V_i = V_i \cup \{t\} \);

\hspace{3cm} \( V = V - \{t\} \);

\hspace{2cm} \} \]

\}

Hierarchical Clustering

- Consider a set of objects and group them depending on some measure of closeness.
  - The two closest objects are clustered first, and considered to be a single object for further partitioning.
  - The process continues by grouping two individual objects, or an object or cluster with another cluster.
  - We stop when a single cluster is generated and a hierarchical cluster tree has been formed.
    - The tree can be cut in any way to get clusters.
Example

\[ \begin{align*}
  v_1 & \quad 7 & v_1 & \quad 7 \\
  v_2 & \quad 1 & v_3 & \quad 5 \\
  v_4 & \quad 4 & v_5 & \quad 1 \\
  v_5 & \quad 9 & v_3 & \quad 4 \\
  v_1 & \quad 4 & v_2 & \quad 1 \\
  v_3 & \quad 4 & v_5 & \quad 6 \\
  v_5 & \quad 4 & v_5 & \quad 4
\end{align*} \]
Min-Cut Algorithm (Kernighan-Lin)

• Basically a bisection algorithm.
  – The input graph is partitioned into two subsets of equal sizes.
• Till the cutsets keep improving:
  – Vertex pairs which give the largest decrease in cutsize are exchanged.
  – These vertices are then locked.
  – If no improvement is possible and some vertices are still unlocked, the vertices which give the smallest increase are exchanged.

Example

Initial Solution

Final Solution
Steps of Execution

Choose 5 and 3 for exchange

• Drawbacks of K-L Algorithm
  – It is not applicable for hyper-graphs.
    • It considers edges instead of hyper-edges.
    • It cannot handle arbitrarily weighted graphs.
    • Partition sizes have to be specified a priori.
  – Time complexity is high.
    • $O(n^3)$.
  – It considers balanced partitions only.
Goldberg-Burstein Algorithm

- Performance of K-L algorithm depends on the ratio R of edges to vertices.
- K-L algorithm yields good bisections if R > 5.
- For typical VLSI problems, 1.8 < R < 2.5.
- The basic improvement attempted is to increase R.
  - Find a matching M in graph G.
  - Each edge in the matching is contracted to increase the density of the graph.
  - Any bisection algorithm is applied to the modified graph.
  - Edges are uncontracted within each partition.

Example of G-B Algorithm

Matching of Graph

After Contracting
Simulated Annealing

- Iterative improvement algorithm.
  - Simulates the annealing process in metals.
  - Parameters:
    - Solution representation
    - Cost function
    - Moves
    - Termination condition
    - Randomized algorithm
  - To be discussed later.

High Level Synthesis

SCHEDULING
What is Scheduling?

- Task of assigning behavioral operators to control steps.
  - Input:
    - Control and Data Flow Graph (CDFG)
  - Output:
    - Temporal ordering of individual operations (FSM states)

- Basic Objective:
  - Obtain the fastest design within constraints (exploit parallelism).

Example

- Solving 2nd order differential equations (HAL)

```verbatim
module HAL (x, dx, u, a, clock, y);
input x, dx, u, a, clock; output y;
always @(posedge clock)
  while (x < a)
    begin
      x1 = x + dx;
      u1 = u – (3 * x * u * dx) – (3 * y * dx);
      y1 = y + (u * dx);
      x = x1;
      u = u1;
      y = y1;
    end
endmodule
```
Scheduling Algorithms

- Three popular algorithms:
  - As Soon As Possible (ASAP)
  - As Late As Possible (ALAP)
  - Resource Constrained (List scheduling)
As Soon As Possible (ASAP)

- Generated from the DFG by a breadth-first search from the data sources to the sinks.
  - Starts with the highest nodes (that have no parents) in the DFG, and assigns time steps in increasing order as it proceeds downwards.
  - Follows the simple rule that a successor node can execute only after its parent has executed.
- Fastest schedule possible
  - Requires least number of control steps.
  - Does not consider resource constraints.

ASAP Schedule for HAL
As Late As Possible (ALAP)

- Works very similar to the ALAP algorithm, except that it starts at the bottom of the DFG and proceeds upwards.
- Usually gives a bad solution:
  - Slowest possible schedule (takes the maximum number of control steps).
  - Also does not necessarily reduce the number of functional units needed.
Resource Constrained Scheduling

- There is a constraint on the number of resources that can be used.
  - List-Based Scheduling
    - One of the most popular methods.
    - Generalization of ASAP scheduling, since it produces the same result in absence of resource constraints.
  - Basic idea of List-Based Scheduling:
    - Maintains a priority list of “ready” nodes.
    - During each iteration, we try to use up all resources in that state by scheduling operations in the head of the list.
    - For conflicts, the operator with higher priority will be scheduled first.

For operator node i,
Mobility of i
\[ <i> = \text{Time for ALAP} - \text{Time for ASAP} \]
• Priority List:
  *: 1 <0>
  2 <0>
  3 <1>
  4 <2>
  +: 10 <2>

• Resources:
  *: 2
  +: 1
  -: 1
  <: 1

HAL List Schedule
2 multipliers, 1 adder, 1 subtractor, 1 comparator
Time Constrained Scheduling

- Given the number of time steps, try to minimize the resources required.
  1. Force Directed Scheduling (FDS)
  2. Integer Linear Programming (ILP)
  3. Iterative Refinement

Force Directed Scheduling

[Ref: paper by Paulin & Knight]

- Goal is to reduce hardware by balancing concurrency
- Iterative algorithm, one operation scheduled per iteration
- Information (i.e. speed & area) fed back into scheduler
The Force Directed Scheduling Algorithm

Input: DFG $G = (N, E)$, Iteration Period $= T$.
Output: Final FDS Schedule.

1. While (Unscheduled nodes exist) {
   1.1 Compute the time frames for each node;
   1.2 Build the distribution graph;
   1.3 Compute the self-forces;
   1.4 Compute the predecessor and successor forces;
   1.5 Schedule the node into the time step that minimizes the total force;
}

Step 1

- Determine ASAP and ALAP schedules
Step 2

- **Determine Time Frame of each op**
  - Length of box ~ Possible execution cycles
  - Width of box ~ Probability of assignment
  - Uniform distribution, Area assigned = 1

![Diagram showing time steps and probability distribution]

Step 3

- **Create Distribution Graphs**
  - Sum of probabilities of each Op type for each c-step of the CDFG
  - Indicates concurrency of similar Ops
  \[ DG(i) = \sum \text{Prob(Op, i)} \]

![Distribution Graphs for Multiply and Add operations]
Conditional Statements

- Operations in different branches are mutually exclusive
- Operations of same type can be overlapped onto DG
- Probability of most likely operation is added to DG

![Diagram of scheduling operations]

Self Forces

- Scheduling an operation will affect overall concurrency
- Every operation has 'self force' for every C-step of its time frame
- Analogous to the effect of a spring: $F = Kx$ (Hooke’s law)
  \[ \text{Force}(i) = \text{DG}(i) \cdot x(i) \]
  \[ \text{DG}(i) \sim \text{Current Distribution Graph value} \]
  \[ x(i) \sim \text{Change in operation’s probability} \]

\[ \text{Total self force associated with the assignment of an operation to C-step } j \ (t \leq j \leq b) \]

\[ \text{Self Force}(j) = \sum_{i=1}^{b} \cdot [\text{Force}(i)] \]

- Desirable scheduling will have negative self force
  - Will achieve better concurrency (lower potential energy)
Example

- Attempt to schedule * (operation 4) in C-step 1

Self Force(1) = Force(1) + Force(2)
= (DG(1) * X(1)) + (DG(2) * X(2))
= [2.833*(0.5) + 2.333*(-0.5)] = +0.25

- This is positive, scheduling the multiply in the first C-step would be bad

Diff Eq Example: Self Force for Node 4

\[
Self_{\text{Force}}_4(1) = Force_4(1) + Force_4(2)
= (DG_M(1) * x_4(1)) + (DG_M(2) * x_4(2))
= (2.833 * (1 - 0.5)) + (2.333 * (0 - 0.5))
= (2.833 * (+0.5)) + (2.333 * (-0.5))
= +0.25
\]

\[
Self_{\text{Force}}_4(2) = Force_4(1) + Force_4(2)
= (DG_M(1) * x_4(1)) + (DG_M(2) * x_4(2))
= (2.833 * (-0.5)) + (2.333 * (+0.5))
= -0.25
\]
Predecessor & Successor Forces

- Scheduling an operation may affect the time frames of other linked operations
- This may negate the benefits of the desired assignment
- Predecessor/Successor Forces = Sum of Self Forces of any implicitly scheduled operations

Example: Successor Force on Node 4

- If node 4 scheduled in step 1
  - no effect on time frame for successor node 8
- Total force = Force4(1) = +0.25
- If node 4 scheduled in step 2
  - causes node 8 to be scheduled into step 3
  - must calculate successor force

\[
\text{Succ. Force}_4(2) = \text{Self. Force}_4(2) + \text{Self. Force}_3(3)\\
= (DG_M(2) \times x_4(2)) + (DG_M(3) \times x_8(3))\\
= (2.333 \times (0 - 0.5)) + (0.833 \times (1 - 0.5))\\
= (2.333 \times (-0.5)) + (0.833 \times (+0.5))\\
= -0.75
\]

\[
\text{Force}_4(2) = \text{Self. Force}_4(2) + \text{Succ. Force}_4(2)\\
= -0.25 - 0.75 = -1.00
\]
Final Time Frame and Schedule

Diff Eq Example: Final DG
Lookahead

- Temporarily modify the constant DG(i) to include the effect of the iteration being considered

\[
\text{Force (i)} = \text{temp\_DG(i)} \times x(i)
\]
\[
\text{temp\_DG(i)} = \text{DG(i)} + \frac{x(i)}{3}
\]

- Consider previous example:

\[
\text{Self Force(1)} = (\text{DG(1)} + \frac{x(1)}{3})x(1) + (\text{DG(2)} + \frac{x(2)}{3})x(2)
\]
\[
= 0.5(2.833 + 0.5/3) - 0.5(2.333 - 0.5/3)
\]
\[
= +0.41667
\]

- This is even worse than before

Minimization of Bus Costs

- Basic algorithm suitable for narrow class of problems
- Algorithm can be refined to consider “cost” factors
- Number of buses ~ number of concurrent data transfers
- Number of buses = maximum transfers in any C-step
- Create modified DG to include transfers: Transfer DG

\[
\text{Trans DG(i)} = \sum [\text{Prob (op,i)} \times \text{Opn\_No\_InOuts}]
\]
\[
\text{Opn\_No\_InOuts} \sim \text{combined distinct in/outputs for Op}
\]

- Calculate Force with this DG and add to Self Force
Minimization of Register Costs

• Minimum registers required is given by the largest number of data arcs crossing a C-step boundary
• Create *Storage Operations*, at output of any operation that transfers a value to a destination in a later C-step
• Generate *Storage DG* for these “operations”
• Length of storage operation depends on final schedule

![Diagram showing ASAP, MAX, and ALAP lifetimes]

---

Minimization of Register Costs (contd.)

• \( \text{avg life} = \frac{[\text{ASAP life}] + [\text{ALAP life}] + [\text{MAX life}]}{3} \)
• \( \text{storage DG}(i) = \begin{cases} \frac{[\text{avg life}]}{[\text{max life}]} & \text{(no overlap between ASAP & ALAP)} \\ \frac{[\text{avg life}] - [\text{overlap}]}{[\text{max life}] - [\text{overlap}]} & \text{(if overlap)} \end{cases} \)

• Calculate and add “Storage” Force to Self Force

![Diagram showing storage distribution and force directed graphs]
Pipelining

- **Functional Pipelining**
  - Pipelining across multiple operations
  - Must balance distribution across groups of concurrent C-steps
  - Cut DG horizontally and superimpose
  - Finally perform regular Force Directed Scheduling

- **Structural Pipelining**
  - Pipelining within an operation
  - For non data-dependant operations, only the first C-step need be considered

Other Optimizations

- **Local timing constraints**
  - Insert dummy timing operations -> Restricted time frames

- **Multiclass FU’s**
  - Create multiclass DG by summing probabilities of relevant ops

- **Multistep/Chained operations.**
  - Carry propagation delay information with operation
  - Extend time frames into other C-steps as required

- **Hardware constraints**
  - Use Force as priority function in list scheduling algorithms
Scheduling Under Realistic Constraints

• Functional units can have varying delays.
  – Several approaches:
    • Unit-delay model
    • Multicycle model
    • Chaining model
    • Pipelining model
High Level Synthesis

ALLOCATION and BINDING
Basic Idea

- Selection of components to be used in the register transfer level design.
- Binding of hardware structures to behavioral operators and variables.
  - Register
  - ALU
  - Interconnection (MUX)

Example

\[
\begin{align*}
\text{e} &= a + b; \\
\text{g} &= a + e; \\
\text{f} &= c + d; \\
\text{h} &= f + d;
\end{align*}
\]
An Integrated Approach

- From the paper by “C-J. Tseng and D.P. Sieworek”