1 Introduction
2 Layout Languages
3 Random Logic Module Generation
4 Regular Module Generation
5 Silicon Compilation
6 Future Trends
### Introduction

- Design objects on different domain levels (Fig. 7.1).

<table>
<thead>
<tr>
<th>Level</th>
<th>Domain</th>
<th>Behavioral Domain</th>
<th>Structural Domain</th>
<th>Physical Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Level</td>
<td>Performance, Specification</td>
<td>Processor Mem. Sw., Controllers Buses</td>
<td>Physical partitions</td>
<td></td>
</tr>
<tr>
<td>Algorithm Level</td>
<td>Algorithms (manipulation of data structures)</td>
<td>H/W modules, Data structures</td>
<td>Clusters</td>
<td></td>
</tr>
<tr>
<td>µarchitecture Level</td>
<td>Operations, Register transfers, State sequencing</td>
<td>ALU, MUX, Registers, Micro sequencer, Micro store</td>
<td>Floorplans</td>
<td></td>
</tr>
<tr>
<td>Logic Level</td>
<td>Boolean equation, FSM</td>
<td>Gates, F/Fs, Cells</td>
<td>Cells, Module Plans</td>
<td></td>
</tr>
<tr>
<td>Circuit Level</td>
<td>Transfer function, Timing</td>
<td>Transistors, Wires, Contacts</td>
<td>Layout</td>
<td></td>
</tr>
</tbody>
</table>
**Introduction**

- **Silicon Compilers**: Programs that translate a component description into:
  - a layout description *(geometric level)*;
  - a simplified structural description for simulation *(simulation model)*;
  - and a timing analysis description *(timing model)*.

- **Ideal Silicon Compiler** (Fig. 7.2 b) consists of:
  1. *synthesis*: translation from behavioral to structural representation.
**Introduction**

- **Present Silicon Compiler Methodology** (Fig. 7.3) has:

  1. **Behavioral Silicon Compiler** (for synthesis) supported by by a module compiler (which is in turn supported by a cell compiler) with a Behavior Representation as I/P. 5 tasks must be performed.

  2. **Structural Silicon Compiler** (for layout) with a Structural Representation as I/P. only 4 tasks (2-5) are performed.
Introduction

- **Tasks needed** for silicon compiler are:

  (1) behavioral description is *synthesized* into a structural description of well-defined components.

  (2) layout of each component (structural) is *instantiated* by a *cell or module compiler*.

  (3) compiled *layouts are placed and routed*.

  (4) *packing is selected*.

  (5) *test vector set is generated*.

Behavioral Silicon Compiler

= synthesizer (1) + structural Silicon Compiler(2, 3, 4, 5).
Layout Languages

In general, IC languages are based on three types of layout architectures:

- **Fixed Grid Architecture** (Fig. 7.4)
  - layout area is divided into a uniformly spaced grid.
  - graphical language is used.

- **Virtual Grid Architecture** (Fig. 7.5)
  - transistors, contacts and wires are placed on a grid for easy design capture and simplified interface to other tools.
  - languages has 5 parts: *size*, *device*, *wire*, *contact*, and *pin*. 
Layout Languages

- **Relative Grid Architecture** (Fig. 7.6)

  - grid is used to indicate relative placement of transistors, contacts and wires, and to determine electrical connectivity of the circuit.

  - layout language must:

    (a) make the layout most more like programming than editing.
    (b) eliminate the need for DRC after the layout is generated.
    (c) permit the creation of easy to use cell libraries.
    (d) allow a hierarchical description in which most of the details at one level is truly hidden from all higher levels.
Random Logic Module Generation

- Because of the irregular structure of basic components, it is difficult to create a library for random logic module.

- **Random logic Module Generation** can be classified according to:
  
  1. architecture styles (constrained, flexible);
  
  2. dimensionality (1-D, 2-D);
  
  3. type of input descriptions (transistor or logic schematics, boolean expression).
Random Logic Module Generation

§ Constrained Linear Structure (1-D array)

- **Weinberger Array** [Wei 67]
  - (Fig. 7.8) Weinberger Array layout.
  - (Fig. 7.9) Row Assignment Using LEA.
  - (Fig. 7.10) Folding results (letting 2 gates share a column).
  e.g., nets 1, 2, 5 used by A is Shared with nets 5, 6, 7 by C.
Single-Strip Static CMOS Layout

- Optimized for combinational strips in static CMOS

VDD

N-Well (pMOS)

GND

P-Well (nMOS)

Polysilicon

$V_{DD}$

1 2 3 4 5

$pMOS$

$nMOS$

GND
Single-Strip Static CMOS

CMOS Standard-Cell Layout

\[ X + YZ \]
Single-Strip Static CMOS

- Dual graphs for P and N sides
Optimization of Static Strip Layout

Uehara & Van Cleemput, 1981

- Every gate-drain & gate-source potential is represented by a vertex.
- Drains or sources at the same potential (connected) are represented by the same vertex.
- Every transistor is represented by an edge connecting the drain and source vertices of that transistor.
- “If two edges $x$ and $y$ in the graph are adjacent, then it is possible to place the corresponding transistors in physically adjacent positions of the same row, and hence connect them by a diffusion area. In order to minimize the number of separation areas, it is necessary to find a set of paths, of minimum size, which corresponds to chains of transistors in the row.”
Euler Path

- Is a path with no repeating edges that contains all the edges of the graph.
- If such a path does not exist, the graph can be partitioned into subgraphs with Euler paths.
- Must find paths in the dual graphs with the same sequence of edges.
Uehara and VanCleemput (1981)

- Enumerate all possible decompositions of the graph and find the minimum number of paths that cover the graph.
- Chain the gates by means of shared diffusion areas according to the order of the edges in each Euler path.
- If more than two Euler paths are needed to cover the graph model, provide separation between chains.
Heuristic Algorithm

- Relies on the fact that if the number of inputs to every AND/OR element is odd, then the corresponding graph has a single Euler path and there exists a graph such that the sequence of edges on the Euler path corresponds to the order of the inputs on a planar representation of the logic diagram.
- Convert every even number of parallel (series) edges into an odd one by adding pseudo pins.
- Minimize the interlace of pseudo pins and real pins.
- Find the Euler path (input sequence) and lay out the circuit.
- Delete pseudo pins and replace with separation areas.
Single-Strip Static CMOS

N-side Euler path: [1,3,2,4,5]

Dual Euler path: [2,3,1,4,5]
Random Logic Module Generation

- **Gate Matrix Layout** Lopez and Law [Lop 82] (fig. 7.11)
  - Uehara et al. [Uhe 81] use Euler path to minimize the number of rows used.
  - Omar Wing [Win 82, 85] solves the problem using graph-theoretic technique, "Improved GM Layout" [ICCAD 86].

  => Given an *Interval graph*: \(I(L)=(V,E)\) with *clique*, a fully connected (complete) subgraph of \(I(L)\). The **Gate Matrix Layout** problem is transformed into "finding an interval graph in which the size of the largest clique is minimized."
Gate Matrix Layout

- nMOS & pMOS implemented separately
Gate Matrix Layout

Example Circuit

Gate Matrix Layout
Gate Matrix Layout

- Abstract representation, the connection graph and the interval graph
Random Logic Module Generation

- Ohtsuki [Oht 79] uses column ordering + net assignment with LEA (Fig. 7.12).

- (Fig. 7.13) Realizable layout.

- Hwang et al. [Hwa 86b]: dynamical-net (partition-based).

- Leong [Leo 86]: Simulated Annealing.

- Nakatani [Nak 86] solves pMOS + nMOS simultaneously.
Random Logic Module Generation

§ **Constrained Quadratic Architecture** (2-D)

- **Standard Cell Architecture** (Fig. 7.14)
  - Layout = Placement + Routing
  - module generation starts by selecting from the cell library the cells that correspond to units in the input schematics.

- **Programmable Logic Array** (PLA) (fig. 7.15)
  - maps a set of boolean function in canonical, two-level sum of product form into a geometrical structure.
  - consists of an AND-plane and an OR-plane.
  - logic minimization can reduce the number of rows (Fig. 7.17).
AND-OR and NOR-NOR PLAs
Programmable Logic Array

Symbolic Layout
Random Logic Module Generation

- **Storage/Logic Array** (SLA) [K.F. Smith et al. 82], [C.K. Leung 84]
  - an extension of the PLA concept (Fig. 7.18).
Random Logic Module Generation

§ Flexible Architectures.

- **TOPOLOGIZER** (Fig. 7.19) [Kol 85] and LES [Lin 87]
  - rule-based CMOS layout generator.
  - layout style: all p-type transistor // Vdd, n-type // Vss.
  - input a transistor net-list and an environment constraints (layout size, pin constraints.)
  - output a symbolic file of CMOS layout.
  - placement expert is used to simplify the routing tasks by reducing number of wires needed to connect transistors (Fig. 7.20).
Random Logic Module Generation

- the number of wires is reduced by applying two strategies:
  
  (1) Placing connected transistors next to each other in the same row so that the connection can be realized through the diffusion layer.
  
  (2) Placing transistors with the same gate connection in the same column so that the gates can be connected through the poly layer.

- placement is improved by:
  
  (1) moving transistor from one location to another;
  
  (2) exchanging two transistors;
  
  (3) rotating a transistor (i.e., exchanging its drain and source).
Random Logic Module Generation

- "rough routing" by assigning a unique track to each pair of terminals to be connected (Fig. 7.21).

- refinement router applies rules to improve the rough routing (Fig. 7.23).

- Examples of improvement rules: (Fig. 7.22).

  1) U-turn elimination.

  2) Row sharing.
Random Logic Module Generation

- LES (Fig. 7.24)
  - rule based CMOS cell compiler.
  - input: a set of boundary constraints (height/width or aspect ratio, i/o pin positions) and a list of components (net-list).
  - output: symbolic layout file.
  - instead of routing channels, LES makes connections among leaf cells with through-the-cell routing.
  - the cell library is replaced by FLEX (a Flexible Leaf Cell Layout Expert.)
Random Logic Module Generation

- LES has seven Experts:
  
  (1) Analysis Expert (AE)
  
  (2) Architecture Expert (ARE)
  
  (3) Placement Expert (PE)
  
  (4) Characterization Expert (CE)
  
  (5) Flexible Layout Expert (FLEX)
  
  (6) Evaluation Expert (EE)
  
  (7) Optimization Expert (OE)
Regular Module Generation

- **module**: a micro-architectural entity that performs specific functions and consists of arrays of cells or tiles of a specific type.

  - each module is defined by a template (Fig. 7.26 a) and a set of cells that populate the template (Fig. 7.26 b).

- **module compiler**: describes the template, tiles, interfaces, template personalization, and modules to be used by other tools.

  - A template is described by a procedural language or an embedded language (e.g., menus or graphics).

  - the interface between cells are usually by abutment, where the designer specifies the bounding box and assignment for each cell.

  - template personalization is achieved by a personalization matrix and defining a function that maps the matrix's symbols into a set of tiles.
Regular Module Generation

- **APSS** [Stebnisky et al. 1982]
  
  - an automatic, technology-independent PLA generator.
  
  - input: Boolean exp.
  
  - output: PLA layout.
  
  - include logic translation, Boolean minimization, PLA folding, PLA topology generator and interfacing with placement and routing tools.

- **Multiplier Generator** [Chu and Sharmon 1984]
  
  - implement a modified Booth's algorithm.
**Regular Module Generation**

- **Generator-Generators**
  - A module generator-generator allows designers to create their own module generator.
  - It allows a module generator to be tailored to the specific design environment instead of adjusting the environment to fit available generators.
Regular Module Generation

- **Structure Compiler** [Low and Mosby 1985], [Bamji 1985]
  - allows designers to build module generator without modifying the design program.
  - the designers must supply the compiler two types of information:

(1) a parameterized block diagram called the array-structure template (which carries the floorplan information.)

(2) a library of tiles or master cells to be used.
Regular Module Generation

- **Mocha Chip** [Mayo R.N. 1986]
  - use an assembly diagrams to specify the structure of the module generator to be produced.
  - the diagram is drawn with a graphic editor.
  - Assembly diagrams show the relative placement and orientation of the module subcells.
Silicon Compilation

- A VLSI chip can be described by a structure of components. A **structure silicon compiler** calls cell or module generator, and then places, routes, and compacts the component layouts. (Fig. 7.27)

- The structural description can be synthesized with a **behavioral silicon compiler**.

- **Behavioral silicon compiler** translates behavioral into structural description (= synthesizer + structural silicon compiler).

- An intelligent silicon compiler is a behavioral silicon compiler that selects different design styles, components or optimization goals in order to satisfy the design constraints.
Silicon Compilation

§ Structural Silicon Compiler

- Design issues of a structural silicon compiler are:
  
  (1) the designer and technology interface. e.g., block-diagram, schematics, procedural language, etc.

  (2) controllability of the design process (level of human interaction).

  (3) level of integration (level of coupling all tools).

  (4) richness of the component generator set (availability of cell and module compilers).
Silicon Compilation

(5) performance expressed by quality measures:
- transistor density [mils/transistor] mils = \( \frac{1}{1000 \text{ inches}^2} \);
- compiler performance [transistor/hr];
- design time [transistor/person-hr].
Silicon Compilation

§ Behavioral Silicon Compiler

• Behavioral silicon compiler binds language constructs to structural components in the design model.

• Most of the behavioral silicon compiler assume a well-defined design model or target architecture (Fig. 7.29).

• Every variable must be allocated to a storage element ($F1: \text{variable} \rightarrow \text{memory}$) and every operator to a functional module ($F2: \text{operator} \rightarrow \text{functional unit}$).

  - Transformation is achieved by converting the language description into a control/data flow graph, in which all unnecessary data and/or control dependencies have been removed.
Silicon Compilation

**Scheduling** is applied to partition the graph into states.

- **Scheduling** can be performed using as-soon-as-possible (ASAP) or as-late-as-possible (ALAP) strategies (Fig. 7.30 a,b,c).

\[
x = \max(|a|, |b|)
\]

\[
x = \min(|a|, |b|)
\]

\[
result = \max(0.875x + 0.5y, x)
\]

(a) Program
Silicon Compilation

- There are two types of scheduling algorithms:

  (1) minimize the cost of the given design.

  (2) minimize time (number of control states) given the design cost (Fig. 7.30 d).

- **F1: variable —> memory**
  - Variable are merged into registers by partitioning the connectivity graph into maximal clique (Fig. 7.31).
  - Registers can be merged into register files (bank) if their read/write times do not overlap (Fig. 7.32).
Fig 7.31 Register sharing through variable merging.
Fig 7.31 Register sharing through variable merging.
Option 1: 
\[ R_1 = [\text{max}, \text{result}] \]
\[ R_2 = [\text{min}, \text{max}/8, 0.875\text{max}, \text{sum}] \]
\[ R_3 = [\text{min}/2] \]

Option 2: 
\[ R_1 = [\text{max}, \text{result}] \]
\[ R_2 = [\text{min}, \text{max}/8, 0.875\text{max}] \]
\[ R_3 = [\text{min}/2, \text{sum}] \]

Option 3: 
\[ R_1 = [\text{max}, \text{result}] \]
\[ R_2 = [\text{max}/8, 0.875\text{max}, \text{sum}] \]
\[ R_3 = [\text{min}, \text{min}/2] \]

Option 4: 
\[ R_1 = [\text{max}, \text{result}] \]
\[ R_2 = [\text{max}/8, 0.875\text{max}] \]
\[ R_3 = [\text{min}, \text{min}/2, \text{sum}] \]

Fig 7.31 Register sharing through variable merging.
### a) Lifetime Table

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>max</td>
<td>min</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>max</td>
<td>max/8</td>
<td>min/2</td>
</tr>
<tr>
<td>2</td>
<td>max</td>
<td>.875 max</td>
<td>min/2</td>
</tr>
<tr>
<td>3</td>
<td>max</td>
<td>sum</td>
<td>R</td>
</tr>
<tr>
<td>4</td>
<td>result</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

### Combine

#### Figure 7.32 Register Merging
Silicon Compilation

- **F2: operator —> functional unit**
  - After operations assigned to states and variables assigned to registers, functional units can be created by merging operators that are not used in the same state (Fig. 7.33).
(a) Dataflow Graph with State and Register Assignment
Figure 7.33 Unit Merging
Silicon Compilation

- **Bus**

  Connections between registers and units can be achieved from the data flow graph. These connections can be merged into buses by combining connections used in different states (Fig. 7.33 c, 7.34 a,b).
### (a) Lifetime Table

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L</td>
<td></td>
<td>L</td>
<td>L</td>
<td></td>
<td>L</td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>2</td>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
<td>L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
</tr>
</tbody>
</table>

**Buses = \{A, B, E\} \{C, D\} \{F\} \{G, H\}**

### (b) Implementation

**Figure 7.34 Connection Merging**
Silicon Compilation

- **Examples** of Behavioral Silicon Compiler
  - MAHA, Pangrle, ICCAD 1986.
  - EMUCS, Thomas et al., Computer, Dec. 1983.

- **Design Process Model**
  - The design process can be defined as a sequence of *bindings* with their associated algorithmic and optimization goals.
Silicon Compilation

- **Approaches to bindings:**
  
  (1) constructive method (creates an initial design).
  
  (2) iterative method (modifies the design to satisfy constraints).

- Most behavioral silicon compilers use the constructive method or divide binding responsibilities between the designer and the compiler.

  e.g., some silicon compiler systems specify the machine states in the language description. This eliminates the need for compiler variable-to-register binding (it needs only state-binding.)
What is High-Level Synthesis?

Behavioral Description or Specification

High-Level Design or Synthesis

Implementation Model

VHDL Behavioral Description

Von Neumann-style Control & Datapath

Signal Flow Graph

Systolic Array
High-Level Synthesis

How does one measure the quality of a solution?

→ Number of control steps (clock cycles in synchronous single-clock implementations)
  • Chaining and overlapped operations.
→ Number of hardware functional units (e.g. ALU, adder, multipliers)
  • Combinational or pipelined
→ Number of registers
→ Number of MUX inputs
→ Total connections and "unique connections."
Conventional Steps in High-Level Synthesis

- **Scheduling**: Assignment of operations to control steps

- **Allocation**: Determining the number and types of operators needed to implement behavior

- **Mapping**: Assigning operations to specific operators

  - **Type selection**: which type of operator should be used? (e.g. add on ALU or adder)
  - **Instance mapping**: actual mapping to specific hardware module
Components of a Conventional High-Level Synthesis Target

- **Data-part Components:**
  - Functional Units (operators): ALU, mult., adders, etc.
  - Registers, register files, or memory
  - Interconnection hardware: Buses, MUXs, nets

- **Controller:**
  - Symbolic STG for FSM (e.g. PLA-based)
  - Microcode for microprogrammed controller

- **Connected by:**
  - Control nets (from control to data part): e.g. enable/address storage, MUX select, bus tri-state control
  - Condition nets (from data part to control): condition outputs like result of branch-test, overflow
Parallelism, Pipelining, and Graph Folding

- Differential Equation Example: \( y'' + 5zy' + 3y = 0 \) (Paulin, et. al. 1986)

```c
lv = '1';
while (lv = '1') loop
    zn <= z + dz;
    un <= u - 5\cdot z\cdot u\cdot dz - 3\cdot y\cdot dz;
    yn <= y + u\cdot dz;
    lv <= (zn < a);
    z <= zn; u <= un; y <= yn;
end loop;
```
Maximally Parallel and Maximally Pipelined

lv = '1';
while (lv = '1') loop
zn <= z + dz;
un <= u - 5\cdot z\cdot u\cdot dz - 3\cdot y\cdot dz;
yn <= y + u\cdot dz;
lv <= (zn < a);
z <= zn; u <= un; y <= yn;
end loop;

"Control States" (time)
Maximally Parallel, Minimum Control States

\[
\begin{array}{cc}
3 & y \\
y & dz \\
dz & u \\
u & 5 \\
5 & z \\
z & a \\
a & L \\
\end{array}
\]

\[
\begin{array}{ccc}
X & \pm & L \\
3 & 1 & \\
2 & 1 & 1 \\
1 & & \\
1 & & \\
\end{array}
\]

\[
\begin{array}{ccc}
yn & un & zn \\
un & zn & lv \\
\end{array}
\]
Widthwise Folding via Symbolic Dependencies

Symbolic Dependency (control)

\[ x \pm L \]

\[ 2 \quad 1 \]

\[ 2 \quad 1 \quad 1 \]

\[ 1 \quad 1 \]

\[ 1 \]
Parallelism, Pipelining, and Graph Folding

- **Parallelism**: Multiple functional units at the same level in the graph active at the same time.

- **Pipelining**: Functional units on multiple levels of the graph being active at the same time.

- **Widthwise Graph Folding**: restricts parallelism with the addition of symbolic dependencies.
  - **Limiting case**: Multi-stage pipelined processor with feedback and memories between the stages.

- **Lengthwise Graph Folding**: decreases the number of levels in the graph with the addition of symbolic multiplexers.
  - Similar to colored tokens in dataflow computer systems (Arvind, 1978).
  - Implementation of the folded graph need not use actual multiplexers.
  - **Limiting Case**: Parallel processor with an interconnection network to switch values between processors and memories.
Sharing Storage

Constants hard-wired
Approaches to Scheduling

(1) Exhaustive Approaches
   ➔ e.g. EXPL (Barbacci, 1962): exhaustive search, Hafer & Parker: Branch & bound

(2) As-Soon-As-Possible (ASAP)
   ➔ Used by many pioneering systems (e.g. early CMUDA, MIMOLA, and Flamel)
As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) Scheduling
Approaches to Scheduling

(3) List Scheduling
→ Keep list of operations available to be scheduled (i.e. whose predecessors have already been scheduled) and order them for scheduling using some priority function
  • BUD - length of path from operation to end of enclosing block
  • Elf, ISYN - "urgency": length of shortest path from operation to nearest local constraint
→ O(C N\log N) time complexity

(4) Force-Directed Approaches
→ "Force" between operation and control-step proportional to the number of operations of the same type that could go in that control step
→ Scheduling to minimize force tends to minimize the number of resources needed (e.g. HAL)
→ O(C N^2) time complexity
Basic Force-Directed Algorithm

repeat {
    (1) evaluate ASAP and ALAP limits for each operation;
    (2) create or update distribution graphs;
    (3) calculate the self-force for every unscheduled operation and feasible control step;
    (4) add predecessor and successor forces to self-force;
    (5) schedule the operation with lowest force;
} until ( all operations are scheduled; )
Minimizing Resources Using Force-Directed Techniques

<table>
<thead>
<tr>
<th></th>
<th>×± L</th>
<th></th>
<th>×± L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Allocation

- Mapping operations onto operators
- Assigning values to registers
- Providing interconnections between operators and registers using buses and MUX's

- Two general approaches:
  - Constructive/Iterative
  - Global
Basic Constructive/Iterative Allocation

1. select an operation, value, or interconnection;
2. assign it to an available operator, register or bus, using a heuristic;
3. if (all operations, values and interconnections not assigned) goto (1);
4. select an operation, value, or interconnection already assigned;
5. reassign it using a heuristic or stochastic approach
6. if design constraints still not met) goto (4);
Allocation and Assignment

Given:

Schedule
# (& Types) FUs
# Registers
# Busses

Given:

# (& Types) of FUs
Assignment of Operations to FUs
Minimum possible Schedule
Approaches to Allocation and Assignment

- **Left-Edge Algorithm**
  (REAL(ADAM): Registers)
  (CATHEDRAL: extended with heuristic for repetitive schedules)

- **Clique Partitioning with Graph-Based Heuristic**
  (FACET: Combining Operations into ALUs, Registers, Busses)

- **Weight-directed Clique Partitioning**
  (HAL: Registers, Merging Multiplexers into Busses)

- **Vertex Graph Coloring with Compatibility Sets**
  (CADDY: Registers, FUs, Busses)

- **Bipartite Graph Edge Coloring**
  (<ESC>, SPAID : Register files / Busses)

- **0-1 Linear Programming**
  (Balakrishnan et al.: Multi-port Memories)
Register Allocation and Assignment

Left-Edge Algorithm

- Birth Time (write)
- Death Time (last read)

Share condition:
1. Life times do not overlap
2. Overlap only where one variable dies and the other variable is born

Diagrams with nodes labeled v1, v2, v3, v4, v5.
Software Automation Versus Hardware Design Automation

Silicon Compiler

Verification

Gate Netlist

RTL

Synthesis

Analysis

Synthesis

Analysis

Analysis

Mask Layout (GDS2, CIF)
Future Trends

§ There are 4 basic tasks on each level of design:

• **Style Selection**
  - depend on the goal assigned to particular implementation.
  - the design constraints must be translated into different design styles to automate refinement.

• **Refinement**
  - by partitioning or synthesis;
  - involve translating a behavioral description into a structure of pre-defined components.
  - the refinement translation is not unique; usually several design styles can selected.
Future Trends

- **Optimization**
  - improve utilization of allocated resources such as: time, area, number of tracks, number of function units, etc.

- **Strategy formulation**
  - A strategy is a sequence of different style selections, refinement and optimization steps.
  - The type and order of steps in the sequence characterize a strategy.
Future Trends

- **Mechanisms** to formulate a proper strategy are:

  (1) Constrains propagation process which partitions a design module's constraints into constraints for each of its components (Constraint Generation, Propagation, Satisfaction).

  (2) Evaluation process supported by provided timing analysis, cost, and power report.

  (3) Trade-off made by using different design styles.