Module 1

◆ Objectives

• Motivation
• Compiling language models into abstract models
• Behavioral-level optimization and program-level transformations
Synthesis

- Transform behavioral into structural view
- **Architectural-level synthesis:**
  - Architectural abstraction level
  - Determine *macroscopic* structure
  - Example: major building blocks
- **Logic-level synthesis:**
  - Logic abstraction level
  - Determine *microscopic* structure
  - Example: logic gate interconnection

---

Models and flows

- **Verilog**
- **VHDL**
- **SystemC**
- **Esterel**
- **Statecharts**
- **Schematics**
- **GDS2**

**LANGUAGE MODELS**
- HDL
- compilation

**ABSTRACT MODELS**
- Operations and dependencies (Data-flow & sequencing graphs)
- FSMs – Logic functions (State-diagrams & logic networks)
- Interconnected logic blocks (Logic networks)
- Physical design (mask layout)

**ARCHITECTURAL LEVEL**
- **LOGIC LEVEL**
- **STRUCTURAL VIEW**
- **BEHAVIORAL VIEW**

(c) Giovanni De Micheli
Example

diffeq {
    read (x; y; u; dx; a);
    repeat {
        xl = x+dx;
        ul = u – (3 · x · u · dx) – (3 · y · dx)
        yl = y + u · dx ;
        c = x < a;
        X = xl; u = ul; y = yl;
    }
    until ( c )
    write (y)
}

Example of structures
Architectural-level synthesis motivation

◆ Raise input abstraction level
  • Reduce specification of details
  • Extend designer base
  • Self-documenting design specifications
  • Ease modifications and extensions

◆ Reduce design time

◆ Explore and optimize macroscopic structure:
  • Series/parallel execution of operations
Architectural-level synthesis

◆ Translate HDL models into sequencing graphs

◆ Behavioral-level optimization:
  • Optimize abstract models independently from the implementation parameters

◆ Architectural synthesis and optimization:
  • Create macroscopic structure:
    ◦ Data-path and control-unit
  • Consider area and delay information of the implementation

Compilation and behavioral optimization

◆ Software compilation:
  • Compile program into intermediate form
  • Optimize intermediate form
  • Generate target code for an architecture

◆ Hardware compilation:
  • Compile HDL model into sequencing graph
  • Optimize sequencing graph
  • Generate gate-level interconnection for a cell library
Hardware and software compilation

Compilation

◆ Front-end:
  - Lexical and syntax analysis
  - Parse-tree generation
  - Macro-expansion
  - Expansion of meta-variables

◆ Semantic analysis:
  - Data-flow and control-flow analysis
  - Type checking
  - Resolve arithmetic and relational operators
Parse tree example

\[ a = p + q \times r \]

Behavioral-level optimization

- Semantic-preserving transformations aiming at simplifying the model
- Applied to parse-trees or during their generation
- Taxonomy:
  - *Data-flow* based transformations
  - *Control-flow* based transformations
Data-flow based transformations

- Tree-height reduction
- Constant and variable propagation
- Common sub-expression elimination
- Dead-code elimination
- Operator-strength reduction
- Code motion

Tree-height reduction

- Applied to arithmetic expressions
- Goal:
  - Split into two-operand expressions to exploit hardware parallelism at best
- Techniques:
  - Balance the expression tree
  - Exploit commutativity, associativity and distributivity
Example of tree-height reduction using commutativity and associativity

\[
x = a + b \cdot c + d \quad \rightarrow \quad x = (a + d) + b \cdot c
\]

Example of tree-height reduction using distributivity

\[
x = a \cdot (b \cdot c \cdot d + e) \quad \rightarrow \quad x = a \cdot b \cdot c \cdot d + a \cdot e;
\]
Examples of propagation

◆ Constant propagation
  \[ a = 0; \ b = a + 1; \ c = 2 \cdot b; \]
  \[ a = 0; \ b = 1; \ c = 2; \]

◆ Variable propagation:
  \[ a = x; \ b = a + 1; \ c = 2 \cdot x; \]
  \[ a = x; \ b = x + 1; \ c = 2 \cdot x; \]

Sub-expression elimination

◆ Logic expressions:
  • Performed by logic optimization
  • Kernel-based methods

◆ Arithmetic expressions:
  • Search isomorphic patterns in the parse trees
  • Example:
    \[ a = x + y; \ b = a + 1; \ c = x + y \]
    \[ a = x + y; \ b = a + 1; \ c = a; \]
Examples of other transformations

◆ Dead-code elimination:
  \[ a = x; \quad b = x + 1; \quad c = 2 \cdot x; \]
  \[ a = x; \quad \text{can be removed if not referenced} \]

◆ Operator-strength reduction:
  \[ a = x^2, \quad b = 3 \cdot x; \]
  \[ a = x \cdot x; \quad t = x << 1; \quad b = x + t; \]

◆ Code motion:
  \[
  \begin{align*}
  & \text{for (} i = 1; i < a \cdot b) \{ \} \\
  & t = a \cdot b; \quad \text{for (} i = 1; i < t) \{ \}
  \end{align*}
  \]

Control-flow based transformations

◆ Model expansion

◆ Conditional expansion

◆ Loop expansion

◆ Block-level transformations
Model expansion

- Expand subroutine
  - Flatten hierarchy
  - Expand scope of other optimization techniques
- Problematic when model is called more than once
- Example:
  
  \[
  x = a + b; \quad y = a \cdot b; \quad z = \text{foo}(x, y);
  \]
  
  \[
  \text{foo}(p,q) \{ \ t=q - p; \ \text{return}(t); \ \}
  \]

  By expanding \text{foo}:
  
  \[
  x = a + b; \quad y = a^*b; \quad z = y - x;
  \]

Conditional expansion

- Transform conditional into parallel execution with test at the end
- Useful when test depends on late signals
- May preclude hardware sharing
- Always useful for logic expressions
- Example:
  
  \[
  y = ab; \ \text{if} \ (a) \ \{ x = b + d; \ \} \ \text{else} \ { x = bd; } \}
  \]
  
  - Can be expanded to: \( x = a(b + d) + a'bd \)
  - And simplified as: \( y = ab; x = y + d (a + b) \)
**Loop expansion**

- Applicable to loops with data-independent exit conditions
- Useful to expand scope of other optimization techniques
- Problematic when loop has many iterations
- Example:
  \[
  x = 0; \text{ for } (I = 1; I < 3; I++) \{ x = x + 1; \}
  \]
  - Expanded to:
  \[
  x = 0; \ x = x + 1; \ x = x + 2; \ x = x + 3
  \]

**Module2**

- Objectives
  - Architectural optimization
  - Scheduling, resource sharing, estimation
Architectural synthesis and optimization

◆ Synthesize macroscopic structure in terms of building-blocks

◆ Explore area/performance trade-off:
  • maximize performance implementations subject to area constraints
  • minimize area implementations subject to performance constraints

◆ Determine an optimal implementation

◆ Create logic model for data-path and control

Design space and objectives

◆ Design space:
  • Set of all feasible implementations

◆ Implementation parameters:
  • Area
  • Performance:
    ◆ Cycle-time
    ◆ Latency
    ◆ Throughput (for pipelined implementations)
  • Power consumption
Design evaluation space

Hardware modeling

- Circuit behavior:
  - Sequencing graphs

- Building blocks:
  - Resources

- Constraints:
  - Timing and resource usage
Resources

◆ Functional resources:
  • Perform operations on data
  • Example: arithmetic and logic blocks

◆ Storage resources:
  • Store data
  • Example: memory and registers

◆ Interface resources:
  • Example: busses and ports

Resources and circuit families

◆ Resource-dominated circuits.
  • Area and performance depend on few, well-characterized blocks
  • Example: DSP circuits

◆ Non resource-dominated circuits
  • Area and performance are strongly influenced by sparse logic, control and wiring
  • Example: some ASIC circuits
Implementation constraints

◆ Timing constraints:
  - Cycle-time
  - Latency of a set of operations
  - Time spacing between operation pairs

◆ Resource constraints:
  - Resource usage (or allocation)
  - Partial binding

Synthesis in the temporal domain

◆ Scheduling:
  - Associate a \textit{start-time} with each operation
  - Determine \textit{latency} and parallelism of the implementation

◆ Scheduled sequencing graph:
  - Sequencing graph with start-time annotation
Synthesis in the spatial domain

◆ **Binding:**
  - Associate a resource with each operation with the same type
  - Determine the area of the implementation

◆ **Sharing:**
  - Bind a resource to more than one operation
  - Operations must not execute concurrently

◆ **Bound sequencing graph:**
  - Sequencing graph with resource annotation
Estimation

- Resource-dominated circuits.
  - Area = sum of the area of the resources bound to the operations
    - Determined by binding
  - Latency = start time of the sink operation (minus start time of the source operation)
    - Determined by scheduling

- Non resource-dominated circuits
  - Area also affected by:
    - Registers, steering logic, wiring and control
  - Cycle-time also affected by:
    - Steering logic, wiring and (possibly) control
Approaches to architectural optimization

- **Multiple-criteria optimization problem:**
  - Area, latency, cycle-time

- **Determine Pareto optimal points:**
  - Implementations such that no other has all parameters with inferior values

- **Draw trade-off curves:**
  - Discontinuous and highly nonlinear

---

**Approaches to architectural optimization**

- **Area/latency trade-off**
  - for some values of the cycle-time.

- **Cycle-time/latency trade-off**
  - for some binding (area)

- **Area/cycle-time trade-off**
  - for some schedules (latency)
Area-latency trade-off

◆ Rationale:
  • Cycle-time dictated by system constraints

◆ Resource-dominated circuits:
  • Area is determined by resource usage

◆ Approaches:
  • Schedule for minimum latency under resource usage constraints
  • Schedule for minimum resource usage under latency constraints
  ♦ for varying cycle-time constraints
Summary

◆ Behavioral optimization:
  • Create abstract models from HDL models
  • Optimize models without considering implementation parameters

◆ Architectural synthesis and optimization
  • Consider resource parameters
  • Multiple-criteria optimization problem:
    ♦ area, latency, cycle-time