Unit 2: High-Level Synthesis

- Course contents
  - Hardware modeling
  - Data flow
  - Scheduling/allocation/assignment

- Reading
  - Chapter 11
High-Level Synthesis (HLS)

- Hardware-description language (HDL) synthesis
  - Starts from a register-transfer level (RTL) description; circuit behavior in each clock cycle is fixed.
  - Uses logic synthesis techniques to optimize the design.
  - Generates a netlist.

- High-level synthesis (also called architectural or behavioral synthesis)
  - Starts from an abstract behavioral description.
  - Generates an RTL description.
  - It normally has to perform the trade-off between the number of cycles and the hardware resources to fulfill a task.
HLS Versus RTL Synthesis

• RTL synthesis implements all functionality within a single clock cycle.

• Behavioral synthesis automatically allocates the functionality across multiple clock cycles.
What Is Generated by HLS

- Behavioral Compiler creates a design that consists of a datapath, memory I/O and a control FSM
Benefits of HLS (1)

• Quick specification and verification.
  - Specify behavioral HDL easily, since it's intuitive and natural to write.
  - Save time -- behavioral HDL code is up to 10 times shorter than equivalent RTL.
  - Simulate orders of magnitude faster because of the higher level of abstraction.
  - Reuse designs more readily by starting with a more abstract description.

• Automatically infer memory and generate finite state machine (FSM).
  - Specify memory reads and writes.
  - Schedule memory I/O, resolve contention and build control FSM.
  - Trade-off single vs. multiported memories.
  - Generate a new FSM.
Benefits of HLS (2)

• Explore architectural trade-offs.
  - Create multiple architectures from a single specification.
  - Trade-off throughput and latency using high-level constraints.
  - Analyze various combinations of technology-specific datapath and memory resources.
  - Evaluate cost/performance of various implementations rapidly.

• Reduce design time
  - Model hardware and software components of system concurrently.
  - Easily implement algorithms in behavioral HDL and generate RTL code with a behavioral compiler.
  - Verify hardware in system context at various levels of abstraction.
Hardware Models for High-level Synthesis

• All HLS systems need to restrict the target hardware.
  – The search space is too large, otherwise.

• All synthesis systems have their own peculiarities, but most systems generate synchronous hardware and build it with functional units:
  – A functional unit can perform one or more computations, e.g. addition, multiplication, comparison, ALU.
Hardware Models

- **Registers**: they store inputs, intermediate results and outputs; sometimes several registers are taken together to form a register file.

- **Multiplexers**: from several inputs, one is passed to the output.
Hardware Models (cont’d)

- **Buses**: a connection shared between several hardware elements, such that only one element can write data at a specific time.

- **Three-state (tri-state) drivers** control the exclusive writing on the bus.
Hardware Models (cont’d)

- Parameters defining the hardware model for the synthesis problem:
  - **Clocking strategy**: e.g. single or multiple phase clocks.
  - **Interconnect**: e.g. allowing or disallowing buses.
  - **Clocking of functional units**: allowing or disallowing of
    - multicycle operations
    - chaining
    - pipelined units.
Chaining, Multicycle Operation, Pipelining
Example of a HLS Hardware Model

△ = multiplexer input
● = tristate bus driver

one or more buses

registers and/or register files

one or more FU’s
Hardware Concepts: Data Path + Control

- Hardware is normally partitioned into two parts:
  - **Data path**: a network of functional units, registers, multiplexers and buses.
    - The actual “computation” takes place in the data path.
  - **Control**: the part of the hardware that takes care of having the data present at the right place at a specific time, of presenting the right instructions to a programmable unit, etc.

- High-level synthesis often concentrates on data-path synthesis.
  - The control part is then realized as a finite state machine or in microcode.
Steps of High Level Synthesis

- **Preprocess** the design with high-level optimization
  - Code motion
  - Common subexpression elimination
  - Loop unrolling
  - Constant Propagation
  - Modifications taking advantage of associativity and distributivity, etc.
- **Transform** the optimized design into intermediate format (internal representation) which reveals more structural characteristics of the design.
- **Optimize** the intermediate format
  - Tree height reduction
  - Behavior retiming
- **Allocate** the required resources to implement the design
  - Also called module selection
- **Schedule** each operation to be performed at certain time such that no precedence constraint is violated.
- **Assign (Bind)** each operation to a specific functional unit and each variable to a register.
HLS Optimization Criteria

- Typically, speed, area, and power consumption.
- Optimization is often constrained
  - Optimize area when the minimum speed is given ⇒ time-constrained synthesis.
  - Optimize speed when a maximum for each resource type is given ⇒ resource-constrained synthesis.
  - Minimize power dissipation for a given speed and area requirement => Power-constrained synthesis
Input Format

• The algorithm, that is the input to a high-level synthesis system, is often provided in textual form either
  – in a conventional programming language, such as C, C++, SystemC, or
  – in a hardware description language (HDL), which is more suitable to express the parallelism present in hardware.

• The description has to be parsed and transformed into an internal representation and thus conventional compiler techniques can be used.
Example of High-Level Optimization

- Applying the distributivity law to reduce resource requirement.
Internal Representation

• Most systems use some form of a data-flow graph (DFG).
  - A DFG may or may not contain information on control flow.

• A data-flow graph is built from
  - Vertices (nodes): representing computation, and
  - Edges: representing precedence relations.

\[
\begin{align*}
x & := a \times b; \\
y & := c + d; \\
z & := x + y;
\end{align*}
\]
Token Flow in a DFG

• A node in a DFG fires when all tokens are present at its inputs.
• The input tokens are consumed and an output token is produced.

A token

Firing a node

Generate a token after firing

Z
Conditional Data Flow

- By means of two special nodes:
Explicit Iterative Data Flow

- Selector and distributor nodes can be used to describe iteration.

  Example:
  
  ```
  while (a > b)
    a ← a − b;
  ```

- Loops require careful placement of initial tokens on edges.
Implicit Iterative Data Flow

- Iteration implied by regular input stream of tokens.
- Initial tokens act as buffers.
- Delay elements instead of initial tokens.
Iterative DFG Example

A second-order filter section.
Optimization of Internal Representation

- Restructuring data and control flow graphs prior to the actual mapping onto hardware.

Examples:
  - Replacing chain of adders by a tree.
  - Behavior retiming
Behavior Retiming (BRT)

• By moving registers through logic and hierarchical boundaries, BRT reduces the clock period with minimum area impact.
### Effectiveness of BRT

**Synopsys exp:**

<table>
<thead>
<tr>
<th>Design Type</th>
<th>Speed (Clock Period)</th>
<th>Gates</th>
<th>Speed (Clock Period)</th>
<th>Gates</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>44 ns</td>
<td>10,913 gates</td>
<td>30.6 ns</td>
<td>11,313 gates</td>
<td>30% faster, 4% more area</td>
</tr>
<tr>
<td>Control</td>
<td>23.1 ns</td>
<td>3,598 gates</td>
<td>19.6 ns</td>
<td>4,575 gates</td>
<td>15% faster, 27% more area</td>
</tr>
<tr>
<td>Control</td>
<td>28.6 ns</td>
<td>3,585 gates</td>
<td>28.6 ns</td>
<td>3,399 gates</td>
<td>same-speed, 6% less area</td>
</tr>
<tr>
<td>Dataflow &amp; Control</td>
<td>17 ns</td>
<td>28,900 gates</td>
<td>12.5 ns</td>
<td>30,100 gates</td>
<td>26% faster, 4% more area</td>
</tr>
<tr>
<td>Dataflow &amp; Control</td>
<td>16 ns</td>
<td>7,620 gates</td>
<td>13 ns</td>
<td>8,019 gates</td>
<td>20% faster, 5% more area</td>
</tr>
<tr>
<td>Dataflow</td>
<td>22 ns</td>
<td>4,990 gates</td>
<td>18.5 ns</td>
<td>6,109 gates</td>
<td>16% faster, 2% more area</td>
</tr>
<tr>
<td>Dataflow</td>
<td>28 ns</td>
<td>31,226 gates</td>
<td>26 ns</td>
<td>32,032 gates</td>
<td>8% faster, 2% more area</td>
</tr>
<tr>
<td>Dataflow</td>
<td>26.2 ns</td>
<td>14,351 gates</td>
<td>23.6 ns</td>
<td>13,847 gates</td>
<td>10% faster, 4% more area</td>
</tr>
<tr>
<td>Dataflow</td>
<td>25.9 ns</td>
<td>16,798 gates</td>
<td>20.8 ns</td>
<td>15,550 gates</td>
<td>20% faster, 7% less area</td>
</tr>
<tr>
<td>Dataflow</td>
<td>45 ns</td>
<td>28,705 gates</td>
<td>26 ns</td>
<td>30,987 gates</td>
<td>42% faster, 8% more area</td>
</tr>
</tbody>
</table>

- RTL designs have a single clock net and were synthesized into gates using Synopsys Design Compiler.
- Design type: dataflow implies significant number of operators; control implies state machine dominated.
HLS Subtasks: Allocation, Scheduling, Assignment

• Subtasks in high-level synthesis
  – **Allocation (Module selection):** specify the hardware resources that will be necessary.
  – **Scheduling:** determine for each operation the time at which it should be performed such that no precedence constraint is violated.
  – **Assignment (Binding):** map each operation to a specific functional unit and each variable to a register.

• Remarks:
  – Though the subproblems are strongly interrelated, they are often solved separately. However, to attain a better solution, an iterative process executing these three subtasks must be performed.
  – Most scheduling problems are NP-complete ⇒ heuristics are used.
Example of High Level Synthesis

- The following couple of slides shows an example of scheduling and binding of a design with given resource allocation.
- Given the second-order filter which is first made acyclic:
Example of Scheduling

- The schedule and operation assignment with an allocation of one adder and one multiplier:
Perform Binding to Generate Data Path

- The resulting data path after register assignment.
  - The specification of a controller would complete the design.
Resource Allocation Problem

• This problem is relatively simple. It simply decides the kinds of hardware resources (hardware implementation for certain functional units such as adder, multiplier, etc.) and the quantity of these resources.
  – For example two adders, one multiplier, 4 32-bit registers, etc. for a certain application.

• The decision made in this step has a profound influence on the scheduling which under the given resource constraints decides the time when an operation should be executed by a functional unit.

• This step set an upper bound on the attainable performance.
Problem Formulation of Scheduling

- Input consists of a DFG $G(V, E)$ and a library $\mathcal{R}$ of resource types.
- There is a fixed mapping from each $v \in V$ to some $r \in \mathcal{R}$; the execution delay $\delta(v)$ for each operation is therefore known.
- The problem is time-constrained; the available execution times are in the set $\mathcal{T} = \{0, 1, \ldots, T_0 - 1\}$.
- A schedule maps each operation to its starting time; for each edge $(v_i, v_j) \in E$, a schedule should respect: $\sigma(v_j) \geq \sigma(v_i) + \delta(v_i)$.
- Given the resource type cost $\omega(r)$ and the requirement function $N_r(\sigma)$, the cost of a schedule $\sigma$ is given by:

$$\sum_{r \in \mathcal{R}} \omega(r)N_r(\sigma).$$
ASAP Scheduling

- **As soon as possible (ASAP) scheduling** maps an operation to the earliest possible starting time not violating the precedence constraints.

- **Properties:**
  - It is easy to compute by finding the longest paths in a directed acyclic graph.
  - It does not make any attempt to optimize the resource cost.
Graph for ASAP Scheduling
Mobility-Based Scheduling

- Compute both the ASAP and ALAP (as late as possible) schedules $\sigma_S$ and $\sigma_L$.
- For each $v \in V$, determine the scheduling range $[\sigma_S(v), \sigma_L(v)]$.
- $\sigma_L(v) - \sigma_S(v)$ is called the mobility of $v$.
- Mobility-based scheduling tries to find the best position within its scheduling range for each operation.

- A partial schedule $\tilde{\sigma} : V \rightarrow [\mathcal{T}, \mathcal{T}]$ assigns a scheduling range to each $v \in V$, 
  $$\tilde{\sigma}(v) = [\tilde{\sigma}_{\text{min}}(v), \tilde{\sigma}_{\text{max}}(v)].$$
- Finding a schedule can be seen as the generation of a sequence of partial schedules: $\tilde{\sigma}^{(0)}, \ldots, \tilde{\sigma}^{(n)}$. 
Simple Mobility-Based Scheduling

- A partial schedule $\tilde{\sigma} : V \rightarrow [\mathcal{T}, \mathcal{T}]$ assigns a scheduling range to each
- Finding a schedule can be seen as the generation of a sequence of partial schedules

```
"determine $\tilde{\sigma}^{(0)}$ by computing $\sigma_S$ and $\sigma_L$";
$k \leftarrow 0$;
while ("there are unscheduled operations") {
  $v \leftarrow "one of the nodes with lowest mobility"$;
  "schedule $v$ at some time that optimizes the current resource utilization";
  "determine $\tilde{\sigma}^{(k+1)}$ by updating the scheduling ranges of the unscheduled nodes";
  $k \leftarrow k + 1$
}
```
List Scheduling

- A resource-constrained scheduling method.
- Start at time zero and increase time until all operations have been scheduled.
  - Consider the precedence constraint.
- The ready list $L_t$ contains all operations that can start their execution at time $t$ or later.
- If more operations are ready than there are resources available, use some priority function to choose, e.g. the longest-path to the output node $\Rightarrow$ critical-path list scheduling.
List Scheduling Example
The Assignment Problem

- Subtasks in assignment:
  - operation-to-FU assignment
  - value grouping
  - value-to-register assignment
  - transfer-to-wire assignment
  - wire to FU-port assignment

- In general: task-to-agent assignment
Compatibility and Conflict Graphs

- **Clique partitioning** gives an assignment in a compatibility graph.

- **Graph coloring** gives an assignment in the complementary conflict graph.
The Assignment Problem

- Assumption: assignment follows scheduling.
- The claim of a task on an agent is an interval ⇒ minimum resource utilization can be found by left-edge algorithm.
- In case of iterative algorithm, interval graph becomes circular-arc graph ⇒ optimization is NP-complete.
Tseng and Sieworek’s Algorithm

\[ k \leftarrow 0; \]
\[ G_c^k(V_c^k, E_c^k) \leftarrow G_c(V_c, E_c); \]
\[ \textbf{while } (E_c^k \neq \emptyset) \{ \]
\[ \quad \text{"find } (v_i, v_j) \in E_c^k \text{ with largest set of common neighbors"}; \]
\[ \quad N \leftarrow \text{"set of common neighbors of } v_i \text{ and } v_j"; \]
\[ \quad s \leftarrow i \cup j; \]
\[ \quad V_c^{k+1} \leftarrow V_c^k \cup \{v_s\} \setminus \{v_i, v_j\}; \]
\[ \quad E_c^{k+1} \leftarrow \emptyset; \]
\[ \quad \textbf{for each } (v_m, v_n) \in E_c^k \]
\[ \quad \quad \text{if } (v_m \neq v_i \land v_m \neq v_j \land v_n \neq v_i \land v_n \neq v_j) \]
\[ \quad \quad \quad E_c^{k+1} \leftarrow E_c^{k+1} \cup \{(v_m, v_n)\}; \]
\[ \quad \textbf{for each } v_n \in N \]
\[ \quad \quad E_c^{k+1} \leftarrow E_c^{k+1} \cup \{(v_n, v_s)\}; \]
\[ \quad k \leftarrow k + 1; \]
\[ \} \]
Clique-Partitioning Example

The diagram illustrates a graph with vertices labeled $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$. The graph is partitioned into cliques, with each clique represented by a separate image. The first image shows the complete graph, while the subsequent images highlight different partitions of the graph into cliques.
Example of Behavior Optimization

Behavior Optimization of Arithmetic (BOA)
# Effectiveness of BOA

<table>
<thead>
<tr>
<th>Design Type</th>
<th>RTL Design</th>
<th>BOA</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motion estimation</td>
<td>23.6 ns, 12,793 gates</td>
<td>20.2 ns, 12,215 gates</td>
<td>14% faster, 5% less area</td>
</tr>
<tr>
<td>Graphics interpolation</td>
<td>19.2 ns, 3,507 gates</td>
<td>17.5 ns, 2,952 gates</td>
<td>9% faster, 16% less area</td>
</tr>
<tr>
<td>Color space conversion and scaling</td>
<td>16 ns, 35,866 gates</td>
<td>14.9 ns, 34,397 gates</td>
<td>7% faster, 4% less area</td>
</tr>
<tr>
<td>Sum of 9 operands</td>
<td>7.7 ns, 1,418 gates</td>
<td>5.3 ns, 1,307 gates</td>
<td>31% faster, 8% less area</td>
</tr>
<tr>
<td>(a \times b + 1)</td>
<td>11.6 ns, 2,577 gates</td>
<td>9.3 ns, 2,524 gates</td>
<td>20% faster, 2% less area</td>
</tr>
<tr>
<td>(a \times 4104)(^{010000010000001000})</td>
<td>4.4 ns, 759 gates</td>
<td>3.1 ns, 449 gates</td>
<td>30% faster, 40% less area</td>
</tr>
<tr>
<td>(a \times 3E3E)(^{001111110000111110})</td>
<td>5.7 ns, 927 gates</td>
<td>4.6 ns, 709 gates</td>
<td>19% faster, 23% less area</td>
</tr>
<tr>
<td>(a \times b + c)</td>
<td>11.0 ns, 2,707 gates</td>
<td>10.0 ns, 2,680 gates</td>
<td>9% faster, same area</td>
</tr>
<tr>
<td>(a \times b + c \times d + e \times f)</td>
<td>14.2 ns, 7,435 gates</td>
<td>12.8 ns, 7,110 gates</td>
<td>10% faster, 4% less area</td>
</tr>
<tr>
<td>Sum of 16 operands</td>
<td>8.1 ns, 2,836 gates</td>
<td>6.7 ns, 2,123 gates</td>
<td>17% faster, 25% less area</td>
</tr>
</tbody>
</table>
Summary

- Concepts of High Level Synthesis (HLS)
- Benefits of HLS
- Steps of High Level Synthesis
- Internal Representation and Optimization
- Resource Allocation
- Algorithms for Scheduling
- Algorithms for Binding (Assignment)