Synthesis of VHDL Code
This slide set covers
- Fundamental limitation of EDA software
- Realization of VHDL operator
- Realization of VHDL data type
- VHDL synthesis flow
- Timing consideration

Fundamental limitation of EDA software
Can \textit{C-to-hardware} be done? No, not really

EDA tools consist of:
- Core: optimization algorithms
- Shell: wrappers around the \textbf{core} to carry out conversions, file operations, etc.

Theoretical computer science defines
- Computability (bounds on what algorithms can do)
- Computation complexity (inherent complexity to arrive at an optimal solution)
Computability and Computational Complexity

A problem is computable if an algorithm exists

Some problems are not computable, e.g., the **halting problem**

Can we develop a program that takes any program and its input, and determines whether the computation of that program will eventually halt?

Any attempt to examine the *meaning* of a program is uncomputable

For computable problems, analysis of computation complexity determines how fast an algorithm can run

  Algorithms are analyzed for both *time* and *space* complexity

Computation time depends on the size of the input, the type of processor, programming language, compiler and even coding style

To eliminate the smaller factors, computational analysis focuses only on the **order** of the algorithm, as a function of the input size
Big-O notation

\( f(n) \) is \( O(g(n)) \) if \( n_0 \) and \( c \) can be found to satisfy

\[ f(n) < cg(n) \text{ for any } n, n > n_0 \]

\( g(n) \) is usually a simple function: 1, \( n \), \( \log_2 n \), \( n^2 \), \( n^3 \), \( 2^n \)

For example, the following are \( O(n^2) \)

\[
(0.1n^2) \quad \leftrightarrow \quad (n^2 + 5n + 9) \quad \leftrightarrow \quad (500n^2 + 1000000)
\]

Interpretation of Big-O

- Filter out constants and other less important terms
- Focus on scaling factor of an algorithm, i.e., what happens if the input size increases

<table>
<thead>
<tr>
<th>input size</th>
<th>( n )</th>
<th>( \log_2 n )</th>
<th>( n \log_2 n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 2^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2 ( \mu ) s</td>
<td>1 ( \mu ) s</td>
<td>2 ( \mu ) s</td>
<td>4 ( \mu ) s</td>
<td>8 ( \mu ) s</td>
<td>4 ( \mu ) s</td>
</tr>
<tr>
<td>4</td>
<td>4 ( \mu ) s</td>
<td>2 ( \mu ) s</td>
<td>8 ( \mu ) s</td>
<td>16 ( \mu ) s</td>
<td>64 ( \mu ) s</td>
<td>16 ( \mu ) s</td>
</tr>
<tr>
<td>8</td>
<td>8 ( \mu ) s</td>
<td>3 ( \mu ) s</td>
<td>24 ( \mu ) s</td>
<td>64 ( \mu ) s</td>
<td>512 ( \mu ) s</td>
<td>256 ( \mu ) s</td>
</tr>
<tr>
<td>16</td>
<td>16 ( \mu ) s</td>
<td>4 ( \mu ) s</td>
<td>64 ( \mu ) s</td>
<td>256 ( \mu ) s</td>
<td>4 ( ms )</td>
<td>66 ( ms )</td>
</tr>
<tr>
<td>32</td>
<td>32 ( \mu ) s</td>
<td>5 ( \mu ) s</td>
<td>160 ( \mu ) s</td>
<td>1 ( ms )</td>
<td>33 ( ms )</td>
<td>71 ( min )</td>
</tr>
<tr>
<td>48</td>
<td>48 ( \mu ) s</td>
<td>5.5 ( \mu ) s</td>
<td>268 ( \mu ) s</td>
<td>2 ( ms )</td>
<td>111 ( ms )</td>
<td>9 ( year )</td>
</tr>
<tr>
<td>64</td>
<td>64 ( \mu ) s</td>
<td>6 ( \mu ) s</td>
<td>384 ( \mu ) s</td>
<td>4 ( ms )</td>
<td>262 ( ms )</td>
<td>600,000 ( year )</td>
</tr>
</tbody>
</table>
Computation complexity

Intractable problems are algorithms with $O(2^n)$ -- not computable for large $n$

Frequently tractable heuristic algorithms exist, that run in polynomial time, but generate optimal solutions for only some inputs and/or generate sub-optimal solutions

Many problems encountered in synthesis are intractable

Synthesis software limitations

- Synthesis software cannot obtain the optimal solution
- Synthesis should be viewed as a transformation carried out using a local search
- Good VHDL code helps a lot by providing a good starting point for the local search

There are other design tasks that are intractable, and no amount of fast hardware or clever heuristics can be used to find the optimal solution

Therefore, it is impossible for EDA software to completely automate the design process

This limitation is REAL and is HERE TO STAY!
Realization of VHDL Operators

Logic operators: simple, direct mapping

Relational operators
  =, /=  fast, simple implementation exists
  >, <, etc: more complex implementation, larger delay

Addition operator, and others that can be derived from addition including subtraction, negation and abs, has a multitude of implementations that trade-off speed and area
  Even more complex than the relation operators

Synthesis support for other operators, e.g., shifting, multiplication, division, exponentiation, and floating point operations, is sporadic or non-existent

Because of their complexity, you must be extremely careful about using them in VHDL code
Realization of VHDL Operators

Operator with two constant operands: Simplified in preprocessing such that no hardware is inferred -- used because they clarify the code

```vhdl
constant OFFSET: integer := 8;
signal boundary: unsigned(8 downto 0);
signal overflow: std_logic;
overflow <= '1' when boundary > (2**OFFSET-1) else '0';
```

Operator with one constant operand: Can significantly reduce (cut-in-half) the hardware complexity, e.g., adder vs. incrementer, later implementable with half-adders

```vhdl
y <= rotate_right(x, y); -- full-fledged barrel shifter
y <= rotate_right(x, 3); -- rewiring, easy to implement
y <= x(2 downto 0) & x(7 downto 3); -- rewiring
```

Another example, 4-bit comparator: x=y vs. x=0

\[
(x_3 \oplus y_3)' \cdot (x_2 \oplus y_2)' \cdot (x_1 \oplus y_1)' \cdot (x_0 \oplus y_0)'
\]

Full logic expression

\[
x_3' \cdot x_2' \cdot x_1' \cdot x_0'
\]

Much easier, i.e., only a 4-input NOR gate
An Example 0.55 um Standard-Cell CMOS Implementation

<table>
<thead>
<tr>
<th>width</th>
<th>nand</th>
<th>xor</th>
<th>&gt;a</th>
<th>&gt;d</th>
<th>=</th>
<th>+1a</th>
<th>+1d</th>
<th>+a</th>
<th>+d</th>
<th>mux</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>22</td>
<td>25</td>
<td>68</td>
<td>26</td>
<td>27</td>
<td>33</td>
<td>51</td>
<td>118</td>
<td>21</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>44</td>
<td>52</td>
<td>102</td>
<td>51</td>
<td>55</td>
<td>73</td>
<td>101</td>
<td>265</td>
<td>42</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>85</td>
<td>105</td>
<td>211</td>
<td>102</td>
<td>113</td>
<td>153</td>
<td>203</td>
<td>437</td>
<td>85</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>171</td>
<td>212</td>
<td>398</td>
<td>204</td>
<td>227</td>
<td>313</td>
<td>405</td>
<td>755</td>
<td>171</td>
</tr>
</tbody>
</table>

**area (gate count)**

- a: optimized for area
- d: optimized for delay

**gate count**: in equivalent 2-input NAND gates

<table>
<thead>
<tr>
<th>width</th>
<th>delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.1</td>
</tr>
<tr>
<td>16</td>
<td>0.1</td>
</tr>
<tr>
<td>32</td>
<td>0.1</td>
</tr>
<tr>
<td>64</td>
<td>0.1</td>
</tr>
</tbody>
</table>

**Realization of VHDL data type**

Use and synthesis of 'Z' and '-' (other values other than '0' and '1' not used in synthesis)

'Z' indicates high impedance (or open circuit)

Not a Boolean value but is exhibited in a physical circuit, e.g., as the output of a tri-state buffer
**Tri-State Buffer**

Tri-state buffer

```
   oe  y
  0   Z
  1   a_in
```

Major applications
- Bi-directional I/O pins
- Tri-state bus

VHDL description
```
y <= 'Z' when oe='1' else a_in;

'Z' cannot be used as input or manipulated
f <= 'Z' and a;
y <= data_a when in_bus='Z' else data_b;
```
**Tri-State Buffer**

Because a tri-state buffer is not an ordinary logic value, it is a good idea to separate it from regular code.

Less clear (cannot be synthesized):

```vhdl
with sel select
  y <= 'Z' when "00",
       '1' when "01" | "11",
       '0' when others;
```

Better:

```vhdl
with sel select
  tmp <= '1' when "01" | "11",
       '0' when others;
  y <= 'Z' when sel="00" else tmp;
```
Bi-directional I/O Pins
An important application of a tri-state buffer

```vhdl
entity bi_demo is
  port (bi: inout std_logic;
       ...
begin
  sig_out <= output_expression;
  ...
  <= expression_with_sig_in;
  bi <= sig_out when dir = '1' else 'Z';
  sig_in <= bi;
end;
```
Bi-directional I/O Pins and Tri-State Bus

\[ \text{sig\_in} \leq \text{bi} \text{ when } \text{dir} = '0' \text{ else } 'Z'; \]

Alternative if driving \text{sig\_in}
with \text{sig\_out} when \text{dir} = '1'
is a problem

Tri-state bus
Tri-State Bus

```vhdl
with src_select select
  oe <= "0001" when "00",
       "0010" when "01",
       "0100" when "10",
       "1000" when others;
  y0 <= i0 when oe(0)='1' else 'Z';
  y1 <= i1 when oe(1)='1' else 'Z';
  y2 <= i2 when oe(2)='1' else 'Z';
  y3 <= i3 when oe(3)='1' else 'Z';
data_bus <= y0;
data_bus <= y1;
data_bus <= y2;
data_bus <= y3;
```

Problems with the tri-state bus

- Difficult to optimize, verify and test
- Somewhat difficult to design: is technology dependent and can result in ’contention’
Alternative to Tri-State Bus

Alternative to tri-state bus: \textbf{mux}

\begin{verbatim}
with src_select select
    data_bus <= i0 when "00",
              i1 when "01",
              i2 when "10",
              i3 when others;
\end{verbatim}

Use of ’’-

In conventional logic design, ’’-’’ used as input value: shorthand to make table compact

\begin{tabular}{c|c|}
\textbf{input} & \textbf{output} \\
\hline
\textbf{req} & \textbf{code} \\
\hline
1 0 0 & 10 \\
1 0 1 & 10 \\
1 1 0 & 10 \\
1 1 1 & 10 \\
0 1 0 & 01 \\
0 1 1 & 01 \\
0 0 1 & 00 \\
0 0 0 & 00 \\
\end{tabular}
Use of ‘-’

‘-’ as output value: helps simplification, for example

• If ‘-’ assigned to 0: \( \overline{a}b + a\overline{b} \)

• If ‘-’ assigned to 1: \( a + b \) (much less hardware than if 0)

\[
\begin{array}{c|c|c}
\text{input} & \text{output} \\
\hline
a & b & f \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & - \\
\end{array}
\]

As input value: \( \textbf{(Syntactically correct but Wrong)} \)

\[
y <= "10" \text{ when } \text{req} = "1--" \text{ else } \\
\quad "01" \text{ when } \text{req} = "01-" \text{ else } \\
\quad "00" \text{ when } \text{req} = "001" \text{ else } \\
\quad "00"
\]

Fix

\[
y <= "10" \text{ when } \text{req(3)} = '1' \text{ else } \\
\quad "01" \text{ when } \text{req(3 downto 2)} = "01" \text{ else }
\]
Use of ‘-’

"00" when req(3 downto 1) = "001" else
"00"

Another fix (must include 'use ieee.numeric_std.all):

y <= "10" when std_match(req, "1--") else
"01" when std_match(req, "01-") else
"00" when std_match(req, "001") else
"00"

Wrong (but syntactically correct):

with req select
  y <= "10" when "1--",
      "01" when "01-",
      "00" when "001",
      "00" when others;

Fix:

with req select
  y <= "10" when "100" | "101" | "110" | "111",
      "01" when "010" | "011",
      "00" when others;
Use of ‘-’

'-' as an output value in VHDL may work with some software

```vhdl
sel <= a & b;
with sel select
  y <= '0' when "00",
     '1' when "01",
     '1' when "10",
     '-' when others;
```

VHDL Synthesis Flow

Synthesis: realize VHDL code using logic cells from the target device’s library

Main steps:
- High-level synthesis (translates an algorithm into an architecture consisting of a 
  data path and control path -- done by specialized hardware tools)
- RT level synthesis (the rest generate structural netlists)
- Logic synthesis
- Technology mapping
VHDL Synthesis Flow

Level-by-level transformation and optimization

For complex operators, e.g., adder, comparator
RT Level Synthesis

Realize VHDL code using *generic* RT-level components

Generic implies that the components are technology independent

Components classified into
- function units: those use to implement logic, relational and arith ops
- routing units: various MUXs to construct routing structure
- storage units: registers and latches

During RT-level synthesis, VHDL statements are converted to structural implementations (similar to derivation of the conceptual diagrams given earlier)

Some optimizations such as *operator sharing*, *common code elimination* and *constant propagation* can be applied to reduce hardware and improve performance

Unlike gate- and cell-level synthesis, optimizations are performed in an ad hoc way and scope is very limited

Good design can drastically alter the RT-level structure
Module Generator

The generic RT-level components (from RT-level synthesis) need to be transformed into lower-level components for further processing.

Some components, such as logical operators and MUXs are simple and can be mapped directly into gate-level components.

These are called **random logic** (low regularity) -- can be optimized later in logic synthesis.

Other components such as an adder, subtracter, incrementer, comparator, shifter and multiplier are more complex and need a *module generator*.

They usually show some kind of *repetitive structure*, and are called **regular logic**.

Regular logic is usually designed in advance, as *presynthesized gate-* or *cell-level* netlists.

Manual design can be more efficient than logic synthesis so these components are not flattened or optimized with other components during logic synthesis.
Logic Synthesis

Implement the circuit with the optimal number of *generic* gate level components, such as NAND and NORs.

The result is a structural view, expressed as Boolean functions.

Logic synthesis can be divided into categories:
- Two-level synthesis: sum-of-product format
- Multi-level synthesis (deals with large fan-ins, can trade-off area and speed)

Multi-level synthesis is more efficient and flexible, but more difficult to carry out.
Technology Mapping

Map generic gates to **device-dependent** logic cells

The technology library is provided by the vendors who manufactured, as in FPGAs, or will manufacture, as in ASICs, the device

Mapping in standard-cell ASIC

<table>
<thead>
<tr>
<th>cell name (cost)</th>
<th>symbol</th>
<th>nand-not representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>not (2)</td>
<td>![not symbol]</td>
<td>![nand-not representation]</td>
</tr>
<tr>
<td>nand2 (3)</td>
<td>![nand2 symbol]</td>
<td>![nand-not representation]</td>
</tr>
<tr>
<td>nand3 (4)</td>
<td>![nand3 symbol]</td>
<td>![nand-not representation]</td>
</tr>
<tr>
<td>nand4 (5)</td>
<td>![nand4 symbol]</td>
<td>![nand-not representation]</td>
</tr>
<tr>
<td>aoi (4)</td>
<td>![aoi symbol]</td>
<td>![nand-not representation]</td>
</tr>
<tr>
<td>xor (4)</td>
<td>![xor symbol]</td>
<td>![nand-not representation]</td>
</tr>
</tbody>
</table>

Technology mapping is a difficult process (intractable) and involves the use of **heuristics** and **rule-based** algorithms to find sub-optimal solutions

Std cell libraries usually contain several hundred cells, such as simple gates, 1-bit full adders, MUXs

The **nand-not** representation is used to facilitate the mapping process
Technology Mapping

Std cells are 'tuned' to a particular technology

They are manually designed at the transistor level

Multiple versions of the same function are common, each trading-off area and delay

Top design is a one-to-one, gate-to-cell mapping -- area is 31

Bottom design is optimized for area by selecting specific std cells from library -- result is 17
Technology Mapping

Mapping into an FPGA (with 5-input LUT (Look-Up-Table) cells)

Effective Use of synthesis software

Logic operators: software can do a good job

Relational/Arith operators: manual intervention needed
Effective Use of Synthesis Software

"layout" and "routing structure":

- Silicon chip is 2-dimensional square
- *rectangular* or *tree-shaped* circuit is easier to optimize
Timing Considerations
  Propagation delay
  Synthesis with timing constraint
  Hazards
  Delay-sensitive design

Propagation Delay
  Delay: time required to propagate a signal from an input port to an output port

Cell level delay (vs. RT-level) is the most accurate b/c netlist is final

Simplified model:
  \[ delay = d_{intrinsic} + rC_{load} \]

The \( d_{intrinsic} \) term is the self-loading component, while the \( r*C_{load} \) term is the driver’s resistance and downstream capacitance components

Remember from basic circuit theory that \( resistance*capacitance = time \) (delay)
Propagation Delay

The routing parasitics are not known at synthesis time, only after place and route.

Lumped model of capacitance ($C_{load}$) adds together all wire loads ($C_{wx}$) and input loads ($C_{gx}$).

In advanced technologies, the impact of wire becomes more significant and must be considered to obtain an accurate delay estimation.

System Delay

There are many paths between the inputs and outputs of a typical circuit.

Each of them have different delays -- for overall system timing, we are interested in the critical path delay.
System Delay

The worst **input-to-output** delay

Can be obtained from the netlist by treating it as a graph and extracting the longest path

Called the *topologically critical path*

This method has the drawback that the critical path obtained may be a **false path**

This critical path is a false path because MUXs don’t allow it

That is, a path along which it is impossible to propagate a signal
**System Delay**

When estimating RT level delay:

- It is difficult if the design is mainly *random* logic because the simple logic will go through many transformations and optimizations.
- However, if the design consists of many complex operators (such as addition) and function blocks, the critical path can be identified.

  This is true because these components are typically *pre-designed* and *optimized*.

**Synthesis with Timing Constraints**

It is possible to reduce by delay at the expense of area, i.e., by adding extra logic.

There are multiple implementations that trade-off area and delay.
Synthesis with Timing Constraints

Multilevel logic is flexible, making it possible to add additional gates to achieve shorter delay.

Timing constraints are sometimes needed to guarantee a specific performance metric.

The synthesis process that considers timing constraints is carried out as follows:

- Obtain the minimal-area implementation
- Identify the critical path
- Reduce the delay by adding extra logic
- Repeat 2 & 3 until meeting the constraint
Synthesis with Timing Constraints

It is also possible to perform this process at the RT level

When the design consists of complex operators (blocks), global optimization can be explored (which is more efficient than synthesis optimization at the cell level)

Improvements can be made at the "architectural" level, which can have a huge impact on critical path delay and size
Timing Hazards

Propagation delay: time to obtain a stable output

Hazard: the fluctuation in the output occurring during the transient period
- Static hazard: glitch in output when the signal should be stable
- Dynamic hazard: a glitch in output during the transition

Hazards are caused by multiple converging paths of an output port

\[ sh = \overline{ab} + bc \] (2-to-1 MUX)

Assume \( a \) and \( c \) are 1

Static '0' hazard because of extra delay along \( a_{\_b\_not} \) on transition of \( b \) from 1 to 0
Timing Hazards
Dynamic hazard

Assume $a = c = d = 1$
Transition of $b$ from 1 to 0

Dealing with hazards
Some hazards can be eliminated in theory

Add an AND gate

(c) Revised Karnaugh map and schematic to eliminate hazards
Timing Hazards
Eliminating glitches is very difficult in reality, and almost impossible for synthesis

Multiple inputs can change simultaneously (e.g., cycling from 1111 -> 0000 in a counter)

How do we deal with them?
- **Ignore** glitches in the transient period, e.g., sample after the signal is stabilized

Delay Sensitive Design and its Danger
Boolean algebra is the theoretical model for digital design and most algorithms used in the synthesis process

This model handles only stabilized signals (no transient behavior)

*Delay-sensitive* design, on the other hand, **depends** on the *transient behavior* (delay characteristics) of the circuit

Consider the addition of the *ac* (AND gate) to eliminate the static hazard -- the *ac* term is redundant UNTIL you consider the transient behavior
Delay Sensitive Design and its Danger

Another circuit that depends on transient behavior is the edge detection circuit, with function \( pulse = a \cdot \overline{a} \)

Unfortunately, synthesis software does NOT consider transient behavior and will optimize and eliminate statements such as:

\[
pulse \leq a \text{ and } (\text{not } a)
\]

Other problems include

- During technology mapping, the gates specified may be re-mapped to other gates
- During placement & routing, wire delays may change creating unexpected results
- Difficult to test and verify (redundant logic is difficult to test for defects)

If delay-sensitive design is really needed, it should be done manually, not by synthesis.