Synthesis Of VHDL Code

Outline
1. Fundamental limitation of EDA software
2. Realization of VHDL operator
3. Realization of VHDL data type
4. VHDL synthesis flow
5. Timing consideration

1. Fundamental limitation of EDA software
   - Can “C-to-hardware” be done?
   - EDA tools:
     - Core: optimization algorithms
     - Shell: wrapping
   - What does theoretical computer science say?
     - Computability
     - Computation complexity

Computability
   - A problem is computable if an algorithm exists.
   - E.g., “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

Computation complexity
   - How fast an algorithm can run (or how good an algorithm is)?
   - “Interferences” in measuring execution time:
     - types of CPU, speed of CPU, compiler etc.

Big-O notation
   - $f(n)$ is $O(g(n))$:
     if $n_0$ and $c$ can be found to satisfy: $f(n) < cg(n)$ for any $n, n > n_0$
   - $g(n)$ is simple function: $1, n, \log_2 n, n^2, n^3, 2^n$
   - Following are $O(n^2)$:
     - $0.1n^2$
     - $n^2 + 5n + 9$
     - $500n^2 + 1000000$
Interpretation of Big-O

- Filter out the “interference”: constants and less important terms
- \( n \) is the input size of an algorithm
- The “scaling factor” of an algorithm:
  What happens if the input size increases

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n )</th>
<th>( \log_2 n )</th>
<th>( n \log_2 n )</th>
<th>( n^2 )</th>
<th>( 2^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2 µs</td>
<td>1 µs</td>
<td>2 µs</td>
<td>4 µs</td>
<td>8 µs</td>
</tr>
<tr>
<td>4</td>
<td>4 µs</td>
<td>2 µs</td>
<td>8 µs</td>
<td>16 µs</td>
<td>64 µs</td>
</tr>
<tr>
<td>8</td>
<td>8 µs</td>
<td>3 µs</td>
<td>24 µs</td>
<td>64 µs</td>
<td>256 µs</td>
</tr>
<tr>
<td>16</td>
<td>16 µs</td>
<td>4 µs</td>
<td>64 µs</td>
<td>256 µs</td>
<td>65536 µs</td>
</tr>
<tr>
<td>32</td>
<td>32 µs</td>
<td>5 µs</td>
<td>160 µs</td>
<td>1 ms</td>
<td>33 ms</td>
</tr>
<tr>
<td>48</td>
<td>48 µs</td>
<td>5.5 µs</td>
<td>268 µs</td>
<td>2 ms</td>
<td>111 ms</td>
</tr>
<tr>
<td>64</td>
<td>64 µs</td>
<td>6 µs</td>
<td>384 µs</td>
<td>4 ms</td>
<td>600,000 year</td>
</tr>
</tbody>
</table>

E.g.,

Theoretical limitation

- Synthesis software does not know your intention
- Synthesis software cannot obtain the optimal solution
- Synthesis should be treated as transformation and a “local search” in the “design space”
- Good VHDL code provides a good starting point for the local search

2. Realization of VHDL operator

- Logic operator
  - Simple, direct mapping
- Relational operator
  - =, /= fast, simple implementation exists
  - >, < etc: more complex implementation, larger delay
- Addition operator
- Other arith operators: support varies

What is the fuss about:
- “hardware-software” co-design?
- SystemC, HardwareC, SpecC etc.?
• Operator with two constant operands:
  – Simplified in preprocessing
  – No hardware inferred
  – Good for documentation
  – E.g.,

```vhd
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 the hardware complexity
  – E.g., adder vs. incrementor
  – E.g

```
y <= rotate_right(x, y);  -- barrel shifter
y <= rotate_right(x, 3);  -- rewiring
y <= x(2 downto 0) & x(7 downto 3);
E.g., 4-bit comparator: x=y vs. x=0
```

An example 0.55 um standard-cell CMOS implementation

<table>
<thead>
<tr>
<th>width</th>
<th>en</th>
<th>nuf</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>MUX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>area (gate count)</td>
<td>delay (ns)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

3. Realization of VHDL data type

• Use and synthesis of ‘Z’
  • Use of ‘-’

Use and synthesis of ‘Z’

• Tri-state buffer:
  – Output with “high-impedance”
  – Not a value in Boolean algebra
  – Need special output circuitry (tri-state buffer)

```
  \[\text{0} \quad \text{a}_{\text{in}} \quad \text{y}\]
  \[\text{0} \quad \text{Z} \quad 0\]
  \[\text{1} \quad \text{a}_{\text{in}} \quad 1\]
```

• Major application:
  – Bi-directional I/O pins
  – Tri-state bus
• VHDL description:
  ```vhd
  y <= 'Z' when oe='1' else a_{in};
  ```
  • ‘Z’ cannot be used as input or manipulated
  ```vhd
  f <= 'Z' and a;
  y <= data_a when in_bus='Z' else data_b;
  ```
• Separate tri-state buffer from regular code:
  – Less clear:
    ```
    with sel select
    y <= 'Z' when "00",
         '1' when "01" or "11",
         '0' when others;
    – better:
    with sel select
    tmp <= '1' when "01" or "11",
           '0' when others;
    y <= 'Z' when sel="00" else tmp;
    ```

---

**Bi-directional i/o pins**

---

**Tri-state bus**

---

```
• Problem with tri-state bus
  – Difficult to optimize, verify and test
  – Somewhat difficult to design: “parking”, “fighting”

• Alternative to tri-state bus: mux

```vhdl
with src_select select
  data_bus <= 10 when "00",
               11 when "01",
               12 when "10",
               13 when others;  -- "/
```

– `'-' as output value: help simplification
– E.g.,
  '‐' assigned to 1: \(a + b\)
  '‐' assigned to 0: \(a'b + ab'\)

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>'‐'</td>
</tr>
</tbody>
</table>

Use of ‘‐’

• In conventional logic design
  – ‘‐’ as input value: shorthand to make table compact
  – E.g.,

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0 1</td>
<td>1 0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
</tr>
</tbody>
</table>

Use ‘‐’ in VHDL

• As input value (against our intuition):

- Wrong:

```vhdl
y <= "10" when req="1--" else
    "01" when req="01--" else
    "00" when req="001--" else
    "00";
```

- Fix #1:

```vhdl
y <= "10" when req="1--" else
    "01" when req="01--" else
    "00" when req="001--" else
    "00";
```

- Fix #2:

```vhdl
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:

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

- Fix:

```vhdl
with req select
    y <= "10" when "100"|"101"|"110"|"111",
        "00" when "010"|"011",
        "00" when others;
```
4. VHDL Synthesis Flow

- **Synthesis:**
  - Realize VHDL code using logic cells from the device’s library
  - A refinement process
- **Main steps:**
  - RT level synthesis
  - Logic synthesis
  - Technology mapping

RT level synthesis

- Realize VHDL code using RT-level components
- Somewhat like the derivation of the conceptual diagram
- Limited optimization
- Generated netlist includes
  - “regular” logic: e.g., adder, comparator
  - “random” logic: e.g., truth table description

Module generator

- “regular” logic can be replaced by pre-designed module
  - Pre-designed module is more efficient
  - Module can be generated in different levels of detail
  - Reduce the processing time

Logic Synthesis

- Realize the circuit with the optimal number of “generic” gate level components
- Process the “random” logic
- Two categories:
  - Two-level synthesis: sum-of-product format
  - Multi-level synthesis
Technology mapping

- Map "generic" gates to "device-dependent" logic cells
- The technology library is provided by the vendors who manufactured (in FPGA) or will manufacture (in ASIC) the device

E.g., mapping in standard-cell ASIC

- Device library

<table>
<thead>
<tr>
<th>cell name</th>
<th>symbol</th>
<th>standard cell notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC(2)</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
<tr>
<td>etc.</td>
<td><img src="image3.png" alt="Diagram" /></td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- Cost: 31 vs. 17

E.g., mapping in 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
- "layout" and "routing structure":
  - Silicon chip is 2-dimensional square
  - "rectangular" or "tree-shaped" circuit is easier to optimize
5. Timing consideration

- 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: most accurate
- Simplified model:
  \[ \text{delay} = d_{\text{intrinsc}} + r \times C_{\text{load}} \]
- The impact of wire becomes more dominant

System delay

- The longest path (critical path) in the system
- The worst input to output delay
- E.g.
• RT level delay estimation:
  – Difficult if the design is mainly “random” logic
  – Critical path can be identified if many complex
    operators (such as adder) are used in the
    design.

• Multi-level synthesis is flexible
• It is possible to reduce by delay by
  adding extra logic
• Synthesis with timing constraint
  1. Obtain the minimal-area implementation
  2. Identify the critical path
  3. Reduce the delay by adding extra logic
  4. Repeat 2 & 3 until meeting the constraint

• E.g.,

• Area-delay trade-off curve

• Improvement in “architectural” level design
  (better VHDL code to start with)

• Timing Hazards
  • Propagation delay: time to obtain a stable
    output
  • Hazards: the fluctuation occurring during
    the transient period
    – Static hazard: glitch when the signal should
      be stable
    – Dynamic hazard: a glitch in transition
  • Due to the multiple converging paths of an
    output port
• E.g., static-hazard (sh=ab'+bc; a=c=1)

• E.g., dynamic hazard (a=c=d=1)

Dealing with hazards
• Some hazards can be eliminated in theory
• E.g.,

Eliminating glitches is very difficult in reality, and almost impossible for synthesis
• Multiple inputs can change simultaneously (e.g., 1111=>0000 in a counter)
• How to deal with it?
  Ignore glitches in the transient period and retrieve the data after the signal is stabilized

Delay sensitive design and its danger
• Boolean algebra
  – the theoretical model for digital design and most algorithms used in synthesis process
  – algebra deals with the stabilized signals
• Delay-sensitive design
  – Depend on the transient property (and delay) of the circuit
  – Difficult to design and analyze

• E.g., hazard elimination circuit: ac term is not needed
• E.g., edge detection circuit (pulse=a a')
• What's can go wrong:
  – E.g., pulse <= a and (not a);
  – During logic synthesis, the logic expressions will be rearranged and optimized.
  – During technology mapping, generic gates will be re-mapped
  – During placement & routing, wire delays may change
  – It is bad for testing verification
• If delay-sensitive design is really needed, it should be done manually, not by synthesis