This lecture is the 1st part of a set of 4 lectures aimed at tackling the problem of manual synthesis of both combinational and sequential circuits at the RT-level. The topic has been split in 4 lectures just for sake of manageability. This first lecture presents some basic approaches.

**Prerequisites**

- Modules 5, 6, and 7

**Homework**

- Students are recommended to try to solve the exercise by themselves, before looking at the proposed solutions.

**Further readings**

- No particular suggestion

**Outline**

- Tackling the complexity
- ROM-based synthesis
- K-map partitioning
- Synthesis by shift registers
Simple FSM formalism

The behavior of simple FSM's are usually specified resorting to “states” in order to implement Mealy or Moore machines.

This formalism is very powerful for small designs, but it becomes too complex when dealing with complex designs.

For example, imagine the complexity of designing a system requiring a memory, a counter, a comparator, and a timer, just using Moore or Mealy states.

Approaches

Several approaches are possible.

We shall focus on two of them:
- RT-Level Manual Synthesis (Lectures 8.4 – 8.7)
- System-Level HDL-Based (Module 11)

RT-level approach

Designing an RT-level circuit, means to connect together standard components (memories, registers, counters, etc…) in order to obtain a system that behaves as desired.

RT-Level Synthesis

User's Requirements

system
RT
logic
behavior
structure
physical
Impl

Approaches

Several approaches are possible, characterized by different applicability – cost trade-offs.

We shall consider the following ones:
- ROM-based synthesis
- Mux-based synthesis
- Synthesis by shift registers
- Synthesis by iterated basic cells
- Synthesis by functional partitioning

Major limitations

The approaches proposed in the sequel are:
- mainly
  - “manual” &
  - “intuitive”
  i.e., not supported by systematic design methodologies
- useful for simple designs involving standard functionalities, but
- very limited if we want to design complex custom systems with not standard functionalities.
Next steps

To overcome these limitations, the systematic approach presented in Module 11 needs to be adopted.

Outline

- Tackling the complexity
  => ROM-based synthesis
- K-map partitioning
- Synthesis by shift registers

Rom-based Synthesis

Any \(n\)-inputs / \(m\)-outputs combinational circuit

\[
\begin{array}{c}
\text{combinational} \\
\text{circuit}
\end{array}
\]

\(\begin{array}{c}
\text{PI} \\
\text{PO}
\end{array}\)

\(\begin{array}{c}
\text{m} \\
\text{m}
\end{array}\)

\[\text{can easily be implemented resorting to a } (2^n \times m) \text{ ROM.}\]

Examples

For sake of clarity, we shall consider some of the examples already seen in lecture 6.3.

Exercise # 06.31: Hexadecimal Threshold comparator

Design a combinational circuit with 1 output \(U\) and an input \(X\) (3 downto 0) that, when it receives on \(X\) an unsigned hexadecimal digit \(X\), provides, on \(U\), a logical value 1 iff:

\[X < 4 \text{ or } X > 8.\]
8.4 – RT-level Manual Synthesis (1)

**Solution**

*(ROM 16x1)*

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
</tr>
<tr>
<td>1010</td>
<td>1</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>1110</td>
<td>1</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
</tr>
</tbody>
</table>

**Solution**

*(ROM 16x3)*

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>000</td>
</tr>
<tr>
<td>0001</td>
<td>001</td>
</tr>
<tr>
<td>0010</td>
<td>010</td>
</tr>
<tr>
<td>0011</td>
<td>010</td>
</tr>
<tr>
<td>0100</td>
<td>011</td>
</tr>
<tr>
<td>0101</td>
<td>011</td>
</tr>
<tr>
<td>0110</td>
<td>011</td>
</tr>
<tr>
<td>0111</td>
<td>011</td>
</tr>
<tr>
<td>1000</td>
<td>011</td>
</tr>
<tr>
<td>1001</td>
<td>100</td>
</tr>
<tr>
<td>1010</td>
<td>100</td>
</tr>
<tr>
<td>1011</td>
<td>100</td>
</tr>
<tr>
<td>1100</td>
<td>100</td>
</tr>
<tr>
<td>1101</td>
<td>100</td>
</tr>
<tr>
<td>1110</td>
<td>100</td>
</tr>
<tr>
<td>1111</td>
<td>100</td>
</tr>
</tbody>
</table>

**Exercise # 06.09:**

*Upper approximation of sqrt*

Design a combinational circuit with a 4-bit input and a 3-bit output such that, when it receives on its input a hexadecimal digit X, provides, on its output:

\[ \text{[ } \text{sqrt}(X) \text{]} \]

---

**Look-up tables**

*ROM-based synthesis* is particularly suited for combinational circuits acting as look-up table, i.e., tables storing values to be randomly read.

---

**Outline**

- Tackling the complexity
- ROM-based synthesis
  - \( K \)-map partitioning
- Synthesis by shift registers

---

**K-map partitioning or MUX-based synthesis**

Such an approach stems from the Boole’s expansion theorem (slide # 55 of lecture 3.1):

every Boolean function \( f : \mathbb{B}^n \rightarrow \mathbb{B} \):

\[
f(x_1, x_2, \ldots, x_n) = x_1' \cdot f_0(0, x_2, \ldots, x_n) + x_1 \cdot f_1(1, x_2, \ldots, x_n)
\]

\[ \forall (x_1, x_2, \ldots, x_n) \in \mathbb{B} \]
8.4 – RT-level Manual Synthesis (1)

Basic idea

The expression

\[ f(x_1, x_2, \ldots, x_n) = x'_1 \cdot f_0 + x_1 \cdot f_1 \]

can obviously be interpreted as:

- If \( x_1 = 0 \) (i.e., \( x'_1 = 1 \))
  - then \( f = f_0 \)
  - else \( f = f_1 \)

which leads to the following implementation:

... and so on ...

The process can be easily iterated...

Example

Consider the following map:

<table>
<thead>
<tr>
<th>( a ) ( b ) ( c )</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Example

Consider the following map:

<table>
<thead>
<tr>
<th>( a ) ( b ) ( c )</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

It can be partitioned as follows:
Consider the following map:

It can be partitioned as follows:

**Example**

If $c = 0$ then:

If $c = 1$ then:

It can be partitioned as follows:

**Alternative implementation**

If $c = 0$ then $U = a'$

If $c = 1$ then $U = b'$

It can be partitioned as follows:

**Alternative implementation**

If $c = 0$ then $U = a'$

If $c = 1$ then $U = b'$

It can be partitioned as follows:
Exercise #06.11: Comparator for multiple codes

Design a comparator which, receiving in input:

- \( A(1 \text{ downto } 0) \) and \( B(1 \text{ downto } 0) \)
- a signal \( 2C/-U \text{M} \) specifying the coding used for
  \( A \) and \( B \) (1=2's complement, 0=unsigned magnitude)
provides 3 outputs \( A_{GT_B}, A_{EQ_B} \) and \( A_{LT_B} \) such that:

- \( A_{GT_B} = 1 \) iff \( A > B \)
- \( A_{EQ_B} = 1 \) iff \( A = B \)
- \( A_{LT_B} = 1 \) iff \( A < B \).
On a serial transmission line X, bits are transmitted synchronously w.r.t. a clock signal CLK, one bit per clock cycle. A circuit to be connected to the serial line is to be designed. It samples the line X at each clock cycle. Whenever the sequence ‘010’ has been received, it asserts its output signal Z for one clock cycle. A ‘0’ can be used, at a same time, as the last bit of a sequence and as the first bit of the next one.

**Question**

Is this solution equivalent to the one we obtained at the logic level?

**Answer**

NO: they differ in the “transient” period, i.e., during the first 3 clock cycles immediately following a reset:

- Reset: \( x = 1 \)
- \( x = 0 \)
On a serial transmission line X, bits are transmitted synchronously w.r.t. a clock signal CLK, one bit per clock cycle.

A circuit to be connected to the serial line is to be designed. It has an output U which is asserted whenever the last 4 values got in input form a palindrome string.
Solution

\[
\begin{align*}
\text{Solution} & \\
\text{Diagram} & \\
\text{Tree} & \\
\end{align*}
\]