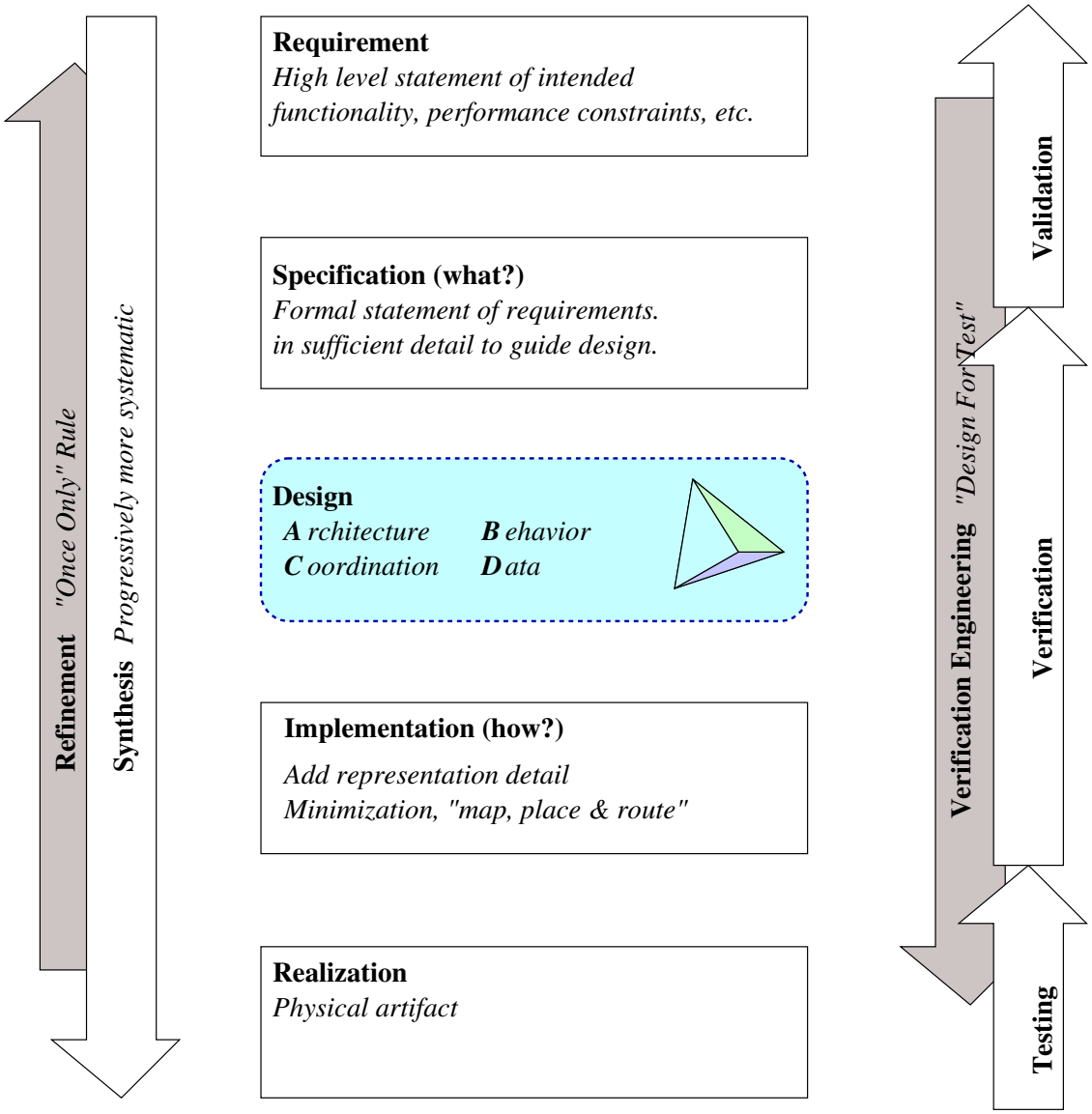
Design Overview



Example

CAUTION: *This is an illustration of design "flow." It is* not *an example of good design. Shortly, we will develop better and more systematic ways to synthesize implementations of this kind.*
(And anyway, much of what we are about to see is done automatically in the ISE design environment used in the Lab!)

Requirement.

Design Something and implement it using 74LS00 devices.

Specification.

Truth tables are a common form of specification at lower levels.

a	b	c	z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Design.

Synthesize a system of Boolean equations. Here it is "easy" to see that in the case that $a = 0$, z reduces to $b\,c$; and when $a = 1$, z reduces to $b + c$. So z can be expressed as

$$z = \overline{a}bc + a(b + c) \equiv (\neg a \wedge b \wedge c) \vee (a \wedge (b \vee c))$$

There is a direct correspondence between Boolean terms and logical expressions, both shown above. Boolean terms are more conducive to algebraic manipulation. The logical form more clearly shows how to implement this circuit, using one gate per logical operation—six gates in this case. Since we want to minimize the number of gates used (all else considered), we can try to algebraically "simplify" this initial implementation...

$$\begin{aligned}
& \overline{a}bc + a(b + c) \\
= \;& \overline{a}bc + ab + ac && \text{(distributivity)} \\
= \;& b(\overline{a}c + a) + ac && \text{(distributivity)} \\
= \;& b(c + a) + ac && \text{(absorbtion)}
\end{aligned}$$

This seems to be an improvement, assuming the algebra is correct. We should verify the result. I have a simple Scheme truth-table generator, whose output is shown below. The truth tables appear the same but *CAUTION*: the rows are listed in the wrong order ($[b, c, a]$ rather than $[a, b, c]$). The rows must be re-arranged in order to compare with the specification, the truth tables are still identical, so our optimization is correct.

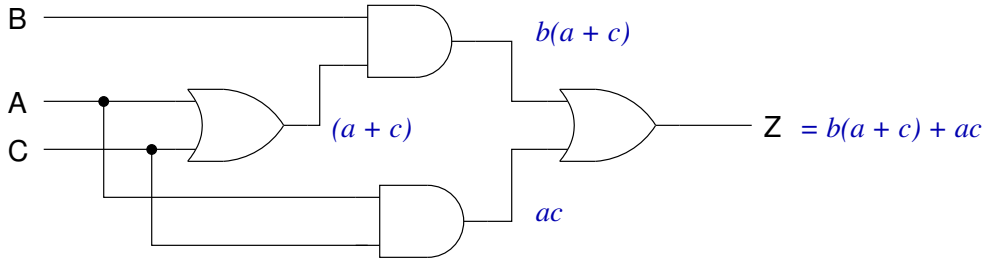
```
% scheme tt.ss
Chez Scheme Version 7.1
Copyright (c) 1985-2006 Cadence Research Systems

> (pft ((b & (c + a)) + (a & c)))

((b & (c + a)) + (a & c))   [abc]
-------------------------
  0 0  0 0 0   0  0 0 0     [000]
  0 0  0 1 1   0  1 0 0     [100]
  0 0  1 1 0   0  0 0 1     [001]
  0 0  1 1 1   1  1 1 1     [101]
  1 0  0 0 0   0  0 0 0     [010]
  1 1  0 1 1   1  1 0 0     [110]
  1 1  1 1 0   1  0 0 1     [011]
  1 1  1 1 1   1  1 1 1     [111]
```

```
((b & (c + a)) + (a & c))   [abc]
-------------------------
  0 0  0 0 0   0  0 0 0     [000]
  0 0  1 1 0   0  0 0 1     [001]
  1 0  0 0 0   0  0 0 0     [010]
  1 1  1 1 0   1  0 0 1     [011]
  0 0  0 1 1   0  1 0 0     [100]
  0 0  1 1 1   1  1 1 1     [101]
  1 1  0 1 1   1  1 0 0     [110]
  1 1  1 1 1   1  1 1 1     [111]
```

Maybe we can do better, but let's not try.

Implementation.

Only four operations, hence four *logical* gates, are needed. Experssing $b(c+a)+ac$ in schematic form, we get



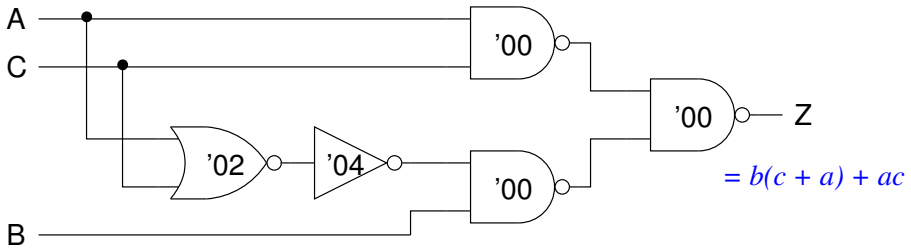
However, 74LS00 devices provide *nand* and *nor* gates, so we need to transforn this *and-or* implementation to a *nand-nor* one. A traditional (and laborious, hence wrong) way to do this is to use Boolean algebra, again, to derive something like:

$$
\begin{aligned}
& b \cdot (c+a) + (a \cdot c) \\
= \quad & \overline{\overline{b \cdot (c+a) + (a \cdot c)}} && \text{(double negation)} \\
= \quad & \overline{\overline{b \cdot (c+a)} \cdot \overline{(a \cdot c)}} && \text{(DeMorgan's Law)} \\
= \quad & (b \odot (c+a)) \odot (a \odot c) && \text{(Rewrite } \overline{x \cdot y} \text{ as } x \odot y\text{)} \\
= \quad & (b \odot \overline{\overline{(c+a)}}) \odot (a \odot c) && \text{(DeMorgan's Law)} \\
= \quad & (b \odot \overline{(c \oplus a)}) \odot (a \odot c) && \text{(Rewrite } \overline{x+y} \text{ as } x \oplus y\text{)}
\end{aligned}
$$

We should again verify our derivation, of course. This time we'll check it agains the computed truth table, not the specification. It is easier to check because the two tables use the same variable ordering, $[b, c, a]$.

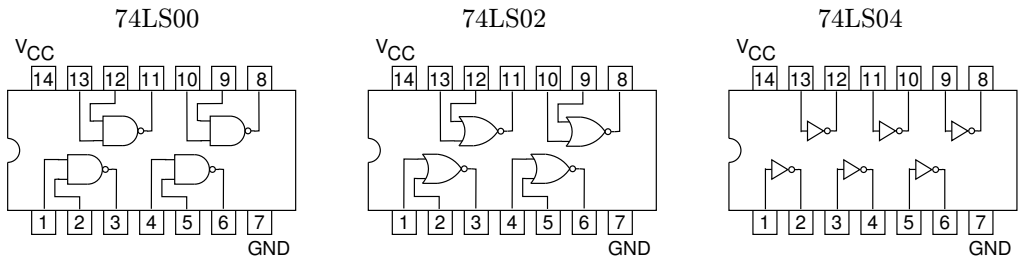
```
(~ ((~ (b & (~ (~ (c + a))))) & (~ (a & c))))
-----------------------------------------------
 0   1  0 0  0  1  0 0 0      1  1  0 0 0
 0   1  0 0  1  0  0 1 1      1  1  1 0 0
 0   1  0 0  1  0  1 1 0      1  1  0 0 1
 1   1  0 0  1  0  1 1 1      0  0  1 1 1
 0   1  1 0  0  1  0 0 0      1  1  0 0 0
 1   0  1 1  1  0  0 1 1      0  1  1 0 0
 1   0  1 1  1  0  1 1 0      0  1  0 0 1
 1   0  1 1  1  0  1 1 1      0  0  1 1 1
```

The *nand-nor* circuit becomes



Realization.

74LS00 devices contain 4–6 gates per chip. Here are the devices containing *nand* gates, *nor* gates, and *inverters*:



Generally, synthesizing a realization involves three problems:

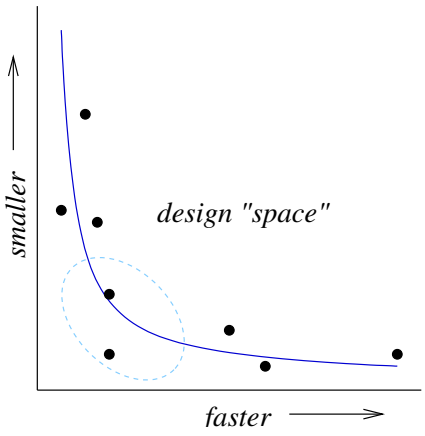
1. *Mapping:* assigning logical functions to physical devices. In this case, this means assigning each gate in the schematic to one in a device.

2. *Placement:* assigning each physical device to a physical (usually planar) location.

3. *Routing:* connecting the physical pins on the device to each other.

Each of these problems is of exponential complexity, NP-complete in the best case. Furthermore, the problems are interdependent: a near-optimal result requires that the all by solve simultaneously. CAD systems usually attack the problems in some order, using heuristics to abstractly model the other problems.

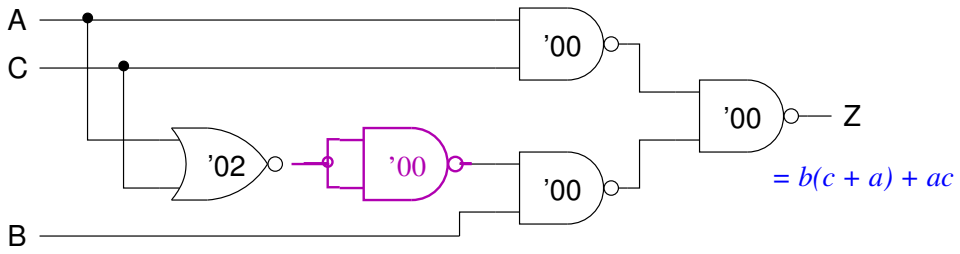
It is important to understand that "optimization" is multi-faceted. There is no truely optimum implementation. The most basic example in digital design is the so-called *speed-area* tradeoff: one can make an implementation faster by adding more gates. This leads to notion of a "design space" in which one is usually looking for an implementation instance that balances both concerns, somewhere in the "sweet spot" of the trade-off curve.



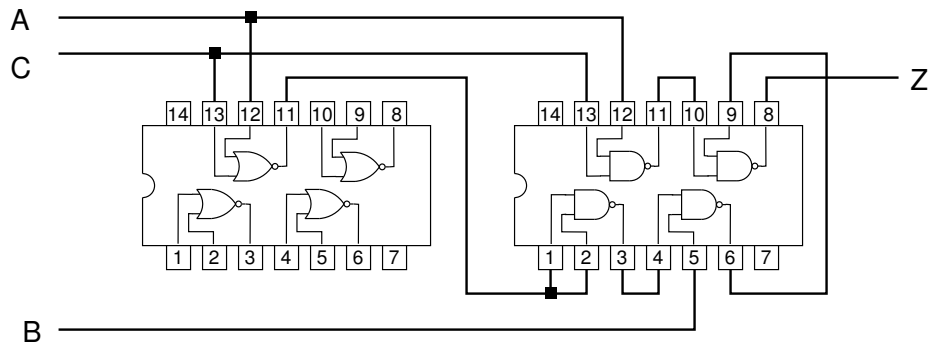
The design Requirements will dictate whether it is more important to have an implementation that is faster or smaller.

In our example, the implementation schematic contains three *nand*-gates, one *nor*-gate and one *inverter*, five gates in all. This is pretty good, relative to the four-logical-gate solution we came up with. However, a direct mapping would require three chips, which is not good at all! For one thing, it leaves nine gates unused. Minimizing the number of gates was the wrong goal: we really want to minimize the number of

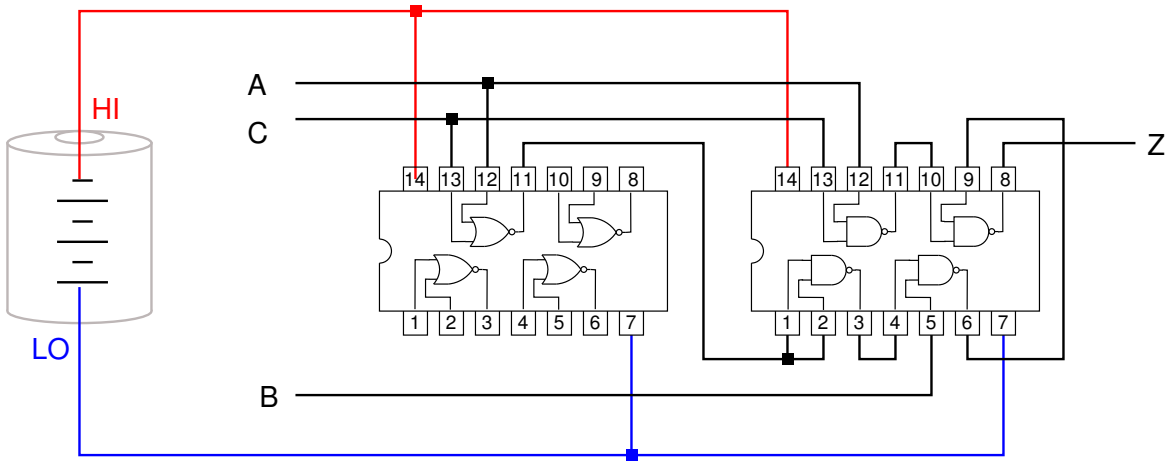
devices. One thing we might do is implement the *inverter* with the left-over *nand*-gate:



Now we need only two chips, a 74LS00 and a 74LS02. Let's leave it at that and map the four *nand*-gates clockwise, starting with the left-most gates in both the schematic and the 74LS00.



The only way that we can tell whether we've done everything right is to build the actual circuit and *test* it to determine if it satisfies the specification. Were you to do this for the physical layout shown above, the first thing you'd discover is that the circuit doesn't work at all! Physical circuits are active devices; they need a power supply to perform their functions.



"Think globally; act locally"

The previous example follows the design for the *carry-out* output of a one-bit adder, whose inputs are the two summand bits and a *carry-in* bit from a previous addition. In order to complete the design, we also need to generate a *sum* bit. pass along to the next-most-significant bit of addition.

a	b	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

We could develop a design for generating s in the same way, but a better implementation results if we implement both outputs at once. The c_o output (previously z) is the *majority* function. true when two or more of the inputs are true. The s output is the *parity* function, true when an odd number of inputs are true. Hence, we can develop equations

$$\begin{aligned} s &= a\,\overline{b}\,\overline{c} + \overline{a}\,b\,\overline{c} + \overline{a}\,\overline{b}\,c + a\,b\,c \\ c_o &= ab + ac + bc \end{aligned}$$

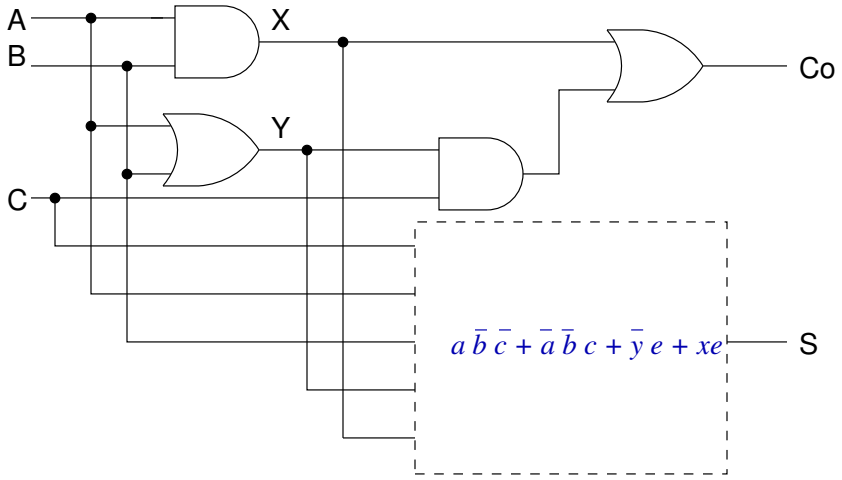
In designing an implementation, one thing to ask is whether the two output experssion have any subterms in common. For example, we can derive an equivalent system of equations:

$$\begin{aligned} s &= a\,\overline{b}\,\overline{c} + \overline{a}\,b\,\overline{c} + \overline{(a+b)}\,c + (a\,b)\,c \\ c_o &= a\,b + (a+b)\,c \end{aligned}$$

Identifying like terms, add two more equations to the system:

$$\begin{aligned} x &= a\,b \\ y &= a + b \\ s &= a\,\overline{b}\,\overline{c} + \overline{a}\,b\,\overline{c} + \overline{y}\,c + x\,c \\ c_o &= x + yc \end{aligned}$$

The term $a\,b$ can be implemented by one gate, whose output x is shared by two other gates in the system.



Epilogue

In 1978, Niklaus Wirth published a book entitled *Algorithms + Data Structures = Programs.* This seminal work proclaims the "Structured Programming Languages" movement and marks (historically at least) the dawn of Object Oriented Programming.

Though addressing the lowest level of digital design, the Example we have so painstakingly examined, exposes issues and concerns common to all levels of technology, whether based in software, or hardware, or both.

$$\textit{Architecture} + \textit{Behavior} + \textit{Coordination} + \textit{Data} = \textit{Systems}$$

A design *strategy* must balance all these aspects and understand their interplay. At the same time, the vast complexity of a design problem necessitates that design *tactics* be applied locally, within the conceptual structure of a particular aspect.

For an implementation to satisfy its specification, it must not only function correctly, but it must do so within the required performance constraints. True optimality is not a dominant concern. Meeting constraints is what drives the design process forward. Within those constraints, doing the simplest thing is paramont.