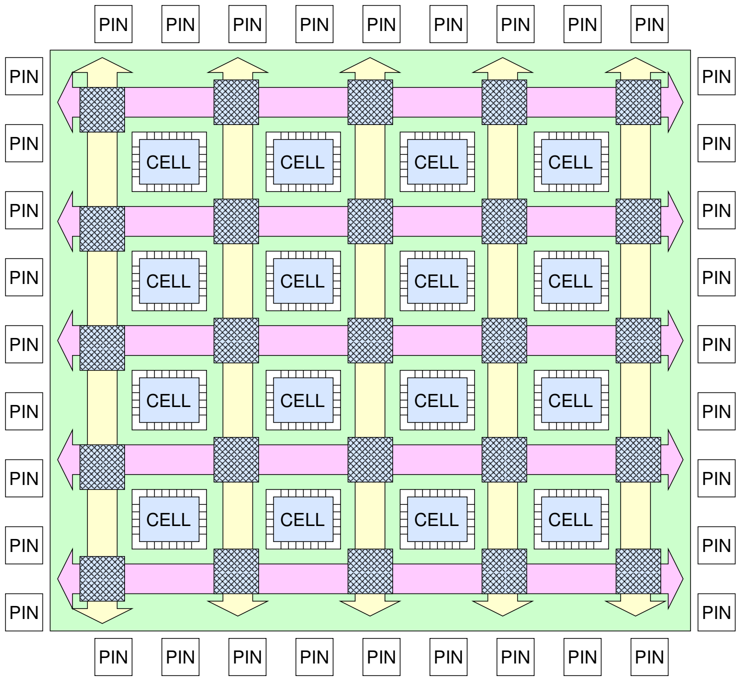
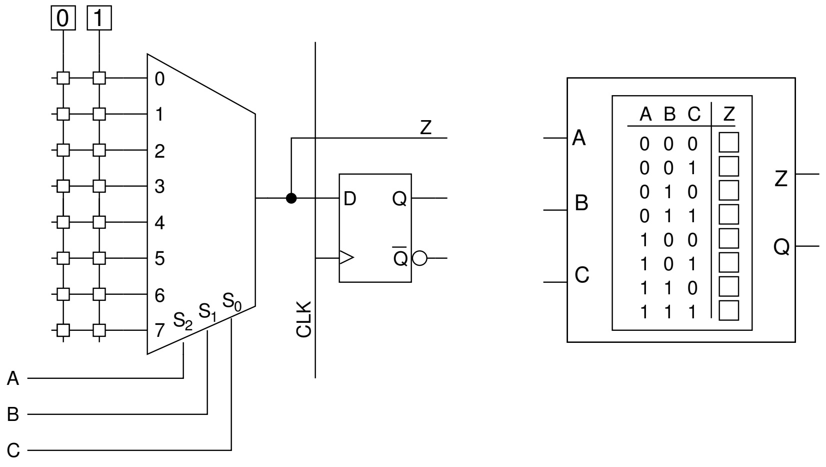
Field Programmable Gate Arrays

Field Programmable Gate Arrays (*FPGA*s) are flexible, programmable devices with a broad range of capabilities. Their basic structure consists of an array of universal, programmable logic *cells* embedded in a configurable *connection matrix*. Cells and their connections are determined by programmable settings using a data file generated by design software. The software presents the designer with various abstract views of the technology and is responsible for translating a design instance into a device configuration.



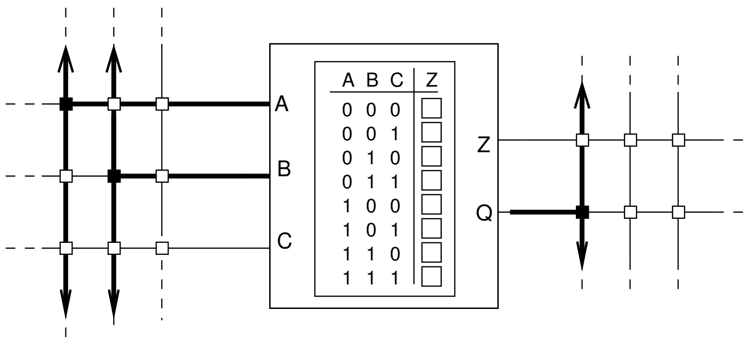
Logic cells vary in their complexity, but the idea is to provide a universal design element. The example developed here is simple. Our goal is to present the main ideas. Our cell consists of a single 8×3 selector composed with a D flip-flop



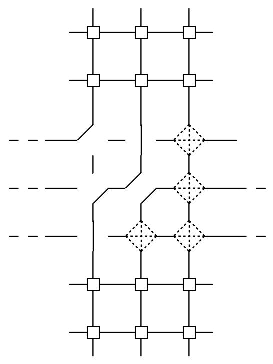
As we have already seen, a 3×8 selector can implement any 3-input logic gate by using the selection inputs for operands and hard-coding the truth table. The FPGA cells provide a progammable means of fixing a truth table for the cell. In some devices this is done with a once-only fuse-burning technology. In others, the truth table is implemented with a 8×1 bit *random access memory*, and so may be re-programmed with different data.

The selector's output is one output of the cell. This output is also fed into a clocked D flip-flop providing two additional outputs from the cell. Assume all clock inputs are common (many devices provide more than one clock network), and the clocking signal is carefully distributed to assure that there are no drift or noise problems.

Cells can be connected the surrounding connection matrix by using fuse-based, or memory controlled connections. In our example, cell inputs and outputs are connected to vertical signal traces in this fashion.



The vertical signal traces are connected to horizontal traces by a configurable switch, which, in our example, can be programmed to tie together any combination of its sources.



A small FPGA configuration is depicted in Fig. 1. It contains six cells.

FPGA Synthesis

FPGAs are complex enough that it is rare to see them configured manually. A *computer-aided design* (*CAD*) environment is usually involved in translating the designer's intent into an FPGA configuration. The designer's input into the CAD environment may take several forms.

- *Schematic.* Many CAD environment come with a graphical editor for schematic design entry. The resulting graphical data structure is linked to a library of elementary components representing a generic network of logical elements.

- *Netlists.* The network of elementary components, or *netlist*, is the standard bridge between the design tool set and the collection of routines need for translation into configuration data for the FPGA.

- *Hardware Description Languages.* Since the implementation process is now the responsibility of intermediate tools, any means of producing a netlist may be used. These means may include design description languages that correspond to programming languages used in software development. A hardware description language typically has a dialect for describing netlists in a hierarchical fashion, with features for data declaration and parameterization that make design development more reminiscent of programming.

Once a design has been entered and compiled into a standard form, another collection of processes perform the translation into FPGA configuration data. This may involve the following steps, individually or in combination.

- *Minimization.* Automated optimizers may are used to minimize the logic and perform other transformations.

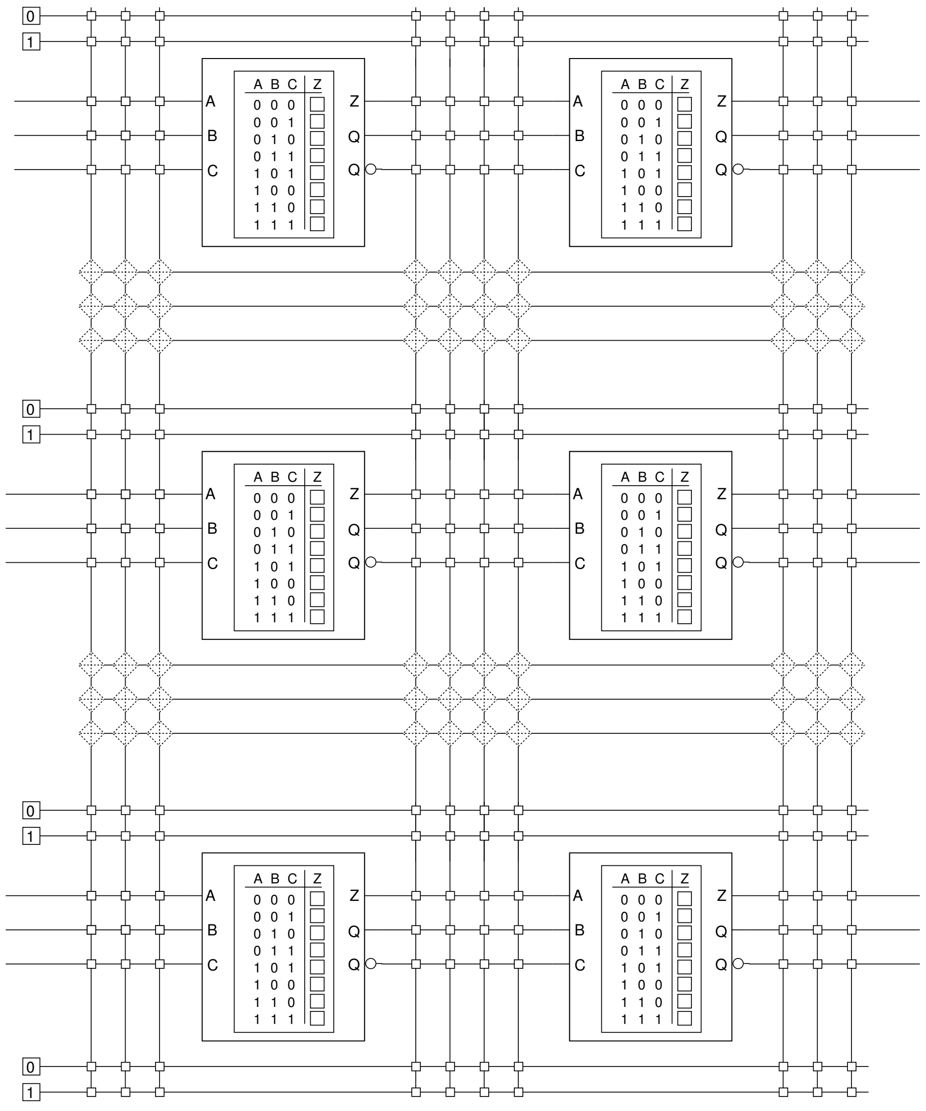


Figure 1: A simplified FPGA architecture.

- *Mapping.* The netlist is broken up into units that correspond to cells. In our example, a cell can implement a combinational logic gate, a simple flip-flop, or a *gated flip-flop.* Thus an early step in design synthesis is to re-generate the design in terms of these design unit.

- *Placement* Once the design is expessed as a network of cells, a *placement* routine assigns each cell to a physical location in the FPGA. The optimization problem in this phase is to shorten the distance between connected cells and I/O pins.

- *Routing* With the cell positions fixed, a *routing* routine determines how the connection matrix is programmed to connect cells and pins.

Each of the translation problems described above is computationally hard to perform. In combination, the complexity of automated synthesis is astronomically high. For this reason, the synthesis routines use heuristic decision making to seek good solutions to their parts of the implementation problem—and usually do a better job than a human can do, making far fewer mistakes.

In spite of all this automation, human decision making is still crucial in large implementation problems. The designer interacts with the CAD environment by setting *constraints*, or limits on how slow or large the resulting implementation is allowed to be. If the tools can meet these contraints, the synthesis process has been successful. Otherwise, the designer may intervene by: ways:

- Changing the synthesis parameters to alter decision heuristics, the time allotted for minimization, placement, routing, etc.

- Changing the constraints, relaxing the timing requirments, perhaps.

- Refining the source design, doing manual optimization, architecture reorganization, and so forth, to help the tools along.

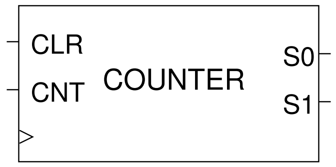
- Revising the design, by decomposing in a different fashion, using different abstractions, or perhaps a different algorithm.

- "Tweaking" the result, by making implementation modification such as:
  - Editing intermediate data.
  - Manually refining the target configuration. CAD environments may provide a *floorplan editor* for this purpose.

Although the last of these options may seem unwise, it is fairly common practice. All of these forms of intervention require intimate knowledge of the CAD algorithms and target technology.

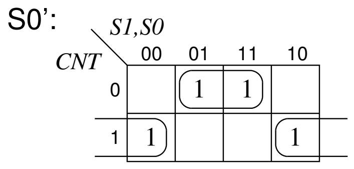
An example

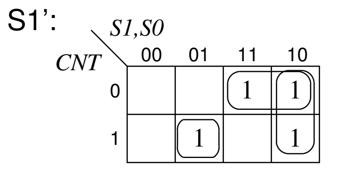
Let us walk through a small example to illustrate the synthesis process. We shall design and implement a two-bit counter with *clear* and *count* control inputs.



- *Minimization.* We can implement the *clear* mode with and *and* gate feeding into a DFF. Since we are doing this problem by hand, it is worthwhile minimizing the next-state values of the two counter bits, S_1 and S_0.

CNT	S_1	S_0	S_1'	S_0'
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0





$$
\begin{aligned}
S0 &= \overline{CNT}\cdot S0 + CNT\cdot\overline{SO}\\
S1 &= \overline{CNT}\cdot S1 + S1\cdot\overline{S0} + CNT\cdot\overline{S1}\cdot S0\\
&= S1\cdot(\overline{CNT}+\overline{S0}) + CNT\cdot\overline{S1}\cdot S0
\end{aligned}
$$

- *Mapping.* Figure 2 is a schematic for the counter. Our next task is to assign the gates of this circuit onto FPGA cells. We might begin by partitioning each flip-flop with its closest input gates. After that, any combination of gates depending on at most three input signals can be assigned to a cell. Left-over combinational blocks are then mapped to cells as shown in Fig. 3. Dealing with shared outputs is what makes this problem, by itself, computationally hard. Of course, there is also the fact that a different minimization of the logic can change the partitioning.

- *Placement.* Fig. 4 shows a placement of the four cells into which our counter design was mapped. The strategy that was used for this placement

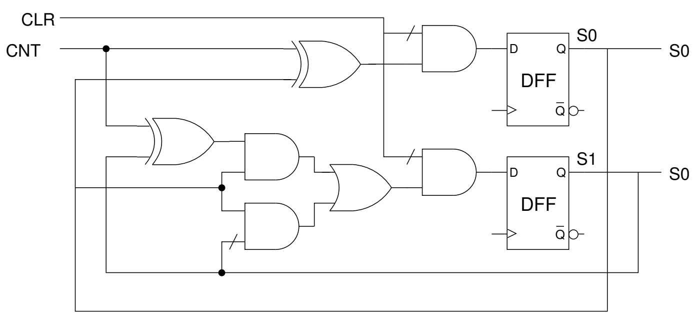


Figure 2: Schematic of the example counter circuit.

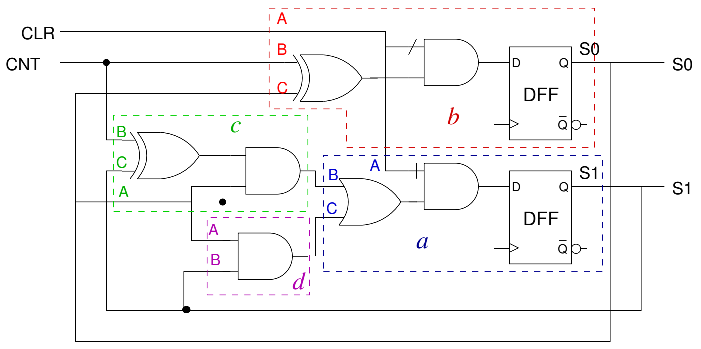


Figure 3: Counter circuit mapped into FPGA cells.

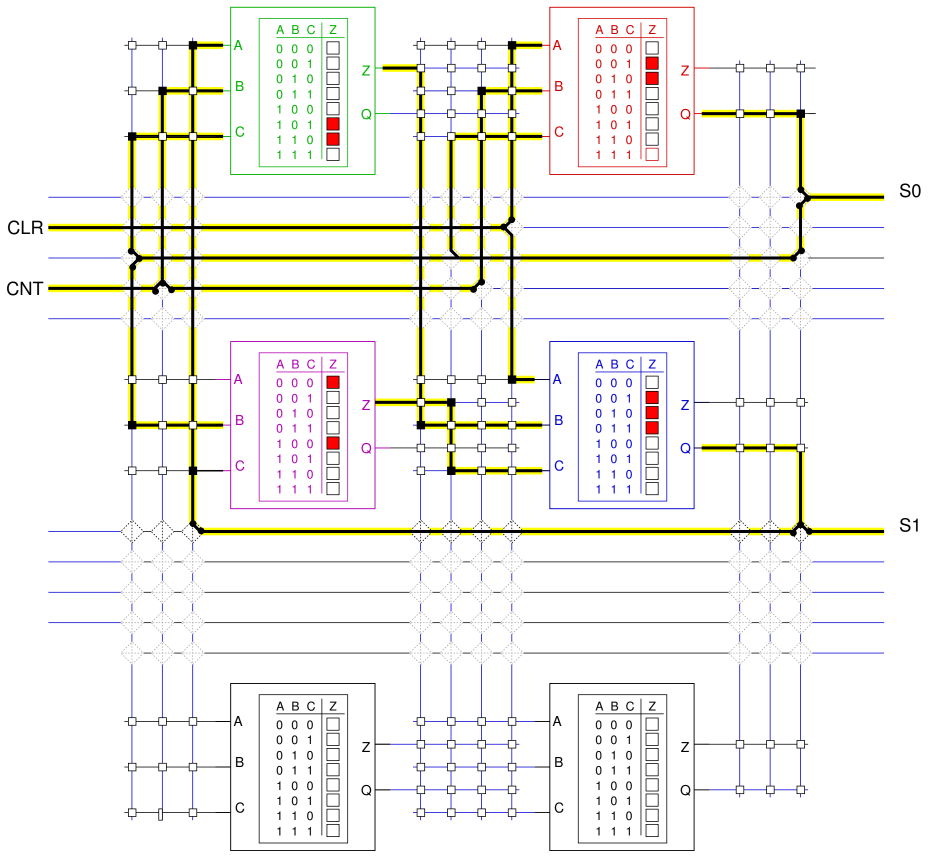


Figure 4: Counter circuit placed and routed on a 2 × 3 cell FPGA.

was to place the flip-flops first, in the right-hand column at opposite ends—the idea was to make the cell outputs close to the periphery for connection to pins and proceed backward through the gate network placing gates in the nearest available cell.

- *Routing.* Finally, the cells are connected according to the schematic. Since only four of the six available cells are used, finding available paths for all the connections was not difficult. The resulting implementation might be improved by rearranging the cells to reduce path lengths, but doing this may increase congestion in the routing.

Design synthesis complexity is made still more complex by the fact that there are multiple aspects by which one measures the quality of the result. Speed and area (as measured by the number of cells) is the most prevalent of these trade-offs. It is usually the case that to obtaining a faster implementation takes more cells, and conversely, an economical implementation is substantially slower. In other applications, the strategies used to optimize may range widely in their impact on power consumption, and so forth.

Our example illustrates the fact that most synthesis processes solve a sequence of problems that are individually complex but also mutually dependent. A different minimization might improve the mapping by reducing the number of cells needed; a different mapping might result in better placement; a different placement in better routing.

Especially in small problem instances, a human may be able to exploit a more global design strategy to find a "clever" solution. But in practice, the situation is much different, with larger designs, many more cells and routing paths, it becomes quite intractable to implement designs without the help of synthesis programs.