Digital Electronics 2
Hierarchical Design and Programmable Logic

Dr. T.J.H. Kluter

March 7, 2012
## Contents

1 **Hierarchical Design**  
   1.1 The Divide and Conquer approach ........................................ 1  
   1.2 Communication ................................................................. 2  
   1.3 Putting it all together .......................................................... 3  

2 **Programmable Logic**  
   2.1 Programmable Array Logic ..................................................... 5  
   2.2 Gate Array Logic ................................................................. 8  
   2.3 Programmable Logic Device ..................................................... 11  
   2.4 Complex Programmable Logic Device ....................................... 13  
   2.5 Field Programmable Gate Array ............................................. 13
This chapter describes the approach on how to solve a problem by dividing it into sub-problems until a level is reached at which we can understand and solve each of the problems. This approach is called the *divide and conquer* approach and can be used not only for digital problem, but for any other problem, like programming problems, mechanical problems, etc.

### 1.1 The Divide and Conquer approach

In the divide and conquer approach we try to divide our **big problem** into smaller problems until we reach a point at which we understand the sub problems and can solve them (see Figure 1.1). The staring problem is referred to as the *Root*, and the sub-problems that we can solve as the *Leaves*. The first phase of the divide and conquer approach is a *Top Down* design. After the identification of the *leaves* of the problem the realization of this leaves can be put together in the second phase to form the solution of the initial problem. This second phase is a *Bottom Up* design.

Unfortunately, most of the time the *leaves* of the problem have to communicate

**Figure 1.1**: The divide and conquer approach: The problem (*root*) is in the first phase (*Top Down*) divided into smaller sub-problems up to a level that is understandable and implementable (*leaves*). In the second phase (*Bottom Up*) all the realizations of the sub-problems are put together to form the solution for our initial **Big Problem**.
Hierarchical Design

with each-other. This communication needs to be identified during phase 1 of the divide and conquer approach.

1.2 Communication

The communication can be divided in two big groups:

- The first group represent sets of bits on which operations are performed. These sets of bits can be for example integers, and the operations can be addition, multiplications, etc. This group of communication is often referred to as data-path.

- The second group represents sets of bits that control operations. These sets of bits say for example if a light needs to be on, a certain operations needs to be done, etc. This second group of communication is often referred to as control-path.

In this course we will focus on the second group, the control-path. The control-path can be sub-divided into two communication patterns, namely:

1. Master-Slave. In the master-slave communication the master imposes the slave when to perform a given action. One could say that the master “kicks the slave”. The simplest form of the master-slave communication pattern is an enable signal. The enable is high when the slave has to perform its action and low when the slave has to do nothing.

2. Master-Master, Slave-Slave, or Slave-Master. In this type of communication one partner asks the other one if it is allowed to perform an action. This request is performed by a request signal. After the request, the other partner receiving the request can either acknowledge the request by raising the acknowledge signal, or can deny the request by raising the nack signal. This form of communication is often referred to as handshaking.

In real designs the control communication is often more complex and formulated in so called protocols.
1.3 Putting it all together

To demonstrate the divide and conquer approach together with the Master-Slave communication pattern we are going to implement a blinking light. For this blinking light is given that the frequency of operation is 500 Hz. The light itself should blink at 1 Hz and should be on and off 0.5 seconds respectively.
Programmable Logic

Programmable logic is a class of ASICs that allows to implement logic functions without the requirement of implementing each and every transistor required. Basically the family of programmable logic leverages the digital design to the level of logic functions. Similar one could compare the design of programs in a programming language like Assembly, rather than coding each single bit.

2.1 Programmable Array Logic

The first member of the family of programmable logic is arguably the Programmable Array Logic (PAL). The architecture of the PAL is based on the fact that each logic function can be written in the Sum of Product-form, e.g., \( Y = \overline{A} \cdot \overline{B} + \overline{A} \cdot B \). The architecture of the PAL, therefore, consists of an array of AND-operations followed by an array of OR-operations. To be able to implement any logic function, the AND operations are connected with fuses to all inputs and their inverse. The architecture of the PAL is shown in Figure 2.1.

To program the logic functions in a PAL, the fuses that represent unwanted connections have to be removed (“blown up”). This “programming” is performed by putting the PAL into a programmer and literally melt the fuses by applying a high voltage (approx. 12V-21V) over the fuses to be removed. This makes that a PAL is an One-Time-Programmable (OTP) device, as fuses that are “blown up” cannot be restored. An example of a programmed PAL is shown in Figure 2.2.

There is something special about the architecture of the PAL shown in Figure 2.1. Assume we want to implement the XOR function: \( Y = A \cdot \overline{B} + \overline{A} \cdot B \). This function only requires two AND-operations; however, the macro-cell shown in Figure 2.1 provides four AND-operations. One would say that this function cannot be implemented; however, we can rewrite the function as:

\[
Y = A \cdot \overline{B} + \overline{A} \cdot B + A \cdot \overline{A} + A \cdot \overline{A} = A \cdot \overline{B} + \overline{A} \cdot B + 0 + 0.
\]

A great advantage of the PAL is that the delay from any input (\( A, B, C, D, \ldots \)) to any output (\( X, Y, \ldots \)) is fixed and represented by \( t_{io} \).

In summary Table 2.1 lists the advantages and disadvantages of the PAL.
Figure 2.1: The basic architecture of a PAL. Each output is generated by a macro-cell. One macro-cell consists of a fuse-array, AND-operations, and OR-operations.
Figure 2.2: The logic functions are implemented in the PAL by “blowing up” (shown in red) the fuses that represent unwanted connections. **Question:** Which logic functions are represented in this programmed PAL?
## 2.2 Gate Array Logic

To overcome the main disadvantage of the PAL, namely its One-Time-Programmability (OTP), Lattice Semiconductors introduced the *Gate Array Logic (GAL)*. The main difference between the GAL and the PAL lies in the Fuse-Array shown in Figure 2.1. In the GAL the fuses are replaced by *Floating Gate Transistors (FGMOS)*, shown in Figure 2.3.

The FGMOS, shown on the top of Figure 2.4, is a storage device that can store a charge on its isolated floating gate. This charge retains on this floating gate for approx. 10-20 years. The functionality of the FGMOS can be explained by separating the program- and normal operation.

In the program operation, the control-gate and bulk interface are used to put or to remove charge from the floating gate, shown at the left-bottom of Figure 2.4. In normal operation, the control gate is left floating, and the charge on the floating gate determines the operation of the FGMOS. If the floating gate is charged, an n-channel is formed between the Source and Drain of the FGMOS, allowing current to flow from Drain to Source. In this case the FGMOS acts as a “short-circuit” between Source and Drain. If the floating gate does not contain any charge, the Source is isolated from the Drain. In this case the FGMOS acts as an “open-

![Floating Gate Transistor](image)

**Figure 2.3:** In the GAL the fuses are replaced by Floating Gate Transistors making GALs In-System-Programmable (ISP).

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low energy consumption</td>
<td>One-Time-Programmable (OTP)</td>
</tr>
<tr>
<td>Fixed input-to-output delay (t_{io})</td>
<td>Only combinational functions</td>
</tr>
<tr>
<td>Any combinational function possible</td>
<td>Limited in number of macro-cells</td>
</tr>
<tr>
<td>Non-volatile</td>
<td></td>
</tr>
</tbody>
</table>

*Table 2.1:* The list of advantages and disadvantages of the PAL.
The charged floating gate creates an n−channel between Source and Drain hence a "short-circuit"

Electrons are tunneled from floating gate

Normal Operation: The charged floating gate creates an n−channel between Source and Drain hence a "short-circuit"

There is no charge on any of the gates; therefore, there is an isolation between Source and Drain

Electrons are tunneled toward floating gate

Programming: There is no charge on any of the gates; therefore, there is an isolation between Source and Drain

Due to physical effects the FGMOS can only be programmed and/or erased for approx. 10,000 times. After this number of program/erase cycles a short circuit between the floating gate and the control gate/bulk can occur.

As the FGMOSs can be electrical programmed/erased, they provide not only the ability to be reprogrammed multiple times, they can even be reprogrammed in system; this is called In-System-Programmability (ISP). For the ISP the GAL is extended with ISP-logic. The complete overview of the GAL is shown in Figure 2.5.

Figure 2.4: Simplified program and erase operation of the Floating Gate Transistor.
Figure 2.5: The GAL provides the same functionality as the PAL; however, it provides more flexibility due to its In-System-Programmability (ISP).
Programmable Logic Device

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low energy consumption</td>
<td>Data retention approx. 10-20 years</td>
</tr>
<tr>
<td>Fixed input-to-output delay ($t_{io}$)</td>
<td>Only combinational functions</td>
</tr>
<tr>
<td>Any combinational function possible</td>
<td>Limited in number of macro-cells</td>
</tr>
<tr>
<td>Non-volatile</td>
<td>Only approx. 10,000 program cycles</td>
</tr>
<tr>
<td>In-System-Programmable (ISP)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: The list of advantages and disadvantages of the GAL.

In summary Table 2.2 lists the advantages and disadvantages of the GAL.

2.3 Programmable Logic Device

One of the main shortcomings of both the GAL and the PAL is that they can only implement combinational logic function. To implement for example a Finite State Machine, external components need to be added to be able to implement a sequential logic function. This aspect has lead to the appearance of the Programmable Logic Device (PLD). In the PLD the macro cells are extended with a memory element in form of a D-type flipflop. To be able to implement both combinational and sequential logic, the output of the macro cell can be connected either to the output of the OR-functions, or to the output of the D-flipflop by means of two FGMOSs. Furthermore, to be able to implement a Finite State Machine in the PLD, the output of the D-flipflop is feed-back to the inputs of the AND-Arrays. The architecture of the PLD is shown in Figure 2.6.

In summary Table 2.3 lists the advantages and disadvantages of the PLD.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low energy consumption</td>
<td>Data retention approx. 10-20 years</td>
</tr>
<tr>
<td>Fixed input-to-output delay ($t_{io}$)</td>
<td>Limited in number of macro-cells</td>
</tr>
<tr>
<td>Any logic function possible</td>
<td>Only approx. 10,000 program cycles</td>
</tr>
<tr>
<td>Non-volatile</td>
<td></td>
</tr>
<tr>
<td>In-System-Programmable (ISP)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: The list of advantages and disadvantages of the PLD.
Figure 2.6: The PLD adds memory to the macro-cell in form of a D-type flipflop. By adding these memory elements the PLD provides the means to implement sequential and combinational logic.
2.4 Complex Programmable Logic Device

As the complexity of digital circuits continued to grow the “small” PLDs were not any more able to contain all the functionality. This has lead to the introduction of the Complex Programmable Logic Device (CPLD). In the CPLD the architecture has changed, such that the inputs and outputs have been separated from the macro-cell (Note: in all the previous devices each output was directly connected to the output of a macro cell. Similar all the device inputs were connected to the inputs of the macro cell). The CPLD provides the connections by a routing array. The device inputs, outputs and macro cells are all connected to the routing array. The routing array consists of a so called cross-bar that allows to make any connection from a given input to any of the arrays outputs. These connections are made by FGMOSs shown in Figure 2.7.

Figure 2.7 shows the simplified block diagram of the CPLD, and Table 2.4 lists the advantages and disadvantages of the CPLD.

![Routing Array Diagram](image)

**Figure 2.7:** Simplified diagram of a 4-input 8-output routing array. The FGMOSs can be used to connect any of the inputs to any of the outputs. The number of FGMOSs required is the product of the number of inputs with the number of outputs.

2.5 Field Programmable Gate Array

The scalability of the CPLD is limited by its routing array and connections (read delays). Furthermore, for certain applications the limited number of program cycles renders the CPLD unusable. This has arguably initiated an alternative
Figure 2.8: Simplified block diagram of the CPLD. Each Macro Cell contains the logic shown on the left. All connections are made in the programmable routing array (see Figure 2.7).
Field Programmable Gate Array

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low energy consumption</td>
<td>Data retention approx. 10-20 years</td>
</tr>
<tr>
<td>Fixed input-to-output delay ( t_{io} )</td>
<td>Only approx. 10,000 program cycles</td>
</tr>
<tr>
<td>Any logic function possible</td>
<td></td>
</tr>
<tr>
<td>Non-volatile</td>
<td></td>
</tr>
<tr>
<td>In-System-Programmable (ISP)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: The list of advantages and disadvantages of the CPLD.

volatile architecture called *Field Programmable Gate Array (FPGA)*. The FPGA, shown in Figure 2.9, bears many similarities with the CPLD; however, the big routing array found in the CPLD has been split-up in several small ones. Furthermore, the macro-cells found in the CPLD have been replaced by *Look Up Tables (LUTs)*.

The splitting-up of the routing arrays has the advantage that the FPGA is very scalable—indeed, the combination of routing array and LUT can be replicated as much in the x- and y- directions. However, the disadvantage is that the FPGA loses its predictability on input-to-output delay. The delay between an input and an output is depended on (1) the number of LUTs used for the logic function and (2) the number of routing arrays used to make the required connections. Figure 2.10 demonstrates the implementation of a XOR function on a FPGA.

As stated before, the routing in the FPGA is done by routing arrays. These routing arrays are very similar to those of the ones found in the CPLD. Contrary to the CPLD, the FPGA does not use the FGMOSs to make the connections, but it uses pass-gates, shown in Figure 2.11. In the CPLD the “connection information” is stored as a charge on the floating gate of the FGMOS; the FPGA uses a volatile memory element, represented by the D-flip-flops on the left of Figure 2.11. A memory element containing a 0 represents a connection and a 1 forces an isolation. The FPGA has two views, namely (1) the programming view that contains all the memory elements determining its functionality, shown to the left of Figure 2.11, and (2) the functional view presenting the logic functions implemented on the FPGA fabric, shown in Figure 2.10 and on the right of Figure 2.11.

Similar to the routing array, the LUT also has the programming and functional view, shown in Figure 2.12. Figure 2.12 also shows the main difference between the macro-cell as used in the CPLD and the LUT. The macro-cell is
Programmable Logic

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time re-programmable</td>
<td>Non-zero start-up time $T_{\text{conf}}$</td>
</tr>
<tr>
<td>Any logic function possible</td>
<td>Respectable energy consumption</td>
</tr>
<tr>
<td>Scalable architecture</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: The list of advantages and disadvantages of the CPLD.

Based on a hardware implementation of the *Sum-Of-Product* form. The LUT is based on a hardware implementation of the *Truthtable*. Each entry in the *Truthtable* corresponds to one memory element in the configuration memory, shown in Figure 2.13. By the use of the 16-input multiplexor the corresponding truth-table-entry is selected and put on the output. Furthermore, the LUT also includes a D-flipflop to be able to implement sequential functions. Note that the D-flipflop contained in the LUT can be used for logic functions, where as those in the configuration logic are used to “store” the functionality.

When inspection the programming view of the FPGA, we can see that it represents one big shift-register. This shift-register needs to be loaded with the configuration (also sometimes referred to as *bit-file*), as the FPGA loses this information when the power is turned off. This loading of the configuration requires some time $T_{\text{conf}}$. During this configuration time, the FPGA is not able to function. After this configuration time the FPGA is ready to perform the logic functions. In summary Table 2.5 lists the advantages and disadvantages of the FPGA.
Figure 2.9: Simplified block diagram of the FPGA. The FPGA is based on a set of a routing array with a LUT that is replicated multiple times inside the architecture.

Figure 2.10: A XOR function implemented on an FPGA. The input-output delay is dependent on (1) the number of LUTs used and (2) the number of routing arrays visited to make the required connections.
Figure 2.11: Simplified block diagram of the routing array. The routing array has two views, namely (1) the programming view (on the left) that is configured prior to its normal operation and (2) the logic view (to the right) that is used in normal operation.
Field Programmable Gate Array

**Figure 2.12:** Simplified block diagram of the LUT. The LUT has two views, namely (1) the programming view (on the left) that is configured prior to its normal operation and (2) the logic view (to the right) that is used in normal operation.

**Figure 2.13:** In the LUT the *Truth table* view is used for mapping a logic function onto it. Each entry in the truth-table corresponds to one bit in the configuration logic. **Question:** How do I implement a logic function with five inputs?