Programmable logic

Geoff Bostock looks at the steps involved in designing field-programmable logic arrays.

The one thing that all the devices described in these articles have in common is that they are all logic devices. Because of this, they all share a common approach to the design process.

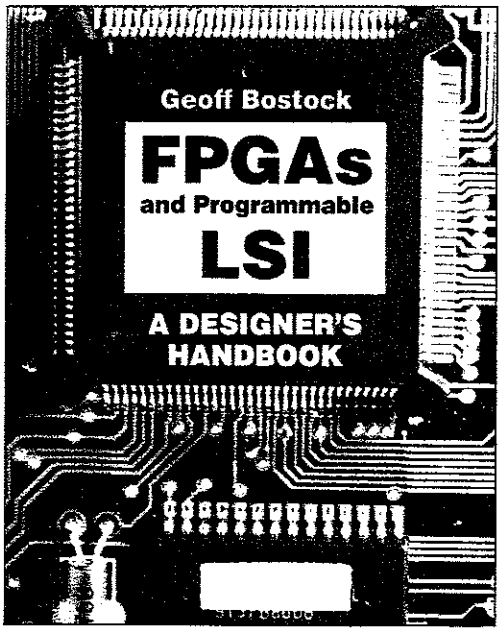
In principle, any logic circuit will fit into any fpga within the restraints of logic content and connectivity. Crudely, as long as there are enough gates and enough i/o lines, the choice of fpga does not affect the way in which the logic is defined. In practice this is not quite true. I will explain dedicated approaches to the various families in a later article. This article cover the general points of design.

There are only two general ways to design logic; the required function may be described in terms of some written language, or it may be drawn in some symbolic manner. Written methods include logic equations, state equations and hardware description langnages; symbolic descriptiom are covered by circuit diagrams and state diagrams.

All design packages make use of one or more of these categories of input to define the logic. Once the logic has been defined it is usually not dependent on any one target device or architecture, so this may be looked on as just the first stage in a design. The second stage is to ensure that the defined logic does the job which it is intended to do. This is achieved by simulating the design; that is, applying inputs to a software model of the design and checking that the outputs are as expected.

At this stage the target device can be considered. A translation from the general logic definition to specific architectural units is undertaken; abstract logic is mapped onto the physical components of the target device and

This article is derived from Geoff Bostock's new book 'FPGAs and programmable LSI – a designer's handbook'. The work covers designing FPGAs, large PAL structures, RAM and antifuse-based FPGAs and FPGA selection. Comprising 215 pages, this book is available by sending a postal order or cheque with a request for the book to Electronics World, Quadrant House, The Quadrant, Sutton, Surrey SM2 5AS. The fully-inclusive price is £27.50 UK, £30 Europe or £33 rest of world. Alternatively, fax your full credit card details and address on 0181 652 8956 or e-mail jackie.lowe@rbp.co.uk.



Geoff Bostock runs his own FPGA/PLD Design Consultancy, and may be contacted on 01380 828241, or by e-mail at geoff.bostock@zetnet.co.uk

```
INF = I3 & I2 & I1 & I0
IN7 = !I3 & I2 & I1 & I0
IN6 = !I3 & I2 & I1 & !I0
IN4 = !I3 & I2 & !I1 & !I0
IN1 = !I3 & !I2 & !I1 &I0
```

If you are designing for a device with D-type bistable devices, the state transitions must be defined by noting which bits have to be set high for each state transition, as follows:

```
Q3.D = S7 & INF & !DOOR; transition S7 to S8
# S8 & !DOOR; hold in S8
# S0 & !(IN6 # INF); transition SO to S9
# S2 & !(IN7 # INF); transition S2 to S9
# S4 & !(IN1 # INF); transition S4 to S9
# S6 & !(IN4 # INF); transition S6 to S
# S9 & !ALARM; hold in S9
Q2.D = S3 & INF; transition S3 to S4
# S4 & INF; hold in S4
# S4 & IN1; transition S4 to S5, etc.
```

The outputs are simply defined by:

```
UNLOCK = S7;
SOUND = S9;
```

The notation 'Q3.D' implies that this function is applied to the D-input of the bistable device driving the Q_3 output. At the active clock edge, the output will be set high if the function is true.

From the first few lines, it is apparent that equations do not give a transparent view of the function being implemented. However, by considering each transition it is possible to generate a set of equations which, when mapped onto the device, will produce a working part. The logic compiler minimises each equation and, if there are enough product terms driving each bistable device, the mapping will be successful.

The next section shows how state machines can be specified in a way which is directly associated with their function.

Simulating the logic design

The final stage of a logic design is usually a simulation. This has two functions; it checks that the logic will operate as intended, and it produces a set of test vectors for performing functional tests on the device after it has been programmed. Simulation can be defined by means of a text-based entry, or with a truth table, in most logic compilers.

I will show how the door lock can be simulated with an example of each method. The PALASM format would yield the following:

```
SIMULATION; keyword for the simulation
   segment
TRACE_ON I3 I2 I1 I0 DOOR ALARM Q3 Q2 Q1 Q0
   RESET CLK ;defining the signals we wish
   to view on the simulation output. Reset
   is added to initialise the state machine
SETF I3 I2 I1 I0/DOOR/ALARM RESET; initial
   values of the inputs at reset
SETF/RESET; release the reset condition
CLOCKF CLK; clock the state machine
CHECK/Q3/Q2/Q1/Q0; check that it is state
   '0' (an alternative notation is CHECK S0)
SETF/I3 I2 I1/I0 ;put '6' onto the inputs
CLOCKF CLK; clock it again
CHECK S1
CLOCKF CLK
CHECK S1; check that it holds in S1

TRACE_OFF ;end trace at finish of simulation
```

The same test sequence in truth table format would appear as in **Table 1**. Although the truth table requires more typing effort, the result is clearer in terms of the device operation. Also, the output is automatically checked on each line, not just when specified as in the text-based method.

Simulation results may usually be viewed in tabular form or as a waveform display after compilation of the logic equations. Any discrepancies between the expected result of the simulation and the actual result will show up and allow modification of the logic equations to give the fiesired logic function.

Compilation of logic equations, together with the simulation segment will give a programming file with test vectors where this option is possible. Most PAL-type devices will accept programming files with vector testing; most fpgas are either configured in-circuit or programmed on dedicated programmers which do not allow for vector testing. In-circuit testing, using JTAG protocols, is more usual for fpgas.

Logic equation shortcomings

There are two problems with using logic equations in fpga designs. Firstly there is no global standard for the symbols and syntax; in these articles I have standardised on the ABEL symbols, but other standards, such as PALASM are equally valid. This is not an obstacle in itself for these different standards have worked very well for classical pld designs, but it would be very useful to have a universal standard which would be accepted by any fpga design system.

More important is the scale of fpgas compared with plds. A fairly complex pld such as the 22V10, contains over one hundred product terms, each of which can be defined by an equation similar to the decoder example above. To try to design with equations, even at this scale, can make a design extremely difficult to comprehend. Writing down the equations is a time consuming task in itself. But trying to decide where modifications should be made in the event that simulation throws up a mistake, for example, may prove even more arduous.

The architecture of fpgas dictates that logic equations are not the best way to define their logic content. Complex plds have a fixed structure which means that a logic equation maps directly into the AND-array of the logic cell. The only variable is the way in which signals are routed from the i/o lines into the logic blocks. But even this task can be approached in a fairly mechanical way, for there are fixed paths for the signals to travel inside the device.

In an fpga, the structure is much freer. For a start, there is not a fixed two-level AND-OR structure for a one-to-one association with the typical sum of products logic equation.

Table 1. Test sequence in truth-table format for simulating the combination lock.

[I3	I2,	I1	I0,	DOOR,	ALARM,	RESET	CLK]→	[Q3,	Q2	Q1,	Q0]
[1,	1,	1,	1,	0,	0,	1,	0]→	[0,	0,	0,	0]
[1,	1,	1,	1,	0,	0,	0,	C]→	[0,	0,	0,	0]
[0	1,	1,	0,	0,	0,	0,	C]→	[0,	0,	0,	1[
J,	1.	1,	0,	0,	0,	0,	C]→	[0,	0,	0,	1]

and so on...

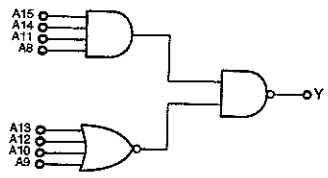


Fig. 1. The most common application for pals was address decoding in microprocessor circuit, where one pal replaced two discrete logic ICs.

potential internal connections decided. This third stage – device fitting – shows whether the target device has the capability to contain all the logic functions in the abstract design, but does not guarantee that the programmed device will work in the designers circuit. This can be taken a step doser by post-layout simulation.

Now that a real device is involved, with internal components and connections of predictable performance, the time-dependent factors can be added to the simulation result. Not only can we confirm that output Y = input A AND input B, but we can also predict that signal Y will go high within 5ns of both inputs A and B going high. Thus it is possible to program a device and plug it into a circuit with a high confidence of success – provided that all critical eventualities have been covered in the simulation.

We can now examine each design method in detail and see how they measure up to the requirements outlined above.

Logic equations

Input methods

Logic equations have been the standard method for designing plds ever since their introduction to the market in the late 1970s. As I showed in an earlier article, the early pals were simple sum-of-product devices whose outputs depended on a straightforward logic relationship between the inputs, often without any feedback, or other complications. Logic equations are the most common way of defining the logic content of classical plds.

Compilers for logic equations commonly consist of four sections. These are an introduction, a pin-out definition, an equations section and a simulation segment.

The introduction can include the drawing number, designer's name and company and a brief functional description plus other relevant information. It needs no further discussion as it merely annotates the design with information needed for future reference.

The pin-out definition section is also self-explanatory. It lists the signals used in the design and allocates them to the device i/o pins.

Simple examples are a good way of illustrating the logic equation part of the design input. The most common application for pals was as address decoders in microprocessor circuits. In this application, standard ttl parts did not have the input width or flexibility to provide an economical solution.

For example, an equation such as:

```
!Y = A15 & A14 & !A13 & !A12 & A11 & !A10 & !A9 & A8
```

was – and still is – commonplace in pal design sheets. It fits comfortably into one eighth of a standard pal whereas it would need the best part of two gate packages to implement it in discrete logic.

It is also easy to understand what is meant by this equation; when the processor puts out address 36xx, this output is taken low and will, presumably, enable some peripheral chip. **Figure 1** shows how this circuit may be implemented in discrete logic. Although not completely incomprehensible, it is not as immediately apparent as the logic equation.

If there were six or seven decoded outputs, all looking like Fig. 1, there might well be some confusion, especially without some annotation on the drawing. Annotating equations is straightforward enough; our address decode may be commented as:

```
!Y = A15 & A14 & !A13 & !A12 & A11 & !A10 & !A9 & A8;
     decode of 36xx
```

the semi-colon delimiting the comment from the actual equation.

More complex functions may be defined as logic equations, with equal clarity, by using various shorthand techniques. For example, a four-bit identity comparator, which is constructed from four exclusive-OR gates and an AND gate may be defined in the following way:

```
EQ0 = A0 & B0 # !A0 & !B0; 'zero' bits equal
EQ1 = A1 & B1 # !A1 & !B1; 'one' bits equal
EQ2 = A2 & B2 # !A2 & !B2; 'two' bits equal
EQ3 = A3 & B3 # !A3 & !B3; 'three' bits equal
EQ  = EQ0 & EQ1 & EQ2 & EQ3; all bits equal
```

Using the exclusive-OR symbol '#' instead of' the expanded logic definition would make the equations even clearer, and would be understood by most logic compilers.

You can also define a state machine with logic equations. Consider a four-digit identification number detector for use in a keypad combination lock. Its state diagram is reproduced in **Fig. 2.**

The lock is unlocked by entering the code '6714'; any other code sets off an alarm which can be cancelled with a remote reset. The machine is also reset when it detects that the door has been dosed after a successful entry. Physically, the state register has four bits, outputs $Q_{3\text{-}0}$, with four inputs $I_{3\text{-}0}$ for the code entry. Signal DOOR closes the door while ALARM cancels the alarm.

The keypad decoder sends out a digit encoded by $I_{3\text{-}0}$ or 'F' if no key is being depressed. System outputs are UNLOCK to enable the door, and SOUND to set the alarm.

First, it is necessary to define the states and the 'input codes, as follows:

```
S0 = !Q3 & !Q2 & !Q1 & !Q0
S1 = !Q3 & !Q2 & !Q1 & Q0
S2 = !Q3 & !Q2 & Q1 & !Q0
and so on through
S9 = Q3 & !Q2 & !Q1 & Q0
```

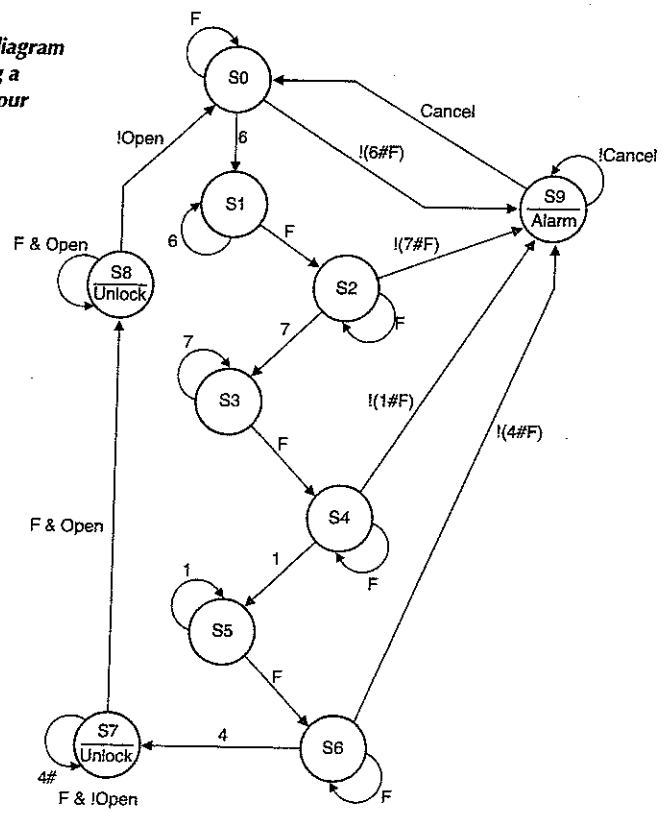


Fig. 2. State diagram for dectecting a sequence of four numbers.

This may have to be broken down into a chain of gates in order to achieve the required input connectivit. If the same logic expression is used in two different equations the inference is that it is recreated at each location in the fpga where the overall equations are implemented. It may be more efficient to generate it only once and route it to the second area of the fpga. This will save logic modules but use routing resources.

I have stressed that fpga structures are more like an integrated form of discrete logic. It is most unusual to design pcbs for discrete logic with logic equations. While logic equations are useful and help visibility for pal-type plds, they are not the ideal way to design fpgas.

State machine basics

A state machine is a system which, as its name implies, can exist in a number of stable states. Each state is usually defined by a unique number stored in a set of bistable devices called the state register.

Inputs to a state machine are those signals which can influence the sequence in which the states are entered. There is also a clock to define the time intervals at which the inputs are sampled and the decision made as to whether the state register is changed, and which state should be entered next.

Outputs from a state machine may depend on the state register only, in which case it is called a Moore machine. Alternatively, they may be a logical combination of inputs and state register, when it is known as a Mealy machine.

A state machine may be described by a state diagram; the door lock in Fig. 2 is an example. Each of the ten possible states is represented by a circle labelled with the value of the state register for that state. Transitions between states are represented by arrows labelled with the logic condition which enables that transition.

The 'main sequence' runs vertically downwards from S0 to S8, wrapping around back to S0 when the sequence ends with the door closing again. Transitions to S9 are triggered by incorrect key depressions. Each state has a hold condition which is shown by an arrow wrapped round back to the same state. Thus, keying '6' triggers a jump from S0 to S1 but, if key '6' remains depressed for more clock cycles the state machine remains in S1. Releasing the key sends 'F' to the inputs and triggers the jump to S2, ready for the next key push.

The outputs are decoded directly from the states, making this design a Moore machine.

State equation syntax

Most pld and fpga logic-entry systems allow logic equations and state equations to be mixed in the same design file. The syntax for entering state equations varies from system to system but the commonest methods use if-then-else or case statements. With either, it is mandatory to define the states in terms of the state register elements. This may be done with the following syntax:

```
[Q3, Q2, Q1, Q0]
S0 = 0000b;
S1 = 0001b;
S2 = 0010b;
```

and so on, or by:

```
STATE
S0 = !Q3 & !Q2 & !Q1 & !Q0;
S1 = !Q3 & !Q2 & !Q1 & Q0;
S2 = !Q3 & !Q2 & Q1 & !Q0;
etc.
```

The if-then-else structure may be written as:

```
WHILE [S0]
  IF I3 & I2 & I1 & I0 THEN [S0]
  IF !I3 & I2 & I1 & !I0 THEN [S1]
  ELSE [S9] WITH SOUND
WHILE [S1]
  IF !I3 & I2 & I1 & !I0 THEN [S1]
  IF I3 & I2 & I1 & I0 THEN [S2]
  .
  .
  .
WHILE [S9]
  IF ALARM THEN [S0]
  ELSE [S9] WITH SOUND
```

The else statements, above, have different effects. In the [S0] statement, the 'else' sends the machine to state [S9] if the input is not 'F' or '6'; in the [S9] statement it defines the hold condition.

The 'with' operator defines a combinatorial output. In a Moore machine, as in this case, an output is always associated with the same state; in Mealy machines, output conditions may depend on the path by which a state is reached.

The 'case' construct defines conditions to be tested, and the action to be taken when the condition is true. In the door-lock example, you could define the state machine with the following case statement:

```
CASE (ALARM, DOOR, Q[3..0])
BEGIN
   #h00:          CASE   (I[3..0])
       BEGIN
       #hF:       BEGIN  Q[3..0] = #h0 END
       #h6:       BEGIN  Q[3..0] = #h1 END
       OTHERWISE:        Q[3..0] = #h9 END
       END
   #h01=          CASE (I[3..0])
       BEGIN
       #hF:       BEGIN  Q[3..0] = #h2 END
       #h6:       BEGIN  Q[3..0] = #h1 END
       END
.
.
.
   #h09:          BEGIN  Q[3..0] = #h9 END
   #h29:          BEGIN  Q[3..0] = #h0 END
OTHERWISE:                Q[3..0] = #h0 END
END
```

This illustrates the use of nested case statements. In this example, it is necessary to allow the default jumps to [S9]; if the input conditiom were not nested within each present state 'case', every possible input combination would have to be specified to define the jump to the error state. Nesting, the 'otherwise' operator takes care of defaults as the else operator does in the if-then-else construct. ■

The next article in this series will cover hardware description languages – including VHDL