9 Field Programmable Gate Array

This chapter introduces the principles, implementation and programming of configurable logic circuits, from the point of view of cell design and interconnection strategy.

9.1 Introduction

Field programmable gate arrays (FPGA) are specific integrated circuits that can be user-programmed easily. The FPGA contains versatile functions, configurable interconnects and an input/output interface to adapt to the user specification. FPGAs allow rapid prototyping using custom logic structures, and are very popular for limited production products. Modern FPGA are extremely dense, with a complexity of several millions of gates which enable the emulation of very complex hardware such as parallel microprocessors, mixture of processor and signal processing, etc… One key advantage of FPGA is their ability to be reprogrammed, in order to create a completely different hardware by modifying the logic gate array. The usual structure of FPGA is given in figure 9-1.

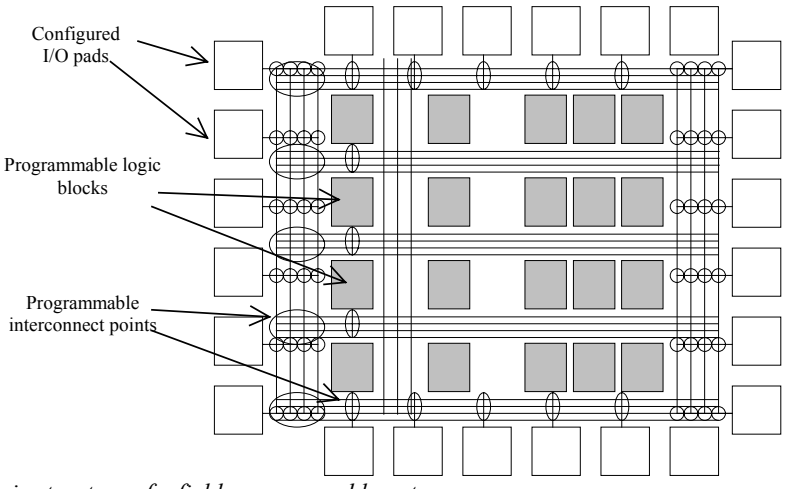


Figure 9-1: Basic structure of a field programmable gate array

One example of a very simple function (3-input XOR) implemented in a FPGA is given in figure 9-2. Three pads on the left are configured as inputs, one logic block is used to create the 3-input XOR and one pad on the right is used as output. The propagation of signals is handled by interconnect lines, connected together at specific programmable interconnect points.

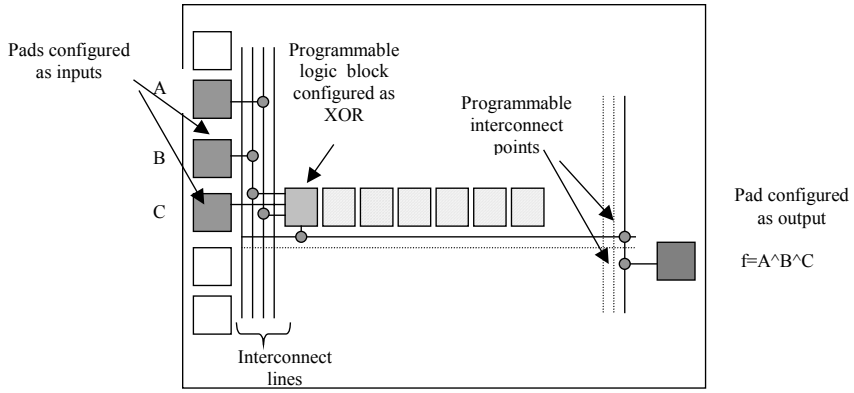


Figure 9-2: Using a field programmable gate array to build a 3-input XOR gate

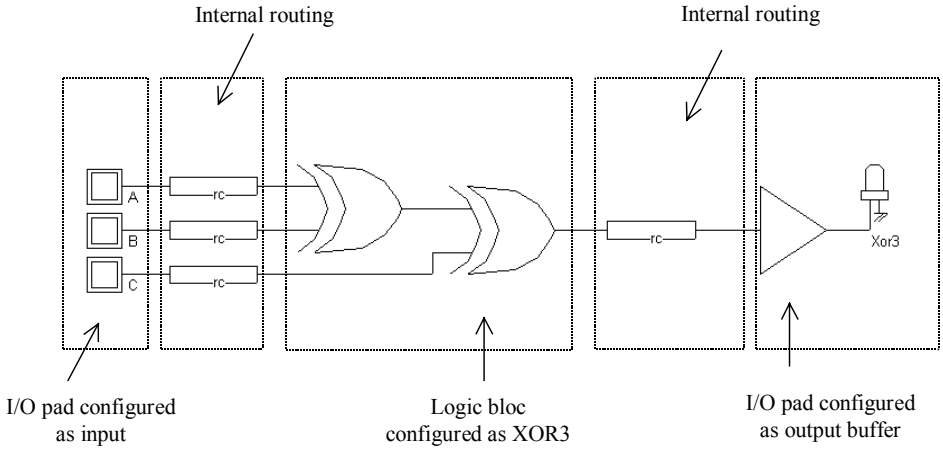
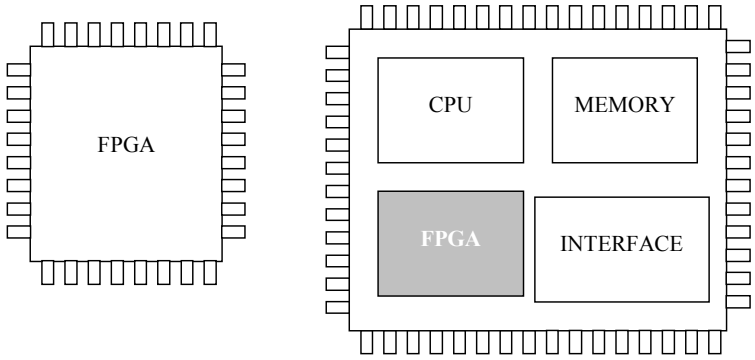


Figure 9-3: Equivalent circuit for the FPGA configured in XOR3 gate

Three pads are configured as inputs and represent the logical information A,B and C (Figure 9-3). An internal routing path is created to establish an electrical link between the I/O region and the logic bloc. Internally, the logic bloc may be configured in any combination of sequential basic function. Each logic bloc usually supports 3 to 8 logic inputs. In our example, the bloc is configured as a 3-input XOR. Then, other internal routing wires are configured in order to carry out the signal to an I/O pad configured as an output. The global propagation delay of such architecture is evidently very high, if compared to a 3-input XOR gate that may be found in the cell library. This is usually the price to pay for configurable logic circuits.

Notice that FPGA not only exist as simple components, but also as macro-blocs in system-on-chip designs (Figure 9-4). In the case of communication systems, the configurable logic may be dynamically changed to adapt to improved communication protocol. In the case of very low power systems, the configurable logic may handle several different tasks in series, rather than embedding all corresponding hardware that never works in parallel.



(a) Stand-alone FPGA *(b) FPGA internal block*

Figure 9-4: FPGA exist as stand-alone Ics or blocs within a system-on-chip

9.2 Configurable Logic Circuits

The programmable logic block must be able to implement all basic logic functions, that is INV, AND, NAND, OR, NOR, XOR, XNOR, etc... Several approaches are used in FPGA industry to achieve this goal. The first approach consists in the use of multiplexor, the second one in the use of look-up tables.

MULTIPLEXORS

Surprisingly, a two-input multiplexor can be used as a programmable function generator, as illustrated in table 9-1. Remember that the multiplexor output *f* is equal to *i0* if *en=0*, and *i1* if *en=1*. For example, the inverter is created if the multiplexor input *i0* is equal to 1, *i1* is equal to 0, and enable is connected to *A*. In that case, the output *f* is the *~A*. The figure 9-5 describes the use of multiplexor to produce the OR, AND, NOT and BUF functions.

Function	Boolean expression for output f	i0	i1	en
BUF(A)	f=A	0	A	1
NOT(A)	f=~(A)	1	0	A
AND(A,B)	f=A&B	0	B	A
OR(A,B)	f=A|B	B	1	A

Table 9-1: use of multiplexor to build logic functions

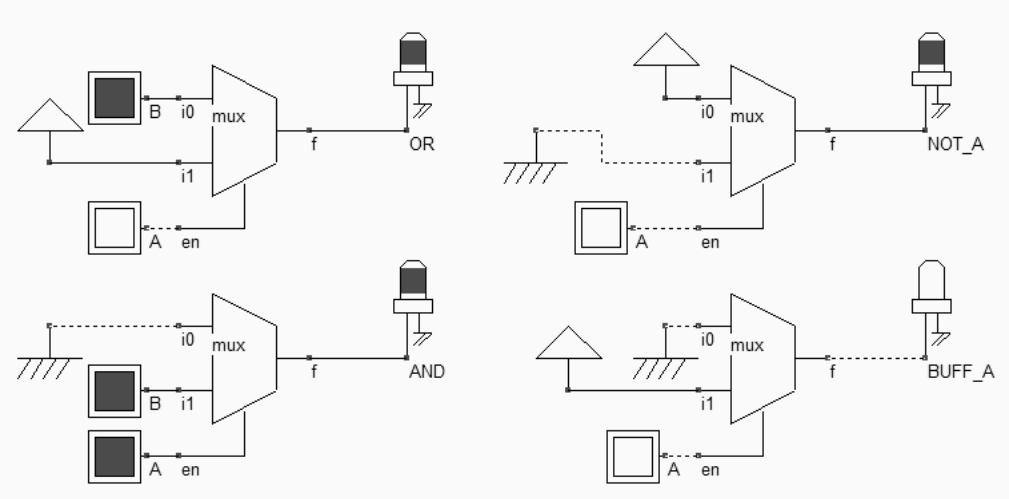


Figure 9-5: use of multiplexor to build logic functions (fpgaMux.SCH)

Although NOT, AND and OR are directly available, other functions such as NAND, NOR and XOR cannot be built directly using a single 2-input multiplexor, but need at least two multiplexor circuits.

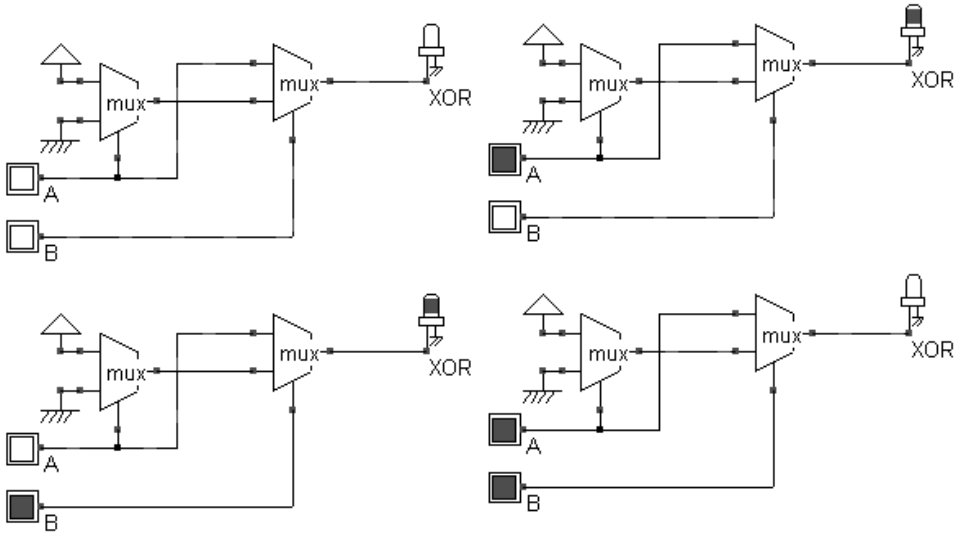
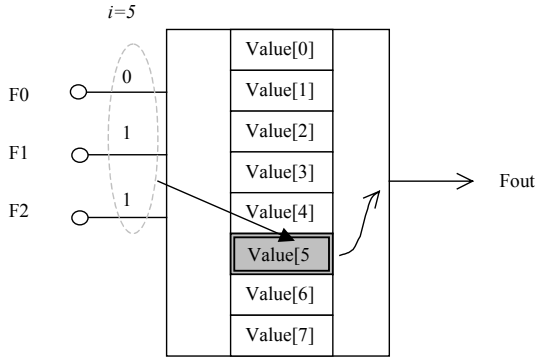


Figure 9-6: The XOR gate built from 2 multiplexor circuits (fpgaMux.SCH)

The XOR function is shown in figure 9-6. The 4-input XOR gate would require 6 multiplexor cells. Remember that each multiplexor cell consists of a minimum of 6 transistors for a buffered output, and has 3 delay stages (The two inverters and the pass transistor). The XOR4 implementation would comprise a total of 18 delay stages, which is far too important. Therefore, the multiplexor approach is not very efficient for many logical functions.

Look Up Table

The look-up table (LUT) is by far the most versatile circuit to create a configurable logic function. The look-up table shown in table 9-2 has 3 main inputs *F0*,*F1* and *F2*. The main output is *Fout*, which is a logical function of *F0*, *F1* and *F2.* The output *Fout* is defined by the values given to *Value[0]..Value[7]*. The three values *F0*,*F1*, *F2* create a 3-bit address *i* between 0 and 7, so that *Fout* gets the value of *Value[i]*. In the example of table 9-2, the input creates the number 5, so Value[5] is routed to *Fout*. The table below gives *Value[i]* for the most common logical functions of *F0*,*F1* and *F2.*



Look-up-table

Function	Value [0]	Value [1]	Value [2]	Value [3]	Value [4]	Value [5]	Value [6]	Value [7]
~F0	0	1	0	1	0	1	0	1
~F1	0	0	1	1	0	0	1	1
~F2	0	0	0	0	1	1	1	1
F0&F1	0	0	0	1	0	0	0	1
F0|F1|F2	0	1	1	1	1	1	1	1
F0^F1^F2	0	1	1	0	1	0	0	1

Table 9-2: Link between basic logic functions and the information stored in Value[0]..[7]

In the case of the 3-input XOR, the set of values of *Fout* given in the truth-table of table 9-3, must be assigned to *Value[0]..Value[7].* In the schematic diagram shown in figure 9-7, we must assign manually the *Fout* truth-table to each of the 8 buttons. Then *Fout* produces the XOR function of inputs *F0*, *F1* and *F2*.

F2	F1	F0	Fout= F0^F1^F2	Assigned to
0	0	0	0	Value[0]
0	0	1	1	Value[1]
0	1	0	1	Value[2]
0	1	1	0	Value[3]
1	0	0	1	Value[4]
1	0	1	0	Value[5]
1	1	0	0	Value[6]
1	1	1	1	Value[7]

Table 9-2: Truth-table of the 3-input XOR gate for its implementation in a look-up-table

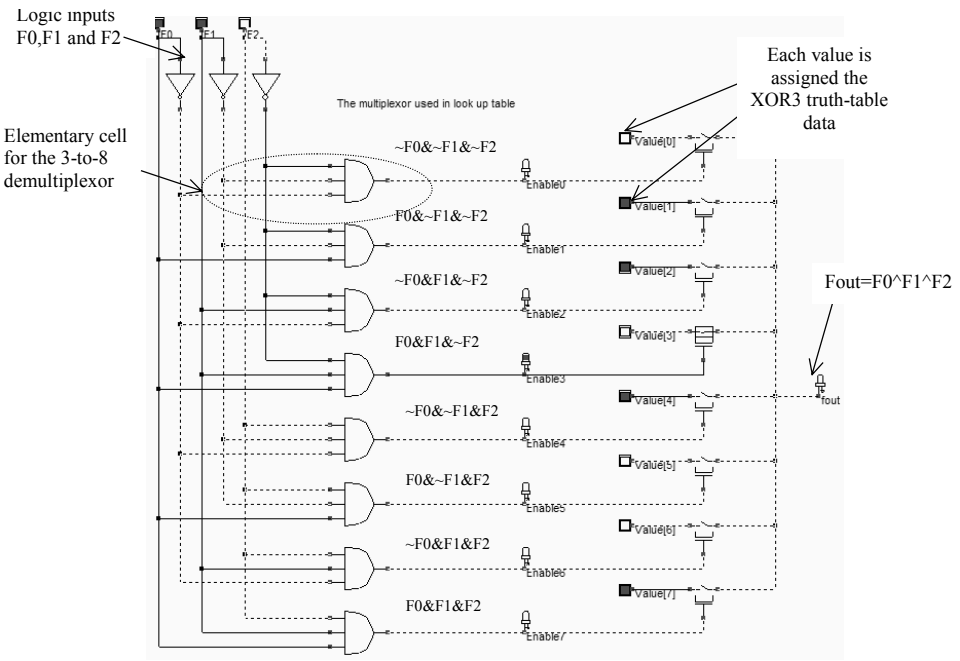


Figure 9-7: The output f produces a logical function Fout according to a look-up-table stored in memory points Value[i] (FpgaLutStructure.SCH)

Memory Points

Memory points are essential components of the configurable logic blocks. The memory point is used to store one logical value, corresponding to the logic truth table. For a 3-input function (F0,F1,F2 in the previous LUT), we need an array of 8 memory points to store the information *Value[0]..Value[7]*. There exist here also several approaches to store one single bit of information. The one that is illustrated in figure 9-8 consists of D-reg cells. Each register stores one logical information *Value[i]*. The *Dreg* cells are chained in order to limit the control signals to one clock *ClockProg* and one data signal *DataProg.* The logical data *Value[i]* is fully programmed by a word of 8 bits sent in series to the signal *DataProg*

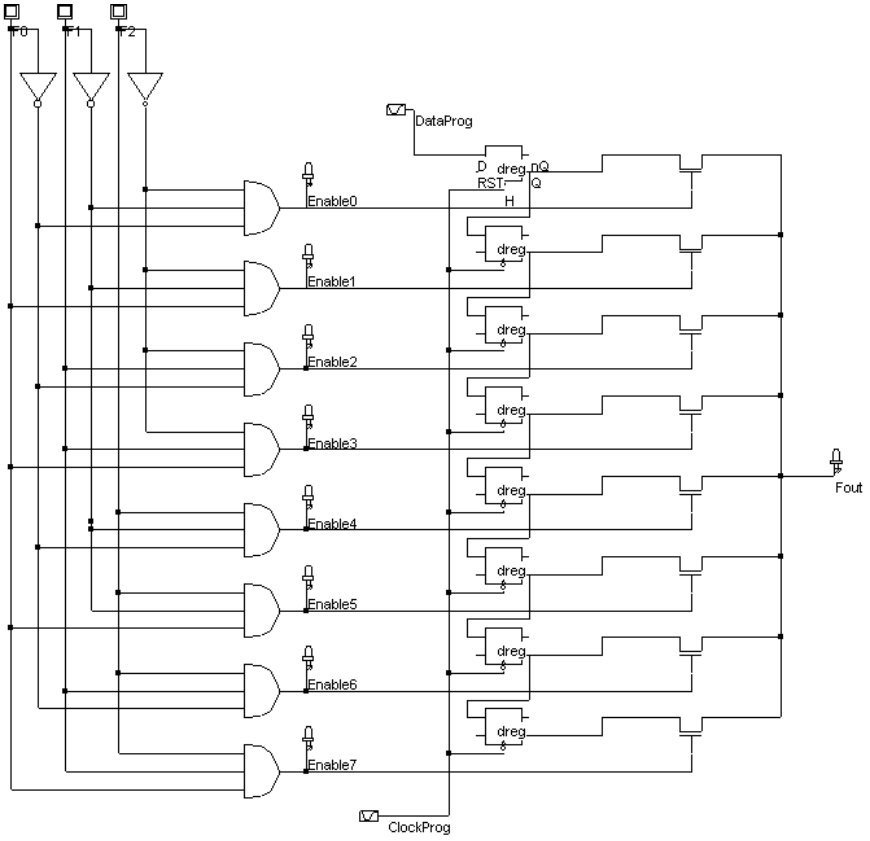


Figure 9-8: The look-up information is given by a shift register based on D-reg cells (FpgaLutDreg.sch).

The configuration of the 3-input LUT into a 3-input XOR gate obeys to a strict protocol described in figure 9-9. A series of 8 active edges is generated by the *ClockProg* signal (*Dreg* is active on fall edges). This is done by configuring a pulse generator with series of 0-1 as shown below.

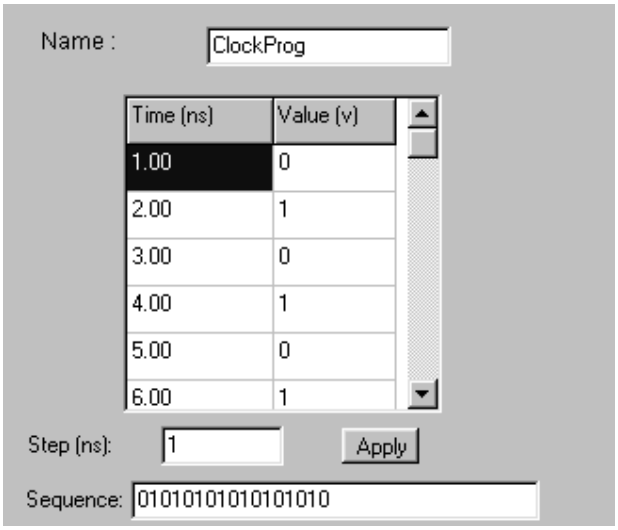


Figure 9-9: Programming the ClockProg pulse to generate 8 active edges (FpgaLutDreg.sch)

At each active edge, the shift register is fed by a new value presented sequentially at input *DataProg*. As the D-reg is active on fall edge, data may be changed on each rise edge. Notice that the last *Dreg* corresponds to *Value[7]*. Therefore, *Value[7]* must be inserted first, and *Value[0]* last. This means that the *DataProg* pulse must describe the truth table in reverse order, as shown below.

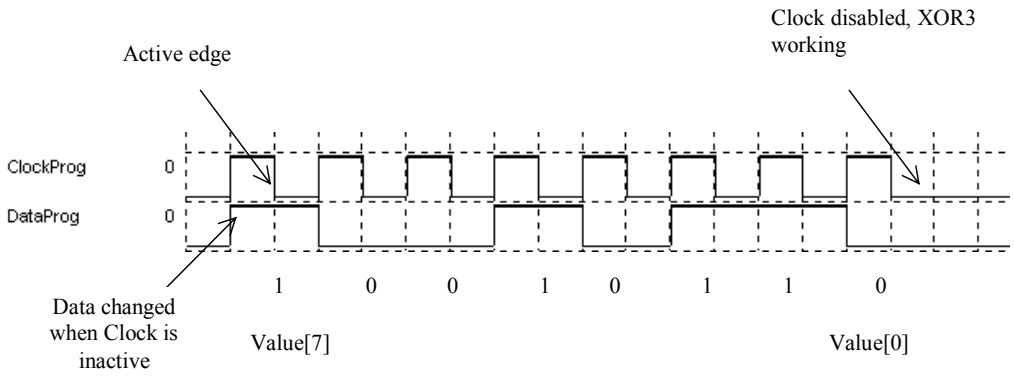


Figure 9-10: At the end of the 8-th clock period, the LUT is configured as a 3-input XOR (FpgaLutDreg.sch)

Most FPGA designs use *Dreg* to store the LUT configuration. Notice that the configuration is lost when the power supply is down.

Fuse and Antifuse

To retain the configuration even without power supply, non volatile memories must be used. A one-time programmable non-volatile memory is the fuse [Sharma][Uyemura]. Usually, a contact between metal layers is used as a fuse, as an over-current would blow its structure, as illustrated in figure 9-11. Although this technique induces severe damages close to the contact, no specific technological layer is required as it is a CMOS compatible approach.

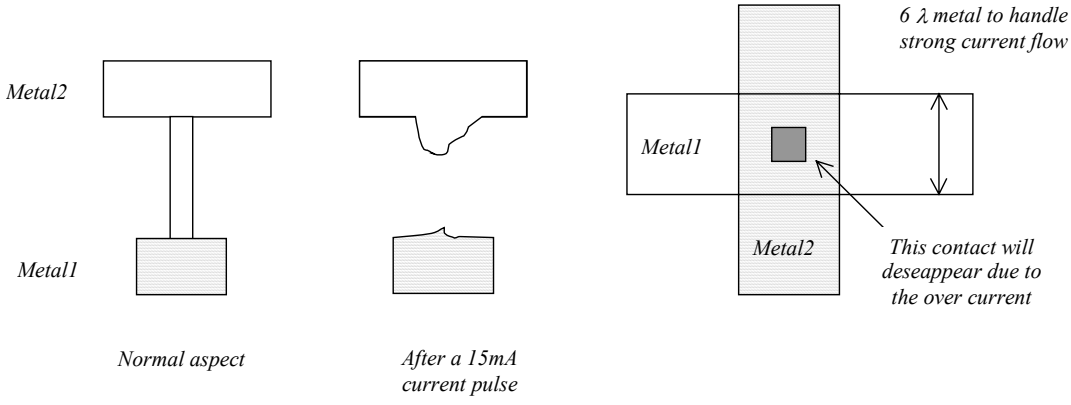


Figure 9-11 Contact fuse

A driver with large channel width (Several µm), supplied by the highest available voltage (VDDH) performs the drive of very strong current pulse. The schematic diagram of the fuse circuit is shown in figure 9-12. When the command *BlowFuse* is active, both nMOS and PMOS devices are on, leading to a short circuit current. This current must be higher than 15mA to destroy the contact.

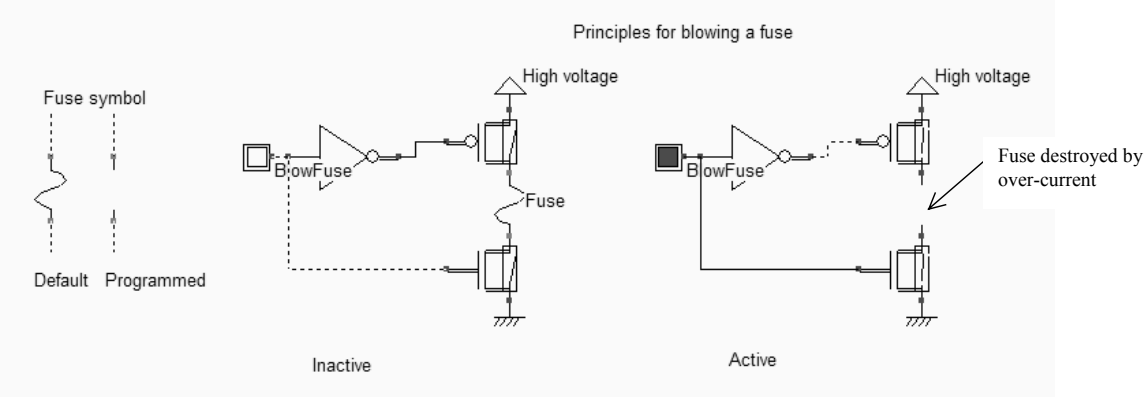


Figure 9-12: Fuse circuit programming (FuseCircuits.SCH)

In contrast to the fuse, the normal state of the antifuse is to be opened. In the example shown in figure 9-13, a thin insulator interrupts the contact between metal1 and metal2. A very high voltage applied between metal1 and metal2 (Typically 10V) breaks the oxide and provokes a conductive path between the metal layers. The use of very high voltage on the chip requires a careful use of high-voltage MOS, and of specific I/O pads, to ensure that no part of the circuit is damaged.

Another popular structure, called ONO (Oxide, Nitride, Oxide) leads to a resistive path when programmed. The typical value of the resistance is 500 ohm. Statistically, the spread of the resistance is much larger for the SiO2 than for the ONO fuse [Smith], which makes the ONO fuse more attractive, at the price of supplementary process steps.

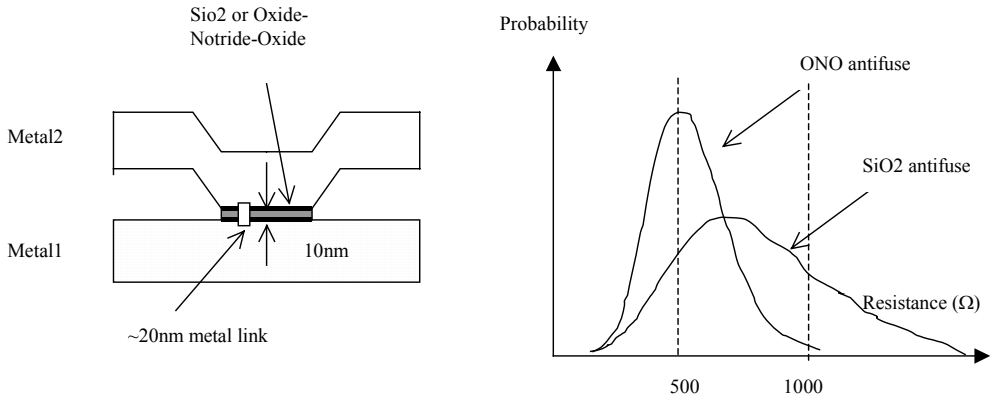


Figure 9-13: the antifuse principles and the comparative resistance spread for ONO and SiO2

Other types of non-volatile memories are being used for hardware programming of FPGA array: EEPROM and FRAM memories. These memories are not altered when the power supply is down, and can be re-programmed a large number of times. These types of memory cells are detailed in chapter 10.

Implementation in DSCH

In DSCH, a look-up-table symbol is proposed in the symbol menu (Figure 9-14). It is equivalent to the schematic diagram of figure 9-8. An important property of the LUT symbol is its ability to retain the internal programming as a non-volatile memory would do. The user's interface of the LUT symbol is given in figure 9-14. There are three ways of filling the look-up-table: one consists in defining each array element with a 0 or a 1. The number corresponds to the logic combination of inputs F2,F1,F0. For example n°4 is coded 100 in binary, corresponding to F2=1,F1=0 and F0=0. A second solution consists in choosing the function description in the list. The logic information *Fout* assigned to each combination of the inputs updates the look-up-table. A third solution is also proposed: enter a description based on inputs F0,F1 and F2, and the logic operators "~" (Not), "&' (And), "|" (Or) and "^" (Xor). Then click the button **Fill LUT** to transfer the result of the expression to the table.

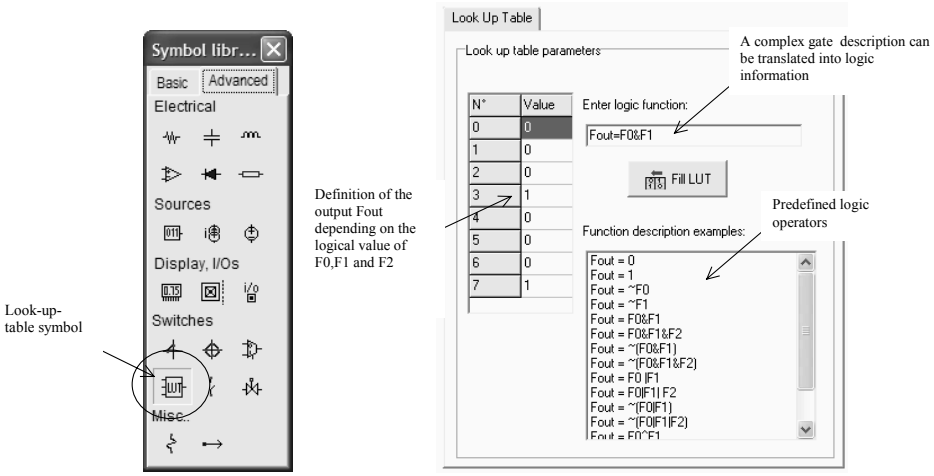


Figure 9-14: The look-up-table symbol

9.3 Programmable Logic Block

The programmable logic block consists of a look-up table, a D-register and some multiplexors. There exist numerous possible structures for logic blocks. We present in figure 9-15 a simple structure which has some similarities with the Xilinx XC5200 series (See [Smith] for detailed information on its internal structure). The configurable block contains two active structures, the Lut and the D-reg, that may work independently or be mixed together.

The output of the look-up-table is directly connected to the block output *Fout*. The output can also serve as the input data for the D-register, thanks to the multiplexor controlled by *DataIn_Fout*. The *DataOut* net can simply pass the signal *DataIn*, in that case the cell is transparent. The *DataOut* signal can also pass the signal *nQ*, depending on the multiplexor status controlled by *DataIn_nQ*

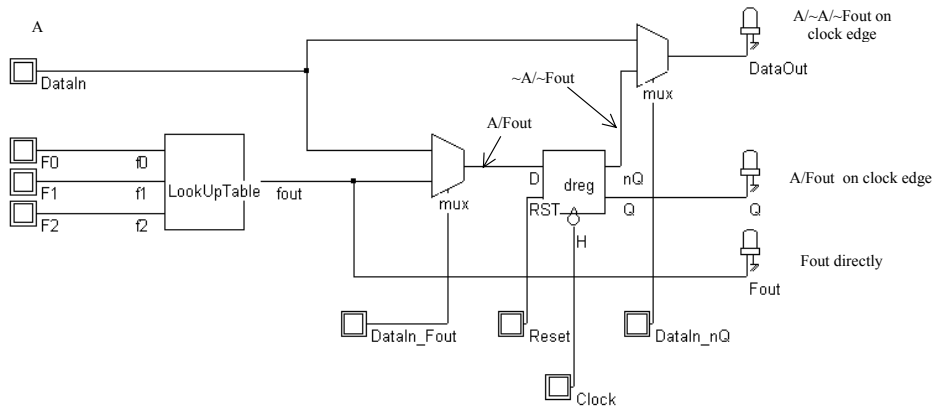


Figure 9-15: Simple configurable logic block including the Look Up Table and a D-register (FpgaCell.SCH)

The block now consists of the LUT and the D-register. We chain the information *DataIn_Fout* and *DataIn_nQ* on the path of the shift register by adding 2 supplementary *Dreg* cells. Each *Dreg* still uses the same clock *ClockProg* and chained input data *DataProg*. The complete circuit is shown in figure 9-16.

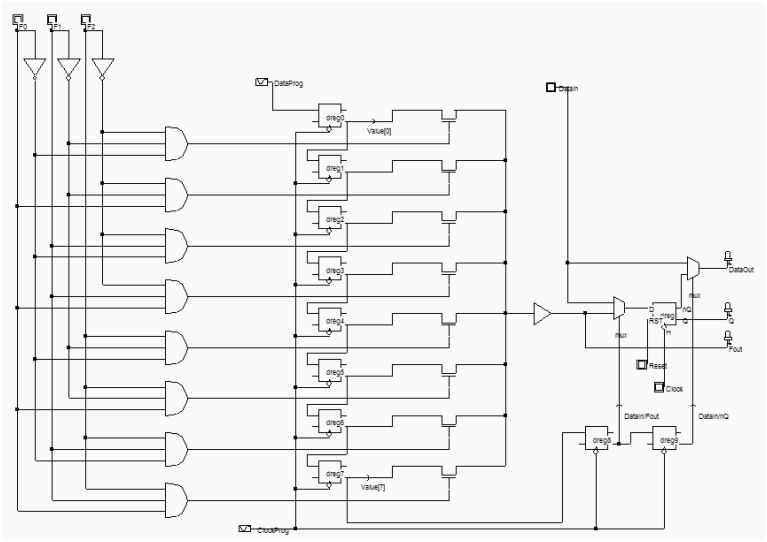
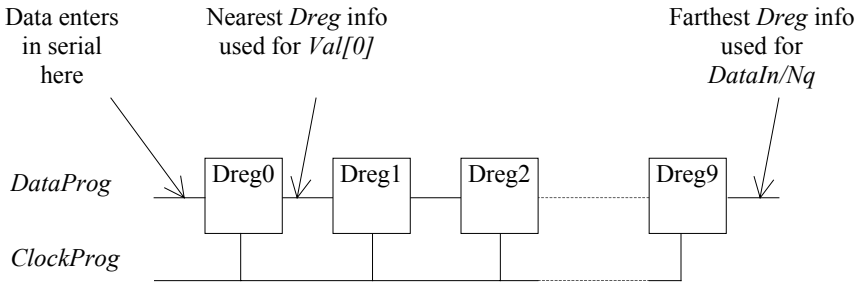


Figure 9-16: The Look Up Table, the D-register and the shift register including the 2 multiplexor cells (FpgaBlockStructure.SCH)

The configuring of the block is achieved thanks to 10 active clock edges on *ClockProg*, and 10 serial data on *DataProg*. The chain of *Dreg* starts at Dreg0 (Upper *Dreg* in figure 9-16, which produced *Value[0])* and stops at *Dreg9* (Right side of figure 9-16 which produced *DataIn/nQ*). The information that flows at the far end of the register chain is defined at the first cycle, while the closest register is configured by the data present at the last active clock edge.,



Clock cycle	1	2	3	4	5	6	7	8	9	10
DataProg	DataIn/ Nq	Datain /Fout	Val [7]	Val [6]	Val [5]	Val [4]	Val [3]	Val [2]	Val [1]	Val [0]

Table 9-3: Serial data information used to program the LUT memory points

9.4 Interconnection between blocks

The interconnection strategy between logic blocks is detailed in this paragraph. We shall focus on the programmable interconnect point and the programmable switching matrix. Then, we will discuss the global implementation of the structure.

Programmable Interconnect Point

The elementary programmable interconnect point (PIP <Gloss>)may be found in the **Advanced** set of **Switches** symbols (Figure 9-17). It consists of a configurable bridge between two interconnects.

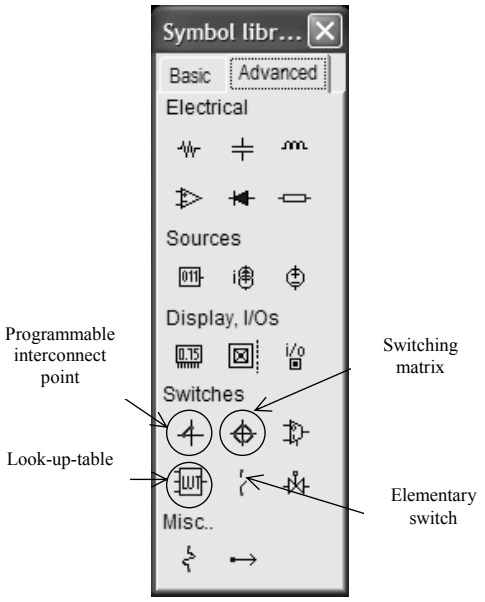


Figure 9-17: The programmable interconnect point (PIP) in the palette of symbols

The PIP may have two states: "On" and "Off". You may switch from "On" to "Off" by a double click on the symbol (Screen shown in figure 9-18) and a click on the button **On/off**.

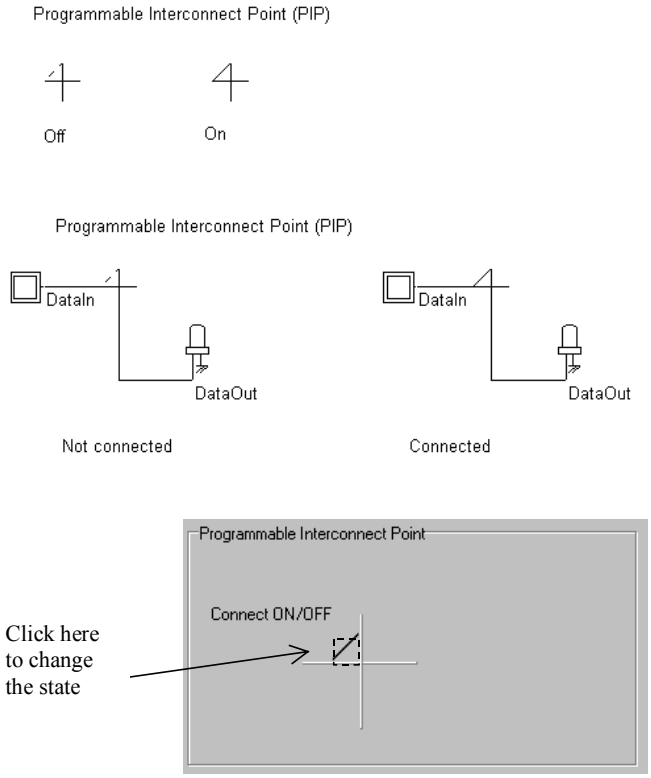
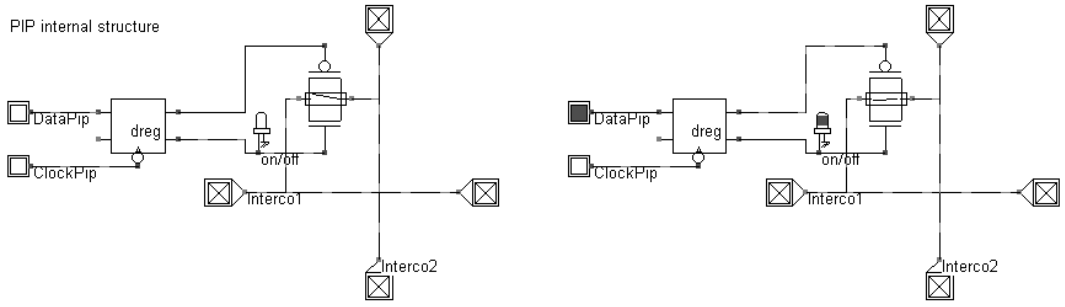


Figure 9-18: Changing the state of the PIP (FpgaPip.SCH)

The bridge can be built from a transmission gate, controlled once again by a *D-reg* cell (Figure 9-19). When the register information contains a 0, the transmission gate is off and no link exists between *Interco1* and *Interco2*. When the information held by the register is 1, the transmission gate establishes a resistive link between *Interco1* and *Interco2*. The resistance value is around 100 Ω.



(a) Switch off *(b) Swicth on*

Figure 9-19: Internal structure of the PIP and illustration of its behavior when Off (a) and On (b) (FpgaPip.SCH)

The regrouping of programmable interconnect points into matrix is of key importance to ensure the largest routing flexibility. Examples of 3x3 and 3x2 PIP matrix are shown in figure 9-20. The link between *In1* and *Out1*, *In2* and *Out2*, *In3* and *Out3* is achieved by turning some PIP on. A specific routing tool usually handles this task, but the manual re-arrangement is not rare in some complex situations. In DSCH, just press the key "O" to switch the PIP On and off.

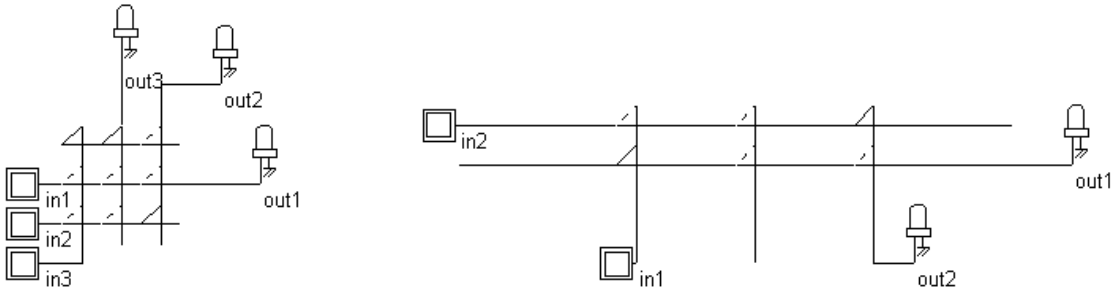


Figure 9-20: Matrix of programmable interconnects points (FpgaPip.SCH)

Switching Matrix

The switching matrix is a sophisticated programmable interconnect point, which enables a wide range of routing combinations within a single interconnect crossing. The aspect of the switching matrix is given in figure 9-21. The matrix includes 6 configurable bridges between the two main interconnects.

The switching matrix symbol may be found in **Advanced** set of **Switches** symbols. By a double click on the matrix symbol, you get access to the 6 **On/Off** switches.

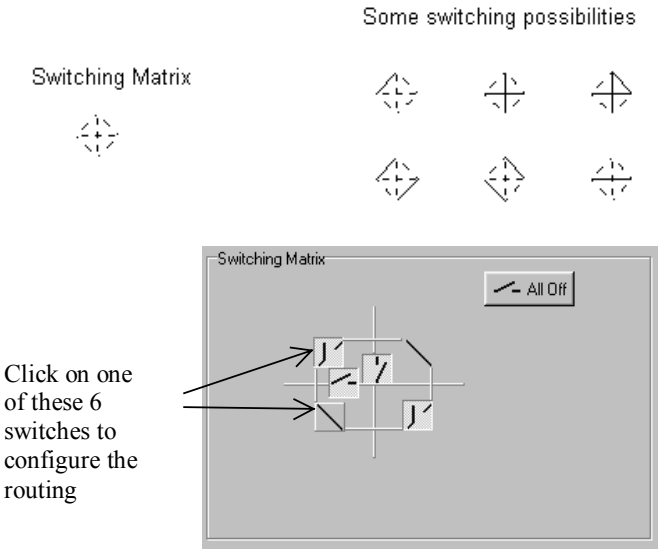


Figure 9-21: Changing the state of the matrix (FpgaMatrix.SCH)

To ease the programming of the matrix, short cuts exist in DSCH. You can change the state of the matrix by placing the cursor on the desired symbol and pressing the following keys:

- To switch off the matrix, press the key "o".
- To switch on the matrix, press the key "O".
- To enable an horizontal link, press the key "-".
- To enable a vertical link, press the key "|".

Examples of 3x2 and 3x3 switching matrix are given in figure 9-22. The routing possibilities are numerous, which improves the configurability of the logic blocs.

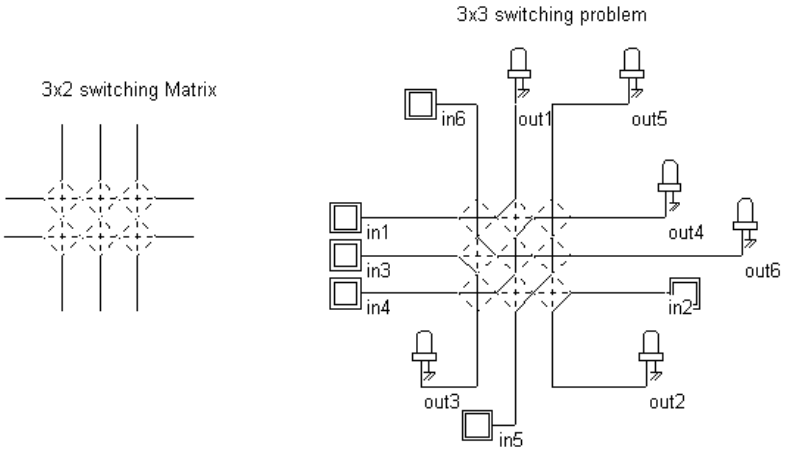


Figure 9-22: 3x2 switching matrix and example of routing strategy between 6 inputs and outputs (fpgaMatrix.SCH)

Implementation of the Switching Matrix

From a practical point of view, the switching matrix can be built from a regrouping of 6 transmission gates (Figure 9-23). Each transmission gate is controlled by an associated *Dreg* cell, which memorizes the desired configuration. The Dreg cells are chained so that one single input *DataIn* and one clock *LoadClock* are enough to configure the matrix.

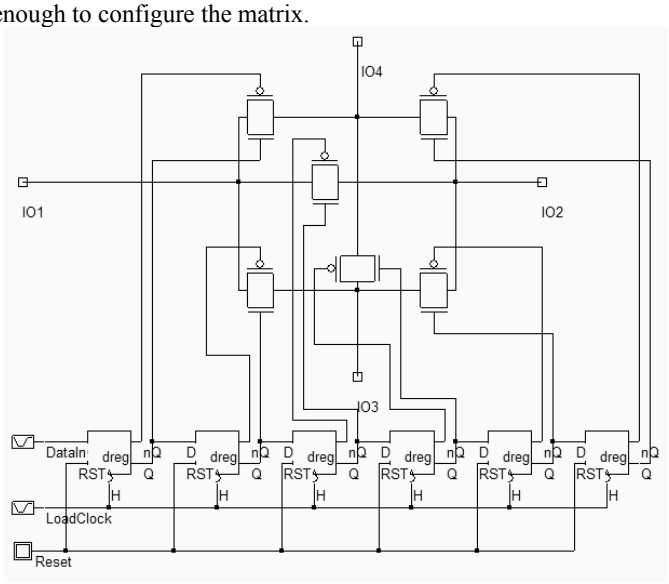


Figure 9-23: The transmission gates placed on the routing lines to build the matrix (FpgaMatrix3.SCH)

Array of Blocs

The configurable blocs are associated with programmable interconnect points and switching matrix to create a complete configurable core. An example of double configurable block and its associated configurable routing is proposed in figure 9-24.

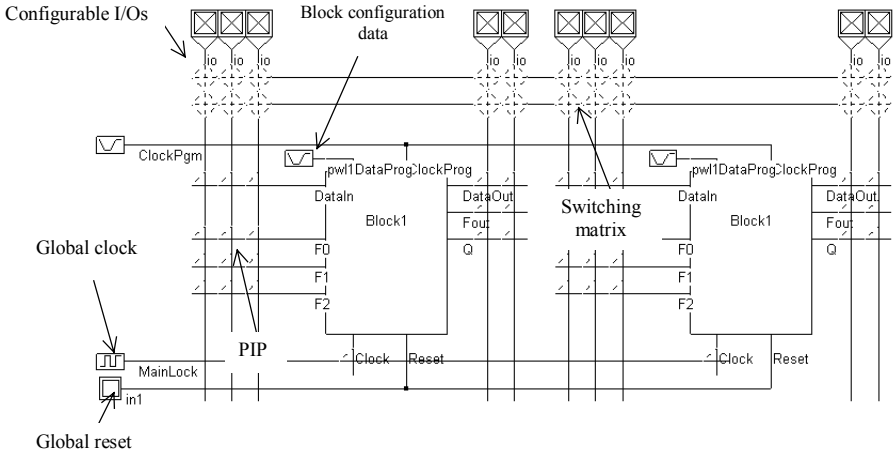


Figure 9-24: Configurable blocks, switching matrix, configurable I/Os and arrays of PIP (fpga2blocks.SCH)

Full-Adder Example

The truth table and logical expression for the full-adder are recalled in Table 9-3. The implementation of the CARRY and SUM function is achived by programming two look-up tables according to the truth-tables reported in table 9-3.

FULL ADDER

A	B	C	SUM	CARRY	RESULT
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	0	1	2
1	0	0	1	0	1
1	0	1	0	1	2
1	1	0	0	1	2
1	1	1	1	1	3

Table 9-3. The full-adder truth-table

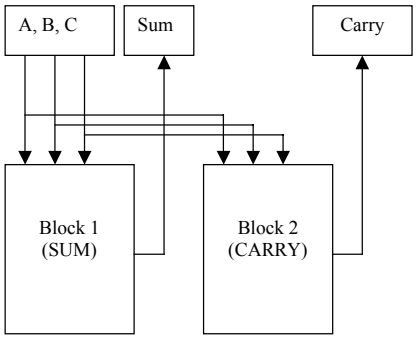


Figure 9-25: The SUM and CARRY functions to realize the full-adder in FPGA (fpgaFullAdder.SCH)

The general diagram of the Full adder implementation is given in figure 9-25. One programmable logic block *Block1* supports the generation of the sum for given logic values of the inputs *A,B* and *C*. The information needed to configure Block1 as a *Sum* function (3-input XOR) is given in table 9-4. Notice that we only use the LUT in this programmable logic block. The *Dreg* is not active, and we only exploit the output of the LUT *Fout*, which is configured as the *Sum*.

The signal *Sum* propagates outside the block to the output interface region by exploiting the interconnect resources and switching matrix. The other programmable logic block *Block2* supports the generation of the signal *Carry*, from the same inputs *A,B* and C. The programming of *Block2* is also given in table 9-4. The result *Carry* is exported to the output interface region as for the *Sum* signal. Again, in this block, only the LUT is active.

Block 1 (Sum of F0,F1 and F2)									
Cycle 1	2	3	4	5	6	7	8	9	10
DataIn Nq	Datain Fout	Val[7]	Val[6]	Val[5]	Val[4]	Val[3]	Val[2]	Val[1]	Val[0]
0	0	1	0	0	1	0	1	1	0

Block 2 (Carry of F0,F1 and F2)									
Cycle 1	2	3	4	5	6	7	8	9	10
DataIn Nq	Datain Fout	Val[7]	Val[6]	Val[5]	Val[4]	Val[3]	Val[2]	Val[1]	Val[0]
0	0	1	1	1	0	1	1	0	0

Table 9-4. Serial data used to configure the logic blocks 1&2 as SUM and CARRY

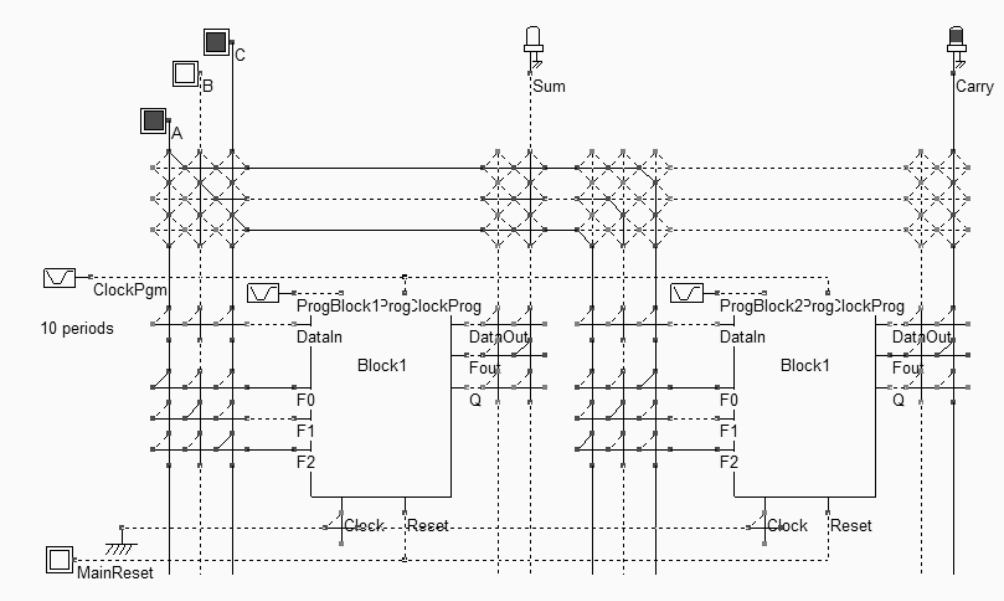


Figure 9-26: Simulation of the full-adder implemented in 2 configurable blocks (fpgaFullAdder.SCH)

The programming sequence is contained in the piece-wise-linear symbols *ProgBlock1* and *ProgBlcok2*. As seen in the chronograms of figure 9-27, the program clock *ClockPgm* is only active at the initialization phase, to shift the logic information to the memory points inside the blocks which configure each multiplexor. The routing of the signals *A,B* and *C* as well as *Sum* and *Carry* has been done manually in the circuit shown in figure 9-26. In reality, specific placement/routing tools are provided to generate the electrical structure automatically from the initial schematic diagram, which avoids manual errors and limits conflicts or omissions.

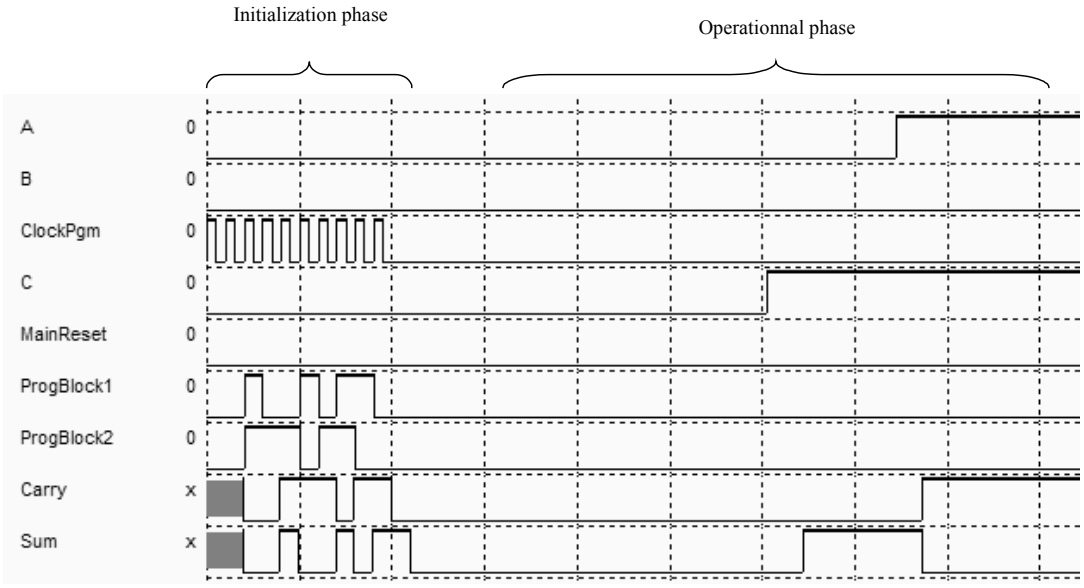
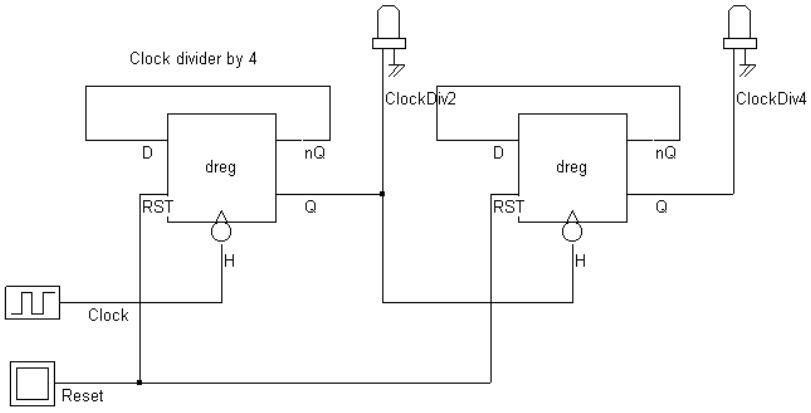


Figure 9-27: Chronograms of the full-adder FPGA (fpgaFullAdder.SCH)

Clock Divider Example

A second example is proposed as an application of the FPGA circuits. It concerns the clock division. We recall in figure 9-28 the general structure and the typical chronograms of the clock division by four, which requires two *Dreg* cells, with a feedback from the output *~Q* to the input *D*.



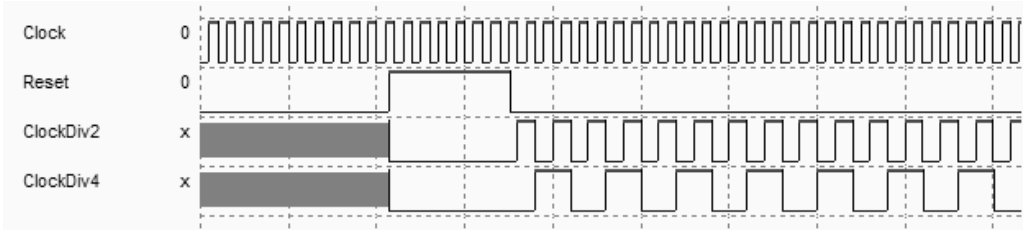


Figure 9-28: Diagram and typical simulation of the clock divider by 4 (ClockDiv4.SCH)

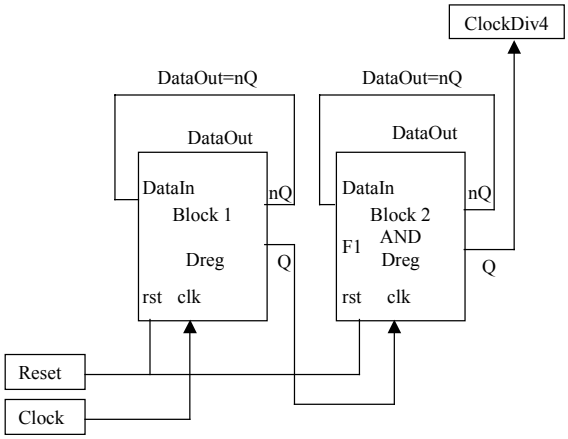


Figure 9-29: Implementation of the clock divider in 2 configurable blocs (FpgaDiv4.SCH)

The general diagram of the clock divider implementation is given in figure 9-29. Each programmable logic block is configured as a single stage clock divider. The information needed to configure *Block1* as a simple *Dreg* function is given in table 9-5. This serial data information creates a direct path from *DataIn* to input *D* of the *Dreg* cell, while *nQ* propagates to *DataOut*, as detailed in figure 9-30.

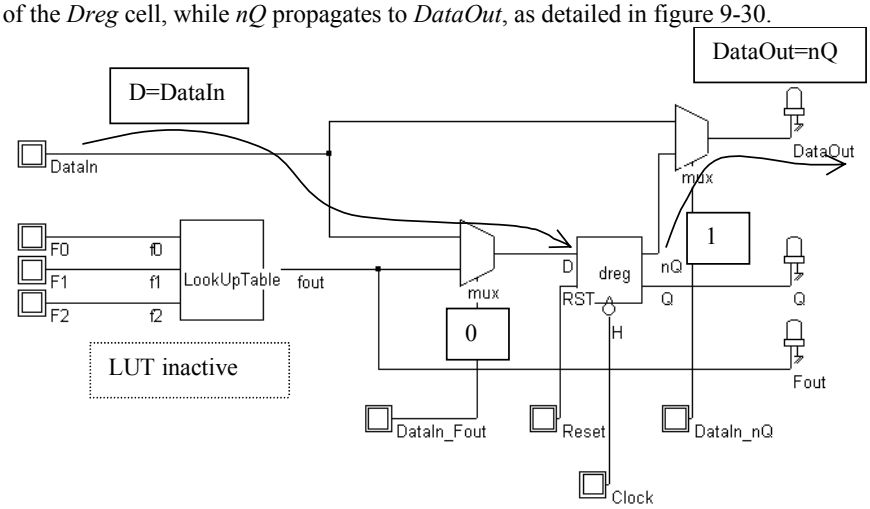


Figure 9-30: Use of the configurable block as a DReg (FpgaDiv4.SCH)

Block 1 (DataOut=nQ, D=DataIn)									
Cycle 1	2	3	4	5	6	7	8	9	10
DataIn Nq	Datain Fout	Val[7]	Val[6]	Val[5]	Val[4]	Val[3]	Val[2]	Val[1]	Val[0]
1	0	0	0	0	0	0	0	0	0

Block 2 (DataOut=nQ, D=DataIn)									
Cycle 1	2	3	4	5	6	7	8	9	10
DataIn Nq	Datain Fout	Val[7]	Val[6]	Val[5]	Val[4]	Val[3]	Val[2]	Val[1]	Val[0]
1	0	0	0	0	0	0	0	0	0

Table 9-5 Serial data used to configure the logic blocks 1&2 as clock dividers (FpgaDiv4.SCH)

Outside the programmable block, the signal *nQ* propagates to the input *DataIn*. Notice that the look-up table is inactive in this configuration. The other programmable logic block *Block2* is also programmed as a *Dreg* circuit with a feedback from *nQ* to *DataIn*.

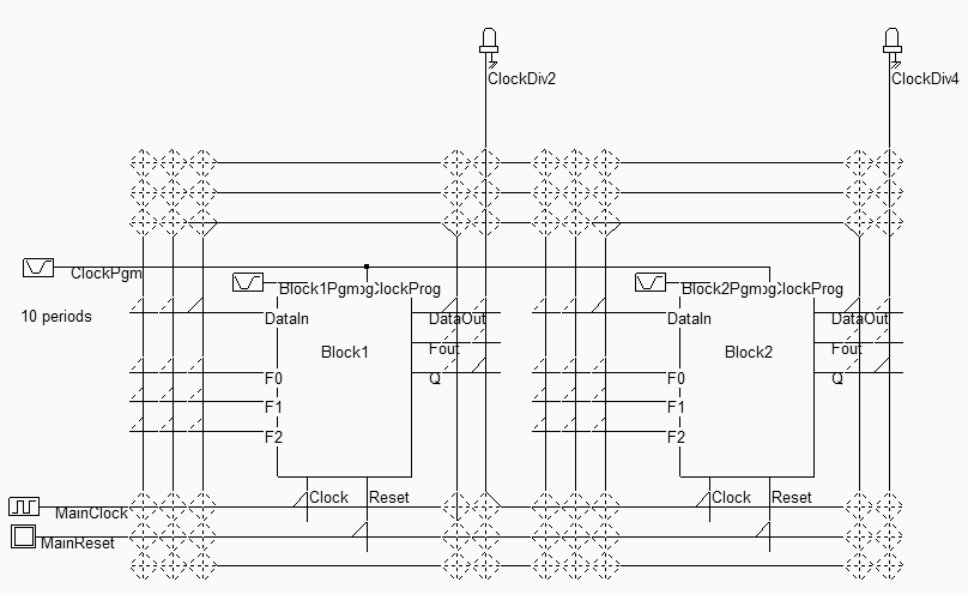


Figure 9-31: Routing of the clock divider in 2 configurable blocs (FpgaDiv4.SCH)

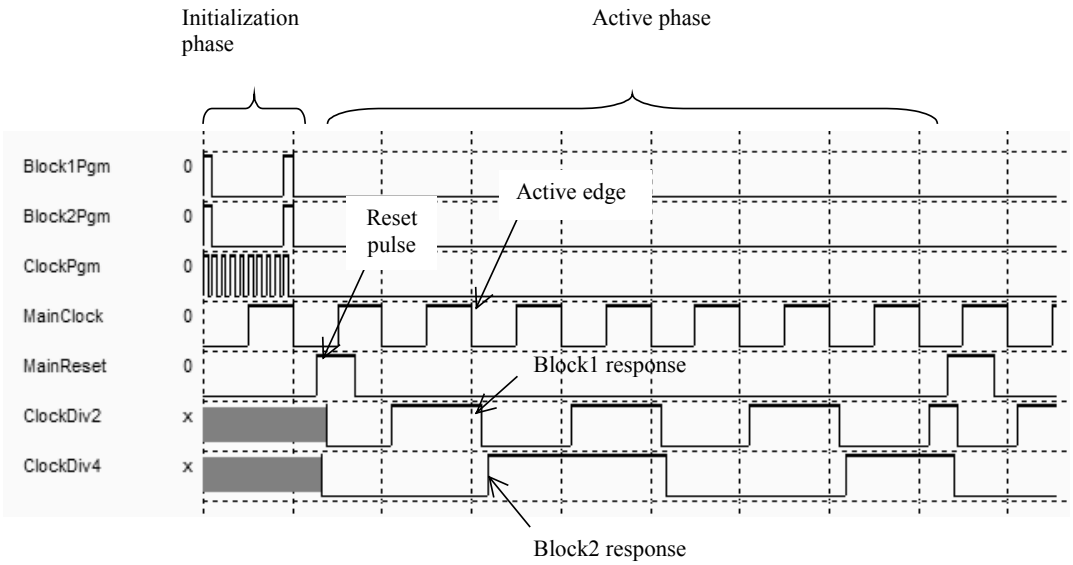


Figure 9-32: The chronograms of the clock divider circuit (ClockDiv4.SCH)

The simulation of the counter is proposed in figure 9-32. The first nanoseconds are dedicated to the programming of the blocks. Once properly configured, the counter starts to work according to the specifications of figure 9-28. Notice the very important delay in responding to the active edges, which is due to the intrinsic complexity of the configuration block, and to the long interconnect delay through the connection points and switching matrix.

9.5 Conclusion

In this chapter, we have given a brief introduction to field programmable gate arrays, from the point of view of cell design. Firstly, the use of multiplexor and look-up-tables for building configurable logic circuits has been illustrated. Secondly, the programming of memory points using chained D-registers and fuse has been described. Thirdly, we have described the programmable interconnect points and switching matrix, with their implementation in DSCH. Finally, the implementation of a full adder and a clock divider have been performed using two configurable logic blocks, programmable interconnect points and switching matrix.

References

[Smith] Michael. J. S. Smith "Application Specific Integrated Circuits", Addison Wesley, ISBN 0-201-50022-

[Sharma] <add>

[Uyemura] <add>

EXERCISES

9.1 Using DSCH (file exo9_1.sch), configure the 16 switching matrix in order to connect: switch1 to Lights L3 and L5, switch2 to Lights L1 and L6, switch3 to Lights L2 and L4, switch4 to Lights L7 and L8.

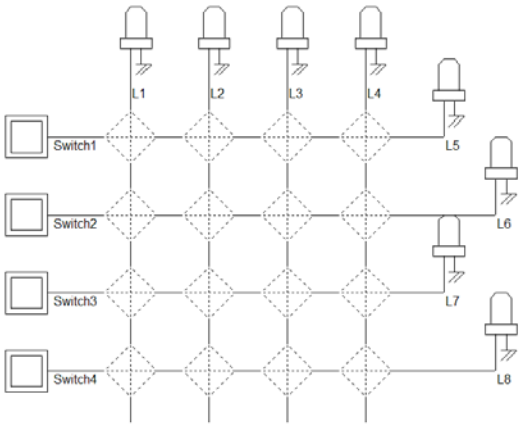


Figure 9-33: Routing exercise

Answer: see file ch91.sch

9.2 Store the 8 following bits (01110111) (reading from left to right) in the look-up-table, as in figure 9-4. How many active edges on *ClockProg* do you need to configure the LUT ? Which logical function have you realized ?

Answer: a) 8 *b)* $Fout=F2+\overline{F1\cdot F0}$

9.3 Store the 8 bits (00000111) (reading from left to right) in the LUT of figure (9-8). Demonstrate that you have realized the following logical function: $\overline{F2+(F1\cdot F0)}$. Using 2 LUT and one inverter, create the *D_Latch* below. Give the serial data sequence for *DataProg*.

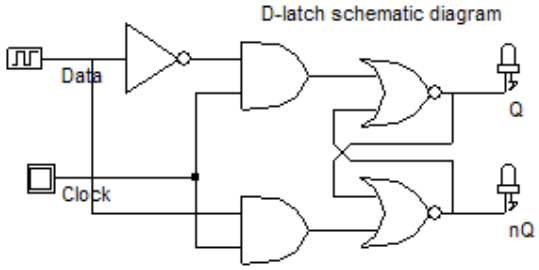


Figure 9-34: Implementing a D_latch in PFGA

Answer: a) $F0\cdot\overline{F1}\cdot F2+\overline{F0}\cdot F1\cdot F2+F0\cdot F1\cdot F2=\overline{F2+(F1\cdot F0)}$ *b) LUT N°1:* $F0=\overline{Data}$, $F1=Clock$, $F2=nQ$, *DataProg=00000111, LUT N°2:* $F0=Data$, $F1=Clock$, $F2=Q$, *DataProg=00000111*

9.4 How many programmable logic blocs do you need to create the following asynchronous counter (Figure 9-35)? Give the programmable sequences for each block.

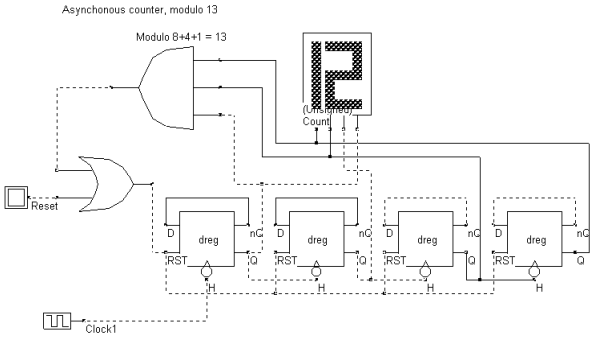


Figure 9-35: Implementing an asynchronous counter in PFGA

Answer: See ch94.SCH, where only 3 programmable logic blocs are used, as explained in figure 9-36.

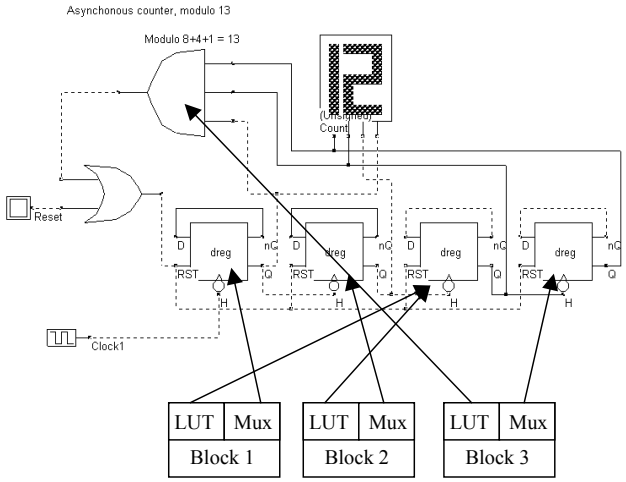


Figure 9-36: A compact solution for implementing an asynchronous counter in PFGA

9.6 Using programmable logic blocs create a one-bit comparator. The truth table is given below.

A	B	A>B	A<B	A=B
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	1

Table 9-8: The comparator