Introduction to Programmable Logic Devices

1 Arbitrary Logic Tables

Here we briefly review two strategies used in programmable logic devices to realize arbitrary truth tables in digital hardware: the sum-of-products which uses explicit ANDs and ORs, and the lookup table (LUT) method used in most modern FPGAs and CPLDs.

1.1 Sum of Products, &c.

Classic programmable logic devices, PALs, GALs, and older CPLDs, implemented logic functions using combinations of ANDs and ORs. An arbitrary function of N boolean variables may be expressed as the sum (OR) of product *minterms* or the product (AND) of sum *maxterms*, defined below.

Minterm The *minterms* associated with N boolean variables, $\{a_i\}$ are the product terms of all possible combinations of $\{a_i\}$ and $\{\bar{a}_i\}$. There are 2^N distinct terms. Perhaps an easier way to think of this is to construct the full boolean truth table and associate a minterm with each row of the table, such that the product term evaluates to true.

Maxterm The *maxterms* associated with N boolean variables, $\{a_i\}$ are the sum terms of all possible combinations of $\{a_i\}$ and $\{\bar{a}_i\}$. There are also 2^N possible combinations. Whereas the minterms rows evaluate to true, the maxterm rows evaluate to false.

For example, the minterms and maxterms of 3 logic signals A, B, and C are shown in Table 1. The truth table output P expresses whether the binary number formed by the inputs is prime (true) or not (false). One can easily deduce that the minterms are formed by placement of the variable or its complement depending on whether the corresponding entry in the table is true or false, respectively.

To find the boolean function $P(A, B, C)$, pick out the rows where $P = 1$ and sum the corresponding minterms, forming the sum-of-products, or SOP:

$$P_1 = \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}C + ABC \tag{1}$$

or, equivalently, pick the rows where $P = 0$ and multiply the corresponding maxterms, to form the product-of-sums,

A	B	C	Minterm	Maxterm	P
0	0	0	$\bar{A}\cdot\bar{B}\cdot\bar{C}$	$A+B+C$	0
0	0	1	$\bar{A}\cdot\bar{B}\cdot C$	$A+B+\bar{C}$	0
0	1	0	$\bar{A}\cdot B\cdot\bar{C}$	$A+\bar{B}+C$	1
0	1	1	$\bar{A}\cdot B\cdot C$	$A+\bar{B}+\bar{C}$	1
1	0	0	$A\cdot\bar{B}\cdot\bar{C}$	$\bar{A}+B+C$	0
1	0	1	$A\cdot\bar{B}\cdot C$	$\bar{A}+B+\bar{C}$	1
1	1	0	$A\cdot B\cdot\bar{C}$	$\bar{A}+\bar{B}+C$	0
1	1	1	$A\cdot B\cdot C$	$\bar{A}+\bar{B}+\bar{C}$	1

Table 1: Minterms and maxterms of 3 logic variables A, B, and C in a prime number truth table.

or POS:

$$P_2 = (A+B+C)(A+B+\bar{C})(\bar{A}+B+C)(\bar{A}+\bar{B}+C). \tag{2}$$

It is easy to show that $\bar{P}_1 = P_2$ using DeMorgan's identity. The above expression for P_1 can be simplified using the rules of Boolean algebra to obtain $P_1 = AC + \bar{A}B$ which is easier to extract directly from the truth table.

1.2 Look-Up Table Implementation

Another means of realizing the prime computer of Table 1 would be to treat the 3 inputs as an address into an 8-element look-up table and then storing either true or false in this memory table, as appropriate. This is the strategy taken by FPGAs and more modern CPLDs. It is trivial to determine the LUT elements given a truth table or boolean logic expression. For example, the CR muon pre-trigger discussed previously took 3 inputs:

$$T = S_1 S_2 \bar{S}_3 \tag{3}$$

This could be expressed in an 8-element LUT ($8 = 2^3$ where 3 is the number of inputs) with all elements zero except for the element at address 6 (110 in binary) if S_1 was the MSB of the address.

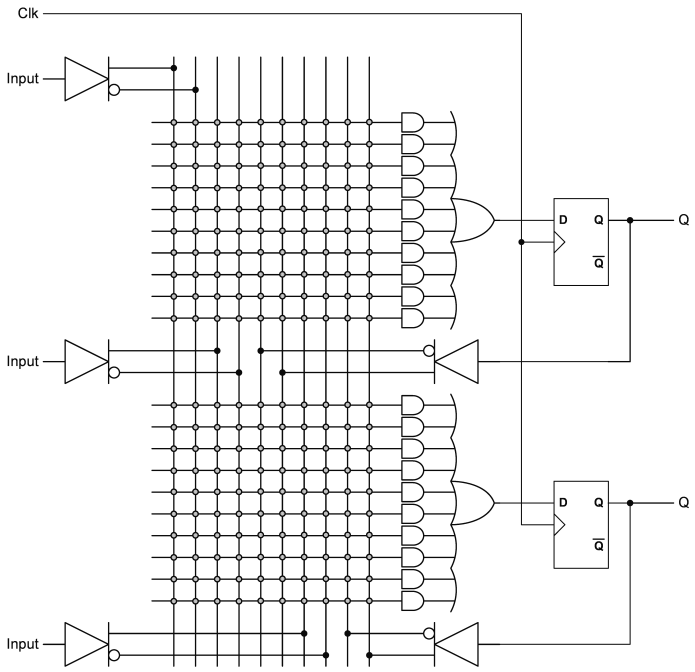


Figure 1: Typical PAL internal structure. The small shaded dots interconnecting the rows and columns are (re-)programmable switches and allow arbitrary product minterms to be presented to the summing OR gate at right.

2 Programmable Logic Types

2.1 CPLDs

Programmable array logic devices (PALs) implement the sum-of-products using a structure shown in Figure 1. External and feedback inputs are overlaid on an array of AND gates. Fuse links connect these inputs into the AND gates and can be programmed open or closed. Arbitrary product sums are then formed using the multiple input OR gates. The PAL may contain a number of registers on the outputs as well to implement sequential logic.

Complex Programmable Logic Devices, or CPLDs, evolved the registered sum-of-product structures of PAL devices to include more flexiblity, called it a *macrocell*, and then packed many macrocells connected by an intricate network of interconnects into a single IC. CPLDs replaced the fuses with switches whose state is held in nonvolatile flash RAM: once programmed the switch states persist across power down and reset conditions which is a nice convenience. The flash memory can be reprogrammed thousands of times.

CPLDs are typically distinguished from the related logic device to be taken up shortly, the *field-programmable gate array*, in their non-volatility, relatively low cost, clock speeds, logic density, and power consumption. A popular CPLD, the MAX V series from Altera was introduced in 2010 on a 180 nm process, costs between $1 and $30, contains the equivalent of between 32 and 1700 macrocells, and has maximum clocking speeds of 300 MHz.

Newer (post-2010) CPLDs have moved away from the sum-of-product logic implementation and adopt the lookup table used in FPGAs, along with other features historically found only on FPGAs: PLLs, memory, &c. They retain the non-volatile flash memory, lower power, and clock speeds of the classic CPLD.

2.2 FPGAs

The FPGA, short for field-programmable gate array, has become a popular solution for digital designers in the past two decades due to its flexibility and power. Totally reprogrammable, it is possible to purchase generic PCBs with hard-wired on-board peripherals but in addition, expansion headers that connect to off-board user-custom hardware, and develop complex systems with minimal hardware development. The programmable logic can then be configured with *firmware*[1] images which define the behavior of the digital system.

FPGAs, unlike CPLDs, are based on configurations defining the logic to be implemented held in static RAM (SRAM) rather than flash. This means that the devices must be reconfigured each time they are powered on or reset due to the volatile nature of SRAM. Despite this inconvenience, the much higher densities and speeds which can be achived with FPGAs relative to CPLDs has given them a market edge.

2.2.1 Logic Cells

The fundamental element of the FPGA is the *logic element*, or LE (Altera); *configurable logic block*, or CLB (Xilinx); or *programmable logic block*, or PLB (Lattice Semiconductor). They all function similarly, though differ in the details. A schematic of the LEs found in Altera's Cyclone IV series of FPGAs is shown in Figure 2.2.1.

Signals from the global interconnect matrix arrive from the left and exit at right. Signals at top and bottom are local (more on this later). The Cyclone IV LE's each contain a single 4-input (*i.e.*16-element) LUT and a D register. The LUT may be configured either as a single 16-element LUT or a special arithmetic mode where the LUT is split

[1] Note, firmware is also often used when discussing the software image running on an embedded microcontroller. I normally refer to microprocessor and microcontroller instruction bitstreams as software to disambiguate the various image files in cases where a soft IP microprocessor core runs on an FPGA and needs its own executable image or images.

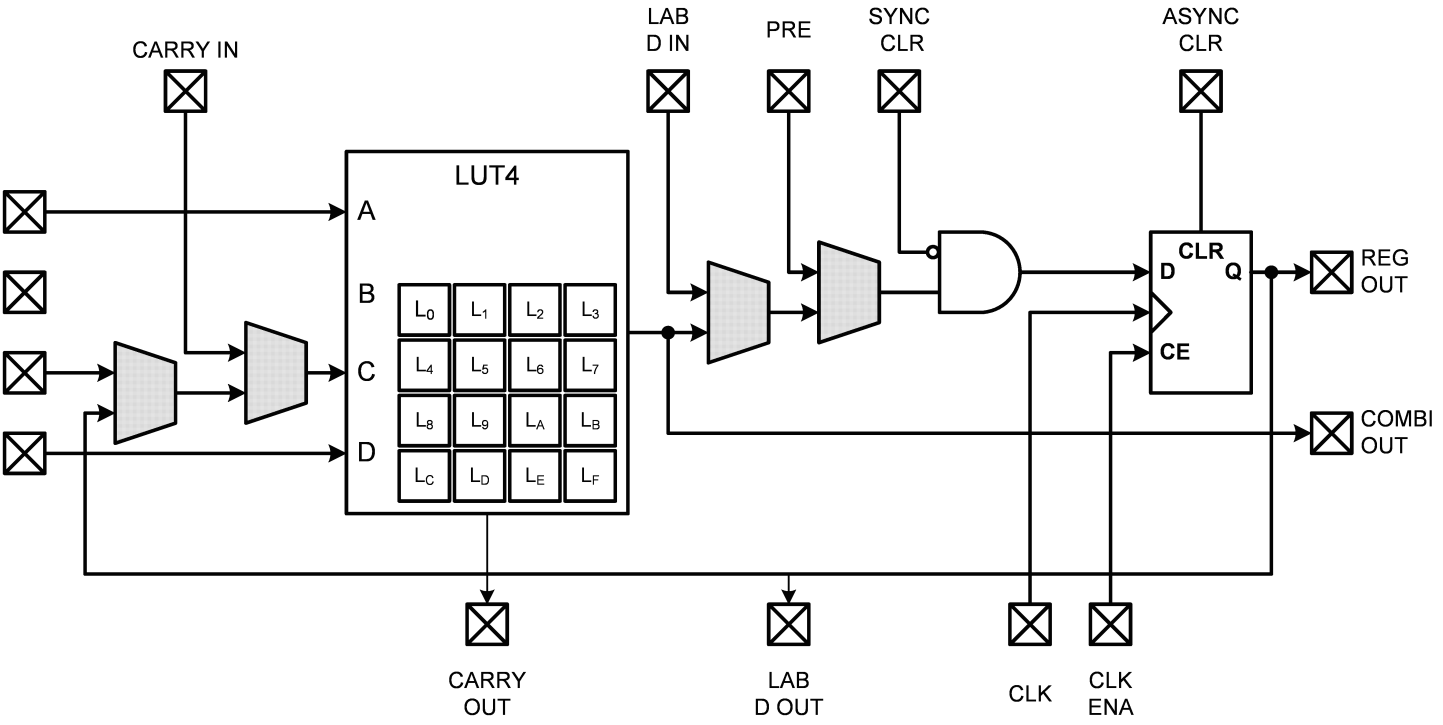


Figure 2: Logic element (LE) found in Altera's Cyclone IV series of FPGA.

into 2 8-element LUTs where in addition to the normal output which is derived from the LUT upper half there is a fast carry output derived from the lower half. This is useful for implementing adders and counters. It should be noted that the carries propagate along low propagation delay paths which increases the maximum speed of adders and counters. The shaded muxes allow configuration-time selection of signal paths (*i.e.*these muxes cannot be changed dynamically by logic). The D register have a dedicated clock enable (CE) input. Clock enables are an important technique employed in FPGA designs to control clock skew. More on this later (see section 5.1).

The LEs are arranged, in Cyclone IV devices in groups of 16, in LABs or *logic array blocks* which contain, in addition to the LEs, local interconnect lines which offer low skew connections to LEs within the same LAB. The LABs are then packed in array fashion onto the die. The layout of the smallest member of the Cyclone IV family - the EP4CE6E22 with only 6k LEs is shown in Figure 3.

2.2.2 Other Resources

In addition to LEs, the FPGA fabric contains other special functions of general use:

- Random access memory (RAM) blocks. On Altera devices these are arranged in 9kbit blocks called M9K blocks. Xilinx calls these BlockRAM and each block contains 36kbit on the newest generation "7 series" FPGAs. In both cases the RAMs may be arranged in varying word lengths. In addition to the dedicated RAM blocks, the LUTs may also be configured for distributed RAM in cases where small blocks or extra flexibility is needed.
- Hardware multipliers;
- Hard processor cores - dedicated resources which implement high-performance ARM microprocessors are offered by Altera (SoC variants of the Cyclone, Arria, and Stratix families) and Xilinx (Zynq);
- Dedicated DDR physical interfaces;
- Hardware multipliers (DSP slices);
- Digital clock management tiles (fractional-N PLLs with precision delay taps);
- Gigabit transceivers for high-speed serial interfaces;
- PCIe hard endpoints.

2.2.3 Clock Networks

Clocks are handled differently from normal logic in FPGA designs: they are routed along low-skew networks, interconnect freeways, while normal logic must travel the surface streets.

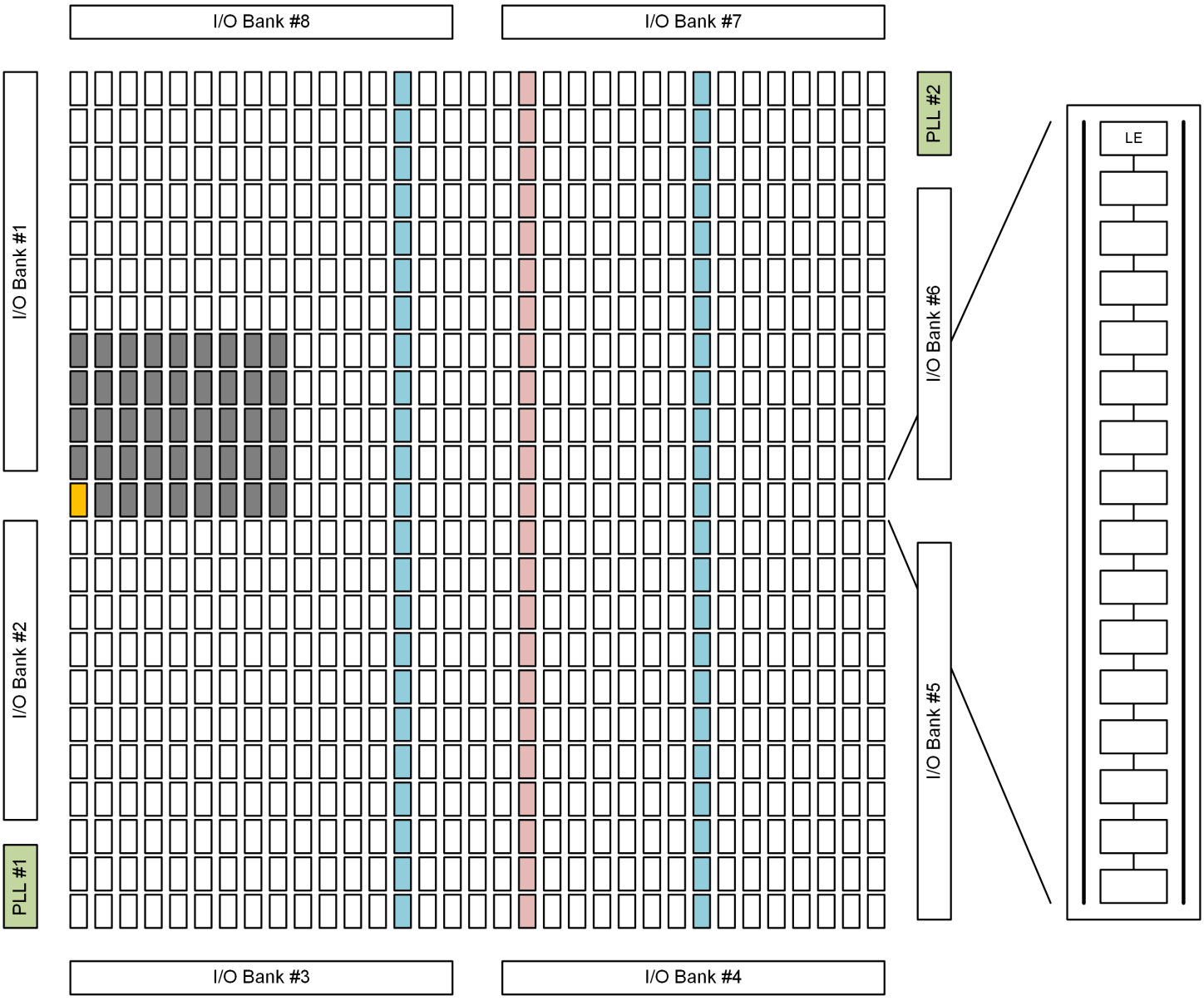


Figure 3: Structure of Altera EP4CE6E22 6k LE Cyclone IV FPGA. Each small rectangular tile in the matrix is a logic array block (LAB) which itself contains 16 logic elements (LEs) and local interconnects (zoom at right). The greyed-out tiles are unused. The two light blue strips are M9K memory blocks and the light red stripe contains the DSP multiplier blocks. Two PLLs are located in opposite corners of the die. Around the periphery are located the various I/O banks. Each I/O bank is capable of supporting several signaling standards, however, as I/Os within each bank share the power rail, designers are constrained to common signaling within a bank.

There are only a few entry points to these preferred routes where normal logic can access the clock networks. Because logic signals can accumulate a considerable delay in travelling these routes, it is strongly discouraged in firmware designs to connect the output of LUTs or registers to clock inputs. The standard work around for instances where gated clocks or clock dividers would normally be used is to use clock enables. These are described further in Section 5.1

3 Firmware Design Tutorial

Firmware development using a hardware description language (HDL) offers many advantages over schematic entry, however it presents a steep learning curve. We will plunge in, tutorial fashion: after giving some high-level guidance, the reader is presented with familiar examples of logic gates and shown how they are modeled in the two common HDLs in the hope that firmware design patterns will be recognized and generalized. Then the important subject of simulation testbenches will be used to demonstrate good design practice (test early, test often) and explore additional features of HDLs.

3.1 Three Levels of Abstraction

There are three levels on which a firmware designer can, and ought to, regard the firmware design. Starting with the most concrete there is the *technology level*: at this level the design exists mapped onto the various logic elements and other resources on the PLD. It is probably impossible to comprehend an entire design of even middling complexity at this level, nevertheless it is often necessary for reasons of optimization to examine critical path elements at this low level. Also, in order to understand the reports generated by the logic synthesis tools, resource utilization for example, some familiarity with the technology level is useful.

At the next level, detaching itself slightly from the reality of the underlying logic cells, the design exists on the RTL, *register transfer level*, or sometimes called gate level. A designer entering the firmware design in schematic capture mode inputs directly at this level. It is considerably easier to navigate a design here: hierarchical structure exists or can exist at this level so generic (*i.e.*, counters) and user-defined composed logic blocks are used to improve design readability. The design, while not a literal representation of the low-level structure, is a functionally equivalent view and, moreover, is able to be realized on the PLD, unlike some constructions found at the higher level. This may seem an odd comment to make: what's the point of designing firmware that cannot be implemented in hardware? The answer should become clearer after discussing the highest level of design abstraction.

This level is called the *behavioral modeling level* and is described using one or more textual-based HDLs as opposed to graphical-based schematic capture. The designer specifies how the logic signals are to behave and how they are linked togethers but does not explicitly use gates to do so. Statements to the effect of "make a group of signals C which are the binary sum of signals A and B" or "at the rising edge of the clock signal set the state of a state machine to some specified state if the value of an input signal is high, otherwise remain in the current state" are how the design is expressed in behavioral modeling. To be quite honest, designers are free to model at the gate or even technology level: as will soon be shown, a basic modeling concept present in all HDLs is the module which allows encapsulation of firmware sub-circuits in boxes with input and output ports. It is entirely possible[2] to define boolean gates or D or JK flip-flops as firmware modules or even use vendor-supplied modules that implement the primitive logic cells and connect these together in the firmware source text.

In my experience, design expression at the behavioral level in an HDL is the most efficient method of design entry and I see the design globally as a collection of the firmware modules. However, within a given firmware module I always seem to have a running guess at least of how the synthesizer will render the RTL.

Coming back briefly to the comment about behavioral modeling statements that are not synthesizable (*i.e.*, unable to be realized in hardware), one may well guess that there are applications of behavioral modeling beyond firmware implementation. HDLs are also used for documenting and simulating digital circuits. Only a subset of the languages are used to implement firmware and this is a particularly steeply-sloped section of the learning curve for HDLs. It is not always clear which constructs will synthesize; worse, it is not consistent across different toolchains. An intuitive understanding of the RTL representation of the behavioral model helps: if a statement would take a great number of logic gates to implement, be on guard that it may not synthesize. Having said that, most synthesizers will infer arbitrarily long adders from a single line of HDL code which adds two signals. Now that many FPGAs have dedicated hardware multipliers, inference of DSP multiplier blocks is often supported by synthesis tools. *Recommendation:* invest time reading the documentation of each tool to see what it will do.

3.2 VHDL and Verilog

VHDL (**V**ery High Speed Integrated Circuit **HDL**) and Verilog are the two most popular HDLs in use currently. These

[2] and occassionally necessary for design optimization

languages have evolved over time, again read the tool documentation to see what standard is supported. Despite an admitted personal bias toward VHDL, there is not a right choice nor a wrong one - each have strengths and weaknesses: VHDL supports some high-level constructs which Verilog does not but Verilog syntax is much more succint. Verilog syntax is close to C and thus more intuitive to a wider group of people while VHDL has an Ada-like syntax which takes some getting used to. SystemVerilog with its Verilog syntax and support for more abstraction is now supported by major toolchains and so may be the right choice for future-looking designers. To give a flavor for both VHDL and Verilog, the basic introductory portions of this tutorial on HDLs will include and compare both languages. However, the later sections will present only VHDL.

3.3 Entities and Modules

VHDL and Verilog designs are entered into text files with extension `.vhd` or `.vhdl` for VHDL, `.v` for Verilog files. By convention one file holds one design unit. The basic design unit is the **entity** (VHDL) or **module** (Verilog). All firmware designs start with the top module whose I/O ports then correspond to the physical I/O pins of the IC. Modules are then hierarchically arranged to arbitrary depth.

This tutorial starts with a familiar circuit element, the 2-input NAND gate of Figure 4.

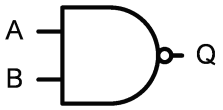


Figure 4: The 2-input NAND modeled in HDLs.

3.3.1 Verilog NAND

The following 3-line Verilog module implements the NAND gate. The syntax should be pretty obvious to those familiar with C-like languages: a module definition looks like a C function definition with the input and output signals taking the place of function arguments. Inside the module, the only action taken is the assignment of the result of the NAND operation to the output port q.

```
1 module my_nand(input a, input b, output q);
2     assign q = ~(a & b);
3 endmodule
```

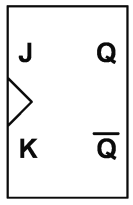
3.3.2 VHDL NAND

The equivalent code in VHDL is some four times lengthier and requires a bit more explanation. The first two lines inform the synthesizer that the IEEE code library called **std_logic_1164** must be loaded to gain access to the **std_logic** type. VHDL contains many built-in types however the type most used for logic synthesis, **std_logic**, is contained in an add-on (but omnipresent) library. VHDL makes a distinction between module interface and implementation and so requires the designer to declare the input and output ports in the **entity** declaration between lines 4 and 6, while putting the implementation in an **architecture** block, here between lines 8 and 11.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity my_nand is
5     port (a, b : in std_logic; q : out std_logic);
6 end entity my_nand;
7
8 architecture behavioral of my_nand is
9 begin
10     q <= a nand b;
11 end architecture behavioral;
```

3.4 Modeling Sequential Logic

While similar concepts exist in both languages for concurrent (combinational) and sequential logic, Verilog and VHDL differ substantially on how they handle stateful and stateless signals. Before tackling this more difficult topic, let's first cover each language's syntax to deal with sequential events: Verilog's **always** blocks and VHDL's **process** statements which are of similar nature. Again, we take a known example from the real digital world: a JK flip-flop, Figure 5.



J	K	CLK	Q_{n+1}
0	0	⎍	Q_n
0	1	⎍	0
1	0	⎍	1
1	1	⎍	Q_n

Figure 5: JK flip-flop and truth table.

3.4.1 Verilog JK Register

```
1 module jkff(
2     input wire clk,
3     input wire rst,
4     input wire j,
5     input wire k,
6     output reg q
7     );
8     always @(posedge clk, rst) begin
9         if (rst == 1) begin
10            q <= 1'b0;
11        end
```

```
12          else begin
13              case ({j,k})
14                  2'b00 : ;
15                  2'b01 : q <= 1'b0;
16                  2'b10 : q <= 1'b1;
17                  2'b11 : q <= ~q;
18                  default : ;
19              endcase
20          end
21      end
22  endmodule
```

The module's input and output ports are enumerated in lines 2-6. It is more conventional to write port lists in this manner, one port per line, as opposed to several per line unless the list fits on a single line. Note that the output port has the **reg** type because it is a stateful signal. We still have not seen explicit **wire** declarations, however, the inputs are implicitly wires possibly connecting to registers elsewhere. An alternative to declaring q a **reg** in the port list would have been to declare a register within the module and then connect an output wire to it but the style presented in the listing saves one line of typing. Inputs are always wires. The **always** keyword introduces a procedural block of statements which are evaluated sequentially, in order.

The block is triggered when one or more of the signals inside the (), called the *sensitivity list* change state. The **posedge** keyword on clk restricts the trigger to only the rising edge of clk. Inside the procedural block an **if-else** statement determines which signal triggered the block: the asynchronous reset or the clock edge. In case the asynch reset and clock edge both trigger simultaneously, the asynch reset wins. If the async reset triggers, the q output is set to logical '0': the notation 1'b0 is the Verilog way to express a 0 bit, and 1'b1 expresses a 1 bit.

If the clock edge triggers then the **else** branch is selected and the output is determined by the state of j and k. By enclosing j and k in curly braces, they get smashed together to form a 2-element bit vector which can succinctly be compared against the **case** statement cases which echo the JK's truth table (Fig. 5).

3.4.2 VHDL JK Register

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity jkff is
6      port (
7          clk : in  std_logic;
8          rst : in  std_logic;
9          j   : in  std_logic;
10         k   : in  std_logic;
11         q   : out std_logic);
12 end entity jkff;
13
14 architecture behavioral of jkff is
15     signal q_int : std_logic;
16 begin
17     q <= q_int;
18     process (clk, rst)
19     begin
20         if rst = '1' then
21             q_int <= '0';
22         elsif rising_edge(clk) then
23             case std_logic_vector'(j & k)is
24                 when "00" => null;
25                 when "01" => q_int <= '0';
26                 when "10" => q_int <= '1';
27                 when "11" => q_int <= not q_int;
28                 when others => null;
29             end case;
30         end if;
31     end process;
32 end architecture behavioral;
```

The **library** and **use** clauses are identical to the first VHDL listing (these two lines begin practically *every* VHDL source file). Like Verilog, it is more usual to write out port lists one port per line. Because VHDL enforces write-only access for **out** ports, the port read on line 25 would cause an error on synthesis if we'd tried to read from it - so an internal signal is used and the output simply connected to it. VHDL offers the **buffer** type of port, however that also has some limitations on what can be connected to it so this simple work-around is IMO preferred in the majority of cases.

VHDL **process** statements also include a sensitivity list but will trigger on every signal change, therefore the edge direction of clk is tested on line 20. You may be tempted to model a dual edge triggered register to react at double speed. This will work in simulation. Currently, however, such dual edge flip flops do not exist in FPGAs. Don't worry, there are other ways to achieve the same effect.

The same trick is used here to write a compact **case** statement to handle the JK's truth table: signals j and k are concatenated using the & operator on line 21. The resulting bit string must be type cast to **std_logic_vector** type: VHDL is a strongly typed language and requires that any type ambiguities be resolved explicitly.

3.5 Simulation Constructs

The preceeding listings were models of real gates, so by design synthesizable. We now explore models that will *not* synthesize to real gates but are rather for the sole purpose of simulating models. We will construct so-called simulation testbenches to test the functional behavior of the previous listings. This is also show how to instantiate modules, that is how to build circuit structure by inclusion of sub-modules into containing modules. Again, both Verilog and VHDL testbench drivers will be demonstrated.

3.5.1 Verilog Testbench

Testbenches are special examples of modules with no inputs and outputs – they exist only in their own isolated simulation universes. The first line of the testbench is a directive telling the simulator what is the fundamental time step in the simulation. This will become important to understand time in the # delay statements on lines 25 to 34. Otherwise the file is a normal Verilog file with the caveat that it will not produce hardware files of course.

Module local nets and registers are declared in lines 5–6. The **parameter** is a constant giving the clock period. It is defined here to define clock half period delays, described in the very next paragraph, in one place so that, if the clock period changes, it can be changed in just one place.

Clocks are simulated using continuously retriggered **always** blocks, seen in lines 9–13. At the beginning of the block the clk is set to '0'. Line 11 contains a delay statement, only useful in simulation, where the number after the # specifies how many time units, specified by the **'timescale** directive, should elapse before the statement is executed. It waits one half clock period, sets the clock high, then waits another half clock period on line 12 before repeating the endless loop.

```
1 `timescale 1ns / 1ps
2
3 module jkff_tb( );
4     parameter CLKPER = 10;
5     reg  clk, j, k, rst;
6     wire q;
7
8     // Clock generator
9     always begin
10        clk = 1'b0;
11        #(CLKPER/2) clk = 1'b1;
12        #(CLKPER/2);
13    end
14
15    // Stimulus generator
16    initial begin
17        {j, k} = 2'b00;
18        rst    = 1'b1;
19        #10 k    = 1'b1;
20        #30 rst  = 1'b0;
21        #50 j    = 1'b1;
22        #412 rst = 1'b1;
23        #32 rst  = 1'b0;
24    end
25
26    // DUT
27    jkff jk_inst(.clk(clk), .rst(rst),
28        .j(j), .k(k), .q(q));
29 endmodule
```

The **initial** block, lines 16–24, is similar to the **always** block but is executed only once. It is used here to define stimuli for the device under test (DUT). It is used in synthesizable code to set initial conditions for registers, memories, &c.

The JK flip-flop DUT is placed in lines 27–28. The ports are mapped using the syntax *.port_name(net)*, where *port_name* is the name of the port in the module definition, and *net* is the name of the wire or register in the current module to connect to that port.

3.5.2 VHDL Testbench

Similar to Verilog testbenchs, VHDL testbenches are entities with no ports. VHDL entity local signals are declared in the lines between the **architecture** keyword and the **begin**, here lines 8–21. The **component** declaration, lines 14–21, is needed to declare to VHDL that there exists an entity or module somewhere with those I/O ports. Note that the module could be written in Verilog.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity jkff_tb is
5  end entity jkff_tb;
6
7  architecture simulation of jkff_tb is
8      signal clk  : std_logic;
9      signal rst  : std_logic;
10     signal j, k : std_logic;
11     signal q    : std_logic;
12     constant CLKPER : time := 10 ns;
13
14     component jkff is
15         port (
16             clk : in  std_logic;
17             rst : in  std_logic;
18             j   : in  std_logic;
19             k   : in  std_logic;
20             q   : out std_logic);
21     end component jkff;
22 begin
23
24     -- clock generator
25     clkgen: process
26     begin
27         clk <= '0';
28         wait for CLKPER/2;
29         clk <= '1';
30         wait for CLKPER/2;
31     end process clkgen;
32
33     stimuli: process
34     begin
35         rst <= '0', '1' after 15 ns, '0' after 40 ns;
36         j <= '0', '1' after 60 ns;
37         k <= '0', '1' after 80 ns;
38         wait;
39     end process;
40
41     jk_inst: jkff port map(clk=>clk, rst=>rst,
42         j=>j, k=>k, q=>q);
43
44 end architecture simulation;
```

Lines 25–31 mirror the Verilog clock generation using VHDL **wait** statements. The stimuli are generated in another pro-

cess which terminates in a **wait** halting the process, functionally equivalent to the Verilog **initial** block. Note that in VHDL, signal assignments happen asynchronously (they act like non-blocking assignments in Verilog) so that, while rst goes high at 15 ns and then back low at 55 ns, the rising edge of j happens at 60 ns, and k at 80 ns. This is in contrast to the cumulative delays of Verilog delay statements. Not surprisingly, VHDL **wait** statements and delayed assignments are not synthesizable.

4 More on VHDL

At this point we conclude the tutorial and backup to describe the VHDL language syntax only in enough detail to understand the following sections. It's a vast language, and readers interested in a full specification of the language are referred to the bibliography, in particular the book by Ashenden.

N.B.: VHDL is not case sensitive. Verilog is.

4.1 Data Types

VHDL is a strongly typed language and provides many types, each one serving a specific purpose. Data types specify the nature of **signals** and **variables** (sec. 4.3).

4.1.1 Scalar Types

Booleans With the prior duality between digial logic and boolean algebra, one might imagine the fundamental logic type to be a boolean. A boolean type does exist, with possible values false and true, however since logic signals tend to come in groups of many bits, and character strings are less bulky typograpically than boolean arrays, booleans are not the dominant type. Nonetheless, they are common and quite useful.

Example signal declaration (with initial value):

```
signal triggered : boolean := false;
```

Bits and Standard Logic Bit types take on (character) values '0' and '1' and are easier to gang into bit vectors. However, the **std_logic** type is normally used in preference to **bit** types because logic synthesis tools need to model logic states other than **LO** and **HI**: other possible states are tri-stated (high impedance), and weak pullups or pulldowns, to name a few. To encompass models with these states, the IEEE has developed a standard package (std_logic_1164) which defines the **std_logic** type. Valid **std_logic** values are: '0' logic **LO**; '1' logic **HI**; 'Z' tri-stated; 'U' uninitialized; 'X' unknown, *i.e.* driven to different levels by multiple sources; 'L' weak pull-down; 'H' weak pull-up; '-' don't care.

std_logic objects can be used where **bit** types are used.

Example signal declarations (with initial values):

```
signal clk : bit := '0';
signal sda : std_logic := 'H';
```

Note that setting **std_logic** objects to anything other than '0' or '1' in code meant for hardware synthesis is likely to result in the synthesizer silently (!!) ignoring the assignment.

Integer Signed, whole numbers are modeled by the **integer** type. Because logic resources are often at a premium, it is common, and encouraged to restrict integers to the range needed using range constraints:

```
signal count : integer range 0 to 255;
```

The above declaration would result in allocation of only 8 flip flops instead of potentially 32. Standard VHDL defines a **natural** integer subtype which only includes positive integers and zero.

4.1.2 Array Types

Standard Logic The type **std_logic_vector** is an array of **std_logic** elements used ubiquitously to describe multibit logic arrays. It is common practice to order bit arrays with MSB on the left and LSB on the right, the way numbers are normally written. For example, a signal group to hold the 32-bit sum of two addends could be declared like this:

```
signal sum32 : std_logic_vector(31 downto 0);
```

std_logic_vector literals are bit strings delineated by double quote marks. By default the bit strings are binary base-2 but can be written as hexadecimal base-16 by prepending an x before the bit string:

```
sum32 <= X"4000C8B3";
```

which is equivalent to

```
sum32 <= X"01000000000000001100100010110011";
```

Individual elements can be accessed read or write:

```
if sum8(11) = '1' then
        -- true
end if;
```

as can entire slices

```
if sum32(15 downto 12) = "1100" then
        -- also true
end if;
```

Signed and Unsigned The **signed** and **unsigned** types are used for modeling numeric computations. They are bit strings, not integers, however the packages which define them, numeric_std and numeric_bit, define a number of operators which allow bitwise and arithmetic operations between signed, unsigned, and integer types.

```
constant u1 : unsigned(7 downto 0) := x"44";
constant u2 : unsigned(3 downto 0) := b"1010";

if u2 < 10 then
    -- not true
end if;
```

4.1.3 User-Defined Types

Enumerations Enumerated data types specify a type which can taken on a small number of discrete values. Finite state machine states are the textbook example of uses for enumerated types. The type definition defines a type which is later attached to a data object:

```
type state_t is (idle, edge, waiting);
signal s : state_t := idle;
```

4.2 Operators

4.2.1 Logical Operators

Taking bit or bit vector arguments of equal length and returning bit or bit vector, the following logical operators compute the named logic operation: **not**, **and**, **nand**, **or**, **nor**, **xor**, and **xnor**. Examples:

```
signal a : std_logic_vector(3 downto 0) := "1010";
signal b : std_logic_vector(3 downto 0) := "0011";
signal c : std_logic_vector(3 downto 0);
signal d : std_logic_vector(3 downto 0);
signal e : std_logic_vector(3 downto 0);
signal f : std_logic_vector(3 downto 0);

c <= not b;           -- result = "1100"
d <= a xor b;         -- result = "1101"
e <= a nor b;         -- result = "0100"
f <= b and (c or d);  -- result = "0001"
```

4.2.2 Arithmetic Operators

Addition and subtraction via the '+' and '-' operators works on **integer** and **signed** and **unsigned** types and generally is synthesizable. It is possible with a bit of extra typing to add and subtract **std_logic_vector** types (Sec. 4.4). Multiplication ('*' operator) of integer and bit types often infers the correct hardware strutures, with the special case of power-of-two multiplication always being OK as it is equivalent to a shift operation. Division ('/'), modulus (**mod**), and remainder (**mod**) will synthesize to division and modulus logic in the Quartus synthesizer:

```
signal w : unsigned(15 downto 0) := x"3f0a";
signal x : unsigned(15 downto 0) := x"0049";
signal y : unsigned(15 downto 0);
signal z : unsigned(31 downto 0);
y <= w / x;
z <= w * x;
```

should work but will consume a fair number of logic resources.

4.2.3 Shift Operators

Two directions (left, right) and three varieties (rotations, logical, arithmetic) of shift operation work on bit vectors. Here is the list:

rol Rotate bits left N places, bits which fall off the left hand side return to fill the rightmost bit;

ror Rotate bits right N places, new bits fill leftmost bit from bits exiting on right;

sll Logical shift left. Bits shift N places to the left filling right bits with zeros;

srl Logical shift right. Bits shift N places to the right filling left bits with zeros;

sla Arithmetic shift left. Bits shift N places to the left. The rightmost bit holds its state and is propagated to the neighboring bits in the shift;

sra Arithmetic shift right. Bits shift N places to the right. The leftmost bit holds its state and is propagated to the neighboring bits in the shift.

4.2.4 Comparison Operators

The comparison operations are equality: =; inequality: /=; less than: <; less than or equal: <=; greater than: >; greater than or equal: >=. All return boolean.

4.2.5 Concatenation Operator

The & operator concatenates two bit vectors:

```
signal c1 : std_logic_vector(5 downto 0) := "101110";
signal c2 : std_logic_vector(2 downto 0) := "010";
signal c3 : std_logic_vector(8 downto 0);
c3 <= c1 & c2;  -- result = "101110010"
```

4.3 Signals and Variables

Now to address the difficult subject of VHDL signals and variables. Unlike Verilog, VHDL makes no distinction between stateless connections and stateful signals: all physical signals are modeled as **signal** objects. Signals are declared in the lines before the **begin** line of **architecture** blocks. Input and output ports of entities are signals. Signals are assigned values, either literal values or the results of operations using the signal assignment operator, <= (yes, it looks like less than or equal), or by attachment to ports in module instantiations. They can be assigned within **process** statements but this is where it gets weird: signal assignments in processes don't occur immediately but rather when the process reaches the suspend state[3] Sometimes this gets in the way, so another data object, the VHDL **variable**, must be used.

Variables are declared and exist only within a specific process. If the value of the variable needs to be communicated outside the process, it must be assigned to a signal. Variables can take on any type a signal can. Variables do update immediately. The variable assignment operator is :=. The use of variables is illustrated in Section **??**.

4.4 Standard Logic Arithmetic

Frequently, it is necessary to perform arithmetic operations on **std_logic_vector** types. It's a bit bulky, however, casting to either **signed** or **unsigned** types is the right way to do this. Don't forget to cast back to **std_logic_vector**:

```
signal x, y : std_logic_vector(9 downto 0);
signal z : std_logic_vector(19 downto 0);
z <= std_logic_vector(signed(x)*signed(y));
```

5 VHDL Design Patterns

5.1 Clock Enable

Often a divided clock or gated clock is needed. Clock signals are routed differently than other logic, along special low-skew lines, because of their critical role in synchronous circuits and timing is hard to control when mixing clocks and logic and is thus not recommended. Looking back to Figure 2.2.1, the register features an enable input: the edge trigger only functions when the CE input is high. By controlling the state of the enable, functionality equivalent to clock gating or division can be achieved. A VHDL module to generate clock enables is presented in the code listing below.

[3] A side-effect of the process model: processes execute sequentially but the time steps should be regarded as arbitrarily small.

```
library ieee;
use ieee.std_logic_1164.all;

entity clkenable is
    generic (MODULUS : integer);
    port ( clk : in  std_logic;
           ce  : out std_logic);
end clkenable;

architecture behavioral of clkenable is
begin
    clken: process (clk)
        variable count : integer range
                0 to MODULUS-1 := 0;
    begin
        if rising_edge(clk) then
            ce <= '0';
            if count = MODULUS-1 then
                count := 0;
                ce <= '1';
            else
                count := count + 1;
            end if;
        end if;
     end process;
end behavioral;
```

5.2 Example: CR Muon Trigger

Here is presented the firmware solution to the problem of triggering on a muon decay in flight. The testdeck:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity crmu_tb is
end crmu_tb;

architecture Behavioral of crmu_tb is
    constant CLOCK_PERIOD : time := 10 ns;
    signal clk      : std_logic := '0';
    signal pmt      : std_logic_vector(2 downto 0);
    signal daq_trig : std_logic;
    component muon_decay_trigger is
        port (
            clk     : in  std_logic;
            s       : in  std_logic_vector(2 downto 0);
            trig    : out std_logic);
    end component muon_decay_trigger;
begin
    clkgen: process (clk)
    begin
        clk <= not clk after CLOCK_PERIOD/2;
    end process clkgen;

    physics: process
    begin
        pmt(0) <= '0', '1' after 500 ns,
                  '0' after 700 ns;
        pmt(1) <= '0', '1' after 510 ns,
                  '0' after 650 ns;
        pmt(2) <= '0', '1' after 8000 ns,
                  '0' after 8200 ns;
        wait;
    end process physics;
```

```
35
36      trigger: muon_decay_trigger port map (
37          clk=>clk, s=>pmt, trig=>daq_trig);
38  end Behavioral;
```

And the trigger code

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  use ieee.numeric_std.all;
5
6  entity muon_decay_trigger is
7      port (
8          clk  : in std_logic;
9          s    : in std_logic_vector (2 downto 0);
10         trig : out std_logic);
11 end muon_decay_trigger;
12
13 architecture behavioral of muon_decay_trigger is
14
15     type mu_state_t is (
16         idle, wait500, wait15k,
17         triggered);
18     signal state : mu_state_t := idle;
19     signal t, u  : std_logic;
20
21 begin
22     t <= s(0) and s(1) and not s(2);
23     u <= s(1) or s(2);
24     trig <= '1' when state = triggered else '0';
25
26     states: process (clk)
27         variable count : integer range 0 to 1499;
28     begin
29         if rising_edge(clk) then
30             case state is
31                 when idle =>
32                     if t = '1' then
33                         state <= wait500;
34                         count := 49;
35                     end if;
36                 when wait500 =>
37                     if count /= 0 then
38                         count := count - 1;
39                     else
40                         state <= wait15k;
41                         count := 1499;
42                     end if;
43                 when wait15k =>
44                     if u = '1' then
45                         state <= triggered;
46                     elsif count /= 0 then
47                         count := count - 1;
48                     else
49                         state <= idle;
50                     end if;
51                 when triggered =>
52                     state <= idle;
53             end case;
54         end if;
55     end process states;
56
57 end behavioral;
```

5.3 Fixed-Point Math

There may arise the need to perform mathematical operations on real physical quantities in firmware. The DE0 Nano board, for example, contains an accelerometer which reads out 10-bit signed numbers with a range of $\pm 2g$. The mapping from digital count value, A to real acceleration value, a is

$$a = \left(\frac{A}{256}\right) g \tag{4}$$

This is known as *fixed point* representation of reals. How can you use this acceleration measurement to compute derived physical quantities such as the velocity and the position? First, let's consider velocity. I suppose that you know how to do this in theory:

$$v = v_0 + a\Delta t \tag{5}$$

The accelerometer reads a value every 0.02 s: how can we calculate the velocity change? This is tantamount to asking how to represent the Δt quantity? We expect the range of possible Δts to be small, actually fixed at 0.02 s because the FPGA should never skip a beat. So really there is only an implict multiplication going on here; combining equations 4 and 5:

$$v = v_0 + (0.766\,\mathrm{mm/s}) \cdot A \tag{6}$$

Likewise to compute the position of the DE0 board,

$$x = x_0 + v\Delta t = x_0 + (0.02\,\mathrm{s}) \cdot v \tag{7}$$

again substituting the fixed interval, $\Delta t = 0.02\,\mathrm{s}$.

The procedure for computing the realtime velocity and position of the DE0 is then the following:

- Initialize the position and velocity to zero;
- Readout the acceleration, A;
- Update the velocity: $v_n \leftarrow v_{n-1} + A$;
- Update the position: $x_n \leftarrow x_{n-1} + v_{n-1}$

In VHDL, this looks like the following. Assuming the following declarations in the declarative part of the architecture,

```
signal ax : std_logic_vector(9 downto 0);
signal vx : signed(15 downto 0);
signal x  : signed(15 downto 0);
```

The statements to update the position are:

```
update: process (clk)
begin
    if rising_edge(clk) and accel_ok = '1' then
        vx <= vx + signed(ax);
        x  <= x + vx;
    end if;
end process update;
```

So far, it has not even been necessary to do any multiplications! The units attached to the velocities and positions are 0.766 mm/s for velocity and 15.32 μm. To convert to more useful units, millimeters, it is necessary to divide by 65.274 (or, equivalently, multiply by 0.01532). This is pretty close to a power of two so you might consider your requirements on accuracy: if the application can tolerate a 2% error, the simplest approximation is to divide by 64, or equivalently, shifting the bits to the right six steps:

```
x_in_near_mm <= x(15 downto 6);
```

If higher accuracy is needed, the least expensive method is to use a rational approximation of the scaling factor where the denominator is a power of two, again we can profit from the ability to shift right instead of dividing[4]. Luck is on our side: 0.01532 is very close to the rational number $^{251}/_{16384}$. We can get an accuracy of %0.001 (way better than the acceleration measurement, by the way, so be on guard to not overestimate your accuracy!) by multiplying by 251 and then right shifting 14 bits. If we want to use the Cyclone IV's 9×9 bit multiplier hardware, the most efficient VHDL transformation is:

```
tmp  := to_signed(251, 9) * x(15 downto 7);
x_mm <= tmp(15 downto 7);
```

where **tmp** is an 18-bit signed **variable** defined to capture the result of the 9×9 multiplication:

```
variable tmp : signed(17 downto 0);
```

Looking at the synthesis reports I see that the synthesizer has even gotten around using the multipliers! How? Multiplying x by 251 is equivalent to multiplying x by 256 (*i.e.* left shifting by 8) and then subtracting x times 4 and finally again subtracting x. Keep that trick in your pocket in case you are using PLDs without hardware multipliers.

5.4 Pseudorandom Numbers

(*see Horowitz and Hill section 9.33 pg 655*)
Apparently random sequences of bits can be generated using a FSM configuration known as the *linear feedback shift register* (LFSR). It is simply a shift register whose input bit is determined by taking XORs of various bit positions further into the register, these are often called *taps*. For example a 4-bit LFSR could be constructed like this:

$$Q_3 \leftarrow Q_0 \oplus Q_1 \tag{8}$$

This would then produce the bit sequence

0110, 1011, 0101, 1010, 1101, 1110, 1111, 0111
0011, 0001, 1000, 0100, 0010, 1001, 1100, 0110

which then repeats. Hardly a random sequence, but if you make it long enough it looks random. In general it is possible using an n bit register to produce an sequence of repetition length $2^n - 1$. Note that the sequence all zeros must be excluded to prevent the sequence from getting stuck in that state.

Getting the taps right is not trivial unless you are familiar with advanced algebra. Fortunately several magic taps have been tabulated[5]. Horowitz and Hill gives several suggestions for various LFSR lengths.

Producing random integers from the random bits is not difficult. Returning to the random bits above by taking every 4th clock cycle (to avoid explicit bit correlations), a repeating sequence of integers ranging from 1 to 15 can be produced:

6, 13, 3, 2, 11, 14, 1, 9, 5, 15, 8, 12, 10, 7, 4, 6, ...

Often, random numbers in the real interval $(0, 1)$ are needed. In the spirit of fixed-point numbers, discussed above, the random integer sequences can be interpreted as having implicit scale factors. For example, the sequence above could be interpreted as representing the numbers

0.3750, 0.8125, 0.1875, 0.1250,
0.6875, 0.8750, 0.0625, 0.5625,
0.3125, 0.9375, 0.5000, 0.7500,
0.6250, 0.4375, 0.2500, 0.3750...

5.5 ROMs

Read Only Memories are useful constructions to hold constants and implement functions. Let's start just by introducing the syntax for forcing VHDL to synthesize a ROM using distributed storage elements (*i.e.* LEs). The ROM is just an array of **std_logic_vector** types (array of bit arrays). The VHDL syntax requires a 2-step definition; first, define the array element type, then define the array type and finally declare a concrete **constant** object of this array type (lines 12–14 in the listing below):

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity romexa is
5      port (
6          clk  : in std_logic;
7          ce   : in std_logic;
8          data : out std_logic_vector (7 downto 0));
9  end romexa;
10
11 architecture behavioral of romexa is
12     subtype word_t is std_logic_vector(7 downto 0);
13     type rom_t is array(0 to 31) of word_t;
14     constant mem : rom_t := (
```

[4] As a general rule, avoid expensive divisions whenever possible.

[5] Of course, this is a security issue - don't use these popular numbers for crypto applications othewise your ciphers will be easily cracked!

```
        ("10000000"), ("01000000"),
        ("00100000"), ("00010000"),
        ("00001000"), ("00000100"),
        ("00000010"), ("00000001"),
        ("00000010"), ("00000100"),
        ("00001000"), ("00010000"),
        ("00100000"), ("01000000"),
        ("00100000"), ("00010000"),
        ("00001000"), ("00000100"),
        ("00000010"), ("00000100"),
        ("00001000"), ("00010000"),
        ("00100000"), ("00010000"),
        ("00000100"), ("00001000"),
        ("00010000"), ("00011000"),
        ("00100100"), ("01000010"),
        ("10000001"), ("00000000"));
begin
    process (clk)
        variable addr : integer range 0 to 31 := 0;
    begin
        if rising_edge(clk) and ce = '1' then
            data <= mem(addr);
            addr := (addr + 1) mod 32;
        end if;
    end process;

end behavioral;
```

ROMs can be filled with more complicated patterns. An example of this is used in the gaussian entity of the LED Snow project. The method is to define a VHDL function which generates (at compile time, so you *can* use **real** variables) the ROM contents programmatically.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

entity gaussian is
    port (
        u, v   : in  std_logic_vector(8 downto 0);
        n1, n2 : out std_logic_vector(17 downto 0)
    );
end entity gaussian;

architecture rom of gaussian is
    subtype word_t is signed(8 downto 0);
    type mem_t is array(0 to 511) of word_t;

    function init_logrom return mem_t is
        variable r, u : real;
                variable tmp : mem_t := (others =>
                        (others => '0'));
        begin
        for i in 0 to 511 loop
            u := (real(i)+0.5) / 512.0;
            r := sqrt(-2.0*log(u));
            tmp(i) := to_signed(integer(r*64.0), 9);
        end loop;
        return tmp;
        end init_logrom;

        function init_sinerom return mem_t is
        variable r, v : real;
                variable tmp : mem_t := (others =>
                        (others => '0'));
        begin
        for i in 0 to 511 loop
            v := (real(i)+0.5) / 512.0;
            r := sin(2.0*MATH_PI*v);
            tmp(i) := to_signed(integer(r*128.0), 9);
        end loop;
        return tmp;
        end init_sinerom;

    signal sine     : mem_t := init_sinerom;
    signal ln       : mem_t := init_logrom;
    signal x, y, z  : word_t;
begin
    x <= ln(to_integer(unsigned(u)));
    y <= sine(to_integer(unsigned(v)));
    z <= sine((to_integer(unsigned(v))+128) mod 512);
    n1 <= std_logic_vector(x*z);
    n2 <= std_logic_vector(x*y);
end architecture rom;
```