1. Introduction

This chapter gives an overview of the algorithms and technology we will discuss in the book. It starts with an introduction to digital signal processing and we will then discuss FPGA technology in particular. Finally, the Altera EP2C35F672C6 and a larger design example, including chip synthesis, timing analysis, floorplan, and power consumption, will be studied.

1.1 Overview of Digital Signal Processing (DSP)

Signal processing has been used to transform or manipulate analog or digital signals for a long time. One of the most frequent applications is obviously the *filtering* of a signal, which will be discussed in Chaps. 3 and 4. Digital signal processing has found many applications, ranging from data communications, speech, audio or biomedical signal processing, to instrumentation and robotics. Table 1.1 gives an overview of applications where DSP technology is used [6].

Digital signal processing (DSP) has become a mature technology and has replaced traditional analog signal processing systems in many applications. DSP systems enjoy several advantages, such as insensitivity to change in temperature, aging, or component tolerance. Historically, analog chip design yielded smaller die sizes, but now, with the noise associated with modern submicrometer designs, digital designs can often be much more densely integrated than analog designs. This yields compact, low-power, and low-cost digital designs.

Two events have accelerated DSP development. One is the disclosure by Cooley and Tuckey (1965) of an efficient algorithm to compute the discrete Fourier Transform (DFT). This class of algorithms will be discussed in detail in Chapter 6. The other milestone was the introduction of the programmable digital signal processor (PDSP) in the late 1970s, which will be discussed in Chap. 9. This could compute a (fixed-point) "multiply-and-accumulate" in only one clock cycle, which was an essential improvement compared with the "Von Neuman" microprocessor-based systems in those days. Modern PDSPs may include more sophisticated functions, such as floating-point multipliers, barrelshifters, memory banks, or zero-overhead interfaces to A/D and D/A converters. EDN publishes every year a detailed overview of available PDSPs

Table 1.1. Digital signal processing applications.

Area	DSP algorithm
General-purpose	Filtering and convolution, adaptive filtering, detection and correlation, spectral estimation and Fourier transform
Speech processing	Coding and decoding, encryption and decryption, speech recognition and synthesis, speaker identification, echo cancellation, cochlea-implant signal processing
Audio processing	hi-fi encoding and decoding, noise cancellation, audio equalization, ambient acoustics emulation, audio mixing and editing, sound synthesis
Image processing	Compression and decompression, rotation, image transmission and decompositioning, image recognition, image enhancement, retina-implant signal processing
Information systems	Voice mail, facsimile (fax), modems, cellular telephones, modulators/demodulators, line equalizers, data encryption and decryption, digital communications and LANs, spread-spectrum technology, wireless LANs, radio and television, biomedical signal processing
Control	Servo control, disk control, printer control, engine control, guidance and navigation, vibration control, power-system monitors, robots
Instrumentation	Beamforming, waveform generation, transient analysis, steady-state analysis, scientific instrumentation, radar and sonar

[7]. We will return in and Chap. 2 (p. 116) and Chap. 9 to PDSPs after we have studied FPGA architectures.

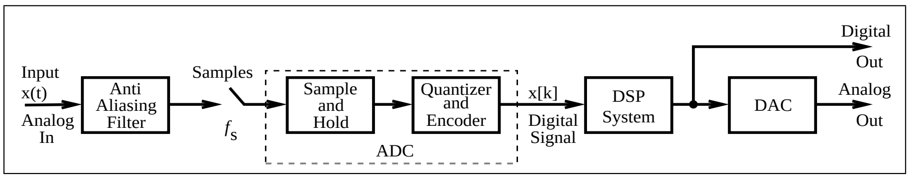


Fig. 1.1. A typical DSP application.

Figure 1.1 shows a typical application used to implement an analog system by means of a digital signal processing system. The analog input signal is feed through an analog anti aliasing filter whose stopband starts at half the sampling frequency f_s to suppress unwonted mirror frequencies that occur during the sampling process. Then the analog-to-digital converter (ADC) follows that typically is implemented with a sample-and-hold and a quantize (and encoder) circuit. The digital signal processing circuit perform then the steps that in the past would have been implemented in the analog system. We may want to further process or store (i.e., on CD) the digital processed data, or we may like to produce an analog output signal (e.g., audio signal) via a digital-to-analog converter (DAC) which would be the output of the equivalent analog system.

1.2 FPGA Technology

VLSI circuits can be classified as shown in Fig. 1.2. FPGAs are a member of a class of devices called field-programmable logic (FPL). FPLs are defined as programmable devices containing repeated fields of small logic blocks and elements[2]. It can be argued that an FPGA is an ASIC technology since FPGAs are application-specific ICs. It is, however, generally assumed that the design of a classic ASIC required additional semiconductor processing steps beyond those required for an FPL. The additional steps provide higher-order ASICs with their performance and power consumption advantage, but also with high nonrecurring engineering (NRE) costs. At 65 nm the NRE cost are about $4 million, see [8] . Gate arrays, on the other hand, typically consist of a "sea of NAND gates" whose functions are customer provided in a "wire list." The wire list is used during the fabrication process to achieve the distinct definition of the final metal layer. The designer of a *programmable* gate array solution, however, has full control over the actual design implementation without the need (and delay) for any physical IC fabrication facility. A more detailed FPGA/ASIC comparison can be found in Sect. 1.3, p. 10.

1.2.1 Classification by Granularity

Logic block size correlates to the *granularity* of a device that, in turn, relates to the effort required to complete the wiring between the blocks (routing channels). In general three different granularity classes can be found:

- Fine granularity (Pilkington or "sea of gates" architecture)
- Medium granularity (FPGA)
- Large granularity (CPLD)

Fine-Granularity Devices

Fine-grain devices were first licensed by Plessey and later by Motorola, being supplied by Pilkington Semiconductor. The basic logic cell consisted of a

[2] Called configurable logic block (CLB) by Xilinx, logic cell (LC) or logic elements (LE) by Altera.

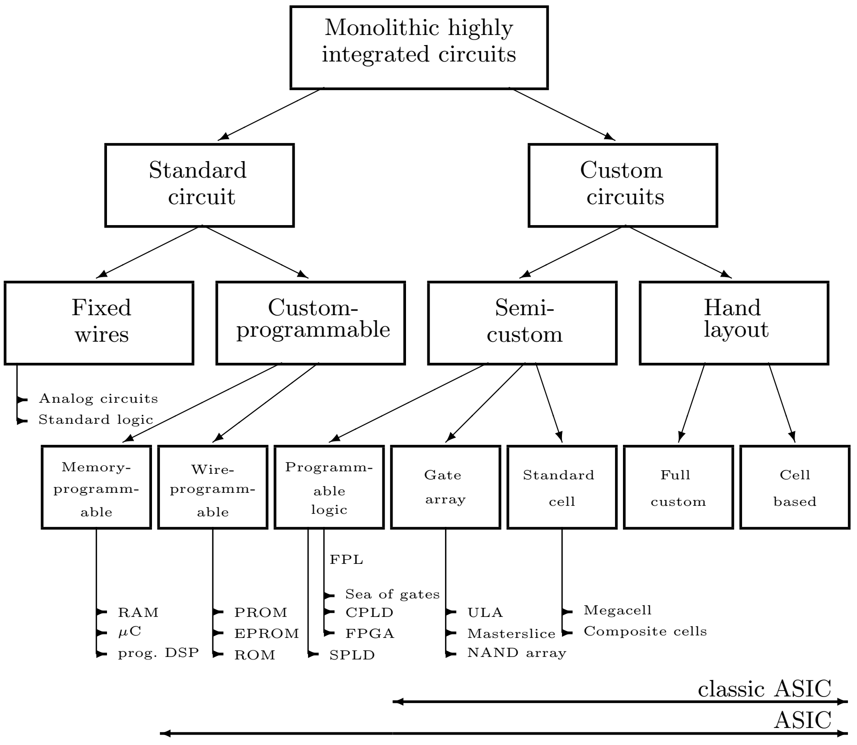
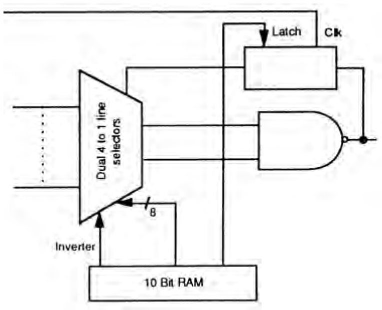


Fig. 1.2. Classification of VLSI circuits (©1995 VDI Press [4]).

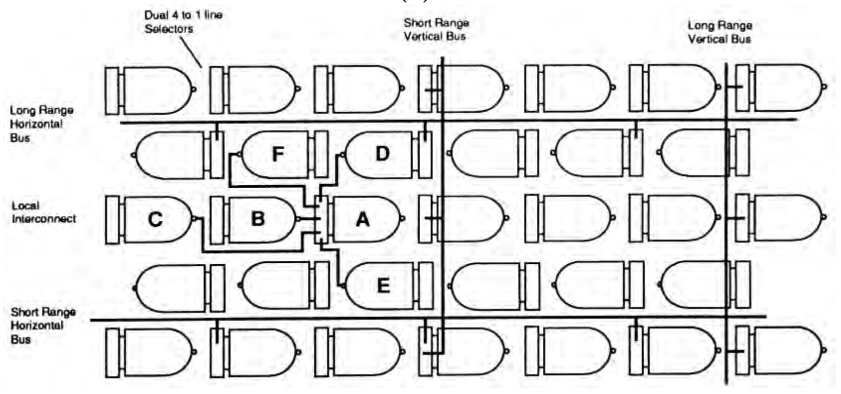
single NAND gate and a latch (see Fig. 1.3). Because it is possible to realize any binary logic function using NAND gates (see Exercise 1.1, p. 42), NAND gates are called *universal* functions. This technique is still in use for gate array designs along with approved logic synthesis tools, such as ESPRESSO. Wiring between gate-array NAND gates is accomplished by using additional metal layer(s). For programmable architectures, this becomes a bottleneck because the routing resources used are very high compared with the implemented logic functions. In addition, a high number of NAND gates is needed to build a simple DSP object. A fast 4-bit adder, for example, uses about 130 NAND gates. This makes fine-granularity technologies unattractive in implementing most DSP algorithms.

Medium-Granularity Devices

The most common FPGA architecture is shown in Fig. 1.4a. A concrete example of a contemporary medium-grain FPGA device is shown in Fig. 1.5. The elementary logic blocks are typically small tables (e.g., Xilinx Virtex with 4- to 5-bit input tables, 1- or 2-bit output), or are realized with ded-



(a)



Dual 4 to 1 line Selectors
Short Range Vertical Bus
Long Range Vertical Bus
Long Range Horizontal Bus
Local Interconnect
Short Range Horizontal Bus
F
D
C
B
A
E

(b)

Fig. 1.3. Plessey ERA60100 architecture with 10K NAND logic blocks [9]. **(a)** Elementary logic block. **(b)** Routing architecture (©1990 Plessey).

icated multiplexer (MPX) logic such as that used in Actel ACT-2 devices [10]. Routing channel choices range from short to long. A programmable I/O block with flip-flops is attached to the physical boundary of the device.

Large-Granularity Devices

Large granularity devices, such as the complex programmable logic devices (CPLDs), are characterized in Fig. 1.4b. They are defined by combining so-called simple programmable logic devices (SPLDs), like the classic GAL16V8 shown in Fig. 1.6. This SPLD consists of a programmable logic array (PLA) implemented as an AND/OR array and a universal I/O logic block. The SPLDs used in CPLDs typically have 8 to 10 inputs, 3 to 4 outputs, and support around 20 product terms. Between these SPLD blocks wide busses (called programmable interconnect arrays (PIAs) by Altera) with short delays are available. By combining the bus and the fixed SPLD timing, it is possible to provide predictable and short pin-to-pin delays with CPLDs.

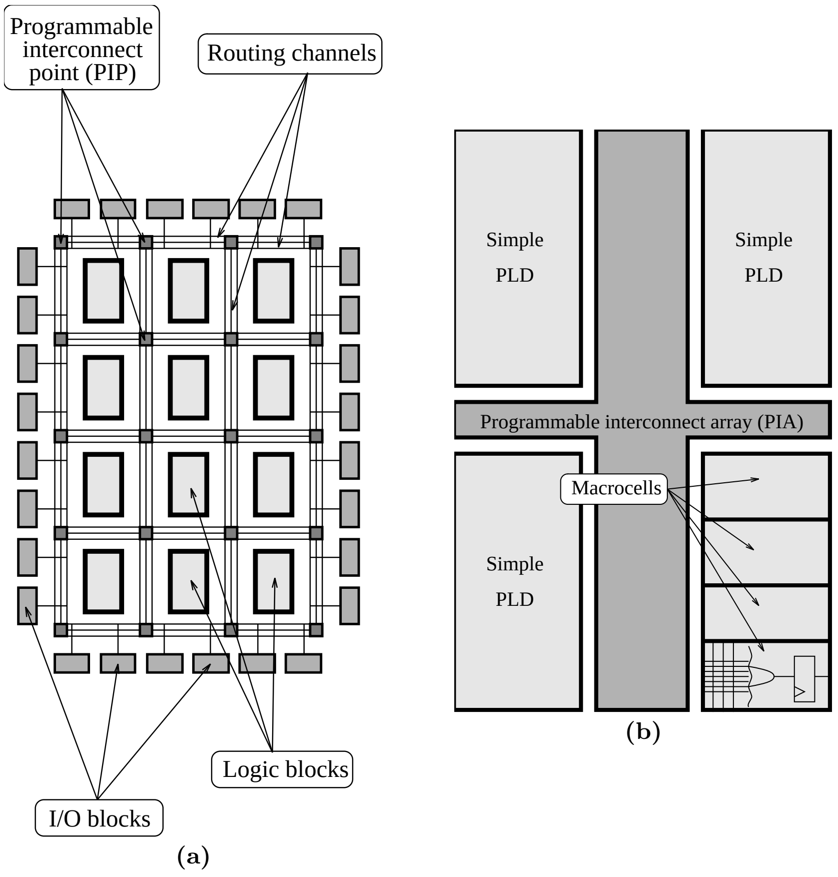


Fig. 1.4. **(a)** FPGA and **(b)** CPLD architecture (©1995 VDI Press [4]).

1.2.2 Classification by Technology

FPLs are available in virtually all memory technologies: SRAM, EPROM, E^2PROM, and antifuse [11]. The specific technology defines whether the device is *reprogrammable* or *one-time programmable*. Most SRAM devices can be programmed by a single-bit stream that reduces the wiring requirements, but also increases programming time (typically in the ms range). SRAM devices, the dominate technology for FPGAs, are based on static CMOS memory technology, and are re- and in-system programmable. They require, however, an external "boot" device for configuration. Electrically programmable read-only memory (EPROM) devices are usually used in a one-time CMOS programmable mode because of the need to use ultraviolet light for erasure. CMOS electrically erasable programmable read-only memory (E^2PROM) can be used as re- and in-system programmable. EPROM and E^2PROM have the advantage of a short setup time. Because the programming information is

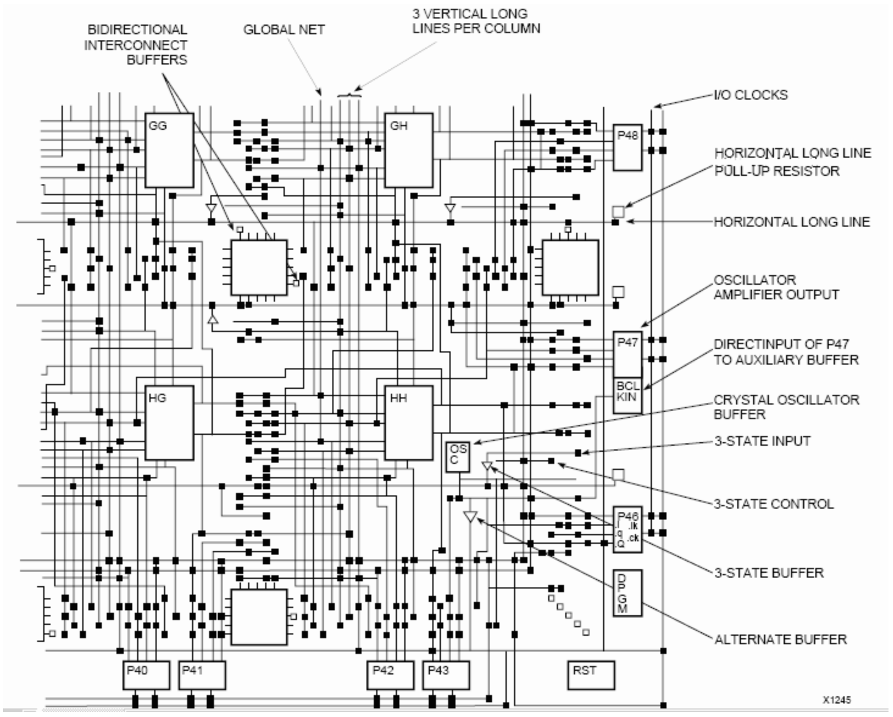
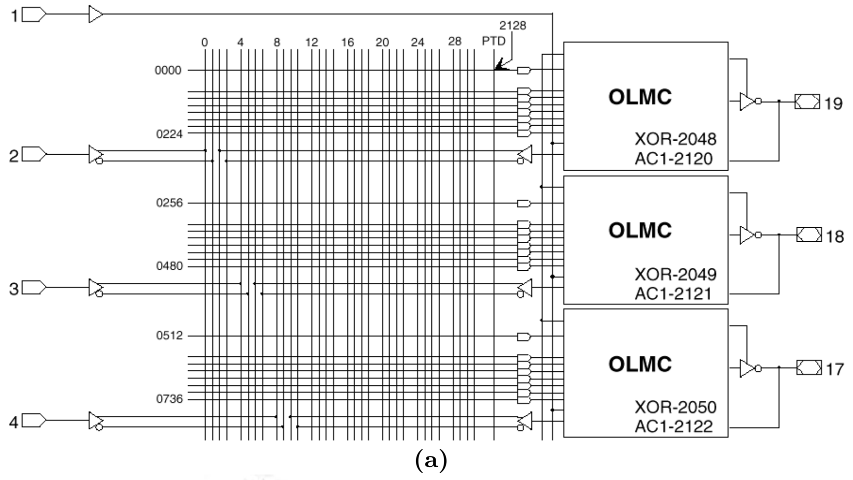

BIDIRECTIONAL INTERCONNECT BUFFERS
GLOBAL NET
3 VERTICAL LONG LINES PER COLUMN
I/O CLOCKS
HORIZONTAL LONG LINE PULL-UP RESISTOR
HORIZONTAL LONG LINE
OSCILLATOR AMPLIFIER OUTPUT
DIRECTINPUT OF P47 TO AUXILIARY BUFFER
CRYSTAL OSCILLATOR BUFFER
3-STATE INPUT
3-STATE CONTROL
3-STATE BUFFER
ALTERNATE BUFFER
GG
GH
HG
HH
P48
P47
BCL KIN
OSC
P46
DPGM
P40
P41
P42
P43
RST
X1245

Fig. 1.5. Example of a medium-grain device (©1993 Xilinx).

not "downloaded" to the device, it is better protected against unauthorized use. A recent innovation, based on an EPROM technology, is called "flash" memory. These devices are usually viewed as "pagewise" in-system reprogrammable systems with physically smaller cells, equivalent to an E^{2}PROM device. Finally, the important advantages and disadvantages of different device technologies are summarized in Table 1.2.

1.2.3 Benchmark for FPLs

Providing objective benchmarks for FPL devices is a nontrivial task. Performance is often predicated on the experience and skills of the designer, along with design tool features. To establish valid benchmarks, the Programmable Electronic Performance Cooperative (PREP) was founded by Xilinx [12], Altera [13], and Actel [14], and has since expanded to more than 10 members. PREP has developed nine different benchmarks for FPLs that are summarized in Table 1.3. The central idea underlining the benchmarks is that each vendor uses its own devices and software tools to implement the basic blocks as many times as possible in the specified device, while attempting to maximize speed. The number of instantiations of the same logic block within



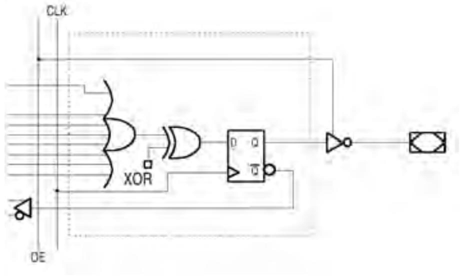


Fig. 1.6. The GAL16V8. **(a)** First three of eight macrocells. **(b)** The output logic macrocell (OLMC) (©1997 Lattice).

one device is called the *repetition rate* and is the basis for all benchmarks. For DSP comparisons, benchmarks five and six of Table 1.3 are relevant. In Fig. 1.7, repetition rates are reported over frequency, for typical Actel (A_k), Altera (o_k), and Xilinx (x_k) devices. It can be concluded that modern FPGA families provide the best DSP complexity and maximum speed. This is attributed to the fact that modern devices provide fast-carry logic (see Sect. 1.4.1, p. 18) with delays (less than 0.1 ns per bit) that allow fast adders with large bit width, without the need for expensive "carry look-ahead" decoders. Although PREP benchmarks are useful to compare equivalent gate counts and maximum speeds, for concrete applications additional attributes are also important. They include:

- Array multiplier (e.g., 18 × 18 bits)
- Embedded hardwired microprocessor (e.g., 32-bit RISC PowerPC)
- On-chip RAM or ROM (LE or large block size)
- External memory support for ZBT, DDR, QDR, SDRAM

Table 1.2. FPL technology.

Technology	SRAM	EPROM	E^2PROM	Antifuse	Flash
Repro-grammable	✓	✓	✓	–	✓
In-system programmable	✓	–	✓	–	✓
Volatile	✓	–	–	–	–
Copy protected	–	✓	✓	✓	✓
Examples	Xilinx Spartan	Altera MAX5K	AMD MACH	Actel ACT	Xilinx XC9500
	Altera Cyclone	Xilinx XC7K	Altera MAX 7K		Cypress Ultra 37K

- Pin-to-pin delay
- Internal tristate bus
- Readback- or boundary-scan decoder
- Programmable slew rate or voltage of I/O
- Power dissipation
- Ultra-high speed serial interfaces

Some of these features are (depending on the specific application) more relevant to DSP application than others. We summarize the availability of some of these key features in Tables 1.4 and 1.5 for Xilinx and Altera, respectively. The first column shows the device family name. The columns $3-9$ show the (for most DSP applications) relevant features: (3) the support of fast-carry logic for adder or subtractor, (4) the embedded array multiplier of 18×18 bit width, (5) the on-chip RAM implemented with the LEs, (6) the on-chip kbit memory block of size larger of about 1-16 kbit,(7) the on-chip Mbit memory block of size larger of about 1 mega bit, (8) embedded microprocessor: IBM's PowerPC on Xilinx or the ARM processor available with Altera devices, and (9) the target price and availability of the device family. Device that are no longer recommended for new designs are classified as mature with `m`. Low-cost devices have a single \$ and high price range devices have two \$\$.

Figure 1.8 summarizes the power dissipation of some typical FPL devices. It can be seen that CPLDs usually have higher "standby" power consumption. For higher-frequency applications, FPGAs can be expected to have a higher power dissipation. A detailed power analysis example can be found in Sect. 1.4.2, p. 27.

Table 1.3. The PREP benchmarks for FPLs.

Number	Benchmark name	Description
1	Data path	Eight 4-to-1 multiplexers drive a parallel-load 8-bit shift register (see Fig. 1.27, p. 44)
2	Timer/counter	Two 8-bit values are clocked through 8-bit value registers and compared (see Fig. 1.28, p. 45)
3	Small state machine	An 8-state machine with 8 inputs and 8 outputs (see Fig. 2.59, p. 160)
4	Large state machine	A 16-state machine with 40 transitions, 8 inputs, and 8 outputs (see Fig. 2.60, p. 161)
5	Arithmetic circuit	A 4-by-4 unsigned multiplier and 8-bit accumulator (see Fig. 4.23, p. 243)
6	16-bit accumulator	A 16-bit accumulator (see Fig. 4.24, p. 244)
7	16-bit counter	Loadable binary up counter (see Fig. 9.40, p. 642)
8	16-bit synchronous prescaled counter	Loadable binary counter with asynchronous reset (see Fig. 9.40, p. 642)
9	Memory mapper	The map decodes a 16-bit address space into 8 ranges (see Fig. 9.41, p. 643)

1.3 DSP Technology Requirements

The PLD market share, by vendor, is presented in Fig. 1.9. PLDs, since their introduction in the early 1980s, have enjoyed in the last decade steady growth of 20% per annum, outperforming ASIC growth by more than 10%. In 2001 the worldwide recession in microelectronics reduced the ASIC and FPLD growth essentially. Since 2003 we see again a steep increase in revenue for the two market leader. The reason that FPLDs outperformed ASICs seems to be related to the fact that FPLs can offer many of the advantages of ASICs such as:

- Reduction in size, weight, and power dissipation
- Higher throughput
- Better security against unauthorized copies
- Reduced device and inventory cost
- Reduced board test costs

without many of the disadvantages of ASICs such as:

Table 1.4. Xilinx FPGA family DSP features.

Family	Feature						
	Fast adder carry logic	Emb. mult. 18×18 bits	LE RAM	Kbit RAM	Mbit RAM	Emb. µP	Low cost/ mature
XC2000	–	–	–	–	–	–	m
XC3000	–	–	–	–	–	–	m
XC4000	✓	–	✓	–	–	–	m
Spartan-XL	✓	–	✓	–	–	–	$
Spartan-II	✓	–	✓	✓	–	–	$
Spartan-3	✓	✓	✓	✓	–	–	$
Virtex	✓	–	✓	✓	–	–	$$
Virtex-II	✓	✓	✓	✓	–	–	$$
Virtex-II Pro	✓	✓	✓	✓	–	✓	$$
Virtex-4-LX	✓	✓	✓	✓	–	–	$$
Virtex-4-SX	✓	✓	✓	✓	–	–	$$
Virtex-4-FX	✓	✓	✓	✓	–	✓	$$
Virtex-5	✓	✓	✓	✓	–	–	$$

Table 1.5. Altera FPGA family DSP features.

Family	Feature						
	Fast adder carry logic	Emb. mult. 18×18 bits	LE RAM	Kbit RAM	Mbit RAM	Emb. µP	Low cost/ mature
FLEX8K	✓	–	–	–	–	–	m
FLEX10K	✓	–	–	✓	–	–	m
APEX20K	✓	–	–	✓	–	–	m
APEX II	✓	–	–	✓	–	–	m
ACEX	✓	–	–	✓	–	–	m
Mercury	✓	–	–	✓	–	–	m
Excalibur	✓	–	–	✓	–	✓	m
Cyclone	✓	–	–	✓	–	–	$
Cyclone II	✓	✓	–	✓	–	–	$
Stratix	✓	✓	–	✓	✓	–	$$
Stratix II	✓	✓	–	✓	✓	–	$$

- A reduction in development time (rapid prototyping) by a factor of three to four
- In-circuit reprogrammability
- Lower NRE costs resulting in more economical designs for solutions requiring less than 1000 units

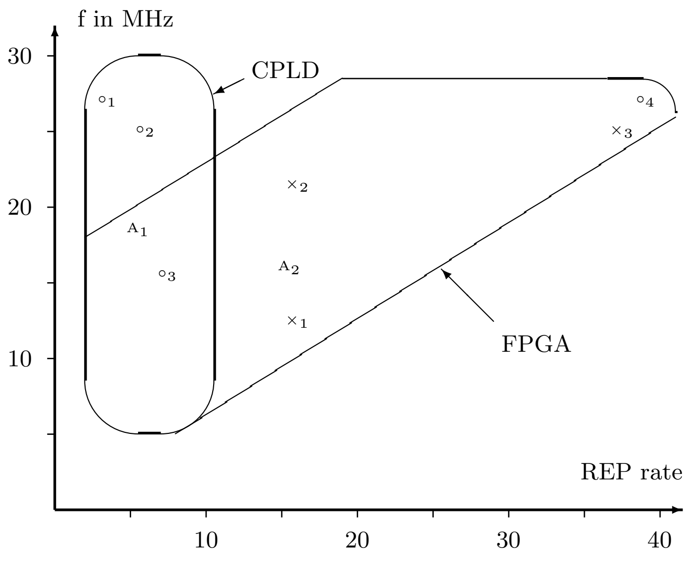


Fig. 1.7. Benchmarks for FPLs (©1995 VDI Press [4]).

CBIC ASICs are used in high-end, high-volume applications (more than 1000 copies). Compared to FPLs, CBIC ASICs typically have about ten times more gates for the same die size. An attempt to solve the latter problem is the so-called hard-wired FPGA, where a gate array is used to implement a verified FPGA design.

1.3.1 FPGA and Programmable Signal Processors

General-purpose programmable digital signal processors (PDSPs) [6, 15, 16] have enjoyed tremendous success for the last two decades. They are based on a reduced instruction set computer (RISC) paradigm with an architecture consisting of at least one fast array multiplier (e.g., 16×16-bit to 24×24-bit fixed-point, or 32-bit floating-point), with an extended wordwidth accumulator. The PDSP advantage comes from the fact that most signal processing algorithms are multiply and accumulate (MAC) intensive. By using a multistage pipeline architecture, PDSPs can achieve MAC rates limited only by the speed of the array multiplier. More details on PDSPs can be found in Chap. 9. It can be argued that an FPGA can also be used to implement MAC cells [17], but cost issues will most often give PDSPs an advantage, if the PDSP meets the desired MAC rate. On the other hand we now find many high-bandwidth signal-processing applications such as wireless, multimedia, or satellite transmission, and FPGA technology can provide more bandwidth through multiple MAC cells on one chip. In addition, there are several al-

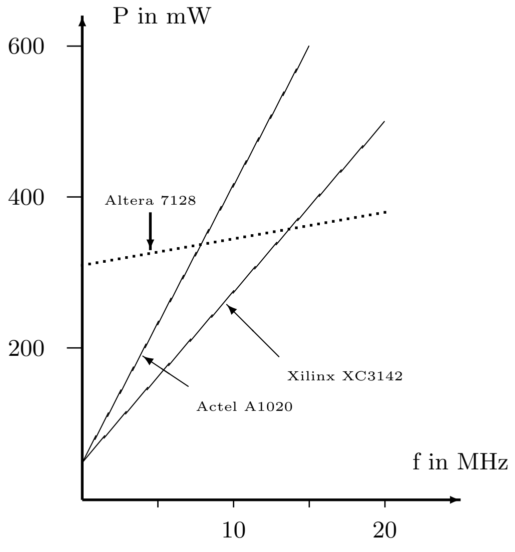


Fig. 1.8. Power dissipation for FPLs (©1995 VDI Press [4]).

gorithms such as CORDIC, NTT or error-correction algorithms, which will be discussed later, where FPL technology has been proven to be more efficient than a PDSP. It is assumed [18] that in the future PDSPs will dominate applications that require complicated algorithms (e.g., several `if-then-else` constructs), while FPGAs will dominate more front-end (sensor) applications like FIR filters, CORDIC algorithms, or FFTs, which will be the focus of this book.

1.4 Design Implementation

The levels of detail commonly used in VLSI designs range from a geometrical layout of full custom ASICs to system design using so-called set-top boxes. Table 1.6 gives a survey. Layout and circuit-level activities are absent from FPGA design efforts because their physical structure is programmable but fixed. The best utilization of a device is typically achieved at the gate level using register transfer design languages. Time-to-market requirements, combined with the rapidly increasing complexity of FPGAs, are forcing a methodology shift towards the use of intellectual property (IP) macrocells or mega-core cells. Macrocells provide the designer with a collection of predefined functions, such as microprocessors or UARTs. The designer, therefore, need only specify selected features and attributes (e.g., accuracy), and a

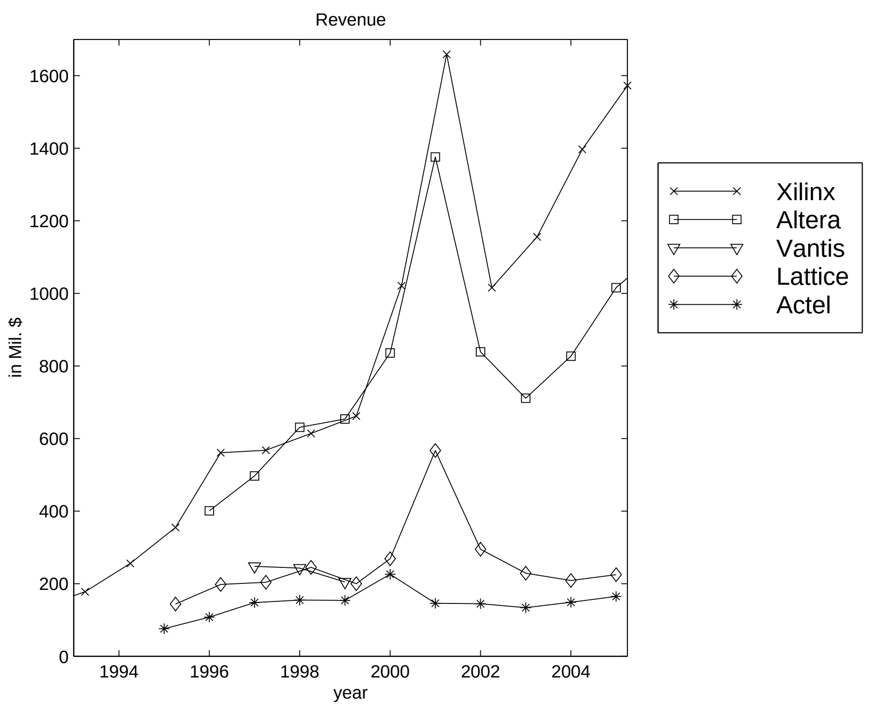


Fig. 1.9. Revenues of the top five vendors in the PLD/FPGA/CPLD market.

Table 1.6. VLSI design levels.

Object	Objectives	Example
System	Performance specifications	Computer, disk unit, radar
Chip	Algorithm	µP, RAM, ROM, UART, parallel port
Register	Data flow	Register, ALU, COUNTER, MUX
Gate	Boolean equations	AND, OR, XOR, FF
Circuit	Differential equations	Transistor, R, L, C
Layout	None	Geometrical shapes

synthesizer will generate a hardware description code or schematic for the resulting solution.

A key point in FPGA technology is, therefore, powerful design tools to

- Shorten the design cycle
- Provide good utilization of the device
- Provide synthesizer options, i.e., choose between optimization speed versus size of the design

A CAE tool taxonomy, as it applies to FPGA design flow, is presented in Fig. 1.10. The design entry can be graphical or text-based. A formal check that eliminates syntax errors or graphic design rule errors (e.g., open-ended wires) should be performed before proceeding to the next step. In the function extraction the basic design information is extracted from the design and written in a functional netlist. The netlist allows a first functional simulation of the circuit and to build an example data set called a testbench for later testing of the design with timing information. If the functional test is not passed we start with the design entry again. If the functional test is satisfactory we proceed with the design implementation, which usually takes several steps and also requires much more compile time then the function extraction. At the end of the design implementation the circuit is completely routed within our FPGA, which provides precise resource data and allows us to perform a simulation with all timing delay information as well as performance measurements. If all these implementation data are as expected we can proceed with the programming of the actual FPGA; if not we have to start with the design entry again and make appropriate changes in our design. Using the JTAG interface of modern FPGAs we can also directly monitor data processing on the FPGA: we may read out just the I/O cells (which is called a boundary scan) or we can read back all internal flip-flops (which is called a full scan). If the in-system debugging fails we need to return to the design entry.

In general, the decision of whether to work within a graphical or a text design environment is a matter of personal taste and prior experience. A graphical presentation of a DSP solution can emphasize the highly regular dataflow associated with many DSP algorithms. The textual environment, however, is often preferred with regard to algorithm control design and allows a wider range of design styles, as demonstrated in the following design example. Specifically, for Altera's Quartus II, it seemed that with text design more special attributes and more-precise behavior can be assigned in the designs.

Example 1.1: Comparison of VHDL Design Styles

The following design example illustrates three design strategies in a VHDL context. Specifically, the techniques explored are:

- Structural style (component instantiation, i.e., graphical netlist design)
- Data flow, i.e., concurrent statements
- Sequential design using `PROCESS` templates

The VHDL design file `example.vhd`[4] follows (comments start with `--`):

```
PACKAGE eight_bit_int IS    -- User-defined type
  SUBTYPE BYTE IS INTEGER RANGE -128 TO 127;
END eight_bit_int;

LIBRARY work;
USE work.eight_bit_int.ALL;

LIBRARY lpm;                    -- Using predefined packages
USE lpm.lpm_components.ALL;
```

[4] The equivalent Verilog code `example.v` for this example can be found in Appendix A on page 663. Synthesis results are shown in Appendix B on page 731.

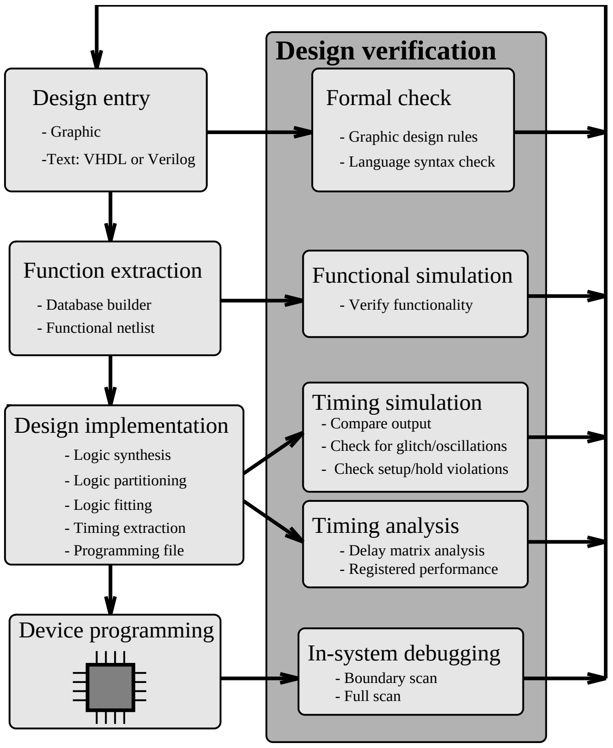


Fig. 1.10. CAD design circle.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY example IS                          ------> Interface
  GENERIC (WIDTH : INTEGER := 8);   -- Bit width
  PORT (clk  :  IN STD_LOGIC;
        a, b :  IN BYTE;
        op1  :  IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        sum  :  OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        d    :  OUT BYTE);
END example;

ARCHITECTURE fpga OF example IS

  SIGNAL  c, s        :  BYTE;       -- Auxiliary variables
  SIGNAL  op2, op3    :  STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);

BEGIN

  -- Conversion int -> logic vector
  op2 <= CONV_STD_LOGIC_VECTOR(b,8);

  add1: lpm_add_sub          ------> Component instantiation
    GENERIC MAP (LPM_WIDTH => WIDTH,
                 LPM_REPRESENTATION => "SIGNED",
                 LPM_DIRECTION => "ADD")
    PORT MAP (dataa => op1,
              datab => op2,
              result => op3);
  reg1: lpm_ff
    GENERIC MAP (LPM_WIDTH => WIDTH )
    PORT MAP (data => op3,
              q => sum,
              clock => clk);

  c <= a  + b ;                    ------> Data flow style

  p1: PROCESS                      ------> Behavioral style
  BEGIN
    WAIT UNTIL clk = '1';
    s <= c + s;              ----> Signal assignment statement
  END PROCESS p1;
  d <= s;

END fpga;
```

$\boxed{1.1}$

To start the simulator[5] we first copy the file from the CD to the project directory and use File→Open Project to select the example project. Now select Simulator Tool under the Processing menu. A new window to control the simulation parameter will pop up. To perform a function simulation in Quartus II the Generate Functional Simulation Netlist button needs to be activated first by selecting Functional as Simulation mode. If successful we can proceed and start with the design implementation as shown in Fig. 1.10. To do this with the Quartus II compiler, we choose Timing as the Simulation mode. However, the timing simulation requires that all compilation steps (Analysis & Synthesis, Fitter, Assembler and Timing Analyzer) are first performed. After completion of the compilation we can then conduct a simulation with timing, check for glitches, or measure the Registered Performance of the design, to name just a few options. After all these steps are successfully completed, and if a hardware board (like the prototype board shown in Fig. 1.11) is available, we proceed with programming the device and may perform additional hardware tests using the read-back methods, as reported in Fig. 1.10. Altera supports several DSP development boards with a large set of useful prototype components including fast A/D, D/A, audio CODEC, DIP switches, single and 7-segment LEDs, and push

[5] Note that a more detailed design tool study will follow in section 1.4.3.

buttons. These development boards are available from Altera directly. Altera offers Stratix S25, Stratix II S60,and S80 and Cyclone II boards, in the \$995-\$5995 price range, which differs not only in FPGA size, but also in terms of the extra features, like number, precision and speed of A/D channels, and memory blocks. For universities a good choice will be the lowest-cost Cyclone II board, which is still more expensive than the UP2 or UP3 boards used in many digital logic labs, but has a fast A/D and D/A and a two-channel CODEC, and large memory bank outside the FPGA, see Fig. 1.11a. Xilinx on the other side has very limited direct board support; all boards for instance available in the university program are from third parties. However some of these boards are priced so low that it seems that these boards are not-for-profit designs. A good board for DSP purposes (with on-chip multipliers) is for instance offered by Digilent Inc. for only \$99, see Fig. 1.11b. The board has a XC3S200 FPGA, flash, four 7-segment LEDs, eight switches, and four push buttons. For DSP experiments, A/D and D/A mounted on very small daughter boards are available for \$19.95 each, so a nice DSP board can be built for only \$138.90.

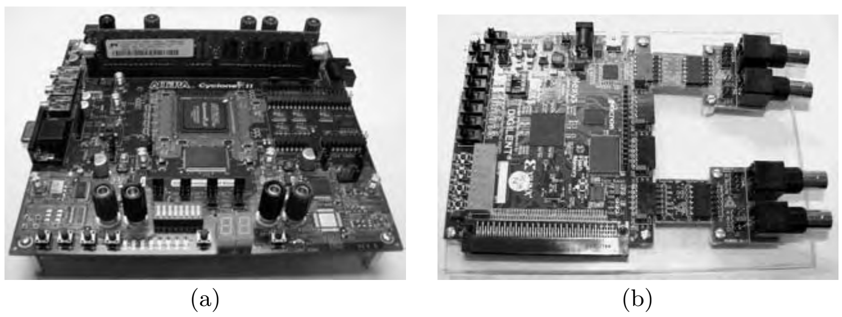


Fig. 1.11. Low-cost prototype boards: **(a)** Cyclone II Altera board. **(b)** Xilinx Nexsys board with ADC and DAC daughter boards.

1.4.1 FPGA Structure

At the beginning of the 21st century FPGA device families now have several attractive features for implementing DSP algorithms. These devices provide fast-carry logic, which allows implementations of 32-bit (nonpipelined) adders at speeds exceeding 300 MHz [1, 19, 20], embedded 18×18 bit multipliers, and large memory blocks.

Xilinx FPGAs are based on the elementary logic block of the early XC4000 family and the newest derivatives are called Spartan (low cost) and Virtex (high performance). Altera devices are based on FLEX 10K logic blocks and the newest derivatives are called Stratix (high performance) and Cyclone (low cost). The Xilinx devices have the wide range of routing levels typical of a FPGAs, while the Altera devices are based on an architecture with the wide busses used in Altera's CPLDs. However, the basic blocks of the Cyclone and Stratix devices are no longer large PLAs as in CPLD. Instead the devices now have medium granularity, i.e., small look-up tables (LUTs), as is typical for FPGAs. Several of these LUTs, called logic elements (LE) by Altera, are grouped together in a logic array block (LAB). The number of LEs in an LAB depends on the device family, where newer families in general have more LEs per LAB: Flex10K utilizes eight LEs per LAB, APEX20K uses 10 LEs per LAB and Cyclone II has 16 LEs per LAB.

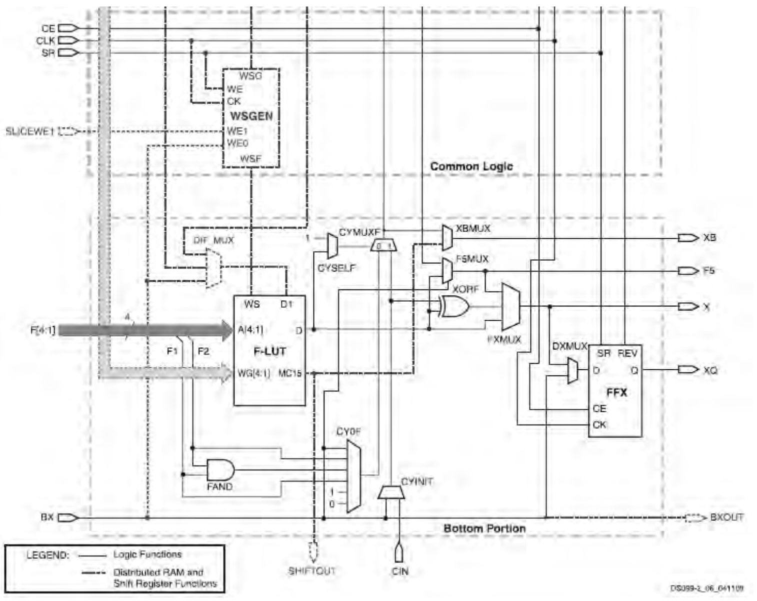


Fig. 1.12. Spartan-3 low portion of a slice/logic element (©2006 Xilinx).

Since the Spartan-3 devices are part of a popular DSP board offered by Digilent Inc., see Figure 1.11b, we will have a closer look at this FPGA family. The basic logic elements of the Xilinx Spartan-3 are called slices having two separate four-input one-output LUTs, fast-carry dedicated logic, two flip-flops, and some shared control signals. In the Spartan-3 family four slices are combined in a configurable logic blocks (CLB), having a total of eight four-input one-output LUTs, and eight flip-flops. Figure 1.12 shows the lower part of the left slice. Each slice LUT can be used as a 16×1 RAM or ROM. The dashed part is used if the slice is used to implement distributed memory or shift registers, and is only available in 50% of the slices. The Xilinx device has multiple levels of routing, ranging from CLB to CLB, to long lines spanning the entire chip. The Spartan-3 device also includes large memory block

Table 1.7. The Xilinx Spartan-3 family.

Device	Total 4-input LUTs	CLB	RAM blocks	DCM	Emb. mult. 18×18	Max. I/O mbit	Conf. file
XC3S50	1536	192	4	2	4	124	0.4
XC3S200	3840	480	12	4	12	173	1.0
XC3S400	7168	896	16	4	16	264	1.7
XC3S1000	15 360	1920	24	4	24	391	3.2
XC3S1500	26 624	3328	32	4	32	487	5.2
XC3S2000	40 960	5120	40	4	40	565	7.6
XC3S4000	55 296	6912	96	4	96	712	11.3
XC3S5000	66 560	8320	104	4	104	784	13.2

(18,432 bits or 16,384 bits if no parity bits are used) that can be used as single- or dual-port RAM or ROM. The memory blocks can be configure as $2^9 \times 32, 2^{10} \times 16, \ldots, 2^{14} \times 1$, i.e., each additional address bit reduces the data bit width by a factor of two. Another interesting feature for DSP purpose is the embedded multiplier in the Spartan-3 family. These are fast 18×18 bit signed array multipliers. If unsigned multiplication is required 17×17 bit multiplier can be implemented with this embedded multiplier. This device family also includes up to four complete clock networks (DCMs) that allow one to implement several designs that run at different clock frequencies in the same FPGA with low clock skew. Up to 13 Mbits configuration files size is required to program Spartan-3 devices. Tables 1.7 shows the most important DSP features of members of the Xilinx Spartan-3 family.

As an example of an Altera FPGA family let us have a look at the Cyclone II devices used in the low-cost prototyping board by Altera, see Fig. 1.11a. The basic block of the Altera Cyclone II device achieves a medium granularity using small LUTs. The Cyclone device is similar to the Altera 10K device used in the popular UP2 and UP3 boards, with increased RAM blocks memory size to 4 kbits, which are no longer called EAB as in Flex 10K or ESB as in the APEX family, bur rather M4K memory blocks, which better reflects their size. The basic logic element in Altera FPGAs is called a logic element (LE)[6] and consists of a flip-flop, a four-input one-output or three-input one-output LUT and a fast-carry logic, or AND/OR product term expanders, as shown in Fig. 1.13. Each LE can be used as a four-input LUT in the normal mode, or in the arithmetic mode, as a three-input LUT with an additional fast carry. Sixteen LEs are combined in a logic array block (LAB) in Cyclone II devices. Each row contains at least one embedded 18×18 bit multiplier and one M4K memory block. One 18×18 bit multiplier can also be used as two signed 9×9 bit multipliers, or one unsigned 17×17 bit multiplier. The M4K memory can be configured as $2^7 \times 32$, $2^8 \times 16, \ldots, 4096 \times 1$ RAM or ROM. In addition one

[6] Sometimes also called logic cells (LCs) in a design report file.

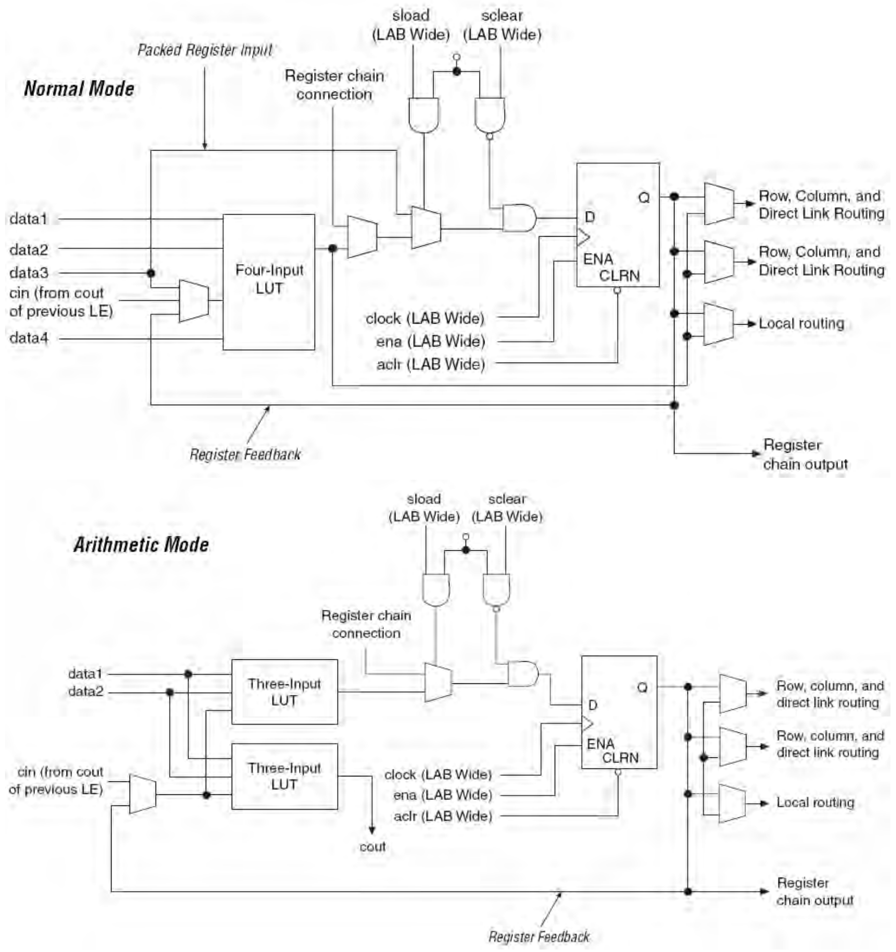


Fig. 1.13. Cyclone II logic cell (©2005 Altera).

parity bit per byte is available (e.g., 128 × 36 configuration), which can be used for data integrity. These M4Ks and LABs are connected through wide high-speed busses as shown in Fig. 1.14. Several PLLs are in use to produce multiple clock domains with low clock skew in the same device. At least 1 Mbits configuration files size is required to program the devices. Table 1.8 shows some members of the Altera Cyclone II family.

If we compare the two routing strategies from Altera and Xilinx we find that both approaches have value: the Xilinx approach with more local and less global routing resources is synergistic to DSP use because most digital signal processing algorithms process the data locally. The Altera approach, with wide busses, also has value, because typically not only are single bits

Table 1.8. Altera's Cyclone II device family.

Device	Total 4-input LUTs	RAM blocks M4K	PLLs/ clock networks	Emb. mul. 18×18	Max. I/O	Conf. file Mbits
EP2C5	4608	26	2/8	13	89	1.26
EP2C8	8256	36	2/8	18	85	1.98
EP2C20	18 752	52	4/16	26	315	3.89
EP2C35	33 216	105	4/16	35	475	6.85
EP2C50	50 528	129	4/16	86	450	9.96
EP2C70	68 416	250	4/16	150	622	14.31

processed in bit slice operations, but normally wide data vectors with 16 to 32 bits must be moved to the next DSP block.

1.4.2 The Altera EP2C35F672C6

The Altera EP2C35F672C6 device, a member of the Cyclone II family, which is part of the DSP prototype board provided through Altera's university program, is used throughout this book. The device nomenclature is interpreted as follows:

```
EP2C35F672C6
 |  |   |  |--> speed grade
 |  |   |-----> Package and pin number
 |  |---------> LEs in 1000
 |------------> Device family
```

Specific design examples will, wherever possible, target the Cyclone II device EP2C35F672C6 using Altera-supplied software. The enclosed Quartus II software is a fully integrated system with VHDL and Verilog editor, synthesizer, simulator, and bitstream generator. The only limitation in the web version is that not all pinouts of every devices are available. Because all examples are available in VHDL and Verilog, any other simulator may also be used. For instance, the device-independent ModelTech compiler has successfully been used to compile the examples using the synthesizable code for `lpm` functions on the CD-ROM provided by EDIF. The use of Xilinx ISE software is also discussed in appendix D.

Logic Resources

The EP2C35 is a member of the Altera Cyclone II family and has a logic density equivalent to about 35 000 logic elements (LEs). An additional 35 multipliers of size 18 × 18 bits (or twice this number if a size of 9 × 9 bit is used) are available. From Table 1.8 it can be seen that the EP2C35 device

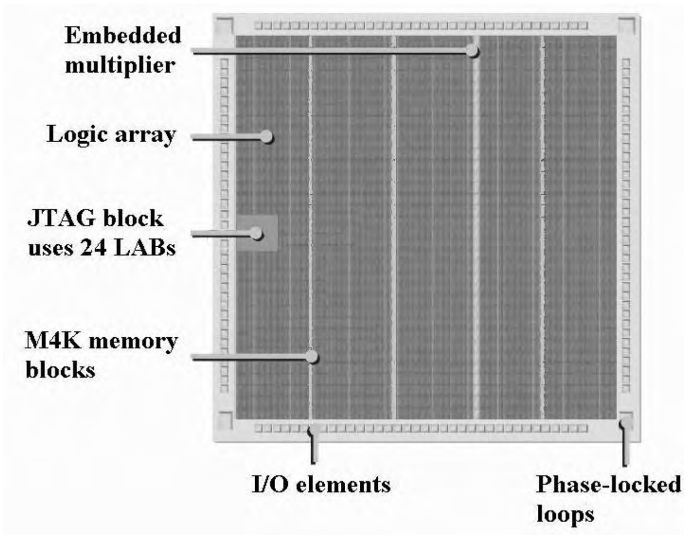


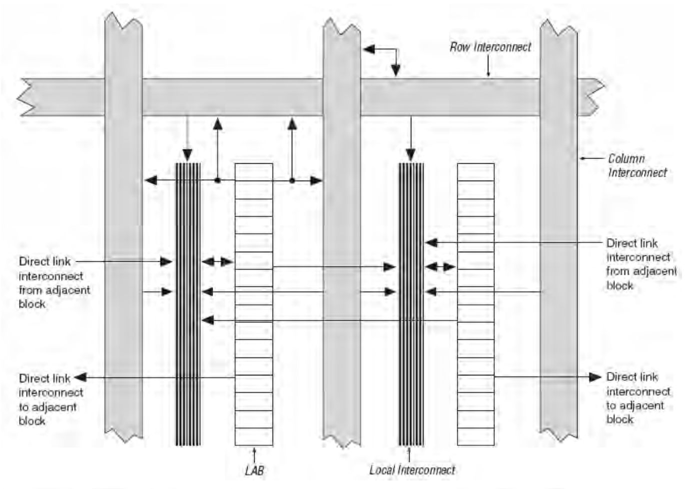
Fig. 1.14. Overall floorplan in Cyclone II devices.

has 33 216 basic logic elements (LEs). This is also the maximum number of implementable full adders. Each LE can be used as a four-input LUT, or in the arithmetic mode, as a three-input LUT with an additional fast carry as shown in Fig. 1.13. Sixteen LEs are always combined into a logic array block (LAB), see Fig. 1.15a. The number of LABs is therefore 33,216/16=2076. These 2076 LABs are arranged in 35 rows and 60 columns. In the left medium area of the device the JTAG interface is placed and uses the area of 24 LABs. This is why the total number of LABs in not just the product of rows $\times$ column, i.e., $35 \times 60 - 24 = 2100 - 24 = 2076$. The device also includes three columns of 4-kbit memory block (called M4K memory blocks, see Fig. 1.15b) that have the height of one LAB and the total number of M4Ks is therefore $3 \times 35 = 105$. The M4Ks can be configured as 128×36, 128×32, 256×18, 256×16, $\ldots 4096 \times 1$ RAM or ROM, where for each byte one parity bit is available. The EP2C35 also has one column of 18×18 bit fast array multipliers, that can also be configured as two 9×9 bit multipliers, see Fig. 1.16. Since there are 35 rows the number of multipliers is 35 for the 18×18 bit type or 70 of the 9×9 bit multiplier type. Figure 1.14 presents the overall device floorplan.

Routing Resources

All 16 LEs in a LAB share the same reset and clock signals. Each LE has a fan-out of 48 for fast local connection to neighboring LABs. The next level of routing are the R4 and C4 fast row and column local connections that allow wires to reach LABs at a distance of ± 4 LABs, or 3 LABs and one embedded multiplier or M4K memory block. The longest connection available

(a)


Row Interconnect
Column Interconnect
Direct link interconnect from adjacent block
Direct link interconnect from adjacent block
Direct link interconnect to adjacent block
Direct link interconnect to adjacent block
LAB
Local Interconnect

(b)

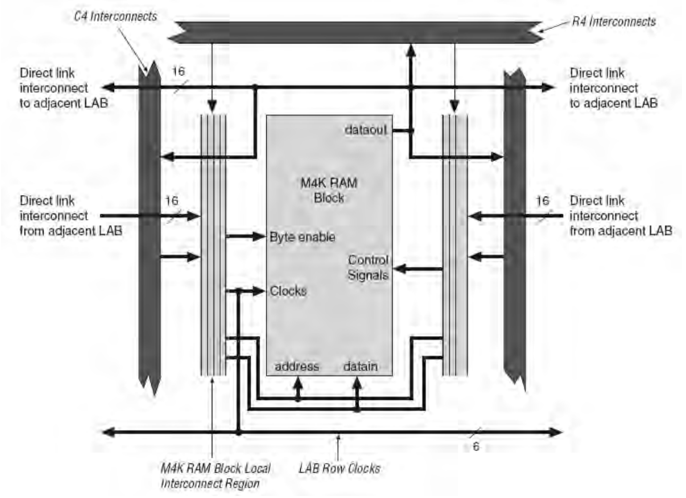

C4 Interconnects
R4 Interconnects
Direct link interconnect to adjacent LAB
Direct link interconnect to adjacent LAB
16
dataout
M4K RAM Block
Direct link interconnect from adjacent LAB
16
16
Direct link interconnect from adjacent LAB
Byte enable
Control Signals
Clocks
address
datain
6
M4K RAM Block Local Interconnect Region
LAB Row Clocks

Fig. 1.15. Cyclone II resources: (a) logic array block structure (b) M4K memory block interface (© 2005 Altera [21]).

are R24 and C16 wires that allows 24 rows or 16 column LAB, respectively, to build connections that span basically the entire chip. It is also possible to use any combination of row and column connections, in case the source and destination LAB are not only in different rows but also in different columns. As we will see in the next section the delay in these connections varies widely and the synthesis tool tries always to place logic as close together as possible to minimize the interconnection delay. A 32 bit adder, for instance, would be best placed in two LABs in two rows one above the other, see Fig. 1.20, p. 33.

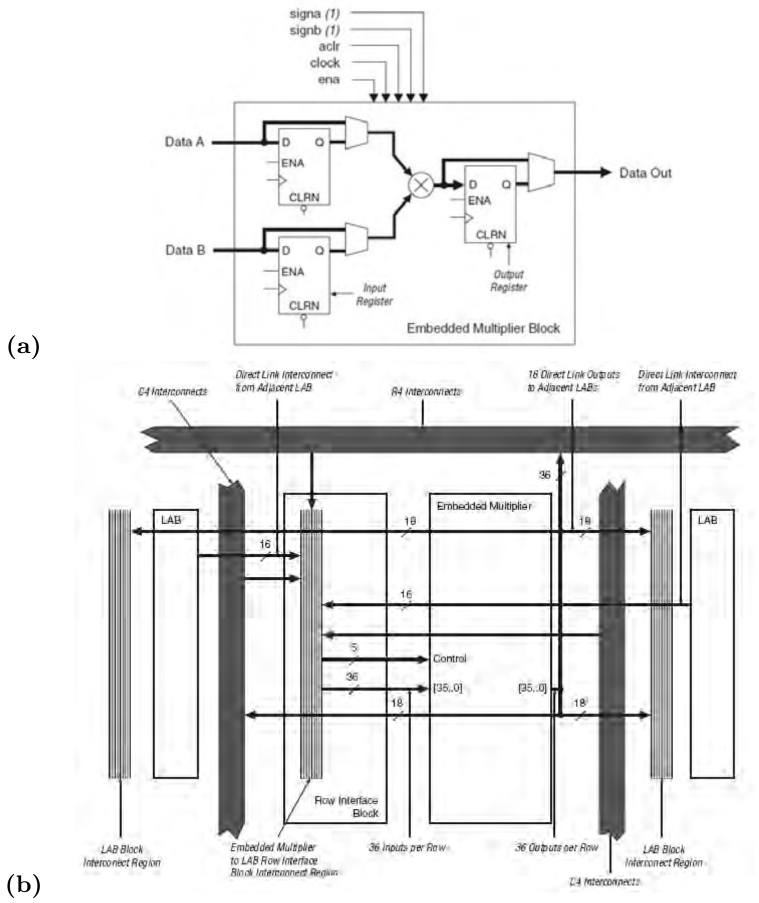

(a)
signa (1)
signb (1)
aclr
clock
ena
Data A
Data B
D
Q
ENA
CLRN
Input Register
Output Register
Data Out
Embedded Multiplier Block
(b)
C4 Interconnects
Direct Link Interconnect from Adjacent LAB
R4 Interconnects
18 Direct Link Outputs to Adjacent LABs
Direct Link Interconnect from Adjacent LAB
LAB
Embedded Multiplier
Control
[35..0]
Row Interface Block
LAB Block Interconnect Region
Embedded Multiplier to LAB Row Interface Block Interconnect Region
36 Inputs per Row
36 Outputs per Row

Fig. 1.16. Embedded multiplier (a) Architecture (b) LAB interface (© 2005 Altera [21]).

Timing Estimates

Altera's Quartus II software calculates various timing data, such as the `Registered Performance`, setup/hold time ($t_{\mathrm{su}}, t_{\mathrm{h}}$) and non-registered combination delay (t_{pd}). For a full description of all timing parameters, refer to the `Timing Analysis Settings` under `EDA Tools Settings` in the `Assignments` menu. To achieve optimal performance, it is necessary to understand how the software physically implements the design. It is useful, therefore, to produce a rough estimate of the solution and then determine how the design may be improved.

Example 1.2: Speed of an 32-bit Adder

Assume one is required to implement a 32-bit adder and estimate the design's maximum speed. The adder can be implemented in two LABs, each using the fast-carry chain. A rough first estimate can be done using the carry-in to carry-out delay, which is 71 ps for Cyclone II speed grade 6. An upper bound for the maximum performance would then be $32 \times 71 \times 10^{-12} = 2.272$ ns or 440 MHz. But in the actual implementation additional delays occur: first the interconnect delay from the previous register to the first full adder gives an additional delay of 0.511 ns. Next the first carry t_{cgen} must be generated, requiring about 0.414 ns. With the group of eight LEs each and in between the LAB we see from the floorplan that stronger drivers are used, requiring an additional 75-88 ps. Finally at the end of the carry chain the full sum bit needs to be computed (about 410 ps) and the setup time for the output register (84 ps) needs to be taken into account. The results are then stored in the LE register. The following table summarizes these timing data:

LE register clock-to-output delay	t_{co}	=	223 ps
Interconnect delay	t_{ic}	=	511 ps
Data-in to carry-out delay	t_{cgen}	=	414 ps
Carry-in to carry-out delay	$27 \times t_{\text{cico}}$	$=27 \times 71\,ps$ =	1917 ps
8 bit LAB group carry-out delay	$2 \times t_{\text{cico8LAB}}$	$=2\times$ 159 ps =	318 ps
Same column carry out delay	$t_{\text{samecolumn}}$	=	146 ps
LE look-up table delay	t_{LUT}	=	410 ps
LE register setup time	t_{su}	=	84 ps
Total		=	4,022 ps

The estimated delay is 4.02 ns, or a rate of 248.63 MHz. The design is expected to use about 32 LEs for the adder and an additional 2×32 to store the input data in the registers (see also Exercise 1.7, p. 43). $\boxed{1.2}$

If the two LABs used can not be placed in the same column next to each other then an additional delay would occur. If the signal comes directly from the I/O pins much longer delays have to be expected. For a 32 bit adder with data coming from the I/O pins the Quartus II `Timing Analyzer Tool` reports a propagation delay of 8.944 ns, much larger than the registered performance when the data comes directly from the register next to the design under test. Datasheets [21, Chap. 5] usually report the best performance that is achieved if I/O data of the design are placed in registers close to the design unit under test. Multiplier and block M4K (but not the adder) have additional I/O registers to enable maximum speed, see Fig. 1.16. The additional I/O registers are usually not counted in the LE resource estimates, since it is assumed that the previous processing unit uses a output register for all data. This may not always be the case and we have therefore put the additional register needed for the adder design in parentheses. Table 1.9 reports some typical data measured under these assumptions. If we compare this measured data with the delay given in the data book [21, Chap. 5] we notice that for some blocks Quartus II limits the upper frequency to a specific bound less than the delay in the data book. This is a conservative and more-secure estimate – the design may in fact run error free at a slightly higher speed.

Table 1.9. Some typical `Registered Performance` and resource data for the Cyclone II EP2C35F672C6.

Design	LE	M4K memory blocks	Multiplier blocks 9×9 bit	`Registered Performance` MHz
16 bit adder	16(+32)	–	–	369
32 bit adder	32(+64)	–	–	248
64 bit adder	64(+128)	–	–	151
ROM $2^9 \times 8$	–	1	–	260
RAM $2^9 \times 8$	–	1	–	230
9×9 bit multiplier	–	–	1	260
18×18 bit multiplier	–	–	2	260

Power Dissipation

The power consumption of an FPGA can be a critical design constraint, especially for mobile applications. Using 3.3 V or even lower-voltage process technology devices is recommended in this case. The Cyclone II family for instance is produced in a 1.2 V, 90-nm, low-k-dielectric process from the Taiwan ASIC foundry TSMC, but I/O interface voltages of 3.3 V, 2.5V, 1.8V and 1.5V are also supported. To estimate the power dissipation of the Altera device EP2C35, two main sources must be considered, namely:

1) Static power dissipation, $I_{\text{standby}} \approx 66\,\text{mA}$ for the `EP2C35F672C6`
2) Dynamic (logic, multiplier, RAM, PLL, clocks, I/O) power dissipation, I_{active}

The first parameter is not design dependent, and also the standby power in CMOS technology is generally small. The active current depends mainly on the clock frequency and the number of LEs or other resources in use. Altera provides an EXCEL work sheet, called `PowerPlay Early Power Estimator`, to get an idea about the power consumption (e.g., battery life) and possible cooling requirements in an early project phase.

For LE the dynamic power dissipation is estimated according to the proportional relation

$$P \approx I_{\text{dynamic}} V_{cc} = K \times f_{\max} \times N \times \tau_{\text{LE}} V_{cc}, \tag{1.1}$$

where K is a constant, $f_{\max}$ is the operating frequency in MHz, N is the total number of logic cells used in the device, and τ_{LE} is the average percentage of logic cells toggling at each clock (typically 12.5%). Table 1.10 shows the results for power estimation when all resource of the EP2C35F672C6 are in use and a system clock of 100 MHz is applied. For less resource usage or lower system clock the data in (1.1) can be adjusted. If, for instance, a system clock is reduced from 100 MHz to 10 MHz then the power would be reduced to

Table 1.10. Power consumption estimation for the Cyclone II EP2C35F672C6.

Parameter	Units	Toggle rate (%)	Power mW
P_{static}			85
LEs 33216 @ 100 MHz	33216	12.5%	572
M4K block memory	105	50%	37
18 × 18 bit multiplier	35	12.5%	28
I/O cells (3.3V,24 mA)	475	12.5%	473
PLL	4		30
Clock network	33831		215
Total			1440

$85 + 1355/10 = 220.5\,\text{mW}$, and the static power consumption would now be account for 38%.

Although the `PowerPlay` estimation is a useful tool in a project planing phase, it has its limitations in accuracy because the designer has to specify the toggle rate. There are cases when it become more complicated, such as for instance in frequency synthesis design examples, see Fig. 1.17. While the block RAM estimation with a 50% toggle may be accurate, the toggle rate of the LEs in the accumulator part is more difficult to determine, since the LSBs will toggle at a much higher frequency than the MSBs, since the accumulators produce a triangular output function. A more-accurate power estimation can be made using Altera's `PowerPlay Power Analyzer Tool` available from the `Processing` menu. The `Analyzer` allows us to read in toggle data computed from the simulation output. The simulator produces a "Signal Activity File" that can be selected as the input file for the `Analyzer`. Table 1.11 shows a comparison between the power estimation and the power analysis.

Table 1.11. Power consumption for the design shown in Fig. 1.17 for a Cyclone II EP2C35F672C6.

Parameter	Estimation 12.5% toggle rate power/mW	Analysis toggle rate measured power/mW
Static	79.91	80.02
Dynamic	5.09	6.68
I/O	50.60	83.47
Total	135.60	170.17

We notice a discrepancy of 20% between estimation and analysis. The analysis however requires a complete design including a testbench, while the estimation may be done at an early phase in the project.

The following case study should be used as a detailed scheme for the examples and self-study problems in subsequent chapters.

1.4.3 Case Study: Frequency Synthesizer

The design objective in the following case study is to implement a classical frequency synthesizer based on the Philips PM5190 model (circa 1979, see Fig. 1.17). The synthesizer consists of a 32-bit accumulator, with the eight most significant bits (MSBs) wired to a SINE-ROM lookup table (LUT) to produce the desired output waveform. A *graphical* solution, using Altera's Quartus II software, is shown in Fig. 1.18, and can be found on the CD-ROM as `book3e/vhdl/fun_graf.bdf`. The equivalent HDL text file `fun_text.vhd` and `fun_text.v` implement the design using component instantiation. In the following we walk through all steps that are usually performed when implementing a design using Quartus II:

1) Compilation of the design
2) Design results and floor plan
3) Simulation of the design
4) A performance evaluation

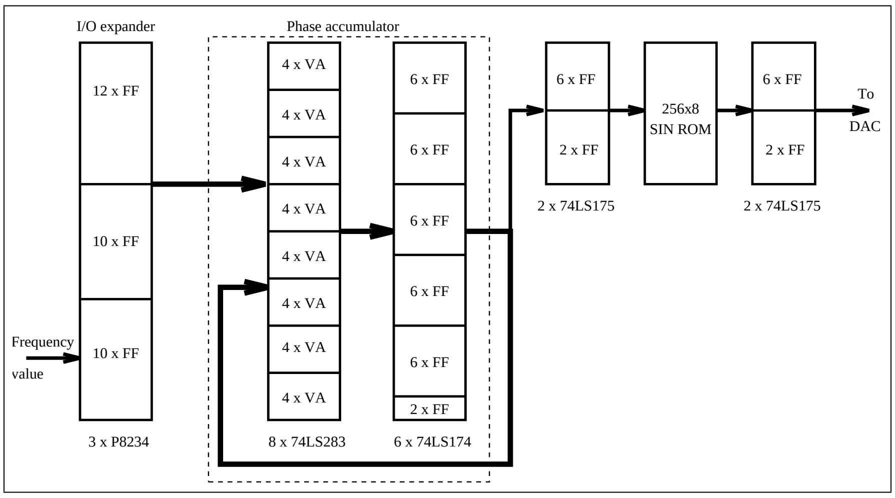


Fig. 1.17. PM5190 frequency synthesizer.

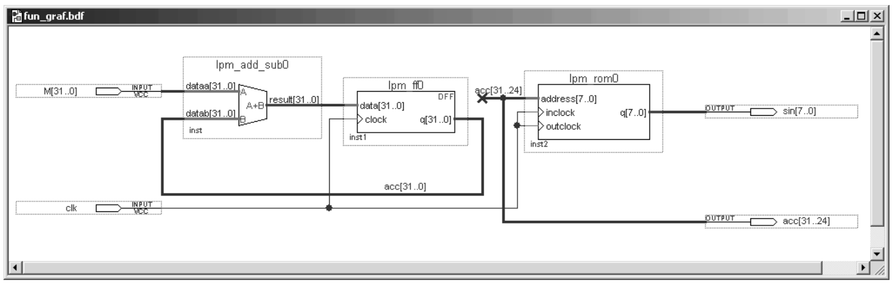


Fig. 1.18. Graphical design of the frequency synthesizer.

Design Compilation

To check and compile the file, start the Quartus II Software and select `File→Open Project` or launch `File→New Project Wizard` if you do not have a project file yet. In the project wizard specify the project directory you would like to use, and the project name and top-level design as `fun_text`. Then press `Next` and specify the HDL file you would like to add, in our case `fun_text.vhd`. Press `Next` again and then select the device `EP2C35F672C6` from the Cyclone II family and press `Finish`. If you use the project file from the CD the file `fun_text.qsf` will already have the correct file and device specification. Now select `File→Open` to load the HDL file. The VHDL design[7] reads as follows:

```
--  A 32-bit function generator using accumulator and ROM

LIBRARY lpm;
USE lpm.lpm_components.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY fun_text IS
  GENERIC ( WIDTH   : INTEGER := 32);    -- Bit width
  PORT ( M        : IN  STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
         sin, acc : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
         clk      : IN  STD_LOGIC);
END fun_text;

ARCHITECTURE fpga OF fun_text IS
```

[7] The equivalent Verilog code `fun_text.v` for this example can be found in Appendix A on page 664. Synthesis results are shown in Appendix B on page 731.

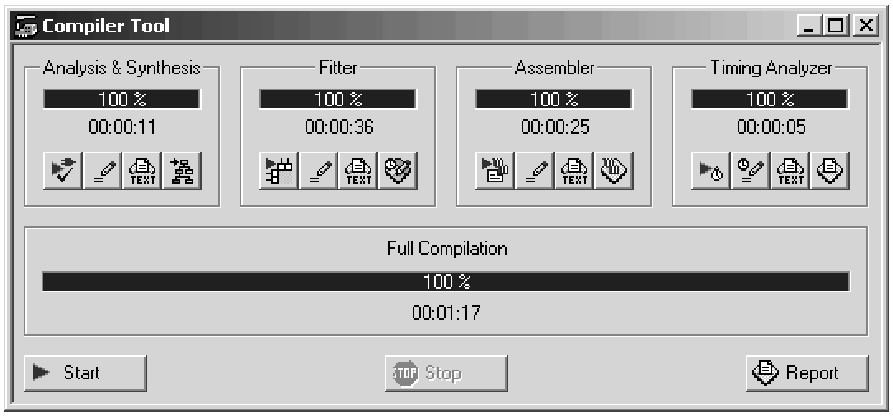
```
    SIGNAL s, acc32 : STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
    SIGNAL msbs     : STD_LOGIC_VECTOR(7 DOWNTO 0);
                                         -- Auxiliary vectors
  BEGIN

    add1: lpm_add_sub             -- Add M to acc32
      GENERIC MAP ( LPM_WIDTH => WIDTH,
                    LPM_REPRESENTATION => "SIGNED",
                    LPM_DIRECTION => "ADD",
                    LPM_PIPELINE => 0)
      PORT MAP ( dataa => M,
                 datab => acc32,
                 result => s );

    reg1: lpm_ff                  -- Save accu
      GENERIC MAP ( LPM_WIDTH => WIDTH)
      PORT MAP ( data => s,
                 q => acc32,
                 clock => clk);

    select1: PROCESS (acc32)
               VARIABLE i : INTEGER;
             BEGIN
               FOR i IN 7 DOWNTO 0 LOOP
                 msbs(i) <= acc32(31-7+i);
               END LOOP;
             END PROCESS select1;

    acc <= msbs;

    rom1: lpm_rom
      GENERIC MAP ( LPM_WIDTH => 8,
                    LPM_WIDTHAD => 8,
                    LPM_FILE => "sine.mif")
      PORT MAP ( address => msbs,
                 inclock => clk,
                 outclock => clk,
                 q => sin);

  END fpga;
```

The object `LIBRARY`, found early in the code, contains predefined modules and definitions. The `ENTITY` block specifies the I/O ports of the device and generic variables. Using component instantiation, three blocks (see labels `add1, reg1, rom1`) are called as subroutines. The `select1 PROCESS` construct is used to select the eight MSBs to address the ROM. Now start the compiler tool (it has a little factory symbol) that can be found under the `Processing` menu. A window similar to the one shown in Fig. 1.19 pops up. You can now see the four steps envolved in the compilation, namely: `Analysis & Synthesis, Fitter, Assembler` and `Timing Analyzer`. Each of these steps has four small buttons. The first arrow to the right starts the processing, the second "equal-sign with a pen" is used for the assignments, the third is used to display the report files, and the last button starts an additional function, such as the hierarchy of the project, timing closure floorplan, programmer, or timing summary. To optimize the design for speed, click on the assignment symbol (equal-sign with a pen) in the `Analysis & Synthesis` step in the compiler window. A new window pops up that allows you to specify the `Optimization Technique` as `Speed`, `Balanced` or `Area`. Select the option `Speed`, and leave the other synthesis options unchanged. To set the target speed of the design click on the assignment button of the `Timing Analyzer` section and set the `Default required fmax` to 260 MHz. Note that you can also find all the assignments also under `EDA Tools Settings` under the `Assignment` menu. Next, start the `Analysis & Synthesis` by clicking on the right arrow in the compiler window or with `<Ctrl+K>` or by selecting `Start Analysis & Synthesis` in the `Start` item in the `Processing` menu. The compiler checks for basic syntax errors and produces a report file that lists resource estimation for the design. After the syntax check is successful, compilation can be started by pressing the large `Start` button in the lower left corner of the compiler tool window, or by pressing `<Ctrl+L>`. If all compiler steps were successfully completed, the design is fully implemented. Press the `Report` button in the compiler window to get a flow summary report that should show 32 LEs and 2048 memory bits use. Check the memory initialization file `sine.mif`, containing the sine table in offset binary form. This file was generated using the program `sine3e.exe` included on the CD-ROM under `book3e/util`. Figure 1.19 summarizes all the processing steps of the compilation, as shown in the Quartus II compiler window.

Floor Planing

The design results can be verified by clicking on the 4. button (i.e., `Timing Closure Floorplan` or opening the `Tool`→`Chip Editor`) to get a more-detailed view of the chip layout. The `Chip Editor` view is shown in Fig. 1.20. Use the `Zoom in` button (i.e., the ± magnifying glass) to produce the screen shown in Fig. 1.20. Zoom in to the area where the LAB and an M4K are highlighted in a different color. You should then see the two LABs used by the accumulation highlighted in blue and the M4K block highlighted in green. In addition several I/O cell are also highlighted in brown. Click on the `Critical Path Setting`[8] button and use the slider to select graph display.

[8] Note, as with all MS Window programs, just move your mouse over a button (no need to click on the button) and its name/function will be displayed.


Compiler Tool

Analysis & Synthesis	Fitter	Assembler	Timing Analyzer
100 %	100 %	100 %	100 %
00:00:11	00:00:36	00:00:25	00:00:05

Full Compilation
100 %
00:01:17

Fig. 1.19. Compilation steps in Quartus II.

You should then see in the `Chip Editor` a blue line that shows the worst-case path running from the first to the last bit of the accumulator. Now click on the `Bird's Eye View` button on the left menu buttons and an additional window will pop up. Now select the `Coordinate` option, i.e., the last entry in the `Main Window` options. You may also try out the connection display. First select for instance the M4K block and then press the button `Generate Fan-In Connections` or `Generate Fan-Out Connections` several times and more and more connections will be displayed.

Simulation

To simulate, open the `Simulator Tool` under the `Processing` menu. Under simulation mode you can select between `Functional, Timing` and `Timing using fast Timing Model`. The `Functional` simulation requires the func-

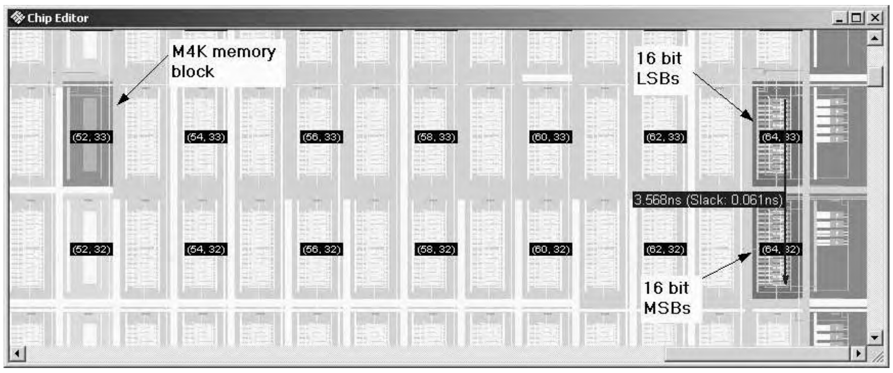
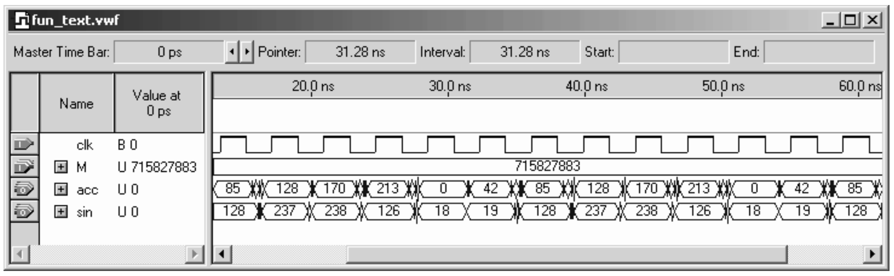


Fig. 1.20. Floorplan of the frequency synthesizer design.


fun_text.vwf

Master Time Bar: 0 ps Pointer: 31.28 ns Interval: 31.28 ns Start: End:

Name	Value at 0 ps
clk	B 0
M	U 715827883
acc	U 0
sin	U 0

Fig. 1.21. VHDL simulation of frequency synthesizer design.

tional netlist to be generate first; this takes additional time, but is much faster than the full compilation of the design. The `Timing` simulation requires that you first make a full compile of the design as described in the previous section. You should then click the `Open` button and the waveform window will pop up. Notice also that 24 new buttons on the left have been added. Move your mouse (without clicking) over the buttons to become familiar with their name and function. If you have copied the waveform file `fun_text.vwf` from the CD into the project directory you will see a simulation with timing loaded. If you start with an empty waveform Quartus II helps you to assemble a waveform file. Double click the space under the `Name` section to insert a node or bus. Click the `Node Finder` button and select in the `Node Finder` window as `Filter` the entry `Pins: all`. Now press `List` and all I/O pins are listed. Note that the I/O pins are available with both functional and timing simulation, but internal signals may not all be available in the netlist generated. Now select all bus signals, i.e., `acc`, `clk`, `M`, and `sin`, and press the single arrow to the right button. Now the four signals should be listed in the `Selected Nodes` window. If you selected only one bus signal repeat the selection by pressing the arrow with the other signals too. After all four signals are listed in the right window, press `OK`. Now the signal should be listed in the waveform window. Sort the signal according to Fig. 1.21, i.e., list first input control signals like `clk` followed by input data signal(s) `M`. In the last place in the list put the output data (`acc` and `sin`). To move a signal make sure that the arrow is selected from the left menu buttons. Then select the signal you would like to move and hold down the mouse button while moving the signal. When all signals are arranged correctly we can start defining the input signals. Select first the `clk` signal and then press the clock symbol from the left menu buttons. As the period select $1/260\,\text{MHz} = 3.846\,\text{ns}$. Next set $M = 715\,827\,883$ ($M = 2^{32}/6$), so that the period of the synthesizer is six clock cycles long. After we have specified the input signals we are ready to simulate. The default simulation time is always $1\mu s$. You can change the default value by selecting `End Time` under the `Edit` menu. You may set it to about 60 ns to display the first 15 clock cycles. You may want to start with a functional simulation first and then proceed with the timing simulation. Select `Functional Simulation` and do not forget to generate the functional netlist first. You can then press the `Start` button in the `Simulator Tool` window. You should see the waveforms without delays, i.e., the output signals should change exactly at the same time samples that the clock has a rising edge. You can then proceed with the timing simulation. Remember to conduct a full compile first if you have not done so already. Then press the `Start` button in the `Simulator Tool` window and you should see a waveform result similar to Fig. 1.21 that shows a simulation with delay. Notice that now the output signals no longer change exactly at the rising edge and that the signals also need some time to settle down, i.e., become stable. Make sure that the period is of length 6, both in the `accu` as well in the `sin` signal. Notice that the ROM has been coded in binary offset (i.e., zero = 128). This is a typical coding used in D/A converters and is used in the D/A converter of the Cyclone II DSP prototype board. When complete, change the frequency so that a period of eight cycles occurs, i.e., ($M = 2^{32}/8$), and repeat the simulation.

Performance Analysis

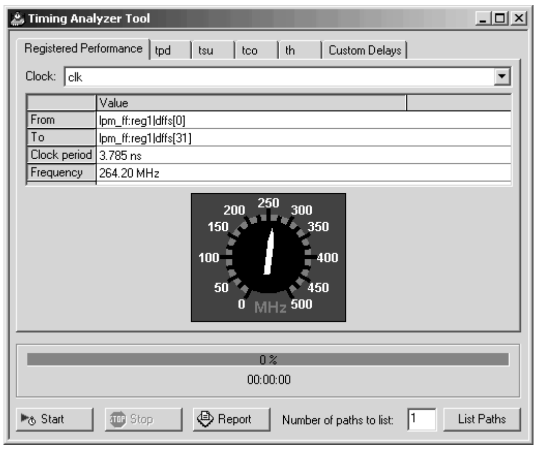
To initiate a performance analysis, select the `Timing Analyzer Tool` under the `Processing` menu. Usually the `Registered Performance` is the most important measurement. For a combination circuit (only) the propagation delay t_{pd} should be monitored. You can change the goals in the `EDA Tools Setting` under the `Assignment` menu. Note that it not sufficient to set the synthesis options to `Speed`; if you do not specify a requested `Default Required fmax` the synthesis results will most likely not reach the best performance that can be achieved with the device.

In order to display timing data a full compile of the design has to be done first. The `Registered Performance` is then displayed using a speed meter, while the other timing data like t_{pd} are shown in table form. The result for `Registered Performance` should be similar to that shown in Fig. 1.22. You can also list the worst-case path. Select 1 in `Number of path to list` and press the `List Paths` button. The path is shown as information in the message window. Pressing the plus sign, expand the information to see the full path detail. The path information of each node includes interconnect delay, cell delay, the LAB cell location with x and y coordinates, and the local signal name. You can verify this data using the `Chip Editor` described in the previous "Floor Planning" section.

This concludes the case study of the frequency synthesizer.

1.4.4 Design with Intellectual Property Cores

Although FPGAs are known for their capability to support rapid prototyping, this only applies if the HDL design is already available and sufficiently



Clock: clk

	Value
From	lpm_ff:reg1|dffs[0]
To	lpm_ff:reg1|dffs[31]
Clock period	3.785 ns
Frequency	264.20 MHz

Number of paths to list: 1

Fig. 1.22. Register performance of frequency synthesizer design.

tested. A complex block like a PCI bus interface, a pipelined FFT, an FIR filter, or a μP may take weeks or even months in development time. One option that allows us to essentially shorten the development time is available with the use of a so-called intellectual property (IP) core. These are predeveloped (larger) blocks, where typical standard blocks like numeric controlled oscillators (NCO), FIR filters, FFTs, or microprocessors are available from FPGA vendors directly, while more-specialized blocks (e.g., AES, DES, or JPEG codec, a floating-point library, or I2C bus or ethernet interfaces) are available from third-party vendors. On a much smaller scale we have already used IP blocks. The library of parameterize modules (LPM) blocks we used in the `example` and `fun_text` designs are parameterized cores, where we could select, for instance, bitwidth and pipelining that allow fast design development. We can use the LPM blocks and configure a pipelined multiplier or divider or we can specify to use memory blocks as CAM, RAM, ROM or FIFO. While this LPM blocks are free in the Quartus II package the larger more-sophisticated blocks may have a high price tag. But as long as the block meets your design requirement it is most often more cost effective to use one of these predefined IP blocks.

Let us now have a quick look at different types of IP blocks and discuss the advantages and disadvantages of each type [22, 23, 24]. Typically IP cores are divided into three main forms, as described below.

Soft Core

A *soft core* is a behavioral description of a component that needs to be synthesized with FPGA vendor tools. The block is typically provided in a hardware description language (HDL) like VHDL or Verilog, which allows easy modification by the user, or even new features to be added or deleted before synthesis for a specific vendor or device. On the downside the IP block may also require more work to meet the desired size/speed/power requirements. Very few of the blocks provided by FPGA vendors are available in this form, like the Nios microprocessor from Altera or the PICO blaze microprocessor by Xilinx. IP protection for the FPGA vendor is difficult to achieve since the block is provided as synthesizable HDL and can quite easily be used with a competing FPGA tool/device set or a cell-based ASIC. The price of third-party FPGA blocks provided in HDL is usually much higher than the moderate pricing of the parameterized core discussed next.

Parameterized Core

A *parameterized* or firm core is a structural description of a component. The parameters of the design can be changed before synthesis, but the HDL is usually not available. The majority of cores provided by Altera and Xilinx come in this type of core. They allow certain flexibility, but prohibit the use of the core with other FPGA vendors or ASIC foundries and therefore offers better IP protection for the FPGA vendors than soft cores. Examples of parameterized cores available from Altera and Xilinx include an NCO, FIR filter compiler, FFT (parallel and serial) and embedded processors, e.g., Nios II from Altera. Another advantage of parameterized cores is that usually a resource (LE, multiplier, block RAMs) is available that is most often correct within a few percent, which allows a fast design space exploration in terms of size/speed/power requirements even before synthesis. Testbenches in HDL (for ModelSim simulator) that allow cycle-accurate modeling as well as `C` or MatLab scripts that allow behavior-accurate modeling are also standard for parameterized cores. Code generation usually only takes a few seconds. Later in this section we will study an NCO parameterized core and continue this in later chapters (Chap. 3 on FIR filter and Chap. 6 on FFTs).

Hard Core

A *hard core* (fixed netlist core) is a physical description, provided in any of a variety of physical layout formats like EDIF. The cores are usually optimized for a specific device (family), when hard realtime constrains are required, like for instance a PCI bus interface. The parameters of the design are fixed, like a 16-bit 256-point FFT, but a behavior HDL description allows simulation and integration in a larger project. Most third-party IP cores from FPGA vendors and several free FFT cores from Xilinx use this core type. Since the layout is fixed, the timing and resource data provided are precise and do not depend on synthesis results. But the downside is that a parameter change is not possible, so if the FFT should have 12- or 24-bit input data the 16-bit 256-point FFT block can not be used.

IP Core Comparison and Challenges

If we now compare the different IP block types we have to choose between design flexibility (soft core) and fast results and reliability of data (hard core). Soft cores are flexible, e.g., change of system parameters or device/process technology is easy, but may have longer debug time. Hard cores are verified in silicon. Hard cores reduce development, test, and debug time but no VHDL code is available to look at. A parameterized core is most often the best compromise between flexibility and reliability of the generated core.

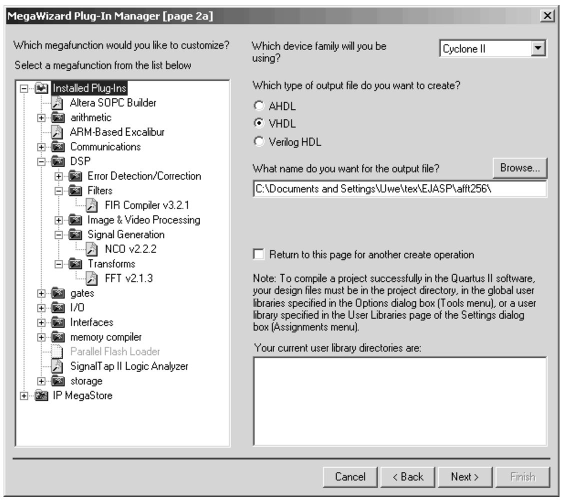
There are however two major challenges with current IP block technology, which are pricing of a block and, closely related, IP protection. Because the cores are reusable vendor pricing has to rely on the number of units of IP blocks the customer will use. This is a problem known for many years in patent rights and most often requires long licence agreements and high penalties in case of customer misuse. FPGA-vendor-provided parameterized blocks (as well as the design tool) have very moderate pricing since the vendor will profit if a customer uses the IP block in many devices and then usually has to buy the devices from this single source. This is different with third-party IP block providers that do not have this second stream of income. Here the licence agreement, especially for a soft core, has be drafted very carefully.

For the protection of parameterized cores FPGA vendor use FlexLM-based keys to enable/disable single IP core generation. Evaluation of the parameterized cores is possible down to hardware verification by using time-limited programming files or requiring a permanent connection between the host PC and board via a JTAG cable, allowing you to program devices and verify your design in hardware before purchasing a licence. For instance, Altera's OpenCore evaluation feature allows you to simulate the behavior of an IP core function within the targeted system, verify the functionality of the design, and evaluate its size and speed quickly and easily. When you are completely satisfied with the IP core function and you would like to take the design into production, you can purchase a licence that allows you to generate non-time-limited programming files. The Quartus software automatically downloads the latest IP cores from Altera's website. Many third-party IP providers also support the OpenCore evaluation flow but you have to contact the IP provider directly in order to enable the OpenCore feature.

The protection of soft cores is more difficult. Modification of the HDL to make them very hard to read, or embedding watermarks in the high-level design by minimizing the extra hardware have been suggested [24]. The watermark should be robust, i.e., a single bit change in the watermark should not be possible without corrupting the authentication of the owner.

IP Core-Based NCO Design

Finally we evaluate the design process of an IP block in an example using the case study from the last section, but this time our design will use Altera's


MegaWizard Plug-In Manager [page 2a]

Which megafunction would you like to customize?
Select a megafunction from the list below

- Installed Plug-Ins
  - Altera SOPC Builder
  - arithmetic
  - ARM-Based Excalibur
  - Communications
  - DSP
    - Error Detection/Correction
    - Filters
      - FIR Compiler v3.2.1
    - Image & Video Processing
    - Signal Generation
      - NCO v2.2.2
    - Transforms
      - FFT v2.1.3
  - gates
  - I/O
  - Interfaces
  - memory compiler
  - Parallel Flash Loader
  - SignalTap II Logic Analyzer
  - storage
- IP MegaStore

Which device family will you be using? Cyclone II

Which type of output file do you want to create?
AHDL [✓ VHDL] Verilog HDL

What name do you want for the output file? Browse...
C:\Documents and Settings\Uwe\tex\EJASP\afft256\

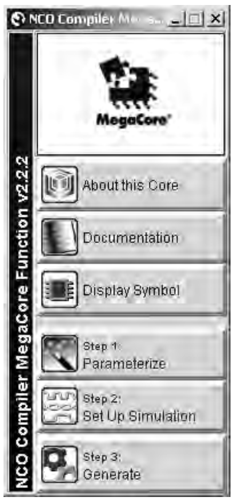
Return to this page for another create operation

Note: To compile a project successfully in the Quartus II software, your design files must be in the project directory, in the global user libraries specified in the Options dialog box (Tools menu), or a user library specified in the User Libraries page of the Settings dialog box (Assignments menu).

Your current user library directories are:

Cancel | < Back | Next > | Finish

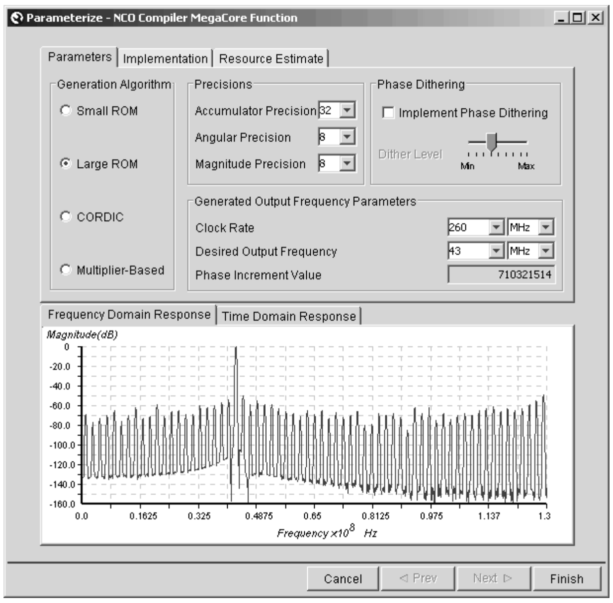
(a)



(b)

Fig. 1.23. IP design of NCO **(a)** Library element selection. **(b)** IP toolbench.

NCO core generator. The NCO compiler generates numerically controlled oscillators (NCOs) optimized for Altera devices. You can use the IP toolbench interface to implement a variety of NCO architectures, including ROM-based, CORDIC-based, and multiplier-based options. The MegaWizard also includes time- and frequency-domain graphs that dynamically display the functionality of the NCO based on the parameter settings. For a simple evaluation we use the graphic design entry. Open a new project and `BDF` file, then double click in the empty space in the `BDF` window and press the button `MegaWizard Plug-In Manager`. In the first step select the NCO block in the `MegaWizard Plug-In Manager` window, see Fig. 1.23a. The NCO block can by found in the `Signal Generation` group under the `DSP` cores. We then select the desired output format (AHDL, VHDL, or Verilog) and specify our working directory. Then the IP toolbench pops up (see Fig. 1.23b) and we have access to documentation and can start with step 1, i.e., the parametrization of the block. Since we want to reproduce the function generator from the last section, we select a 32-bit accumulator, 8 bit output precision, and the use of a large block of RAM in the parameter window, see Fig. 1.24. As expected for an 8 bit output we get about 50 dB sidelope suppression, as can be seen in the `Frequency Domain Response` plot in the lower part of the NCO window. Phase dithering will make the noise more equally distributed, but will require twice as many LEs. In the `Implementation` window we select `Single Output` since we only require one sine but no cosine output as is typical for I/Q receivers, see Chap. 7. The `Resource Estimation` provides as data 72 LEs, 2048 memory bits and one M4K block. After we are satisfied with our parameter selection we then proceed to step 2 to specify if we want


Parameterize - NCO Compiler MegaCore Function

Parameters | Implementation | Resource Estimate

Generation Algorithm: ( ) Small ROM [✓ Large ROM] ( ) CORDIC ( ) Multiplier-Based

Precisions
Accumulator Precision 32
Angular Precision 8
Magnitude Precision 8

Phase Dithering
☐ Implement Phase Dithering
Dither Level Min Max

Generated Output Frequency Parameters
Clock Rate 260 MHz
Desired Output Frequency 43 MHz
Phase Increment Value 710321514

Frequency Domain Response | Time Domain Response

Fig. 1.24. IP parametrization of NCO core according to the data from the case study in the previous section.

to generate behavior HDL code, which speeds up simulation time. Since our block is small we deselect this option and use the full HDL generated code directly. We can now continue with step 3, i.e., `Generate` on the Toolbench. The listing in Table 1.12 gives an overview of the generated files.

We see that not only are the VHDL and Verilog files generated along with their component file, but MatLab (bit accurate) and ModelTech (cycle accurate) testbenches are also provided to enable an easy verification path. We decide to instantiate our block in the graphical design and connect the input and outputs, see Fig. 1.25a. We notice that the block (outside that we have asked for) has some additional useful control signal, i.e., `reset`, `clken`, and `data_ready`, whose function is self-explanatory. All control signals are high active. We start with a `Functional` simulation first and then proceed (after a full compile) with the `Timing` simulation. With the full compile data available we can now compare the actual resource requirement with the estimate. The memory requirement and block RAM predictions were correct, but for the LEs with 86 LEs (actual) to 72 LEs (estimated) we observe a 17% error margin. Using the same value $M = 715\,827\,883$ as in our function generator (see Fig. 1.21, p. 34) we get a period of 6 in the output signal, as shown in Fig. 1.25b. We may notice a small problem with the IP block, since the output is a signed value, but our D/A converter expects unsigned (or more precisely binary offset) numbers. In a soft core we would be able to change the HDL code of the design, but in the parameterized core we do not have this option.

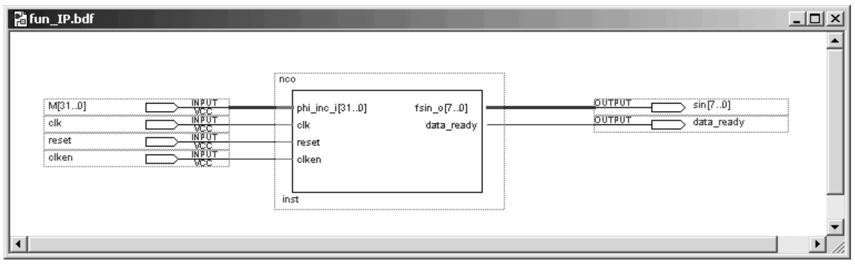
Table 1.12. IP file generation for the NCO core.

File	Description
`nco.bsf`	Quartus II symbol file for the IP core function variation
`nco.cmp`	VHDL component declaration for the IP core function variation
`nco.html`	IP core function report file that lists all generated files
`nco.inc`	AHDL include declaration file for the IP core function variation
`nco.vec`	Quartus vector file
`nco.vhd`	VHDL top-level description of the custom IP core function
`nco.vho`	VHDL IP functional simulation model
`nco_bb.v`	Verilog HDL black-box file for the IP core function variation
`nco_inst.vhd`	VHDL sample instantiation file
`nco_model.m`	MatLab M-file describing a MatLab bit-accurate model
`nco_sin.hex`	Intel Hex-format ROM initialization file
`nco_st.v`	Generated NCO synthesizable netlist
`nco_tb.m`	MatLab Testbench
`nco_tb.v`	Verilog testbench
`nco_tb.vhd`	VHDL testbench
`nco_vho_msim.tcl`	ModelSim TCL Script to run the VHDL IP functional simulation model in the ModelSim simulation software
`nco_wave.do`	ModelSim waveform file

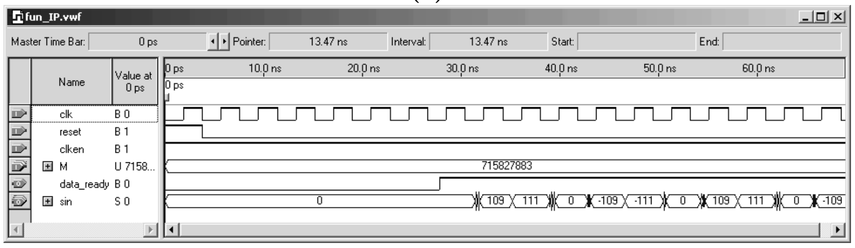
But we can solved this problem by attaching an adder with constant 128 to the output that make it an offset binary representation. The offset binary is not a parameter we could select in the block, and we encounter extra design effort. This is a typical experience with the parameterized cores – the core provide a 90% or more reduction in design time, but sometimes small extra design effort is necessary to meet the exact project requirements.

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 29. If not otherwise noted use the



(a)



(b)

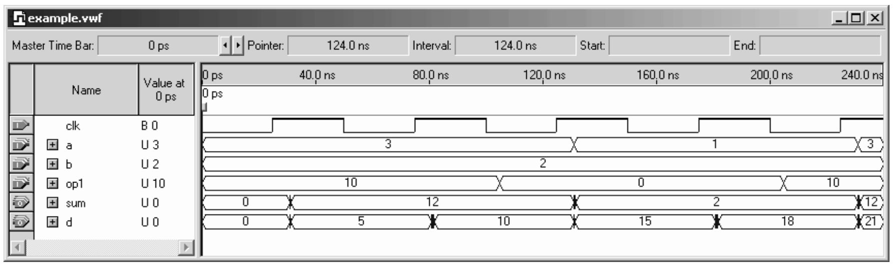
Fig. 1.25. Testbench for NCO IP design **(a)** Instantiation of IP block in graphical design. **(b)** Verification via simulation.

EP2C35F672C6 from the Cyclone II family for the Quartus II synthesis evaluations.

1.1: Use only two input NAND gates to implement a full adder:
(a) $s = a \oplus b \oplus c_{\text{in}}$
(Note: $\oplus$=XOR)
(b) $c_{\text{out}} = a \times b + c_{\text{in}} \times (a + b)$
(Note: +=OR; ×=AND)
(c) Show that the two-input NAND is *universal* by implementing NOT, AND, and OR with NAND gates.
(d) Repeat (a)-(c) for the two input NOR gate.
(e) Repeat (a)-(c) for the two input multiplexer $f = xs' + ys$.

1.2: **(a)** Compile the HDL file `example` using the Quartus II compiler (see p. 15) in the `Functional` mode. Start first the `Simulation Tool` under the `Processing` menu. Then select `Functional` as `Simulation mode` and press the button `Generate Functional Simulation Netlist`.
(b) Simulate the design using the file `example.vwf`.
(c) Compile the HDL file `example` using the Quartus II compiler with `Timing`. Perform a full compilation using the `Compiler Tool` under the `Processing` menu. Then select `Timing` as `Simulation mode` option in the the `Simulation Tool`.
(d) Simulate the design using the file `example.vwf`.
(e) Turn on the option `Check Outputs` in the simulator window and compare the functional and timing netlists.

1.3: **(a)** Generate a waveform file for `clk,a,b,op1` that approximates that shown in Fig. 1.26.


example.vwf

Master Time Bar: 0 ps Pointer: 124.0 ns Interval: 124.0 ns Start: End:

Fig. 1.26. Waveform file for Example 1.1 on p. 15.

(b) Conduct a simulation using the HDL code `example`.
(c) Explain the algebraic relation between `a,b,op1` and `sum,d`.

1.4: (a) Compile the HDL file `fun_text` with the synthesis optimization technique set to `Speed`, `Balanced` or `Area` that can be found in the `Analysis & Synthesis Settings` under `EDA Tool Settings` in the `Assignments` menu.
(b) Evaluate `Registered Performance` and the LE's utilization of the designs from (a). Explain the results.

1.5: (a) Compile the HDL file `fun_text` with the synthesis `Optimization Technique` set to `Speed` that can be found in the `Analysis & Synthesis Settings` under `EDA Tool Settings` in the `Assignments` menu.
For the period of the clock signal
(I) 20 ns,
(II) 10 ns,
(III) 5 ns,
(IV) 3 ns,
use the waveform file `fun_text.vwf` and enable
(b) `Setup and hold time violation detection`,
(c) `Glitch detection`, and
(d)`Check outputs`.
Select one option after the other and not all three at the same time. For `Check outputs` first make a `Functional Simulation`, then select `Check outputs`, and perform then `Timing` simulation. Under `Waveform Comparison Setting` select `sin` for comparison and deselect all other signals. Set the `Default comparison timing tolerance` to `<<default>>`, i.e., halve the clock period or the falling edge of clock. Click on the `Report` button in the `Simulation Tool` window if there are violation.

1.6: (a) Open the file `fun_text.vwf` and start the simulation.
(b) Select `File`→`Open` to open the file `sine.mif` and the file will be displayed in the `Memory editor`. Now select `File`→`Save As` and select `Save as type: (*.hex)` to store the file in Intel HEX format as `sine.hex`.
(c) Change the `fun_text` HDL file so that it uses the Intel HEX file `sine.hex` for the ROM table, and verify the correct results through a simulation.

1.7: (a) Design a 32-bit adder using the `LPM_ADD_SUB` macro with the Quartus II software.
(b) Measure the `Registered Performance` and compare the result with the data from Example 1.2 (p. 26).

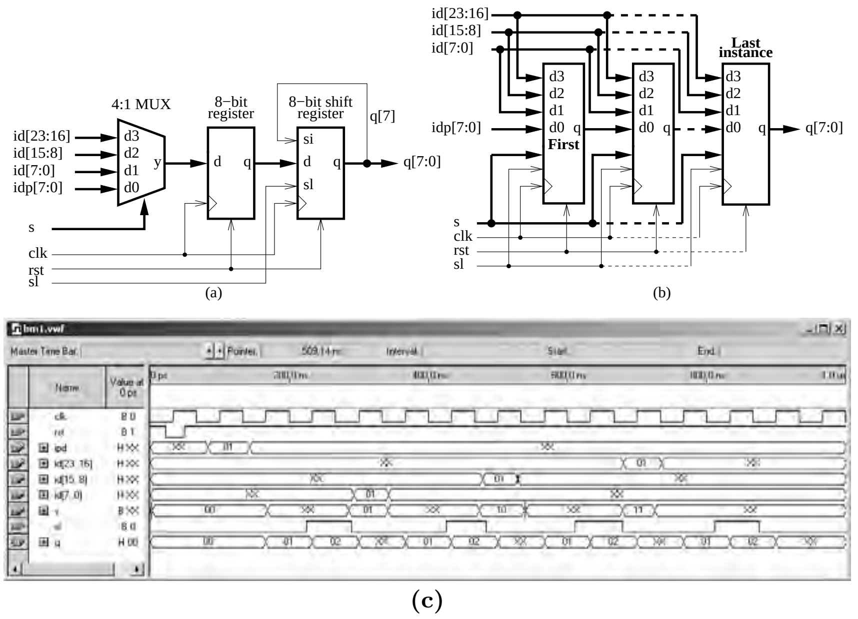
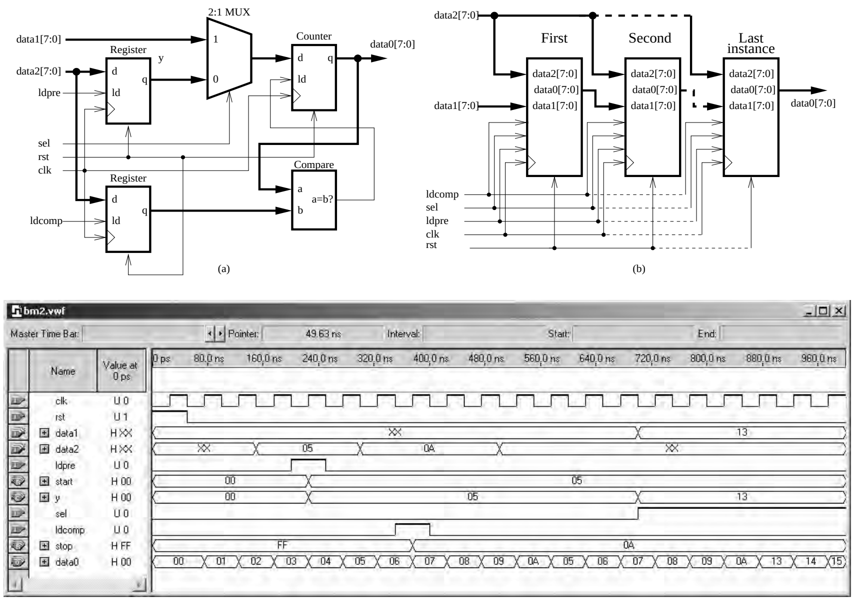


Fig. 1.27. PREP benchmark 1. **(a)** Single design. **(b)** Multiple instantiation. **(c)** Testbench to check the function.

1.8: **(a)** Design the PREP benchmark 1, as shown in Fig. 1.27a with the Quartus II software. PREP benchmark no. 1 is a data path circuit with a 4-to-1 8-bit multiplexer, an 8-bit register, followed by a shift register that is controlled by a shift/load input `sl`. For `sl=1` the contents of the register is cyclic rotated by one bit, i.e., $q(k) = q(k-1), 1 \leq k \leq 7$ and $q(0) <= q(7)$. The reset `rst` for all flip-flops is an asynchronous reset and the 8-bit registers are positive-edge triggered via `clk`, see the simulation in Fig. 1.27c for the function test.
(b) Determine the `Registered Performance` and the used resources (LEs, multipliers, and M2Ks/M4Ks) for a single copy. Compile the HDL file with the synthesis `Optimization Technique` set to `Speed`, `Balanced` or `Area`; this can be found in the `Analysis & Synthesis Settings` section under `EDA Tool Settings` in the `Assignments` menu. Which synthesis options are optimal in terms of size and `Registered Performance`?
Select one of the following devices:
(b1) EP2C35F672C6 from the Cyclone II family
(b2) EPF10K70RC240-4 from the Flex 10K family
(b3) EPM7128LC84-7 from the MAX7000S family
(c) Design the multiple instantiation for benchmark 5 as shown in Fig. 1.27b.
(d) Determine the `Registered Performance` and the used resources (LEs, multipliers, and M2Ks/M4Ks) for the design with the maximum number of instantiations of PREP benchmark 1. Use the optimal synthesis option you found in (b) for the following devices:
(d1) EP2C35F672C6 from the Cyclone II family
(d2) EPF10K70RC240-4 from the Flex 10K family
(d3) EPM7128LC84-7 from the MAX7000S family



(c)

Fig. 1.28. PREP benchmark 2. **(a)** Single design. **(b)** Multiple instantiation. **(c)** Testbench to check the function.

1.9: **(a)** Design the PREP benchmark 2, as shown in Fig. 1.28a with the Quartus II software. PREP benchmark no. 2 is a counter circuit where 2 registers are loaded with start and stop values of the counter. The design has two 8-bit register and a counter with asynchronous reset `rst` and synchronous load enable signal (`ld, ldpre` and `ldcomp`) and positive-edge triggered flip-flops via `clk`. The counter can be loaded through a 2:1 multiplexer (controlled by the `sel` input) directly from the `data1` input or from the register that holds `data2` values. The load signal of the counter is enabled by the equal condition that compares the counter value `data` with the stored values in the `ldcomp` register. Try to match the simulation in Fig. 1.28c for the function test. Note there is a mismatch between the original PREP definition and the actual implementation: We can not satisfy, that the counter start counting after reset, because all register are set to zero and `ld` will be true all the time, forcing counter to zero. Also in the simulation testbench signal value have been reduced that simulation fits in a 1 μs time frame.
(b) Determine the `Registered Performance` and the used resources (LEs, multipliers, and M2Ks/M4Ks) for a single copy. Compile the HDL file with the synthesis `Optimization Technique` set to `Speed`, `Balanced` or `Area`; this can be found in the `Analysis & Synthesis Settings` section under `EDA Tool Settings` in the `Assignments` menu. Which synthesis options are optimal in terms of size and `Registered Performance`?
Select one of the following devices:
(b1) EP2C35F672C6 from the Cyclone II family
(b2) EPF10K70RC240-4 from the Flex 10K family
(b3) EPM7128LC84-7 from the MAX7000S family

(c) Design the multiple instantiation for benchmark 2 as shown in Fig. 1.28b.
(d) Determine the `Registered Performance` and the used resources (LEs, multipliers, and M2Ks/M4Ks) for the design with the maximum number of instantiations of PREP benchmark 2. Use the optimal synthesis option you found in (b) for the following devices:
(d1) EP2C35F672C6 from the Cyclone II family
(d2) EPF10K70RC240-4 from the Flex 10K family
(d3) EPM7128LC84-7 from the MAX7000S family

1.10: Use the Quartus II software and write two different codes using the structural (use only one or two input basic gates, i.e., `NOT`, `AND`, and `OR`) and behavioral HDL styles for:
(a) A 2:1 multiplexer
(b) An XNOR gate
(c) A half-adder
(d) A 2:4 decoder (demultiplexer)
Note for VHDL designs: use the `a_74xx` Altera SSI component for the structural design files. Because a component identifier can not start with a number Altera has added the `a_` in front of each 74 series component. In order to find the names and data types for input and output ports you need to check the library file `libraries\vhdl\altera\MAXPLUS2.VHD` in the Altera installation path. You will find that the library uses `STD_LOGIC` data type and the names for the ports are `a_1`, `a_2`, and `a_3` (if needed).
(e) Verify the function of the design(s) via
(e1) A `Functional` simulation.
(e2) The `RTL viewer` that can be found under the `Netlist Viewers` in the `Tools` menu.

1.11: Use the Quartus II software language templates and compile the HDL designs for:
(a) A tri-state buffer
(b) A flip-flop with all control signals
(c) A counter
(d) A state machine with asynchronous reset
Open a new HDL text file and then select `Insert Template` from the `Edit` menu.
(e) Verify the function of the design(s) via
(e1) A `Functional` simulation
(e2) The `RTL viewer` that can be found under the `Netlist Viewers` in the `Tools` menu

1.12: Use the `search` option in Quartus II software help to study HDL designs for:
(a) The 14 counters, see `search`→`implementing sequential logic`
(b) A manually specifying state assignments, `Search`→`enumsmch`
(c) A latch, `Search`→`latchinf`
(d) A one's counter, `Search`→`proc`→`Using Process Statements`
(e) A implementing CAM, RAM & ROM, `Search`→`ram256x8`
(f) A implementing a user-defined component, `Search`→`reg24`
(g) Implementing registers with clr, load, and preset, `Search`→`reginf`
(h) A state machine, `Search`→`state_machine`→`Implementing...`
Open a new project and HDL text file. Then `Copy/Paste` the HDL code, save and compile the code. Note that in VHDL you need to add the STD_LOGIC_1164 IEEE library so that the code runs error free.
(i) Verify the function of the design via
(i1) A `Functional` simulation

(i2) The `RTL viewer` that can be found under the `Netlist Viewers` in the `Tools` menu

1.13: Determine if the following VHDL identifiers are valid (true) or invalid (false).

(a) `VHSIC`	**(b)** `h333`	**(c)** `A_B_C`
(d) `XyZ`	**(e)** `N#3`	**(f)** `My-name`
(g) `BEGIN`	**(h)** `A__B`	**(i)** `ENTITI`

1.14: Determine if the following VHDL string literals are valid (true) or invalid (false).

(a) `B"11_00"`	**(b)** `O"5678"`	**(c)** `O"0_1_2"`
(d) `X"5678"`	**(e)** `16#FfF#`	**(f)** `10#007#`
(g) `5#12345#`	**(h)** `2#0001_1111_#`	**(i)** `2#00_00#`

1.15: Determine the number of bits necessary to represent the following integer numbers.
(a) `INTEGER RANGE 10 TO 20;`
(b) `INTEGER RANGE -2**6 TO 2**4-1;`
(c) `INTEGER RANGE -10 TO -5;`
(d) `INTEGER RANGE -2 TO 15;`
Note that `**` stand for the power-of symbol.

1.16: Determine the error lines (Y/N) in the VHDL code below and explain what is wrong, or give correct code.

VHDL code	Error (Y/N)	Give reason
`LIBRARY ieee;  /* Using predefined packages */`		
`ENTITY error is`		
`  PORTS (x: in BIT; c: in BIT;`		
`  Z1: out INTEGER; z2 : out BIT);`		
`END error`		
`ARCHITECTURE error OF has IS`		
`SIGNAL s ; w : BIT;`		
`BEGIN`		
`  w := c;`		
`  Z1 <= x;`		
`  P1: PROCESS (x)`		
`  BEGIN`		
`    IF c='1' THEN`		
`      x <= z2;`		
`  END PROCESS P0;`		
`END OF has;`		

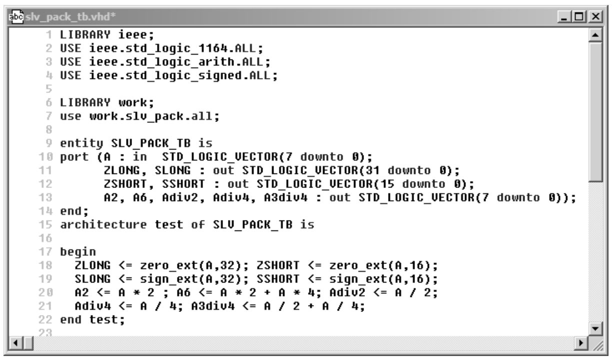
1.17: Determine the error lines (Y/N) in the VHDL code below, and explain what is wrong, or give correct code.

VHDL code	Error (Y/N)	Give reason
`LIBRARY ieee;  /* Using predefined packages */`		
`USE altera.std_logic_1164.ALL;`		
`ENTITY srhiftreg IS`		
`  GENERIC (WIDTH : POSITIVE = 4);`		
`  PORT(clk, din : IN STD_LOGIC;`		
`       dout     : OUT STD_LOGIC);`		
`END;`		
`ARCHITECTURE a OF shiftreg IS`		
`  COMPONENT d_ff`		
`    PORT (clock, d : IN  std_logic;`		
`          q       : OUT std_logic);`		
`  END d_ff;`		
`  SIGNAL b : logic_vector(0 TO witdh-1);`		
`BEGIN`		
`  d1: d_ff PORT MAP (clk, b(0), din);`		
`  g1: FOR j IN 1 TO width-1 GENERATE`		
`      d2: d-ff`		
`      PORT MAP(clk => clock,`		
`               din => b(j-1),`		
`               q => b(j));`		
`  END GENERATE d2;`		
`  dout <= b(width);`		
`END a;`		

1.18: Determine for the following process statements
(a) the synthesized circuit and label I/O ports
(b) the cost of the design assuming a cost 1 per adder/subtractor
(c) the critical (i.e., worst-case) path of the circuit for each process. Assume a delay of 1 for an adder or subtractor.

```
-- QUIZ VHDL2graph for DSP with FPGAs
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY qv2g IS
 PORT(a, b, c, d : IN  std_logic_vector(3 DOWNTO 0);
      u, v, w, x, y, z : OUT std_logic_vector(3 DOWNTO 0));
END;
ARCHITECTURE a OF qv2g IS BEGIN

  P0: PROCESS(a, b, c, d)
  BEGIN
    u <= a + b - c + d;
  END PROCESS;

  P1: PROCESS(a, b, c, d)
  BEGIN
    v <= (a + b) - (c - d);
  END PROCESS;

  P2: PROCESS(a, b, c)
  BEGIN
    w <= a + b + c;
    x <= a - b - c;
  END PROCESS;

  P3: PROCESS(a, b, c)
  VARIABLE t1 :  std_logic_vector(3 DOWNTO 0);
  BEGIN
    t1 := b + c;
    y <= a + t1;
    z <= a - t1;
  END PROCESS;
END;
```

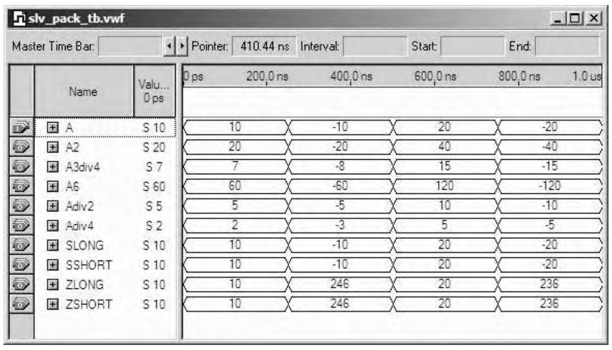
(a)



slv_pack_tb.vhd*

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

LIBRARY work;
use work.slv_pack.all;

entity SLV_PACK_TB is
port (A : in  STD_LOGIC_VECTOR(7 downto 0);
      ZLONG, SLONG : out STD_LOGIC_VECTOR(31 downto 0);
      ZSHORT, SSHORT : out STD_LOGIC_VECTOR(15 downto 0);
      A2, A6, Adiv2, Adiv4, A3div4 : out STD_LOGIC_VECTOR(7 downto 0));
end;
architecture test of SLV_PACK_TB is

begin
  ZLONG <= zero_ext(A,32); ZSHORT <= zero_ext(A,16);
  SLONG <= sign_ext(A,32); SSHORT <= sign_ext(A,16);
  A2 <= A * 2 ; A6 <= A * 2 + A * 4; Adiv2 <= A / 2;
  Adiv4 <= A / 4; A3div4 <= A / 2 + A / 4;
end test;
```

(b)



slv_pack_tb.vwf

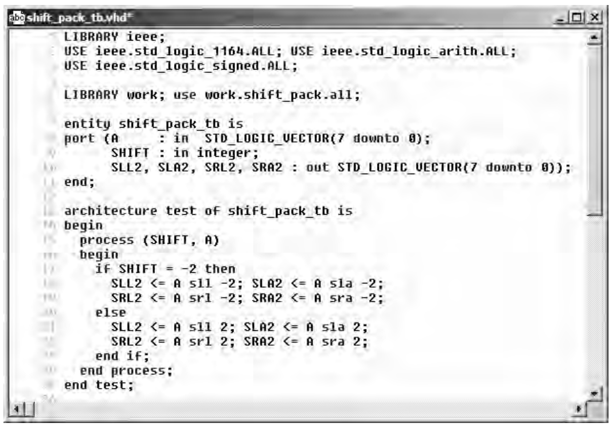
Master Time Bar: Pointer: 410.44 ns Interval: Start: End:

Name	Valu... 0 ps	0 ps	200,0 ns	400,0 ns	600,0 ns	800,0 ns
A	S 10	10	-10		20	-20
A2	S 20	20	-20		40	-40
A3div4	S 7	7	-8		15	-15
A6	S 60	60	-60		120	-120
Adiv2	S 5	5	-5		10	-10
Adiv4	S 2	2	-3		5	-5
SLONG	S 10	10	-10		20	-20
SSHORT	S 10	10	-10		20	-20
ZLONG	S 10	10	246		20	236
ZSHORT	S 10	10	246		20	236

Fig. 1.29. STD_LOGIC_VECTOR package testbench. **(a)** HDL code. **(b)** Functional simulation result.

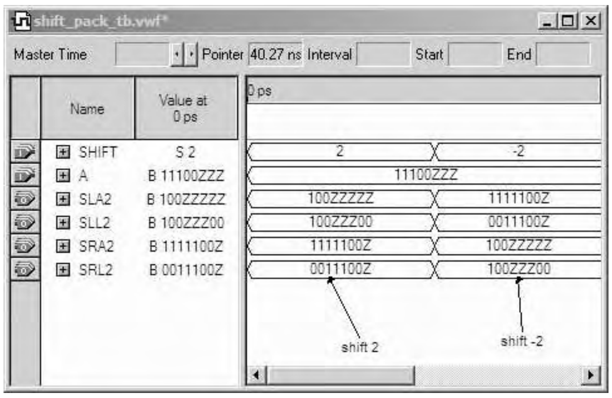
1.19: **(a)** Develop a functions for zero- and sign extension called `zero_ext(ARG,SIZE)` and `sign_ext(ARG,SIZE)` for the `STD_LOGIC_VECTOR` data type.
(b) Develop "$*$" and "/" function overloading to implement multiply and divide operation for the `STD_LOGIC_VECTOR` data type.
(c) Use the testbench shown in Fig. 1.29 to verify the correct functionality.

(a)



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL; USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

LIBRARY work; use work.shift_pack.all;

entity shift_pack_tb is
port (A     : in  STD_LOGIC_VECTOR(7 downto 0);
      SHIFT : in integer;
      SLL2, SLA2, SRL2, SRA2 : out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture test of shift_pack_tb is
begin
  process (SHIFT, A)
  begin
    if SHIFT = -2 then
      SLL2 <= A sll -2; SLA2 <= A sla -2;
      SRL2 <= A srl -2; SRA2 <= A sra -2;
    else
      SLL2 <= A sll 2; SLA2 <= A sla 2;
      SRL2 <= A srl 2; SRA2 <= A sra 2;
    end if;
  end process;
end test;
```

(b)



Name	Value at 0 ps		
SHIFT	S 2	2	-2
A	B 11100ZZZ	11100ZZZ	
SLA2	B 100ZZZZZ	100ZZZZZ	1111100Z
SLL2	B 100ZZZ00	100ZZZ00	0011100Z
SRA2	B 1111100Z	1111100Z	100ZZZZZ
SRL2	B 0011100Z	0011100Z	100ZZZ00

Fig. 1.30. `STD_LOGIC_VECTOR` shift library testbench. **(a)** HDL code. **(b)** Functional simulation result.

1.20: **(a)** Design a function library for the `STD_LOGIC_VECTOR` data type that implement the following operation (defined in VHDL only for the `bit_vector` data type):
(a) `srl` **(b)** `sra` **(c)** `sll` **(d)** `sla`
(e) Use the testbench shown in Fig. 1.30 to verify the correct functionality. Note the high impedance values `Z` that are part of the `STD_LOGIC_VECTOR` data type but are not included in the `bit_vector` data type. A left/right shift by a negative value should be replaced by the appropriate right/left shift of the positive amount inside your function.

1.21: Determine for the following `PROCESS` statements the synthesized circuit type (combinational, latch, D-flip-flop, or T-flip-flop) and the function of `a`, `b`, and `c`, i.e., clock, a-synchronous set (AS) or reset (AR) or synchronous set (SS) or reset (SR). Use the table below to specify your classification.

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;

ENTITY quiz IS
  PORT(a, b, c : IN  std_logic;
       d       : IN  std_logic_vector(0 TO 5);
       q       : BUFFER std_logic_vector(0 TO 5));
END quiz;
ARCHITECTURE a OF quiz IS BEGIN
  P0: PROCESS (a)
  BEGIN
    IF rising_edge(a) THEN
      q(0) <= d(0);
    END IF;
  END PROCESS P0;

  P1: PROCESS (a, d)
  BEGIN
    IF a= '1' THEN  q(1) <= d(1);
              ELSE  q(1) <= '1';
    END IF;
  END PROCESS P1;

  P2: PROCESS (a, b, c, d)
  BEGIN
    IF a = '1' THEN q(2) <= '0';
    ELSE IF rising_edge(b) THEN
           IF c = '1' THEN q(2) <= '1';
                      ELSE q(2) <= d(1);
           END IF;
         END IF;
    END IF;
  END PROCESS P2;

  P3: PROCESS (a, b, d)
  BEGIN
    IF a = '1' THEN q(3) <= '1';
    ELSE IF rising_edge(b) THEN
           IF c = '1' THEN q(3) <= '0';
                      ELSE q(3) <= not q(3);
           END IF;
         END IF;
    END IF;
  END PROCESS P3;

  P4: PROCESS (a, d)
  BEGIN
    IF a = '1' THEN q(4) <= d(4);
    END IF;
  END PROCESS P4;

  P5: PROCESS (a, b, d)
  BEGIN
    IF rising_edge(a) THEN
      IF b = '1' THEN q(5) <= '0';
                 ELSE q(5) <= d(5);
      END IF;
    END IF;
    END PROCESS P5;
```

Process	Circuit type	CLK	AS	AR	SS	SR
P0						
P1						
P2						
P3						
P4						
P5						
P6						

1.22: Given the following MatLab instructions,

```
a=-1:2:5
b=[ones(1,2),zeros(1,2)]
c=a*a'
d=a.*a
e=a'*a
f=conv(a,b)
g=fft(b)
h=ifft(fft(a).*fft(b))
```

determine `a`-`h`.