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 EP4CE115F29C7N 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 Chap. 2 (p. 126) and Chap. 9 to PDSPs after we have studied FPGA architectures.

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 track-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 (e.g., 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.

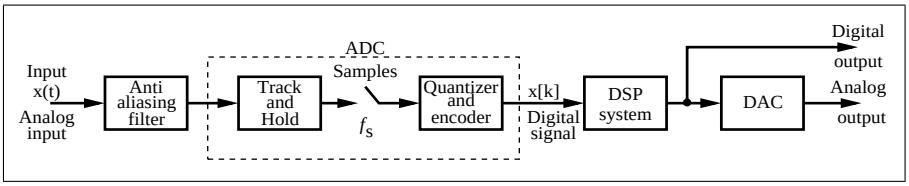


Fig. 1.1. A typical DSP application.

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 40 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. 12.

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 single NAND gate and a latch (see Fig. 1.3). Because it is possible to realize

[2] Called slice or configurable logic block (CLB) by Xilinx, logic cell (LC), logic element (LE), or adaptive logic module (ALM) by Altera.

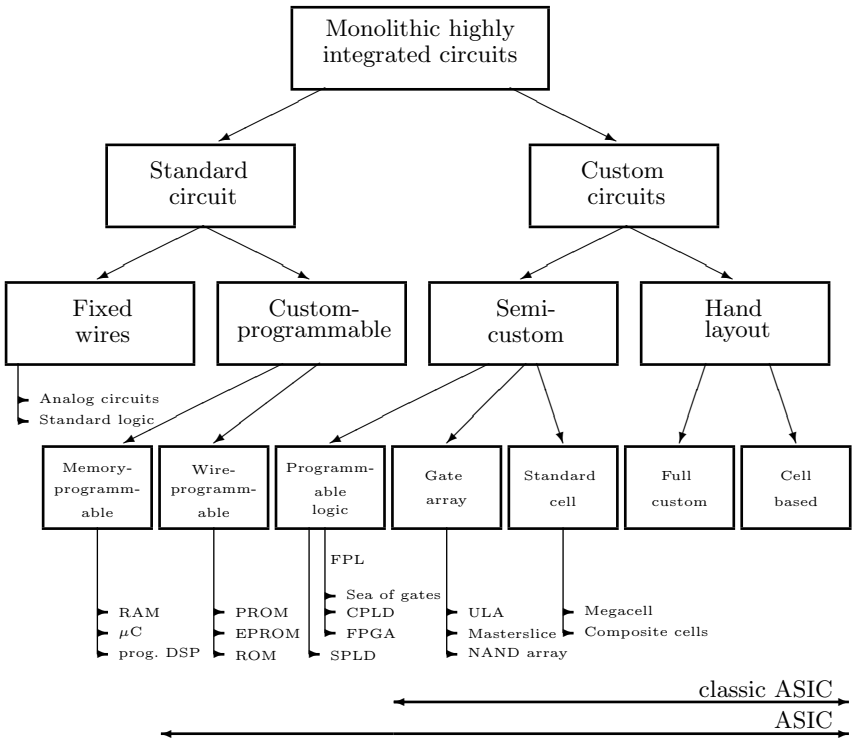


Fig. 1.2. Classification of VLSI circuits.

any binary logic function using NAND gates (see Exercise 1.1, p. 46), 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. Concrete examples of contemporary medium-grain FPGA devices are shown in Fig. 1.11 and Fig. 1.12. The elementary logic blocks are typically small tables, (typically with 4- to 5-bit input tables, 1- or 2-bit output), or are realized with dedicated multiplexer (MPX) logic such as that used in Actel ACT-2 devices

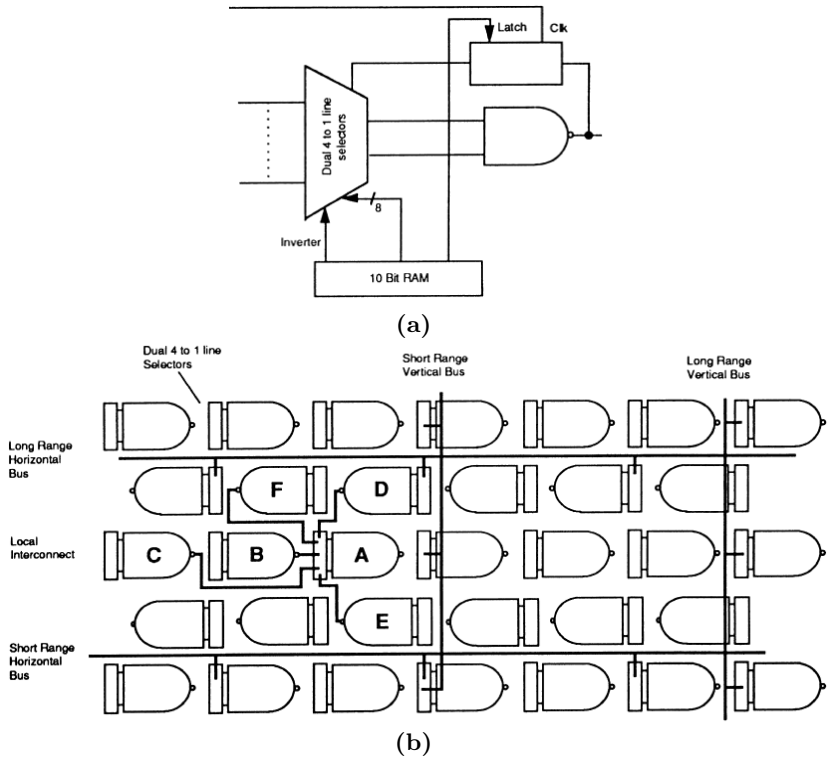


Fig. 1.3. Plessey ERA60100 architecture with 10K NAND logic blocks. **(a)** Elementary logic block. **(b)** Routing architecture (©Plessey [9]).

[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. 1.5. 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.

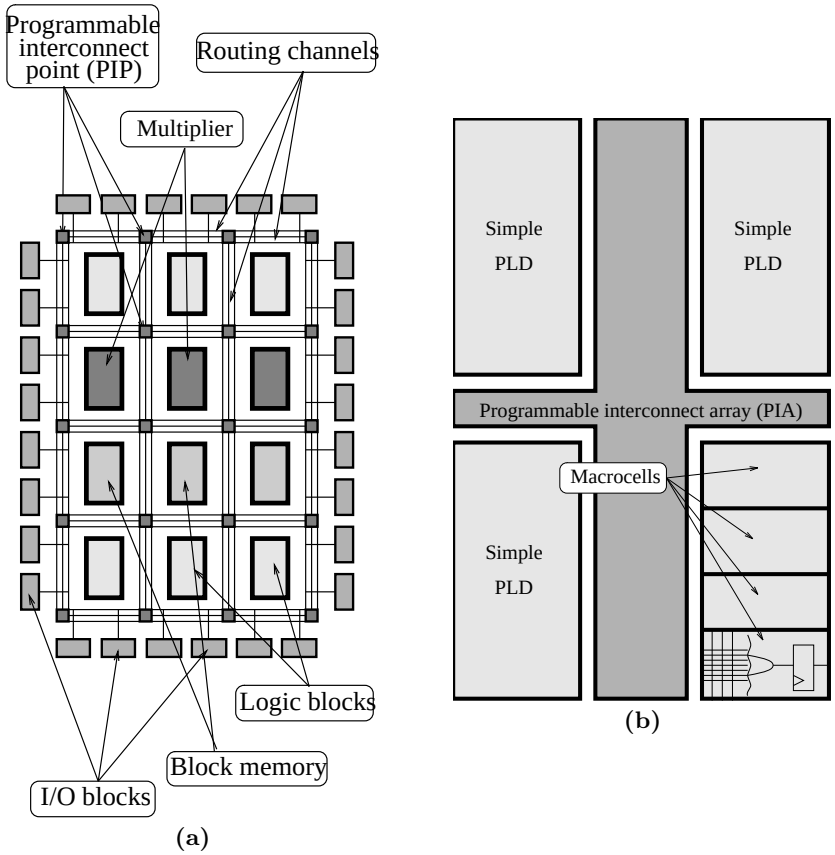


Fig. 1.4. (a) FPGA and **(b)** CPLD architecture.

1.2.2 Classification by Technology

FPLs are available in virtually all memory technologies: SRAM, EPROM, E^2PROM, and antifuse [12]. 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

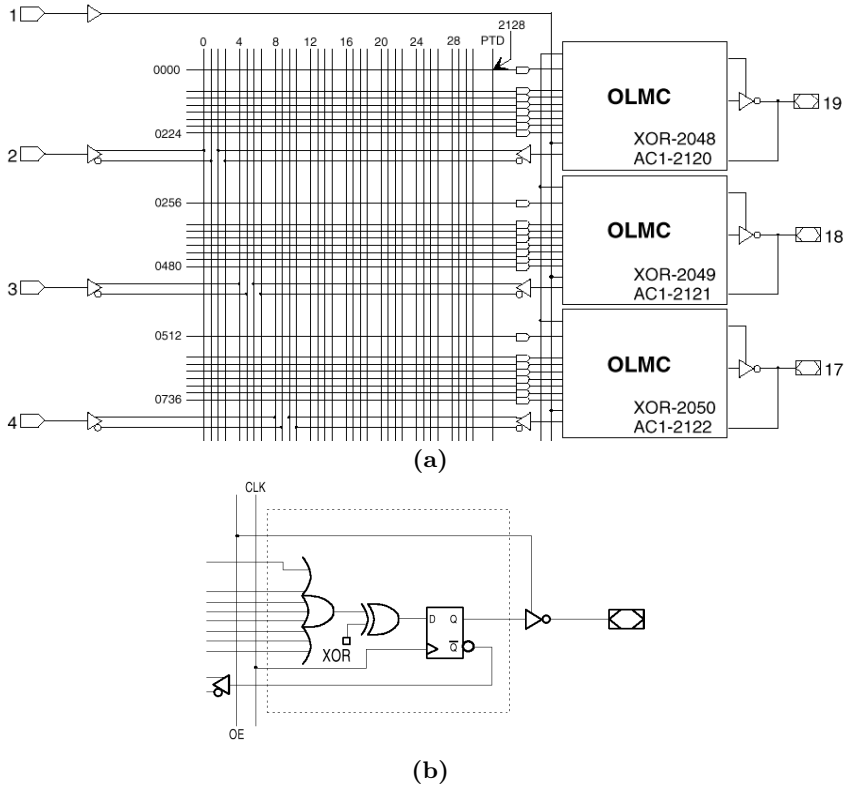


Fig. 1.5. The GAL16V8. **(a)** First three of eight macrocells. **(b)** The output logic macrocell (OLMC) (©Lattice [11]).

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^2PROM 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 [13], Altera [14], and Actel [15], and later expanded to more than 10 members. PREP has developed nine different benchmarks for FPLs that are summarized in

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

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 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. 1.6, repetition rates are reported over frequency, for typical Altera (A_k) and Xilinx (X_k) FPGA and CPLD devices that are currently used on the University development boards. This are not always the largest devices available, but all devices are supported by the web based version of the design tools. Xilinx seemed to achieve the larger speed, while the Altera FPGAs have a larger number of repetitions. Compared with the CPLDs 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. 20) 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, 18 × 25 bits)
- Package such as BGA, TQFP, PGA
- Configuration data stream encryption via DES or AES
- Embedded hardwired microprocessor (e.g., 32-bit ARM Cortex-A9)
- On-chip large block RAM or ROM
- On-chip fast analog-to-digital converter
- External memory support for ZBT, DDR, QDR, SDRAM
- Pin-to-pin delay

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. 1.26, p. 48)
2	Timer/counter	Two 8-bit values are clocked through 8-bit value registers and compared (see Fig. 1.27, p. 49)
3	Small state machine	An 8-state machine with 8 inputs and 8 outputs (see Fig. 2.64, p. 174)
4	Large state machine	A 16-state machine with 40 transitions, 8 inputs, and 8 outputs (see Fig. 2.65, p. 175)
5	Arithmetic circuit	A 4-by-4 unsigned multiplier and 8-bit accumulator (see Fig. 4.61, p. 302)
6	16-bit accumulator	A 16-bit accumulator (see Fig. 4.62, p. 303)
7	16-bit counter	Loadable binary up counter (see Fig. 9.42, p. 738)
8	16-bit synchronous prescaled counter	Loadable binary counter with asynchronous reset (see Fig. 9.42, p. 738)
9	Memory mapper	The map decodes a 16-bit address space into 8 ranges (see Fig. 9.43, p. 739)

- Internal tristate bus
- Readback- or boundary-scan decoder
- Programmable slew rate or voltage of I/O
- Power dissipation
- Hard IP block for $\times 1, \times 2$, or $\times 4$ PCIe

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 $2-7$ show the (for most DSP applications) relevant features: (2) the number of address inputs (a.k.a. Fan-in) to the LUT, (3) the size of the embedded array multiplier, (4) the size of the on-chip block RAM measured as kilo (1024) bits, (5) embedded microprocessor: 32-bit ARM Cortex-A9 on current Xilinx ZYNQ and Altera devices, (6) For Xilinx devices the on-chip (Virtex 6: 10 bit, 0.2 MSPS; Series 7: 12 bit 1 MSPS) fast Analog-to-Digital converter,

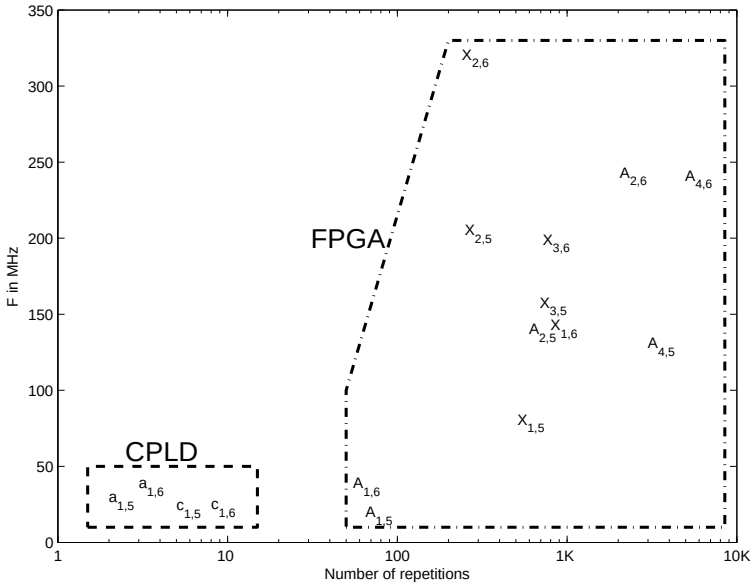


Fig. 1.6. PREP benchmark 5 and 6 (i.e., second subscript) average for FPLs from the Digilent and TERASIC development boards: A_1=FLEX10K from UP2; A_2=Cyclone 2 from DE2; A_3=Cyclone IV from DE2-115; a_1=EPM7128 from UP2; X_1=Spartan 3 from Nexys; X_2=Spartan 6 from Nexys III; X_3=Spartan 6 LX45 from Atlys; and c_1=CoolRunnerII CPLD.

(7) 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 $$, (8) Year the device family was introduced, (9) The process technology used measured in nanometer.

At time of writing Xilinx supports four device families: The Virtex family for leading performance and capacity; the Kintex family for DSP intensive application and low cost and the Artix family of lowest cost, replacing the Spartan family of devices. In addition a embedded microprocessor centric family called ZYNQ has been introduced. The Virtex-II, Virtex-4-FX or Virtex-5-FXT families that included one or more IBM PowerPC RISC processor no longer are recommended for new designs. The Xilinx device have 18×18 bit and 18×25 bits embedded multipliers. Most current devices provide a 18 or 36 Kbits memory. An 0.2 MSPS 10 bit fast A/D converter on-chip has been added for the 6th Virtex generation. The 7th generation includes 12 bit 1 MSPS dual channel ADC with additional senors for power supply and on-chip temperature with possible 17 senors sources for the ADCs, see Fig. 1.7b. Keep in mind that only a larger number of the Spartan families are available

Table 1.4. Recent Xilinx FPGA family DSP features.

Family	Feature							
	LUT Fan-in	Emb. mult.	BRAM size kbits	Emb. µP	Fast A/D	Cost / mature	Year	Process nm
Spartan 3	4	18 × 18	18	–	–	m	2003	90
Spartan 6	6	18 × 18	18	–	–	$	2009	45
Virtex 4	4	18 × 18	36	PPC	–	m	2004	90
Virtex 5	6	25 × 18	36	PPC	–	m	2006	65
Virtex 6	6	25 × 18	36	–	✓	$$	2009	40
Artix 7	6	25 × 18	36	–	✓	$	2010	28
Kintex 7	6	25 × 18	36	–	✓	$$	2010	28
Virtex 7	6	25 × 18	36	–	✓	$$	2010	28
ZYNQ-7K	6	25 × 18	36	ARM	✓	$$	2011	28

Table 1.5. Altera FPGA family DSP features.

Family	Feature							
	LUT Fan-in	Emb. mult. size	BRAM size Kbits	Emb. µP	Fast A/D	Cost ... mature	Year	Process nm
FLEX10K	4	–	4	–	–	m	1995	420
Cyclone	4	–	4	–	–	$	2002	130
Cyclone II	4	18 × 18	4	–	–	$	2004	90
Cyclone III	4	18 × 18	9	–	–	$	2007	65
Cyclone IV	4	18 × 18	9	–	–	$	2009	60
Cyclone V	8	27 × 27	10	–	–	$	2011	28
Arria	8	18 × 18	576	–	–	$	2007	90
Arria II	8	18 × 18	9	–	–	$	2009	40
Arria V	8	27 × 27	10	ARM	–	$$	2011	28
Stratix	4	18 × 18	0.5,4,512	–	–	$$	2002	130
Stratix II	8	18 × 18	0.5,4,512	–	–	$$	2004	90
Stratix III	8	18 × 18	9,144	–	–	$$	2006	65
Stratix IV	8	18 × 18	9,144	–	–	$$	2008	40
Stratix V	8	27 × 27	20	–	–	$$	2010	28

in the web edition of the development software; most other devices need a subscription edition of the Xilinx ISE software.

Altera offers three main classes of FPGA devices: The Stratix family if the high performance devices; the Arria family has the midrange devices and the Cyclone devices are the devices with lowest cost, lowest power, lowest density, and lowest performance of all three. Logic block size of recent device has been increased from 4 input LUT to maximum 8 different inputs, that allows for instance to build 3 input adders at almost the same speed as

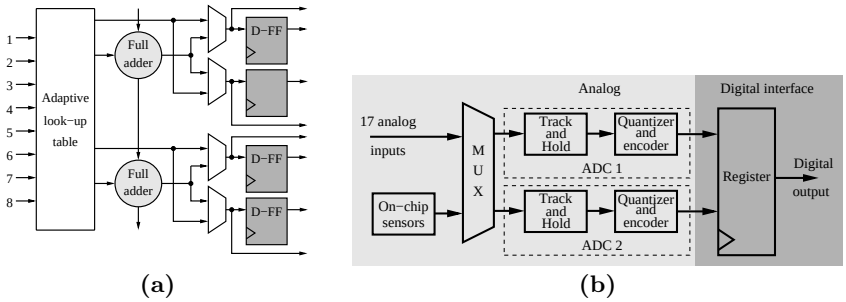


Fig. 1.7. New architecture features used in recent FPGA families. (a) Altera's ALM block. (b) Xilinx series 7 high speed on-chip ADC.

two input adders. Physically the ALM has two flip-flops, two full adders, two 4-input LUT and four 3-input LUTs and many multiplexer that allow a general 6 input function to be implemented, see Fig. 1.7a. The embedded multiplier size in Altera devices ranges from 9×9 bit, 18×18 bits, to 27×27 bits. Larger multiplier can be build by grouping this blocks together at the cost of a reduced speed. Starting with 5th generation three 9 bits blocks are combined to one fast 27×27 bits multiplier. The memory are available in a wide variety range from 0.5K, M4K, M9K, M10K, M144K bits to M512K bit memory. Keep in mind that only the Cyclone family is available in the web 12.1 edition of the Quartus II development software; Arria and Stratix device design need a subscription edition of the software.

Power dissipation of FPL is another important characteristic in particular in mobil applications. It has been found 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. 29.

1.3 DSP Technology Requirements

The PLD market share, by vendor, is presented in Fig. 1.8. 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. Actel became in November 2010 part of Microsemi Inc. 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

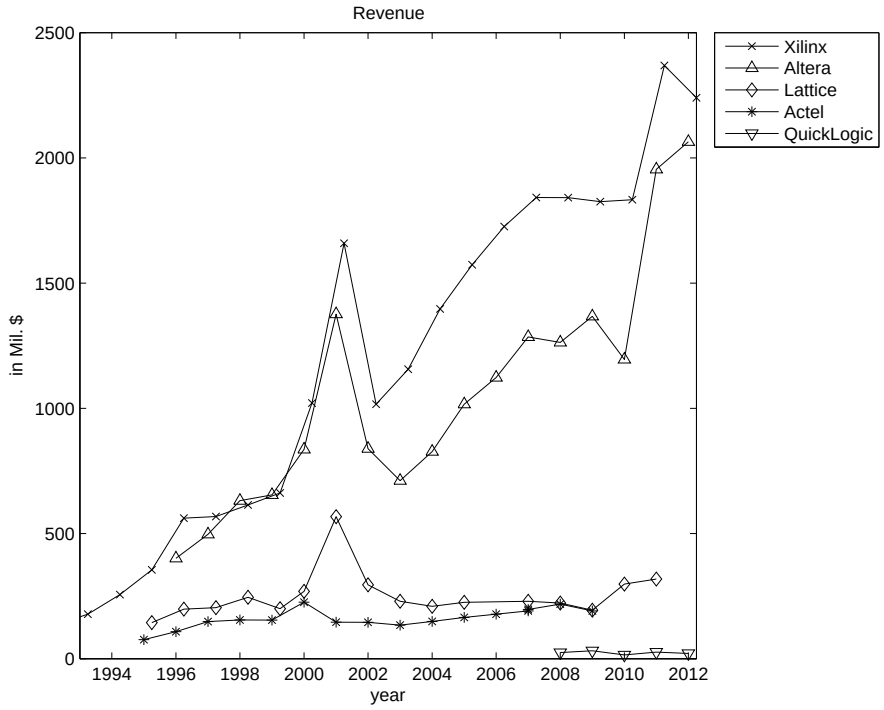


Fig. 1.8. Revenues of the top five vendors in the PLD/FPGA/CPLD market.

- Better security against unauthorized copies
- Reduced device and inventory cost
- Reduced board test costs

without many of the disadvantages of ASICs such as:

- 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

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 (Altera named HardCopy ASICs and Xilinx now EasyPath FPGAs), where a gate array is used to implement a verified FPGA design.

Table 1.6. Floating-point multiply-accumulate performance comparison of Stratix IV and TI PDSP.

Feature	Stratix IV EP4SGX230	TI TMS320C6727B-350
F_{rmmax}	227 MHz	350 MHz
# Devices	1	62
Total GFPMACS	43.5	43.4
Total Cost (1K units)	\$871	\$1,799 (62 × \$29.03)

1.3.1 FPGA and Programmable Signal Processors

General-purpose programmable digital signal processors (PDSPs) [6, 16, 17] 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 [18], 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 algorithms 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 [19] 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.

FPGAs have outperformed fixed-point PDSP in cost and power since many years now. In recent years FPGAs also have improved in the floating-point performance and in particular from a board size, power and cost standpoint offer now an attractive alternative to a floating-point PDSP solution. For a typical design the Table 1.6 list some key factors. Clearly, on such a large scale design FPGAs offer substantial improvement compared to a classical floating-point PDSP design. In the example shown in Table 1.6 the same giga floating-point multiply-accumulate operations per second (GFPMACS) are designed using an FPGA and PDSP. We would need 62 floating-point PDSP for the task that can be accomplished with a single FPGA used for the DE4 University boards (DE4 boards are available for ca. \$1000 through

Table 1.7. 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

the University program; $3K is the commercial DE4 price). A single FPMAC requires about 475 ALMs and 4 embedded 9x9 bit multipliers for the Stratix-IV family [20]. The fastest FP PDSP from TI is the TI320C6727B-350 which is a low-cost high speed floating-point PDSP that runs at 350 MHz and delivers 700 MMACs [21]. We will also observe an substantial board size and power dissipation improvement. The overall device cost is improved by over 100%.

We will discuss floating-point design via IP cores and using the new VHDL 2008 standard (and compatible with the VHLD-1993 via a synthesizable library provided by David Bishop) in section 2.6 in more details.

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 1.7 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 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. 1.9. 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 (a.k.a. RTL level simulation) of the circuit and to build an example data set called a testbench for later testing of the design with timing information. The functional netlist also allows a RTL view of the circuit that gives a quick overview of the circuit describe in HDL. If the RTL view verification or functional simulation 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 (a.k.a. gate level simulation) as well as performance measurements. Some synthesis tools also offer a technology map view of the circuit that shows how the HDL elements are mapped to LUTs, memory and embedded multipliers. 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 `--`):

[4] The equivalent Verilog code `example.v` for this example can be found in Appendix A on page 797. Synthesis results are shown in Appendix B on page 882.

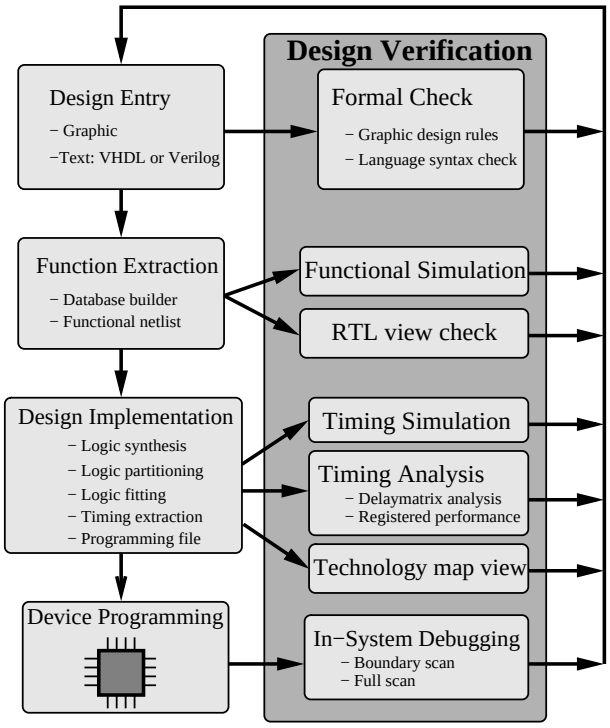


Fig. 1.9. CAD design circle.

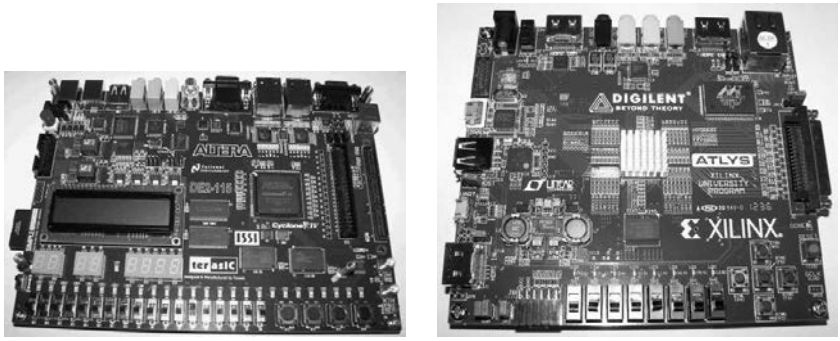
```
PACKAGE n_bit_int IS    -- User defined type
  SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
END n_bit_int;
LIBRARY work; USE work.n_bit_int.ALL;

LIBRARY ieee;                  -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-- ------------------------------------------------------
ENTITY example IS                          ------> Interface
  GENERIC (WIDTH : INTEGER := 8);   -- Bit width
  PORT (clk   :  IN STD_LOGIC;    -- System clock
        reset : IN  STD_LOGIC;    -- Asynchronous reset
        a, b, op1  : IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
                                  -- SLV type inputs
        sum   :  OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
                                  -- SLV type output
        c, d  :  OUT S8);         -- Integer output
END example;
-- ------------------------------------------------------
ARCHITECTURE fpga OF example IS
  COMPONENT lib_add_sub
```

```
    GENERIC  (LPM_WIDTH : INTEGER;
              LPM_DIRECTION :  string := "ADD");
  PORT(dataa : IN  STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
       datab : IN  STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
       result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
  END COMPONENT;

  COMPONENT lib_ff
  GENERIC (LPM_WIDTH : INTEGER);
  PORT (clock : IN  STD_LOGIC;
        data  : IN  STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
        q     : OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
  END COMPONENT;

  SIGNAL  a_i, b_i   :  S8 := 0;    -- Auxiliary signals
  SIGNAL  op2, op3   :  STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
BEGIN

  -- Conversion int -> logic vector
  op2 <= b;

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

  c <= a_i + b_i;      ------> Data flow style (concurrent)
  a_i <= CONV_INTEGER(a); -- Order of statement does not
  b_i <= CONV_INTEGER(b); -- matter in concurrent code

  P1: PROCESS(clk, reset) ----> Behavioral/sequential style
  VARIABLE  s :  S8 := 0;    -- Auxiliary variable
  BEGIN
    IF reset = '1' THEN               -- Asynchronous clear
      s := 0; d <= 0;
    ELSIF rising_edge(clk) THEN -- pos. edge triggered FFs
      s := s + a_i;       ----> Sequential statement
      -- d <= s;          -- "d" at this line: b_i would
      s := s + b_i;       -- not be added to output.
      d <= s;             -- Ordering of statements matters
    END IF;
  END PROCESS P1;

END fpga;
```

All HDL code start with the definition of I/O ports followed by the internal nets. In VHDL the used library need to be specified before the port declara-

tions. Then the actual circuit description starts. The example code shows sort coding examples of all three styles: components, concurrent and sequential. The `lpm` blocks are components from a component library that are instantiated similar to a graphic design. We need to specify the major parameters and connect the ports of the components to the internal nets or I/O port of our designs. Next shown is a short sequence of concurrent code. Here the type `SIGNAL` must be used that describe unique nets that only can be used once. However, the order of these statements does not matter. The type conversion does not need to be placed before the arithmetic operation as is necessary in sequential coding (i.e., `C`-code). Finally, a `PROCESS` examples shows some sequential coding style. Here we can also use local type `VARIABLE` that only can be used within the `PROCESS`. This `VARIABLE` do not need to be unique nets in the circuit and can be used (e.g., `VARIABLE s`) multiple times. The ordering of the statements within the `PROCESS` does matter just in a usual sequential program code. Only the statement up to the assignment are evaluated. For instance, if the assignment to `d` is not at the end of the `IF` statement then `b_i` would not be adder to `d`. Another point to notice is the strict typing of the VHDL language. The conversion between `INTEGER` and `STD_LOGIC` needs to be done via a function call. 1.1

A detail study how to synthesize and simulate a circuit (according to our flow from Fig. 1.9) will follow in section 1.4.3. At the end of the CAD tool flow we will have a programming file ready that can be downloaded to a hardware board (like the prototype board shown in Fig. 1.10) We proceed with programming the device and may perform additional hardware tests using the read-back methods. 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 buttons. These development boards are available from Altera directly. Altera offers Stratix and Cyclone boards, in the $199-$24,995 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 low-cost DE2-115 Cyclone IV board, which is still more expensive than the UP2 or UP3 boards used in many digital logic labs, but has a two-channel CODEC, large memory bank outside the FPGA, and many other useful ports (USB, VGA, PS/2 Ethernet,7-segment LEDs, LCD, switches, bush buttons, etc.), see Fig. 1.10a. 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 and audio CODEC) is the `Atlys` board offered by Digilent Inc. for only $199 or $349, for academic and regular pricing, respectively, see Fig. 1.10b. The board has a Spartan-6 XC6SLX45 FPGA, 16 MByte flash, 128 MByte DDR, 8 LEDs, eight switches, and five push buttons. For DSP and video experiments, we can take advantage of the A/D and D/A in the AC-97 audio codec and the 4 HDMI ports.



(a) **(b)**

Fig. 1.10. Low-cost prototype boards: **(a)** Cyclone IV DE2-115 Altera board. **(b)** Xilinx Atlys with on-board ADC and DAC CODEC.

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, 22, 23], 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-IV has 16 LEs per LAB.

Since the Spartan-6 device XC6SLX45 is part of a popular Atlys DSP board offered by Digilent Inc., see Figure 1.10b, we will have a closer look at this FPGA family. The basic logic elements of the Xilinx Spartan-6 are called slices and come in three different versions: M, L and X. In the Spartan-6 family two slices are combined in a configurable logic blocks (CLB), having a total of eight six-input one-output LUTs (or sixteen 5-input LUTs), and 16 flip-flops, 256 bit distributed RAM, or 128 bit shift register. 25% of all slices are type M and have all these features. 25% slices are of type L and do not

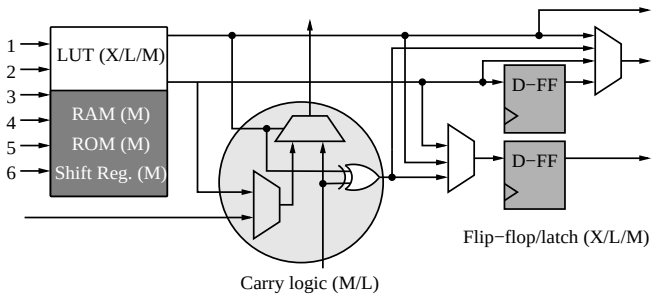


Fig. 1.11. Spartan-6 a quarter portion of slice. The X slice only has LUT and 2 flip-flops. The L slice add the fast carry logic. The M slice has all features. (©Xilinx).

have the memory shift register function. 50% of the slices are type X and these slices do not have shift register option, arithmetic carry and wide multiplexer. Figure 1.11 shows one quarter of a slice and the features for each type. Each LUT in the M type slice can be used as a 64×1 RAM or ROM. The Xilinx device has multiple levels of routing, ranging from CLB to CLB, to long lines spanning the entire chip. The Spartan-6 device also includes large memory blocks (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-6 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 (or phases) in the same FPGA with low clock skew. Up to 33 Mbits configuration files size is required to program Spartan-6 devices. Tables 1.8 shows the most important DSP features of members of the Xilinx Spartan-6 family.

As an example of an Altera FPGA family let us have a look at the Cyclone IV E device EP4CE115 used in the low-cost prototyping board DE2-115 by Altera, see Fig. 1.10a. The basic block of the Altera Cyclone IV device achieves a medium granularity using small LUTs. The Cyclone device is similar to the Altera 10K device used in the mature UP2 and UP3 boards, with increased RAM blocks memory size to 9 kbits, which are no longer called EAB as in Flex 10K or ESB as in the APEX family, but rather M9K memory blocks, which better reflects their memory size. The basic logic element in Altera FPGAs is called a logic element (LE)[5] 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. 1.12. Each LE can be

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

Table 1.8. The Xilinx Spartan-6 family.

Device	Total 5-input LUTs	Slices	RAM blocks	CMT 2 DCM 1 PLL	Emb. mult. 18×18	Conf. file mbit
XC6SLX4	4800	600	12	2	8	2.7
XC6SLX9	11440	1430	32	2	16	2.7
XC6SLX16	18224	2278	32	2	32	3.7
XC6SLX25	30 064	3758	52	2	38	6.4
XC6SLX45	54 576	**6822**	**116**	**4**	**58**	**11.9**
XC6SLX75	93 296	11662	172	6	132	19.7
XC6SLX100	126 576	15822	180	6	180	26.7
XC6SLX150	184 304	23038	180	6	180	33.9

Table 1.9. Altera's Cyclone IV E device family.

Device	Total 4-input LUTs	RAM blocks M9K	PLLs/ clock networks	Emb. mul. 18×18	Max. I/O	Conf. file Mbits
EP4CE6	6272	30	2/10	15	179	2.94
EP4CE10	10 320	46	2/10	23	179	2.94
EP4CE15	15 408	56	4/20	56	343	4.09
EP4CE22	22 320	66	4/20	66	153	5.75
EP4CE30	28 848	66	4/20	66	532	9.53
EP4CE40	39 600	126	4/20	116	532	9.53
EP4CE55	55 856	260	4/20	154	374	14.89
EP4CE75	75 408	305	4/20	200	426	19.97
EP4CE115	**114 480**	**432**	**4/20**	**266**	**528**	**28.57**

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 IV devices. Each device contains at least one column with embedded 18×18 bit multipliers and one column M9K memory blocks. One 18×18 bit multiplier can also be used as two signed 9×9 bit multipliers. The M9K memory can be configured as $2^8 \times 32$, $2^9 \times 16, \ldots, 8192 \times 1$ RAM or ROM. In addition one parity bit per byte is available (e.g., 256×36 configuration), which can be used for data integrity. These M9Ks and LABs are connected through wide high-speed busses as shown in Fig. 1.13. Several PLLs are in use to produce multiple clock domains with low clock skew in the same device. At least 29 Mbits configuration files size is required to program the EP4CE115. Table 1.9 shows the members of the Altera Cyclone IV E 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

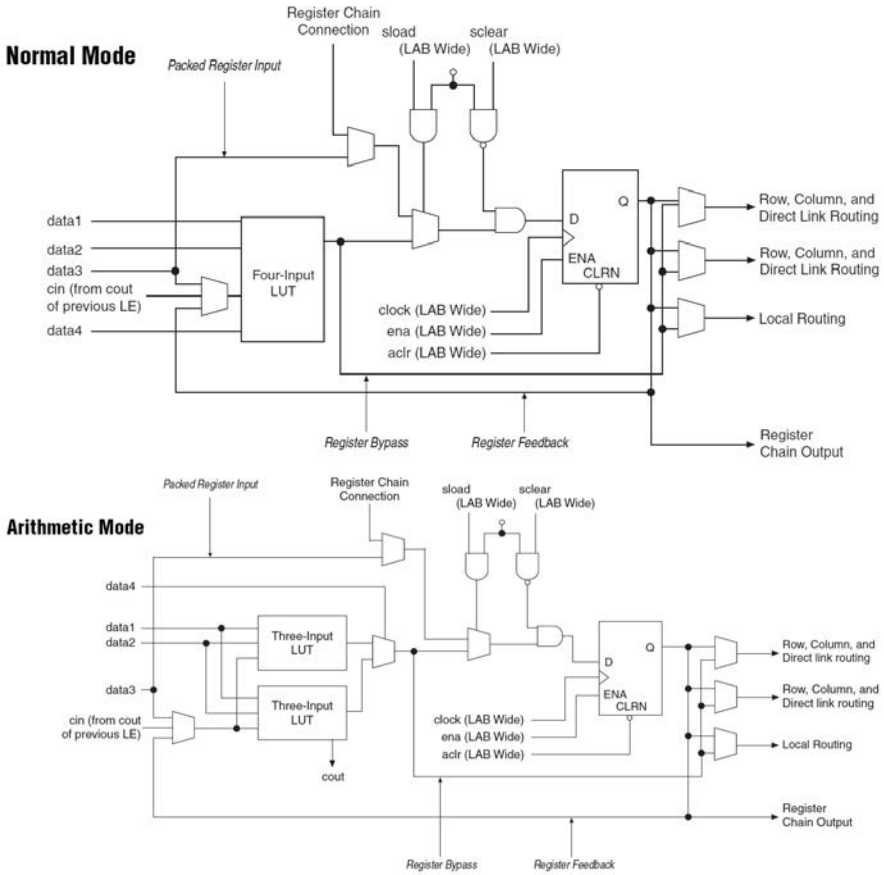


Fig. 1.12. Cyclone IV logic cell (©Altera [24]).

signal processing algorithms process the data locally. The Altera approach, with wide busses, also has value, because typically not only are single bits 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 EP4CE115F29C7N

The Altera EP4CE115F29C7N device, a member of the Cyclone IV E family, which is part of the DSP prototype board DE2-115 provided through Altera's university program, is used throughout this book. The device nomenclature is interpreted as follows:

```
EP4CE115F29C7N
 |    | | ||||-> Lead-free (N) or low-voltage (L)
```

```
|    | | |||--> Speed grade 7 (lower is faster)
|    | | ||---> Commercial temperature 0-85 Celsius
|    | | |----> Package with 780 pins
|    | |------> Package FineLine BGA
|    |--------> LEs in 1000
|-------------> Device family: Cyclone IV E
```

Specific design examples will, wherever possible, target the Cyclone IV device EP4CE115F29C7N using Altera-supplied software. The web based Quartus II software is a fully integrated system with VHDL and Verilog editor, synthesizer, timing evaluation and bitstream generator that can be download free of charge from `www.altera.com`. 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 version then 12.1 or simulator may also be used. For instance, the Xilinx ISE compiler and the `ISIM` simulator has successfully been used to compile the examples. For other Quartus II software versions the included `qvhdl.tcl` and `qv.tcl` can be used to compile all examples for a new software version using just one script.

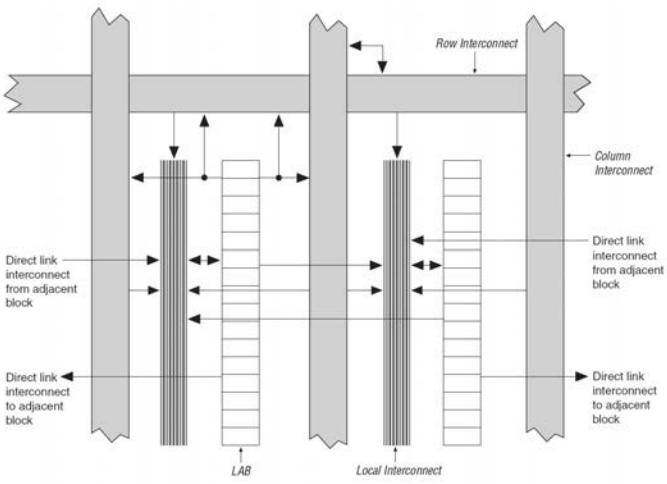


Fig. 1.13. Cyclone IV logic array block resources. (©Altera [24]).

Logic Resources

The EP4CE115 is a member of the Altera Cyclone IV family and has a logic density equivalent to about 115 000 logic elements (LEs). An additional 266 multipliers of size 18×18 bits (or twice this number if a size of 9×9 bit is used) are available, see Fig. 1.14. From Table 1.9 it can be seen that the

EP4CE115 device has 114 480 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. 1.12. Sixteen LEs are always combined into a logic array block (LAB), see Fig. 1.13. The number of LABs is therefore $114,480/16 = 7155$. In the left medium area of the device the JTAG interface is placed and uses the area of typically $5 \times 9 = 45$ LABs. This is why the total number of LABs in not just the product of rows $\times$ column. The device also includes six columns of 9-kbit memory block (called M9K memory blocks) that have the height of one LAB and the total number of M9Ks is therefore $6 \times 72 = 432$. The M9Ks can be configured as 256×36, 256×32, 512×18, 512×16, $\ldots 8192 \times 1$ RAM or ROM, where for each byte one parity bit is available. The EP4CE115 also has 4 columns of 18×18 bit fast array multipliers, that can also be configured as two 9×9 bit multipliers. Lower left corner of the EP4CE115 is shown in Fig. 1.19 (p. 36) along with the `bird eye view` that shows the overall device floorplan.

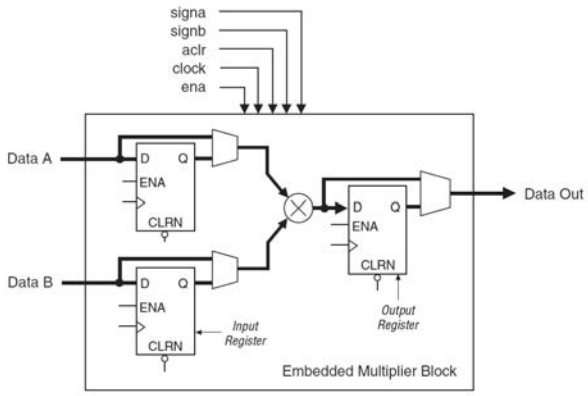


Fig. 1.14. Embedded multiplier architecture (©Altera [24]).

Additional Resources and Routing

While the exact number of horizontal and vertical bus lines for the EP4CE115 in no longer specified in the data sheet as in the older (e.g., FLEX10K) families [25] from the Compiler Report we can determine to over all routing resources of the device, see `Fitter`→`Resource Section`→`Logic and Routing Section`. Local connection are provided by block interconnects and direct links. Each LE of the Cyclone IV devices can drive up to 48 LEs through fast local and direct link interconnects. The next level of routing are the R4 and C4 fast row and column local connections that allow wires to reach LABs

Table 1.10. Routing resources for the Altera's Cyclone IV E device EP4CE115.

Block	Direct	C4	R4	C16	R24	Clock
342 891	342 891	209 544	289 782	10 120	9963	20

at a distance of ±4 LABs, or 3 LABs and one embedded multiplier or M9K memory block. The longest connection for logic available are R24 and C16 wires that allows 24 rows or 16 column LAB, respectively. 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. The global clocks network span the entire FPGA. Table 1.10 give an overview of the available routing resources in the EP4CE115.

The datasheets provide more detail about the available internal LAB control signals and global clocks. The EP4CE115 has 4 PLLs and total of 20 clock networks spanning the entire chip to guaranty short clock skew. Each PLL has 5 output counters that can be used to generated different frequencies or phase offsets. The DRAM on the DE2 boards for instance require a clock with 0 and -3 ns offset that we will generated with one PLL.

All 16 LEs in a LAB share the same synchronous clear and load signals. Two asynchronous clear, two clocks and two enable signals are shared by the LEs in the LAB. This will limit the freedom of the control signals within one LAB, but since DSP signals are usually processed in wide bus signals this should not be a problem for our designs.

The LAB local interconnect is driven by row or columns bus signals or LEs in the LAB. Neighboring LABs, PLLs, BRAM, embedded multipliers from left or right can also drive the local LAB. 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. 1.19, p. 36.

Timing Estimates

Altera's Quartus II and the Xilinx ISE software in the past had only two timing optimization goals: Area or Speed. However, with increased device density and system-on-chip designs with multiple clock domains, different I/O clock modes, clocks with multiplexer and clock dividers today this strategy optimizing area or speed for the complete device may not be a good approach anymore. Altera now offer a more sophisticated timing specification that looks more like a cell-based ASIC design style and is based on the Synopsis Design Constrain (SDC) files. The idea here is that a synthesis tool may have for the same circuit different library elements such a ripple carry, carry save or fast look-ahead styles for an adder. The tool then first optimizes the design to

meet the specified timing constrains and in a second step then optimizes area. If you like similar synthesis results as with previous tool such as minimum area or maximum speed you can achieve this by over or under constraining the design. For instance, if you do not specify a SDC file then a 1 GHz target performance is assumed, what is most likely over constraining of your design and device. If you like only to optimize the area, than you should use a very low target clock rate, since then all effort is put in the area optimization. Since the flow based on SDC specification has been recently introduced it maybe a good idea to go through some tutorials. The Altera University program offers a introduction in the "Using TimeQuest Timing Analyzer" tutorial. More details are available in the "Quartus II TimeQuest Timing Analyzer Cookbook" with many different multi clock domain examples.

Since in our design examples we are only interested in optimization for speed we can use the default setting (i.e., without a SDC file) that over constrains the design to run at 1 GHz desired clock rate. We should expect to see a Warning message in the compilation report that timing has not been met but that is intended and should not be a concern.

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 66 ps for Cyclone IV speed grade 7. An upper bound for the maximum performance would then be $32 \times 66\,\text{ps} = 2.112\,\text{ns}$ or 473 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 0.501 ns. Next the first carry t_{cgen} must be generated, requiring about 0.414 ns. Finally at the end of the carry chain the full sum bit needs to be computed (about 536 ps) and the setup time for the output register (87 ps) needs to be taken into account. The results are then stored in the LE register. The following table summarizes these data path timing delays:

LE register clock-to-output delay	t_{co}	=	232 ps
Interconnect delay	t_{ic}	=	501 ps
Data-in to carry-out delay	t_{cgen}	=	414 ps
Carry-in to carry-out delay	$30 \times t_{\text{cico}}$	=	$30 \times 66\,ps$ = 1980 ps
LE look-up table delay	t_{LUT}	=	536 ps
LE register setup time	t_{su}	=	87 ps
Total		=	3,802 ps

For possible clock skew an additional 82 ps should be added and the total estimated delay is 3.884 ns, or a rate of 257.47 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. 48). The `TimeQuest Analyzer`

in Fig. 1.15 shows the full timing path including the clock delays. Since we did not specify the desired clock period `TimeQuest` uses the default 1 ns or 1 GHz value, which is in most cases an over constrain and the `Slack` (see row 7 in Fig. 1.15a) becomes negative shown in red color, i.e., as a violation. To meet timing we would usually take the default 1 ns clock and add the (absolute) negative slack. If we specify a relaxed timing using a `Synopsys Design Constrain` file to 200 MHz (i.e., clock period 5 ns) then the analysis will show that the timing is meet by using a positive slack that is displayed in green color, see 1.15b. 1.2

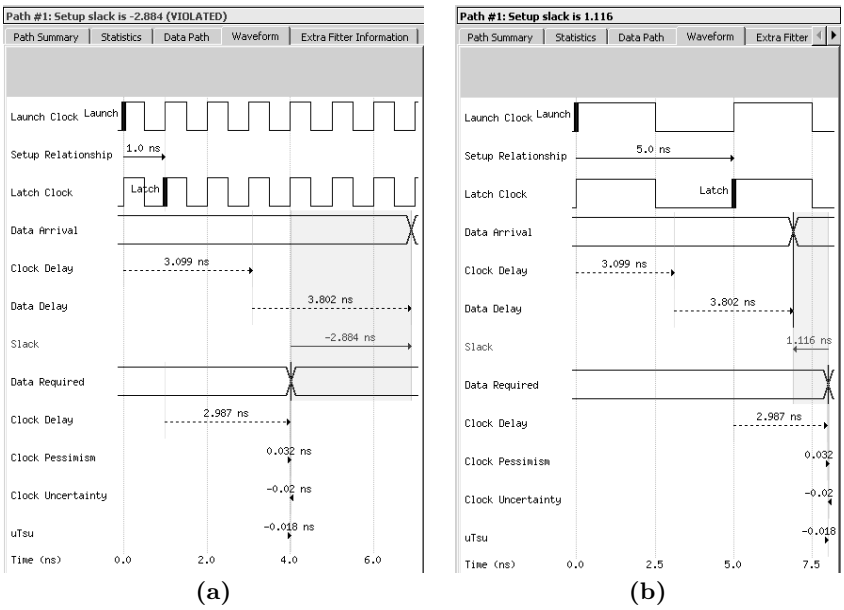


Fig. 1.15. `Time Quest` analysis **(a)** Negative slack shows a timing violation. **(b)** Positive slack indicating that timing was meet.

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. The Quartus II `TimeQuest Analyzer` reports for instance propagation delay of 11.3 ns for a direct FPGA connection from input `SW[1]` to output pin `HEX0[1]` on the `DE2_115` board. The delays from I/O pins are much larger than the registered performance `Fmax` when data comes directly from the register next to the design under test. Datasheets [24, Vol. 3] 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 M9K (but not the adder)

Table 1.11. Some typical registered performance `Fmax` and resource data for the Cyclone IV C7 Speed grade.

Design	LE	M9K memory blocks	Multiplier blocks 9×9 bit	`Registered Performance` MHz
16 bit adder	16(+32)	–	–	363
32 bit adder	32(+64)	–	–	257
64 bit adder	64(+128)	–	–	169
FIFO $2^8 \times 36$	47	1	–	274
RAM $2^8 \times 36$	–	1	–	274
9×9 bit multiplier	–	–	1	300
18×18 bit multiplier	–	–	2	250

have additional I/O registers to enable maximum speed, see Fig. 1.14. 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 1.11 reports some typical data measured under these assumptions. If we compare measured data with the delay given in the data book [24, Vol. 3] we notice that for some blocks the `TimeQuest Analyzer` 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.

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 IV family for instance is produced in a 2.5V transistor, 60-nm low-k dielectric process from the Taiwan ASIC foundry TSMC, but I/O interface voltages of 3.3 V, 2.5V, 1.8V, 1.5V and 1.2V are also supported. To estimate the power dissipation of the Altera device EP4CE115, two main sources must be considered, namely:

1) Static power dissipation, $P_{\text{static}} \approx 135\,\text{mW}$ for the `EP4CE115F29C7N` using typical industrial temperature grade
2) Dynamic (logic, multiplier, RAM, PLL, clocks, I/O) power dissipation, I_{active}

The first parameter (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.

Table 1.12. Power consumption estimation for the Cyclone IV EP4CE115F29C7N.

Parameter	Units	Toggle rate (%)	Power mW
P_{static}			159
LEs 114 480 @ 100 MHz	114480	12.5%	1170
M9K block memory	432	50%	74
9×9 bit multiplier	532	12.5%	153
I/O cells (2.5V, 4 mA)	528	12.5%	164
PLL	4		32
Clock network	115706		486
Total			2238

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 1.12 shows the results for power estimation when all resource of the EP4CE115F29C7N 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 $159 + 2079/10 = 366.9\,\text{mW}$, and the static power consumption would now be account for 43%.

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. 1.16. 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" or "Value Change Dump" file that can be selected as the input file for the `Analyzer`. The `PowerPlay Power Analyzer` tool allows different options we can select. We can just specify a clock rate via the `TimeQuest Timing`

`Analyzer` SDC file and set a default toggle rate, without a simulator help. More accuracy should be expect if we use a functional or RTL simulation as input for the power estimation that only require one full compilation of the design. The best estimation should be achieved if we use the compiled design in the ModelSim simulator input that also includes precise LE switching, glitches, bus driver data etc. Table 1.13 shows a comparison between the power estimation and the 3 power analysis.

Table 1.13. Power consumption for the design shown in Fig. 1.16 for a Cyclone IV EP4CE115F29C7 at 50 MHz using SDC TimeQuest file and Excel-based PowerPlay Early Power Estimator (PPEPE) or Quartus-based PowerPlay Power Analyzer (PPPA). The estimated toggle percentage is 12.5%

	PPEPE Estimation	PPPA Estimation	PPPA RTL Sim.	PPPA Timing
VCD needed	–	–	✓	✓
Parameter	power/mW	power/mW	power/mW	power/mW
Static	135	98.4	98.4	98.5
Dynamic	2	2.3	2.6	3.7
I/O	4	38.9	38.9	50.4
Total	141	139.6	139.9	152.6

We notice a discrepancy of 10% between early estimation and analysis using timing information. The analysis however requires a complete design including a reliable 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. 1.16). 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. The equivalent HDL text file `fun_text.v` and `fun_text.vhd` implement the design using behavioral HDL code, thus avoiding LPM component instantiations. The challenge of the design is to have behavioral HDL code that is synthesized by Altera Quartus II and Xilinx ISE software and work well with the recommended simulators (ModelSim for Altera and `ISIM` for Xilinx) for both, RTL and timing simulation. 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

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 `EP4CE115F29C7` from the Cyclone IV E family (5. from bottom in device listing in Quartus 12.1). Click `Next` and the select ModelSim-Altera as Simulation tool 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[6] reads as follows:

```
--  A 32 bit function generator using accumulator and ROM
LIBRARY ieee;
USE ieee.STD_LOGIC_1164.ALL;
USE ieee.STD_LOGIC_arith.ALL;
USE ieee.STD_LOGIC_signed.ALL;
```

[6] The equivalent Verilog code `fun_text.v` for this example can be found in Appendix A on page 798. Synthesis results are shown in Appendix B on page 882.

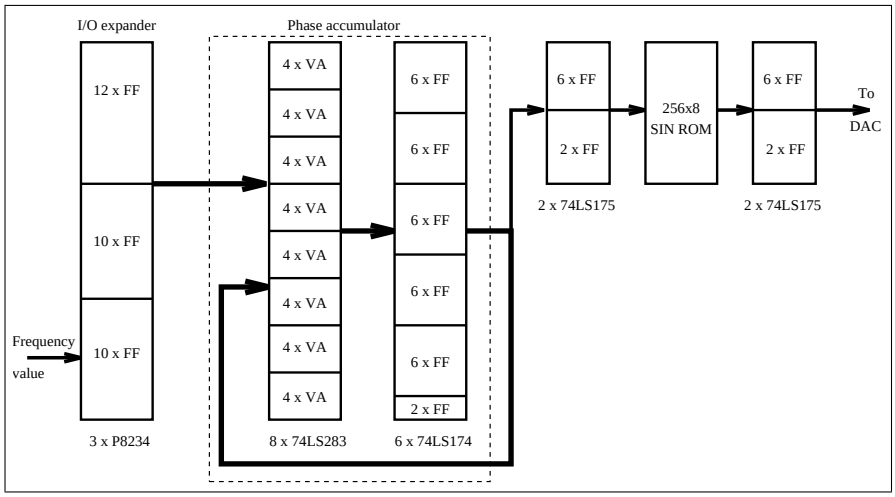
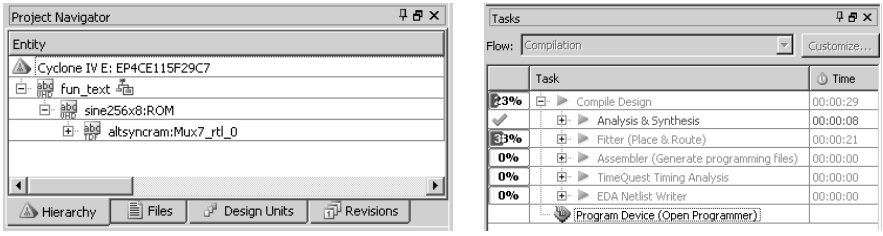


Fig. 1.16. PM5190 frequency synthesizer.

```
-- ------------------------------------------------------
ENTITY fun_text IS
  GENERIC ( WIDTH   : INTEGER := 32);    -- Bit width
  PORT (clk   : IN  STD_LOGIC; -- System clock
        reset : IN  STD_LOGIC; -- Asynchronous reset
        M     : IN  STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
                                  -- Accumulator increment
        acc   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                                       -- Accumulator MSBs
        sin   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END fun_text;                           -- System sine output
-- ------------------------------------------------------
ARCHITECTURE fpga OF fun_text IS

  COMPONENT sine256x8
    PORT (clk : IN STD_LOGIC;
          addr : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
          data : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
  END COMPONENT;

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

  PROCESS (reset, clk, acc32)
  BEGIN
    IF reset = '1' THEN
      acc32 <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      acc32 <= acc32 + M; -- Add M to acc32 and
    END IF;               -- store in register

  END PROCESS;

  msbs <= acc32(31 DOWNTO 24); -- Select MSBs
  acc <= msbs;

  -- Instantiate the ROM
  ROM: sine256x8 PORT MAP
                  (clk => clk, addr => msbs, data => 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. Next in the coding comes the component in use and additional SIGNAL definitions. The HDL coding starts after the key word BEGIN. The first PROCESS includes the 32-bit accumulator, i.e., an adder followed by a register. The accumulator has an asynchronous active high reset. The next two statements connect the local signal to I/O ports. Finally the ROM table is instantiated as a component and the port signals of the component are connected to the local signal within our design. You may want to have a look at the ROM table design file sine256x8.vhd. You can load the file or you can double click it in the Project Navigator window (top left), see Fig. 1.17a. In general a synthesizable ROM or RAM design with initial data loaded in the memory is not a trivial task. A good stating point is either to have a look in the VHDL 1076.6-2004 subset of VHDL for synthesizable code or a little easier is to have a look at the language templates suggest by the tool vendors (Altera: Edit→ Insert Template→ VHDL→ Full Designs→ RAMs and ROMs→ Dual-Port ROM. or Xilinx: Edit→ Language Template→ VHDL→ Synthesis Constructs→ Coding Examples→ ROM→ Example Code). Altera recommend to use a function call for the initialization, and Xilinx a CONSTANT array initial definition. It turn out that the latter approach work well with both tools as well with the simulators for functional and timing simulator and will be the preferred method for the rest of the book. The synthesis attributes as defined in VHDL 1076.6-2004 are currently not supported by either vendors. In VERILOG RAM and ROM initialization is already specified in the language reference manual (LRM) and the most reliable method is to use the $readmemh() function in combination with an initial statement.

To optimize the design for speed, go to Assignment → Settings → Analysis & Synthesis Settings. Under Optimization Technique click on Speed. Timing requirement can be set using SDC file. The default speed is set to 1 ns that usually works well with the Speed optimization. Now start the compiler tool (it's the thick right arrow symbol) that can be found under the Processing menu. A window left to our HDL window as shown in Fig. 1.17b will show the progress in the compilation. You can see all steps envolved in the compilation, namely: Analysis & Synthesis, Fitter, Assembler Timing Analysis, Netlist Writer and Program Device. Alternative you can just start the Analysis & Synthesis by clicking on Processing → Start→ Start Analysis & Synthesis or with <Ctrl+K>. 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 or by pressing <Ctrl+L>. If all compiler steps were successfully completed, the design is fully implemented. Press the Compilation Report button (IC symbol with a paper) from the top menu buttons and the flow summary report should show 32 LEs and



(a) **(b)**

Fig. 1.17. (a) Project Navigator. **(b)** Compilation steps in Quartus II.

2048 memory bits use. Check the memory initialization file `sine256x8.vhd` for VHDL and `sine256x8.txt` for Verilog. These files were generated using the program `sine.exe` included on the CD-ROM under `util`. Figure 1.17 summarizes all the processing steps of the compilation, as shown in the Quartus II compiler window.

For a *graphical* verification that the HDL design describes the desired circuit, using Altera's Quartus II software, we can use the RTL viewer. The results for the `fun_text.vhd` circuit is shown in Fig. 1.18. To start the RTL viewer click on `Tools` →`Netlist Viewer` →`RTL Viewer`. The other netlist viewer called `Technology Map Viewer` give a precise picture how the circuit is mapped onto the FPGA resources. However, to verify the HDL code even for a small design as the function generator the technology map provides too much details to be helpful, and we will not use it in our design studies.

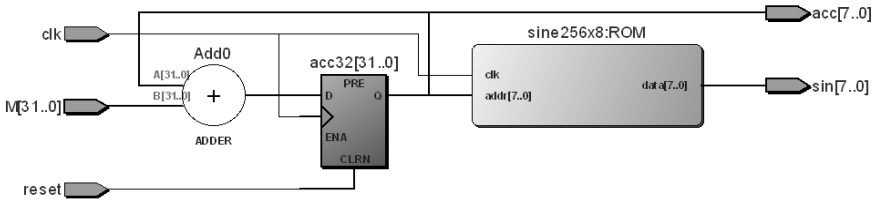


Fig. 1.18. RTL view of the frequency synthesizer.

Floor Planing

The design results can be verified by clicking on the 6. button (i.e., `Chip Planner` or opening the `Tool`→`Chip Planner`) to get a more-detailed view of the chip layout. The `Chip Planner` view is shown in Fig. 1.19. Use the `Zoom in` button (i.e., the ± magnifying glass) to produce the screen shown in Fig. 1.19. Zoom in to the area where the LAB and an M9K are highlighted in

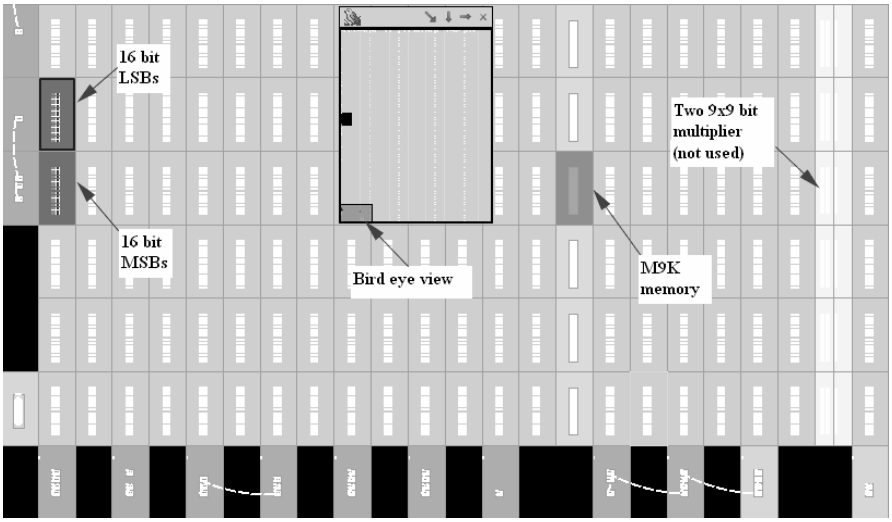


Fig. 1.19. Floorplan of the frequency synthesizer design.

a different color. You should then see the two LABs used by the accumulator highlighted in blue and the M9K block highlighted in green. In addition several I/O cell are also highlighted in brown. Click on the `Bird's Eye View` button[7] on the left menu buttons and an additional window will pop up. Now select for instance the M9K 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

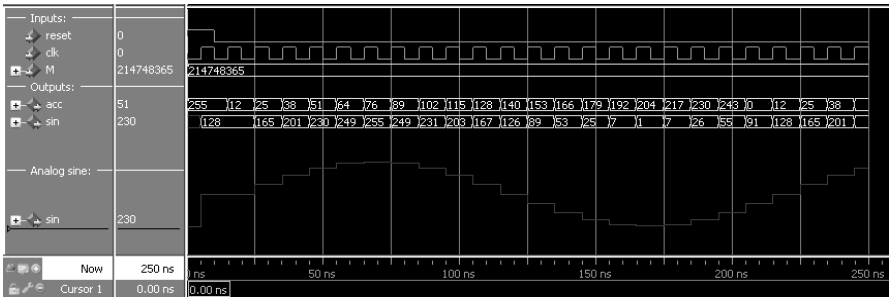


Fig. 1.20. ModelSim RTL simulation of frequency synthesizer design.

[7] 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.

In terms of simulator we have seen the two FPGA market leader taken diagonal directions in recent years. In the past Altera favored the internal `VWF` simulator (up to Quartus II version 9.1) and now recommends to use the external ModelSim-Altera or `Qsim`. Xilinx on the other side since version 12.3 (end of 2010) provides no long a free ModelSim simulator and instead provides a free embedded `ISIM` simulator that is integrated within the ISE tool.

When simulating a design such as the `ISIM` simulator we have the option to use a stimuli file from a `TCL` script similar to ModelSim `DO` files, or we can write a test bench in HDL. A test bench is a short HDL file, where we instantiate the circuit to be tested and then generate and apply our test signals with a statement like

```
clk <= NOT clk AFTER 5 ns;
```

to generate a clock with a $2 \times 5\,\text{ns} = 10\,\text{ns}$ clock period. However, the difficulty with the Xilinx test bench comes from the fact that the circuit with timing information (i.e., `*_timesim.vhd`) is synthesized directly from the netlist and the `STD_LOGIC` is used throughout the whole `ENTITY` description. The original `ENTITY` data types and `GENERIC` variables are ignored. If we like to use the same VHDL test bench for RTL and timing simulation then the `ENTITY` are restricted to a single data type. More precise, we can not use `INTEGER`, `SIGNED` or `FLOAT` data types, even `BUFFER` or `GENERIC` parameter would not be permitted. This would therefore highly interfere with the coding for design reuse and we would need to use a separate test bench for RTL and timing simulation. However, if we do not use a test bench and simulate our circuit directly using the TCL stimuli script then we can use the same script for RTL and timing simulation. Furthermore for VHDL and Verilog the same stimuli file can be used, only the compile sequence will be different. The `ISIM TCL` scripts and ModelSim `DO` files are also very similar in their coding style to simplify a transition between the two simulators.

The Altera Quartus II software comes with two free simulator options. The ModelSim-Altera allow us to use the professional tool from Mentor Graphic Inc. The second alterative is the Altera `Qsim` tool that may have a few less feature than ModelSim (e.g., no analog waveform) but is also a little easier to handle since is does not require to write HDL testbenches or `DO` file scripts to assign I/O signals. However, at time of writing the `Qsim` in 12.1 does not support the Cyclone IV devices and we therefore select the ModelSim-Altera as default simulator. Moving between VHDL and Verilog stimuli file and Altera and Xilinx is also simplified when using ModelSim-Altera `DO` files and not test benches.

Before we discuss a simulation example let use first have a look at the file name used, where `*` stands for the project name. For the RTL simulation we use the `*.vhd` and `*.v` files. For the timing simulation four different files names are used: `*.vho` and `*.vo` for VHDL and Verilog using Altera tools

and `*_timesim.vhd` and `*_timesim.v` for Xilinx. For both RTL and timing simulation we use the same stimuli files `*.do` (Altera) and `*.tcl` (Xilinx).

To simulate, open the ModelSim-Altera Tool. You should see that many predefined libraries are already loaded. Use `File` → `Change Directory` to move to the directory that includes the HDL files and the simulation scripts. Use `dir *.do` and `dir *.vhd` to verify that the files are indeed in the current path. To run the script type `do fun_text.do 0` for RTL and `do fun_text.do 1` for timing simulation. For functional simulation you should get a result similar to Fig. 1.20. For timing simulation you need first to compile the design in Quartus II or ISE and then simulate the HDL code with timing information. The script compiles the files, open a waveform and simulate the circuit. The do script for this example looks as follows:

```
set project_name "fun_text"
do tb_ini.do $1 sine256x8

########## Add I/O signals to wave window
add wave -divider  "Inputs:"
add wave reset clk
radix -unsigned
add wave M
add wave -divider  "Outputs:"
add wave acc sin
add wave -divider -height 80 {Analog sine:}
add wave -color Red -format Analog-Step \
         -radix unsigned -scale 0.25 sin

######### Add stimuli data
force clk 0 0 ns, 1 5 ns -r 10 ns
force reset 1 0 ns, 0 10 ns
force M 214748365 0 ns

########## Run the simulation
run 250 ns
wave zoomfull
configure wave -gridperiod 5ns
configure wave -timelineunits ns
```

The simulation script has typically four parts.

1) First the project name is defined and all components in the project are compile via `vcom` or `vlog` for VHDL and Verilog, respectively. The top level project is compiled in a second brief script call to `tb_ini.do` with one parameter (0 for RTL and 1 for timing simulation). It also includes a function to add local signals. These local signals may not be available in the timing simulation.

2) Next signals are added to the wave window in the order they should appear in the wave window. We start with the input signals, follow by the outputs. Divider are used to help with differentiations or to add extra information. Note in particular the last entry that shows how a "analog" display of a signal can be defined.
3) Next follows in any order the stimuli data to the wave form signals we just had defined. We can define periodic signals as for the `clk` and non-periodic as for `reset` and `M`.
4) Last we run the simulation for a specific time and zoom to display the full time frame. A grid and time unit can also be defined for the wave window.

As can be seen from the script or the wave window the following data has been used: As the period 1/100 MHz = 10 ns was selected. We set $M = 214\,748\,364 (M = 2^{32}/20)$, so that the period of the synthesizer is twenty clock cycles long. Notice that the ROM has been coded in binary offset (i.e., zero = 128). This is a typical coding used in D/A converters. A short MatLab script or `C`-program can be used to generate the data. The CD includes a short `C`-program `sine.exe` that generate the VHDL code for the component as well as the table data for the Verilog code. For brevity hex display has been used but the program `sine.exe` also allows binary or octal.

Performance Analysis

In order to display timing data a full compile of the design has to be done first. For the Altera tool we use the default setting with period of 1 ns equivalent to 1 GHz clock frequency that will appear as `FMAX_REQUIREMENT "1 ns"` in Quartus II `QSF` files. Since our design most likely runs slower this will ensure the Quartus compiler is synthesizing the fasted design possible. On the downside we will always get a compiler warning that `Synopsys Design Constraints File 'fun_text.sdc'` was not found and a `Critical Warning: Timing requirements not met`, but we can ignore these messages.

The result of the `TimeQuest` analysis is provided in the `Compilation Report` shown in Fig. 1.21. It includes the 3 timing data for slow 85C, 0C and fast 0C model. We use the most pessimistic, i.e., the slow 85C model. The `Fmax Summary` frequency is the maximum frequency our circuit can run with. Sometime the performance is further restricted due to the maximum I/O pin speed of 250 MHz. If there is no register-to-register path in the circuit as in a pure combinational design this `Fmax` will come up empty with the message `No paths to report`.

The performance analysis with the Xilinx software is a little simpler since we have just the two settings for `Speed` and `Area`. In the Implementation view just right click the `Synthesize - XST` entry and under `Synthesis Options` select for the `Optimization Goal` the entry `Speed`. After a full

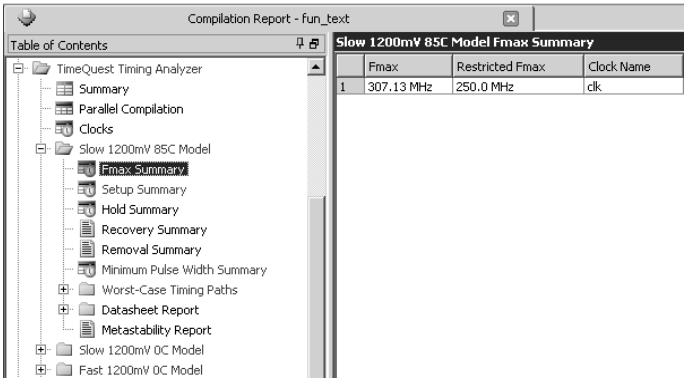


Fig. 1.21. Registered performance of frequency synthesizer design from the `TimeQuest Timing Analyzer`.

compile check the `Post-PAR Static Timing Report` and you will find the maximum clock as last timing entry in the report as `Clock to Setup on destination clock clk`.

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 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, I2C bus or ethernet interfaces) are available from third-party vendors. While some 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 [26, 27, 28]. 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 first generation 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.

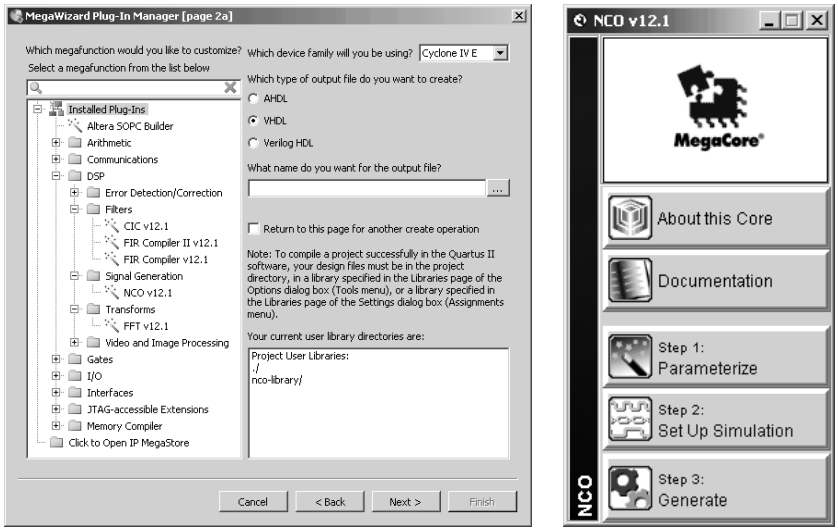
There are however two major challenges with current IP block technology, which are pricing of a block and, closely related, IP protection. Because the cores are reusable vendor pricing has to rely on the number of units of IP blocks the customer will use. This is a problem known for many years in patent 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 [28]. 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



(a) **(b)**

Fig. 1.22. 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 name the IP core as our project, i.e., `NCO`. Open a new project and name it `NCO` and press the button `Tools` → `MegaWizard Plug-In Manager`. In the first step select the NCO block in the `MegaWizard Plug-In Manager` window, see Fig. 1.22a. 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. 1.22b) 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. However, the smallest output bitwidth is 10 bits, and we will not be able to use the 8 bit as in our previous design. We use the `Large ROM` generation algorithm in the parameter window, see Fig. 1.23. The clock rate in the `fun_text` study was 100 MHz and the output period had 20 clock cycles or equivalent 5 MHz frequency. Phase dithering will make the noise more equally distributed, but will require more than twice as many LEs. With phase dithering we would get about 60 dB sidelope suppression, without dithering we get about 50 dB sidelope suppression, as can be seen in the `Frequency Domain`

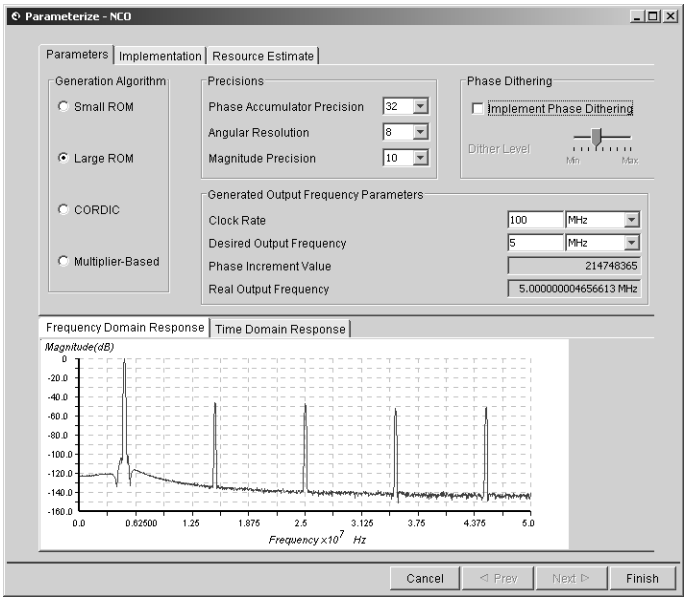


Fig. 1.23. IP parametrization of NCO core according to the data from the `fun_text` case study in the previous section.

`Response` plot in the lower part of the NCO window. 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, 2560 memory bits or one M9K block. After we are satisfied with our parameter selection we then proceed to step 2 to specify if we want 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 1.14 gives an overview of the generated files.

We see that not only are the VHDL 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 use the core directly as out top level design entry, therefore avoiding the need to instantiate our block in another design and connect the input and outputs. By inspecting the top level VHDL file `nco.vhd` we notice that the `ENTITY` has the expect block inputs `clk, phi_inc_i` and output `fsin_o` signals, but has some additional useful control signal, i.e., `reset_n`, `clken`, and `out_valid`, whose function is self-explanatory. We start then a full compilation to generate the `nco.vho` file for 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

Table 1.14. IP file generation for the NCO core.

File	Description
`nco.vhd`	VHDL top-level description of the custom IP core function
`nco.cmp`	VHDL component declaration for the IP core function variation
`nco.bsf`	Quartus II symbol file for the IP core function variation
`nco_st.v`	Generated NCO synthesizable netlist
`nco.vho`	VHDL IP functional simulation model
`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
`nco_model.m`	MatLab M-file describing a MatLab bit-accurate model
`nco_tb.m`	MatLab Testbench
`nco_sin.hex`	Intel Hex-format ROM initialization file
`nco.vec`	Quartus vector file
`nco_nativelink.tcl`	NativeLink simulation testbench
`nco.qip`	Quartus project information
`nco.html`	IP core function report file that lists all generated files

were correct, but for the LEs with 88 LEs (actual) to 72 LEs (estimated) we observe a 18% error margin.

To simulate the design we use the generated `TCL` script. Start the ModelSim simulator and change to the `NCO` directory and then type in the command window `do nco_vho_msim.tcl`. Several libraries and the design files are compiled and a 22 000 ns long simulation is performed. We zoom in to the first couple of clock cycles to see the output valid delay (≈ 6 clock cycles) and the sine period, as shown in Fig. 1.24. The same phase increment value $M = 214\,748\,365$ as in our function generator (see Fig. 1.20, p. 36) is used and we get a period of 20 clock cycles in the output signal. 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. But we can solved

this problem by attaching an adder with constant 512 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.

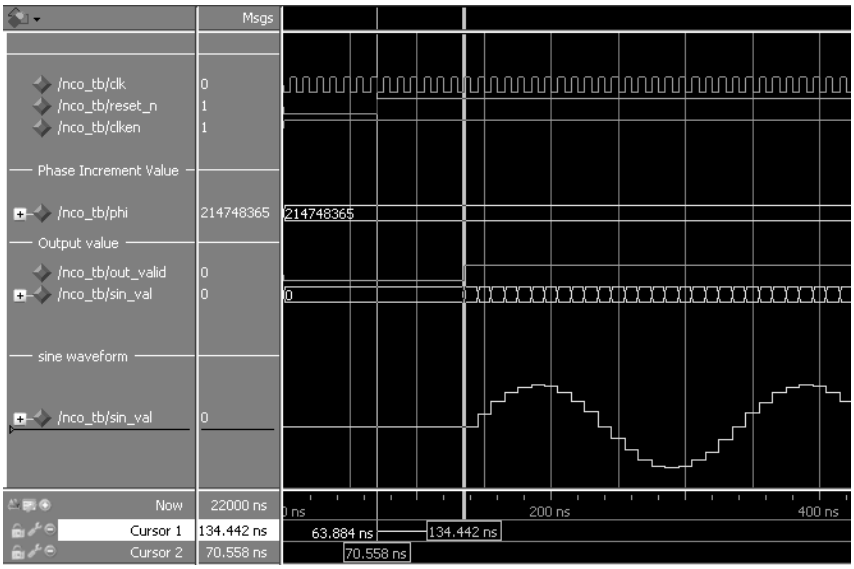


Fig. 1.24. Testbench for NCO IP design. Verification via timing simulation.

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. 31. If not otherwise noted use the EP4CE115F29C7N from the Cyclone IV E 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; $\times$=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 ModelSim-Altera tool (see p. 16). Start the ModelSim-Altera tool and change to the directory with the HDL file. Generate a work directory with `vlib work`. Then start the compilation with `vcom example.vhd`.
(b) Simulate the design using the file `example.do`. Start the script with `do example.do 0` for a functional simulation.
(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.
(d) Simulate the design with timing using the script `example.do`. Start the script with `do example.do 1` for a timing simulation.

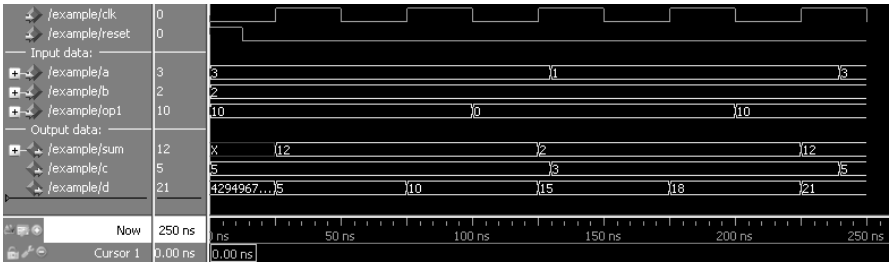


Fig. 1.25. Waveform file for Example 1.1 on p. 16.

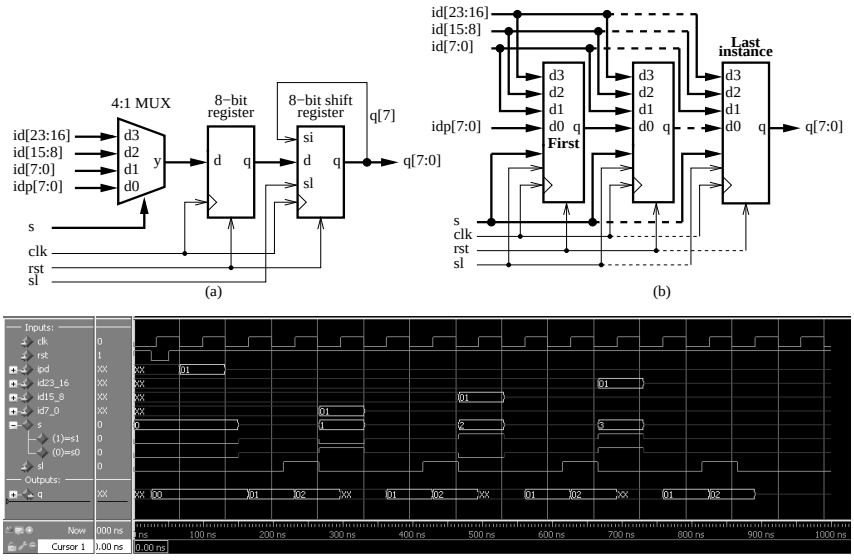
1.3: **(a)** Write a ModelSim-Altera simulation script `example.do` and match the stimuli for `clk,a,b,op1` that approximates that shown in Fig. 1.25.
(b) Conduct a simulation using the script `example.do`.
(c) Explain the algebraic relation between `a,b,op1` and `sum,d`.

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 `Fmax` for Slow 85C timing model and the LE's utilization of the designs from (a). Explain the results.

1.5: Develop a SDC file for the design as shown in the Altera tutorial `Using TimeQuest Timing Analyzer`. 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
(a) 20 ns,
(b) 10 ns,
(c) 5 ns,
(d) 3 ns,
use the clock report to determine the slack.
Note that for each clock period specification you need to run a complete recompile of the circuit. Also report any change in the resource of the circuits.

1.6: **(a)** Modify the file `fun_text.vhd` to use a `lpm_rom` component and a MIF file `sine.mif`. Simulate the circuit in ModelSim-Altera.
(b) In Quartus 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 the Quartus II software.
(b) Add I/O register and measure the registered performance `Fmax`. Compare the result with the data from Example 1.2 (p. 27).

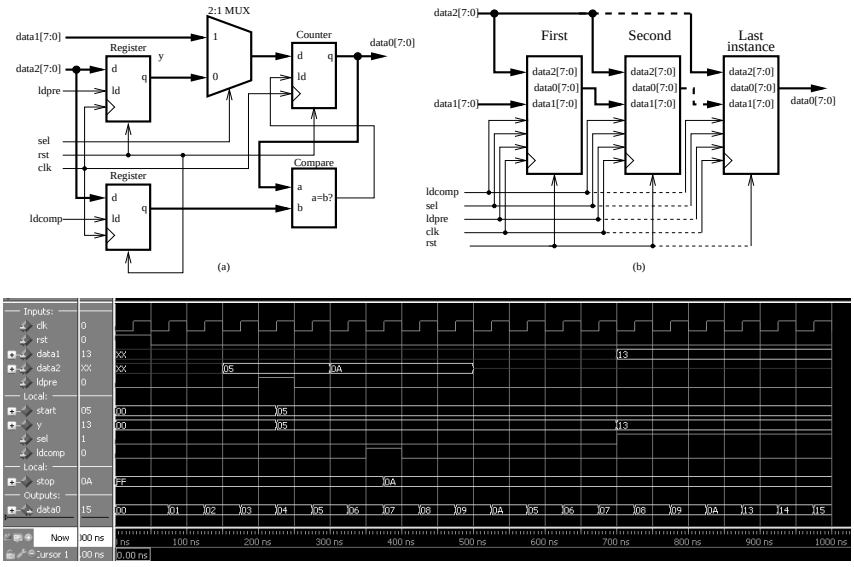


(c)

Fig. 1.26. 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. 1.26a 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. 1.26c for the function test.
(b) Determine the registered performance `Fmax` using the `TimeQuest` slow 85C model and the used resources (LEs, multipliers, and M2Ks/M9Ks) 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 `Fmax`?
Select one of the following devices:
(b1) EP4CE115F29C7N from the Cyclone IV E family
(b2) EPF10K70RC240-4 from the Flex 10K family
(b3) EPM7128LC84-7 from the MAX7000S family
(c) Design the multiple instantiation for benchmark 1 as shown in Fig. 1.26b.
(d) Determine the registered performance `Fmax` using the `TimeQuest` slow 85C model and the used resources (LEs, multipliers, and M2Ks/M9Ks) 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) EP4CE115F29C7N from the Cyclone IV E family
(d2) EPF10K70RC240-4 from the Flex 10K family
(d3) EPM7128LC84-7 from the MAX7000S family



(c)

Fig. 1.27. 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. 1.27a 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.

1.27c 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 `Fmax` using the `TimeQuest` slow 85C model and the used resources (LEs, multipliers, and M2Ks/M9Ks) 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 `Fmax`?
Select one of the following devices:
(b1) EP4CE115F29C7N from the Cyclone IV 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. 1.27b.
(d) Determine the registered performance `Fmax` using the `TimeQuest` slow 85C model and the used resources (LEs, multipliers, and M2Ks/M9Ks) 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) EP4CE115F29C7N from the Cyclone IV E 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 Quartus II 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, see `Logic`→ `Tri-State`
(b) A flip-flop with all control signals, see `Logic` → `Registers` → `Full-Featured Positive Edge Register with All Secondary Signals`
(c) A binary counter, see `Full Designs` → `Arithmetic` → `Counters`
(d) A Moore State Machine with asynchronous reset, see`Full Designs` → `State Machines`
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;`		

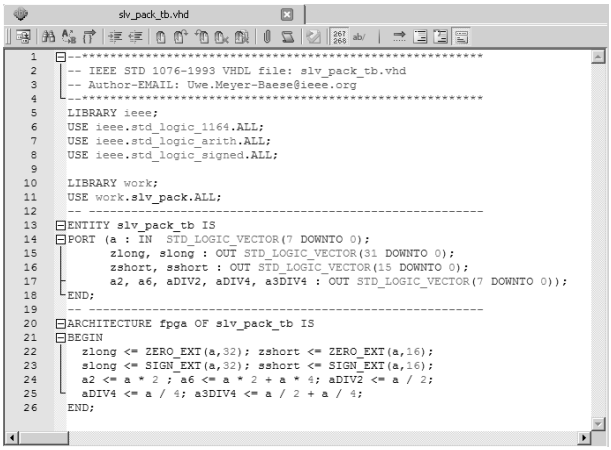
1.18: Determine for the following process statements
(a) the synthesized circuit and label I/O ports
(b) the cost of the design assuming a cost 1 per adder/subtractor
(c) the critical (i.e., worst-case) path of the circuit for each process. Assume a delay of 1 for an adder or subtractor.

```
-- 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;
```

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. 1.28 to verify the correct functionality.

1.20: **(a)** Design a function library for the `STD_LOGIC_VECTOR` data type that implement the following operation (defined in VHDL-1993 only for the `BIT_VECTOR` data type):
(a) `SRL` **(b)** `SRA` **(c)** `SLL` **(d)** `SLA`
(e) Use the testbench shown in Fig. 1.29 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.

(a)



(b)

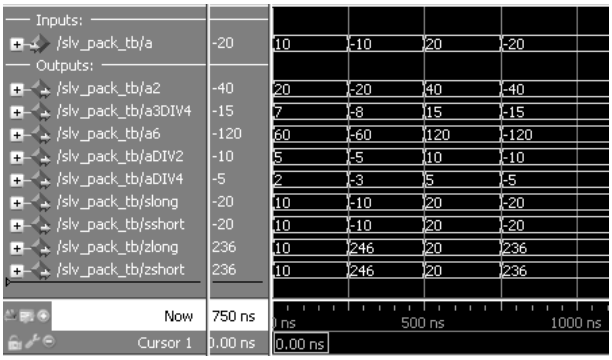
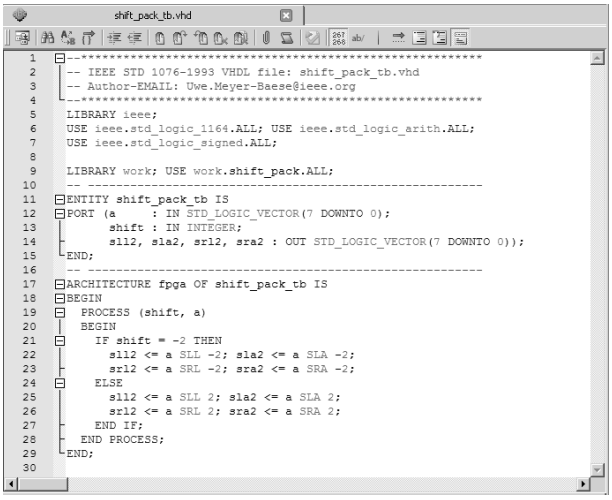


Fig. 1.28. `STD_LOGIC_VECTOR` package testbench. **(a)** HDL code. **(b)** Functional simulation result.

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;
```

(a)



(b)

Fig. 1.29. `STD_LOGIC_VECTOR` shift library testbench. **(a)** HDL code. **(b)** Functional simulation result.

```
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.