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 EPF10K20 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 Chapters 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 submicron designs, digital designs can often be much denser 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. 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 [7]. Fig. 1.1 shows a typical application used to implement an analog system by means of a PDSP. We

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

will return in Section 1.2.1 and Chapter 2 (p. 62) to PDSPs after we have studied FPGA architectures.

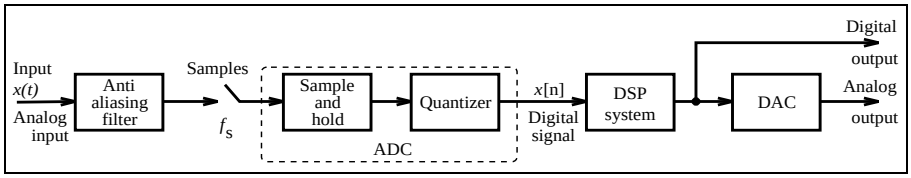

Fig. 1.1. A typical DSP application.

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 advantage, but also with high non-reoccurring engineering (NRE) costs. 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.

1.2.1 Classification by Granularity

Logic block size correlates to the *granularity* of a device which, 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 any binary logic function using NAND gates (see Exercise 1.1, p. 25), 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.

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

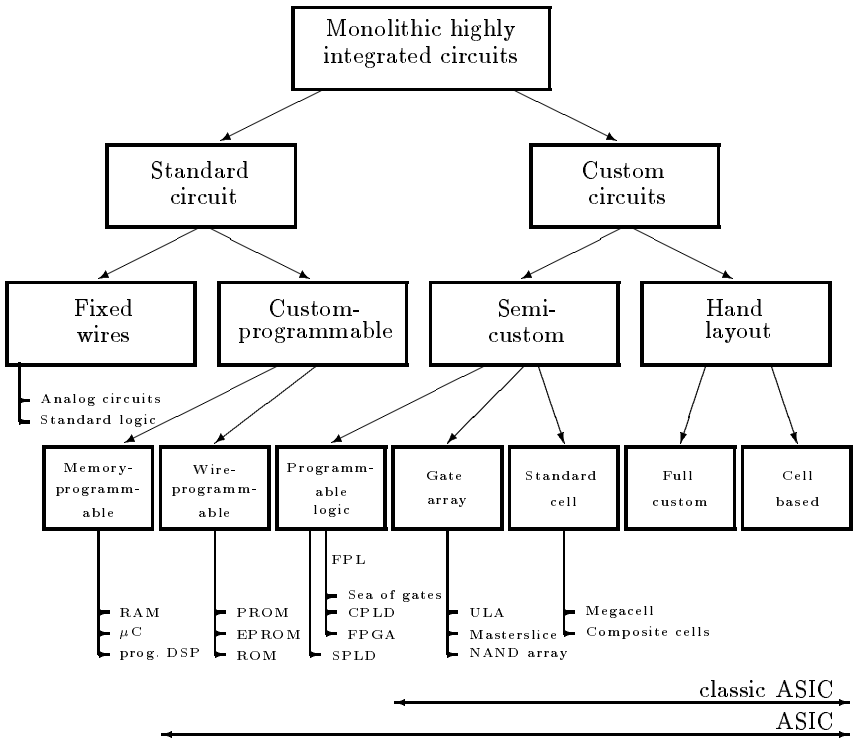

Fig. 1.2. Classification of VLSI circuits (©1995 VDI Press [4]).

Medium-Granularity Devices

The most common FPGA architecture is shown in Fig. 1.4(a). A concrete example of a contemporary medium-grain FPGA device is shown in Fig. 1.5. The elementary logic blocks are typically small tables (e.g., Xilinx XC2k-4k 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 [9]. 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 complex programmable logic devices (CPLD), are characterized in Fig. 1.4(b). They are defined by combining so-called simple programmable logic devices (SPLDs), like the classic GAL16V8 shown in Fig. 1.6. This SPLD consists of a programmable logic array (PLA) implemented as an AND/OR array and a universal I/O logic block. The SPLDs used in CPLDs typically have 8 to 10 inputs, 3 to 4 outputs, and

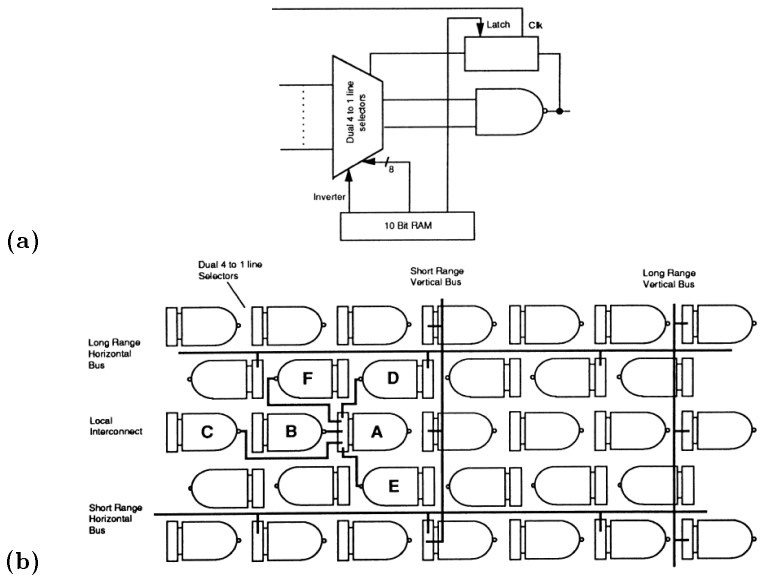

Fig. 1.3. Plessey ERA60100 architecture with 10K NAND logic blocks [8]. **(a)** Elementary logic block. **(b)** Routing architecture (©1990 Plessey).

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.

1.2.2 Classification by Technology

FPLs are available in virtually all memory technologies: SRAM, EPROM, E^2PROM, and antifuse [10]. 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 not "downloaded" to the device, it is better protected against unauthorized use. A recent innovation, based on an EPROM technology, is called "flash"

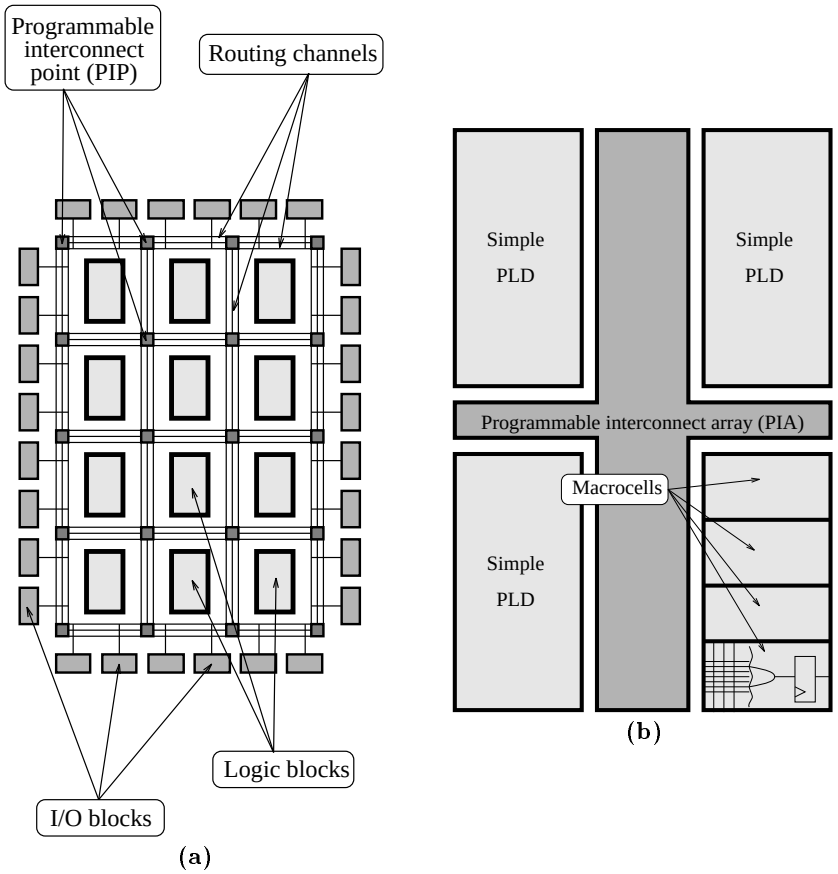

Fig. 1.4. (a) FPGA and **(b)** CPLD architecture (©1995 VDI Press [4]).

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 [11], Altera [12], and Actel [13], and has since expanded to more than 10 members. PREP has developed nine different benchmarks for FPLs that are summarized in Table 1.3. The central idea underlining the benchmarks is that each

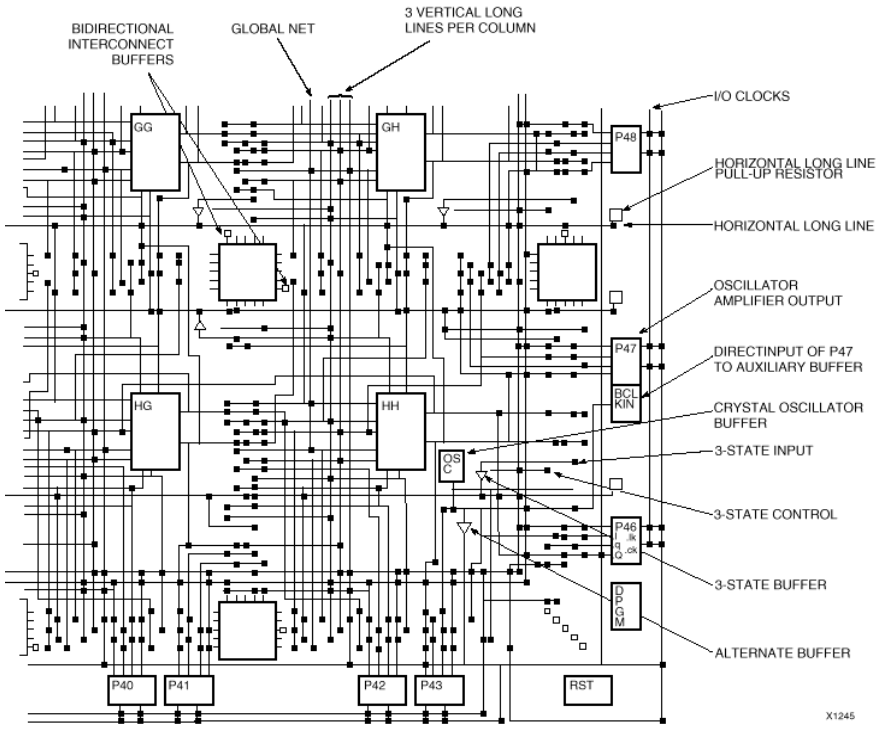

Fig. 1.5. Example of a medium-grain device (©1993 Xilinx).

vendor uses its own devices and software tools to implement the basic blocks

Table 1.2. FPL technology.

Technology	SRAM	EPROM	E^2PROM	Antifuse	Flash
Repro-grammable	✓	✓	✓	–	✓
In-system programmable	✓	–	✓	–	✓
Volatile	✓	–	–	–	–
Copy protected	–	✓	✓	✓	✓
Examples	Xilinx XC4K Altera Flex	Altera MAX5K Xilinx XC7K	AMD MACH Altera MAX 9K	Actel ACT	Xilinx XC9500 Cypress Ultra 37K

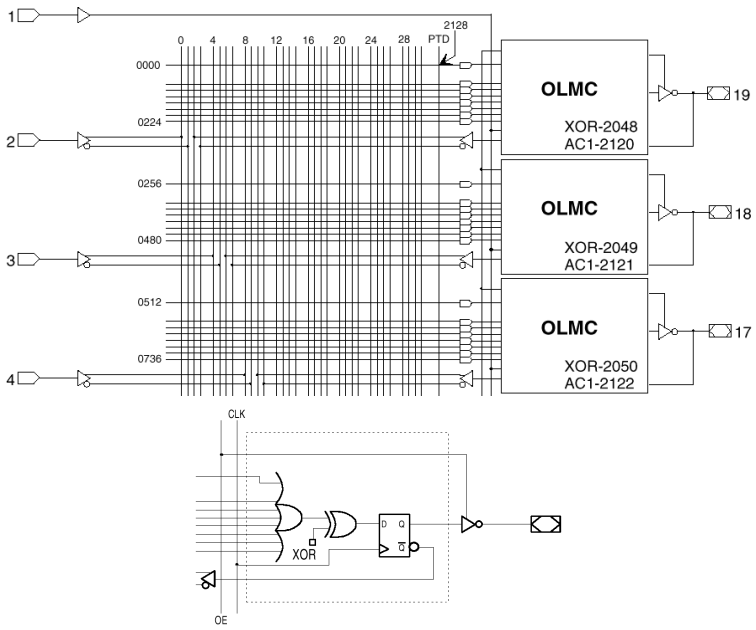

Fig. 1.6. The GAL16V8. **(a)** First three of eight macrocells. **(b)** The Output Logic macrocell (OLMC) (©1997 Lattice).

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. 1.7, repetition rates are reported over frequency, for typical Actel (A_k), Altera (o_k), and Xilinx (x_k) devices. It can be concluded that modern FPGA families provide the best DSP complexity and maximum speed. This is attributed to the fact that modern devices provide fast carry logic (see Sect. 1.4.1, p. 15) with delays (less than 0.5 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 a concrete applications additional attributes are also important. They include:

- On-chip RAM or ROM
- Pin-to-pin delay
- Internal tri-state bus
- Readback- or boundary-scan decoder
- Programmable slew rate or voltage of I/O
- Power dissipation

Table 1.3. The PREP benchmarks for FPLs.

Number	Benchmark Name	Description
1	Data path	Eight 4-to-1 multiplixers drive a parallel-load 8-bit shift register
2	Timer counter	Two 8-bit values are clocked through 8-bit value registers and compared
3	Small state machine	An 8-state machine with 8 inputs and 8 outputs
4	Large state machine	A 16-state machine with 40 transitions, 8 inputs, and 8 outputs
5	Arithmetic	A 4-by-4 unsigned multiplier an 9-bit accumulator
6	Accumulator	A 16-bit accumulator
7	Up counter	A 16-bit loadable binary up counter
8	Down counter	A 16-bit loadable binary down counter
9	Memory map	The map decodes address spaces ranging in size from 4Kbyte to 1Kbyte

Fig. 1.8 summarizes the power dissipation of some typical FPL devices. It can be seen that CPLDs (Altera) usually have higher "standby" power consumption. For higher frequency applications, FPGAs (Xilinx and Actel) can be expected to have a higher power dissipation. A detailed power analysis example can be found in Sect. 1.4.2, p. 20.

1.3 DSP Technology Requirements

The PLD market share, by vendor, is presented in Fig. 1.9. PLDs, since their introduction in early eighties, have enjoyed steady growth of 20% per annum, outperforming ASIC growth by more than 10%. The reason seems to be related to the fact that FPLs can offer many of the advantages of ASICs such as:

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

without many of the disadvantages of ASICs such as:

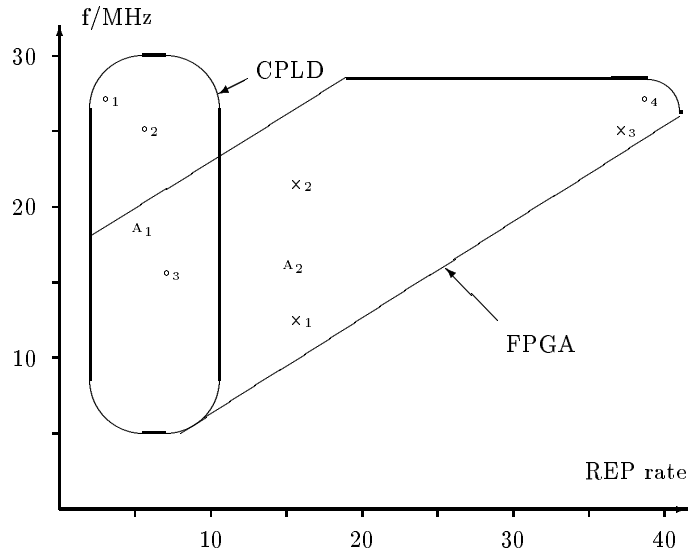

Fig. 1.7. Benchmarks for FPLs (©1995 VDI Press [4]).

- A reduction in development time (rapid prototyping) by three to four
- In-circuit reprogrammability
- Lower NRE costs resulting in more economical designs for solutions requiring less than 1,000 units

CBIC ASICs are used in high-end, high-volume applications (more than 1,000 copies). Compared to FPLs, CBIC ASICs typically have about ten times more gates for the same die size. An attempt to solve the second problem is the so-called hard wired FPGA, where a gate array is used to implement a verified FPGA design.

1.3.1 FPGA and Programmable Signal Processors

General purpose programmable digital signal processors (PDSPs) [14, 15, 6] 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. It can be argued that an FPGA can also be used to implement MAC cells [16], but cost issues will most often give PDSPs an advantage, if the PDSP meets the desired MAC rate. On the other side we

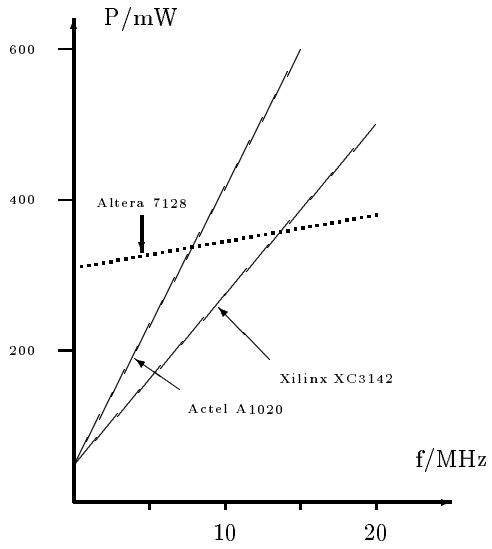

Fig. 1.8. Power dissipation for FPLs (©1995 VDI Press [4]).

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 [17] that in the future PDSPs will dominate applications that require complicated algorithms (i.e., several `if-then-else` constructs), while FPGAs will dominate more front end (sensor) applications like FIR filters, CORDIC algorithms, or FFTs, which will be the focus of this book.

1.4 Design Implementation

The levels of detail commonly used in VLSI designs range from geometrical layout of full custom ASICs to system design using so-called set top boxes. Table 1.4 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) macro cells or "mega core cells." Macro cells provide the designer with a collection of pre-

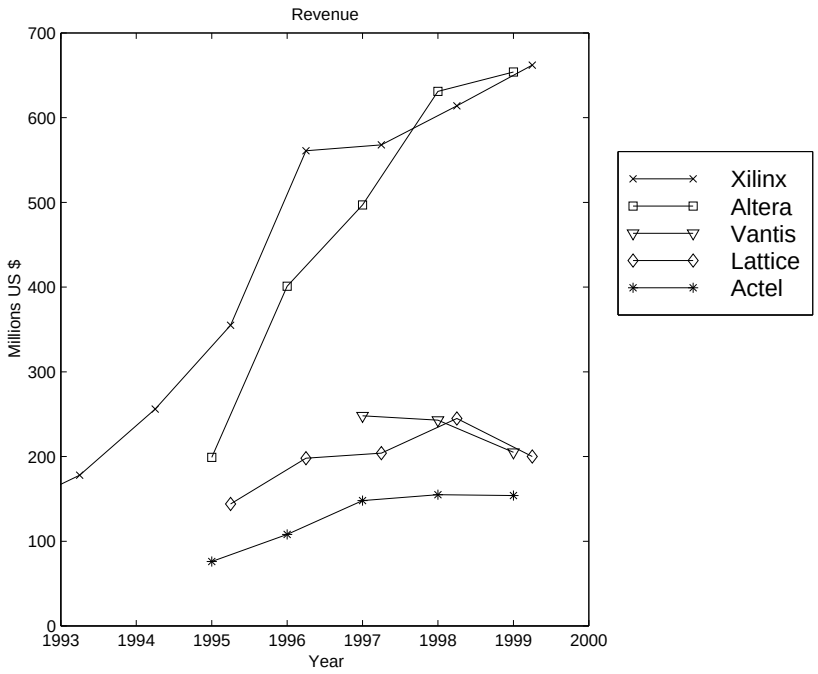

Fig. 1.9. Revenues of the top five vendors in the PLD/FPGA/CPLD market.

Table 1.4. 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

defined functions, such as microprocessors or UARTs. The designer, therefore, need only to specify selected features and attributes (i.e., 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

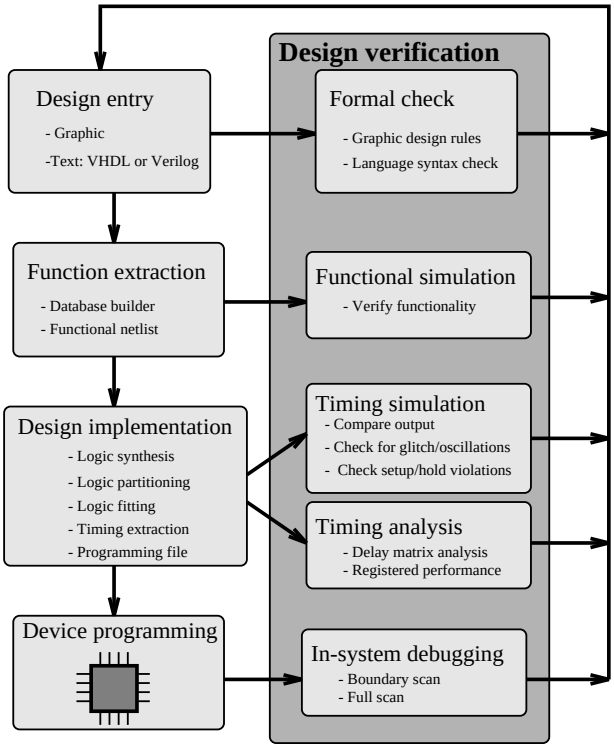

Fig. 1.10. CAD design circle.

A CAE tool taxonomy, as it applies to FPGA design flow, is presented in Fig. 1.10. In general, the decision 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 regards to algorithm control design and allows a wider range of design styles as demonstrated in the following design example. Specifically, for Altera's MaxPlusII, 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:

- Component instantiation (stuctural style, i.e., graphical netlist design)
- Data flow
- Sequential design using `PROCESS` templates (i.e., behavioral style)

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

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

LIBRARY work;
USE work.eight_bit_int.ALL;

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

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

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

ARCHITECTURE flex OF example IS

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

BEGIN

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

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

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

  p1: PROCESS                   ------> Behavioral style
  BEGIN
```

[4] The equivalent Verilog code `example.v` for this example can be found in Appendix A on page 343.

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

END flex;
```

1.1

After a successful functional (only) simulation of the design (for the MaxPlusII compiler mode select the option `Processing→Functional SNF Extractor`) we can proceed and start with the design implementation as reported in Fig. 1.10. To do this with the MaxPlusII compiler, we choose `Processing→Timing SNF Extractor,` and we will then notice that the compiler window now has three more entries, namely `Logic Synthesizer, Fitter`, and `Timing SNF Extractor`. After starting the compiler we can then conduct a simulation with timing, check for glitches, or measure the `Registered Performance` of the design, to name just a few options. After all these steps are successful, and if a hardware board (like the Altera University board) is available, we proceed with programming the device and may perform additional hardware tests using the "read back" methods, as reported in Fig. 1.10.

1.4.1 FPGA Structure

At the beginning of the twenty-first century two FPGA device families seemed to have the most attractive features for implementing DSP algorithms, due to the fact that these families provide fast carry logic, which allows implementations of 32-bit (nonpipelined) adders at speeds exceeding 50 MHz [1, 18, 19].

These two families are the Xilinx XC4000 family and the Altera FLEX 10K devices, which are Altera's 8K devices with additional 2Kbit RAM blocks called embedded array blocks (EAB). The Xilinx devices have the wide range of routing levels typical in FPGAs, while the Altera devices were based on the architecture with wide busses used in Altera's CPLDs. But the basic blocks of the FLEX 10K 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.

The basic logic elements of the Xilinx XC4000 family are called configurable logic blocks (CLB) and have two separate 4-input 1-output LUTs, fast carry, one additional 3-input 1-output LUT to combine the two separate LUTs, and two flip-flops, as shown in Fig. 1.11. The Xilinx device has five levels of routing, ranging from CLB to CLB, to long lines spanning the entire chip. Each CLB can be used as 16×2- or 32×1-bit RAM or ROM. Tables 1.5 shows some members of the Xilinx XC4000 family.

Table 1.5. The Xilinx XC4000 family.

Device	Total CLBs	Flip-flop bits	Max. RAM Kbits	Max. I/O
XC4003	100	360	3.2	80
XC4005	196	616	6.3	112
XC4010	400	1120	12.8	160
XC4025	1024	2560	32	256
XC4085	3136	7168	100	448
XC40150	5184	11520	165	448
XC40250	8464	18400	270	448

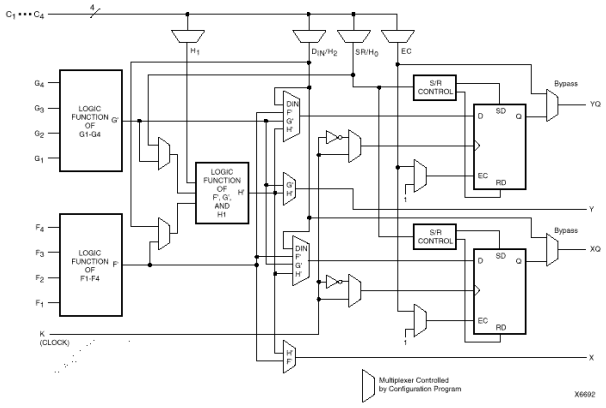

Fig. 1.11. XC4000 logic cell (©1993 Xilinx).

The basic block of the Altera FLEX 10K device achieves a medium granularity using small LUTs. The 10K device is an Altera 8K device with added 2Kbit RAM blocks, called embedded array blocks (EAB). The basic logic element in Altera FLEX 10K devices is called a logic element (LE)[5] and consists of a flip-flop, a 4-input 1-output LUT, or 3-input 1-output and a fast carry, logic or AND/OR product term expanders as shown in Fig. 1.12. Eight LCs are combined in a logic array block (LAB). Each row contains an embedded array block (EAB; i.e., a 2Kbit RAM or ROM) which can be configured as 256×8, 512×4, 1024×2, or 2048×1 memory devices. These EABs and LABs are connected through wide high-speed busses with 100 to 300 lines per column as shown in Fig. 1.13. Table 1.6 shows some members of the Altera FLEX 10K 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

[5] Sometimes also called logic cells (LCs) in a "design report file." See `example.rpt`.

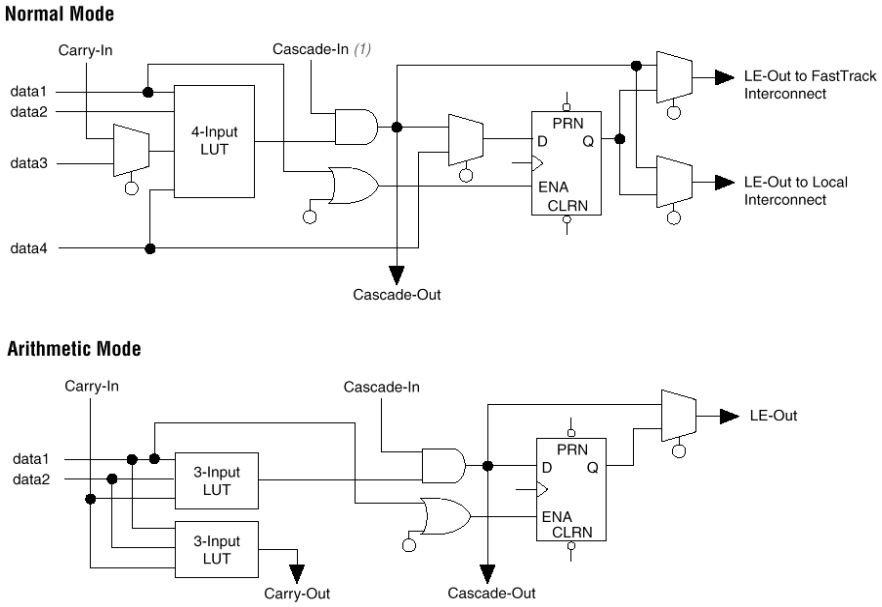

Fig. 1.12. FLEX logic cell (©1996 Altera).

less global routing resources is synergistic to DSP use because most digital signal processing algorithms process the data locally. The Altera approach, with wide busses, also has value, because typically not only are single bit processed in "bit slice" operations, but normally wide data vectors with 16 to 32 bits must be moved to the next DSP block.

Table 1.6. The FLEX 10K family.

Device	Total logic elements	Flip-flop bits	EAB Blocks	Max. RAM Kbits	Max. I/O
EPF10K10	576	720	3	6	134
EPF10K20	1152	1344	6	12	189
EPF10K30	1728	1968	6	12	246
EPF10K40	2304	2576	8	16	189
EPF10K50	2880	3184	10	20	310
EPF10K70	3744	4096	9	18	358
EPF10K100	4992	5392	12	24	406
EPF10K130	6656	7120	16	32	470
EPF10K250	12160	12624	20	40	470

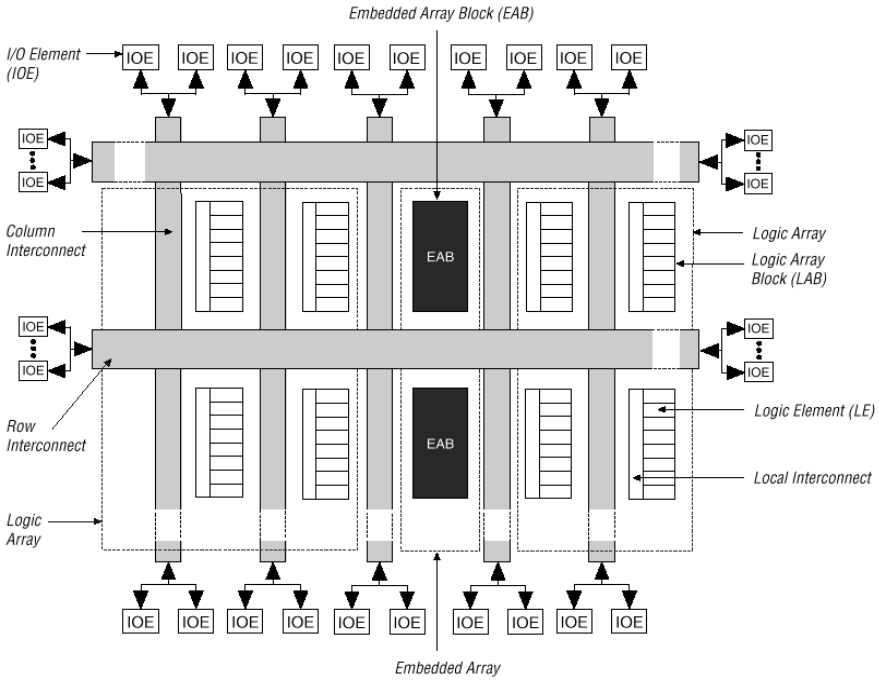

Fig. 1.13. Overall bus structure in FLEX 10K devices (©1996 Altera).

1.4.2 The Altera EPF10K20RC240-4

The Altera EPF10K20RC240-4 device, which is part of the demo board provided through Altera's University Program, is used throughout this book. The device nomenclature is interpreted as follows:

```
EPF10K20RC240-4
   |  |   |   |-> 4 ns device
   |  |   |-----> Package and pin number
   |  |---------> Equivalent gate count
   |------------> Device family
```

Specific design examples will, wherever possible, target Altera devices using Altera supplied software. The enclosed `MaxPlusII` software is a fully integrated system with VHDL and Verilog editor, synthesizer, simulator, and bitstream generator. Because all examples are available in VHDL and Verilog, any other simulator may also be used. For instance, the device-independent Synopsys FC2 or ModelTech compiler has successfully been used to compile the examples using the synthesizable code for `lpm` functions on the CD-ROM provided by EDIF.

Logic Resources

The EPF10K20 is a member of Altera 10K family and has a gate complexity equivaltent to about 20,000 two-input NAND gates. The maximum number of full adders which can be implemented may, however, be a more useful metric for DSP applications. From Table 1.6, it can be seen that the EPF10K20 device has 1,152 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. Eight LEs are always combined into a logic array block (LAB). The number of LABs is therefore 1152/8=144. These 144 LABs are arranged in six rows and 24 columns. The device also includes one 2Kbit memory block (called an embedded array block, or EAB) in the center of each row. The EPF10K20 has therefore six EABs, or a total of 12Kbits of memory. Fig. 1.13 presents part of the device floorplan.

Routing Resources

Each LAB has 22 inputs from each row and eight signals coming from the logic elements. There are four additional LAB control signals (e.g., preset of registers) and two local carry and cascade interconnects. To connect the LABs, the EPF10K20 uses fast, wide row and column busses, called "Fast-Track Interconnects." Each row bus is 144 lines wide with 24 channels per column. For improved routability, Altera has divided the row interconnect into full-length (a total of 96 channels) and half-length channels ($2 \times 48 = 96$ channels). The half-length channels end toward the middle of the channel where the EABs are located. The EABs can access both half-length channels. It is also interesting to note that the long carry chains skip alternate rows, so that only each second EAB occupies the same carry chain (see Fig. 1.17, p. 24).

Timing Estimates

Altera's `MaxPlusII` software calculates various timing data, such as the `Delay Matrix`, `Registered Performance`, and `Setup/Hold Matrix`. For a full description of all timing parameters, refer to Altera's web-page [19]. 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 16-bit Adder

Assume one is required to implement a 16-bit adder and estimate the design's maximum speed. The adder can be implemented in two LABs, each using the fast carry chain. The delay through the "same row" delay must be taken into account. The total delays are computed as follows: First, the two inputs must be stable t_{co}. Next, the first carry t_{cgen} must be generated, followed by

seven more carries inside the first LAB. The signal then goes through the row interconnect t_{samerow}. Inside the second LAB, seven additional carries must be computed and the MSB then must run through an LUT to complete the sum. The results are then stored in the LE register. The following table summarizes these timing data:

LE register clock-to-output delay	t_{co}	=	0.2 ns
Data-in to carry-out delay	t_{cgen}	=	1.5 ns
Carry-in to carry-out delay	$7 \cdot t_{\text{cico}}$	=	$7 \cdot 0.3 = 2.1$ ns
Row routing delay	t_{samerow}	=	2.9 ns
Carry-in to carry-out delay	t_{cico}	=	$7 \cdot 0.3 = 2.1$ ns
LE look-up table delay	t_{LUT}	=	1.9 ns
LE register setup time	t_{su}	=	2.7 ns
Total		=	13.4 ns

The estimated delay is 13.4 ns, or a rate of 74.6 MHz. The design is expected to use about 16 LEs (see also Exercise 1.7, p. 27). 1.2

If the two LABs used can not be placed in the same row then the same-column delay $t_{\text{samecolumn}} = 4.4$ ns applies (instead of t_{samerow}). The worst case occurs if the two LABs used are placed in different rows. The worst case delay becomes $t_{\text{diffrow}} = 10.1$ ns. It is therefore very important to check the floorplan and check for possible improvements "by hand" changes in the floorplan as described in the Altera "Getting Started" manual, pages 231-241 [20], or see `Ug/Maxiigs.pdf` on the CD-ROM.

Power Dissipation

The power consumption of an FPGA can be a critical design constraint, especially for mobile applications. Using 3.3V or 2.5V class devices is recommended in this case. To estimate the power dissipation of the Altera device EPF10K20RC240-4, three main sources must be considered, namely:

1) Standby power dissipation $I_{\text{standby}} \approx 0.5$ mA
2) I/O power dissipation $I_{\text{I/O}}$
3) Active power dissipation I_{active}

The first two are not design-dependent, and also the standby power in CMOS technology is generally small. The active current depends mainly on the clock frequency and the number of LEs in use. Altera provides the following empirical formula to estimate the active current:

$$I_{\text{active}} = 98 \cdot f_{\text{max}} \cdot N \cdot \tau_{\text{LE}} \cdot \frac{\mu\text{A}}{\text{MHz} \cdot \text{LE}} \tag{1.1}$$

where f_{max} is the maximum operating frequency in MHz, N is the total number of logic cells used in the device, and τ_{LE} the average percent of logic cells toggling at each clock (typically 12%). If, for instance, a design uses all LEs of the EPF10K20RC240-4 and the maximum frequency is 25 MHz, then the current will be estimated at 338 mA.

The following case study should be used as a detailed scheme for the examples and self-study problems in the next 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. 1.14). The synthesizer consists of a 32-bit accumulator, with the eight most significant bits (MSBs) wired to a SIN-ROM lookup table (LUT) to produce the desired output waveform. A *graphical* solution, using Altera's `MaxPlusII` software, is shown in Fig. 1.15, and can be found on the CD-ROM as `book/vhdl/fun_graf.gdf`. The following VHDL text file implements the design using "component instantiation," consisting of

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

Design Compilation

To check and compile the file, start the MaxPlusII Software and select `File`→`Open` to load `fun_text.vhd`. Notice that the top and left menus have changed. The VHDL design[6] reads as follows:

[6] The equivalent Verilog code `fun_text.v` for this example can be found in Appendix A on page 344.

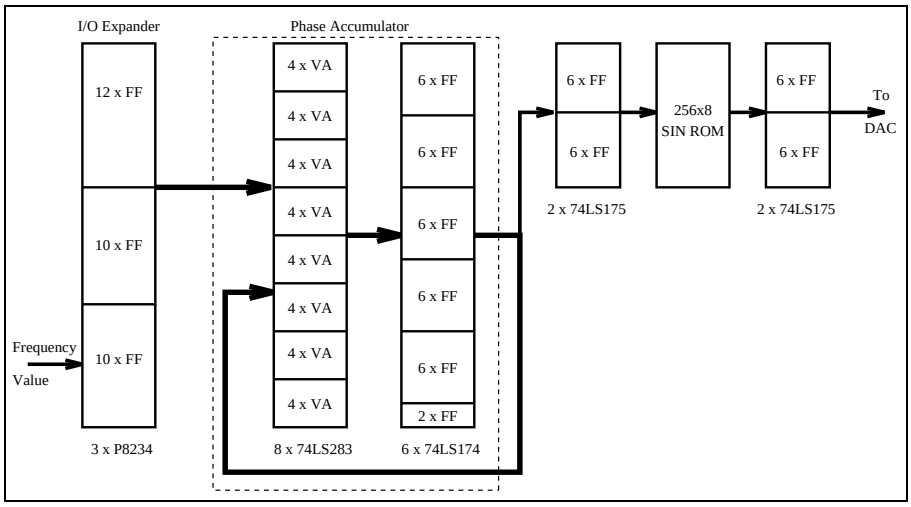

Fig. 1.14. PM5190 frequency synthesizer.

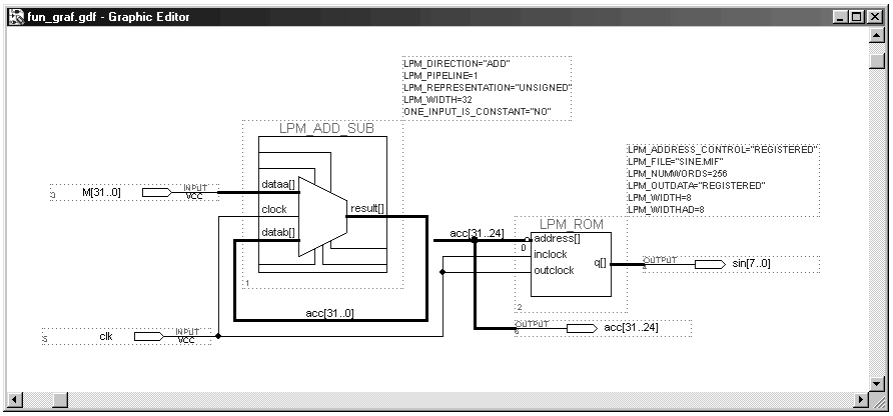

Fig. 1.15. Graphical design of frequency synthesizer.

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

LIBRARY lpm;
USE lpm.lpm_components.ALL;

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

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

ARCHITECTURE fun_gen OF fun_text IS

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

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

```
              datab => acc32,
              result => s );

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

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

  acc <= msbs;

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

END fun_gen;
```

The object `LIBRARY`, found early in the code, contains predefined modules and definitions. The `ENTITY` block specifies I/O ports of the device and generic variables. Using component instantiation, three blocks (see labels `add1, reg1, rom1`) are called like subroutines. The "`select1`" `PROCESS` construct is used to select the eight MSBs to address the ROM. To set the project to the current file, select `File` → `Project` → `Set Project to Current File`. To optimize the design for speed, choose the menu `Assign`→ `Global Project Logic Synthesis` option `Optimize 10` (Speed), and set `Global Project Synthesis Style` to `FAST`. Set the device type to FLEX10K20 by selecting in the menu `Assign`→ `Device` for `Device Family`, the option `FLEX10K`. For `Devices` we select `EPF10K20RC240-4`. Next, start the syntax checker with `<Ctrl+K>` or by selecting `File` → `Project` → `Save & Check`. The compiler checks for basic syntax errors and produces the netlist file `fun_text.cnf`. After the syntax check is successful, compilation can be

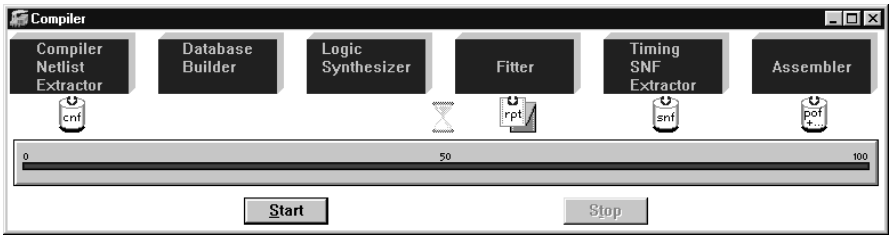

Fig. 1.16. Compilation steps in MaxPlusII.

started by pressing the `START` button in the compiler window or selecting `File` → `Project` → `Save & Compile`. If all compiler steps were successfully completed, the design is fully implemented. Fig. 1.16 summarizes all the processing steps of the compilation as shown in the MaxPlusII compiler window.

Floor Planing

The design results can be verified by opening `File`→`Open` → `fun_text.rpt`, or double click on the "rpt" button found in the compiler window (see Fig 1.16). Under `Utilities`→ `Find Text` →`LCs`, find in "device summary" the number of LCs and memory blocks used. In the report file, find the pin-out of the device and the result of the logic synthesis (i.e., the logic equations). Check the memory initialization file `sine.mif`, containing the sine table in offset binary form. This file was generated using the program `sine.exe` included on the CD-ROM under `book/util`. Select `MaxPlusII` → `Floorplan Editor` to view the physical implementation. Use the "reduce scale" button to produce the screen shown in Fig. 1.17. Notice that the accumulator uses

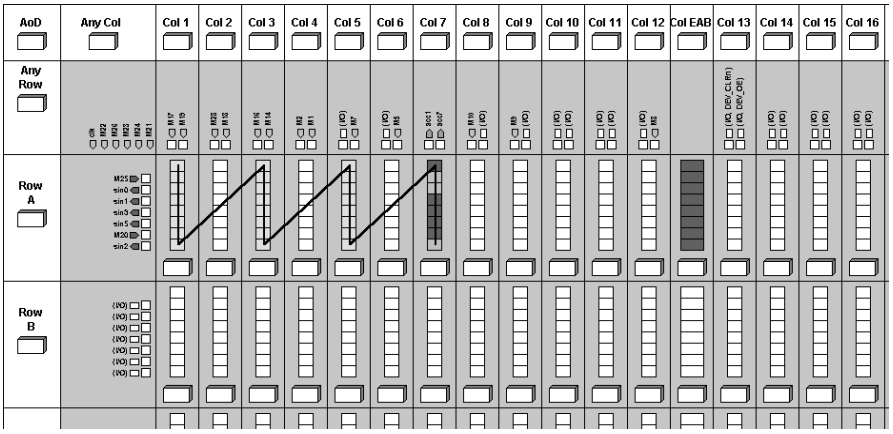

Fig. 1.17. Floorplan of frequency synthesizer design.

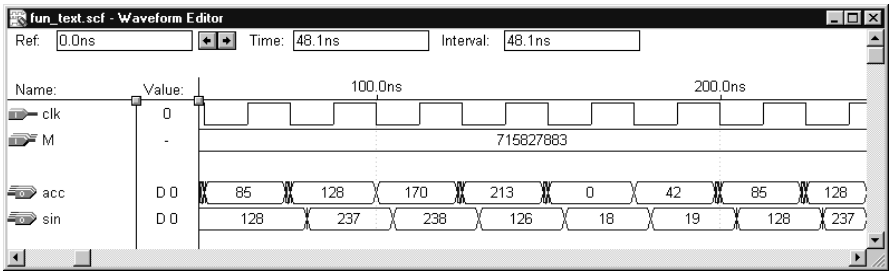

Fig. 1.18. VHDL simulation of frequency synthesizer design.

fast carry chains, and that only every second column has been used for the improved routing as explained in Sect. 1.4.2, p. 19.

Simulation

To simulate, open the prepared waveform `File`→`Open`→`fun_text.scf`. Notice that the top and left menu lines have changed. Set the time from the menu `File`→`End Time` to 1μs. In the `fun_text.scf` window, click on the `clk` symbol and set (left menu buttons) the `Clock Period` to 25 ns in the `Overwrite Clock` window. Set $M = 715827883$ ($M = 2^{32}/6$), so that the period of the synthesizer is 6 clock cycles long. Start the simulation by selecting `MaxPlusII`→`Simulator` and press the start button. The simulation should give an output similar to Fig. 1.18. Notice that the ROM has been coded in binary offset (i.e., zero=128). When complete, change the frequency so that a period of 8 cycles occurs, i.e., ($M = 2^{32}/8$), and repeat the simulation.

Performance Analysis

To initiate a performance analysis, enter the `MaxPlusII`→`Timing Analyzer`. Note that the menu line has again changed. Select `Analysis`→`Registered Performance` and the appropriate `Registered Performance` screen will appear. Click on the `Start` button to measure the register performance. The result should be similar to that shown in Fig. 1.19.

This concludes the case study of the frequency synthesizer.

Exercises

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) $c_{\text{out}} = a \cdot b + c_{\text{in}} \cdot (a + b)$
(Note: +=OR; $\cdot$=AND)
(c) Show that the two-input NAND is *universal* by implementing NOT, AND, and OR with NAND gates.

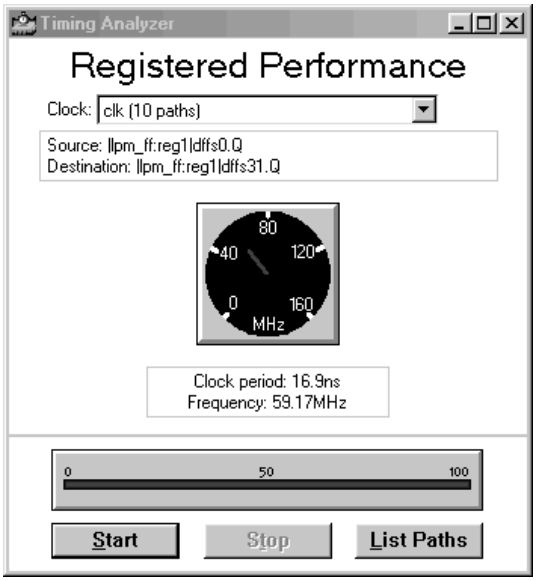

Fig. 1.19. Register performance of frequency synthesizer design.

Exercises Using MaxPlusII

1.2: **(a)** Compile the file `example.vhd` using the `MaxPlusII` compiler (see p. 13) in the functional mode. Select as compiler option `Processing`→`Functional SNF Extractor`.
(b) Simulate the design using the file `example.scf`.
Note: If you have no prior experience with the MaxPlusII software, refer to the case study found in Sect. 1.4.3, p. 21.
(c) Compile the file `example.vhd` using the `MaxPlusII` compiler with timing extraction. Select as compiler option `Processing`→`Timing SNF Extractor`.
(d) Simulate the design using the file `example.scf`.
(e) Turn on the option `Check Outputs` in the simulator window and compare the functional and implemented SNF.

1.3: **(a)** Generate a waveform file for `clk,a,b,op1` that approximates that shown in Fig. 1.20.
(b) Conduct a simulation using the VHDL code `example.vhd`.
(c) Explain the algebraic relation between `a,b,op1` and `sum,d`.

1.4: **(a)** Compile the file `fun_text.vhd` with the synthesis style (`Assign`→`Global Project Logic Synthesis`) `Fast` and `Normal`.
(b) Evaluate `Registered Performance` and the LC's utilization of the two designs from (a). Explain the results.

1.5: **(a)** Compile the file `fun_text.vhd` with the synthesis style (`Assign` → `Global Project Logic Synthesis`) `Fast` and compiler option `Processing`→ `Timing SNF Extractor`.
Use the waveform file `fun_text.snf` and
(b1) Set the period of the clock signal to 20 ns and use the simulator to check

`Setup/Hold`, `Check Ouputs`, `Oscillation`, and `Glitch`.
(b2) Set the period of the clock signal to 15 ns and use the simulator to check `Setup/Hold`, `Check Ouputs`, `Oscillation`, and `Glitch`.

1.6: **(a)** Open the file `fun_text.scf` and start the simulation.
(b) Select the simulator window with the top menu line labelled `Initialize`. Select `Initialize Memory` and export the ROM table in Intel HEX format as `sine.hex`.
(c) Change the `fun_text.vhd` file so that it uses the Intel HEX file `sine.hex` for the ROM table, and verify the correct results through a simulation.

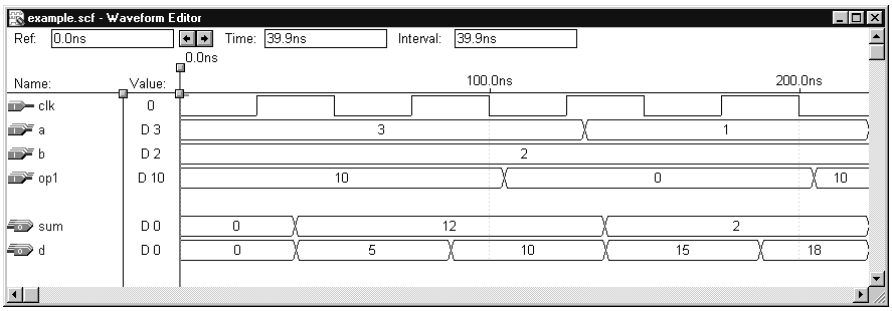

Fig. 1.20. Waveform file for example 1.1 on p. 13.

1.7: **(a)** Design a 16-bit adder using the `LPM_ADD_SUB` macro with the MaxPlusII software.
(b) Measure the `Registered Performance` and compare the result with the data from Example 1.2 (p. 19).