Hideharu Amano *Editor*

Principles and Structures of FPGAs

Springer

Principles and Structures of FPGAs

Hideharu Amano
Editor

Principles and Structures of FPGAs

Springer

Editor
Hideharu Amano
Keio University
Yokohama
Japan

ISBN 978-981-13-0823-9   ISBN 978-981-13-0824-6 (eBook)
https://doi.org/10.1007/978-981-13-0824-6

Library of Congress Control Number: 2018943376

Preface

The field-programmable gate array (FPGA) is one of the most important electronic devices to emerge in the past two decades. Now we can use more than 50,000-gate fully programmable digital devices for only 300 USD by using the free WebPACK including sophisticated high-level synthesis (HLS) design tools. FPGA has been used in most recent IT products including network controllers, consumer electronics, digital TVs, set-top boxes, vehicles, and robots. It is a leading technology device for advanced semiconductors; that is, the most advanced semiconductor chips are fabricated not for CPUs but for FPGAs. Recently, Intel acquired Altera, a leading FPGA vendor, and has employed accelerators for various types of applications in cloud computing. Especially, big data processing and deep learning used in artificial intelligence are killer applications of FPGAs, and "FPGAs in the cloud" has currently become an extremely popular topic in this field.

This book introduces various aspects of FPGA: Its history, programmable device technologies, architectures, design tools, and examples of application. Although a part of the book is for high school or university students, the advanced part includes recent research results and applications so that engineers who use FPGAs in their work can benefit from the information. To the best of our knowledge, it is the first comprehensive book on FPGA covering everything from devices to applications.

Novice learners can acquire a fundamental knowledge of FPGA, including its history, from Chap. 1; the first half of Chap. 2; and Chap. 4. Professionals who are already familiar with the device will gain a deeper understanding of the structures and design methodologies from Chaps. 3 to 5. Chapters 6–8 also provide advanced techniques and cutting-edge applications and trends useful for professionals.

Most of the descriptions in this volume are translated from a Japanese book published by Ohmsha, *The Principle and Structure of FPGA* (2016), but new material has been added. We are very grateful to Ohmsha for generously allowing this kind of publishing venture.

The chapters are written by top-level Japanese researchers in the field. The manuscripts were thoroughly checked and corrected by Dr. Akram Ben Armed of Keio University and Dr. Doan Anh Vu of the Technical University of Munich. I express my sincere gratitude for their efforts.

Yokohama, Japan Hideharu Amano
April 2018

Contents

Contributors

Motoki Amagasaki Kumamoto University, Kumamoto, Japan

Masanori Hariyama Tohoku University, Sendai, Japan

Masahiro Iida Kumamoto University, Kumamoto, Japan

Tomonori Izumi Ritsumeikan University, Kusatsu, Japan

Tsutomu Maruyama University of Tsukuba, Tsukuba, Japan

Yukio Mitsuyama Kochi University of Technology, Kami, Japan

Masato Motomura Hokkaido University, Sapporo, Japan

Hiroki Nakahara Tokyo Institute of Technology, Tokyo, Japan

Yasunori Osana University of the Ryukyus, Ryukyus, Japan

Kentaro Sano RIKEN, Kobe, Japan

Yuichiro Shibata Nagasaki University, Nagasaki, Japan

Toshinori Sueyoshi Kumamoto University, Kumamoto, Japan

Minoru Watanabe Shizuoka University, Shizuoka, Japan

Yoshiki Yamaguchi University of Tsukuba, Tsukuba, Japan

Chapter 1
Basic Knowledge to Understand FPGAs

Toshinori Sueyoshi

Abstract An FPGA is a wonderful digital device which can implement most of the practically required digital circuits with much easier effort than other solutions. For understanding FPGAs, fundamental digital design techniques such as logic algebra, combinational circuits design, sequential circuits design, and static timing analysis are required. This chapter briefly introduces them first. Then, the position of FPGA among various digital devices is discussed. The latter part of this chapter is for 40-year history of programmable devices. Through the history, you can see why SRAM style FPGAs have become dominant in various types of programmable devices, and how Xilinx and Altera (Intel) have grown up major FPGA vendors. Various small vendors and their attractive trials that are not existing now are also introduced.

Keywords Digital circuits' design · Static timing analysis · Programmable logic devices · Field-programmable gate array

1.1 Logic Circuits

Field-programmable gate array (FPGA) is a logic device that can implement user-desired logics by programming logic functions. To understand the structure and design of FPGAs, the basis of logic circuits is briefly introduced in [1, 2].

1.1.1 Logic Algebra

In logic algebra, also called Boolean algebra, all variables can take either the value 0 or 1. Logic algebra is an algebraic system defined by the operators AND, OR, and NOT applied to such logic values (0,1). AND, OR, and NOT are binary or unary operators defined in Table 1.1. Here, we use the symbols "·", "+", and "$^-$" for these

T. Sueyoshi (✉)
Kumamoto University, Kumamoto, Japan
e-mail: sueyoshi@cs.kumamoto-u.ac.jp

H. Amano (ed.), *Principles and Structures of FPGAs*,
https://doi.org/10.1007/978-981-13-0824-6_1

Table 1.1 Axioms of logic algebra

AND (·)	OR (+)	NOT ($^-$)
$0 \cdot 0 = 0$	$0 + 0 = 0$	$\bar{0} = 1$
$0 \cdot 1 = 0$	$0 + 1 = 1$	
$1 \cdot 0 = 0$	$1 + 0 = 1$	$\bar{1} = 0$
$1 \cdot 1 = 1$	$1 + 1 = 1$	

Table 1.2 Theorems of logic algebra

Zero element	$x \cdot 0 = 0, x + 1 = 1$
Neutral element	$x \cdot 1 = x, x + 0 = x$
Idempotent law	$x \cdot x = x, x + x = x$
Complement law	$x \cdot \bar{x} = 0, x + \bar{x} = 1$
Involution law	$\bar{\bar{x}} = x$
Commutative law	$x \cdot y = y \cdot x, x + y = y + x$
Associative law	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
	$(x + y) + z = x + (y + z)$
Distribution law	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
	$x + (y \cdot z) = (x + y) \cdot (x + z)$
Absorption	$x + (x \cdot y) = x$
	$x \cdot (x + y) = x$
De Morgan's laws	$\overline{x + y} = \bar{x} \cdot \bar{y}$ $\overline{x \cdot y} = \bar{x} + \bar{y}$

three logic operators, respectively. AND ($x \cdot y$) is an operation whose result is 1 when both x and y are 1. OR ($x + y$) is an operation whose result is 1 when either x or y is 1. NOT ($\bar{x}$) is a unary operation giving the inverse of x; that is, when x is 0 its result is 1, otherwise its result is 0. In logic algebra, the theorems shown in Table 1.2 are satisfied. Here the symbol "=" shows that both sides are always equal or equivalent. By exchanging logic value 0 into 1, and operation AND into OR, the equivalent logic system is formed. This is called a dual system. In logic algebra, if a theorem is true, its dual is also true.

1.1.2 Logic Equation

A logic equation consists of an arbitrary number of logic operations, logic variables, and binary constants, separated by parentheses if needed to represent the order of computation. When a logic equation is formed with n logic variables $x_1, x_2, x_3, \ldots, x_n$, its result is either 0 or 1 according to the procedure represented with an equation by substituting 0 or 1 in the variables (2^n in total), following an arbitrary combination. That is, a logic equation represents a logic function $F(x_1, x_2, x_3, \ldots, x_n)$. If the priority is not defined by parentheses, AND is given a higher priority than OR. The AND operator "·" is often omitted. Arbitrary logic functions can be represented by logic equations, but there are a lot of logic equations for representing the same logic function. Thus, by giving some restrictions, a logic function can be 1 to 1 corresponding to a logical equation. It is called a standard logic form. A single variable or its inverse is called a literal. Logical AND of literals which does not allow duplication of itself is called a product term. The sum-of-products form is a logic equation only with logical OR of products. A product term formed from all literals is called a minterm. A sum-of-products only containing minterms is called sum-of-products canonical form. A product-of-sums is a dual of a sum-of-products. A maxterm is formed with OR of literals for all inputs without duplication. A product-of-sums canonical form is formed only with maxterms.

1.1.3 Truth Table

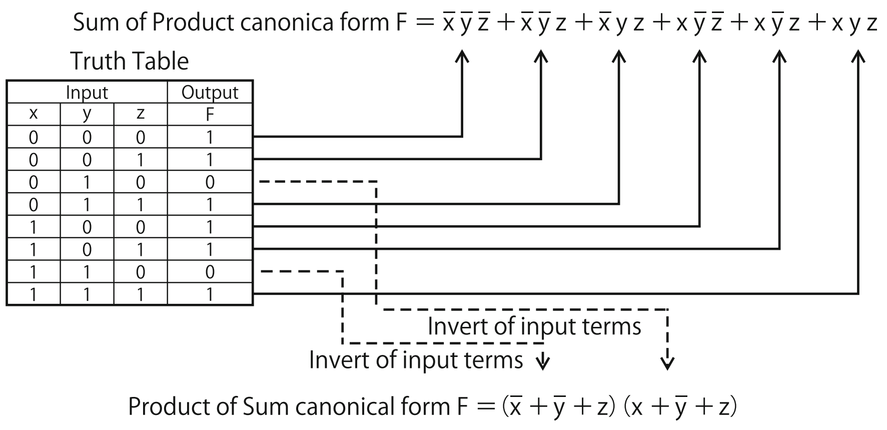
Truth tables and logic gates (shown later) are representations of logic functions other than logic equations. The table which enumerates all combinations of inputs and corresponding outputs is called the truth table. In the case of combinational circuits, a truth table can represent all combinations of inputs, and so it is a complete representation of the circuit. The specification of a combinational circuit is defined in the form of a truth table. For n inputs, the number of entries of the truth table is 2^n. The corresponding output is also added to the entry.

A truth table is a unique representation of a logic function. Although a logic equation also represents a unique logic function, a logic function can be represented with various equivalent logic equations. A straightforward implementation of a truth table is called lookup table(LUT), which is used in major FPGAs.

From a truth table, two canonical forms such as sum-of-products or product-of-sums can be induced. The sum-of-products canonical form is derived by making minterms of input variables when the corresponding output is 1, and then applying the OR operator. On the other hand, the product-of-sums canonical form is derived by making maxterms of inverted input variables when the corresponding output is 0, and then applying the AND operator. An example of making a logical equation from a truth table is shown in Fig. 1.1.

1.1.4 Combinational Circuits

A logic circuit can be classified into combinational or sequential whether it includes memory elements or not. In combinational circuits, which do not include memory elements, the output is defined only with current input values. Combinational circuits have a given number of inputs and outputs and consist of logic gates computing basic


Sum of Product canonica form $F = \bar{x}\,\bar{y}\,\bar{z} + \bar{x}\,\bar{y}\,z + \bar{x}\,y\,z + x\,\bar{y}\,\bar{z} + x\,\bar{y}\,z + x\,y\,z$
Truth Table
Invert of input terms
Invert of input terms
Product of Sum canonical form $F = (\bar{x} + \bar{y} + z)\,(x + \bar{y} + z)$

Input			Output
x	y	z	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Fig. 1.1 An example of making a logical equation

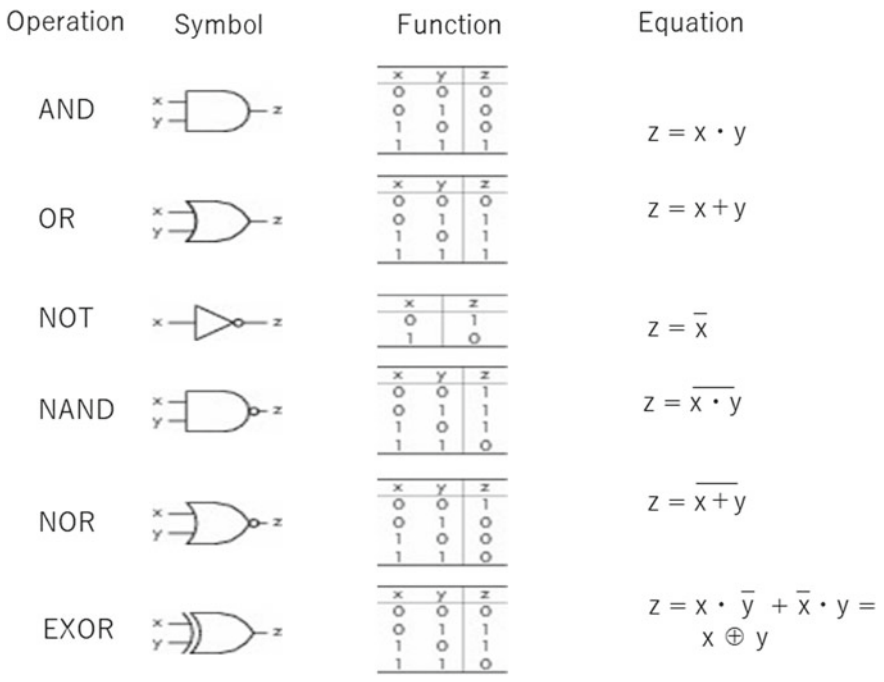
logical functions such as AND, OR, and NOT connected with wires. These logic gates correspond to three basic operations: Logical and, logical or, and logical not are called AND gate, OR gate, and NOT gate, respectively. Additionally, there are gates for well-known binary operations: NAND gate, NOR gate, and EXOR gate. NAND gate, NOR gate, and EXOR gate compute inverted AND, inverted OR, and exclusive OR, respectively. Figure 1.2 shows their symbols (MIL symbols), truth tables, and logical equations. $\oplus$ is used for the symbol for logic operation of exclusive OR. The table shows two inputs gates for binary operations, while gates with more than three inputs are also used. CMOS used in most of the current major semiconductor LSIs often includes compound gates like OR-AND-NOT or AND-OR-NOT.

Any logic circuits can be represented with the sum-of-products canonical form. Thus, any combinational circuits can represent any arbitrary logic function by a NOT-AND-OR form. This is called AND-OR two-stage logic circuits or AND-OR array. AND-OR two-stage logic circuits are implemented by a programmable logic array (PLA).

1.1.5 *Sequential Circuits*

Logic circuits including memory elements are called sequential circuits. While combinational circuits decide their outputs only with the current inputs, outputs of sequential circuits are not fixed with only current inputs. That is, the prior inputs influence the current output.

Sequential circuits are classified into synchronous and asynchronous. In synchronous sequential circuits, outputs and internal states are changed synchronously following a clock signal, while asynchronous sequential circuits do not have a clock signal. Here, only synchronous circuits used in most FPGA design are introduced.



Operation	Function	Equation
AND	x y z: 0 0 0 / 0 1 0 / 1 0 0 / 1 1 1	$z = x \cdot y$
OR	x y z: 0 0 0 / 0 1 1 / 1 0 1 / 1 1 1	$z = x + y$
NOT	x z: 0 1 / 1 0	$z = \overline{x}$
NAND	x y z: 0 0 1 / 0 1 1 / 1 0 1 / 1 1 0	$z = \overline{x \cdot y}$
NOR	x y z: 0 0 1 / 0 1 0 / 1 0 0 / 1 1 0	$z = \overline{x + y}$
EXOR	x y z: 0 0 0 / 0 1 1 / 1 0 1 / 1 1 0	$z = x \cdot \overline{y} + \overline{x} \cdot y = x \oplus y$

Fig. 1.2 Basic logic gates

Outputs of synchronous circuits are determined both by the inputs and the memorized values. That is, states depending on past inputs value influence the current outputs in sequential circuits. They are represented with a model of finite-state automaton as shown in Fig. 1.3. Figure 1.3a shows Mealy finite-state machine, while Fig. 1.3b illustrates Moore finite-state machine. Outputs are determined by the internal states and inputs in Mealy machine, while in Moore machine, they are only depending on their internal states. Compared with Mealy machine, Moore machine can decrease the size of the circuits, since a smaller number of states are required for the target function. However, outputs are directly influenced by the change of input signals and so the signal can glitch because of the difference of gate or wiring delay which may lead to unpredicted hazards. On the other hand, Moore machine can directly use states to generate outputs; thus, high-speed operation without hazard can be achieved. The circuits' size can become large because of the increasing number of states.

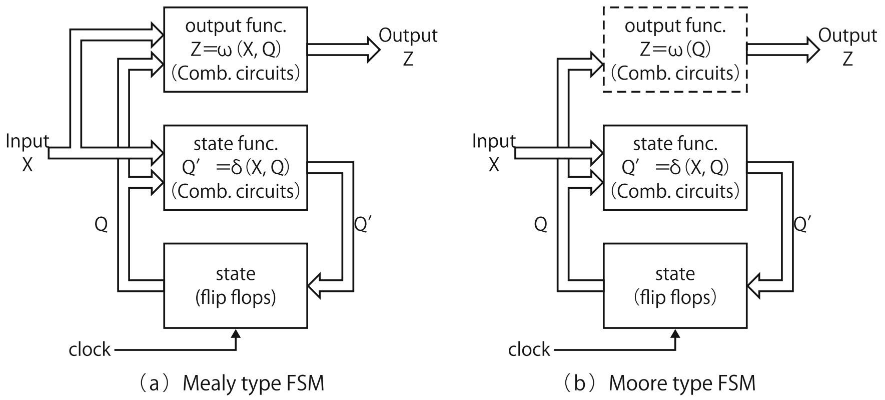


Fig. 1.3 Mealy machine (**a**) and Moore machine (**b**)

1.2 Synchronous Logic Design

In synchronous logic design, all states of the system are idealized to change synchronously with a clock so as to make the design simple. It is a fundamental design policy used in FPGAs.

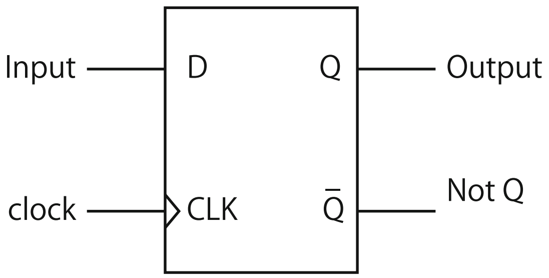
1.2.1 Flip-Flop

A one-bit memory element called flip-flop (FF) is used as a memory element in sequential circuits. D-flip-flops (D-FFs), embedded in basic blocks of an FPGA, change their outputs at the rising edge (or falling edge) of the clock. That is, they are edge-trigger type. The symbol and truth table of a D-FF are shown in Fig. 1.4. Here, it stores the value at D input at the rising edge of the clock and outputs it at Q-output.

1.2.2 Setup Time and Hold Time

A CMOS D-FF has a master–slave structure consisting of two latch (loop) circuits, each of which uses a couple of transfer gates and inverters (NOT gates), as shown in Fig. 1.5. A transfer gate takes the role of a switch, and it changes to on/off according to CLK. The front-end latch stores the input with the inverse of the clock in order to avoid the hazard appearing just after the change of the clock. The operation of a D-FF is shown in Fig. 1.6.

When CLK = 0 (master operating), the D input is stored into the front-end latch, and the back-end latch holds the data of the previous cycle. Since the transfer gate



D	CLK	Q	$\bar{Q}$
X	L	Qn	$\overline{Qn}$
L	↑	L	H
H	↑	H	L
X	H	Qn	$\overline{Qn}$
X	↓	Qn	$\overline{Qn}$

X：Don’t care
(Both L and H are OK)

↑：L → H
(Up edge)

↓：H → L
(Down edge)

Fig. 1.4 D-Flip-flop

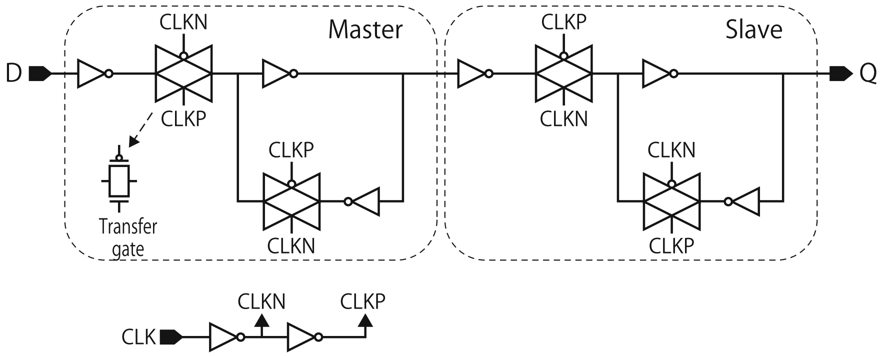


Fig. 1.5 Master–slave D-Flip-flop

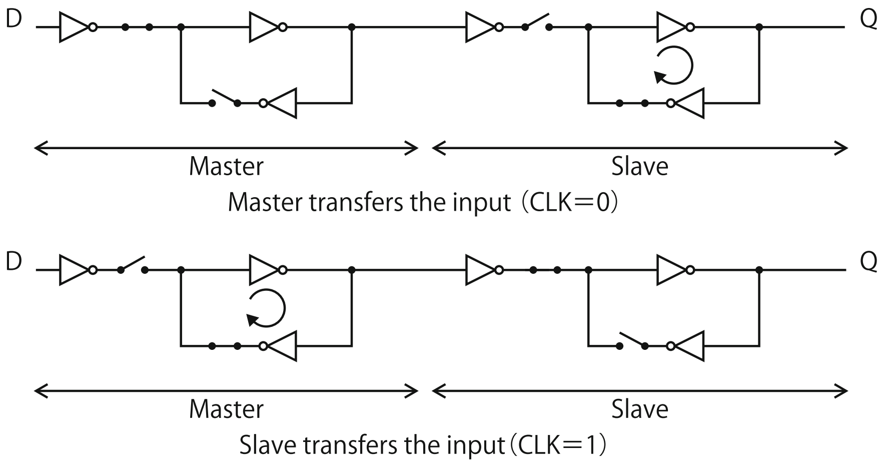

D
Q
Master
Slave
Master transfers the input（CLK=0）
D
Q
Master
Slave
Slave transfers the input（CLK=1）

Fig. 1.6 Operation of master–slave D-Flip-flop

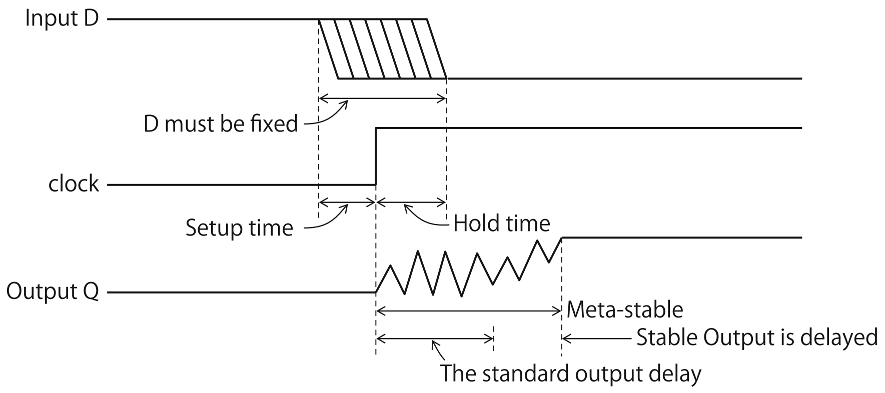


Fig. 1.7 Setup time and hold time

connecting the front-end and back-end is cut off, the signal is not propagated. When CLK = 1 (slave operating), the data stored in the front-end is transferred to the back-end. At that time, the signal from D input is isolated. If the data is not well propagated between both inverters of the front-end loop when CLK becomes 1, the signal may become unstable, taking an intermediate level called meta-stable, as shown in Fig. 1.7. Since the meta-stable continues longer than the delay time of a gate, the data might be stored incorrectly. To prevent this, the restriction of setup time must be satisfied.

Also if the D input is changed just after CLK = 1 and the gate at the D input is cut off, illegal data can be stored or unstable state can occur. In order to avoid it, the restriction of hold time must also be satisfied.

For all the FFs in an FPGA, a timing limitation such as setup time and hold time should be defined for correct operation.

1.2.3 Timing Analysis

Translating register-transfer level (RTL) description in hardware description language (HDL) into a netlist (wiring information between gates) is called logic synthesis. The design step for fitting circuits of the netlist into an FPGA implementation is called "place & route." In an FPGA, an array of predefined circuits and interconnections between them are provided on a chip. The FPGA design stages fix where the circuits translated by the synthesis are located and how to connect them.

In order to verify the correct operation of the designed circuits, not only the function (logic) must be ensured, but also the timing constraints have to be satisfied. In the design of FPGAs, the circuits must be evaluated through the logic synthesis and the place & route. The correctness of the logic is verified by RTL simulations. Since dynamic timing analysis by post place & route simulations with delay requires a large amount of computation time, static timing analysis (STA) is used instead. STA can be executed only with a netlist, and comprehensive verification can be done. Moreover, since it basically traces the circuits only once, the execution speed of the STA is high. It is commonly used in other EDA tools besides FPGAs, to certify whether the design works at a required speed to cope with recent increasing size of target circuits.

Timing analysis includes setup and hold time analysis for timing verification. It verifies whether the delay of the design implemented on FPGA satisfies the timing restrictions. Wiring delay depends on the mapping and routing of the design to the resource of the FPGA, that is, the compilation result of the place & route tool. The design is relatively easy if the performance and number of gates of the target FPGA are large enough, but if the size of the design uses almost all of its resources, the place & route can require a considerable amount of time. The delay of the elements and interconnections of all paths must be checked including the timing margin so as to certify whether the setup time and hold time are satisfied.

1.2.4 Single-Clock Synchronous Circuits

Since FPGAs have a large flexibility in place & route, synchronous circuits are widely used; thus, the target of STA is focused on synchronous circuits. Although the STA is fast, the target circuits can have a certain limitation. That is, the start point and the end point of the delay analysis must be a FF with the same clock input, and the delay between them is accumulated. The transient time of the signal is different since the wiring delay is not the same. Thus, an FPGA design receives all input data at FFs and outputs all data through FFs, as shown in Fig. 1.8. In other words, the system's circuits work with the same edge of the same clock. Inverse clock or reverse edge is basically not allowed, and such a single-clock system is recommended.

The precondition of the synchronous design is to deliver the clock to all FFs at the same timing. The wiring length of real clock signals is often long, and so the

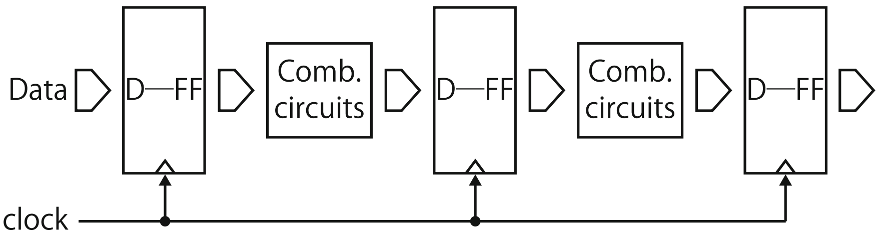


Fig. 1.8 Single-clock system

wiring delay becomes large. Also, the fan-out influences the delay time. Because of their influence, clock timing is slightly different for each FF. This effect is called the clock skew. Jitter is a fluctuation of the clock edge by the variance of the oscillator or distortion of the wave. In order to deliver the clock at the same time, such skew or jitter must be managed under a certain bound.

The clock skew influences the cycle time as well as the delay of logic gates. That is, the most important step in integrated circuits is the clock tree design. In the case of FPGAs, the hierarchical clock tree is already embedded with global buffers providing a high drive capability in the chip to distribute a clock to all FFs, and thus, a low skew clock distribution can easily be achieved. Compared with ASIC designs, in FPGAs, the design step for clock distribution is easier.

1.3 Position and History of FPGAs

Here, the position of FPGA in the logic devices is introduced, and then about 30 years of history of development are reviewed [3, 4].

1.3.1 The Position of FPGA

Logic devices are classified into standard logic devices and custom ICs, as shown in Fig. 1.9. In general, the performance (operational speed), density of integration (the number of gates), and flexibility of given design are advantageous for devices close to custom ICs. On the other hand, non-recurring engineering (NRE) cost for IC designs becomes high and the turnaround time (TAT) from an order to its delivery becomes longer.

Custom ICs are classified into full-custom and semi-custom ICs. The former uses cells designed from scratch, and the latter uses standard cells. Semi-custom ICs are further classified into various types depending on how the NRE cost and TAT are reduced. A cell-based ASIC uses a standard cell library. On the other hand, a gate array uses a master-slice consisting of an array of standard cells, and only steps for

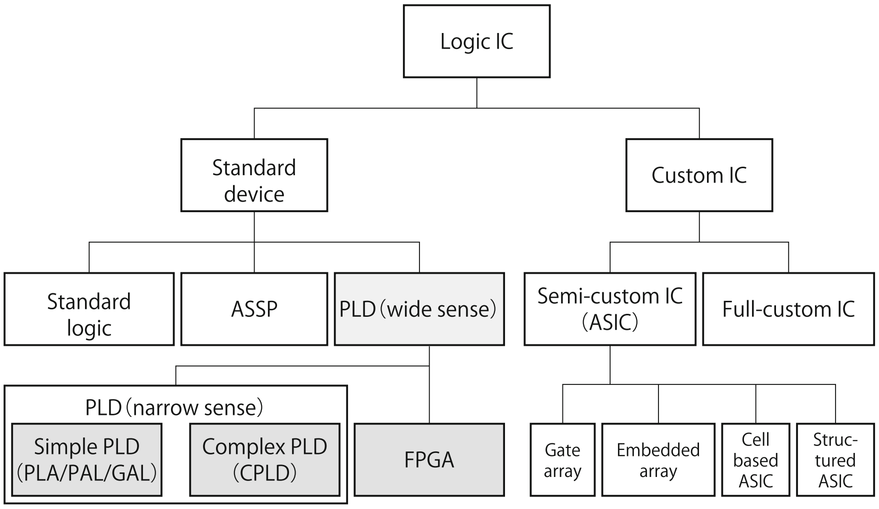


Fig. 1.9 FPGA position in semiconductor devices

wiring follow. An embedded array is a compromise method of cell-based and gate array. The structured ASIC includes standard functional blocks such as SRAM and PLL with a gate array part so as to minimize the design cost. They focus on reducing the NRE cost and shortening the TAT.

Unlike application-specific standard parts (ASSPs), a programmable logic device (PLD) can realize various logic circuits depending on a user program. PLDs have been widely developed by introducing the properties of field programming and freedom of reconfiguring. An FPGA is a PLD which combines multiple logic blocks in the device for a high degree of programming. Since it has a gate array like a structure, it is called field-programmable gate array. FPGA can be mass-produced with a blank (initial) state. So, it can be treated as a standard device from semiconductor vendors, but it can also be considered as an easy-made ASIC with a small NRE cost, and without any mask fee.

More than 40 companies have tried to join the FPGA/PLD industry so far. Here, the history is introduced for each of the era shown in Tables 1.3 and 1.4.

1970s (The Era of FPLA and PAL)

The PLDs started from a programmable AND-OR array with a similar structure to a programmable read only memory (PROM).

The circuit information was stored in memory elements. In 1975, Signetics Co. (became later Philips, and now it is now known as NXP Semiconductors) sold a fuse-based programmable field-programmable logic array (FPLA). Then NMI Co. announced programmable array logic (PAL) which used a simpler structure but

Table 1.3 History of FPGA (1)

Age	Max. num of gates	Represented devices	Features	Companies
1970s	10s–100	Field-programmable logic array (FPLA)	User-programmable, fuse-type, one-time	Signetics (join to Philips, now NXP Semiconductors)
		Programmable array logic (PAL)	Fixed OR-array, high-speed, bipolar, one-time	NMI (join to Vantis, now Lattice Semiconductors)
1980s	100s	Genetic array logic (GAL)	Low-power CMOS electric erasable EEPROM	Lattice
	100s–1000s	FPLA (field-programmable logic array)	Array of CLB interconnect I/O cells are programmable	Xilinx
		Complex programmable gate array (CPLD)	Multiple AND-OR Arrays, high density, high capacity and high-speed	Altera, AMD Lattice
		Anti-fuse FPGA	High-speed, non-volatile but one-time	Actel, Quick Logic
1990s	1000s–Millinon's	SRAM-based FPGA	New products to glowing SRAM-based FPGA (Flex, ORCA, VF1, AT40K families)	Altera, AT & T (Lucent), AMD (Vantis, Lattice) Atmel
		Flash-based FPGA	Non-volatile electrically re-programmable	GateField
		BiCMOS FPGA	High-speed ECL using BiCMOS FPGA(DL5000 family)	DynaChip

achieved high-speed operation using bipolar circuits. PAL was widely used by taking a fixed OR array and bipolar PMOS. On the other hand, it consumed a large amount of power, and the erase and re-program were not allowed.

1980s

(1) The appearance of GAL, EPLD, and FPGA:
In 1989, low-power and erase/re-programmable CMOS EPROM-/EEPROM-

Table 1.4 History of FPGA (2)

Age	Max. num of gates	Represented devices	Features	Companies
2000s	1 Million–15 millions	Million-gate FPGA, SoPD (System on Prog. Device)	Processor-core HardIP, SoftIP, Multi-input LB, Hi-speed I/F, Multi-platform	Altera, Xilinx
		Startup vendors' FPGA Ultra-low-power FPGA, High-Speed ASYNC FPGA, Dynamic Reconf. FPGA, A large-scale FPGA, Monolisic 3D FPGA	Low leak process or power gating, Data tokens transfers, Virtually 3D DRP tech., Scalable wire structure, Amorphas Si TFT techniques	Silicon Blue, Achronix, Tabula, Abound Logic, Tier Logic
2010s	20 Millions (28 nm)–50 millions (20 nm)	28 nm gen. FPGA 20 nm gen. FPGA 16/14 nm gen. FPGA, New gen. SoPD (SoC FPGA), Dynamic PR FPGA, 3D (2.5D), FPGA for Automobile Optical FPGA	TSMC 28 nm, 20 nm,16 nm FIN FET Intel's 14 nm FIN FET, ARM embedded Zynq, Cyclone V SoC Standard support of PR TSV, SiP AEC-Q100 standard ISO-26262 standard Vivado HLS OpenCL	Altera, Xilinx
		Oligopoly	Withdraw of Quicklogic and Atmel. Termination of new FPGA vendors. Frequent M & A	Big 4 vendors Xilinx, Altera Lattice, Actel
		Industry consolidation	Data center, IoT Big data analysis, machine learning, network virtualization, high-performance computing	Microsemi acquired Actel Lattice acquired Silicon Blue Intel acquired Altera

based PLDs were pushed into the market from various vendors. In this era, Japanese semiconductor companies grew rapidly using DRAM technologies, while US traditional big vendors were relatively in depression. Thus, the leading companies were mostly newly developed US venture companies. Various PLD architectures including Lattice's (established in 1983) generic array logic (GAL) and Altera's erasable PLD (EPLD) were developed, and especially GAL was popularly used. It was upper compatible of PAL with the fixed OR array, and a CMOS-based EEPROM was adopted as a programmable element. PLDs with a single AND-OR array such as GAL, FPLA, and PAL, described before, are called simple PLD (SPLD). Their number of gates is about 10s–100s. Advances in semiconductor technologies allowed to implement more gates than for GALs, since increasing the size of a single AND-OR array was not efficient. So, as a flexible large PLD, FPGA and CPLD were introduced.

Xilinx (established in 1984), the first to design FPGAs, was a venture company established by Ross H. Freeman and Bernard V. Vonderschmitt who had spun out from Zilog. Freeman adopted a basic logic cell with a combination of 4-input 1-output LUT and FF and commercialized a practical FPGA (XC2064 series) based on CMOS SRAM technologies. William S. Carter, who joined a little later, invented a more efficient interconnection method to connect logic cells. Their innovations are known as famous patents in FPGA: Freeman's patent and Carter's patent. Ross H.Freeman was included to the US National Inventors Hall of Fame in 2009 for his innovation with FPGAs. Xilinx's FPGA (the product name was then LCA) was highly flexible where erase/re-programming can be done by using CMOS SRAM technology, and its power consumption was low. Based on the advanced research of Petri-net at the Massachusetts Institute of Technology (MIT), Concurrent Logic (now Atmel) commercialized an FPGA with a partial reconfigurable capability. Also, based on the research on virtual computer at Edinburgh University, Algotronix (now part of Xilinx) announced a flexible partial reconfigurable FPGA. The former was Atmel's AT6000, and the latter was Xilinx's XC6200. They are the origins of the dynamically reconfigurable FPGAs.

(2) The second half of the 80s (Appearance of anti-fuse FPGAs and CPLD):
In the latter half of the 1980s, in order to accelerate the implementation density and operational speed, anti-fuse FPGAs, which do not allow erase/re-program, appeared. On the other hand, since the early FPGAs could not achieve the desired performance, other structures of large-scale PLDs were investigated. Altera, AMD, and Lattice, which had produced AND-OR array PLD, developed a large-scale PLD by combining multiple blocks of AND-OR PLDs. They were called complex PLD (CPLD) later. While their flexibility and degree of integration could not compete with FPGAs, CPLDs had the advantage of high-speed design, and re-writable non-volatile memory devices could be easily introduced. Thus, CPLD was a representative of large-scale PLDs comparable to FPGAs until the early 1990s. However, from the late 1990s, since the degree of integrity and speed of SRAM-based FPGAs were improved rapidly, CPLDs started to be considered as economical small devices.

(3) Venture companies until the 80s:
FPGA industry has been mainly driven by various venture companies. Xilinx, which first commercialized FPGAs, was established in 1984. Altera and Lattice were established almost the same year SPLDs were commercialized, and then entered the FPGA industry. Actel is also a venture company established slightly later. They had grown as the big-four vendors in the FPGA industry. QuickLogic appeared later, and these five vendors lead the industry. From major semiconductor companies, only AT&T (former Lucent and Agere, whose FPGA project

was sold to Lattice), and Motorola (Freescale) entered the industry. AT&T was the second source of Xilinx, and Motorola developed products with a license from Pilkington. Consequently, there were no major semiconductor companies which developed FPGAs from scratch. TI and Matsushita (now Panasonic) tried to enter the FPGA industry in cooperation with Actel. Infineon and Rhom started FPGA business with Zycad (Gatefield, later); however, all of them have withdrawn from this initiative.

(4) Japanese semiconductor vendors and major semiconductor vendors:
Venture companies established in the 80s such as Lattice, Altera, Xilinx, and Actel are all fabless maker, meaning that they have no facility for producing semiconductors. Therefore, they relied their fabrication on Japanese semiconductor vendors; for example, Xilinx and Lattice relied on Seiko Epson, and Altera relied on Sharp. Actel had a comprehensive contract with TI and Matsushita including fabrication. In 1990s, GateField, which developed flash FPGAs, had a comprehensive contract with Rohm. However, recently, most FPGAs are produced by Taiwan semiconductor companies such as TSMC and UMC which provide an advanced CMOS technology. Since Japanese major semiconductor vendors focused on DRAM as a standard product and on gate arrays as custom products, they had no intention to enter the PLD industry.
US major semiconductor companies such as TI and National Semiconductor, which focused on logic LSI and memory ICs, had already been part of the market of bipolar AND-OR array. They also tried to produce CMOS EPROM- or EEPROM-based PLDs. However, they could not compete against the aggressive venture companies which developed new architectures, and most of them ceased their activities in the PLD industry. Although AMD purchased MMI in 1987 and aggressively developed new CPLD architectures, it split the activity to Vantis and sold it to Lattice in 1999 in order to concentrate on CPU business.

1990s

(1) Increasing the size of FPGAs:
In the 1990s, both Xilinx and Altera increased the size (gate number) of their FPGAs by improving and extending their XC4000 and FLEX architectures. The size was increased from 1000s to 10,000 in the early 1990s and reached to a hundred thousand in the late 1990s. A large rapid prototyping platform using large-scale FPGAs, as shown in Fig. 1.10, was then developed. The FPGA industry grew up rapidly, and AT&T, Motorola, and Vantis entered SRAM-based FPGAs in these years. In Japan, Kawasaki Steel, NTT, and Toshiba tried to produce their own devices, but eventually products were never released.
It is said that some vendors gave up the production because of the risk of conflict with Xilinx's basic patents (Freeman's patent and Carter's patent). Regarding Altera's PLD products (FLEX family), there has been a long dispute whether they infringe Xilinx's patents. The case was settled in 2001, and after that, Altera

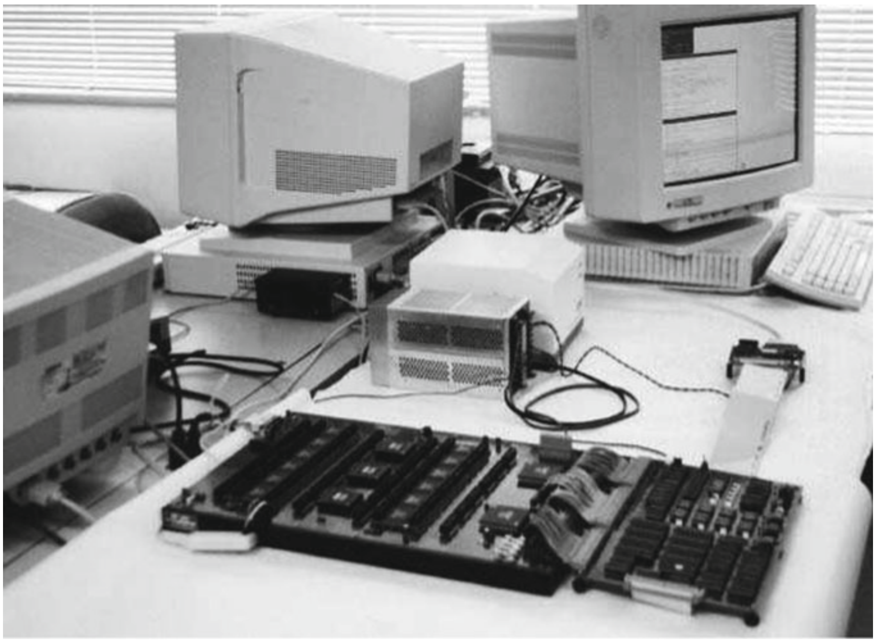


Fig. 1.10 A rapid prototype using 12 FPGAs

was able to start using the word "FPGA," too. Some novel FPGAs appeared in the late 90s. For example, GateField (currently acquired by Actel then Microsemi) announced FPGAs with non-volatile yet erase/re-writable flash memory, and DynaChip commercialized high-performance FPGAs with ECL logic using BiC-MOS process. After the late 1990s, the degree of integration and operational speed of FPGAs rapidly increased, and the difference with CPLDs widened. From that era, FPGAs became a representative device of PLD. On the other hand, since the performance gap between semi-custom LSIs such as gate array or cell-based ICs has been drastically reduced, FPGAs expanded into the semi-custom (especially gate array) market.

Through the 1990s, general-purpose FPGAs pursued their growth, and the mixed integration of MPUs and DSPs was an inevitable result. In 1995, Altera's FLEX10K integrated memory blocks to expand its application, and phase-locked loop (PLL) to manage high-speed clock signals was also provided. From this era, FPGAs were mass-produced and widely spread. In 1997, the logic size reached 250,000 gates and the operational speed increased from 50 to 100 MHz. In 1999, Xilinx announced an FPGA with a new architecture called Virtex-E, and Altera announced the APEX20K for the coming million-gate era.

(2) New companies in the 1990s:
In the early 90s, a few companies including Crosspoint, DynaChip (Dyna Logic), and Zycad (Gatefield) entered the industry. Zycad had had a certain experience

as an EDA vendor based on logic emulators, but sold this project later. In this era, four major leading companies such as Xilinx, Altera, Actel, and Quicklogic grew steadily. Crosspoint and DynaChip canceled their projects. Crosspoint was established in 1991, and it was the last established vendor of anti-fuse FPGAs. In 1991, it applied the basic patents and announced its products, but closed in 1996. Crosspoint FPGA used amorphous silicon anti-fuse for through-holes between aluminum layers to form user-programmable gate array. The finest logic cells with a pair of transistors were used to realize similar density of integrity as gate arrays. This type of programmable devices never appeared again. In the late 1990s, Xilinx and Altera became so strong that there were almost no new FPGA vendors. Instead, there were a lot of venture companies for dynamically reconfigurable coarse-grained reconfigurable devices. However, most of them have vanished, and none has achieved a big success.

2000s

(1) Million-gate era, and becoming a system LSI:
In the 2000s, FPGA became a system LSI. Altera's soft-core processor Nios is a processor IP supported by the vendor. Altera also announced "Excalibur," the first FPGA with hard-core processor (Fig. 1.11). Excalibur integrated an ARM

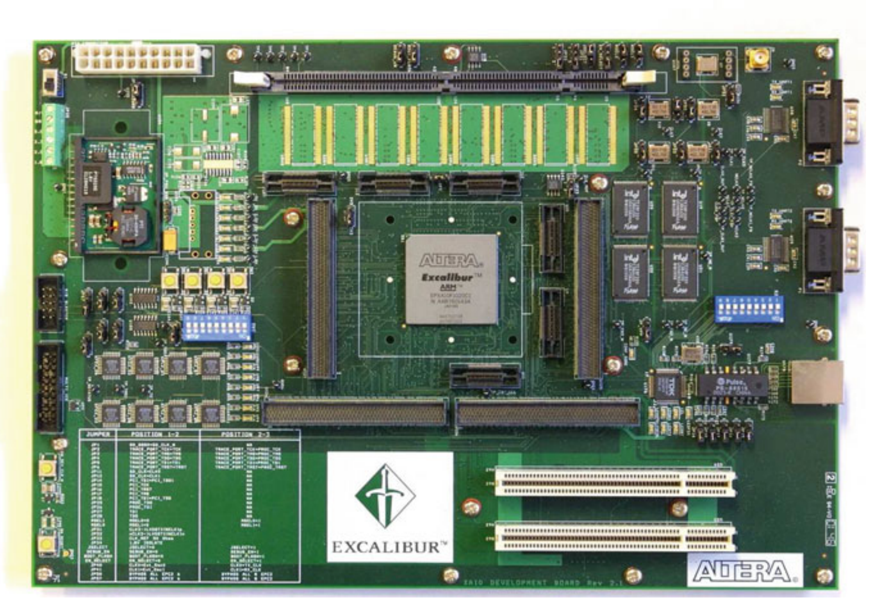


Fig. 1.11 First SoC FPGA excalibur

processor (ARM922 with peripherals) and an FPGA into a chip. On the other hand, Xilinx supported MicroBlaze as a soft-core processor and commercialized a PowerPC embedded FPGA core (Virtex II Pro). For a system LSI, a high-performance interface is important. So, FPGAs also provided serializer–deserializer (SERDES) and low-voltage differential signal (LVDS) for high-speed serial data transfer. In order to cope with the computational performance requirements for image processing, dedicated computation blocks of multipliers or multipliers + adders were embedded. Many-input logic blocks with high performance and density of integration were also introduced. However, such hard IPs are wasteful if unused, so multi-platform (or subfamily) with various product lineups for different target application were provided. For example, Altera introduced new products every two years: Stratix (2002, 130 nm), Stratix II (2004, 90 nm), Stratix III (2006, 65 nm), and Stratix IV (2008, 40 nm). In 1995, FLEX10K supported 100,000 gates and worked with a maximum of 100 MHz clock. In 2009, Stratix IV E had 8400,000 gates + DSP blocks corresponding to 1,5000,000 gates and was operational with a 600-MHz internal clock. The number of gates was multiplied 150 times. In the case of Xilinx, Virtex II Pro (2002, 130 nm) changed every two years with Virtex-4 (2004, 90 nm), Virtex-5 (2006, 65 nm), and Virtex-6 (2009, 49 nm). During that era, logic IC process evolved every 2 years and FPGAs quickly followed that trend.

(2) New vendors in the 2000s:
Two basic patents, Freeman's patent and Carter's patent which had been a great barrier for newcomers, expired in 2004 and 2006, respectively. Some new vendors then took the opportunity and entered the FPGA industry. SiliconBlue Technologies, Achronix Semiconductor, Tabula, Abound Logic (former M2000), and Tier Logic entered at that time.
SiliconBlue focused on the power consumption which is the weak point of conventional FPGAs and announced ultra-low-power iCE65 family for embedded application using TSMC 65 nm low leak process. It is an SRAM-based FPGA with embedded non-volatile configuration memory, achieving an operational power divided by 7 and a standby power by about 1000. Achronix commercialized high-speed FPGAs, the "Speedster family," based on the research of Cornel University, USA. The most important characteristic is the token passing mechanism with asynchronous circuits. A data token, which takes the role of data and clock in a common FPGA, is passed by handshaking. The first product SPD60 using TSMC 65 nm process achieved almost three times the throughput of a common FPGA. The maximum throughput was 1.5 GHz.
Tabula's FPGA reduced the cost by dynamic reconfiguration using the same logic cells for multiple functions. ABAX series by Tabula generates a multiple frequency clock from the system clock, and uses it both for the internal logic and dynamic configuration. By time multiplexing a fixed programmable logic region, the effective logic area can be increased. Tabula introduced a new "time" dimension into two-dimensional chips and called their products three-dimensional FPGAs. Abound Logic announced "Rapter" with crossbar switches and a scalable architecture, but closed in 2010. Tier Logic developed a novel

3D-FPGA whose SRAM configuration is formed with amorphous silicon TFT technology on the CMOS circuits in collaboration with Toshiba; however, due to fund shortage, the project was terminated.

2010s

(1) Technology advances and new trends:

In 2010, Xilinx and Altera started the shipping of 28 nm generation FPGAs that can be considered to be more advantageous than ASIC chips. Both companies added a mid-range product line to their high-end and low-end lines. For example, Xilinx changed its fabrication from UMC to TSMC both in Taiwan, and all products of the Xilinx 7 series (High-end Virtex-7, mid-range Kintex-7, and low-end Artix-7) are fabricated with a 28 nm process for low power and high degree of functionality. At that time, both Xilinx and Altera used TSMC for their foundries. The followings are technology trends in 28 nm generation FPGA.

(a) The trend of new generation SoC:
Around 2000, both Xilinx and Altera shipped the first generation of SoC products with FPGA, but their lifetime was relatively short. On the other hand, FPGAs with soft-core processors have been widely used. The demands for embedded hardware cores grew, and by using advanced technologies, CPU cores with enough performance capable of fulfilling such demands could be embedded. This promoted FPGAs for SoC, providing a 32bit ARM processor and enhanced I/O. They are called SoC FPGA, programmable SoC, or SoPD (System on Programmable Device). For example, Xilinx introduced a new family Zynq-7000 which integrates an ARM Cortex-A9 MPCore and the 28 nm 7 series FPGA programmable logic. Altera's new product, "SoC FPGA," integrated dual-core ARM Cortex-A9 MPCore and FPGA fabric into a device. A representative example is the Cyclone V SoC.

(b) Partial reconfiguration:
Partial reconfiguration is a functionality which reconfigures a part of an FPGA, while others are still under operation. The functions can be updated without stopping the system. Xilinx started to support this function in their high-end FPGA devices from Virtex-4 with its EDA tool (after ISE12). Altera also started to support this feature from Stratix V. Since the major two vendors started to support partial reconfiguration in their tool, this technique is becoming widely spread.

(c) 3D-FPGA (2.5D-FPGA):
Xilinx shipped multi-chip products placing multiple FPGAs on a silicon interposer with stacked silicon interconnect. It is called the 2.5D implementation. Unlike the 3D implementation of multiple chips with TSVs, whose cost tends to be high, 2.5D can mount chips without TSVs. Virtex-7 2000 T with TSMC 28 nm HPL process integrated 200 million logic cells corre-

sponding to the largest ASIC with 68 billion transistors and 20,000,000 gates.

(d) FPGAs for automobiles:
Xilinx extended the Artix-7 FPGA and shipped XA Artix-7 FPGA which fully satisfies the AEC-Q100 standard for automobile. XA Artix-7 complements the programmable SoC XA Zynq-7000. Furthermore, the authentication of third-party tools is undergoing to satisfy the ISO-26262 standard. Altera and Lattice also tackle automobile solutions.

(e) C language design environment:
Recently, C language design environments have become popular in FPGA design. Xilinx Vivado HLS can translate the hardware description in C, C++, and System C to devices directly without RTL description. It can be used both from ISE and Vivado. On the other hand, Altera aggressively introduces the OpenCL environment. It is a C-base programming language running on various platforms: CPU, GPU, DSP, and FPGA and allows Altera's FPGAs to be used as hardware accelerators.

(f) Others:
In order to expand the I/O bandwidth of FPGAs with optical interfaces, optical FPGAs have been introduced. Radiation-hardened FPGAs are also being developed.

(2) The road map of process technology for FPGA:
After the 28 nm generation, Xilinx presented the 20 nm FPGA Kintex UltraScale, and the Virtex UltraScale provided a new architecture. The largest series Virtex UltraScale is corresponding to an ASIC with 50,000,000 gates. All of UltraScale devices use TSMC 20 nm process, but high-end Virtex UltraScale use the TSMC 16 nm FinFET. On the other hand, Altera shipped the Arria 10 for next-generation FPGAs, the "Generation 10" devices, and announced Stratix 10 FPGAs. They are all SoCs with embedded processors. Generation 10 devices are fabricated by Intel's 14 nm generation FinFET and TSMC 20 nm technologies. The high-end Stratix 10 can work at 1 GHz clock.

Logic IC process advances to the next generation every 2 years. The Intel processor is a representative example of such evolution; however, since the 2000s, FPGAs mostly caught up with that pace. On the other hand, ASICs followed the advances until the early 2000 s and stalled for about 10 years at 130-90 nm, except for some special applications such as game machines. As shown in Fig. 1.12, FPGAs have been fabricated along with the technology road map. The pace is more than that of general-purpose processors. FPGA will use 28 nm, 20 nm, and 16/14 nm processes and will get a similar competitive performance to ASIC with 130 nm, 90 nm, or 65 nm, two or three generations behind.

(3) Oligopoly and industry restructuring:
In 2010, oligopoly continued in the FPGA industry. Major FPGA vendors, Xilinx and Altera, occupy more than 80% of the shares, and other parts are shared between Lattice and Actel. Actel, at the fourth place in the industry, was acquired

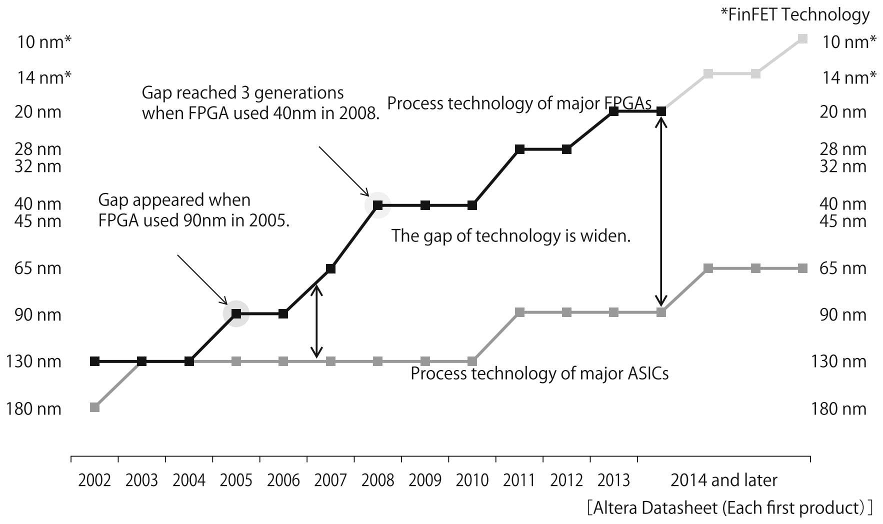


Fig. 1.12 Process road map of FPGA and ASIC

by Microsemi in 2010, and ships flash and anti-fuse non-volatile FPGAs as Microsemi FPGA.
Among the FPGA vendors established in the 1980s, Quicklogic focused on anti-fuse FPGAs, but it changed its strategy and has produced customer specific standard products (CSSP) for specific custom fields. CSSP is not an all-programmable product, but only a part of the chip is programmable. On the other parts, a lot of standard interface circuits are mounted to cope with customers' needs. Also, Atmel's FPGA technology is mostly combined with their AVR controllers, and they withdrew from FPGA industry. Among the new FPGA vendors established in the 2000s, SiliconBlue was acquired by Lattice, and Lattice introduced a new line of the iCE40 family with a 40 nm process. Tabula which proposed a low-cost dynamic reconfiguration finished its projects in March 2015. On the other hand, Achronix produced the Speedster22i FPGA family with Intel's 22 nm tri-gate process technology in 2015.

In the spring of 2016, the semiconductor industry entered a great restructuring era, and large-scale M&As have been carried out. The FPGA industry was naturally involved. Intel acquired the major FPGA vendor Altera in June 2015. The total operation reached 167 billion dollars. It was more than the amount of yearly sales of Altera, the largest scale in the FPGA history. Intel aims to occupy the market of data center and IoT by the integration of processors and FPGAs. For this purpose, Intel selected Altera's FPGA as an essential technology.

On the other hand, Qualcomm and Xilinx announced a strategic cooperation contract. Both companies support solutions for data center with ARM processors for servers and FPGA technologies. They focus on the basic technology of cloud computing including big data analysis and data storage. Furthermore, Xilinx announced a multi-year strategic cooperation contract with IBM. By combining Xilinx FPGAs with IBM Power Systems and using the combination as an accelerator for specific applications, a highly energy efficient data center can be produced. Such systems are suitable for machine learning, network virtualization, high-performance computing, and big data analysis. They try to compete against the “Catapult” of Microsoft (in collaboration with Altera and Intel) with such strategic cooperation [5].

References

1. V. Betz, J. Rose, A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs, (Kluwer Academic Publishers 1999)
2. Z. Kohavi, *Switching and Finite Automata Theory*, 2nd edn. McGraw-Hill (1978)
3. S.D. Brown, R.J. Francis, J. Rose, Z.G. Vranesic, *Field-Programmable Gate Array*, (Kluwer Academic Publishers 1992)
4. S.M. Trimberger, *Field-Programmable Gate Array Technology*, (Kluwer Academic Publishers 1994)
5. A. Putnam et al., A reconfigurable fabric for accelerating large-scale datacenter services, in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, (2014), pp. 13–24

Chapter 2
What Is an FPGA?

Masahiro Iida

Abstract An FPGA is a programmable logic device, which is a type of integrated circuits that can be used to implement any digital circuit, and so the key technique is how to make programmable 'hardware' devices. After the brief introduction of the structure of traditional island-style FPGAs, the technology for programmable devices: antifuse, EEPROM, and SRAM is explained in detail. Then, logic circuits representation with product term, lookup table (LUT), and MUX-type basic logic element are introduced.

Keywords Island-style FPGAs · Antifuse · EEPROM · SRAM product term
Lookup Table (LUT) · MUX-type basic logic element

2.1 Components of an FPGA

An FPGA is a programmable logic device, which is a type of integrated circuits that can be used to implement any digital circuit. The name of FPGA originates from the fact that a user can use a GATE ARRAY that is PROGRAMMABLE on the FIELD of any workplace. However, the structure of an FPGA is not such as spreading gates all over a silicon die.

Figure 2.1 shows the structure of a typical island-style FPGA. The basic part of an FPGA is roughly divided into three parts. The first one consists of the logic elements (the logic block: LB) that realize logic circuits. The second is the input/output elements (the input/output block, IOB) which input and output signals to and from outside. The third is the wiring elements (the switch block, SB, and the connection block, CB) connecting LBs and IOBs. Other than that, there are a clock network, a configuration/scan chain, and a test circuit. Commercial FPGAs also contain circuits

M. Iida (✉)
Kumamoto University, Kumamoto, Japan
e-mail: iida@cs.kumamoto-u.ac.jp

H. Amano (ed.), *Principles and Structures of FPGAs*,
https://doi.org/10.1007/978-981-13-0824-6_2

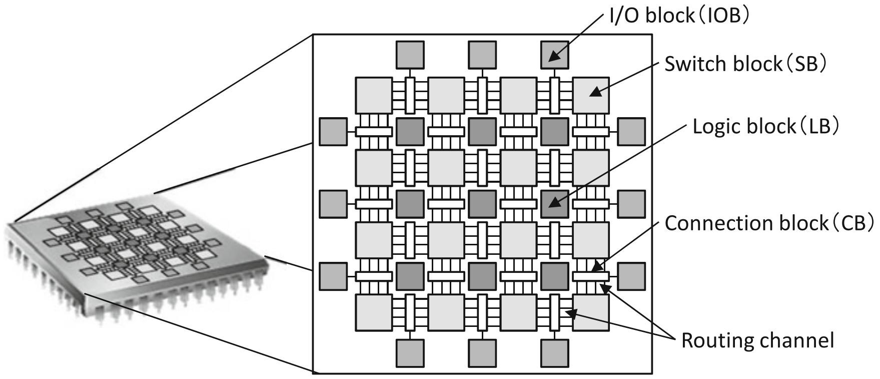


Fig. 2.1 Overview of a traditional island-style FPGA [1, 2]

of specific functions such as processors, block memories, multipliers. The outline of each element is shown below. More details are explained later in Chap. 3.

Logic Element There are three major program logic implementation schemes, such as the lookup table (LUT), the multiplexer (MUX) and the product term logic[1] which is used from the era of generic array logic (GAL). Either method consists of a programmable part that can be used to realize any logic circuit, a circuit that holds logic values such as flip-flop (FF) and selectors.

Input/Output Element It is a block that connects I/O pins and internal wiring elements. It also has some control circuits such as pull-up, pull-down, input/output directions, slew rate, open drain. In addition, it contains a circuit for holding values such as flip-flops. In commercial FPGAs, it several standards are supported, such as LVTTL, PCI, PCIe, and SSTL which are single-ended standard I/Os and LVDS of differential standard I/O.

Wiring Element It consists of wiring channels, connection blocks (CB), and switch blocks (SB) at the connection between logical blocks and between logical blocks and I/O blocks. Besides the island style (shown in Fig. 2.2) arranged in a lattice pattern, there are wiring channels of hierarchical structures, and those constituting H-trees. Each switch is programmable, and it is possible to form any wiring route by using the built-in wiring resources.

Other Elements The logical functions and connection relations of all logical blocks, I/O blocks, switch blocks, and connection blocks are determined by the configuration memory values. The configuration chain is a path to sequentially write the configuration data bits to all configuration memory modules. Basically, the configuration data are serially transferred, and both set and read back are possible. Besides the configuration chain, there are other device-scale networks such

[1]In Boolean logic, a product term is a conjunction of literals, where each literal is either a variable or its negation. A product term logic means an AND-OR array structure.

as the scan path and the clock network. Others include circuits that support LSI testing, embedded circuits for dedicated functions such as embedded processors, block memories, and multipliers.

2.2 Programming Technology

As mentioned above, the circuit on the FPGA is controlled by a *programmable switch*. This programmable switch can be made using various semiconductor technologies. So far, EPROM, EEPROM, flash memory, antifuse, and static memory (SRAM) have been considered. Among these technologies, the flash memory, antifuse, and static memory are three types of programming technologies widely used in modern FPGAs. In this section, they are compared and summarized, regarding their advantages and disadvantages.

2.2.1 Flash Memory

The Principle of Flash Memory The flash memory is a kind of electrically erasable programmable read-only memory (EEPROM), which is classified as a nonvolatile memory. Figure 2.2 shows the structure of the flash memory. Although the flash memory has roughly the same structure as a common MOSFET device, it has a distinctive feature where the transistor has two gates instead of one. The control gate at the top is the same as other MOS transistors, but below there is a floating gate. Normally, this floating gate is formed of a polysilicon film and becomes a floating gate electrode in an insulator (SiO_2) that is not connected to anywhere. Because the floating gate is electrically isolated by its insulating layer, electrons placed on it are trapped until they are removed by another application of electric field.

The flash memory can be classified into two types depending on the writing method. They are of NAND type and NOR type. As a feature, the write of the NAND type is a voltage type requiring a high voltage and the NOR type is a current

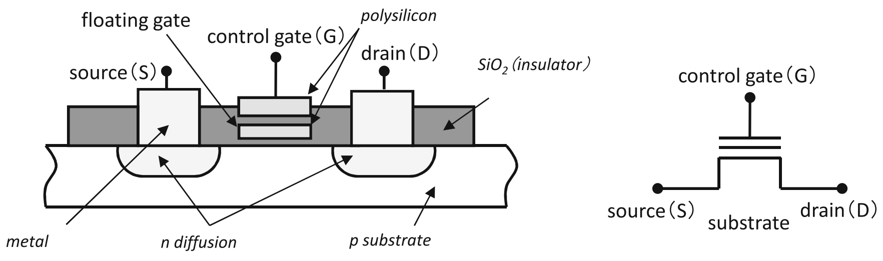


Fig. 2.2 Flash memory structure

type requiring a large current. Hereafter, the principles and operations are explained using the NAND-type flash memory as an example.

In the case where the floating gate is not charged before writing, it is considered as a depletion type in which the current is flowing even at zero bias, as shown in Fig. 2.3a. When the floating gate is charged after writing as shown in (b), it becomes an enhancement type in which there is no current at zero bias in the control gate. By charging the floating gate, the voltage is changed when the current flows, making the state ‘0’ and state ‘1’. If there is an electric charge in the floating gate, current begins to flow even when the voltage applied to the control gate is low voltage (about 1 V); however, in the absence of electric charges, no current flows unless a relatively high voltage (about 5 V) is applied.

When the floating gate is charged, since the electric charge does not have a route to escape, it keeps its state permanently. In order to store electric charges in an unconnected gate, electrons are injected into the floating gate as a tunnel current by applying a high voltage between the drain and the control gate. When erasing, as shown in Fig. 2.3c, by applying a high voltage to the source, electrons in the floating gate are extracted as a tunneling current.

In addition, each bit of a general flash memory can be separately written, but at the time of erasing, it is performed collectively on a block basis.

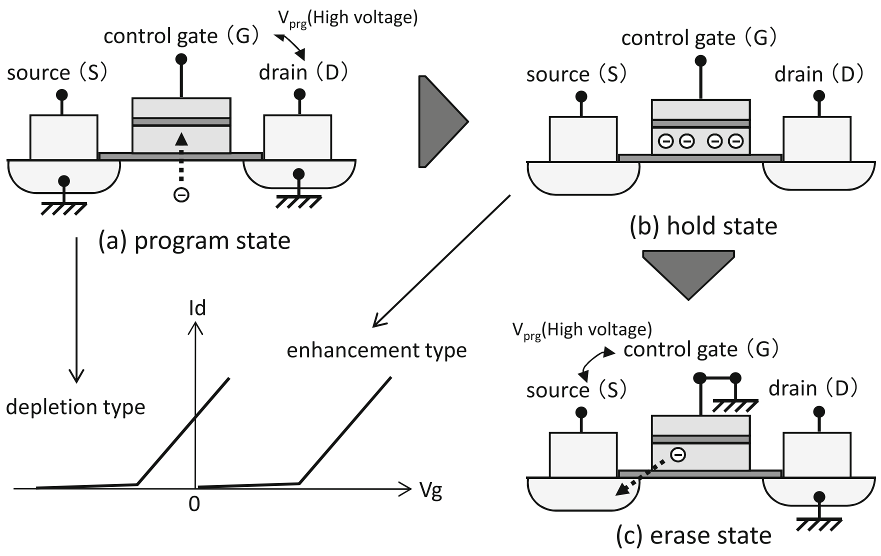


Fig. 2.3 Flash memory principles

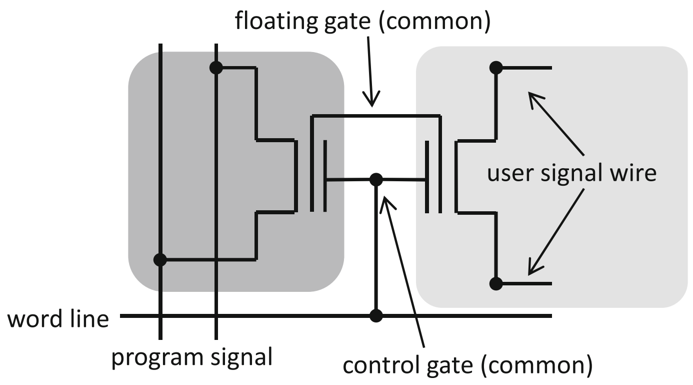


Fig. 2.4 Flash programmable switch

Programmable Switch Using Flash Memory Next, programmable switches using the flash memory in FPGAs are described by taking the Actel's ProASIC series [3–5] as an example.[2]

Figure 2.4 shows the structure of a programmable switch using a flash memory. This switch is made of two transistors: the first one, on the left side, is a small transistor to write/erase the flash memory. The second, on the right side, is a large transistor which acts as a switch to control the connection of the user's circuit implemented on FPGA. The control gate and the floating gate are shared between these two transistors, and the injected electrons within the programming switch directly determine the state of the user's switch. Having dedicated write/erase transistors in this manner not only restricts the connection of switches for users, but also makes programming easier because it is independent of user signals.

Actual programming of NAND-type flash memories is performed using tunneling current as follows [3]. First, the source and drain of the programming transistor are supplied with 5.0 V. Next, when the control gate is supplied with -11.0 V, electrons flow in and the switch turns on. During normal operations, the control gate voltage holds at 2.5 V. By doing so, the potential of the floating gate is maintained approximately at the proper 4.5 V. For the erase operation (switch off), the source and drain of the programming transistor are set to the ground level and the control gate is set to 16.0 V. As a result, the floating gate during normal operations becomes 0 V or less.

Cons and Pros of Programmable Switches Using Flash Memories The advantages of a programmable switch using a flash memory are summarized as follows:

- Nonvolatile;
- Smaller size than SRAM;
- LAPU (Live At Power-UP: Immediate operation after power on) is possible;

[2]The ProASIC series is the first FPGAs using a flash memory and was originally released in 1995 as a product of the Zycad's GateField division. Later in 1997, Zycad changed its firm name to GateField and in 2000 was acquired by Actel, and this series then joined Actel's lineup [6].

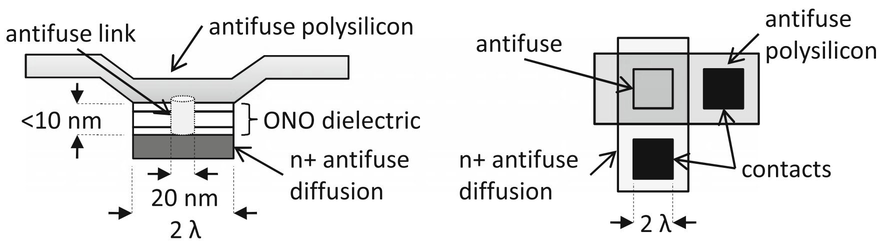


Fig. 2.5 Polysilicon-type structure of PLICE

- Reconfigurability;
- Strong soft error resistance.

The disadvantages are as follows:

- High voltage is required for rewriting;
- CMOS's cutting-edge process cannot be used (flash process is not suitable for miniaturization);
- Restriction on the number of times it can be rewritten[3];
- High on-resistance and load capacity.

2.3 Antifuse Technology

A switch using an antifuse [7] is initially in an open state (insulated). However, it changes to the conducting state when it is burned out by applying a large current (in this case, it is burned to connect). In other words, it is the reason why it is called antifuse, because it acts in an opposite way to a fuse.[4]

Taking the Actel's programmable logic interconnect circuit element (PLICE) [8] and QuickLogic's ViaLink [9, 10] as examples, we take a look at the structure and features of the antifuse switch.

The structure of Actel's antifuse switch PLICE is shown in Fig. 2.5.

PLICE adopts a structure in which polysilicon and n+ diffusion layer are used as conductors and between them an oxide–nitride–oxide (ONO) dielectric is inserted as an insulator. The ONO dielectric has a thickness of 10 nm or less, and it is possible to make connections between the upper layer and the lower layer by applying a voltage of about 10 V and a current of about 5 mA, as a standard. The size of the

[3]Up to 500 times for Actel's ProASIC 3 series [4]. Whether this is enough or not depends on the users and applications.

[4]The fuse is a component that protects a circuit from a current higher than the rated value, to prevent accidents. It normally behaves as a conductor, but by cutting the current path by burning out with its own heat (Joule effect) when the current is over the rating, it protects the target circuit.

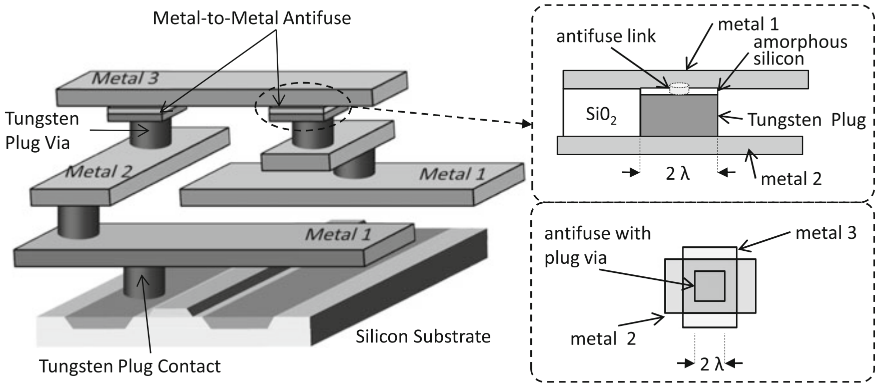


Fig. 2.6 Metal-to-metal-type antifuse structure of ViaLink

antifuse itself is roughly the same as the contact hole.[5] The on-resistance of the ONO dielectric-type antifuse is about 300–500 Ω [1, 7].

On the other hand, the QuickLogic's antifuse switch is also called a metal-to-metal antifuse because it connects layers of wiring. Figure 2.6 shows the structure of QuickLogic's ViaLink. The ViaLink antifuse adopts a structure in which an amorphous silicon layer (insulator) and a tungsten plug (conductor) are placed between the upper and lower metal wires. Like the polysilicon type, the size of the antifuse is approximately the same as that of a contact hole. Also, the amorphous silicon layer exhibits a relatively high resistance until it is programmed and is in an almost insulated state. On the other hand, when program processing is performed by applying a current, the state changes to a low resistance value almost equal to the interconnection between the metal wirings. The on-resistance of ViaLink is roughly 50–80 Ω (standard deviation 10 Ω), and the program current is about 15 mA [1, 7].

Compared to the polysilicon type, there are two advantages of using the metal-to-metal-type antifuse. The first one is its small area since metal wiring can be connected directly. In the polysilicon type, even though the size of the antifuse itself is the same, an additional region for connecting the metal wiring is absolutely necessary. The second point is that the on-resistance of the antifuse is low. For these reasons, the mainly used antifuses now are the metal-to-metal type.

In order to secure the device, it is necessary to take extra efforts such as encryption for static memory-based FPGA because a configuration can be read back. On the other hand, for the antifuse method, since there is no dedicated path at the time of writing, reading by using the right path is impossible given the structure. In order to read the configuration data, it is necessary to perform reverse engineering and to judge the written contents from the state of the antifuse. However, an attempt to reverse engineer a metal-to-metal-type antifuse FPGA by chemical etching will

[5]It is a hole provided to connect the gate and the upper layer wiring on the silicon substrate, or the upper layer and the lower layer of the wiring. Via hole is almost a synonym. This term comes from the PCB terminology.

cause the destruction of the antifuse via the only way to examine the state of each antifuse is to cut in the cross section. However, since it is highly likely that other areas of the chip will be destroyed, it can be said that it is practically impossible to extract the circuit information written in the device. Therefore, the device has a remarkably higher security when compared with the static memory-type FPGA described later.

Pros and Cons of Programmable Switches Using Antifuse The benefits of a programmable switch using an antifuse are summarized as follows:

- Small size;
- Low on-resistance and load capacitance;
- Nonvolatile;
- Reverse engineering is almost impossible;
- Robust against soft errors.

The drawbacks are as follows:

- Cannot be re-programmed;
- In order to carry out the programming, 1–2 transistors per wire are required;
- It needs a special programmer and takes time to program;
- Cannot test write defects;
- The programming yield is not 100%.

2.3.1 Static Memory Technology

Finally, we explain static memories used as programming technology. Figure 2.7 shows the structure of a CMOS-type static memory cell [11]. The diagram on the left is the gate level circuit diagram showing the principle, and the diagram on the right is the transistor level circuit diagram. The static memory consists of a positive feedback loop (flip-flop) composed of two CMOS inverters and two pass transistors (PT). Information is stored in the bistable state (0 and 1) of the flip-flop, and writing is done via PT. The n-MOS type is used for PT.

An ordinary static memory is driven[6] by a word line (connected to the write signal in this figure) that is generated from the address signals and can also be read via PT. Therefore, the high level of the output of the memory cell becomes $V_{DD} - V_{th}$,[7] which is amplified by the sense amplifier and outputted. However, since the FPGA always needs reading, it is always outputted from the flip-flop rather than read through the PT.

[6]A normal static memory reads multiple bits (8 or 16 bits) on a word line determined by an address all at once. At that time, it is also controlled by PT so that it will not collide with data from other words. Here, the term 'drive' means to operate one-word line determined by the address.

[7]V_{DD} stands for Voltage Drain and is the supply voltage. In a CMOS circuit using a field effect transistor (FET), since a power supply is connected to a drain terminal, such a name is used. Vth is the threshold voltage. When the voltage applied to the gate (Gate) terminal exceeds this value, it switches on and off.

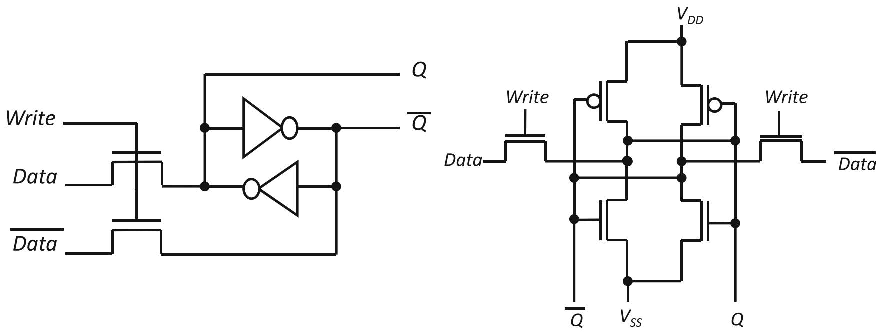


Fig. 2.7 Static memory principles

Many of the FPGAs using a static memory for programmable switches have a lookup table (LUT) in the logic block and use a multiplexer or something similar to switch the connection of the wirings. The lookup table is the memory itself storing the truth table of the logical expression and is composed of a static memory of several bits. On the other hand, a static memory is also used for switching a selector to determine the connection of the multiplexer. Such an FPGA is generally called an SRAM-type FPGA and is currently the mainstream device. The structure of the LUT will be explained in Sect. 2.4.3.

Pros and Cons of Programmable Switches Using Static Memory The advantages of the static memory are as follows:

- Advanced CMOS process can be used;
- Reconfigurability;
- No limit on the number of times of rewriting.

Also, the drawbacks are as follows:

- Memory size is large;
- Volatile;
- Difficult to secure configuration data;
- High sensitivity to soft errors;
- High on-resistance and load capacity.

In this way, the static memory has many disadvantages compared to other programming technologies; however, it overturns all the drawbacks in the single point of '*being able to use CMOS advanced process*.' Now, the static memory-based FPGA is the process driver[8] of advanced CMOS process.

[8]A process driver refers to a product category that leads a semiconductor process. In the past, DRAMs, gate arrays, processors, and so on developed as state-of-the-art processes as products. Currently, high-end processors and FPGAs are at the forefront of miniaturization of semiconductors, and all the latest technologies are being introduced.

2.3.2 Summary of Programming Technology

Table 2.1 compares these the previously explained programming technologies [11].

The antifuse has low power consumption during standby time, and high speed operation is possible thanks to the small on-resistance of the connection switch. Also, since it is difficult to analyze the internal circuit, it is suitable for high confidential use. However, since the circuit is fixed at the time of writing, circuit information cannot be rewritten later. Also, it is difficult to miniaturize, and therefore the degree of integration is low.

On the other hand, since the flash memory is rewritable and nonvolatile, LAPU is possible. Since the static memory constructs one cell with a plurality of transistors, the leak current per cell increases. On the other hand, since the flash memory constitutes one cell with one floating gate transistor, the leakage current is structurally small. In principle, this feature shows that higher integration is possible than static memories, but the actual degree of integration is low. Furthermore, rewriting the circuit information of the flash memory requires much higher energy than rewriting the static memory. In other words, although the power consumption for rewriting is large, this feature also has a secondary effect where the resistance to errors due to radiation is high. As another feature, the flash memory has a drawback on the limited number of rewriting (about 10,000 times). For this reason, it is not suitable for devices that are required frequently or need dynamic reconfiguration.

An FPGA using a static memory operates by externally transferring circuit information when power is turned on. There is no limitation on the number of rewriting of circuit information in the static memory, and it can be rewritten any number of times. Since the most advanced CMOS process can be applied for manufacturing, it is easy to achieve higher integration and higher performance. On the other hand, since the static memory is volatile, circuit information is lost and LAPU cannot be done if the

Table 2.1 Feature comparison of programming technologies

	Flash memory	Antifuse	Static memory
Nonvolatile	Yes	Yes	No
Reconfigurability	Yes	No	Yes
Memory area	Mid (1 Tr.)	Small (none)	Large (6 Tr.)
Process Tech.	FLASH process	CMOS process+Antifuse	CMOS process
ISP[a]	Available	None	Available
Switch resistance (Ω)	500–1,000	20–100	500–1,000
Switch capacitance (fF)	1–2	<1	1–2
Programing yield (%)	100	>90	100
Lifetime	10,000	1	Infinity

[a]In System Programmability, circuit information can be rewritten while it is mounted on an equipment

power supply is cut. Also, since the leakage current is large, the power consumption during standby is large as well. In addition, there are disadvantages such as the risk of errors due to radiation and security risks of stealing circuit information.

2.4 Logic Circuit Representation of FPGA

2.4.1 Circuit Implementation on FPGA

Hereafter, we how a design is implemented on FPGA using the majority vote circuit depicted in Fig. 2.8.[9] It is a circuit that takes the majority vote out of three inputs, and the LED glows when the result is true. In order to realize this, electronic components such as push button switch, resistance, LED, FPGA are necessary. The circuit within the dotted frame in Fig. 2.8 is implemented on FPGA.

Figure 2.9 shows the truth table and *Karnaugh* map of this majority vote circuit with the logical formula after simplification. Since the part to be implemented on FPGA is a logic circuit, it should be simplified so that it occupies less resources; but, a design optimization similar to what is performed for an ASIC is not necessary. Because the logic block of the FPGA adopts the LUT method, an arbitrary logical function, up to the number of inputs, can be implemented. In the case of using the product term method, it is necessary to express it in the product sum standard form.

In this explanation, it is assumed that the number of inputs of the logic block is three. Therefore, the truth table in Fig. 2.9 can be realized with one logic block. Figure 2.10 shows each part used to implement the above logical function on FPGA. The input signals of the logic circuit enter from the I/O pads of the FPGA and are inputted to the logic block through the internal wiring paths. In the logic block, the output is determined based on the above truth table and goes back to the I/O pad again through the wiring route. However, since the output signal needs to turn on the LED outside the FPGA, a buffer is inserted in the output stage to improve the drive capability.

The decomposed circuit shown in Fig. 2.10 is connected inside the FPGA, as shown in Fig. 2.11. The circuit determines the path of a signal line by a switch that can be programmed inside the FPGA and realizes a logic function with a programmable memory, that is, an LUT or something similar.

2.4.2 Logical Expression by Product Term

Here, as an example of a product term where its principle is shown using a programmable logic array (PLA). Figure 2.12 illustrates the schematic structure of the PLA.

[9]Detailed explanations of this circuit are omitted for now in order to focus on the concept of FPGAs. More information will be provided later in the chapter.

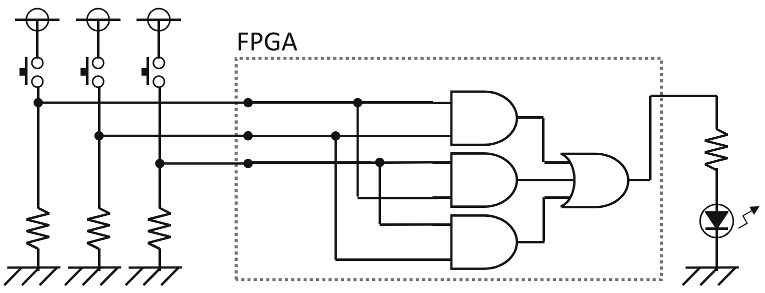
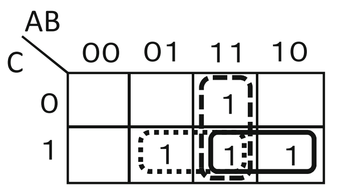


Fig. 2.8 Example of a majority vote circuit

Truth table

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a) Truth table of this majority vote circuit



$$M = AB + AC + BC$$

(b) Karnaugh map of this majority vote circuit

Fig. 2.9 Truth table and Karnaugh map of this majority vote circuit

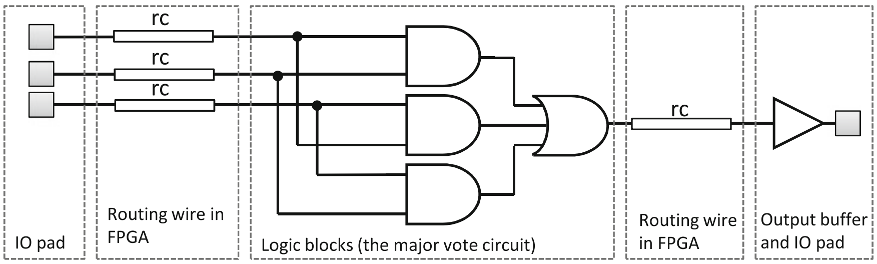


Fig. 2.10 Mapping of a majority vote circuit for FPGA

In the PLA, an AND array and an OR array are connected and each has a programmable connection as configuration. In the product term system, in order to realize a desired circuit with fewer circuit resources, it is necessary to express the logical function in a minimum sum-of-products (SoP) form, so the simplification of the logic is very important in the design. The logic function expressed in the sum-of-products form is decomposed into the logical product term and the logical sum term which are, then, implemented in the AND array and the OR array, respectively.

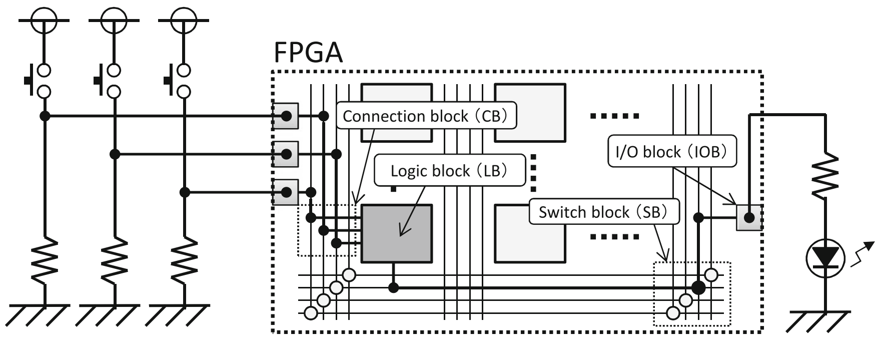


Fig. 2.11 Implementation of a majority vote circuit on FPGA

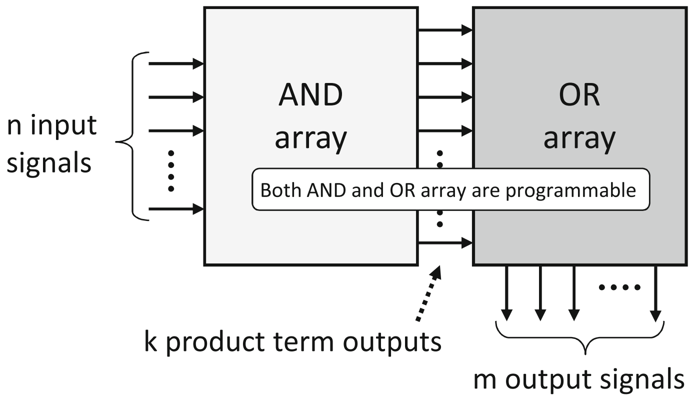


Fig. 2.12 Overview of PLA

Figure 2.13 shows the internal structure of the product term. Within the AND array, the literal of the input signal and the input of each AND gate are connected by a programmable switch. In the OR array, the output of the AND gate and the input of the OR gate are also connected by a programmable switch. In general, in an AND array, *k* logical product terms of literals with up to *n* inputs can be programmed. In addition, the *k* outputs are inputted to the OR array of the next stage and it is possible to program up to *m* logical sum terms of the *k* inputs. In the example shown in Fig. 2.13, it is possible to implement up to four logical functions represented by three-product sum-of-products form.

The majority vote circuit in the previous section is implemented by PLA, as shown in Fig. 2.14. The rhombus at the intersection on the wiring represents a programmable switch where the white ones represent when the switch is off, and the black color indicates that the switch is on. In the AND array of this example, *A* and *B* are inputted to the first AND gate, *A* and *C* are inputted to the second AND gate, *B* and *C* are inputted to the third AND gate. Then, all the AND array outputs are

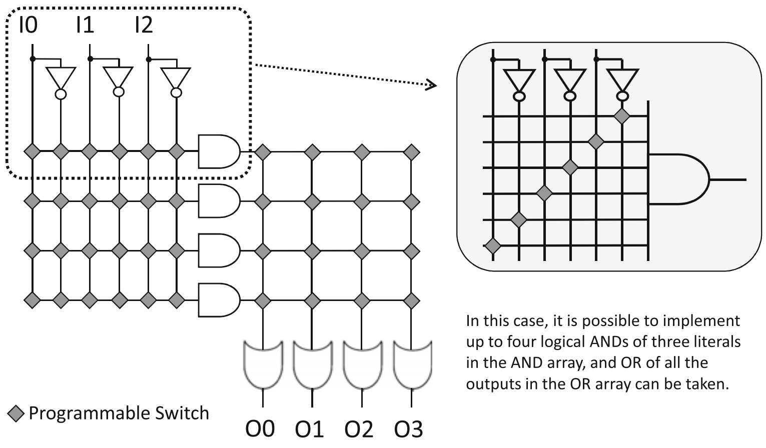
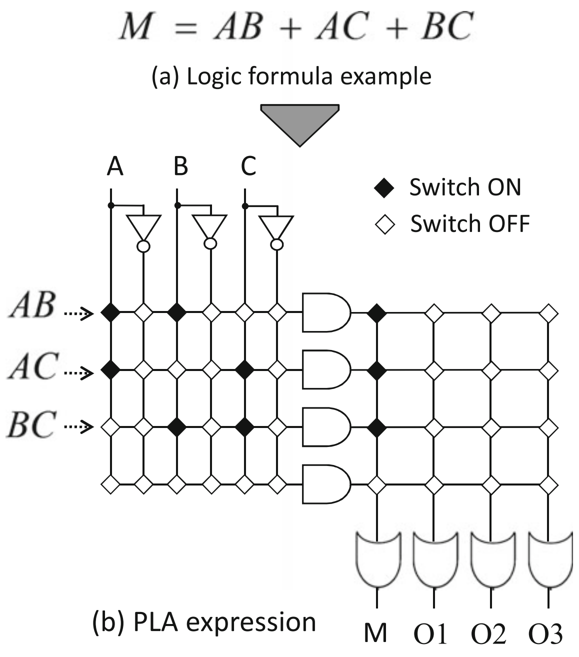

I0
I1
I2
◆ Programmable Switch
O0
O1
O2
O3
In this case, it is possible to implement up to four logical ANDs of three literals in the AND array, and OR of all the outputs in the OR array can be taken.

Fig. 2.13 Structure of PLA

Fig. 2.14 Implementation of majority vote circuit by PLA



connected to the OR gate on the left end of the OR array, so that the logical function $M = AB + AC + BC$ can be realized.

2.4.3 Logical Expression by Lookup Table

A lookup table (LUT) is usually a memory table of 1 word 1 bit, and the number of words is determined according to the number of bits of the address. In FPGAs, SRAM is often employed for memory.

Figure 2.15 shows the schematic of a lookup table. This example shows a 3-input LUT, and it is possible to implement arbitrary logic functions of three inputs. In general, the k-input LUT is composed of 2^k bit SRAM cells and a $2k$-input multiplexer. The input of the LUT is the address itself of the memory table, and it outputs 1 bit. The value of the word is determined according to this address. The k-input LUT can realize a logical function of 2 powered by 2^k. There are 16 kinds of logic functions with $k = 2$, 256 kinds with $k = 3$, and 65,536 kinds with $k = 4$.

Figure 2.16 shows an implementation example for the majority vote circuit that was explained in Sect. 2.4.1 with an LUT. In the LUT implementation, a truth table is created according to the number of inputs of the LUT, and the function value (column of 'f') is written to the configuration memory as it is. If the logic function

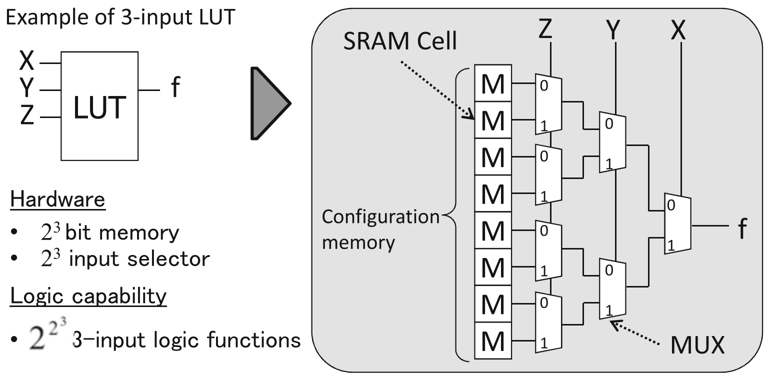
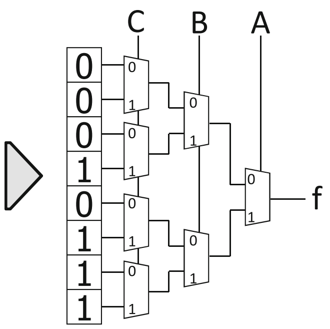


Fig. 2.15 Overview of LUT

Fig. 2.16 Implementation of majority vote circuit by LUT

A	B	C	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a) Truth table



(b) LUT

to be realized has more variables (literals) than the number of inputs to the LUT, it is implemented using multiple LUTs. For this purpose, it is necessary to decompose the logic function to logical functions equal to or less than the number of inputs of the LUT. This method will be described in details later in Chap. 5.

2.4.4 Structure of Lookup Table

In Sect. 2.3.1, we have introduced the static memory outline. Here, we describe the structure of the lookup table. We mainly focus on the structure of the LUT that is adopted in Xilinx FPGAs, along with its historical evolution process.

Figure 2.17 shows the configuration of the static memory and LUT used for Xilinx's initial FPGA. This memory cell is described in US Pat. No. 4,750,155 (September 1985, filed in 1988) [12] and US Pat. No. 4,821,233 (1988 application, established in April 1989) [13], which is an invention of Hsieh from Xilinx Corporation.

The static memory in Fig. 2.17a is a 5-transistor structure which is not currently used. Since the rewriting frequency of the LUT is low, pass transistors, that are necessary for rewriting, can be reduced to prioritize the area over the speed. Figure 2.17b is an example of a 2-input LUT based on this memory. The label 'M' stands for the 5-transistor static memory cell, illustrated in Fig. 2.17a. Since the static memory for the LUT always outputs data, the LUT functions as an arbitrary logic circuit only by selecting values by the inputs $F0$ and $F1$.

Next, Xilinx's Freeman and his colleagues improved the configuration memory of the LUT so that it can be used as a distributed memory in the FPGA. This improvement is described in US patent 5,343,406 (filed in July 1989, established in August 1994) [14]. Figure 2.18 shows this structure wherein the memory configura-

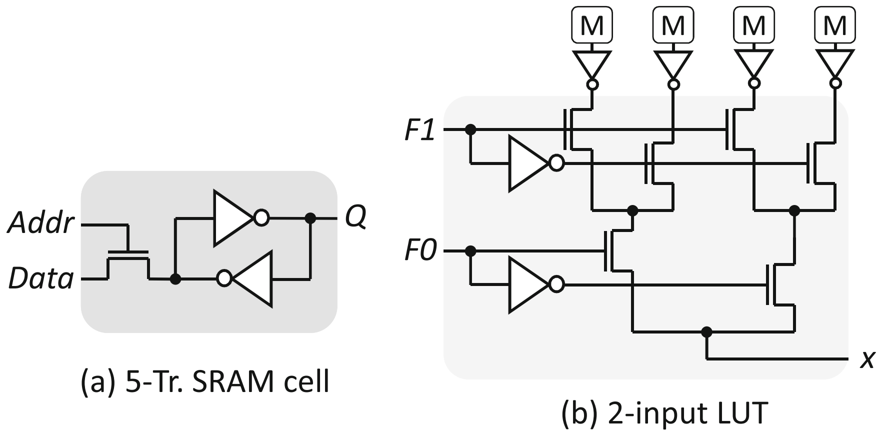


Fig. 2.17 SRAM Cell and a basic structure of LUT

tion depicted in Fig. 2.18a, and another pass transistor is added to the static memory of the above-mentioned 5-transistor structure to form independent write ports (WS and d) to the normal configuration path ($Addr$ and $Data$). When using it as a memory, $F0$ and $F1$, which are the same as the input signal when considered as an LUT, are used. *WS* is a write strobe signal, and the external input signal *Din* is connected to the *d* input, which is selected by the address through the demultiplexer in the upper part of Fig. 2.18b. Reading is performed from the common output, as a conventional LUT. Figure 2.18c represents the block diagram of a 3-input LUT and 8-bit RAM.

Furthermore, Fig. 2.19 is an improved LUT where a shift register can be configured in addition to the memory function. Also, an invention of Bauer from Xilinx, US Patent 5,889,413 (filed in November 1996, approved in March 1999) [15]. In the memory cell of Fig. 2.19a, two pass transistors for shift control are added. $D_{in}/Pre-m$ is the shift input from the external or previous memory. Also, the connection relation is shown at the center of Fig. 2.19b. $PHI1$ and $PHI2$ are signals that control the shifter operation. By applying these control signals with the timing waveform, shown in Fig. 2.20, the shift operation is performed. Note that $PHI1$ and $PHI2$ are none-overlapping signals with opposite phases. When the lower pass transistor is opened by $PHI1$, and the upper pass transistor is opened by $PHI2$, the output of the preceding memory is connected to the input of the subsequent stage and the data are shifted. Figure 2.19c is a configuration diagram in the case of a 3-input LUT, 8-bit RAM, and 8-bit shifter.

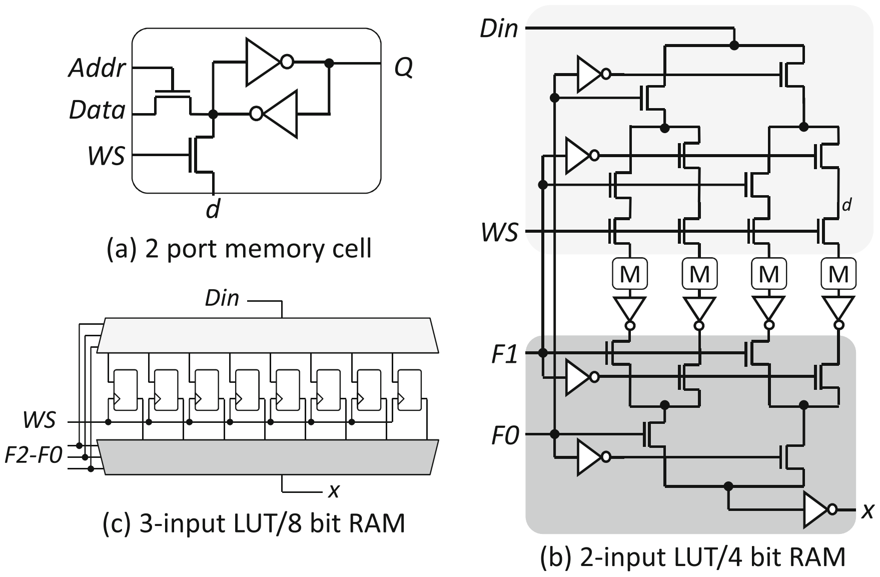


Fig. 2.18 Configuration for using LUT as memory

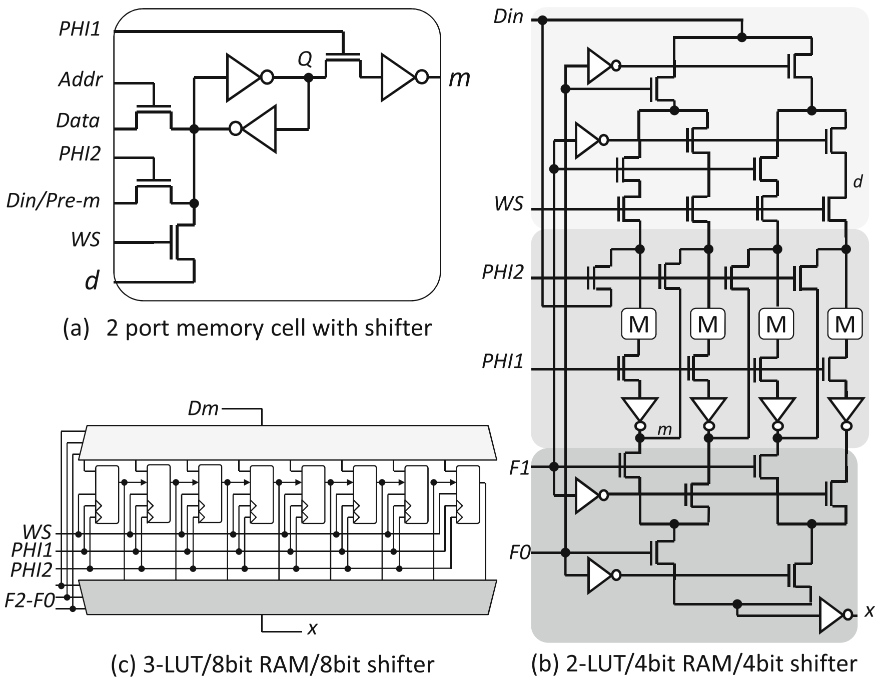

PHI1
Addr
Data
PHI2
Din/Pre-m
WS
d
Q
m
(a) 2 port memory cell with shifter
Din
WS
PHI2
PHI1
F1
F0
M
m
d
x
(b) 2-LUT/4bit RAM/4bit shifter
Dm
WS
PHI1
PHI2
F2-F0
x
(c) 3-LUT/8bit RAM/8bit shifter

Fig. 2.19 Configuration for using LUT as memory and shift register

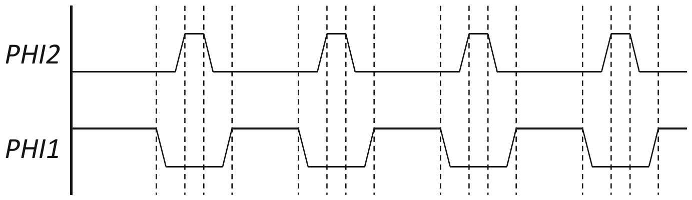


Fig. 2.20 Control timing for shift operation

Furthermore, the current LUT is clustered and adaptive,[10] and realizes a structure that uses multiple LUTs of a small number of inputs as one large LUT. Details about the structure of logical blocks, clustering of LUTs, and adaptive LUT will be described in details in the next chapter.

[10]For example, a method of using an 8-input LUT that can be divided in multiple small LUT clusters like two 7-input LUTs, or a 7-input LUTs and two 6-input LUTs.

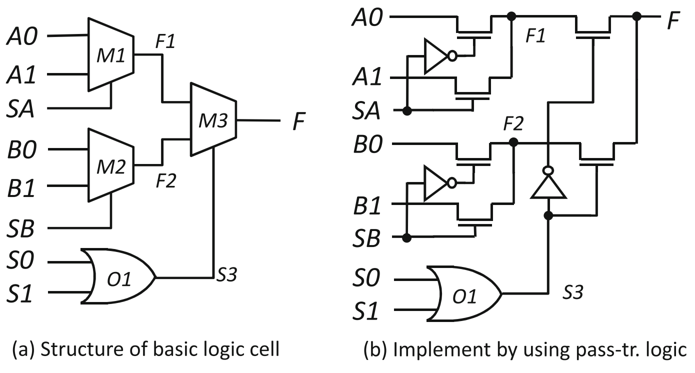

A0
A1
SA
B0
B1
SB
S0
S1
M1
M2
M3
O1
F1
F2
F
S3
(a) Structure of basic logic cell
(b) Implement by using pass-tr. logic

Fig. 2.21 Logic cell using MUX in ACT1

2.4.5 *Logical Expression by Other Methods*

In this section, we describe the logical representation of the structure of logical blocks other than the above. As a representative method, other than the product term method and the lookup table method, there is the multiplexer method. As a representative example, ACT1 FPGA [16, 17][11] is used for explanation.

Figure 2.21 illustrates the logic cell structure of ACT1. The logic cell (shown Fig. 2.21a) consists of three 2-input 1-output multiplexers (2-1 MUX) and one OR gate and can implement up to 8 logic circuits with 1 input. It is possible to implement NAND, AND, OR, and NOR gates up to four inputs and also invert the inputs and make some composite gates (such as AND-OR and OR-AND), latches, flip-flops with this cell.

Unlike the product term and the lookup table, this logic cell cannot implement all logic circuits of a given number of inputs. It combines several fixed circuits like ASIC libraries and assembles the desired circuit. ACT 1 adopts 2-1 MUX as its minimum unit. Table 2.2 represents the logic functions that can be implemented with a 2-1 MUX. The table shows the name, logical expression, and the standard multiply–add form of each function. In addition, the input value of the 2-1 MUX when realizing the function is also included. That is, by connecting it to the input shown in the table, it is possible to implement the logic function.

The 2-1 MUX can be considered as a 3-input 1-output logic cell. In principle, a 3-input and 1-output logic cell (e.g., 3-LUT, etc.) can express logic functions of 2 powered by $2^3 = 256$ kinds of circuits. However, this 2-1 MUX-based cell can only represent 10 types circuits as shown in the table. Still, by combining multiple MUXs, any logic circuit can be implemented. Figure 2.22 shows the function wheel used for searching logic functions that can be realized with a 2-1 MUX. Since basic logic elements such as NOT gates, AND gates, and OR gates are included, it is obvious that

[11]The production of the ACT series has already been stopped, and they are currently unavailable.

Table 2.2 Logic functions that can be represented by 2-1 MUX

	Functions	F	Canonical form	2-1 MUX input		
				A0	A1	SA
1.	'0'	$F=0$	$F=0$	0	0	0
2.	NOR1-1(A,B)	$F=\overline{A+\overline{B}}$	$F=\overline{A}B$	B	0	A
3.	NOT(A)	$F=\overline{A}$	$F=\overline{A}\overline{B}+\overline{A}B$	0	1	A
4.	AND1-1(A,B)	$F=A\overline{B}$	$F=A\overline{B}$	A	0	B
5.	NOT(B)	$F=\overline{B}$	$F=\overline{A}\overline{B}+A\overline{B}$	0	1	B
6.	BUF(B)	$F=B$	$F=\overline{A}B+AB$	0	B	1
7.	AND(A,B)	$F=AB$	$F=AB$	0	B	A
8.	BUF(A)	$F=A$	$F=A\overline{B}+AB$	0	A	1
9.	OR(A,B)	$F=A+B$	$F=\overline{A}B+A\overline{B}+AB$	B	1	A
10.	'1'	$F=1$	$F=\overline{A}\overline{B}+\overline{A}B+A\overline{B}+AB$	1	1	1

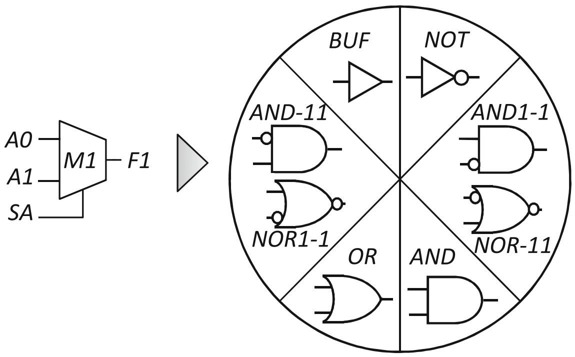


Fig. 2.22 Function wheels used to assign logic to 2-1 MUX

all logic circuits can be made.[12] The function wheel in this figure is used to collate the logic function appearing in it when decomposing the logical function with the *Shannon* expansion performed in the EDA tool. The logical functions shown in this wheel can be realized with one MUX.

Let us take a look at the method to implement the previous majority vote circuit with a MUX-type logic cell. Figure 2.23 shows how to implement the logical function $M = AB + AC + BC$.

First of all, the Shannon expansion of the logical expression with the variable A gives the partial functions $F1$ and $F2$. When these two functions are applied to the function hole, they correspond to AND and OR, so each of them can be realized with one MUX.

The partial function $F1$ is a 2-input logical product function. If the variable B is '1','1' is outputted. When the variable B is '0', no matter what the variable C is, '0'

[12] A set of logical functions that can create all logical functions is called a universal logical function set. In the universal logical function set, there are also sets of only gates such as NAND and NOR besides the NOT, AND, and OR.

$$\begin{aligned} F &= AB + AC + BC \\ &= A \cdot F2(A=1) + \overline{A} \cdot F1(A=0) \\ &= A \cdot (B + C) + \overline{A} \cdot (BC) \end{aligned}$$

$$F2 = 1 \cdot B + 1 \cdot C + BC = B + C$$
$$F1 = 0 \cdot B + 0 \cdot C + BC = BC$$

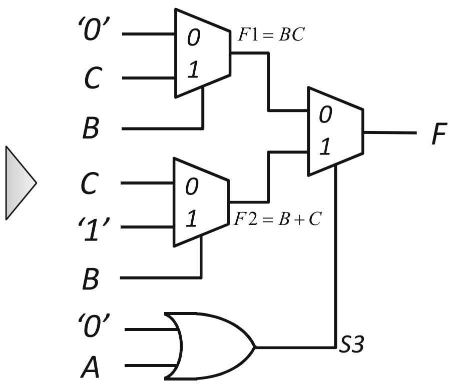


Fig. 2.23 Implementation of majority vote circuit by MUX

is outputted. When this is realized by MUX, it is only necessary to switch between '0'and C by using B as a selection signal (of course, B and C may be reversed). This can be seen at the connection No.7 in Table 2.2. Likewise, the partial function $F2$ can also be realized with a MUX.

On the other hand, the variable A is inputted to the switching signal terminal of the last stage MUX from the input of the OR gate. That is, when A is '1', the output of $F2$ is selected, and when A is '0', the output of $F1$ is selected. The logical expression M is then completely implemented.

Summary of Logic Cells of Other Methods Next, let us summarize the advantages and disadvantages of MUX-type logic cells.

First, the advantage is that one logic cell can be structured with a small number of elements if realized by using a pass transistor, as depicted in Fig. 2.21b. Furthermore, although many transistors, such as a memory for storing a logic function and a memory for determining a connection, are required for the LUT, a memory is unnecessary for the wiring connection because ACT1 employs an antifuse program switch. Therefore, the logic cell of ACT1 has a remarkably smaller area than the other logic cells.

On the other hand, as a disadvantage, ACT1 has high versatility where latches and flip-flops can be structured with logic cells; however, this negatively impacts the performance, especially in terms of degree of integration. The current high-performance FPGAs have dedicated flip-flops circuits and do not make use of logic cells. Otherwise, the degree of integration would be reduced.

There is also a disadvantage that the EDA tool becomes complicated when MUX is used for the logic cell. LUTs or similar methods can be implemented by only dividing the logic into a logical function of a predetermined number of inputs. However, the MUX has to decide whether it can be implemented after dividing it into a logical function of a fixed number of inputs. For logic functions that cannot be implemented, re-division and logic recombination have to be performed. This problem does not matter when the logic scale is small; but, it cannot be ignored for large-scale circuits. Furthermore, because antifuses are employed, logic cells cannot be used for applications requiring reconfiguration. In this fashion, the production of ACT1 was gradually stopped due to the limited use, and now it disappeared from the product lineup. Originally, logic cells that utilized the logic structure of logic cells themselves, such as the MUX type, had better area efficiency than logic cells using memories. They were also superior in terms of delay performance. However, it was not necessarily a big success in commercial terms, and currently there are no products that adopt MUX-type logic cells in commercial FPGAs. At the research level, researchers are also developing logic cells with a performance equal to or better than that of LUTs, such as COGRE [18] and SLM [19].

The key feature of COGRE is its architecture, which helps to reduce the logic area and the number of configuration memory bits. The COGRE cell can only represent high-appearance ratio logic patterns. Moreover, the logic functions are grouped on the basis of the NPN-equivalence class. The investigations' results showed that only small portions of the NPN-equivalence class could cover large portions of the logic functions used to implement circuits. Furthermore, it was found that the NPN-equivalence classes with a high-appearance ratio can be implemented by using a small number of AND gates, OR gates, and NOT gates. On the basis of this observation, 5-input and 6-input COGRE architectures were developed, composed of several NAND gates and programmable inverters.

Moreover, a compact logic cell, named SLM [19], was proposed based on the characteristics of partial functions from Shannon expansion. SLM logic cells use much less configuration memory than used for LUTs with the same input width, without significantly degrading logic coverage.

References

1. S. Brown, J. Rose, FPGA and CPLD architectures: a tutorial. IEEE Des. Test Comput. **13**(2), 42–57 (1996). https://doi.org/10.1109/54.500200
2. T. Sueyoshi, H. Amano (eds.), *Reconfigurable System* (in Japanese) (Ohmsha Ltd., 2005). ISBN-13:978-4274200717
3. T. Speers, J.J. Wang, B. Cronquist, J. McCollum, H. Tseng, R. Katz, I. Kleyner, *0.25 mm FLASH Memory Based FPGA for Space Applications* (Actel Corporation, 2002), http://www.actel.com/documents/FlashSpaceApps.pdf
4. Actel Corporation, ProASIC3 Flash Family FPGAs Datasheet (2010), http://www.actel.com/documents/PA3_DS.pdf
5. R.J. Lipp, et al., General Purpose, Non-Volatile Reprogrammable Switch, US Pat. 5,764,096 (GateField Corporation, 1998)
6. Design Wave Magazine, FPGA/PLD Design Startup2007/2008, CQ (2007)
7. M.J.S. Smith, *Application-Specific Integrated Circuits (VLSI Systems Series)* (Addison-Wesley Professional, 1997). ISBN-13: 978-0201500226
8. Actel Corporation, ACT1 series FPGAs (1996), http://www.actel.com/documents/ACT1_DS.pdf
9. QuickLogic Corporation, Overview: ViaLink, http://www.quicklogic.com/vialink-overview/
10. R. Wong, K. Gordon, Reliability mechanism of the unprogrammed amorphous silicon antifuse, in *International Reliability and Physics Symposium* (1994)

11. I. Kuon, R. Tessier, J. Rose, *FPGA Architecture: Survey and Challenges* (Now Publishers, 2008). ISBN-13: 978-1601981264
12. H. Hsieh, 5-Transistor memory cell which can be reliably read and written, US Pat. 4,750,155 (Xilinx Incorporated, 1988)
13. H. Hsieh, 5-transistor memory cell with known state on power-up, US Pat. 4,821,233 (Xilinx Incorporated, 1989)
14. R.H. Freeman, et al., Distributed memory architecture for a configurable logic array and method for using distributed memory, US Pat. 5,343,406 (Xilinx Incorporated, 1994)
15. T.J. Bauer, Lookup tables which double as shift registers, US Pat. 5,889,413 (Xilinx Incorporated, 1999)
16. Actel Corporation, in *ACT1 Series FPGAs Features 5V and 3.3V Families fully compatible with JEDECs* (Actel, 1996)
17. M. John, S. Smith, *Application-Specific Integrated Circuits* (Addison-Wesley, 1997)
18. M. Iida, M. Amagasaki, Y. Okamoto, Q. Zhao, T. Sueyoshi, COGRE: a novel compact logic cell architecture for area minimization. IEICE Trans. Inf. Syst. **E95-D**(2), 294–302 (2012)
19. Q. Zhao, K. Yanagida, M. Amagasaki, M. Iida, M. Kuga, T. Sueyoshi, A logic cell architecture exploiting the Shannon expansion for the reduction of configuration memory, in *Proceedings of 24th International Conference on Field Programmable Logic and Applications (FPL2014)*, Session T2a.3 (2014)

Chapter 3
FPGA Structure

Motoki Amagasaki and Yuichiro Shibata

Abstract Here, each component in FPGAs is introduced in detail. First, the logic block structures with LUTs are introduced. Unlike classic logic blocks using a couple of 4-input LUTs and flip-flops, recent FPGAs use adaptive LUTs with more number of inputs and dedicated carry logics. Cluster structure is also introduced. Then, routing structure, switch block, connection block, and I/O block which connect basic logic blocks are explained. Next, macromodules which have become critical components of FPGA are introduced. Computation centric DSP block, hard-core processor, and embedded memory can compensate the weak point of random logics with logic blocks. The configuration is inevitable step to use SRAM-style FPGAs. Various methods to lighten burden are introduced here. Finally, PLL and DLL to deliver clock signals in the FPGA are introduced. This chapter treats most of FPGA components with examples of recent real devices by Xilinx and Altera (Intel).

Keywords Adaptive LUT structure · Carry logic · Logic cluster · Routing structure · Switch block · Connection block · I/O block · DSP block · Hard-core processors · Embedded memory · Configuration method · PLL · DLL

3.1 Logic Block

FPGA consists of three basic components: programmable logic element, programmable I/O element, and programmable interconnect element. A programmable logic element expresses a logic function, a programmable I/O element provides an external interface, and a programmable routing element connects different parts. There are also digital signal processing (DSP) units and embedded memory to increase the calculation ability, and phase-locked loop (PPL) or delay-locked loop

M. Amagasaki (✉)
Kumamoto University, Kumamoto, Japan
e-mail: amagasaki@cs.kumamoto-u.ac.jp

Y. Shibata
Nagasaki University, Nagasaki, Japan
e-mail: shibata@cis.nagasaki-u.ac.jp

H. Amano (ed.), *Principles and Structures of FPGAs*,
https://doi.org/10.1007/978-981-13-0824-6_3

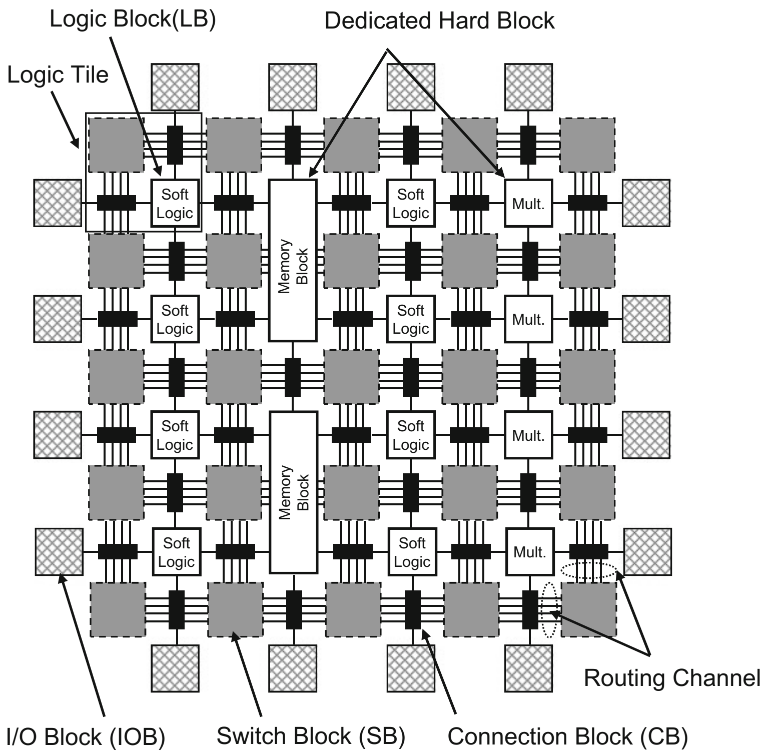


Fig. 3.1 Island-style FPGA overview

(DLL) to provide a clock network. By downloading the design data to these elements, an FPGA can implement the desired digital circuit. Figure 3.1 shows the schematic of an island-style FPGA. An island-style FPGA has logic elements (logic block), I/O elements placed on the periphery (I/O block), routing elements (switch block, connection block, and routing channel), embedded memory, and multiplier blocks. A set of neighboring logic blocks, a connection block, a switch block is called a logic tile. In an island-style FPGA, logic tiles are arranged in a two-dimensional array. The logic block and multiplier block are used as hardware resources to realize logic functions, while the memory block provides storage. Multiplier and memory blocks, dedicated for specific usages, are called "Hard Logic," while functions implemented with logic blocks are called "Soft Logic" [1]. Logic blocks are named differently among FPGA vendors such as configurable logic block (CLB) in Xilinx FPGA and logic array block (LAB) in Altera (now part of Intel). The basic principle is, however, similar. Since commercial FPGAs use LUT, in this section, we focus on LUT-based FPGA architectures.

3.1.1 Performance Trade-Off of Lookup Tables

Although there was a logic block consisting of only LUTs in the early stages, recent FPGAs have basic logic elements (BLEs), as shown in Fig. 3.2. A BLE consists of a LUT, an flip-flop (FF), and a selector. According to the value of the configuration memory bit M0, the selector controls whether the value of the LUT or the one stored in the FF is outputted.

There are several trade-offs between area efficiency and delay when determining the logic block architecture. The area efficiency indicates how efficiently a logic block is used when a circuit is implemented on an FPGA. The area efficiency is high when logic blocks are used without waste. Regarding area efficiency, there are the following trade-offs in logic blocks:

- As functions per logic block increase, the total number of logical blocks required to implement the desired circuit decreases.
- On the other hand, since the area of the logical block itself and the number of inputs and outputs increase, the area per logical tile increases.

The number of LUT inputs is one of the most important factors when determining the structure of a logic block. A k-input LUT can express all k-input truth table. As the input size of the LUT increases, the total number of logical blocks decreases. On the other hand, as a k-input LUT needs 2^k configuration memory bits, the area of the logic block increases. Furthermore, as the number of input/output pins of the logic block increases, the area of the routing part increases. As a result, the area per logical tile increases as well. Since the area of an FPGA is determined by the total number of logic blocks $\times$ the area per logic tile, there is clearly an area trade-off.

The following influences also appear with regards to speed:

- As functions per logical block increases, the number of logic stages (also called logic depth) decreases.
- On the other hand, the internal delay of logic blocks increases.

The number of logical stages is the number of logical blocks existing on the critical path, which is determined at technology mapping. When the number of logic stages is small, the amount of traffic through the external routing track is reduced, which is effective for high-speed operations. Meanwhile, as the function of the logical block increases, the internal delay increases and there is a possibility that the effect of

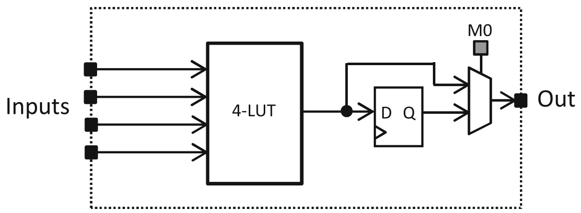


Fig. 3.2 Basic logic element (BLE)

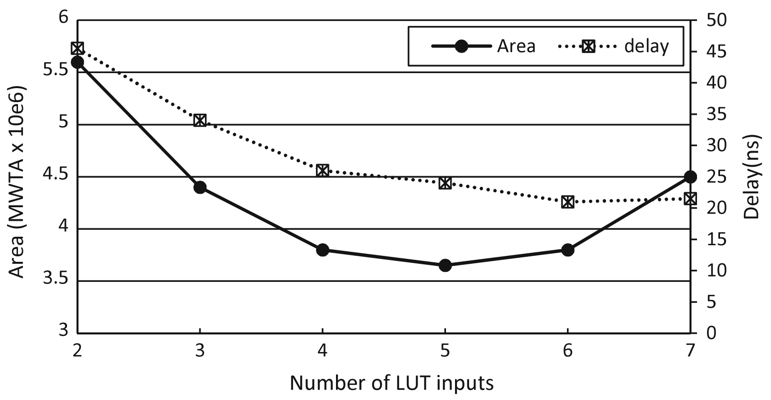


Fig. 3.3 Trade-off between area/delay and LUT inputs

reducing the number of stages may be reduced. In this way, there is also a clear trade-off in terms of speed. Summarizing the above, if the input size of a LUT is large, the number of logic stages is reduced, resulting in a higher operational speed. However, when implementing a logic function with less than *k* inputs, the area efficiency is reduced. On the other hand, if the input size of the LUT is small, the number of logic stages increases and the operational speed is degraded; but, the area efficiency improves. In this fashion, the input size of the LUT is closely related to the area and delay of the FPGA.

The following elements have a great influence on the logic block architecture exploration: the number of LUT inputs, the area and delay model, and the process technology. An architecture evaluation of logic blocks has been studied since the beginning of the 1990s, where it was considered that 4-input was the most efficient [2]. Even in commercial FPGAs, 4-input LUTs were used until the release of Xilinx Virtex 4 [3] and Altera Stratix [4]. Meanwhile, another architecture evaluation was performed using CMOS 0.18 μm 1.8 V process technology [5]. Reference [5] used a minimum-width transistor (MWTAs: minimum-width transistor areas) area model in which the delay is calculated by SPICE simulations after the transistor level design, and each transistor is normalized with a minimum-width transistor. Figure 3.3 shows the transition of the FPGA area and critical path delay when the number of LUT inputs is changed.[1] These results are obtained by placing and routing the benchmark circuits and averaging the obtained values. When the input number of the LUT is 5 or 6, good results are obtained in terms of area and delay. Therefore, recent commercial FPGAs tend to employ larger LUTs like 6-input LUT (also called 6-LUT).

[1]This figure is plotted based on the data presented in [5].

3.1.2 Dedicated Carry Logic

For the purpose of improving the performance of the arithmetic operation, a dedicated carry logic circuit is included in the logic block of commercial FPGAs. In fact, arithmetic operations can be implemented in LUTs; but, using dedicated carry logic is more effective in both degree of integration and operational speed. Figure 3.4 shows two types of arithmetic operation modes in Stratix V [6]. Two full adders (FA) are connected with a dedicated carry logic. The "carry_in" input of FA0 is connected to the "carry_out" of the adjacent logic block. This path is called a high-speed carry chain path enabling high-speed carry signal propagation in arithmetic operations. In the arithmetic operation mode, shown in Fig. 3.4a, each circuit sums the outputs of two 4-LUTs. On the other hand, in the shared computation mode of Fig. 3.4b, 3-input 2-bit additions can be executed by calculating a sum with LUTs. This is used to obtain the sum of the partial products of the multiplier with an addition tree.

Figure 3.5 shows the dedicated carry logic of a Xilinx FPGA. In this FPGA, full adders are not provided as dedicated circuits, and the addition is realized by combining the LUT and the carry generation circuit. The addition (Sum) of the full adder is generated by two 2-input EXOR and the carry-out (Cout) is generated by one EXOR and one MUX. The EXOR of the preceding stage is implemented by a LUT, and the exclusive circuit is prepared for the MUX and EXOR of the latter stage. Similarly to the Stratix V in Fig. 3.4, the expansion to multi-bit adders is possible since the carry signal is connected to the neighboring logic module via the carry chain.

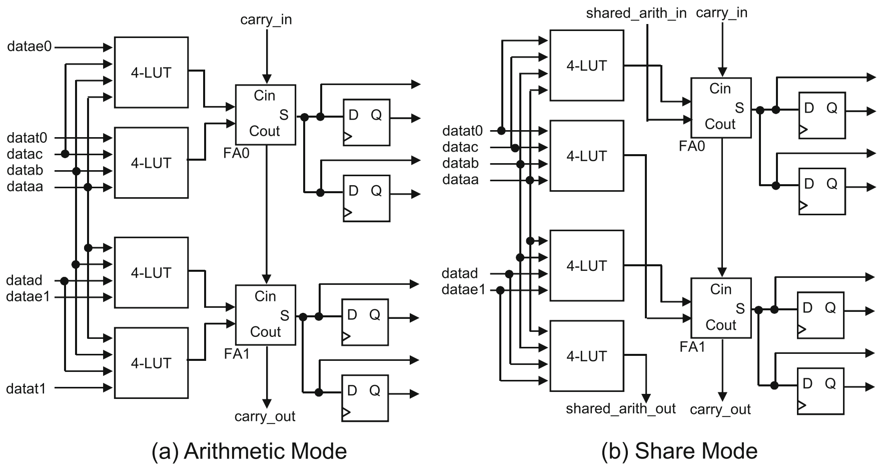


Fig. 3.4 Arithmetic mode in Stratix V [6]

Truth table of full adder

In0	In1	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

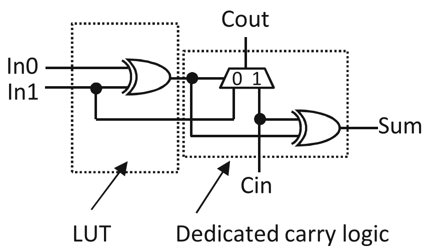


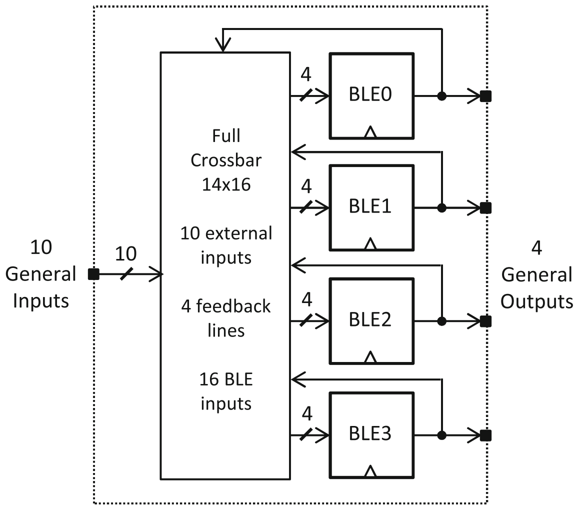
Fig. 3.5 Carry logic in Xilinx FPGA

3.2 Logic Cluster

To increase the number of functions in a logic block without increasing the number of LUT inputs, cluster-based logic blocks (logical clusters) grouping multiple BLEs can be used. Figure 3.6 shows an example of a logic cluster having 4 BLEs and 14 × 16 full crossbar switches. The full crossbar part is called a local connection block or local interconnect, and multiple BLEs are locally interconnected within a logical cluster. Logical clusters have the following features:

1. Since the local wiring in the logic cluster is composed of hard wires, it is faster than the global wiring located outside of the logic cluster.
2. The parasitic capacitance of the local wiring is small when compared with the one of the global wiring [7]. Therefore, using local wiring is effective to reduce the power consumption of an FPGA, especially the dynamic power.

Fig. 3.6 Logic cluster



3. BLEs in a logic cluster can share input signals. Then, the total number of switches of the local connection block can be reduced.

The biggest advantage in the cluster-based logic block is that the total FPGA area can be reduced when the number of functions in the logic block is increased. The LUT area increases exponentially with respect to the input size k. On the other hand, if the size of the logic cluster is N, the area of the logical block increases quadratically. The input signals of a logical cluster can often be shared among multiple BLEs, and in [5], the input I of a logical block is formulated as follows:

$$I = \frac{k}{2}(N + 1) \tag{3.1}$$

The area of the logical block can be reduced by sharing the input signal ($I = N * k$ when treating all inputs of the BLE independently). Similarly to the number of inputs of a LUT, there is a trade-off regarding the area and delay in logic clusters. When N increases, the number of functions per logical block increases, and the number of logical blocks on the critical path decreases, which leads to speedup. On the other hand, since the delay of the local interconnection part also increases as N increases, the internal delay of the logical block itself increases. According to [5], it is reported that $N = 3$–10 and $k = 4$–6 are the most efficient in area delay products.

3.3 Adaptive LUT

In order to obtain higher implementation efficiency, commercial FPGAs' logic blocks have been evolving in the recent years. Figure 3.7 shows the result of technology mapping with 6-LUT for the MCNC benchmark circuit. The technology mapping tool is the area optimization oriented ZMap [8].

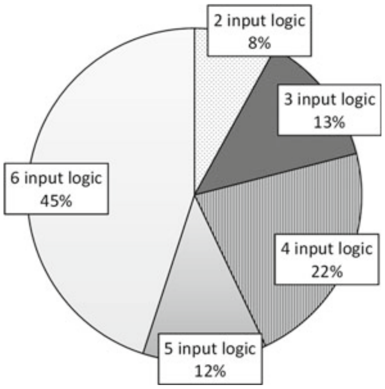


Fig. 3.7 Implementation ratio of logic function after technology mapping

According to this figure, 45% of the total logic function was mapped as 6-input logics. On the other hand, 5-input logics exist by 12%, and in this case, half the configuration memory bits of the 6-input LUT is not used. This is more noticeable as the number of inputs is smaller, and about 93% of the configuration memory bits are actually wasted in the case of a 2-input logic implementation, which is a factor of lowering the implementation efficiency. This problem has been known since the beginning, and in the XC4000 series [9, 10], a complex LB structure containing LUTs of different input sizes was used for logic blocks. However, since the computer-aided design (CAD) support was not sufficient, subsequently it returned to a 4-input LUT-based architecture. Modern FPGAs employ new logic modules called adaptive LUTs since Altera Stratix II [11] and Xilinx Virtex 5 [12]. Unlike conventional single input LUTs, adaptive LUTs are architectures for obtaining high area efficiency by dividing LUTs and implementing multiple logics.

Figure 3.8a shows an example of an adaptive LUT-based logical block [13]. The number of inputs and outputs of the logic block is 40 and 20 (including the carry in and carry out signals), respectively, and the number of clusters is 10. The local connection block is a full crossbar of 60×60, and its inputs include the feedback outputs of each adaptive logic element (ALE). The ALE includes 2-output adaptive LUTs and flip-flops. It has two 5-LUTs sharing all inputs as shown in Fig. 3.8b.

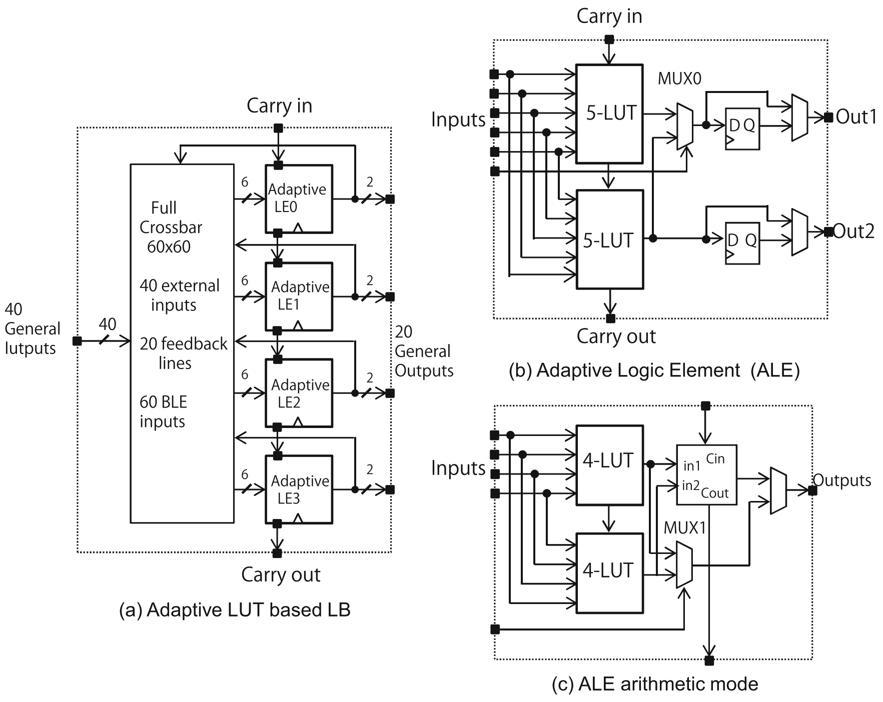


Fig. 3.8 Adaptive LUT-based LB

Thus, an ALE can operate as one 6-LUT or two 5-LUTs sharing the inputs depending on the requirements. In this way, the area efficiency is increased by dividing the 6-LUT into small LUTs and implementing multiple small functions with the circuit resources of the 6-LUT. However, increasing the number of inputs and outputs leads to an increase in the area of the wiring part. For this reason, the number of logical block inputs is suppressed by input sharing. The representative patents related to the adaptive LUT are Altera's US 6943580 [14], Xilinx's US 6998872 (2004 application, established in 2006) [15], and Kumamoto University's US 6812737 [16]. Since 2004, when adaptive LUTs have appeared, the logical block has undergone minor changes; however, its basic structure has not been changed. Hereafter, we explain the logic block architectures of commercial FPGAs using Altera's Stratix II and Xilinx's Virtex 5.

3.3.1 Altera Stratix II

The Stratix II adopts a logical element called adaptive logic module (ALM).[2] ALM consists of an 8-input adaptive LUT, two adders, and two FFs. Figure 3.9a shows the ALM architecture of the Stratix II. The ALM can implement one 6-input logic and two independent 4-input logics, or one 5-input logic and one 3-input logic with independent inputs. In addition, by sharing a part of the inputs, it is possible to implement two logics (e.g, two 5-input logic sharing two inputs) and a subset of a 7-input logic. On the other hand, as shown in Fig. 3.4, two 2-bit adders or two 3-bit adders can be implemented with one ALM. In Stratix II, the LAB with eight ALMs corresponds to a logical block.

3.3.2 Xilinx Virtex 5

Figure 3.9b shows the logic element in Xilinx Virtex 5. Virtex 5 can implement one 6-input logic and two 5-input logics that completely share the inputs. In addition, it has multiple multiplexers. MUXV1 is used to directly output the external input, and MUXV2 is used to select the external input or the output of a 6-LUT. MUXV3 constructs the carry look ahead logic with the input of the carry input C_{in} from the adjacent logic module. At this time, the EXOR generates the SUM signal. MUXV4 and MUXV5 select signals to be outputted to the outside of the LB. In Virtex 5, a set having four logical elements is called a slice, and a CLB having two slices corresponds to a logic block.

[2]In Altera FPGAs, adaptive LUTs are also called fracturable LUTs.

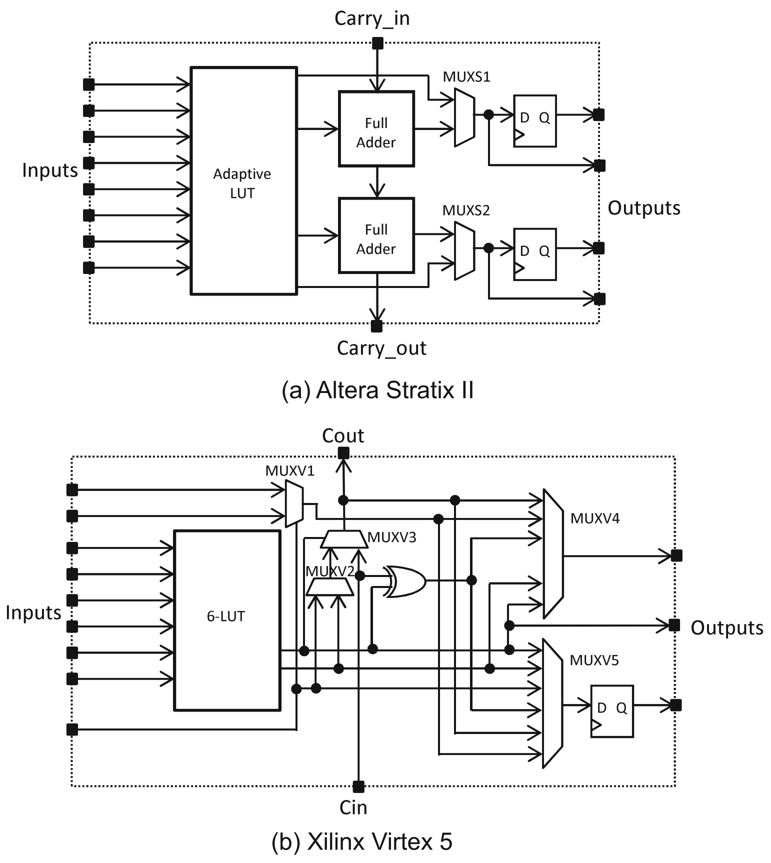


Fig. 3.9 Commercial FPGA architecture

3.4 Routing Part

As shown in Fig. 3.10, the routing structure of an FPGA can be roughly classified into the full crossbar type, the one-dimensional array type, the two-dimensional array type (or island style), and the hierarchical type [17]. These are classified according to how the logical block and I/O block are connected, so-called connection topology. All of them are composed of wiring tracks and programmable switches, and the routing paths are determined according to the values of the configuration memory bits. The full crossbar type shown in Fig. 3.10a has a structure that always inputs the external input and the feedback output of the logic block. This routing structure was commonly seen in programmable array logic (PAL) devices [18] with a programmable AND plane. However, since current FPGAs have enormous logic blocks, such a structure is not efficient. The one-dimensional array type has a structure in which logic blocks are arranged in a column, as depicted in Fig. 3.10b, and the routing channels are

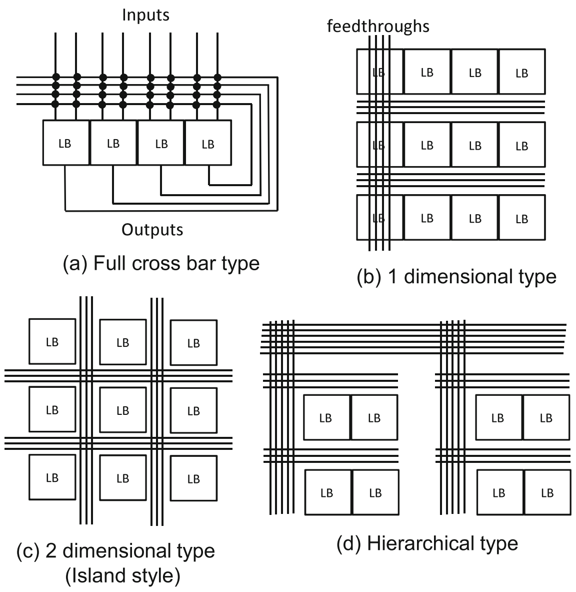


Fig. 3.10 Classification of FPGA routing structure [17]

provided in the row direction. Channels are connected by feed-through wiring, which corresponds to the Actel's ACT series FPGA [19]. In general, a one-dimensional array type wiring part tends to increase the number of switches. Since the ACT series FPGA uses anti-fuse-type switches, even if the number of switches is somewhat larger, the area overhead could be mitigated. However, since SRAM-based switches are mainstream in recent FPGAs, the one-dimensional array type and full crossbar type are not adopted. For these reasons, this chapter introduces the hierarchical and island-style routing structures.

3.4.1 Global Routing Architecture

The routing structure of an FPGA is classified into global and local routing architectures. The global routing architecture is decided from a meta viewpoint that does not consider switch level details, such as connections between logical blocks and the number of tracks per wiring channel. On the other hand, local routing architecture includes detailed connection such as switch arrangement between logic block and the routing channel. All four routing architectures of Fig. 3.10 are global routings.

(1) Hierarchical-type FPGA
Altera Flex 10K [21], Apex [22], and Apex II [23] have a hierarchical routing architecture. Figure 3.11 shows the routing structure in UCB HSRA (high-speed, hierarchical synchronous reconfigurable array). The HSRA has a three-level hierarchical structure from level 1 to level 3. There is a switch at each level, at the intersection of

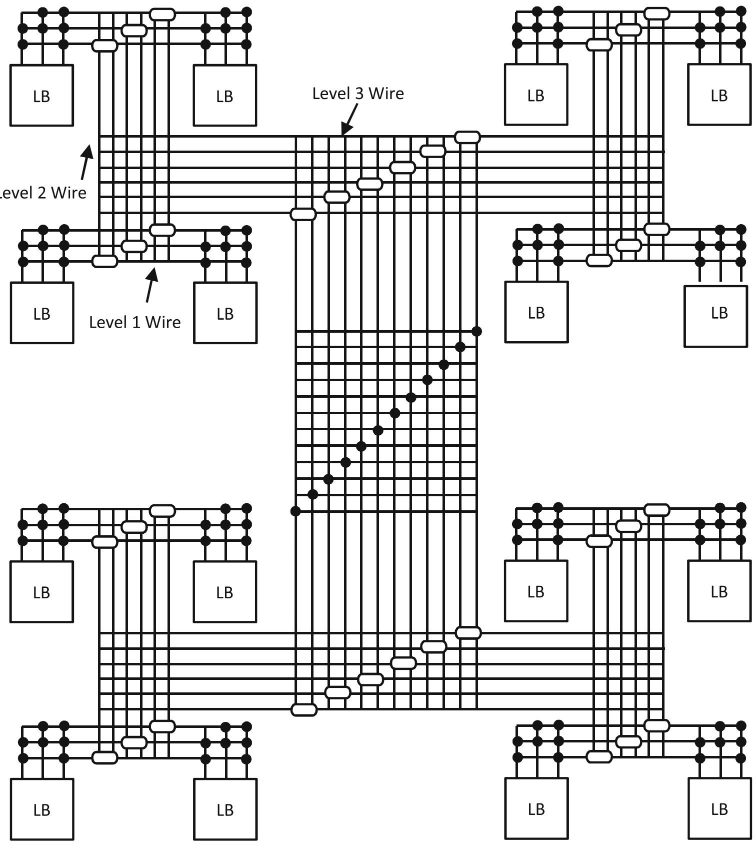


Fig. 3.11 Hierarchical-type FPGA [20]

wire tracks. At the higher hierarchical level, the number of wire tracks per channel increases. At level 1, the lowest hierarchy, wirings between multi-logic blocks are performed. As a merit of the hierarchical FPGA, the number of switches required for signal propagation within the same hierarchy can be reduced. Therefore, high-speed operations are possible. On the other hand, if the application does not match the hierarchical FPGA, the usage rate of the logic block of each hierarchy is extremely low. Also, since there is a clear boundary between two hierarchy levels, once a routing path crosses the hierarchy, it has to pay a delay penalty. For example, if logic blocks physically close to each other are not connected at the same hierarchical level, the delay increases. In addition, since the parasitic capacitance and the parasitic resistance vary greatly in the advanced process, there is a possibility that the delay vary even for connections within the same layer. Although this is not a problem when the worst condition is considered, it can not be ignored when delay optimization is done aggressively. For the above reasons, a hierarchical routing architecture was effective

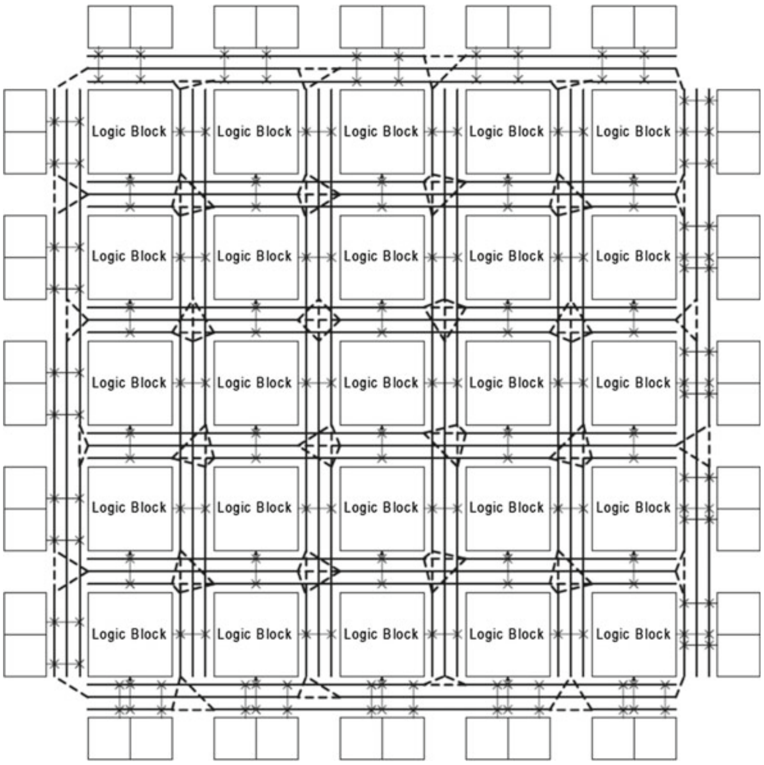


Fig. 3.12 Island-style routing architecture [1]

in older processes where the gate delay was more dominant than the routing delay, but tends not to be used in recent years.

(2) Island-style FPGA
An example of an island-style FPGA is shown in Fig. 3.12 [1]. The island style is adopted by most FPGAs in recent years, and there are routing channels in the vertical and horizontal directions among logic blocks. Connections between logic blocks and routing blocks are generally a two-point connection or a four-point connection, as illustrated in Fig. 3.12. Also, with the uniformization of the logic tile [24, 25], the execution time of routing process can be reduced.

3.4.2 Detailed Routing Architecture

In the detailed routing architecture, the switch arrangement between logic blocks and wire channels, and the length of the wire segment are determined. Fig. 3.13 represents an example of detailed routing architecture. *W* denotes the number of

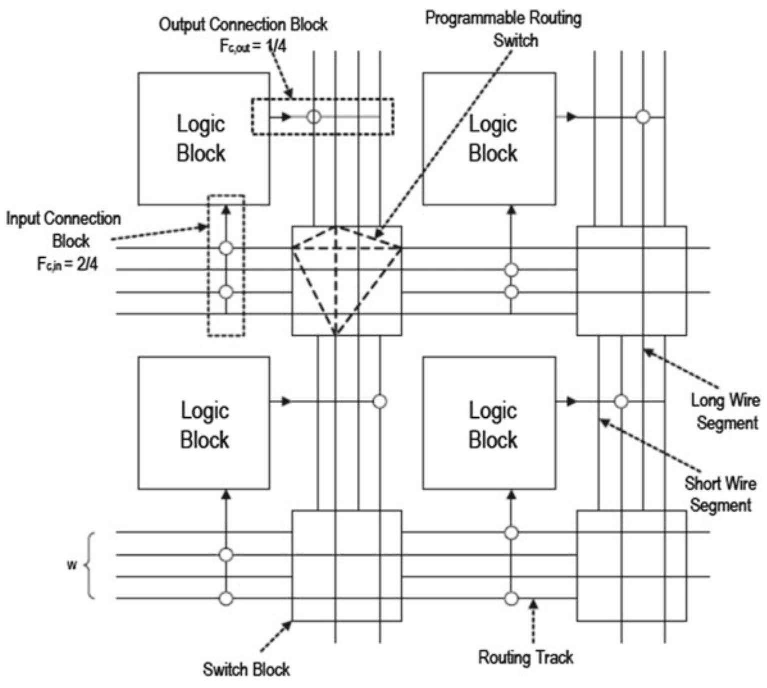


Fig. 3.13 Detailed routing architecture [1]

tracks per channel, and there are several wire segment lengths. There are two types of connection blocks (CBs), one for the input and the other for the output. The flexibility of the input CB is defined as F_{cin}, and as F_{cout} for the output. A switch block (SB) exists at the intersection of the routing channel in the horizontal direction and the vertical direction. The switch block flexibility is defined by F_s. In this example, $W = 4$, $F_{cin} = 2/4 = 0.5$, and $F_{cout} = 1/4 = 0.25$. Also, an SB has inputs from three directions with respect to the one output, $F_s = 3$.

The wiring elements composed of a CB and a SB have a very large influence on the area and the circuit delay of FPGA [26]. Regarding the area, it means that the layout area obtained by the CB and SB is large. As for the delay, in the recent process technology, the wiring delay is dominant over the gate delay. When deciding the detailed routing architecture, it is necessary to consider (1) the relationship between logic blocks and wire channels, (2) the wiring segment length and its ratio, and (3) the transistor level circuit of the routing switch. However, there is a complicated trade-off between routing flexibility and performance. Increasing the number of switches can emphasize flexibility, but also increases the area and delay. On the other hand, if the number of switches is reduced, routing resources become insufficient, rendering routing impossible. The routing architecture is determined considering the balanced use of pass transistors and tristate buffers.

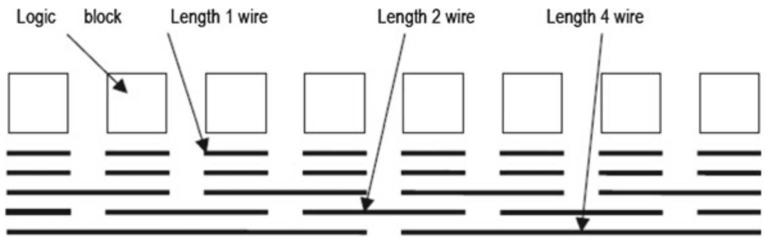


Fig. 3.14 Wire segment length

3.4.3 Wire Segment Length

In the placement and routing using CAD tools, a routing path that satisfies the constraints of speed and electric power is determined. However, it is difficult to conduct an ideal (shortest) wiring for all circuits because of wiring congestion and the number of switch stages. Therefore, it is necessary to perform shortcut routing using medium-distance or long-distance segment length. In fact, there are various segment lengths (e.g. single, quad, and long line) in routing tracks. Three types of wire segments are shown in Fig. 3.14. The distance of the wire segment length is determined by the number of logic blocks. In this figure, there are single lines, double lines, and quad lines. Here, the single lines are distributed at a ratio of 40%, the double lines by 40%, and the quad lines by 20%. In addition, a long-distance wire that crosses the device is called a long line and is used in Xilinx FPGA. The type and the ratio of the wire segment length are often obtained by the architecture exploration using the benchmark circuit.

3.4.4 Structure of Routing Switch

The structure of the programmable switch is important for deciding the routing architecture of an FPGA. In many FPGAs, pass transistors and tristate buffers are used in a mixed form [27–29]. An example of a routing switch is illustrated in Fig. 3.15. The pass transistor is effective for connecting with a small number of switches with respect to a short path. However, since in pass transistors, signal degradation occurs [30], repeaters (buffers) are necessary if the signal passes through many pass transistors. On the other hand, three-state buffers are used for driving long paths. Reference [28] reported that when the allocation ratio of the pass transistor and the three-state buffer is halved, the performance is improved.

Regarding the direction of the wiring track, there are a lot of research on bidirectional wiring and unidirectional wiring [27]. Bidirectional wiring, depicted in Fig. 3.16a, can reduce the number of wiring tracks, but one side of the switch is never used. In addition, since the wiring capacitance also increases, it also affects the delay. On the other hand, in the unidirectional wiring, represented in Fig. 3.16b,

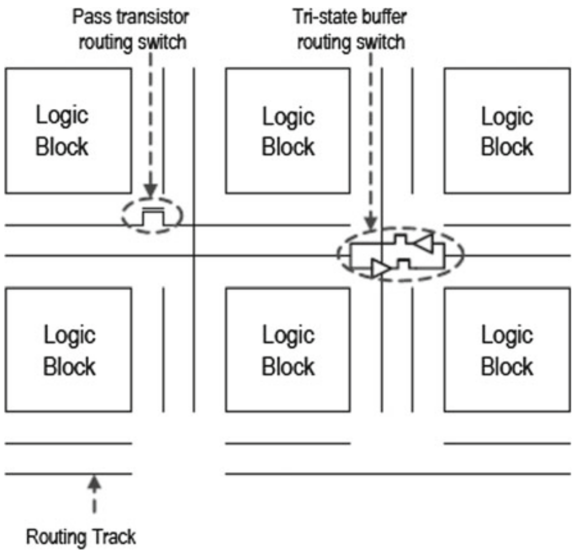


Fig. 3.15 Routing switch

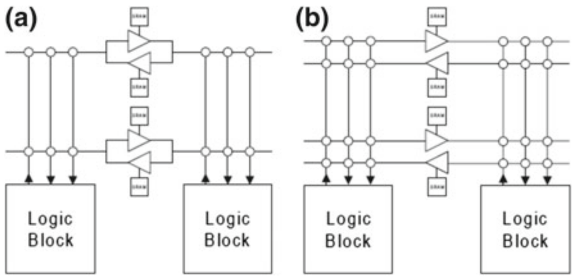


Fig. 3.16 Bidirectional and unidirectional wirings

the number of wiring tracks is twice that of the bidirectional wiring, but the switch is always used and the wiring capacity is small. In this way, there is a trade-off in performance between the bidirectional wiring and the unidirectional wiring. In recent years, the number of transistors' metal layers has increased, and from the viewpoint of ease of design, bidirectional wiring is shifting to unidirectional wiring [1, 31].

Table 3.1 shows the wire length and the number of tracks in commercial FPGAs [26]. However, since details of the routing architecture are not opened for recent FPGAs, the table only includes information for devices of a few generations ago. Xilinx Virtex has single lines ($L = 1$), hex lines ($L = 6$) and long lines ($L = \infty$). One-third of the hex lines is bidirectional, and the rest is unidirectional. On the other hand, Virtex II hex lines are all unidirectional. Altera's Stratix does not have any single line because ALBs are directly connected with dedicated wiring. Also, long-distance segments are connected with $L = 4$, 16, and 24. In this manner, the type and ratio of the wire segments are different depending on the device, and bidirectional and unidirectional wirings are mixed in the wiring direction.

Table 3.1 Wire length and the number of tracks in commercial FPGAs [26]

Architecture	Cluster size N	Array size	Number of tracks of length Lwire						
			1	2	4	6	16	24	∞
Virtex I	4	104×156	24	–	–	48d + 24b	–	–	12
Virtex II	8	136×106	–	40	–	12d	–	–	24
Spartan II	4	48×72	24	–	–	48d + 24b	–	–	12
Spartan III	8	104×80	–	40	–	96d	–	–	24
Stratix 1S80	10	101×91	–	–	160hd + 80vd	–	16vd	24hd	–
Cyclone 1C20	10	68×32	–	–	80d	–	–	–	–

Key "d" unidirectional wiring, "h" horizontal line, "v" vertical line. Bidirectional wiring is not specifically described

3.5 Switch Block

3.5.1 Switch Block Topology

The switch block (SB) is located at the intersection of the wiring channels in the horizontal direction and the vertical direction, and wiring routes are determined by the programmable switches. Figure 3.17 shows three types of typical topologies. The routing flexibility varies depending on SB topologies. In this figure, disjoint type [32], universal type [33], and Wilton type [34] are presented. It also shows the connection relation when the number of tracks is an even number ($W = 4$) or an odd number ($W = 5$). The open circle at the intersection indicates the point at which the programmable switch is placed. Since each SB chooses one output from three input paths, the flexibility of the SB is $F_s = 3$.

(1) Disjoint (Xilinx) type

The disjoint-type SB [32] is used in Xilinx's XC 4000 series, which is also called Xilinx-type SB. Disjoint-type SB connects wiring tracks of the same number in four directions with $F_s = 3$. In Fig. 3.17, when $W = 4$, the left track L0 is connected to T0, R0, B0, and so is the same as $W = 5$. Since the connection is realized by six switch sets, the total number of switches is $6W$. The disjoint-type SB requires a small number of switches, but since it only can connect tracks of the same index value, the flexibility is low.

(2) Universal type

The universal-type SB is a topology proposed in [33]. Like the disjoint-type SB, it consists of $6W$ switches. On the other hand, two pairs of wire tracks can be connected in the SB. When $W = 4$, wire tracks 0, 3 and 1, 2 are paired, respectively, as shown Fig. 3.17. If there is no pair such as wiring track 4 when $W = 5$, it has the same

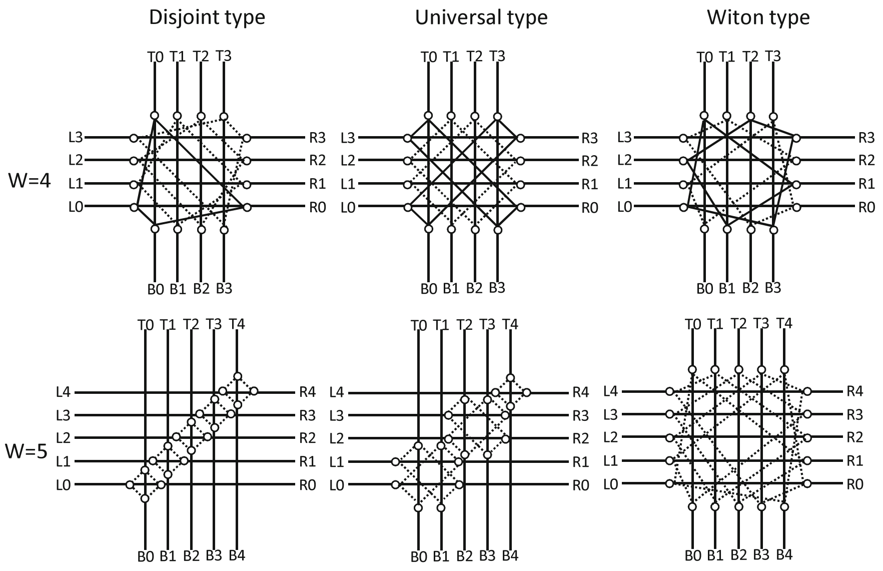


Fig. 3.17 Switch block topology

connection configuration as the disjoint-type SB. It is reported in [33] that the total number of wiring tracks can be reduced with the universal type when compared with disjoint-type SB. However, the universal-type SB assumes only the single line and does not correspond to other wire lengths.

(3) Wilton type

In the disjoint and the universal-type SBs, only the wiring tracks of the same number or two pairs of wiring tracks which are paired are connected. On the other hand, in the Wilton-type SB [34], it is possible to connect wiring tracks of different values with $6W$ switches. In Fig. 3.17, when $W = 4$, the wire track L0 in the left direction is connected to the wire track T0 in the upward direction and the wire track B3 and R0 in the downward direction and the right direction. Here, at least one wire track is connected to the wiring track $(W - 1)$ which is the longest distance. As a result, when routing is performed across several SBs, the routing flexibility is higher than that of other topologies. In addition, it is known that the Wilton type forms a closed circuit by several switch blocks by passing through a clockwise or counterclockwise path. By using this feature, it was shown that the efficiency of manufacturing test of FPGAs can be improved [35, 36].

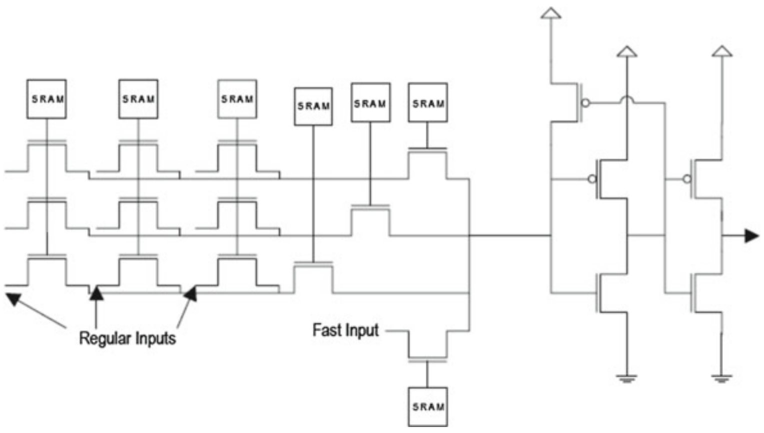


Fig. 3.18 Transistor level structure of SB

3.5.2 *Multiplexer Structure*

The circuit of the programmable switch greatly influences the circuit delay of the SB. Especially in unidirectional wiring, multiple-input multiplexers exist everywhere on routing elements. Since a multi-input multiplexer has a large propagation delay, its circuit structure is important. Figure 3.18 depicts a circuit of multi-input multiplexer in the Stratix II [27]. It has nine normal inputs and one high-speed input for signals on the critical path. Also, since the number of switching stages can be two, this structure is called a two-level multiplexer [27]. In [27], it has been reported that the circuit delay can be improved by 3% without increasing the area. In this manner, the program switch for the routing element has a great importance in reducing the path delay if a configuration memory increase is allowed. On the other hand, an LUT is often composed of pass transistors in a tree structure [30]. Research on repeater placement, transistor sizing, and sizing of multiplexers on wiring is undergoing to reduce the circuit delay of unidirectional wiring. CMOS inverters are usually used for the routing driver of FPGAs. In [37], it is reported that the number of drivers has better delay characteristics in odd-numbered stages than in even numbered stages.

3.6 Connection Block

The connection block (CB) has a role of connecting the input and output of the routing channel and the logic block, which is also composed of programmable switches. Like a local connection block, CB has a trade-off between the number of switches and the flexibility of routing. Particularly, since the routing channel width is very large, if it is simply composed of a full crossbar, the area becomes a problem. For this reason,

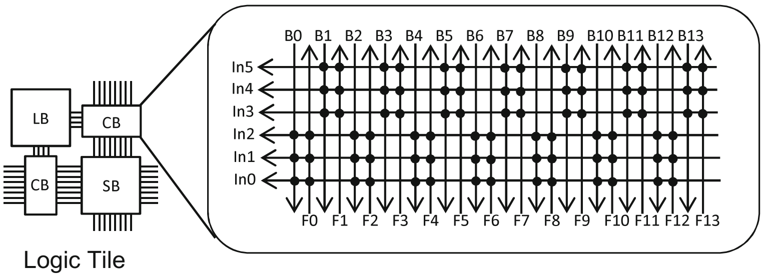

B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13
In5
In4
In3
In2
In1
In0
F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 F13
LB
CB
CB
SB
Logic Tile

Fig. 3.19 Example of switch arrangement of CB

sparse crossbars are used in CB. Figure 3.19 shows an example of a CB composed of sparse crossbars. Wire tracks consist of unidirectional wires, 14 forward wires (F0–F13), and 14 reverse wires (B0–B13). These 28 wire tracks and 6 LB inputs (In0–In5) are connected by the CB. Since each LB input is connected to 14 wiring tracks, $F_{cin} = 14/28 = 0.5$. Sparse crossbars have various configurations [27]; but, the architecture of the CB is determined by exploring the optimum point of flexibility and area.

3.7 I/O Block

An I/O element consists of an I/O dedicated module which interfaces between the I/O pad and an LB. This module is called I/O block (input/output block, IOB), and the IOBs are arranged along the periphery of the chip. The I/O pins of an FPGA have various roles such as power supply, clock, user I/O. An I/O block is placed between an LB and the I/O, such as I/O buffer, output driver, polarity designation, high impedance control and exchanges input/output signals. There are FFs in an IOB so that I/O signals can be latched. Figure 3.20 shows an IOB in the Xilinx XC4000.

The features of this block are shown below, but these basic configurations are the same in recent FPGAs:

- There are a pull-down and pull-up resistors at the output part, and the output of the device can be clamped to 0 or 1.
- An output buffer with an enable signal (OE).
- Each input/output has an FF, so latency adjustment is possible.
- Slew rate of the output buffer can be controlled.
- The input buffer has a threshold of TTL or CMOS. For guaranteeing the input hold time, a delay circuit is provided at the input stage of the MUX 6.

A commercial FPGA has various interfaces providing different output standard, power supply voltage, etc., so it has the role of electrical matching in I/O elements. Many FPGAs also support differential signals (low voltage differential signaling

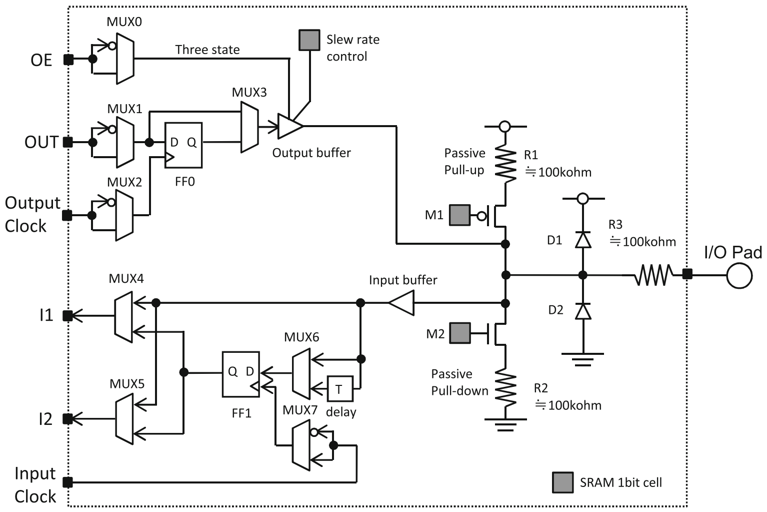


Fig. 3.20 Xilinx XC4000 IOB [38]

(LVDS)) in order to treat high-speed signals and are equipped with a reference voltage for handling different voltages and a clamp diode for handling a specific high voltage. Table 3.2 shows the I/O standard lists of the Stratix V [6]. Since current FPGAs are released according to various application domains, I/O standards are often prepared accordingly. In recent years, as types of I/O standards supported by devices have increased, it has become increasingly difficult for each I/O to respond individually. Then, modern FPGAs also adopt I/O bank method. With this method, each I/O belongs to a predetermined set, which is called a bank, due to supporting various voltages [12, 39]. The number of I/O pins belonging to one bank is different for each device, e.g., 64 pins for Virtex [3] and 40 pins for Virtex 5 [12]. Since each bank shares the power supply voltage and the reference signal, each I/O standard is supported by several banks.

Although an overview of the logical blocks and I/O blocks architectures is available in commercial FPGAs' data sheet, it is difficult to know the details about the layout information and the wiring architecture. A list of literature on FPGA architectures is shown in Table 3.3.

3.8 DSP Block

As described in the previous sections, typical early forms of FPGAs were composed of LUT-based logic blocks, which were connected to each other by wiring elements with programmable switches. Major target applications of those days' FPGAs

Table 3.2 Examples of I/O standards supported by Stratix V [6]

Standard	Output voltage (V)	Usage
3.3-V LVTTL	3.3	General purpose
3.3/2.5/1.8/1.5/1.2-V LVCMOS	3.3/2.5/1.8/1.5/1.2	General purpose
SSTL-2 Class I/Class II	2.5	DDR SDRAM
SSTL-18 Class I/Class II	1.8	DDR2 SDRAM
SSTL-15 Class I/Class II	1.5	DDR3 SDRAM
HSTL-18 Class I/Class II	1.8	QDRII/RLDRAM II
HSTL-15 Class I/Class II	1.5	QDRII/QDRII+/RLDRAM II
HSTL-12 Class I/Class II	1.2	General purpose
Differential SSTL-2 Class I/Class II	2.5	DDR SDRAM
Differential SSTL-18 Class I/Class II	1.8	DDR2 SDRAM
Differential SSTL-15 Class I/Class II	1.5	DDR3 SDRAM
Differential HSTL-18 Class I/Class II	1.8	Clock interface
Differential HSTL-15 Class I/Class II	1.5	Clock interface
Differential HSTL-12 Class I/Class II	1.2	Clock interface
LVDS	2.5	High-speed transfer
RSDS	2.5	Flat panel display
mini-LVDS	2.5	Flat panel display

were glue logic, such as interface circuits between hardware sub-modules and state machines for controlling the system. A configurable circuit was also small in size.

After that, as the FPGA chip size grew, the main application domain of FPGAs moved on to digital signal processing (DSP), such as finite impulse response (FIR) filters and fast Fourier transform (FFT). In such applications, multiplication plays an important role. With standard programmable logic design based on LUT-based logic blocks, however, multipliers need a lot of blocks to be connected to each other, causing large wiring delays and resulting in inefficient execution performance. Thus, early FPGA architectures faced a demand for arithmetic performance improvement, especially for multiplication, to compete with digital signal processors (DSPs).

Under such a background, new FPGA architectures equipped with multiplier hardware blocks in addition to LUT-based logic blocks have emerged since around 2000. Implemented as dedicated hard-wired logic, these multipliers provide high-performance arithmetic capabilities, while flexibility is restricted. Between the multipliers and logic blocks, programmable wiring fabric is provided so that users can freely connect the multipliers with their application circuits. For example, the Xilinx

Table 3.3 References and patents on FPGA architectures

Year	Contents
Commercial	
1994	Architecture around XC4000 [40] D
1998	Flex6000 architecture [41] D
2000	Apex20K architecture and configuration memory bits [42] D
2003	Stratix routing and LB architecture [43] D
2004	Stratix II basic architecture [44] D
2005	Stratix II routing and LB architecture [27] D
2005	eFPGA architecture [45] D
2009	StratixIII and StratixIV architecture [46] D
2015	Virtex ultrascale CLB architecture [47] D
Academic	
1990	Evaluation of optimal input number of lookup table [2] D
1993	Exploring the architecture of homogeneous LUT [10] D
1998	Distribution of wire segment length [26] D
1998	Wire segment and driver [37] D
1999	Cluster-based logic block [48] D
2000	Automatic generation of routing architecture [48] D
2002	Design of programmable switch [30] D
2004	Exploring the cluster size [5] D
2004	Structure of SB and CB [26] D
2008	Evaluation of bidirectional wiring and unidirectional wiring [49] D
2008	FPGA survey includes adaptive LUT [1] D
2009	Evaluation of the gap between FPGA and ASIC [50] D
2014	Adaptive LUT architecture [13] D
Patent	
1987	Patent on internal connectioniCarter patentj [51] D
1989	Patent on basic structure of FPGAiFreeman patent [52] D
1993	Patent on local connection block [53] D
1994	Patent on using LUT as a RAM [54] D
1995	Patent on routing network [55] D
1995	Patent on carry logic [56] D
1996	Patent on cluster-based LB [57] D
1996	Patent on wire segments [58] D
1999	Patent on using LUT as a shift register [59] D
2000	Patent on IOB [60] D
2002	Patent on multi-grain multi-context [16] D
2003	Patent on fracturable LB [14] D
2004	Patent on adaptive LUT [15] D

Virtex-II architecture employed 18-bit signed multipliers. The largest chip in the architecture family offered more than 100 multipliers [61].

Providing such dedicated arithmetic circuits contributes to performance improvement. However, when the provided arithmetic type does not match the application demands, these dedicated circuits are no use at all, resulting in poor hardware utilization efficiency. That is, there are trade-offs between performance, flexibility, and hardware utilization efficiency. Reflecting the increasingly diversified application demands, modern FPGAs provide more sophisticated arithmetic hardware blocks, aiming at both high-speed operations (like dedicated circuits) and programmability to some extent. Generally, these arithmetic hardware blocks are called DSP blocks.

3.8.1 Example Structure of a DSP Block

Figue 3.21 shows the structure of the DSP48E1 slice, which is employed in the Xilinx 7-series architecture [62]. The main components of this block are a signed integer multiplier for 25-bit and 18-bit inputs, and a 48-bit accumulator combined with an arithmetic logic unit (ALU). In many DSP applications, a multiply and accumulate (MACC) operation is frequently performed, which is shown as:

$$Y \leftarrow A \times B + Y.$$

By combining the multiplier and the 48-bit accumulator and by selecting the feedback path from the register of the final stage as an operand, the MACC operation can be implemented using only one DSP48E1 slice.

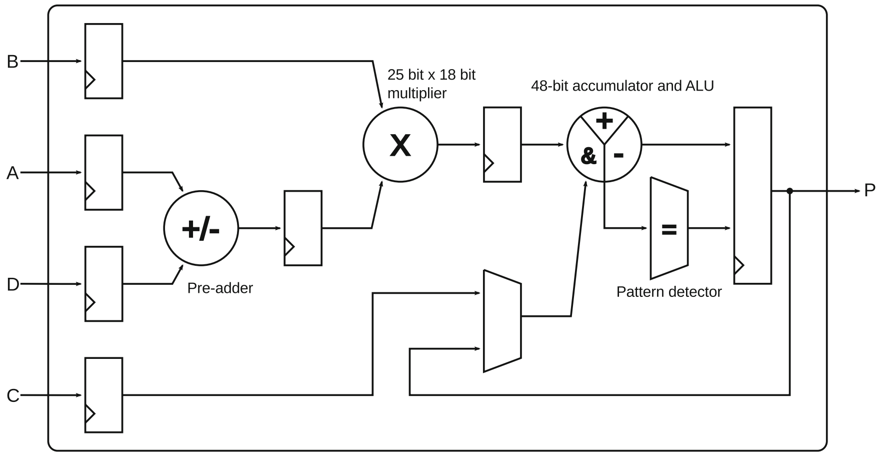


Fig. 3.21 Structure of a Xilinx DSP48E1 slice [62]

The 48-bit accumulator can perform addition, subtraction, and other logic operations. Moreover, the connection of operands is also programmable. Thus, not only multiplication and MACC operations, but also various operations such as three-operand additions and a barrel shifter can be implemented. In front of the final stage register, a programmable 48-bit pattern detector is prepared, which is useful for the determination of a clear condition of a counter and for detecting an exception of an operation result. By enabling register elements between the components, pipelining processing is also performed. As can be seen from the above, the architecture of the DSP48E1 slice is designed to aim at both high degrees of flexibility and performance.

3.8.2 Arithmetic Granularity

Th granularity (bit width) of dedicated arithmetic hardware blocks largely affects the implementation efficiency of applications. When the arithmetic granularity required by an application is coarser than that of the hardware blocks, some hardware blocks must be connected to each other to realize a single operation. In this case, wiring delays among the used blocks will restrict the execution performance. On the other hand, when the arithmetic granularity of the application is finer than that of the hardware blocks, only parts of the hardware resources in the block are utilized, resulting in poor area efficiency of the chip. The required arithmetic granularity, however, naturally varies depending on applications. Thus, the DSP48E1 slice offers the following mechanisms to ensure some flexibility on arithmetic granularity:

- Cascade paths:
  Between two adjacent DSP48E1 slices, dedicate wires called cascade paths are provided. By directly combining a pair of DSP48E1 slices with the cascade paths, arithmetic operations of wider bit width can be implemented without any additional resources in general-purpose logic blocks.
- SIMD arithmetic:
  The 48-bit accumulator also supports operations in a single instruction stream and multiple data streams (SIMD) form. Four independent 12-bit additions or two independent 24-bit additions can be performed in parallel.

On the other hand, Altera Stratix 10 and Aria 10 architectures employ more coarse-grained DSP blocks [63]. As presented in Fig. 3.22, this DSP architecture provides hard-wired IEEE 754 single-precision floating point multiplier and adder, which are useful not only for high-precision DSP applications, but also for acceleration of scientific computation. Between adjacent DSP blocks, dedicated paths for cascade connections are provided. In addition to the floating point arithmetic mode, 18-bit and 27-bit fixed point arithmetic modes are also supported, to offer flexibility in arithmetic granularity to some extent.

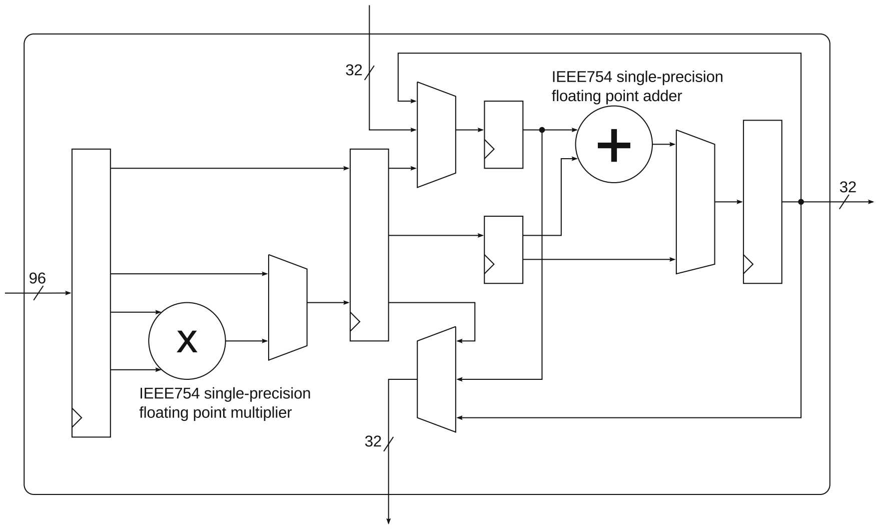

32
IEEE754 single-precision floating point adder
32
96
X
IEEE754 single-precision floating point multiplier
32

Fig. 3.22 Structure of an Altera floating point arithmetic DSP block [63]

3.8.3 *Usage of DSP Blocks*

As described above, even for the same arithmetic operation, modern commercial FPGA architectures offer several different implementation options; for example, using DSP blocks and using logic blocks. Understandably, the arithmetic performance is affected by the implementation option selected by a designer. Basically, the use of DSP blocks has an advantage in terms of performance. However, for example, when DSP blocks are fully utilized, but logic blocks are still left over, an intentional use of logic blocks may allow the implementation with a smaller FPGA chip.

There are several ways for FPGA users to use DSP blocks in their designs. One way is to use intellectual property (IP) generation tools offered by FPGA vendors. With these tools, users can select desired IP from pre-designed libraries, such as arithmetic operators and digital filters. Then, the users can easily generate the module of the IP, after customizing the parameters like bit width. If DSP blocks are available for the selected IP, users can designate or prohibit the usage of DSP blocks through the tool.

In hardware description language (HDL) design, by following HDL coding styles recommended by FPGA vendors, users can make a logic synthesis tool infer usage of DSP blocks. This implementation option has the merit that the same design description can be ported to other FPGA architectures or even to other vendors' chips. Also, users can restrict or prohibit the usage of DSP blocks by setting options of the logic synthesis tools, for the same HDL code.

When low-level access to DSP blocks is preferred, users can directly instantiate primitive modules of DSP blocks in their HDL design. This implementation option, however, lacks design portability for other FPGA architectures.

3.9 Hard Macros

When large-scale FPGAs became available with the improvement of semiconductor integration degree, system-on-FPGA, in which a whole complex digital system is implemented on a single FPGA chip, attracted attention as major FPGA applications. In this case, general-purpose interface circuits, which are commonly used in various systems, are preferred to be implemented as on-chip dedicated hardware in advance, rather than being implemented as part of user logic by each designer. Thus, modern commercial FPGAs are equipped with various on-chip dedicated hardware circuits. Generally, these dedicated hardware circuits are called hard macros.

3.9.1 Interface Circuits as Hard Macros

In addition to the hardware multipliers and DSP blocks described in the previous sections, which are also kinds of hard macros, recent FPGAs also provide PCI Express interface, high-speed serial interface, DRAM interface, analog-to-digital converters (ADC), and so on as hard macros. One of the background factors of this trend is that interface circuits with high-speed clock signals are increasingly needed along with expanding demands for high-performance peripheral devices. Unlike the logic blocks and DSP blocks, only a small number of interface circuits are provided as hard macros for each FPGA. Therefore, layouts of user logic connected to these hard macros require close attention, since unexpected long wires between the user logic and the macros may prevent from achieving the desired circuit performance.

3.9.2 Hard-Core Processors

When FPGA users wish to implement a whole complex digital system on a chip, a microprocessor is often an essential component. Since any logic circuit can be implemented on FPGA, the users can also implement microprocessors on the programmable fabric of the FPGA. Such processors configured on FPGAs are called soft-core processors. Xilinx and Altera offer soft-core processors, which are well optimized for their FPGA families [64, 65]. There are also various open sourced soft-core processors, which can be freely customized by users [66].

While soft-core processors have the merit of flexibility, dedicated processors implemented as on-chip hard macros are advantageous in terms of performance. Such processors are called hard-core processors. Xilinx once commercialized FPGAs equipped with IBM PowerPC hard-core processors. Currently, Xilinx and Altera offer FPGA families which employ ARM cores as hard-core processors.

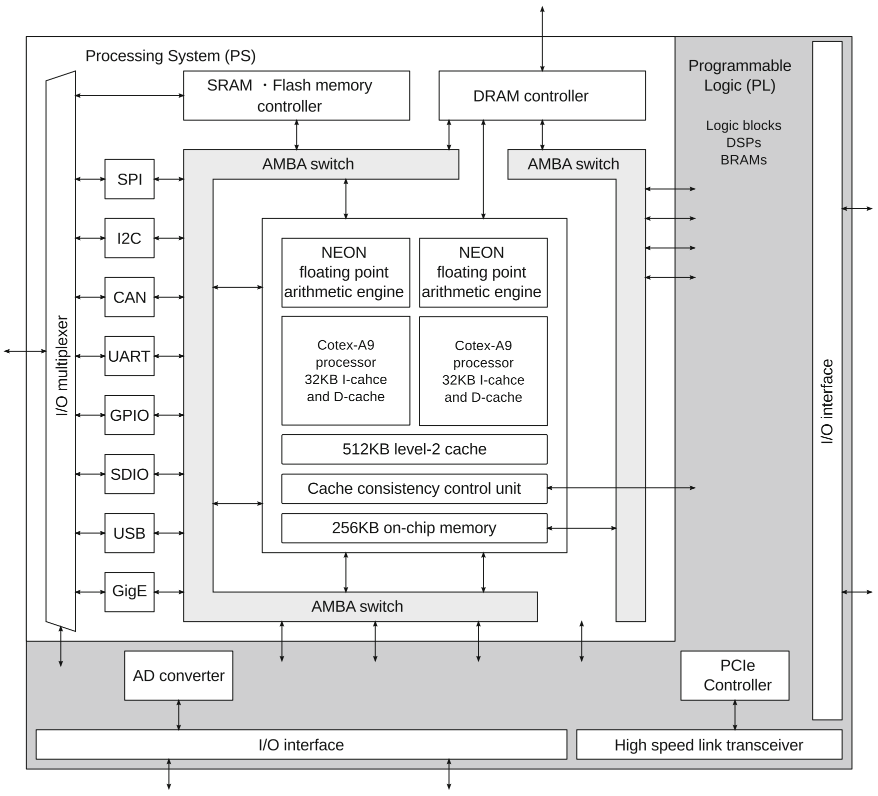


Fig. 3.23 Znyq-7000 EPP architecture [67]

Figure 3.23 shows the Xilinx Zynq-7000 Extensible Processing Platform (EPP) architecture, which provides hard-core processors [67]. The architecture consists of a processor part and a programmable logic part. The former is called processing system (PS), and the latter is called programmable logic (PL). The PS has a multi-core structure with two ARM Cortex A9 cores. Also, interface controllers for external memory and various peripheral devices are provided as hard macros. These hard macros are connected with the cores through a switching fabric based on the AMBA protocol, which is a standard for on-chip interconnection. The hard-core processors can run general-purpose OS such as Linux. The PL has a common structure with ordinary FPGAs, which consists of LUT-based logic blocks, DSP blocks, and embedded memory blocks. By designing an AMBA switch interface on PL according to a predefined protocol, users can connect their application circuits to the hard-core processors. In this way, users can offload parts of the software processing to custom hardware for acceleration.

3.10 Embedded Memory

In early FPGA architectures, which were based on logic blocks consisting of LUTs and flip-flops, flip-flops in the logic blocks were the only memory elements available for user circuits. Thus, it was difficult to store a large amount of data inside the FPGA chip, and external memory was necessary for such data-intensive applications. In this configuration, however, the bandwidth between the FPGA and the external memory often tended to become a performance bottleneck. Therefore, commercial FPGAs, in their development process, have obtained a mechanism to efficiently implement memory elements inside the chip. Such memory elements are collectively called embedded memory. The embedded memory provided by recent commercial FPGAs is roughly classified into two types, as described hereafter.

3.10.1 Memory Blocks as Hard Macros

The first and straightforward approach for providing efficient memory functionality in FPGAs is to introduce on-chip memory blocks as hard macros.

In Xilinx FPGA architectures, such memory blocks provided as hard macros are called Block RAM (BRAM). Figure 3.24 shows the interface of BRAM in the Xilinx 7-series architecture. In this architecture family, dozens to hundreds of BRAM modules are provided depending on the chip size, each of which has a capacity of 36K-bits. One BRAM module can be used as one 36K-bit memory or two independent 18K-bit memories. In addition, two adjacent BRAM modules can be coupled to configure a 72K-bit memory without any additional logic resources.

As shown in Fig. 3.24, a memory access port of BRAM, a group of address bus, data bus, and control signals are duplicated so as to provide two ports: A port and B port. Therefore, BRAM can be used not only as single-port memory, but also as a dual-port memory. This dual-port property allows users to easily configure first-in first-out (FIFO) memory, which is useful for sending and receiving data between sub-modules in user applications. In Xilinx architectures, the access to BRAM must be synchronized with the clock signals. In other words, users cannot obtain the output data of BRAM at the same clock cycle as the one of giving an address to be accessed [68].

3.10.2 Memory Using LUTs in Logic Blocks

The other approach to provide on-chip memories on FPGAs is to use LUTs in logic blocks. As aforementioned, a LUT is a small memory that is used implement the truth table of a combinational circuit. Generally, all the LUTs in an FPGA chip are not consumed to implement combination circuits. Therefore, by allowing users

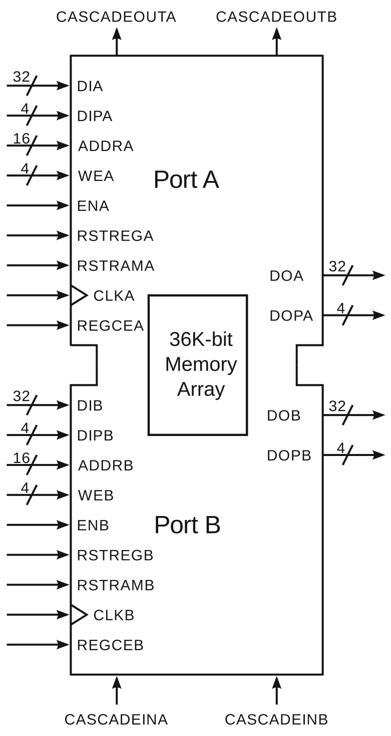


Fig. 3.24 BRAM module in Xilinx 7-series architecture [68]

to access unused LUTs as memory elements, both on-chip memory capacity and hardware utilization efficiency can be increased.

In Xilinx FPGA architectures, such LUT-based on-chip memory is called distributed RAM. However, not all the LUTs in the chip can be utilized; only the LUTs included in logic blocks called SLICEMs can be utilized as distributed RAM. While distributed RAM supports asynchronous access which is not possible with BRAMs, configuring large memory with distributed RAM may leave only small amount of LUTs for logic implementation. Thus, in general, the use of distributed RAM is recommended for configuring relatively a small size memory.

3.10.3 *Usage of Embedded Memory*

Like DSP blocks, there are several ways for FPGA users to utilize embedded memory for their application circuits. FPGA vendors provide memory IP generation tools, with which users can easily generate various memory functions such as RAM, ROM, dual-port RAM, and FIFO. Through these tools, users can also designate the use of BRAM or distributed RAM. Also, by describing HDL code following the coding style recommended by FPGA vendors, users can make a logic synthesis tool to infer the use of embedded memory. A merit of the latter design method is that the portability of the design can be ensured.

Although the capacity of embedded memories available in FPGAs has been increased along with the chip size, a large memory space like DRAM in computer systems cannot be configured inside FPGAs. On the other hand, since multiple banks of BRAMs and distributed RAMs can be accessed in parallel, FPGAs offer a large bandwidth for embedded memories. For efficient application mapping, one of the important keys is how effectively the application can exploit this large bandwidth of embedded memories.

3.11 Configuration Chain

Circuit information used to configure circuits on an FPGA is called a bit stream or configuration data. The configuration data contains all the information required to configure a circuit on an FPGA, including contents of truth tables for each LUT and on/off information for each switch block.

3.11.1 Memory Technologies for Configuration

FPGAs need to store the configuration data inside the chip in some way, and this storage is generally called configuration memory. The configuration memory is classified into the following three types depending on the device technologies used:

1. SRAM type
   SRAM is a rewritable volatile memory. Therefore, while FPGAs in this type can be reconfigured many times, the configured circuits will disappear when the power supply goes off. Generally, external non-volatile memory is used for automatic power-on configuration.
2. Flash memory type
   Flash memory is non-volatile; that is, configuration data will not disappear even when the power supply is off. Although FPGAs in this type can be effectively reconfigured any number of times, the write speed to the flash memory is slower than that of SRAM.
3. Anti-fuse type
   An anti-fuse is an insulator at first, but applying a high voltage makes it a conductor. Using this property, configuration data can be kept like non-volatile memory. However, once an anti-fuse becomes a conductor, it cannot revert back to the original insulator. Therefore, once an anti-fuse type FPGA is configured, it cannot be reconfigured any more.

Since each type of FPGAs has different characteristics, users should select appropriate devices depending on their application demands.

3.11.2 JTAG Interface

In general, SRAM FPGAs are configured from external configuration data memory when the power is turned on. Therefore, most FPGA chips have a mechanism to actively access external configuration memory to obtain configuration data. Conversely, passive configuration, where an external control system sends configuration data to the FPGA, is also generally supported. FPGA users can select a desired mode among these several configurations.

In the developing or debugging phase of circuits, it is convenient that the circuits can be configured many times on an FPGA from a host PC. In most commercial FPGAs, configuration through the Joint Test Action Group (JTAG) interface is supported. JTAG is a common name of the IEEE 1149.1, which is a standard of boundary scan tests. Originally, a boundary scan test makes a long shift register by connecting input/output registers on the chip in a daisy chain manner. Then, by accessing this shift register from outside the chip, it becomes possible to give test values to desired input pins and to observe output values on output pins. To access the shift register, only 1-bit data input, 1-bit data output, and a clock signal are required. Even including a few bits for selecting test modes, a simple interface is enough for this purpose.

In most FPGAs, a configuration mechanism is implemented on top of this boundary scan framework of JTAG. The configuration data is serialized and sent to the FPGA bit-by-bit through the shift register for the boundary scan. This path of the shift register is called a configuration chain. Since shift registers of multiple FPGAs can also be combined and chained into one long shift register, users can select one of the FPGAs and manage the configuration with only one JTAG interface even on multi-FPGA systems.

In recent years, FPGA debug environments using JTAG interface are also provided. For debug purposes, it is convenient that users can observe the behaviors of internal signals while the configured circuit is in operation. However, signals to be observed are needed to be wired to output pins and connected to an observation tool such as a logic analyzer. With the debugging mechanisms offered by JTAG interface, the behaviors of signals to be observed are captured into unused embedded memory. Then, the data can be read back through the JTAG interface to the host PC, so as to be visualized in a waveform. With this environment, users can also set trigger conditions to start capturing, as if a virtual logic analyzer is installed inside the FPGA.

3.12 PLL and DLL

Since FPGAs can configure various circuits whose operation frequencies are naturally varied depending on the critical paths, it is convenient that a clock signal with various frequencies can be generated and used inside the FPGA chip. When circuits configured on the FPGA communicate with external systems, it is desired that the

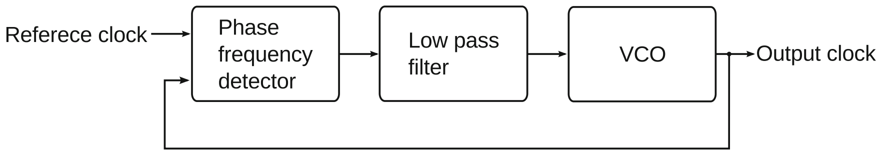


Fig. 3.25 Basic concept of PLL

clock signal which is in phase with an external clock signal is generated. In addition, when the FPGA is connected with multiple peripheral devices, multiple clock signals that have different frequencies and phases are needed according to each interface standard. Thus, recent commercial FPGAs provide an on-chip programmable phase-locked loop (PLL) mechanism, so that various clock signals can be flexibly generated based on an externally given reference clock.

3.12.1 Basic Structure and Operating Principle of PLL

Figure 3.25 shows the basic structure of PLLs. The main part of PLLs is a voltage-controlled oscillator (VCO), whose oscillation frequency can be varied by changing the applied voltage. As shown in Fig. 3.25, PLLs have a feedback structure for the VCO. A reference clock input from the outside and the clock generated by the VCO are compared by a phase frequency detector. When the comparison results show that there is no difference between the two clock signals, the same applied voltage to the VCO can be maintained. If the frequency of VCO is higher or lower than the reference clock, the voltage to the VCO needs to be decreased or increased. Generally, a charge pump circuit is used to translate the comparison results into such an analog voltage signal.

Although it seems that this analog voltage signal can be directly used to regulate the VCO clock, it will make the system unstable. Thus, a low-pass filter is provided in front of the VCO to cut off high-frequency components. In this way, the clock signal that has the same frequency and the same phase with the external reference clock is generated in a stable manner. This operating principle, illustrated in Fig. 3.25, is basically implemented as an analog circuit.

3.12.2 Typical PLL Block

The aforementioned basic structure of PLLs can only generate the clock signal with the same frequency as the reference. However, application circuits on FPGAs often require various clock signals with different frequencies. Therefore, most FPGAs add

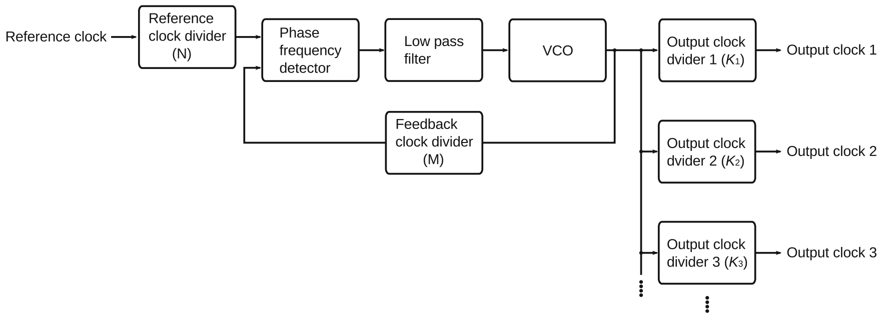

Reference clock
Reference clock divider (N)
Phase frequency detector
Low pass filter
VCO
Feedback clock divider (M)
Output clock dvider 1 (K1)
Output clock dvider 2 (K2)
Output clock divider 3 (K3)
Output clock 1
Output clock 2
Output clock 3

Fig. 3.26 Typical PLL block structure for FPGAs

programmable clock dividers to the basic PLL, shown in Fig. 3.25. A typical structure is depicted in Fig. 3.26.

Firstly, the reference clock is given to the divider in front of the phase frequency detector. When this dividing rate is N, the target frequency of the VCO becomes $1/N$ of the reference frequency. In addition, another clock divider is inserted in the feedback path from the output of the VCO to the input of the phase frequency detector. When this dividing rate is M, the feedback system regulates the VCO frequency so that the target frequency matches with $1/M$ of the VCO frequency. Therefore, the VCO frequency F_{vco} is expressed as

$$F_{vco} = \frac{M}{N} F_{ref} \tag{3.2}$$

where F_{ref} is the frequency of the reference clock. Since programmable clock dividers are used, users can designate desired values for M and N. Depending on the set value of the dividing rate for the feedback clock (M), a clock with a higher frequency than that of the external reference clock is also generated.

As Fig. 3.26 represents, in a typical PLL block for FPGAs, additional clock dividers are also provided behind the VCO, so that the clock generated by the VCO can be further divided. Furthermore, the VCO output branches off into the multiple paths, each of which has individual clock dividers, so that multiple clock signals with different frequencies can be outputted from the common VCO clock. The frequency of the ith output clock F_i is expressed as:

$$F_i = \frac{1}{K_i} F_{vco} \tag{3.3}$$

where K_i is the dividing rate of the corresponding output clock divider. When substituting Eq. (3.3) with Eq. (3.2), the relationship between the reference frequency and the output frequency becomes:

$$F_i = \frac{M}{N \cdot K_i} F_{ref}. \tag{3.4}$$

By setting the appropriate values for M, N, and K_i according to this equation, users can obtain a desired clock frequency from the external reference clock signal.

3.12.3 Flexibility and Restriction of PLL Blocks

Recent commercial FPGA architectures provide further sophisticated clock management mechanisms that can generate clock signals more flexibly. For example, phases and delay amounts of each clock output can be finely varied [69]. For the reference clock and feedback clock, various internal or external signals can be selected and used. Some FPGAs support dynamic reconfiguration of PLLs, which enables changing the clock setting during the circuit is in operation [70].

As Eq. (3.4) shows, a combination of divider rates for M, N, and K_i is not necessarily uniquely determined for a pair of a desired clock frequency F_i and a reference clock frequency F_{ref}. However, a range of available divider rates has some restrictions due to the physical characteristics of clock dividers. In addition, the reference clock frequency F_{ref} and the VCO frequency F_{vco} have their respective upper and lower limits. When multiple PLL blocks are connected in cascade, there are also additional restrictions.

In order to obtain the desired clock frequency by setting appropriate values for M, N, and K_i so that these restrictions are satisfied, users need to carefully go through the documents provided by FPGA vendors. Fortunately, both Xilinx and Altera offer GUI tools to help users. Based on the user inputs on the reference frequency, desired output frequency, and phase shift amount, these tools automatically calculate the division rates so that no restrictions are violated.

3.12.4 Lock Output

As mentioned above, the basic operation principle of PLLs is a feedback system where the VCO frequency is the controlled variable and the reference frequency is the desired value. When the power is turned on, the system reset is activated, and the reference clock is largely fluctuated, a certain time period is required until the feedback systems make the VCO oscillation stable. This means until the PLL output clock gets regulated, the user circuits synchronized with this clock signal may perform unexpected behavior.

To cope with this problem, most PLL blocks are equipped with a mechanism that always monitors the reference clock signal and feedbacks the clock signal. A 1-bit output signal is provided to indicate that the output of the PLL is locked, which means that the VCO output is stable and is tracking well the reference clock. This output signal is called a lock output and is useful for the external circuits to determine the reliability of the clock. For example, by designing a mechanism which keeps the reset signal active while the PLL block continues asserting the lock output, unexpected circuit behavior caused by the unstable clock can be avoided.

3.12.5 DLL

Some FPGA architectures support a clock management mechanism based on a delay-locked loop (DLL) rather than a PLL. Figure 3.27 illustrates the basic concept of a DLL. While a DLL has a feedback structure like a PLL, a VCO is not utilized; but, a delay amount of clock signals is controlled by a variable delay line. Although the variable delay line can also be realized with a voltage-controlled delay element, a digital approach is generally taken for FPGAs. As shown in Fig. 3.28, multiple delay elements are arranged in advance, and the delay amount is varied by changing the number of delay elements that the input signal goes through.

By using a DLL, the phase of the controlled clock can be matched with that of the reference clock. This virtually corresponds to the elimination of the wiring delay from the external reference clock to the controlled clock, that is clock deskew. By the combination with clock dividers, like PLL blocks, DLL blocks can offer

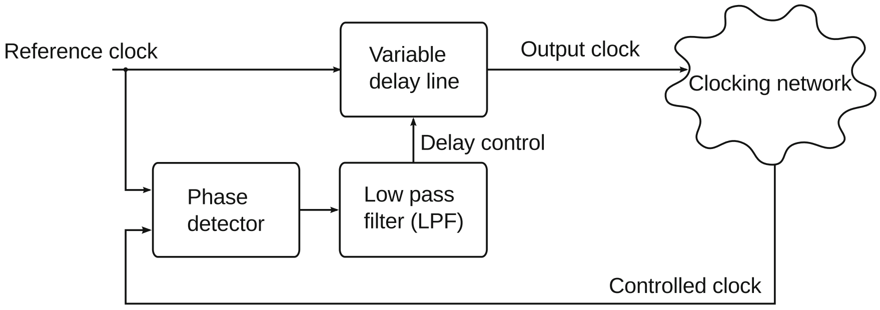


Fig. 3.27 Basic concept of DLL

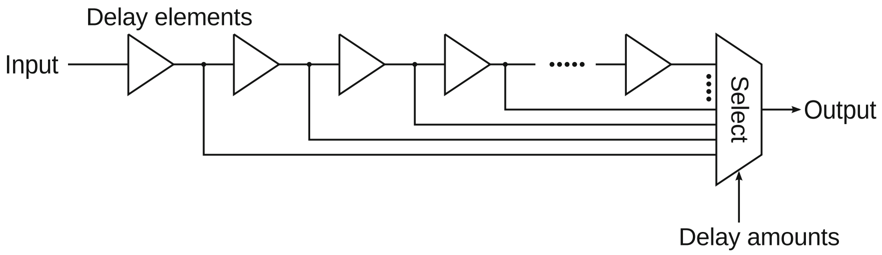


Fig. 3.28 Basic concept of digitally variable delay line

flexibility on the output clock frequency to some extent. Compared to the VCOs used for PLLs, the digitally variable delay lines used in DLLs are more stable and are more robust against to accumulation of phase errors [71]. PLLs, however, have an advantage in terms of flexibility on frequency synthesis. That is why PLL-based clock management mechanisms are the main stream in current commercial FPGA architectures.

References

1. I. Kuon, R. Tessier, J. Rose, FPGA architecture: survey and challenges. Foundat. Trends Electron. Des. Automat. **2**(2), 135–253 (2008)
2. J.S. Rose, R.J. Francis, D. Lewis, P. Chow, Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency. IEEE J. Solid-State Circ. **25**(5), 1217–1225 (1990)
3. Xilinx Corporation, Virtex 4 family overview DS112 (Ver.1.4), Mar 2004
4. Altera Corporation, *Stratix Device Handbook*, vol. 1 (2005)
5. E. Ahmed, J. Rose, The effect of LUT and cluster size on deep-submicron FPGA performance and density, IEEE Trans. Very Large Scale Integrat. (VLSI) Syst. 12(3) (2004)
6. Altera Corporation, Stratix V Device Handbook, Device Interfaces and Integration vol. 1 (2014)
7. J. Lamoureux, S.J.E. Wilton, On the interaction between power-aware computer-aided design algorithms for field-programmable gate arrays. J. Low Power Electron. (JOLPE) **1**(2), 119–132 (2005)
8. UCLA VLSI CAD Lab, *The RASP Technology Mapping Executable Package*, http://cadlab.cs.ucla.edu/software_release/rasp/htdocs
9. Xilinx Corporation, XC4000XLA/XV Field Programmable Gate Array Version 1.6 (1999)
10. J. He, J. Rose, Advantages of heterogeneous logic block architectures for FPGAs, in *Proceedings of IEEE Custom Integrated Circuits Conference (CICC 93)*, May 1993, pp. 7.4.1–7.4.5
11. Altera Corporation, Stratix II Device Handbook, vol. 1. Device Interfaces and Integration (2007)
12. Xilinx Corporation, Virtex 5 User Guide UG190 Version 4.5, Jan 2009
13. J. Luu, J. Geoeders. M. Wainberg, An Somevile, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K.B. Kent, J. Anderson, J. Rose, V. Betz:VTR 7.0: next generation architecture and CAD System for FPGAs. ACM Trans. Reconfig. Technol. Syst. (TRETS), **7**(2), Article No. 6 (2014)
14. D. Lewis, B. Pedersen, S. Kaptanoglu, A. Lee, *Fracturable Lookup Table and Logic Element*, US 6,943,580 B2, Sept 2005
15. M. Chirania, V M. Kondapalli, *Lookup Table Circuit Optinally Configurable as Two or More Smaller Lookup Tables With Independent Inputs*, US 6,998,872 B1, Feb 2006
16. T. Sueyoshi, M. Iida, Programmable Logic Circuit Device Having Lookup Table Enabling To Reduce Implementation Area, US 6,812,737 B1, Nov 2004
17. S. Brown, R. Francis, J. Rose, X.G. Vranesic, *Field-Programmable Gate Arrays*, (Luwer Academic Publishers, 1992)
18. J.M. Birkner, H.T. Chua, *Programmable Array Logic Circuit*, US. 4,124,899, Nov 1978
19. Actel Corporation: ACT 1 Series FPGAs (1996), http://www.actel.com/documents/ACT1DS.pdf
20. W. Tsu, K. Macy, A. Joshi, R. Huang, N. Waler, T. Tung, O. Rowhani, V. George, J. Wawizynek, A. Dehon, HSRA: high-speed, hierarchical synchronous reconfigurable array, in *Proceedings of International ACM Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 125–134, Feb 1999

21. Altera Corporation, FLEX 10K Embedded Programmable Logic Device Family, DS-F10K-4.2 (2003)
22. Altera Corporation, APEX 20K Programmable Logic Device Family Data Sheet, DS-APEX20K-5.1 (2004)
23. Altera Corporation, APEX II Programmable Logic Device Family Data Sheet, DS-APEXII-3.0 (2002)
24. I. Kuon, A. Egier, J. Rose, Design, layout and verification of an fpga using automated tools, in *Proceedings of International ACM Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 215–216, Feb 2005
25. Q. Zhao, K. Inoue, M. Amagasaki, M. Iida, M. Kuga, T. Sueyoshi, FPGA design framework combined with commercial VLSI CAD, IEICE Trans. Informat. Syst. **E96-D**(8), 1602–1612 (2013)
26. G. Lemieux, D. Lewis, *Design of Interconnection Networks for Programmable Logic*, (Springer, formerly Kluwer Academic Publishers, 2004)
27. D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, C. Mcclintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, J. Rose, The stratix II logic and routing architecture, in *Proceedings of the ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 14–20, Feb 2005
28. V. Betz, J. Rose, FPGA routing architecture: segmentation and buffering to optimize speed and density, in *Proceedings of the ACM/SIGDA International symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 140–149, Feb 2002
29. M. Sheng, J. Rose, Mixing bufferes and pass transistors in FPGA routing architectures, in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 75-84, Feb 2001
30. C. Chiasson, V. Betz, Should FPGAs abandon the pass gate?, in *Proceedings of IEEE International Conference on Field-Programmable Logic and Applications (FPL)* (2013)
31. E. Lee, G. Lemieux, S. Mirabbasi, Interconnect driver design for long wires in field-programmable gate arrays. J. Signal Process. Syst. **51**(1) (2008)
32. Y.L. Wu, M. Marek-Sadowska, Orthogonal greedy coupling—a new optimization approach for 2-D field-programmable gate array, in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pp. 568–573, June 1995
33. Y.W. Chang, D.F. Wong, C.K. Wong, Universal switch-module design for symmetric-array-based FPGAs. ACM Trans. Des. Automat. Electron. Syst. **1**(1), 80–101 (1996)
34. S. Wilton, *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories*, Ph.D. thesis, University of Toronto, Department of Electrical and Computer Engineering (1997)
35. K. Inoue, M. Koga, M. Amagasaki, M. Iida, Y. Ichida, M. Saji, J. Iida, T. Sueyoshi, An easily testable routing architecture and prototype chip. IEICE Trans. Informat. Syst. **E95-D**(2), 303–313 (2012)
36. M. Amagasaki, K. Inoue, Q. Zhao, M. Iida, M. Kuga, T. Sueyoshi, Defect-robust FPGA architectures for intellectual property cores in system LSI, in*Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, Session M1B-3, Sept 2013
37. G. Lemieux, D. Lewis, Circuit design of FPGA routing switches, in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* pp. 19–28, Feb 2002
38. M. Smith, *Application-Specific Integrated Circuits*, Addison-Wesley Professional (1997)
39. Altera Corporation, *Stratix III Device Handbook*, vol. 1. Device Interfaces and Integration (2006)
40. S. Trimberger, *Field-Programmable Gate Array Technology* (Kluwer, Academic Publishers, 1994)
41. K. Veenstra, B. Pedersen, J. Schleicher, C. Sung, Optimizations for highly cost-efficient programmable logic architecture, in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 20–24, Feb 1998

42. F. Heile, A. Leaver, K. Veenstra, Programmable memory blocks supporting content-addressable memory, in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 13–21, Feb 2000
43. D. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, S. Marquardt, C. McClintock, V. Pedersen, G. Powell, S. reddy, C. Wysocki, R. Cliff, J. Rose, The stratix routing and logic architecture, in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 15–20, Feb 2003
44. M. Hutton, J. Schleicher, D. Lewis, B. Pedersen, R. Yuan, S. Kaptanoglu, G. Baeckler, B. Ratchev, K. Padalia, M. Bougeault, A. Lee, H. Kim, R. Saini, Improving FPGA performance and area using an adaptive logic module, in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, pp. 135–144, Sept 2013
45. V. AkenOva, G. Lemieus, R. Saleh, An improved "soft" eFPGA design and implementation strategy, in *Proceedings of IEEE Custom Integrated Circuits Conference*, pp. 18–21, Sept 2005
46. D. Lewis, E. Ahmed, D. Cashman, T. Vanderhoek, C. Lane, A. Lee, P. Pan, Architectural enhancements in Stratix-III and Stratix-IV, in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 33–41, Feb 2009
47. S. Chandrakar, D. Gaitonde, T. Bauer, Enhancements in ultraScale CLB architecture, in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 108–116, Feb 2015
48. V. Betz, J. Rose, A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, (Kluwer Academic Publishers, 1999)
49. G. Lemieux, D. Lewis, Circuit design of FPGA routing switches, in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 19–28, Feb 2002
50. K. Ian, J. Rose, *Quantifying and Exploring the Gap Between FPGAs and ASICs* (Springer, 2009)
51. W.S. Carter, Special Interconnect for Configurable Logic Array, US4,642,487, Feb 1987
52. R.H. Freeman, Configurable Electrical Circuit Having Configurable Logic Elements and Configurable Interconnects, US4,870,302, Sept 1989
53. B.B. Pedersen, R.G. Cliff, B. Ahanin, C.S. Lyte, F.B. Helle, K.S. Veenstra, Programmable Logic Element Interconnections for Programmable Logic Array Integrated Circuits, US5,260,610, Nov 1993
54. R.H. Freeman, H.C. Hsieh, Distributed Memory Architecture For A Configurable Logic Array and Method for Using Distributed Memory, US5,343,406, Aug 1994
55. T.A. Kean, Hierarchically Connectable Configurable Cellular Array, US5,469,003, Nov 1995
56. K.S. Veenstra, Universal Logic Module With Arithmetic Capabilities, US5,436,574, Jul 1995
57. R.G. Cliff, L. ToddCope, C.R. McClintock, W. Leong, J.A. Watson, J. Huang, R. Ahanin, *Programmable Logic Array Integrated Circuits*, US5,550,782, Aug 1996
58. K.M. Pierce, C.R. Erickson, C.T. Huang, D.P. Wieland, Interconnect Architecture for Field Programmable Gate Array Using Variable Length Conductors, US5,581,199, Dec 1996
59. T.J. Bauer, Lookup Tables Which Bouble as Shift Registers, US5,889,413, May 1999
60. K.M. Pierce, C.R. Erickson, C.T. Huang, D.P. Wieland, I/O Buffer Circuit With Pin Multiplexing, US6,020,760, Feb 2000
61. Xilinx Corporation, Virtex-II Platform FPGAs: Complete Data Sheet, DS031 (v4.0) Apr 2014
62. Xilinx Corporation, 7 Series DSP48E1 Slice User Guide, UG479 (v1.8), Nov 2014
63. U. Sinha, *Enabling Impactful DSP Designs on FPGAs with Hardened Floating-Point Implementation*, Altera White Paper, WP-01227-1.0, Aug 2014
64. Xilinx Corporation, MicroBlaze Processor Reference Guide, UG984 (v2014.1), Apr 2014
65. Altera Corporation, Nios II Gen2 Processor Reference Guide, NII5V1GEN2 (2015.04.02), Apr 2015
66. R. Jia et al., A survey of open source processors for FPGAs, in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–6, Sept 2014
67. M. Santarini, Zynq-7000 EPP sets stage for new era of innovations. Xcell J. Xilinx **75**, 8–13 (2011)

68. Xilinx Corporation, 7 Series FPGAs Memory Resources UG473 (v1.11) Nov 2014
69. Xilinx Corporation, 7 Series FPGAs Clocking Resources User Guide, UG472 (v1.11.2) Jan 2015
70. J. Tatsukawa, MMCM and PLL Dynamic Reconfiguration, Xilinx Application Note: 7 Series and UltraScale FPGAs, XAPP888 (v1.4), Jul 2015
71. Xilinx Corporation, Using the Virtex Delay-Locked Loop, Application Notes: Virtex Series, XAPP132 (v2.3) Sept 2000

Chapter 4
Design Flow and Design Tools

Tomonori Izumi and Yukio Mitsuyama

Abstract This chapter introduces how to design the target module on an FPGA from designers' point of view. Now, FPGA vendors support integrated design tools which include all steps of design. Here, mainly Xilinx is adopted as an example, and its design flow is introduced from HDL description to programming and debugging devices. Next, high-level synthesis (HLS) which enables to design hardware with high-level programming language is introduced. In order to describe hardware, there are several restrictions and extension in front-end programming languages. The key issue to achieve enough performance is inserting pragmas for parallel processing and pipelining. Then, IP-based design for improving the productivity is introduced. The last subject of this chapter is how to use hard-macro processor in recent SoC-style FPGAs. Designers have to read a large amount of documents from vendors when they start the FPGA design, but by reading this chapter, they can get a brief overview of the total design.

Keywords Design tools · HDL design · HLS design · IP-based design

This chapter introduces the design flow, design tools, and development environments to implement target systems on FPGAs. We explain how a designer elaborates and implements the designed object starting from a given specification and constraints describing source codes and drawing block diagrams reflecting the designer's plan of architecture using a set of design tools. Readers might refer to Chaps. 1, 2, and 3 of this book and references [1] and [2] for general FPGA descriptions.

Although the content of this chapter mainly follows FPGAs and design tools of Xilinx Inc., [3–8, 12–17], we strive to discuss the concepts and principles independent of FPGA vendors, tools, and versions . Similar environments are supplied by

T. Izumi (✉)
Ritsumeikan University, Kusatsu, Japan
e-mail: t-izumi@se.ritsumei.ac.jp

Y. Mitsuyama
Kochi University of Technology, Kami, Japan
e-mail: mitsuyama.yukio@kochi-tech.ac.jp

H. Amano (ed.), *Principles and Structures of FPGAs*,
https://doi.org/10.1007/978-981-13-0824-6_4

Altera Corp. and others. Readers might consider consulting the manuals and tutorials supplied by each vendor or books related to individual design tools and their advanced usage.

4.1 Design Flow

A designer embodies the target by describing source codes, by drawing circuit diagrams, and by setting parameters, based on given specifications and constraints. Figure 4.1 presents an outline of the design flow in this chapter.

We assume source codes in register transfer level (RTL) described in the hardware description language (HDL), such as Verilog HDL and VHDL, as typical design entries. Configuration data to implement the target system on the FPGA chip are generated through RTL description, logic synthesis, technology mapping, and place and route. This design flow is detailed in Sect. 4.2. Verification by simulation, loading configuration data into an FPGA chip and debugging on an actual device are described as well. Recently, high-level synthesis (HLS) that generates hardware modules from

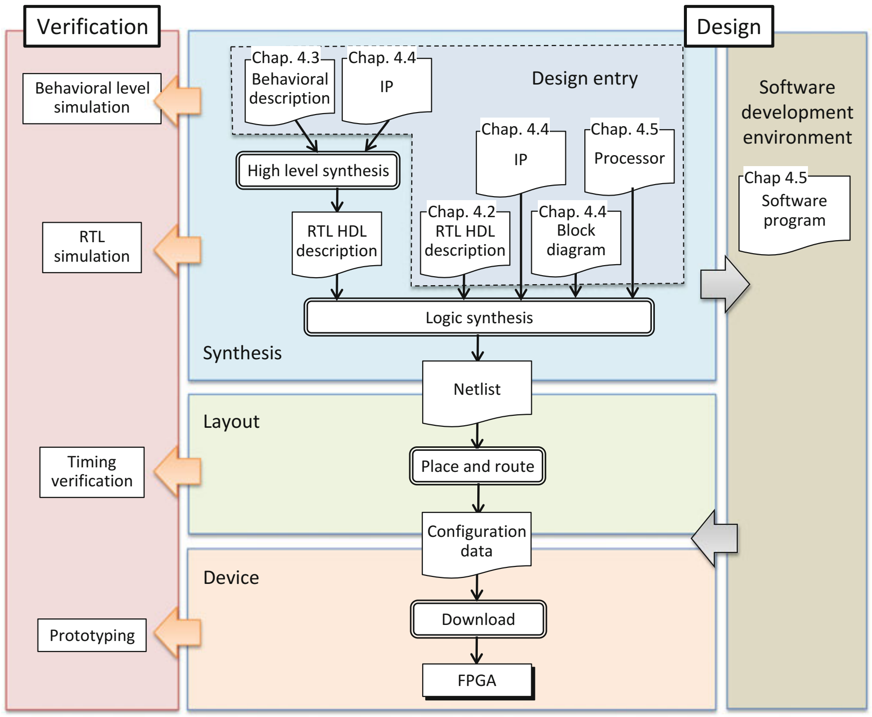


Fig. 4.1 Design flow

behavioral description in C language and others is being introduced to practical use. Section 4.3 explains the design flow starting from behavioral description, processed through simulation in the behavioral level and behavioral synthesis, handed to logic synthesis. Section 4.4 introduces the design method that incorporates existing intellectual properties (IPs) into the design as parts. A design tool working on block diagrams is available for the IP-based design. The block diagram for IPs is in a higher abstraction level than classical circuit diagrams in logic gate level or operation unit level and handles more complicated functional units mutually connected by an interface composed of data and control signals as a unit. Finally, in Sect. 4.5, the design method for a system equipped with a processor is described, including building the system, software development, implementation, and debugging.

4.2 Design Flow by HDL

This section introduces the design flow and related tools starting from RTL descriptions. Here, we presume that the target object is described in an HDL such as Verilog HDL or VHDL in the abstraction level of RTL. The design flow for RTL descriptions is composed of logic synthesis, technology mapping, place and route, generating configuration data, programming the FPGA, and execution on an FPGA. To write a circuit on an FPGA, a designer must supply physical constraints such as the FPGA part number, pin assignment on the board, clocks, timing, and other information in addition to a description of the target object to be designed. In detailed debugging, the designer must specify observed signals and attach the observation system. The performance (speed, size, power consumption, and others) of the resulting circuit implemented on the FPGA varies even for the same function and description, depending on the optimization options. Thus, provision of the optimization strategy for design tools is needed for the specification requested. In the followings, we explain the design and the design tools in each step following the design flow.

4.2.1 Registration of Project

The whole information of the target object to be designed is handled as a unit and called the design project in today's integrated development environment. The unit includes files for tool settings, constraint files, final and intermediate products, and reports in addition to the source codes of the designed object. Figure 4.2 presents an example of information in a project. At the beginning of the design, a designer creates a new project. It has a project name and a working folder (directory). All the files including the project information, products, and reports generated later are stored in that folder .

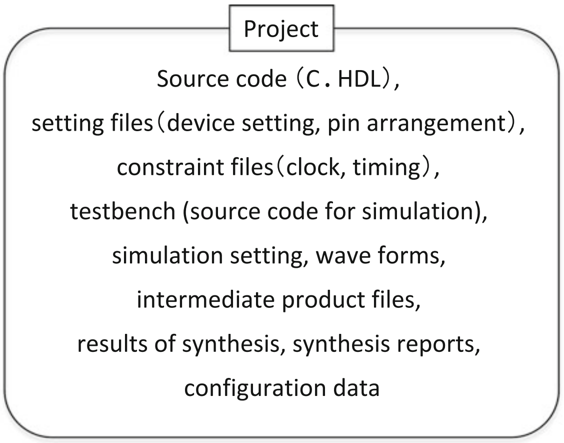

Project
Source code（C．HDL),
setting files(device setting, pin arrangement),
constraint files(clock, timing),
testbench (source code for simulation),
simulation setting, wave forms,
intermediate product files,
results of synthesis, synthesis reports,
configuration data

Fig. 4.2 An example of information in a project

(1) Setting constraints

In this step, the designer registers the physical constraints on the project including the target FPGA device and FPGA board, the pin assignment, and clocks. They are described as a constraint file or set by a menu for constraints in the tool, and stored in the working folder together with the source codes. The setting peculiar to the board is usually supplied by the board vendor as a master constraint file or a board definition file. The designer might modify or add the parameters. In the device setting, the designer sets the series (family), the model number, the package, the number of pins, speed grade, and other settings of the target FPGA device. In the pin arrangement, the designer sets the pin number, direction of I/O, voltage, signaling system, and other information, for each I/O signal of the top module. Furthermore, in the clock signal, the designer sets the source of the clock, period, duty ratio, and other information.

(2) Source code registration

A designer registers the source code of the designed object in the project. If the target object is composed of files of each module, then the designer registers all of them. A design tool analyzes the registered files and identifies instances such as modules, registers, and wires. They are displayed as a tree-structured list designated as an instance tree according to an inclusive relation. Figure 4.3 presents an example of an instance tree and a module hierarchy. In the top description sampletop.v, the modules of SampleTop and FilterModule are described in this example. An instance named as filter of FilterModule is used in the top module of SampleTop. In addition, instances named as ififo and ofifo of FifoModule, described in fifo.v, are used in SampleTop. If any syntax-level errors exist in the source codes, then the tool warns at the registration and the designer can check and correct following the error reports. The module at the root of the tree is usually the top module of the designed object; however, a user might designate another as the top. When a

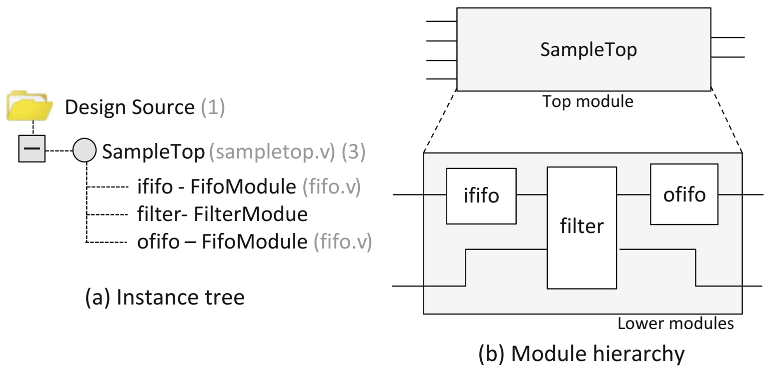


Fig. 4.3 Examples of instance tree and module hierarchy

packaged intellectual property (IP, explained in Sect. 4.4) is used, it is registered additionally.

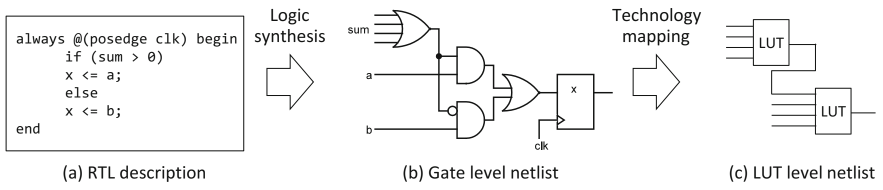
(3) Registration of source code for simulation

Source codes for simulation are registered in the project as well. The source codes for the designed object themselves are registered here naturally as a part of the simulation source codes. In addition, a test bench, which provides input signals from the designed object and which observes the output signals from it, is also registered as the source code designated for simulation. When an IP is supplied as a black box without the source code, a behavioral model of the IP for simulation is supplied by the IP vendor and registered as a simulation source code here in the project in the same way.

4.2.2 Logic Synthesis and Technology Mapping

The process that generates a logic circuit and a sequential circuit from an RTL description of the designed object is designated as logic synthesis. The product of the logic synthesis is referred to as a netlist. The netlist is composed of logic elements, including logic gates and flip-flops, and connections among the elements. The process that allocates the logic elements in the netlist to actual logic elements in the target FPGA is designated as technology mapping. Many FPGAs adopt programmable logic elements named look-up tables (LUTs). The process of logic synthesis from RTL and technology mapping is outlined in Fig. 4.4. In the current development environment, the processes of logic synthesis and technology mapping are largely automated and all the operations are completed with just one click.

The integrated development environment optimizes the operation speed and the circuit size in the processes of logic synthesis and technology mapping. In the optimization process of synthesis, constant inputs and unused outputs are recursively



```
always @(posedge clk) begin
      if (sum > 0)
      x <= a;
      else
      x <= b;
end
```

Fig. 4.4 Logic synthesis and technology mapping from RTL description

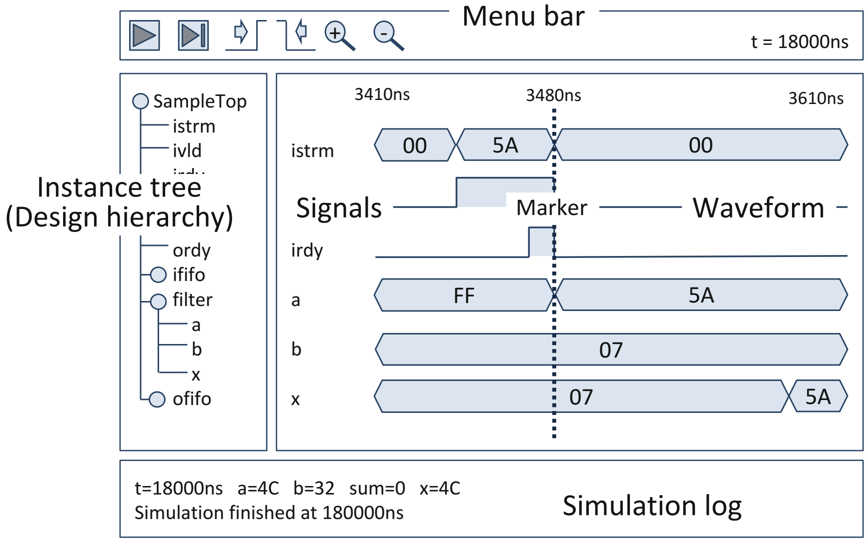
degenerated and sourceless inputs and conflicted outputs, if any, bring about errors or warnings. Because even a bug against the given specifications might be resolved automatically by the default policy, a designer must carefully confirm the messages from the design tools in the debugging. Furthermore, because logics might be modified in the optimization process of synthesis, the designer must notice the fact that the module, registers, and wires described in the source code might disappear in the resultant netlist.

4.2.3 RTL Simulation

Simulation with RTL descriptions of the designed object and the test bench is designated as RTL simulation. It is executed to verify that the behavior of the designed circuit confirms the given specifications. To debug the function and behavior efficiently and to evaluate the performance correctly, the designer must prepare an adequate test bench by considering simulation scenarios. Simulation tools usually include a compiler for simulation, a simulation engine, and a waveform viewer. The compiler analyzes the source code and generates an intermediate code for the efficient execution of the simulation. The simulation engine generates and processes time series events of circuit operation according to the intermediate code. Some tools directly generate a native code for the computer that runs the simulation from the source code. For simulation, a designer might set the termination time and break points (or triggers) and control the break and continue. The messages by the output statements in the simulation source code are displayed on the console and stored in a log file. The result of the simulation is saved as time series data of signal transitions. The waveform viewer displays the waveform of the signal transitions in time series. An example of a typical screen organization of the waveform viewer is shown in Fig. 4.5. Users might select observed objects from the instance tree and set the radix (high/low, binary, decimal, and hexadecimal), number of digits, group, and position for each signal. A designer might move and scale the time axis, set a marker, and search for a signal transition.

Several simulation models exist, with different levels of detail.

(1) Direct simulation of the behavior of the RTL descriptions,
(2) Simulation by the synthesized netlist,
(3) Simulation reflecting the result of place and route.



```
t=18000ns   a=4C   b=32   sum=0   x=4C
Simulation finished at 180000ns
```

Fig. 4.5 An example of waveform viewer

Model (1) can confirm the validity of the function and behavior specified by RTL description. Model (2) can be executed after logic synthesis. It can confirm the timing of the signal transition and delay of the operations based on the estimated delay of each logic and storage element. In addition, it might analyze the power consumption by examining the frequency of the signal transition. Model (3) can be executed after place and route. It can estimate the delay in wiring using the place and route results and can provide the most detailed timing and power analysis reflecting the estimation. Because a detailed simulation requires a long computation time, designers must select the suitable model in accordance with the intentions.

4.2.4 *Place and Route*

During the processes of place and route, a designer maps objects in the netlist to logic and routing resources on an FPGA chip. Usually, the designer processes logic elements, first, and then connections. The place and route processes are outlined in Fig. 4.6. Although the design tool does the placement while estimating the congestion of wiring and delay of signal propagation, it might encounter difficulties that block routing in some connection and that violate the requirement of estimated signal propagation delay. In such a case, a designer must retry the processes of place and route changing the parameters to control the processes based on failed connections, to avoid the congestion and assign higher priority to the connection that is apt to be lengthy. In today's development environment, the processes of place and route are

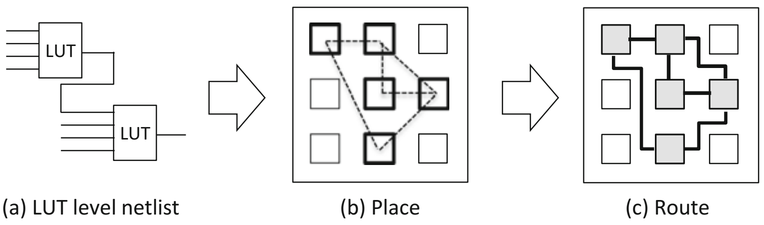


Fig. 4.6 Place and route from LUT level netlist

largely automated and all the processes, including the retries, are completed automatically with just one click. The processes of place and route demand long computation time. A larger circuit causes fewer margin of resource usage, and eventually much longer computation time. In worst cases, it leads to failure in routing. If repetitions of place and route fail to find a feasible solution, then a designer must explore optimization options as explained later, rewrite the source codes with a refined architecture and algorithm, or replace the target FPGA device to larger one.

4.2.5 Programming

The completed circuit is saved as data to program the logic and routing resources on the FPGA device. It is designated as configuration data or a bitstream or a program file in the format of bit, sof (SRAM object file), or others. Figure 4.7 presents an example of the process that generates the configuration data from the place and route results. In FPGAs, the placement of a logic element corresponds to the contents of a programmable logic element at a given position, and the routing of a connection corresponds to a series of on/off values of the switches along the route. The configuration data for the target circuit is generated by gathering such data for all the logic elements and connections. A designer uses a tool designated as a programmer (or designated as program tool) to write the configuration data onto a device. There are several methods to write configuration data as follows.

(1) Directly write to the FPGA device by JTAG,
(2) Write via non-volatile memory dedicated to programming,
(3) Write via a standard memory card or USB memory.

Configuration methods described above are depicted in Fig. 4.8. Because the supported methods differ among FPGA boards, a designer must consult the manuals for the board the designer uses.

(1) Joint Test Action Group (JTAG) is a standard protocol for programming and debugging of printed circuit boards (PCBs). A designer connects the board to the computer (PC) where the programmer or debugger runs using the designated cable or a USB cable. The configuration data is directly written to the target FPGA device using the program tool. This is an easy and simple way but the

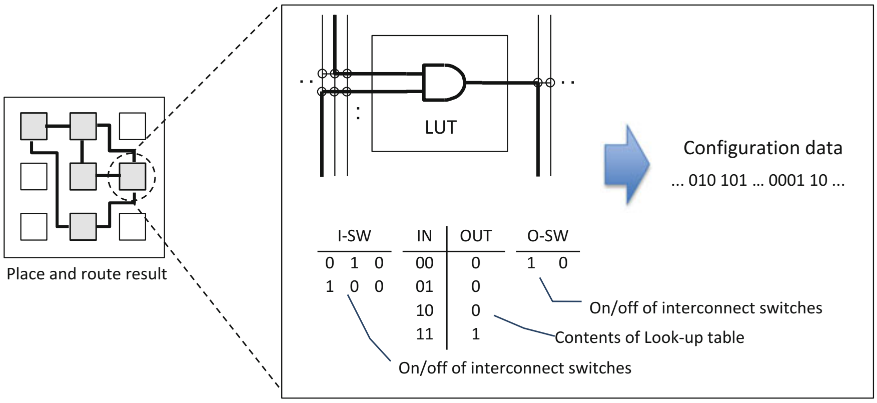


Fig. 4.7 An example of the generation process of configuration data from place and route result

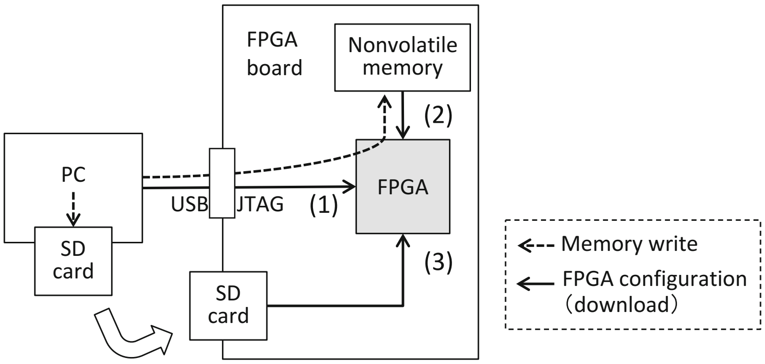


Fig. 4.8 FPGA configuration methods

programmed circuit vanishes on power-off or reset. Debugging on the actual device described later requires this connection.

(2) If the target board has non-volatile memory for FPGA programming (onboard ROM), then a designer might use it. There are several types of memories for programming in terms of the interface standard, bit width, capacity, technology, and other factors. Flash memory with the serial peripheral interface (SPI) is an example. The configuration data is written onto the target FPGA from the programming memory on power-on or reset. There are two modes of programming, passive and active. In the passive mode, the memory controls writing onto the FPGA. In the active mode, the FPGA controls reading out from the memory. Preliminary to writing to a memory device, a file for the memory device is generated from the configuration data. There are several file formats for memory devices including mcs and pof (programmer object files). The programming memory is typically written to through JTAG. Although this programming method is more complicated and takes longer time compared to the direct programming,

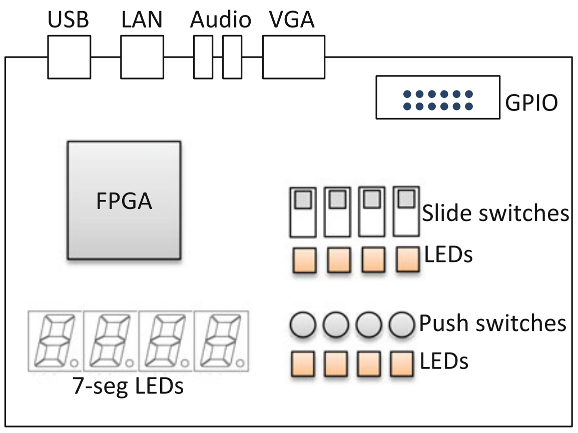


Fig. 4.9 An example of FPGA board for education or training

the programmed circuit boots up just as a board alone without any other equipment. A number of devices such as FPGAs and programming memory devices can be connected to JTAG in cascade. Depending on the type of memory and interface, configuration data of several FPGAs and additional data can be stored in one memory device.

(3) Some FPGA boards are equipped with a microprocessor and a memory card slot or a USB connector controlled by the processor. The designer copies the configuration data to a memory card or USB memory and inserts it to the board. Triggered by the power-on or reset, the processor reads the configuration data from the memory device and writes to the target FPGA device. The designer must be careful about the requirements for the format of the memory device, the file name, the location of the file, and others. Although this method needs manual operation steps of file copying and inserting/removing memory devices, no special cables are needed and the target circuit boots up just as a board alone.

4.2.6 Verification and Debugging on Actual Device

After the target FPGA is programmed with the configuration data, it is ready to verify the behavior and performance on the actual device. Typical educational boards have the best collection of basic I/O devices like light-emitting diodes (LEDs), buttons, and switches, as illustrated in Fig. 4.9. The designer might use these devices to verify the expected behavior of the designed circuit. If the probe pins/connectors or general purpose inputs and outputs (GPIOs) are available on the board, then the designer might make more detailed verifications by connecting measuring instruments such as an oscilloscope and logic analyzer.

A designer might conduct more detailed analysis by adding a special function to the designed circuit in order to probe and record the states and signal values. The

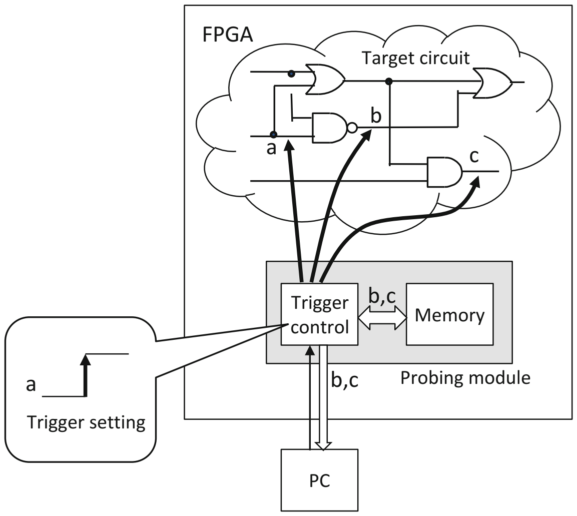


Fig. 4.10 An example of signal probing module

function is incorporated as a circuit module of the designed object and works without using the special measuring instruments. Recent design tools supply such a probing function as a library. The designer might conduct the analysis of signal transitions by setting the trigger conditions (the condition to start recording). Basic triggering conditions include the value or rising/falling edge of a signal and the combinations of these conditions (AND/OR conditions). As a sophisticated triggering condition, a designer might set a sequential pattern of signal transitions.

Figure 4.10 illustrates an example of signal proving. In the verification, a designer specifies the candidates for signal to be observed (signals a, b, and c in the figure). A designer might specify the observed signal in the source code by some special notation or does it by finding and selecting the signal instance in the synthesized netlist. Attention must be devoted to the latter because the target signal might have degenerated or been renamed during the optimization. Then, the designer adds a probing module to the designed object. The designer selects the type of trigger functions (for signal values only or sequences of signal transitions) and sets the buffer memory size to store the observed signals. More sophisticated trigger functions and/or larger buffer memory will make the probing circuit larger, adversely affecting the originally designed circuit. The designer should select the necessary and sufficient functions and size. Because the observed signal is recorded in synchronization with a clock signal, the designer must indicate the clock domain.

The verification and debugging on the actual device are done using a PC connected to the board through JTAG. The designer sets a trigger condition (the rising edge of signal a in the figure) in the probing tool on the PC, switch the probing module to stand-by (the state of waiting for the triggers), and runs the board. Then the result of the probing is transferred to the PC and shown. Similarly with the waveform viewer in the simulation, the designer can set the display format and signal position, move and scale the time axis, and handle the markers. Consequently, the designer makes the verification and debugging based on the observed behavior of the actual board.

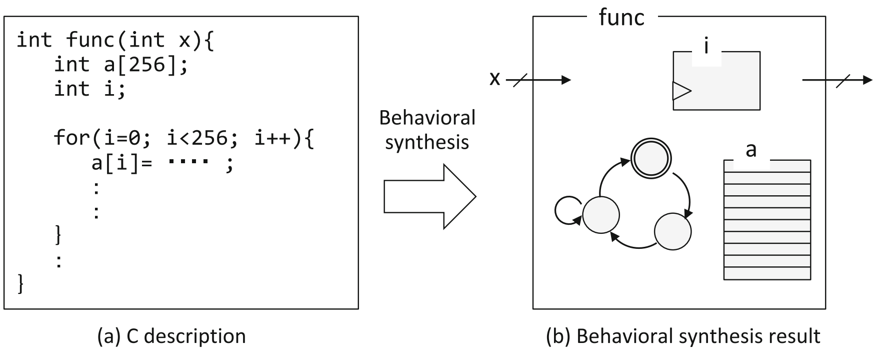
4.2.7 Optimization

The circuit design for a target function has options in general. There are a number of different results in each stage of logic synthesis, technology mapping, or place and route. They might exhibit different characteristics in terms of maximum operating frequency, circuit size, and power consumption. They mostly share a mutual trade-off relation. For example, raising the operating frequency invites an increase of the circuit size.

Although a designer might wish to find the best design among all candidates that fulfill all constraints in all respects, selecting the superior candidate from all possible ones is practically impossible. Then, the designer modifies a candidate to improve toward the target. This process of improvement is designated as design optimization. The parameters to control the optimization differ depending on the algorithm used in each design tool, and the details to tune the parameters are mostly too complicated for ordinary users or are not disclosed to users. In most design tools, a set of options of abstracted design policies (target, objective, or strategy) for optimization are supplied to users as easy-to-use means. Typical policies include assigning priority to accelerating the operating frequency, reducing the circuit size, and reducing the power consumption. If the resultant design fails to attain the target specification, then a designer must explore the optimization options, rewrite the source codes with a refined architecture and algorithm, or perhaps replace the target FPGA device to a better one.

4.3 HLS Design

Digital circuits have been conventionally designed using schematics with logic gates, and then by RTL descriptions. These approaches support manual optimization to achieve high performance. However, they are time consuming and might induce human errors. Consequently, several studies and developments have aimed at highly abstracted design schemes. Nowadays, design technology called high-level synthesis (HLS) or behavioral synthesis is available where a description of the target behavior is synthesized into circuitry [9–13]. This section introduces the design and tools for behavioral description and behavioral synthesis.



```
int func(int x){
   int a[256];
   int i;

   for(i=0; i<256; i++){
      a[i]= ···· ;
      :
      :
   }
   :
}
```

Fig. 4.11 Hardware instance generation by behavioral synthesis from C description

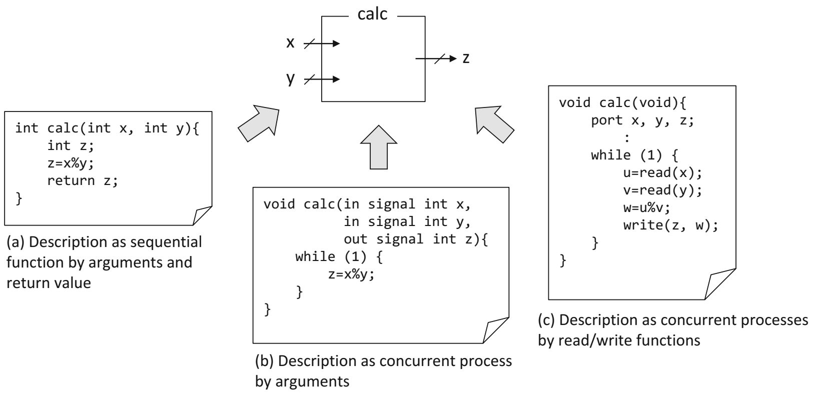
4.3.1 *Behavioral Description*

Many currently available HLS environments adopt C language or some a variation of C as a behavioral description language; however, a C source code just described as a software program might induce poor performance in the synthesized hardware module. The worse case is it may not fails to be synthesized into hardware. Therefore, a designer should carefully describe his/her source codes considering the hardware generated by HLS. Similarly to software programming, a designer describes the behavior of the target using variables, operators, substitutions, control statements such as "if," "for," and "while," and function calls. In general, an HLS environment realizes a variable as a register, an array as a memory, and a function as a hardware module. Furthermore, the control including sequential executions, branches, loops, and function calls is realized as a state machine. Figure 4.11 shows that the behavioral description by C language (left) is synthesized into a hardware module (right) where variable i, array a, and the control flow are realized, respectively, as a register, a memory, and a state machine.

Each HLS environment has specific restrictions for program coding to be synthesized into hardware. Although it depends on each tool and language processor, in general, most HLS tools have common restrictions such as the following, identified at the time of this writing.

(1) Recursive calls are not allowed.
(2) Dynamic pointers are unavailable.

A recursive call of a function implies that the corresponding hardware module is dynamically instantiated at run time, which is beyond the concept of present digital circuits. Therefore, most HLS environments prohibit recursive calls. A dynamic pointer arbitrarily changes the value (i.e., the address) at run time. In contrast to software programs having a large, shared, and identically monolithic main memory, it is a quite normal strategy for hardware designer to localize each memory within modules sharing data to improve the performance. A dynamic pointer may cause



```
int calc(int x, int y){
    int z;
    z=x%y;
    return z;
}
```

(a) Description as sequential function by arguments and return value

```
void calc(in signal int x,
          in signal int y,
          out signal int z){
    while (1) {
        z=x%y;
    }
}
```

(b) Description as concurrent process by arguments

```
void calc(void){
    port x, y, z;
        :
    while (1) {
        u=read(x);
        v=read(y);
        w=u%v;
        write(z, w);
    }
}
```

(c) Description as concurrent processes by read/write functions

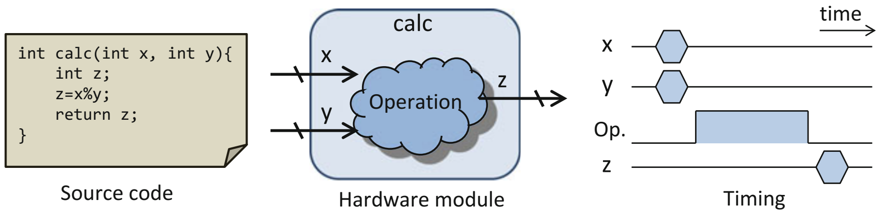
Fig. 4.12 Typical methods of describing I/O

switching memory instances accessed at run time. Since this capability is also beyond the concept of present digital circuits, it is prohibited in behavioral synthesis systems.

(1) Describing input and output
The manner to describe hardware I/O is disputable. In a software function, an input is given as an argument when the function is called and an output as a return value (or an argument by reference) when the function terminates. In a hardware module, on the other hand, the module stays at any time; I/O goes on occasionally. The working principle differs completely between software and hardware. There are several methods of describing I/O, as illustrated in Fig. 4.12. They are explained below one after another. The methods differ among HLS environments.

(a) Description as sequential function by arguments and a return value
If a designer wishes to sequentially call, run, and terminate as in software, then the designer can describe the process similarly to a usual function. An argument of the function corresponds to an input of the hardware module and the return value to the output. Figure 4.13 shows that the arguments `x`, `y`, and return value `z` in the source code (left) correspond the I/O ports in the hardware module (middle), as well as the sequential operational timing of input, operation, and output (right). An argument by reference is used occasionally instead of the return value.

(b) Description as concurrent process by arguments
Some HLS environments have special variable types for I/Os or a way to specify I/O types of variables such as "pragma." A variable for hardware I/O is read or written anytime from outside of the module. An example of such a description is depicted in Fig. 4.12b. Although it seems to be a meaningless code in a normal software program, `z` changes in accordance with occasional changes of `x`, `y` driven from outside. A designer describes



```
int calc(int x, int y){
    int z;
    z=x%y;
    return z;
}
```

Fig. 4.13 An example of arguments and return value in a source code, I/O ports in a hardware module, and operational timing of input, operation, and output

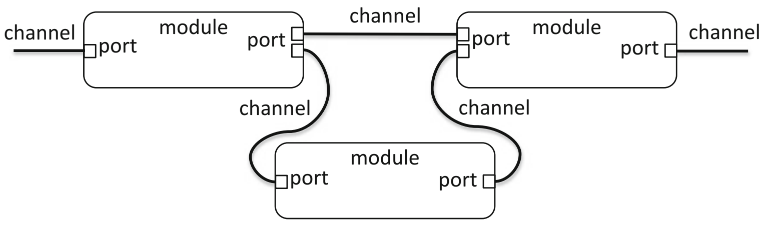


Fig. 4.14 Architecture model composed of modules and channels

an event-driven infinite loop (while (1) {}), typically. Here, one parallel process in software corresponds to a module in hardware. The behavior of the designed functions is described as a model in which I/O ports of each module are connected by communication channels, as shown in Fig. 4.14. The idea of this kind of description is close to statements of "always @ (posedge clock)" or "process (clock)" in HDL code. It is also close to a concurrent software program using a variable with volatile modifier, which can be mapped to an I/O port or shared among multiple processes or threads.

(c) Description as concurrent processes by read/write functions
Some HLS environments have a library of I/O functions using a similar idea to that discussed above in (b). The library supports variable types for identifiers of I/Os and functions of read, write, open, and close for the identifiers, as in programming using files or sockets. A program iterates to read the inputs, executes the operations, and writes the output. Figure 4.12c shows such a description.

The types of I/Os are categorized, in general, into the register type and the stream type. The register type I/O merely holds the last-written value and the value can be read anytime. The stream I/O do handshaking to wait for data transmission and receipt. In the example of Fig. 4.15 (top), the register type I/O is showing 0 or 1 at any state. The example of Fig. 4.15 (bottom) shows the data transfer of the sequence 0, 0, 1, 1, 0, and 1. It is noteworthy that a state (a clock cycle) without data transfer, i.e. an idle state exists. In that state, either the transmitter

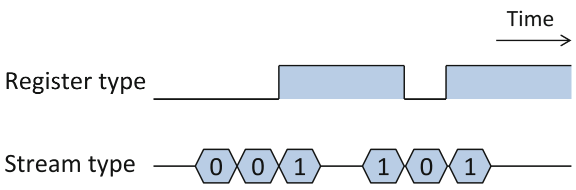


Fig. 4.15 Register type I/O and stream type I/O

or receiver is busy. The program stalls at a read/write statement to wait for the corresponding I/O.

(2) Bit width
In the standard C language, 8-bit character type, 32- or 64-bit integer types, and 32- or 64-bit floating point types are supported as predefined variable types. In FPGA, any bit width and fix-point type can be implemented. By specifying the necessary and sufficient bit width in each part of the program, a designer can optimize the circuit size, operation speed, power consumption, and operation accuracy. For that reason, one can specify the bit width of a variable in details in most HLS environments. Conversely, finding an efficient bit allocation is up to the designer.

(3) Describing parallelization
An HLS environment analyzes the flow of operations and dependency between data and determines the schedule of operations. It automatically schedules such that mutually independent operations run in parallel to an extent. However, speeding up by parallelization increases the circuit size. This trade-off demands consideration. Although the HLS environment automatically analyzes the dependency between statements and a small simple loop, analysis and optimization of global parallelism are difficult, even for the current compiler technology. Consequently, many HLS environments supply descriptive methods for a designer to specify the parallelization. A designer can provide directions for the pipelining of a loop and the parallelization of a block by a "pragma," "directive," or some extended instructions.

Figure 4.16 shows a sample description of loop processing for arrays. When a designer synthesizes the hardware by simply applying the description as it is, the designer obtains the hardware like presented in Fig. 4.17a and the sequential processing depicted in Fig. 4.18a. If the designer gives an instruction of pipelining for this for-loop, then the designer obtains the synthesized hardware to improve the speed performance by pipelining, as in Fig. 4.18b. Furthermore, if the designer designates an instruction of streaming I/O, he/she obtains the hardware without I/O memory, shown in Fig. 4.17b, to achieve further improvement of speed performance by pipelining including I/O, as illustrated in Fig. 4.18c. Conversely, instructions to suppress parallelization are also supplied and, for example, can be used to maintain the clock-wise order of operations.

```
int calc(int x[N], int y[N]){
    for (i=0; i<N; i++) {
        u = {Process A for x[i]};
        v = {Process B for u};
        y[i] = {Process C for v};
    }
}
```

Fig. 4.16 An example of loop processing for arrays

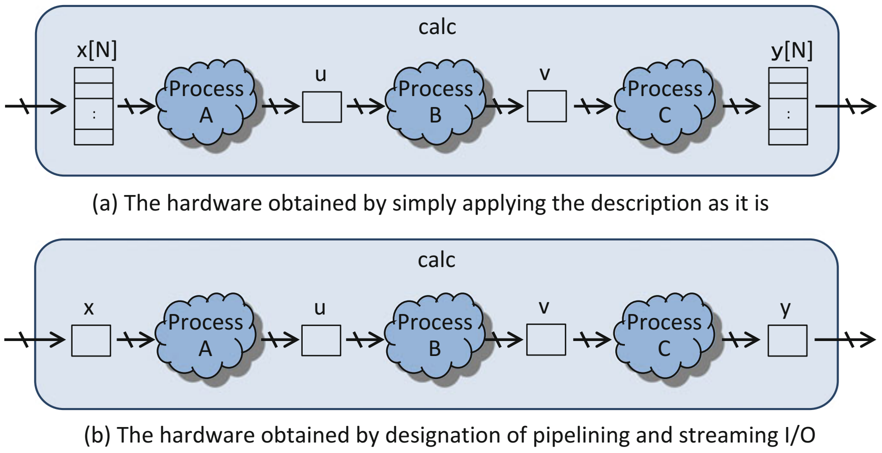
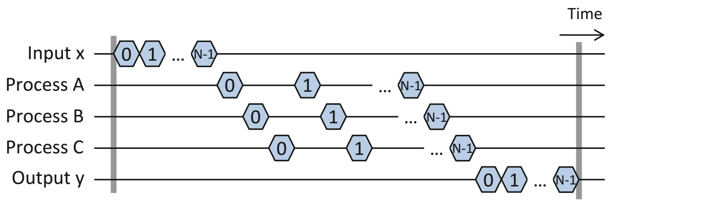

calc
x[N]
Process A
u
Process B
v
Process C
y[N]
(a) The hardware obtained by simply applying the description as it is
calc
x
Process A
u
Process B
v
Process C
y
(b) The hardware obtained by designation of pipelining and streaming I/O

Fig. 4.17 Synthesized hardware

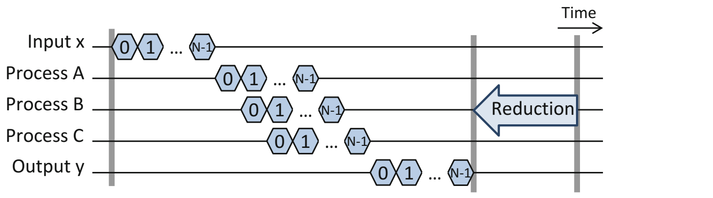
4.3.2 Behavior Level Simulation

The designed module described in C is firstly compiled as a software program to verify its behavior and function. This process is designated as the behavior level simulation. In contrast to the RTL simulation emulating event-driven clock-wise operations, it is significantly faster because the function is executed in the native machine code of the computer that runs the simulation. Consequently, the designer may repeat debugging and improvement of the code in a short iteration period. However, because this process does not take care of accurate timing, the designer cannot examine the behavior considering cycle accuracy. Particularly, in cooperative operation with external devices or with multiple modules, the designer must devote attention to the possibility to obtain different simulation results than those obtained by an actual device in terms of communication delay, operational dependency, and clock-wise order between modules.

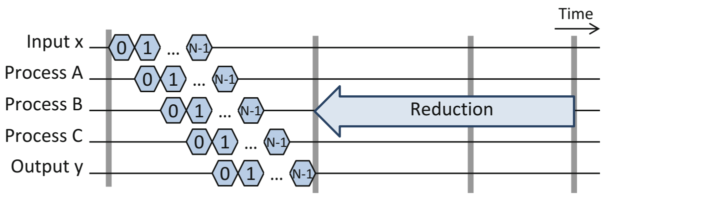
The computer performing the simulation rounds up the bit width of data to a standard one, such as 8-bit “char” or 32- or 64-bit “int,” to speed up the simulation. For this reason, the behavior in the behavior level simulation and the actual device may mutually differ when a calculation produces overflow against the bit width a designer has set. Similarly, the behavior differs when the index of an array exceeds



(a) Sequential operation timing obtained by simply applying the description as it is



(b) Improvement of speed performance by pipelining of for-loop



(c) Improvement of speed performance by streaming I/O

Fig. 4.18 Operation timing of the synthesized hardware

Fig. 4.19 An example of C description including a potential bug

```
uint8  a[16];
uint4  i;

for (i=0; i<16; i++)
    a[i]=i;
```

the range. In the example portrayed in Fig. 4.19, the "for" loop becomes an infinite loop (that may be a bug) because the maximum of the variable "i" is 15 limited by the bit width of 4 in the actual device. However, in the behavior level simulation, 4 bits "i" may be rounded up to 8 bits or more and the loop finishes when repeated 16 times. A designer must devote attention to this point, including intermediate results. Although simulators (or simulator modes) which accurately emulate the bit width are also provided, they run slower.

4.3.3 *Behavioral Synthesis*

The core process of HLS environments is the high-level synthesis (HLS) or the behavioral synthesis to generate an RTL description from a behavioral description in C and others, as outlined in Fig. 4.20. In the behavioral synthesis, a variable, an array, and an operation in behavioral description, respectively, are transformed into a register, a local memory, and an operation unit in RTL. The flow of operations in the behavioral description (sequential operations, branch, and loop) is realized as a state machine. In the behavioral synthesis, a data flow graph (DFG) representing dependency among the data of operations and a control flow graph (CFG) representing the flow of operations are generated based on the analysis of the given behavioral description. The process to determinate the order and timing of operations is referred to as scheduling, and to assign each variable and operation to register and operation unit as binding. The registers and operation units are connected via multiplexers, and this part of the circuit is designated as a data path. A state machine is generated to control the data path by switching the multiplexers following the schedule. The scheduling and binding have options for the amount of hardware resources and the time to complete the operations, but they share a mutual trade-off relation. Each HLS environment provides an optimization strategy to control the trade-offs.

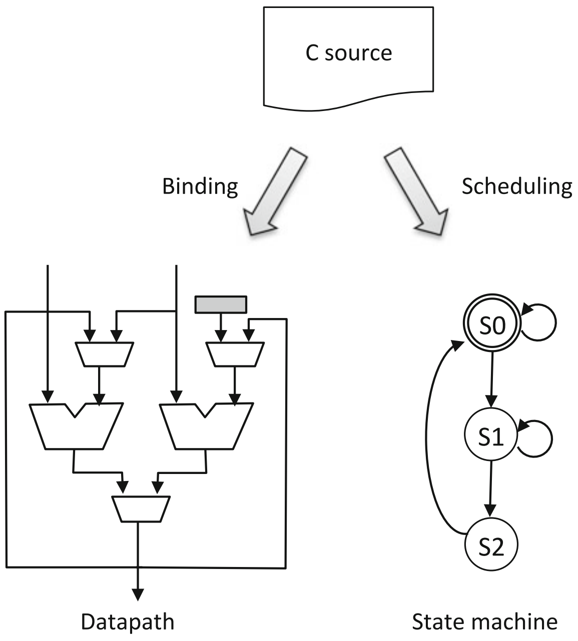


Fig. 4.20 Outline of behavioral synthesis

4.3.4 Evaluation and Optimization

A trade-off relation exists between the required hardware resource and operation time. Generating the best-suited RTL description for the given design specifications, constraints, and objectives is an extremely difficult problem. In the present technological level of behavioral synthesis, a designer must provide detailed guidelines. For that reason, most HLS environments provide measures to help designers analyze and evaluate the design. HLS environments also provide predefined options of optimization policies or strategies to balance the trade-offs. When a designer performs behavioral synthesis, the designer obtains performance metrics as listed below.

(Performance metrics for hardware resource)

- Number of operation units,
- Memory size,
- Number of registers,
- Amount of logic resources such as LUTs (estimation).

(Performance metrics for speed and timing)

- Throughput,
- Latency,
- Timing of each operation,
- Maximum operation frequency (estimation).

A designer uses these results of analysis to establish a balance between the required resources and speed and to identify and characterize countermeasures against bottlenecks and over-performance. A simple way to lead the behavioral synthesis to the designer's preferable result is to set the optimization policy parameters of the synthesizer; for instance, the priority levels for the size and speed, the limit of operation units used and the target throughput and latency. Giving an instruction by "pragma" or "directive" for the source code is a more detailed method. A typical instruction is to dictate pipelining or unrolling for the for-loop in the code. Other instructions include parallelization or sharing (iterative use of shared resource) of operations, memory partitioning and interleaving, fattening of branches and inlining. If all of these methods fail to achieve the requirements for the designed module, then the designer must rewrite the source code while seeking better algorithms and architectures. Although a microscopic optimization for operations is done automatically by the synthesizer, macroscopic optimization for algorithms and architectures is up to the designer. He/she should consider the efficient architecture of the designed module and write a code that the synthesizer can perceive the architecture including parallel processing, pipelining, and others.

4.3.5 *Connection with RTL*

The behavioral synthesized module is handed to the logic synthesis and later design flow, together with modules described in RTL and others. In the conventional design flow, the behavioral synthesized module is instantiated in the upper module described in RTL. In the recent design flow, it can be integrated with other modules including a processor using the IP integration tool, not describing RTL code. The types of the interfaces, the signals of each interface, and the naming rule of the signals are prescribed in each HLS system. The interfaces are generally categorized into three types: a register type where the data value is read and directly written, a stream type where a sequence of data is transferred one by one, and a memory bus type where data is read and written being located by an address. Figure 4.21 shows how the interface of each type of data handling is indicated.

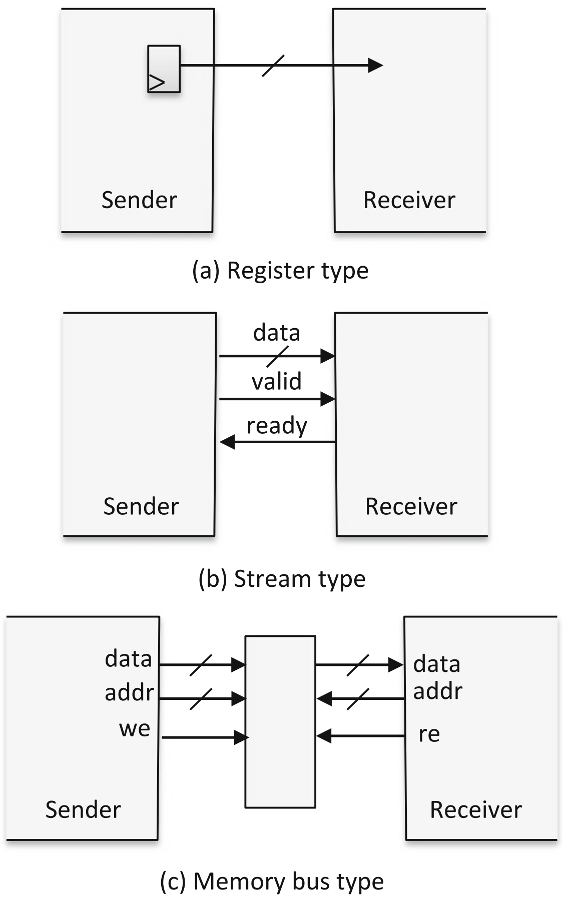


Fig. 4.21 Three types of interface

The interface for the register type consists of just a register. The sender writes the data by asserting a write control signal. The receiver merely reads the data at any time, as shown in Fig. 4.21a. The receiver cannot perceive, in principle, when and how many times the sender wrote.

The interface for the stream type consists of a data signal and control signals of sending and receiving, as depicted in Fig. 4.21b. The sender asserts a control signal ("valid" in the figure) to enable/disable sending data. The receiver asserts a control signal ("ready" in the figure) to enable/disable receiving data. If both the control signals of sending and receiving are enabled at a clock cycle, then the data transfer is enacted and the receiver receives the data at the clock cycle, otherwise, the communication stalls until the condition stands. The data values are sequentially handed over one by one by this control. The stream type interface is used in data-driven processing and is often connected through a FIFO buffer to cope with the variation of data flow speed. It is noteworthy that the meaning of the control signals depends on the standpoint. For the sender, the "valid" signal is a request and the signal "ready" is an acknowledge or a response. For the receiver, "ready" is a request and "valid" is an acknowledge. A request signal is typically named as "valid," "enable," "strobe," "run," and "do." An acknowledge signal is typically named as "ready," "acknowledge," "busy," and "wait."

The interface for the memory bus type consists of data, address, and the control signal of read/write, as shown in Fig. 4.21c. The sender and receiver are connected via a memory in this interface. The sender sets the data signal and address ("data" and "addr" in the figure) signals and asserts the write enable signal ("we" in the figure) to write the data. The receiver sets the address signal and asserts the read enable signal ("re" in the figure) and then reads the data signal after some clock cycles.

4.4 IP-Based Design

The size and complexity of digital systems are growing day by day, bringing about difficulties related to long terms and huge costs of development. There are common modules among many designs, such as interface, peripheral control, communication, cipher coding, data compression, signal and image processing. A designer may efficiently reduce the cost by reusing previously designed modules. These common and reusable design resources are designated as intellectual properties (IPs).

4.4.1 IP and Its Generator

A source code of previously designed module is an IP itself. It enhances the reusability if the source code is scalable and parameterized: for example, bit width, buffer size, and number of elements. There are tools to automatically generate source codes for given parameters and conditions (designated as IP generator, IP wizard, and others) [14–17]. The tools also generate a template code to utilize the generated IP. IP generation tools for fast Fourier transformation unit (FFT), for example, accepts the parameters of the block size, bit width of data, output ordering, target operating frequency. Recent FPGAs have a variety of hardware resources proper to each vendor and having specific features: memory, arithmetic operation block, PLL, and high-speed transceiver. FPGA vendors supply IP generators to generate modules to utilize such resources.

An IP is not merely reused by the designers, their team, and company, but also distributed as commercial products. FPGA vendors supply a variety of IPs and IP generators as options of their integrated development environment. In addition, there are some third parties supplying their own IPs as commercial products and a designer might buy and use them. An IP may be supplied as a netlist after synthesis or circuit data after place and route in view of protecting the IP. In this case, codes for simulation, designated as a behavior model, is supplied in addition to the protected IP.

4.4.2 Use of IP and Its Integration Tool

The first job for a designer to use IPs is finding appropriate IPs for the target design, listing up candidates, and scanning their specifications to select a module dovetailing with the conditions. Finding IPs is time consuming and IDE vendors might be expected to develop an efficient tool to support retrieval from IP database. In traditional practice, the designer finds a candidate, instantiates it as a module, and connects it wire-by-wire to the designer's own circuit description.

Recently, such an IP integration tool is available to help a designer arrange IP modules graphically. In contrast to conventional graphical design tools in the lower abstraction level of simple logic gates, flip-flops, operators, registers, and multiplexers, the IP integration tool handles a block diagram consisting of IPs having more complicated functions. Furthermore, the designer needs not to connect IPs by wires but by interfaces that are composed of a set of wires including data and control signals. For example, an interface of memory bus type composed of data, address, write enable, and read enable signals can be regarded and handled as single entity.

Figure 4.22 shows an example design by block diagram using a processor, a GPIO, an UART, and an FFT. Here, the peripherals of a processor (uProc in the figure) and other peripherals such as GPIO are connected by a memory bus (Bus interconnects in the figure) with an address. FFT, which has stream type I/Os, is connected to the direct memory access (DMA) controller via FIFOs. Consequently, the design productivity considerably improves when compared to when describing wire-by-wire connections after giving unique names to each wire. I/O pins of the FPGA chip may be registered as external access points in the IP integrator and then the designer might be released from the chores of the top description in RTL.

IP integration tools of a high abstraction level provide the designer with a systematic structure of interfaces with properties, such as the interface type, bit width,

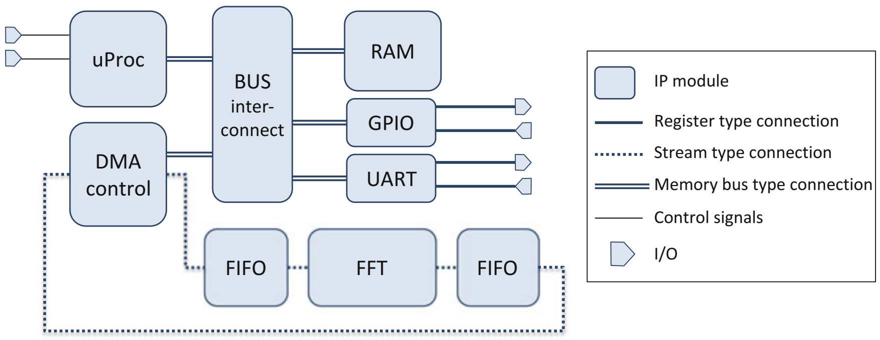


Fig. 4.22 An example of design by block diagram using IP and its integration tool

associated clock signal with its frequency, associated reset signal with its polarity, and allocated address space. Based on this information, the IP integrator supports the detailed design including compatibility check and automatic address allocation. However, these tools have not been standardized yet; they are vendor-dependent or tool dependent. Integration technology with a high abstraction level is in the process of development and has been improving day by day.

4.4.3 Support Tool for Building IP

A designer not only uses IPs as a user, but can also register the designer's own creation as an IP. Consequently, the designer enhances the reusability of his/her own module and might sell it as a commercial product. If the designer equips the module with an interface in accordance with the specifications in an IP integration tool, then the designer can use it in the integration tool. In today's design environment, templates of the predefined interfaces, and tools to make an owned module parameterized and packaged as an IP are available. For an array of arguments in a behavioral description, the interface of the stream type or memory bus type may be synthesized in accordance with the specification of IP integration tool. The synthesized module may be exported to the IP integrator and used as an IP as it is.

4.5 Design with Processor

It is difficult in terms of cost to build a large and complicated system using hardware modules alone. Consequently, designs combining general-purpose processors and hardware modules on an FPGA chip are popular in practice to achieve the speed performance and energy efficiency and to improve flexibility and productivity.

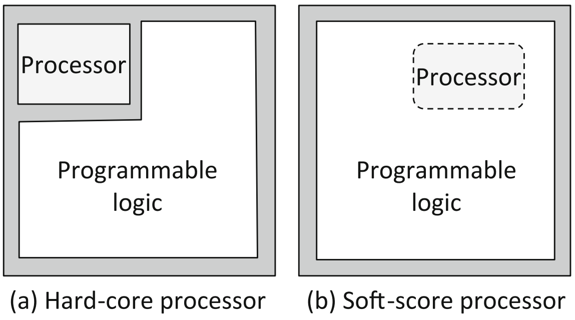


Fig. 4.23 Processors mounted on an FPGA

4.5.1 Hard-Core Processor and Soft-Core Processor

Processors mounted on FPGAs are classified generally into hard-core processors and soft-core processors, as outlined in Fig. 4.23. The hard-core processor is a standard embedded processor inherently mounted on a chip, as depicted in Fig. 4.23a, and has dedicated structure to be connected to the programmable hardware resource in addition to the function and performance of a normal processor. The soft-core processor is a processor programmed on the programmable hardware resource, as shown in Fig. 4.23b. Because a module based on a soft-core processor is doubly programmed (the module is implemented by a software program running on the processor implemented by a hardware program), the performance is inferior to that of a hard-core processor. However, a designer can derive several merits from soft-core processors: It can be used in an FPGA chip with no processor mounted, any number of processors can be mounted as long as the FPGA chip accepts, and the detailed architecture can be configured to fit the target design.

4.5.2 Building Processor System

A designer configures a processor using a designated building tool supplied from the FPGA vendor and builds a processor-based system including memory and peripherals using an IP integration tool. Figure 4.22 in the previous section represents an example of a system configured with a processor.

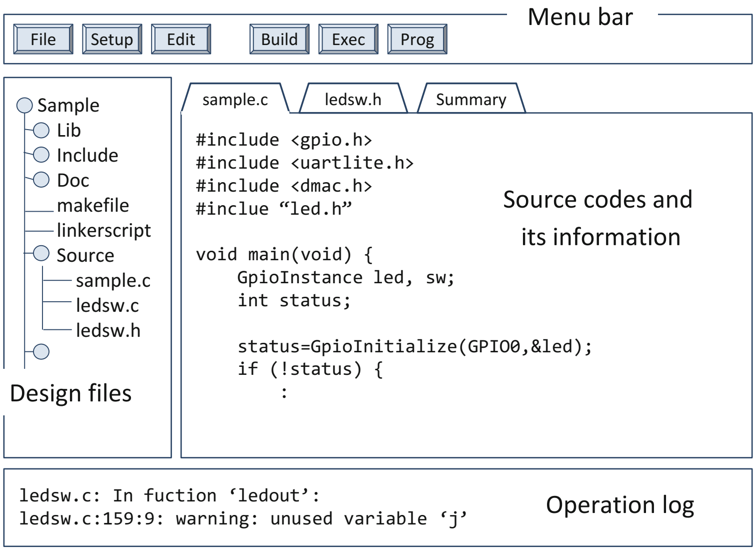
First, the designer selects the processor to be used and sets the parameters. While the parameters in a hard-core processor are limited to the basic ones as operating frequency, that in a software core processor has more options related to pipeline, cache, bus, instruction set, and others in details. The designer determines the configurations of the main processor to fit the target design. Then, the designer builds the bus selected from the standard ones for each processor and connects the processor with memory and peripherals via the bus. The options of bus configurations include operation frequency, transmission mode, hierarchization, and others.

The designer also configures the memory. The hard-core processor is connected to the off-chip main memory through its own (hard-core) memory interface. The soft-core processor is connected to the off-chip main memory through, also, a programmed memory interface. The designer may configure the main memory with on-chip block RAMs instead of an off-chip memory. Peripheral circuits are connected to the bus, similarly to memory. The hard-core processor has some standard interfaces like non-volatile memory and network, and such external peripherals can be connected through the interfaces. In both hard-core and soft-core processors, a designer can place his/her own modules and IP modules in the programmable hardware resources and connect them to the processor via the bus. Then, the address space of each module including the memory and peripherals by which the processor accesses the modules is allocated. The interruption number of the module is allocated if needed. As more advanced configurations, direct memory access (DMA) by a DMA controller and bus hierarchization through the bus bridge are allowed. The generated processor system may be the top instance of the FPGA chip or instantiated as a module in the RTL description.

4.5.3 Software Development Environment

Software drives the processor system. The development environment for software containing compiler and debugger is designated as a software development kit (SDK). Software development requires the information of configuration of the processor, memory and peripherals, mapping of the address space and libraries to control the peripherals (device driver). The set of all the information above is designated simply as the processor configuration. The processor configuration is exported from the tool to build the processor system and imported to the SDK.

In SDK, a development environment in C language is usually supplied. Figure 4.24 presents a common screen arrangement of an SDK as an example. In SDK, a designer selects the operating system (OS) first. Considering the design objectives, the designer might select a powerful one equipped with process management, memory management, file system, and network. The designer might also select a simple one having the minimum functionalities, or even a system without an OS. If the OS used has no functions of memory management, the designer must define the memory mapping specifying the size of stack and heap. The file that defines the memory mapping is designated as a linker script. Using SDK, the designer can describe, edit, and compile (build) source codes. The compilation result is stored in a format referred to as executable and linkable format (elf).



```
#include <gpio.h>
#include <uartlite.h>
#include <dmac.h>
#inclue “led.h”

void main(void) {
    GpioInstance led, sw;
    int status;

    status=GpioInitialize(GPIO0,&led);
    if (!status) {
        :
```

```
ledsw.c: In fuction ‘ledout’:
ledsw.c:159:9: warning: unused variable ‘j’
```

Fig. 4.24 A typical screen arrangement of an SDK

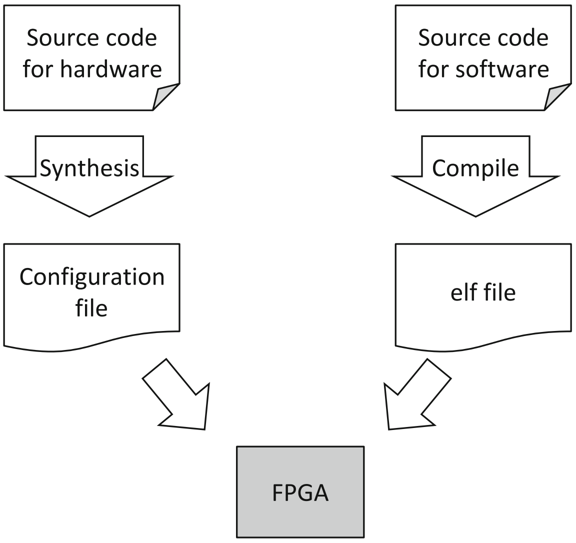


Fig. 4.25 Configuration of hardware and software on an FPGA

4.5.4 Integration and Operation of Software and Hardware

The configuration of the hardware and the software executable are integrated and executed on the FPGA, as shown in Fig. 4.25. There are several ways for this step and they are different among vendors and tools. The following are the typical ones.

(1) Integration and execution in SDK,
(2) Integration on SDK and execution via non-volatile memory,
(3) Integration and execution in a hardware development environment.

(1) A designer exports the configuration data from the hardware development environment and imports it to SDK. After integrating the configuration and the elf file in SDK, the designer writes it to the FPGA or main memory and launches it. Here, the designer can use the software debugger in SDK and can control the execution by break points and observe the variables in the program. This practice is advantageous when the hardware development has been completed and the main target is software development and debugging.
(2) A designer integrates the configuration and the elf file in SDK similar to (1). The integrated files are stored in some kind of non-volatile memory as SD cards, USB memory sticks, flash memory ICs, in the format specified for each device and board. Triggered by the system reset on the FPGA board, the integrated files are loaded from the non-volatile memory and launched following the predefined boot sequence.
(3) If the software program is stored in on-chip memory, the integration and execution may be done in the hardware development environment. The elf file is exported from SDK and imported to the hardware development environment. The elf file is registered as the preloaded content of the on-chip memory instance together with the information of the memory mapping. The generated configuration data can be handled as in the case without a processor. This practice is advantageous when the main target is hardware development and debugging rather than software.

References

An Overview of FPGAs and Their Design Flow

1. S.D. Brown, R.J. Francis, J. Rose, Z.G. Vranesic, *Field Programmable Gate Arrays* (Kluwer Academic Publishers, 1992)
2. V. Bets, J. Rose, A. Marquards, *Architecture and CAD for Deep-Submicron FPGAs* (Kluwer Academic Publishers, 1999)
3. Vivado design suite tutorial: design flows overview, Xilinx UG888, Nov 2015

HDL Design Flow

4. Nexys4 Vivado Tutorial, Xilinx University Program (2013)
5. Vivado design suite tutorial: using constraints, Xilinx UG945, Nov 2015
6. Vivado design suite tutorial: logic simulation, Xilinx UG937, Nov 2015
7. Vivado design suite user guide: programing and debugging, Xilinx UG908, Feb 2016
8. Vivado design suite tutorial: programming and debugging, Xilinx UG936, Nov 2015

HLS Design

9. M. Meeus, K. Van Beeck, T. Goedeme, J. Meel, D. Stroobands, An overview of today's high-level synthesis tools. Design Auto. Embed. Syst. **16**(3), 31–51 (2012)
10. D. Gajski, Z. Jianwen, R. Doemer, A. Gerstlauer, S. Zhao, *SPECC: Specification Language and Methodology* (Springer Science+Business Media, 2000)
11. M. Fujita, SpecC language version 2.0: C-based SoC design from system level down to RTL, Tutorial of ASPDAC (2003)
12. Vivado design suite tutorial: high-level synthesis, Xilinx UG871, Nov 2015
13. D. Pellerin, S. Thibault, *Practical FPGA Programming in C* (Prentice Hall Professional Technical Reference, 2007)

IP Based Design

14. Vivado design suite tutorial: designing with IP, Xilinx UG939, Nov 2015
15. Vivado design suite user guide: creating and packaging custom IP, Xilinx UG1118, Nov 2015
16. Vivado design suite tutorial: creating and packaging custom IP, Xilinx UG1119, Nov 2015

Embedded Processor Design

17. Vivado design suite user guide: embedded processor hardware design, Xilinx UG898, Nov 2015

Chapter 5
Design Methodology

Masahiro Iida

Abstract A typical misunderstanding is that design automation techniques used in FPGA tools are just subsets or extension of those developed for LSI design tools. In reality, a unique FPGA structure requires original design automation techniques, and they are critical to extract enough performance with a limited resource on an FPGA chip. This chapter introduces them from the viewpoint of CAD (Computer Aided Design) developer. Techniques for technology mapping, clustering, and place and routing are introduced. Then, low power design which has become a critical issue is introduced. Although this chapter includes some expert knowledge, even beginners can understand the specialties and challenges of FPGA tools.

Keywords FPGA EDA tools · Technology mapping · Clustering · Place and route · Low power FPGA design techniques

5.1 FPGA Design Flow

EDA (Electronic Design Automation) technology is extremely important to bring out the performance of LSIs. In general, the upper limit of the performance that the FPGA can achieve is limited by physical restrictions such as process technology. However, the performance that the actual circuit can achieve depends on the device architecture and the EDA tool. That is, no matter how high-powered engines (processes) we have, there is no speed without a car body (architecture) and driving skill (EDA tool) to match them. In particular, the EDA tool is directly related to the implementation of the circuit; thus, the influence on the performance is immeasurable.

M. Iida (✉)
Kumamoto University, Kumamoto, Japan
e-mail: iida@cs.kumamoto-u.ac.jp

H. Amano (ed.), *Principles and Structures of FPGAs*,
https://doi.org/10.1007/978-981-13-0824-6_5

Figure 5.1 shows the FPGA design flow. It begins with Logic Synthesis of HDL (Hardware Description Language) source code and ends with Bit stream Generation after Technology Mapping, Clustering, and Placement Routing. In logic synthesis, a gate level netlist is generated from the HDL description, and in the technology mapping, this netlist is converted into a netlist at the LUT level. Clustering is a process of combining a group of LUTs and flip-flops (FFs) into one logical block (LB: Logic Block). Then, LB is placed on the device by the place and route tool, and the connection between the LBs is routed on the wiring structure. Finally, from the arrangement/wiring information, the connection relation of each switch in the FPGA is generated as a bit stream.

In this way, the design of the FPGA is cutting out logics which can be implemented in the LUT having a predetermined number of inputs (the LUT can implement any logical function of the number of inputs) and then allocating on it. Then, by using the wiring which can freely determine the route between them, circuit can be realized on the FPGA.

The difference between FPGAs and ASICs is that ASICs combine elements classified by function in a library, whereas the FPGAs have a uniform structure. The EDA tools of these two technologies are different. The following sections describe the FPGA EDA technology, highlighting the specificities that differs from ASICs, namely for the technology mapping, clustering, and placement and routing.

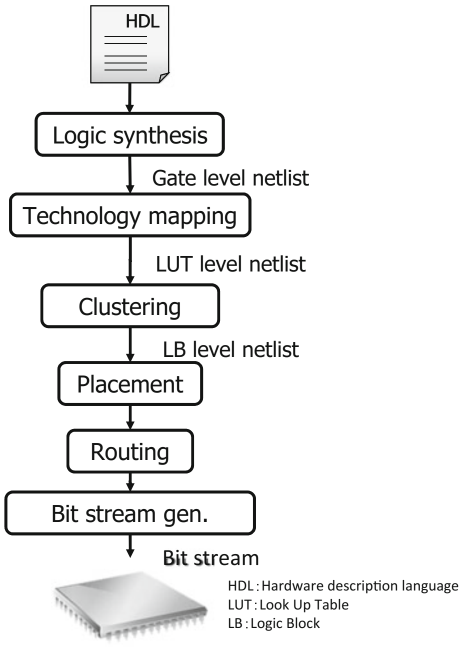


Fig. 5.1 Typical FPGA design flow

5.2 Technology Mapping

Technology mapping is a task of converting a technology-independent gate level netlist to logic cells of a target FPGA. The logic cells referred to here depend on the FPGA architecture and is the minimum unit for realizing a logic circuit such as an LUT and MUX (Multiplexer). The technology mapping is located at the end of the conversion process of logics from HDL. Therefore, the influence of this process on the quality (area, performance, power consumption, etc.) of the final implemented circuit is immeasurable.

Here, we look at the mechanism and behavior of FlowMap [1] which is a representative technology mapping tool. FlowMap is a technology mapping method developed by Cong et al. from UCLA (University of California, Los Angeles). Technology mapping for general k-input LUT (k-LUT) consists of the following two steps:

Step I. Decomposition process: Since the actual gate level netlist is represented by a Boolean network,[1] each node is disassembled until it becomes less than the input number k of the LUT.

Step II. Covering process: The Boolean network decomposed in Step I is converted to reduce the number of inputs based on appropriate criteria, so that several nodes are covered with k-LUT.

FlowMap is a method for obtaining a depth optimal solution of the covering problem of Step II with polynomial computation time.

Figure 5.2 shows the operations of FlowMap, explaining the operation of technology mapping based on the example of mapping to 3-LUT. First, the gate level netlist in (a) is converted to a DAG (Directed Acyclic Graph) in (b). At this time, the uppermost node is called PI (Primary Input) and the last node is called PO (Primary Output). Focusing on the PO on the right side of (c), the nodes related to the PO up to the PI are surrounded by a dotted line. This selection represents the mapping range of PO. Next, (d) illustrates labelling and cutting. Labels are attached in the topological order from the PI, and the conditions are attached as follows:

(1) Set the label of PI to 0,
(2) Next, search the range that can be covered by the 3-LUT among the nodes that take PI inputs, and put a cut on the input.
(3) The label calculation at this time adds its own number of level (that is, level one) to the node with the largest label immediately above the cut, so PI $0 + 1 = 1$, and the label is 1,
(4) Once the label of a node is calculated, we can calculate the labels of the nodes that are connected to the node that calculated the label. However, if the label is not yet determined, it should be done from the finalization of that label first.

[1] A Boolean network is a way of expressing a gate level netlist, represented by a valid graph (DAG). Each node is composed of a logic gate or a combination circuit of logic gates, and the directional branch represents an input/output signal.

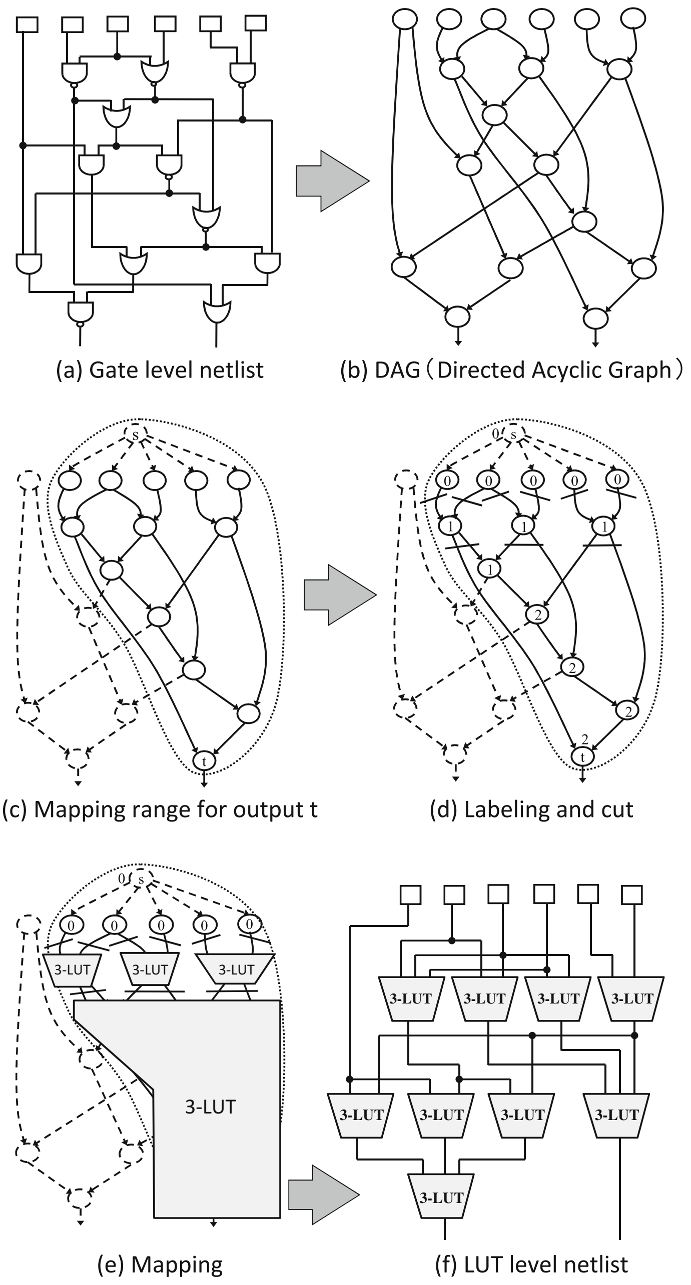


Fig. 5.2 FlowMap operation

(5) After the labels of all involved nodes have been determined, the label calculation of the second level nodes can be performed. At this time, since the remaining nodes can be mapped to a 3-LUT, the labels of these nodes are 1 because all PIs are included in this cut.
(6) When labels are calculated for each node in this manner, the label of *t* finally becomes 2.

The label is guaranteed to be the minimum value since the minimum value is obtained from the upper side of the node. In (e), mapping is done from the PO of the circuit. Then, when executing the above steps for all POs, a final mapping to the 3-LUT is obtained, as shown in (f).

Although cutting and mapping can be performed as described above, various technology mapping methods have been proposed to improve the evaluation function. For example, Cong et al., who developed FlowMap, devised CutMap [2], ZMap [3], DAOMap [4] and methods targeted for depth optimization and the reduction of the number of LUTs, etc. Wilton et al. (Univ. of British Columbia, UBC) developed EMap [5] which considers the power consumption, and Brown et al. (University of Toronto) are developing IMAP [6] . Also, Cong et al. also proposed Hetero Map [7] for heterogeneous LUTs with different numbers of inputs instead of a single LUT.

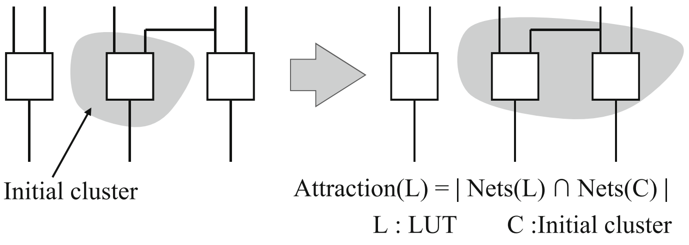
5.3 Clustering

Recent FPGAs are mainly based on cluster-based structures that are grouping multiple LUTs in a logic block. Therefore, the clustering processing has become an essential step. Clustering has two important points. The first one is that the delay greatly differs between the wiring in the cluster (local wiring) and the wiring outside the cluster (wiring of the routing track). The second point is that if there are unused resources in the cluster, the implementation efficiency decreases (as a result, many logical blocks will be used), so we want to pack as much logics as possible in a cluster. VPack [8], an initial clustering tool developed by the research group of Rose et al. from Toronto University, has the following two optimization goals:

(1) Minimize the number of clusters
(2) Minimize intercluster connection.

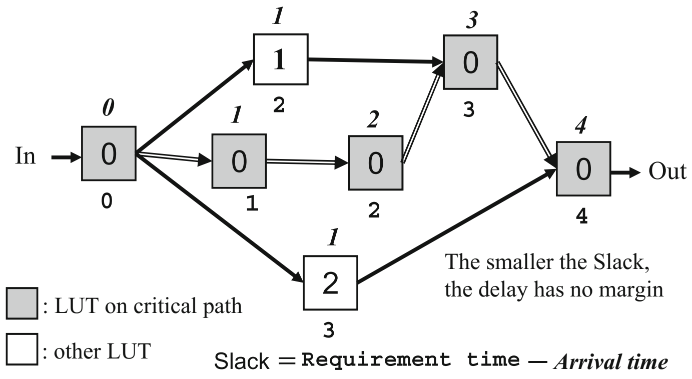
VPack selects the LUT with the largest number of inputs from unclustered LUTs as the seed of a new cluster. Then, it adds the LUTs with the largest number of inputs that can be shared with the current cluster (as depicted in Fig. 5.3).

This method worked well for some target optimizations such as the number of clusters and the number of connections between clusters. However, satisfying performance cannot be achieved with respect to the delay. This tool did not take the delay difference inside and outside the cluster into account, which is the first important point of the clustering. So, it is a clustering tool with large dispersion in delay performance.



Initial cluster
Attraction(L) = | Nets(L) ∩ Nets(C) |
L : LUT C :Initial cluster

Fig. 5.3 Operation of VPack



In
Out
: LUT on critical path
: other LUT
The smaller the Slack, the delay has no margin
Slack = Requirement time — Arrival time

Fig. 5.4 Calculation of connection criticality

Therefore, two years later, Rose et al. proposed T-VPack [9] which addressed this issue. T-VPack is a clustering tool that extends VPack to be a timing-driven system. It selects the LUT with the largest number of inputs on the critical path as the seed of the cluster. To absorb LUTs into the current cluster, not only the number of inputs that can be shared but also both (1) *Connection Criticality* and (2) *Total Path Affected* are taken into consideration. *Connection Criticality* is an index for judging whether it is a route close to the critical path and is calculated based on Slack. Figure 5.4 shows a calculation example of the Slack value. The squares in the figure indicate the LUT, and the numbers in the square represent the Slack values. The arrival time is the number of maximum levels of LUTs from the input, and the request time is the maximum value when tracing backward from the arrival time of the output in the same way. Slack indicates the difference between the request time and the arrival time, and it can be said that the smaller the Slack is, the closer a route is to the critical path, as shown in the figure.

On the other hand, *Total Path Affected* is an index showing how much a certain LUT is involved in critical paths. It is the number of critical paths that can be reduced in the same *Connection Criticality*. It is given by adding paths that become critical paths from inputs. An example is shown in Fig. 5.5. Squares represent LUTs and dotted lines denote critical paths. In addition, the numerical value in the square

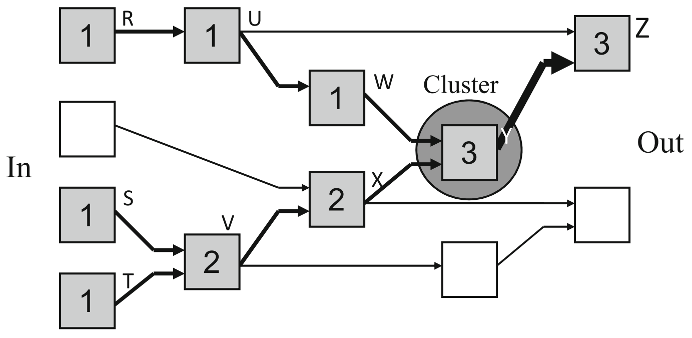

In
Out
Cluster
R
U
W
S
T
V
X
Y
Z
1
2
3
Three path delays on "Connection Criticality" can be reduced by clustering the LUT of Z.

Fig. 5.5 Calculation of total path affected

indicates how many critical paths are involved in this LUT. A signal (thick dotted line) connected from LUT "Y" to LUT "Z" is a common wiring of three critical paths. If this wiring of high Total Path Affected is sped up, 3 critical paths can be improved at the same time. Therefore, in T-VPack, timing-driven clustering is performed so that LUT "Y" and LUT "Z" are in the same cluster.

On the other hand, RPack/t-RPack [10] are other approaches from another viewpoint. They are clustering tools which improve the index of routability metrics in VPack. Routability metrics is a target index to show the flexibility of the circuit wiring at the stage of placement and wiring process of the FPGA circuit design flow. The goal is to eliminate wiring congestion. Improvement in this routability metrics not only eliminates congestion among clusters but also has the effect of minimizing the total number of wires outside the cluster. Furthermore, in iRAC [11], the routability metrics are extended to optimize wires outside of the cluster even when "vacancy" is created in the cluster, as shown in Fig. 5.6. That is, the optimization consider both the inside and outside of the cluster. The authors' research [12] is also effective similar to RPack and iRAC by optimizing the routability metrics and routing resources inside and outside the cluster.

In the aforementioned works, the performance of clustering tools has been greatly improved. However, the clustering algorithms introduced above only consider LUTs of the same type. Recent logical blocks employ adaptive LUTs, as described in the previous section, and constitute more complex logical clusters. Introducing adaptive LUTs not only needs to map logics to an appropriate number of input LUTs at the time of technology mapping, but also greatly influences clustering. When clustering a netlist composed of adaptive LUTs, besides the number of LUTs that can be packed into a logical cluster, we also have to consider delay, routability, total number of primary inputs and the combination of LUTs with different number of inputs allowed by a logical cluster, etc. Therefore, more optimizations are required. However, it

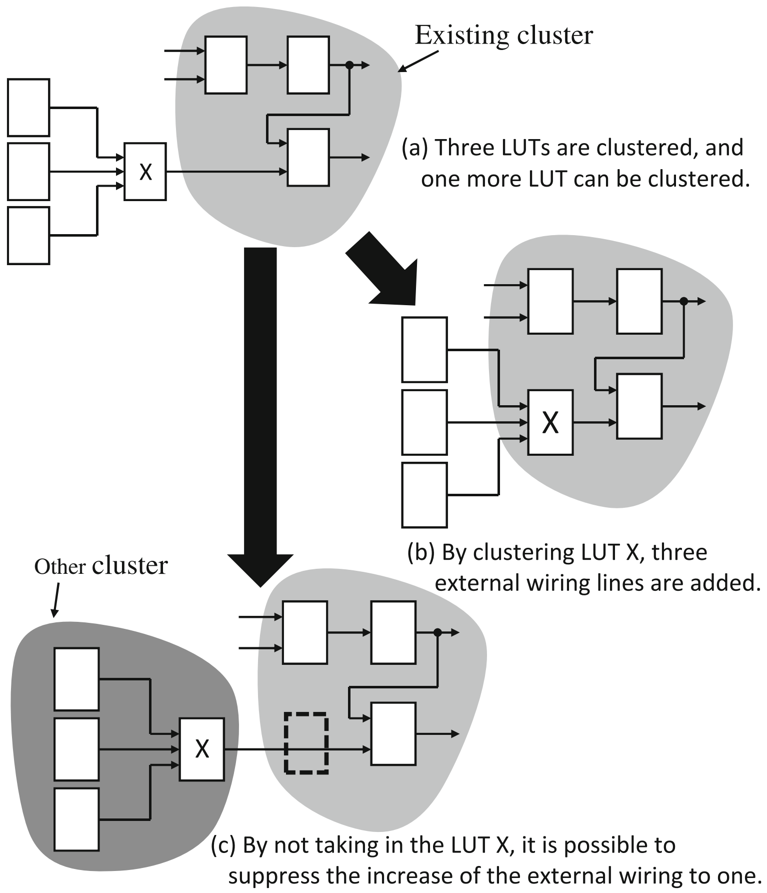

Existing cluster
X
(a) Three LUTs are clustered, and one more LUT can be clustered.
X
(b) By clustering LUT X, three external wiring lines are added.
Other cluster
X
(c) By not taking in the LUT X, it is possible to suppress the increase of the external wiring to one.

Fig. 5.6 Clustering considering routability metrics

is extremely difficult to find a solution that simultaneously satisfies the minimum number of logical clusters, minimum delay and minimum number of wires.

To solve the above challenges, AAPack (Architecture-Aware Packer) [13], incorporated in VTR (Verilog-to-Routing) 6.0 [14], is developed. In the VTR project, the device architecture is modeled using XML.[2] The architecture definition of the device is divided into (a) a cell structure (Physical Block: corresponding to a logic cell or the like in a cluster) and (b) a routing structure (Interconnect: corresponding to a connection relation and connection method between Physical Blocks). The physical block in (a) can describe a nested structure, thereby representing a clustered logical block. Furthermore, a structure having various modes can also be expressed.

[2]Their place and route tool uses XML from VPR 5.0, but VTR 6.0 extends it so that it can more describe complex structures with a simple notation. The VPR place and route tool is described in the next section.

For example, as in Altera's Fracturable LUT-based logic block described in the previous section, an LUT with a large number of inputs can be divided into multiple LUTs with a small number of inputs (multiple modes).

AAPack is a clustering tool corresponding to the above device architecture model. Clustering is performed in the following procedure:

(1) If there is an unclustered LUT, an LUT serving as a seed is selected and a cluster to be inserted is determined,
(2) Insert the LUT into the cluster according to the following procedure:
   (a) Search candidate LUTs to be inserted in the cluster,
   (b) Insert selected LUTs into the cluster,
   (c) If it is still possible to insert LUTs into the cluster, return to (2a),
(3) Add the cluster to the output file and return to (1).

(1) is the same as VPack and T-VPack. (2)-(a) determines the LUT to be loaded into the cluster from "the number of inputs that can be shared in the cluster" and "the relationship between the LUT outside the cluster". (2)-(b) judges whether the selected LUT from the cluster structure information can be inserted into the cluster. As shown in Fig. 5.7, this judgement is made by searching the graph of the cluster structure. This graph is made so that the granularity of the LUT is "fine" to "coarse" from right to left. The search is performed in depth-first order from the right side of the graph (the side with smaller grain size). When there are LUTs with multiple granularities in the cluster, logic is implemented in the smallest LUT that satisfies the number of inputs of the circuit to be mapped. For example, when mapping a 4-input and 1-output combinational circuit to the LUT, shown in Fig. 5.7, it can be implemented with either a 5-LUT or a 6-LUT. When mapped to 5-LUT, another 5-LUT circuit can be combined in the same cluster. However, if it is mapped to a 6-LUT, no more circuits can be implemented in this cluster. Therefore, it is more efficient to be allocated from the smaller LUT.

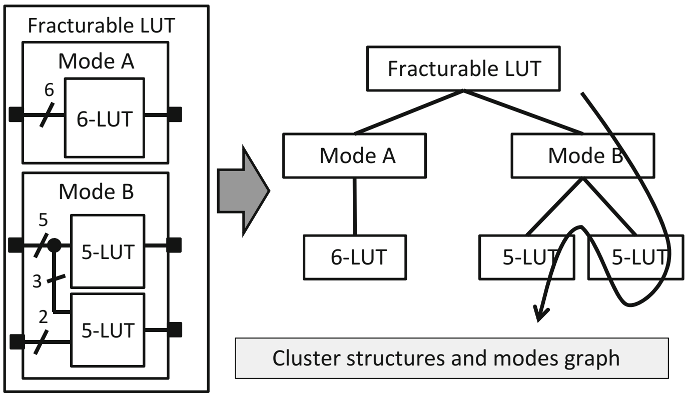


Fig. 5.7 Clustering of AAPack

When an LUT that can be inserted into a cluster is found, it can be decided whether wiring can be performed next. Then, when the insertion of the LUT is successful, an LUT that can be inserted as a child having the same parent is searched from the netlist and inserted. This process is repeatedly executed.

Whether wiring is possible or not is judged as follows. First, we create a graph with the input/output pins in the cluster as nodes and the connection relation of each pin as a directed edge. Next whether it can be wired to the LUT to be inserted by the same processing as PathFinder (VPR 5.0 routing algorithm) is checked.

As described above, AAPack provides a method to cluster LUTs for complex structured logical clusters.

5.4 Place and Route

The last step, placement and routing, is the task of establishing the physical positions of the logical blocks and the signal paths connecting them. Generally, logical blocks are allocated first, then wiring between them is performed.

Since many FPGAs have a two-dimensional array of logical blocks, the placement process can be formulated as a slot placement problem or quadratic assignment problem (QAP). However, these problems are known as NP-hard,[3] and usually uses an approximation algorithm such as Simulated Annealing[4] (SA).

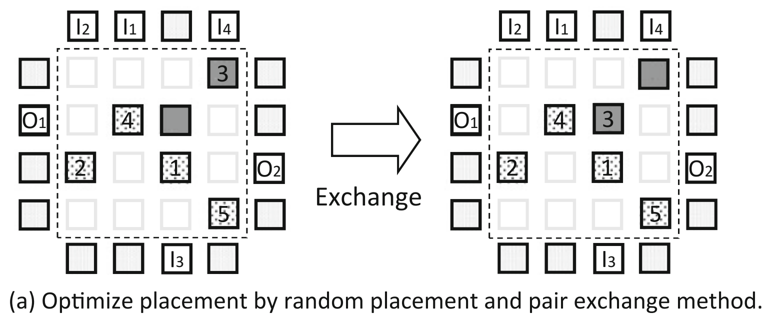
On the other hand, two methods are used for the routing process: the Global Routing and the Detailed Routing. Global Routing determines the rough wiring route of each net. That is, which channels are used to establish the connection. Detailed Routing determines which routing resources and switches each net connects to based on the information obtained by Global Routing.

This section introduces VPR (Versatile Place and Route) [14–18], which is the most widely used place and route tool in academia. The VPR 4.3 deployment process is performed with the following procedure (represented in Fig. 5.8):

(1) Randomly allocate logical blocks and I/O blocks.
(2) Calculate the cost of congestion when wiring is done in this state.
(3) From this state, randomly select two blocks and swap them (pair exchange method).
(4) Recalculate the cost for the replaced state.
(5) Compare with the cost before replacement and judge whether to accept the change or not.

[3]A NP-hard problem is at least equal to or more difficult than the problem belonging to the Non-Deterministic Polynomial Time (NP) class in computational complexity theory. Quadratic Assignment Problem (QAP) is said to be one of particularly difficult problems among NP-hard combinatorial optimization problems.

[4]The Simulated Annealing method is a general-purpose stochastic meta-heuristic algorithm. The feature of SA is to accelerate the convergence by decreasing its acceptance probability due to temperature change when trying to escape local solution using randomness.


I2
I1
I4
3
O1
4
2
1
O2
5
I3
Exchange
(a) Optimize placement by random placement and pair exchange method.

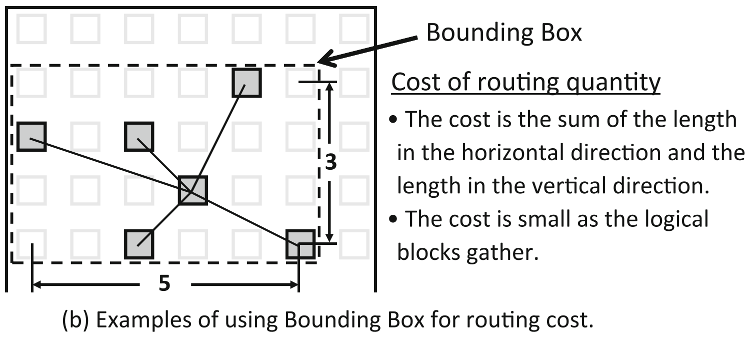

Bounding Box
Cost of routing quantity
• The cost is the sum of the length in the horizontal direction and the length in the vertical direction.
• The cost is small as the logical blocks gather.
3
5
(b) Examples of using Bounding Box for routing cost.

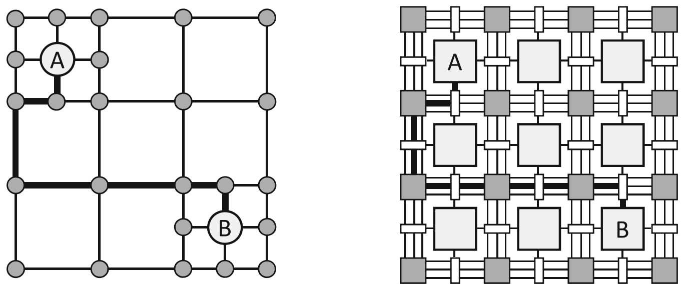
Fig. 5.8 Placement procedure of VPR

As shown in Fig. 5.8a, the routing process uses SA, and the exchange is accepted with a certain probability even if the wiring cost and the timing cost are improved or worsened from the previous state. The cost of the routing amount is a cost representing the amount of routing resources in the case of wiring. Fig. 5.8b depicts a diagram showing only logical blocks, and the dotted frame shows the Bounding Box (boundary rectangle). The Bounding Box cost is the sum of the horizontal and vertical lengths of the range created by a net placed on the FPGA device, and the closer the blocks are placed, the smaller the Bounding Box cost is. The timing cost determines the cost from the net source-sink delay and the Slack value of the path.

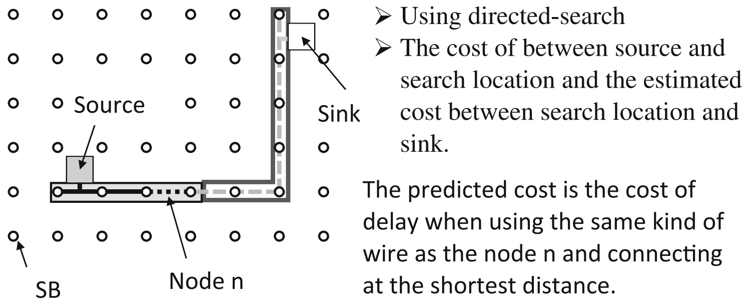
In this way, in the placement process, logic blocks having deeper relationships are placed closer to each other to shorten the wiring length, and the entire layout is arranged to be as compact as possible, too. Furthermore, ideally, the wiring congestion degree of each channel should be equal.

Although the initial VPR placement algorithm (VPlace) was only optimized by the Bounding Box cost, the above timing cost was added to the placement process in T-VPlace [19] since VPR 4.3.

As previously mentioned, the routing process is divided into Global Routing and Detailed Routing, as shown in Fig. 5.9. Global Routing is not limited to only FPGAs, and it results in general route search problem for the graph. On the other hand, Timing-Driven Router [20], which is the Detailed Routing part of VPR 4.3, is an algorithm


A
B
A
B
Global routing is to solve the route search problem of the graph
Detailed routing is determined by the following method

(a) Global routing and detail routing


Source
Sink
SB
Node n
➢ Using directed-search
➢ The cost of between source and search location and the estimated cost between search location and sink.
The predicted cost is the cost of delay when using the same kind of wire as the node n and connecting at the shortest distance.

(b) Method of searching route between source and sink of netlist

Fig. 5.9 Routing procedure of VPR

based on Path finder [21]. It uses directed-search for net route search between source and sinks. For this reason, costs between the source and search positions, as well as estimated cost between the search position and the sink, are required. Fig. 5.9b shows the route searches between a source and a sink. Black wires from the source are already fixed to their route. The black dotted line shows the node *n* currently being searched, and the gray broken line shows the shortest path between the searched position and the sink using the same type of wiring as the node *n*. The bright gray frame, surrounding the black and broken lines, represents the cost of congestion and timing between source and search positions. On the other hand, thick gray frame surrounding the gray broken line represents the estimated cost of the timing between the search position and the sink.

In this fashion, the cost is calculated. Wiring is performed for each net using the following procedure:

(1) Routing to each net with minimum cost.
(2) Because wires that compete for multiple nets tend to be crowded, congestion costs are added to the evaluation values.

(3) If a net cannot be routed because of conflict, searching for the least cost path for each net is executed again.
(4) Since the wire cost has been updated since last search, the route is possibly changed.
(5) Conflict decreases from last search due to route change.
(6) If conflicts still exist, similarly add cost and search wiring route again.
(7) Execute this operation until there is no conflict.

The placement and routing process is time-consuming because it involves many optimization problems. In order to improve the practicality, speeding up these processes is essential. In addition, it is necessary to cope with an architecture which makes it more functional, such as cluster structure of complicated logical blocks, dedicated circuit, dedicated wiring and so on. In order to cope with such requirements, the VPR development team introduced device definitions based on XML in the VPR 6.0 version [14], and further started the VTR (Verilog-to-Routing) project [13] in 2012. Lately, VTR 7.0 (Current Version: 7.0—Full Release—Last updated: April 22, 2014) has been released as an open source FPGA development framework [22, 23].

5.5 Low Power Design Tools

Up to now, we have looked at the basic operations of the FPGA design tool, but the recent major trends require low power consumption. There are various problems with FPGAs, but among them, large power consumption is an obstacle when considering implementations on SoCs. Therefore, methods for reducing the power consumption from technology mapping to clustering, and placement and routing have been studied, for example the technology mapping tool EMap of Wilton et al. of UBC, the clustering tool P-T-VPack, and the P-VPR place and route tool [5]. In this section, we take these as examples and introduce the low power consumption methods and their effects.

First, we briefly explain the FPGA dynamic power consumption. Equation (5.1) is the general equation for calculating dynamic power consumption in LSIs. V is the power supply voltage, f_{clk} is the clock frequency, $Activity(i)$ is the switching activity[5] of the node i, C_i is the load capacitance of the node i. $Activity(i)$ in Eq. (5.1) indicates the transition probability of node i in f_{clk} and takes a value of 0.0–1.0. This value indicates how much the target node switches on average. Charge and discharge to this capacity cost energy consumption, and its time/space integration effects the power consumption. That is, in order to reduce the power consumption, it is necessary to lower the power supply voltage, the operating frequency, the load capacity of the circuit, or the activity. Reducing the power supply voltage is the most effective method, but this has an impact on the manufacturing process and the

[5]The switching activity has almost the same meaning as the toggle rate (TR). The toggle rate is the number of transitions from the logic value 0 to the logic value 1 of the target node and the transition from the logic value 1 to the logic value 0 per unit time.

peripheral circuits. On the other hand, dropping the frequency is often not the most efficient option because it directly results in performance degradation. Therefore, in reducing the power consumption using design tools, we focus on reducing the load capacity and activity.

$$Power_{dynamic} = 0.5 \cdot V^2 \cdot f_{clk} \cdot \sum_{i \in nodes} Activity(i) \cdot C_i \qquad (5.1)$$

In addition, since FPGAs use SRAMs, the static power consumption, such as leakage current, is also large; however, current low power consumption design tools (EMap, P-T-VPack, P-VPR) only consider the dynamic power consumption.

5.5.1 Emap: Low Power Consumption Mapping Tool

The research presented by Wilton et al. is consistently aiming to reduce $Activity(i)$ in Eq. (5.1). In technology mapping, it is possible to optimize the power by taking the high activity wiring to the inside of the LUT in the post-mapping netlist. Figure 5.10 shows an example of mapping process considering the switching activity in Emap.

The circuits in Fig. 5.10a, b map the same netlist to three 3-input LUTs. The number next to each wiring is the switching activity value. In Fig. 5.10a, by moving the wires with high switching activity to the inside of the LUT, the average value of the activity is minimized. On the other hand, in Fig. 5.10b, the number of stages of LUT corresponding to the delay time is the same. However, since the wiring with high switching activity value is outside the LUT, even though the delay performance product is the same, the power consumption consequently increases.

Another way to reduce power consumption is to minimize the duplication of nodes. In technology mapping, nodes are duplicated for delay optimization or the like. However, the number of nodes increases due to the replication, and the power consumption also increases because the number of branches of wirings further increases. Therefore, in Emap, duplication of critical paths is allowed; but, in other wiring, the duplication is suppressed. Also, the duplication of a node occurs when nodes with multiple fanouts are inside the cone, so Emap adopts measures such as making nodes with multiple fanouts as root nodes. As a result, any increase of nodes is suppressed when compared to conventional FlowMap.

Figure 5.11 shows an example of node duplication. In Fig. 5.11a, cutting is selected so that node 3 becomes the vertex (root node) of the cone. However, in Fig. 5.11b it is in the middle. At the time of the cut selection, both Fig. 5.11a, b are the same for both the number of cuts and the number of LUT stages. But in Fig. 5.11b, since the fanout of node 3 is 2, the duplication of the node results in the generation of an LUT. That is, the LUT increases by one. Emap suppresses the duplication of such nodes.

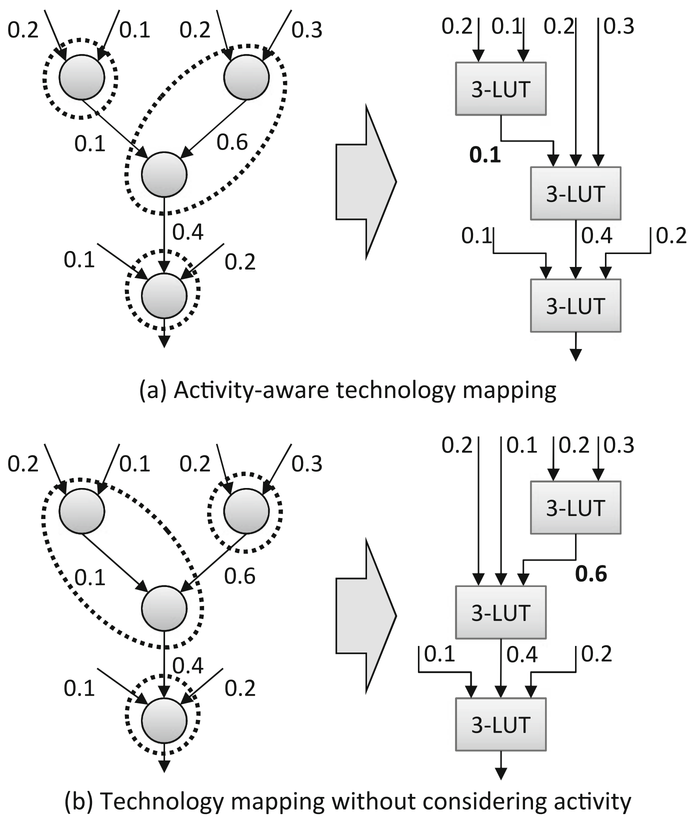

0.2
0.1
0.2
0.3
0.1
0.6
0.4
0.1
0.2
3-LUT
(a) Activity-aware technology mapping
0.6
(b) Technology mapping without considering activity

Fig. 5.10 Mapping procedure of Emap

5.5.2 P-T-VPack: Low Power Consumption Clustering Tool

P-T-VPack, a low power consumption clustering tool, basically uses the same method as Emap to reduce the activity. Considering the load capacity of the wiring existing inside and outside the BLE, the wiring outside the BLE has a large wiring length. In addition, since many transistors such as a connection block and a switching block are also connected, the capacity is generally large. Therefore, it is more advantageous for power consumption to include wirings with higher activity as much as possible within the cluster (within BLE). Figure 5.12 shows an example of P-T-VPack clustering.

In this example, when the combination of clustering in Fig. 5.12a and the clustering in Fig. 5.12b are compared, the activity of the wiring connecting between clusters is lower (0.1) in (a) than in (b) (0.4). Therefore, when the load capacities of the wirings between the clusters are the same, the power consumption of (a) is lower.

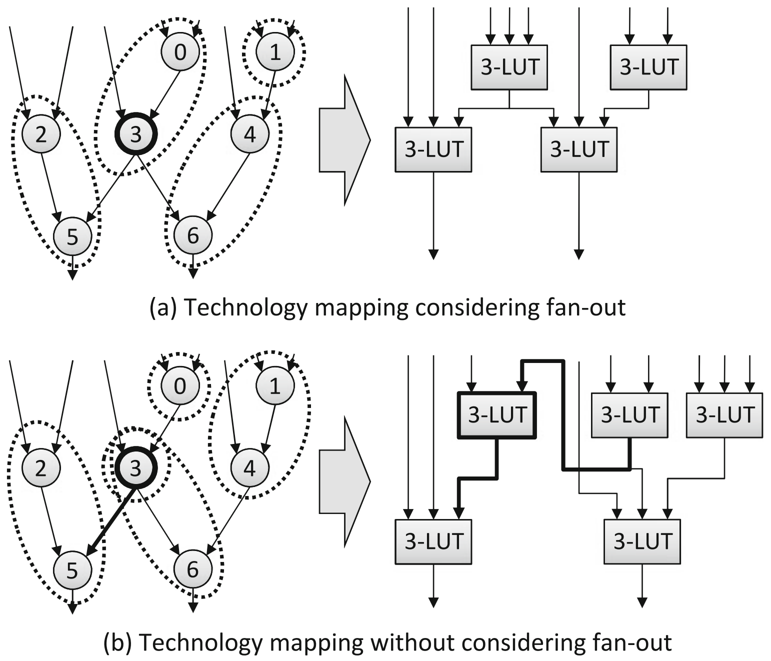

0
1
2
3
4
5
6
3-LUT
(a) Technology mapping considering fan-out
(b) Technology mapping without considering fan-out

Fig. 5.11 Duplicate nodes in the mapping process

5.5.3 P-VPR: Low Power Placement and Routing Tool

The basic idea of P-VPR, a low power consumption placement and routing tool, is also similar to P-T-VPack and Emap. However, since wiring with high activity cannot be hidden in placement and routing, connection for such nets should be as short as possible. When priority is given to the switching activity, there are cases where a signal line, whose switching activity is low but its timing is critical, may eventually be detoured and the delay may increase. Such a trade-off is adjusted with the weight parameter in the cost function to find the balance that minimizes the energy.

5.5.4 ACE: Activity Measurement Tool

As in previous mentioned tools, it is possible to reduce the power at the design tool level by considering the activity. However, what is important here is how to accurately obtain the activity of each node. Wilton et al. proposed a tool called ACE (Activity Estimator) for activity measurement.

There are two ways to roughly calculate the activities: The first one is a dynamic method based on simulation, and the second is a static method stochastically obtained.

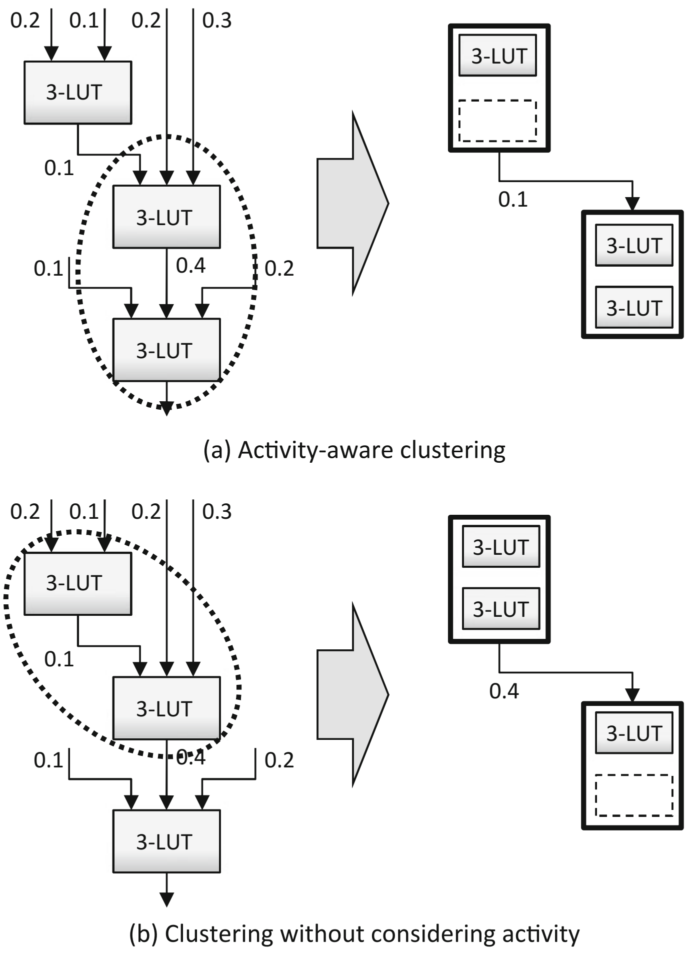

0.2
0.1
0.2
0.3
3-LUT
0.1
0.4
0.2
(a) Activity-aware clustering
0.4
(b) Clustering without considering activity

Fig. 5.12 Mapping procedure of P-T-VPack

Generally dynamic methods have high precision; but, they are time consuming. Furthermore, the accuracy is easily affected by the test pattern. On the other hand, the static method based on probability is low in accuracy, and the execution speed is fast. Also, since the probabilistic method reacts sensitively to setting the transition probability of the input signal, the initial value is an important factor determining the accuracy.

ACE is a tool to statically analyze netlists. The netlist can calculate activities from any of the netlists at gate level when used for technology mapping, netlist at the LUT level for clustering, and BLE level netlist for placement and routing. There are two types of values that this tool calculates: static property (SP: rate of signal "High") and switching activity (SA: probability of signal transition).

5.6 Summary

In this chapter, we have briefly summarized how the FPGA design tools are made and how the research was carried out in the background. All of them aimed at optimizing delay, implemented area, and power consumption. Even if the optimization is performed independently in all processes, it is not necessarily the optimum in the final circuit. That is, cooperation between processes is necessary. For example, it is possible to improve the performance by simultaneously optimizing the technology mapping and clustering, and by also processing the clustering and placing and routing in multiple processes simultaneously. Future FPGA design tools are expected to progress with the aim of simultaneous optimization of such multiple processes.

References

1. J. Cong, Y. Ding, FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. IEEE Trans. CAD **13**(1), 1–12 (1994)
2. J. Cong, Y. Hwang, Simultaneous depth and area minimization in LUT-based FPGA mapping, in *Proceedings of FPGA'95* (1995), pp. 68–74
3. J. Cong, J. Peck, Y. Ding, RASP: a general logic synthesis system for SRAM-based FPGAs, in *Proceedings of FPGA'96* (1996), pp. 137–143
4. D. Chen, J. Cong, DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs, in *Proceedings of ICCAD2004* (2004), pp. 752–759
5. J. Lamoureux, S.J.E. Wilton, On the interaction between power-aware computer-aided design algorithms for field-programmable gate arrays. J. Low Power Electron. (JOLPE) **1**(2), 119–132 (2005)
6. V. Manohararajah, S.D. Brown, Z.G. Vranesic, Heuristics for area minimization in LUT-based FPGA technology mapping. IEEE Trans. CAD **25**(11), 2331–2340 (2006)
7. J. Cong, S. Xu, Delay-oriented technology mapping for heterogeneous FPGAs with bounded resources, in *Proceedings of ICCAD'98* (1998), pp. 40–45
8. V. Betz, J. Rose, Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size, in *Proceedings of CICC'97* (1997), pp. 551–554
9. A. Marquardt, V. Betz J. Rose, Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density, in *Proceedings of FPGA'99* (1999), pp. 37–46
10. E. Bozorgzadeh, S.O. Memik, X. Yang, M. Sarrafzadeh, Routability-driven packing: metrics and algorithms for cluster-based FPGAs. J. Circuits Syst. Comput. **13**(1), 77–100 (2004)
11. A. Singh, G. Parthasarathy, M. Marek-Sadowska, Efficient circuit clustering for area and power reduction in FPGAs. ACM Trans. Des. Autom. Electron. Syst. (TODAES), **7**(4), 643–663 (2002)
12. M. Kobata, M. Iida, T. Sueyoshi, Clustering technique to reduce chip area and delay for FPGA (in Japanese). IEICE Trans. Inf. Syst. (Japanese Edition), **J89-D**(6), 1153–1162 (2006)
13. J. Rose, J. Luu, C.W. Yu, O. Densmore, J. Goeders, A. Somerville, K.B. Kent, P. Jamieson, J. Anderson, The VTR project: architecture and CAD for FPGAs from verilog to routing, in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'12)* (2012), pp. 77–86, https://doi.org/10.1145/2145694.2145708
14. J. Luu, J.H. Anderson, J. Rose, Architecture description and packing for logic blocks with hierarchy, modes and complex interconnect, in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'11)* (2011), pp. 227–236, https://doi.org/10.1145/1950413.1950457

15. V. Betz, J. Rose, VPR: a new packing, placement and routing tool for FPGA research, in *Proceedings of FPL'97* (1997), pp. 213–222
16. V. Betz, J. Rose, A. Marquardt, Architecture and CAD for deep-submicron FPGAS. The Springer International (1999)
17. V. Betz, VPR and T-VPack userfs manual (Version 4.30). University of Toronto (2000)
18. J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W.M. Fang, K. Kent, J. Rose, VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling. ACM Trans. Reconfig. Technol. Syst. **4**(4), Article 32, 23 (2011), https://doi.org/10.1145/2068716.2068718
19. A. Marquardt, V. Betz, J. Rose, Timing-driven placement for FPGAs, in *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays (FPGA'00)* (2000), pp. 203–213, https://doi.org/10.1145/329166.329208
20. J.S. Swartz, V. Betz, J. Rose, A fast routability-driven router for FPGAs, in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field programmable Gate Arrays (FPGA'98)* (1998), pp. 140–149, https://doi.org/10.1145/275107.275134
21. L. McMurchie, C. Ebeling, PathFinder: a negotiation-based performance-driven router for FPGAs, in *Proceedings of FPGA'95* (1995), pp. 111–117
22. J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K.B. Kent, J. Anderson, J. Rose, V. Betz, VTR 7.0: next generation architecture and CAD system for FPGAs. ACM Trans. Reconfig. Technol. Syst. **7**(2), Article 6, 30 (2014), https://doi.org/10.1145/2617593
23. Verilog to Routing—Open Source CAD Flow for FPGA Research, GitHub, https://github.com/verilog-to-routing/vtr-verilog-to-routing

Chapter 6
Hardware Algorithms

Kentaro Sano and Hiroki Nakahara

Abstract Just implementing with hardware is almost nothing to contribute to achieve high performance. The performance of FPGA computing is depends on how to use efficient hardware algorithms for the target application. This chapter introduces various types of hardware algorithms useful for FPGA implementation. First, pipelining is the most popularly used technique. Recently, it is often automatically formed with HLS design tool. Then, general parallel processing techniques are introduced along Flynn's classic taxonomy. Systolic algorithms and data-flow models are also classic methods researched in 1970s' and 1980s', but they have been practically used after large-scale FPGAs are available for computation. Then, stream processing, simple but powerful framework, is introduced with a practical example. Next, cellular automaton, hardware sorting and pattern matching which are important in network processing a killer application of FPGAs are introduced.

Keywords Pipeline processing · SIMD processing · Systolic algorithm
Data-flow machines · Streaming processing · Cellular automaton · Hardware sorting · Pattern matching

A hardware algorithm is a procedure suitable for hardware implementation and the target hardware model. This chapter presents an outline of several hardware algorithms used for processing implementation in hardware, with specific emphasis on parallelism, control, and data-flow of processing.

K. Sano (✉)
RIKEN, Kobe, Japan
e-mail: kentaro.sano@riken.jp

H. Nakahara
Tokyo Institute of Technology, Tokyo, Japan
e-mail: nakahara@ict.e.titech.ac.jp

H. Amano (ed.), *Principles and Structures of FPGAs*,
https://doi.org/10.1007/978-981-13-0824-6_6

6.1 Pipelining

6.1.1 Principle of Pipelining

Pipelining is a technique for speeding up many processing iterations done continuously. The flow production at a production plant is a typical example. Figure 6.1 presents the pipelining concept. Figure 6.1a depicts the non-pipelining case where processing iteration 2 is done sequentially after the completion of iteration 1. With the pipelining shown in Fig. 6.1b, we divide a processing iteration into n stages of uniform proportion. The processing iterations start without waiting for the completion of all stages of a preceding processing iteration. Figure 6.1b portrays an example of five-stage pipelining in which each processing iteration has five stages: n = 5. In the non-pipelining case, each processing iteration is completed within a time length L. In pipelining, a processing iteration is started in each time length of L/n after the processing of iteration 1 has finished. Therefore, throughput speedup is achieved by five times at most, where five is the number of processing iterations per unit time. In the five-stage example shown in Fig. 6.1b, six processing iterations are done in two time lengths of the non-pipelining case. Parallel processing with different parts of the entire sequential process is the principle of pipelining, which is understood as five stages processed in parallel when stage 5 of processing iteration 1 is started [1, 2].

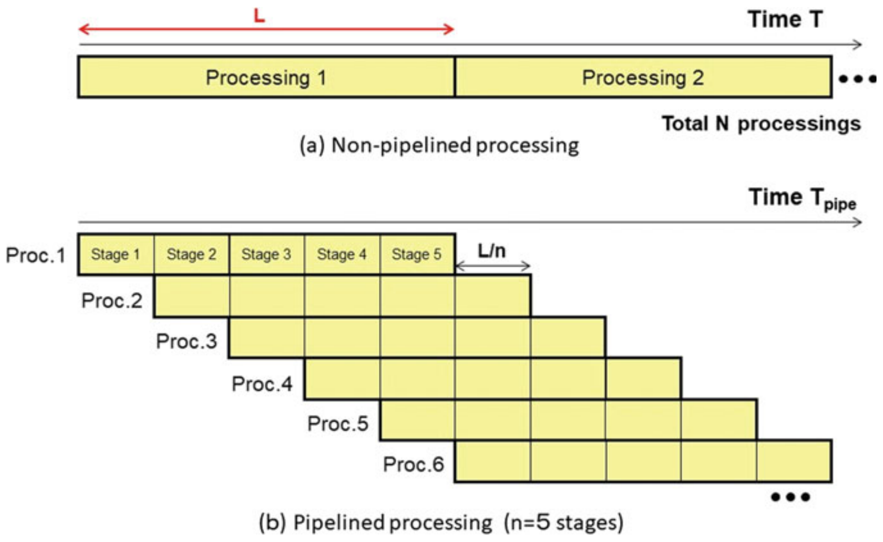


Fig. 6.1 Overview of pipelining

6.1.2 Performance Improvement by Pipelining

Actually, n stage pipelining does not necessarily mean five times speedup. Here, we develop the expression of the speedup factor, a measurement of performance improvement using pipelining, for a time length of one processing iteration L, N processing iterations to be done, and n stages in a single processing iteration.

Figure 6.1a shows that N processing iterations by a non-pipelining operation require $T(N) = LN$ time length for completion. We can derive the time length $T_{pipe}(N)$ for completion of N processing iterations for pipelining with n stages, as described below. First, the completion of processing iteration 1 requires L time length. However, the next processing iteration finishes L/n later after the completion of processing iteration 1. Consequently, $N - 1$ processing iterations, except for processing iteration 1, complete at intervals of L/n. These N operations end after $T_{pipe}(N) = L + (N - 1)L/n = (n + N - 1)L/n$ time length.

The speedup factor by pipelining described above, $S_{pipe}(N)$, is the ratio of $T(N)$ to $T_{pipe}(N)$, and can be calculated as:

$$S_{pipe}(N) = \frac{T(N)}{T_{pipe}(N)} = \frac{nN}{n + N - 1} = \frac{n}{1 + \frac{n-1}{N}}$$

If $n \ll N$, then $S_{pipe}(N) \cong n$ and the speedup factor over non-pipelining is n: the number of stages. However, this just means that n times higher throughput can be achieved while the latency of each processing iteration is not shortened. Although n times improvement of performance means the reduction of the whole time length for N processing iterations, it does not bring about the shortening of the time length for each processing iteration. If this is compared to the flow production in the automotive industry, then the car production per day (throughput) might increase by dividing a processing iteration into segments. However, the time between the car order and the delivery of the parts of that car to factories and the completion of production of that car (latency) does not change.

If the number of stages, n, is not much greater than that of processing iterations N, then the speedup by pipelining remains much smaller than n. For pipelining with $n = 6$ and $N = 5$, for example, $S_{pipe}(5)$ is $6/(1 + 1) = 3$ and the time of processing iteration is reduced to just one-third. Considering that the maximum speedup by pipelining with n stages is n, the pipelining efficiency, which is the percentage of the actually achieved speedup to the maximum, is given as:

$$E_{pipe}(n, N) = \frac{S_{pipe}(N)}{n} = \frac{1}{1 + \frac{n-1}{N}} = \frac{N}{N + n - 1}$$

For the example presented above, $E_{pipe}(6, 5)$ is $\frac{5}{5+6-1} = 0.5$. This means that the speeding up is just 50% at maximum. A long time length of insufficient parallelism has led to this small decrease in efficiency. In Fig. 6.1b, only one process is executed at the first stage of iteration 1. When the fifth processing stage of iteration 1 starts, the number of parallel processing iterations never attains 5, the maximum degree of parallelism. Consequently, at the beginning of pipeline processing, there exists a prologue period where perfect parallelism is not achieved. Quite similarly, at the end of pipeline processing, there exists an epilogue period during which the degree of parallelism decreases step by step. Perfect elimination of the prologue and epilogue periods is impossible. However, if the number of processing iterations, N, is sufficiently large, then the lengths of the prologue and epilogue periods are diminished in comparison to the total processing time. Consequently, the speedup factor approaches the maximum value of n. In contrast, if N is small, then the effects of prologue and epilogue periods are greater in a relative sense. Therefore, the speedup factor is smaller.

For other several reasons not mentioned above, hardware pipelining can intrinsically block the performance improvement. Thus, a lot of attention must be devoted during the pipelining design. The hardware configuration for pipelining is depicted in Fig. 6.2. Figure 6.2a presents the hardware configuration for a non-pipelining-based system, where the total processing is implemented by one combinational logic circuit. When the register of the preceding processing iteration is updated at the rising edge of the clock signal, a new data is outputted to the combinational logic circuit after the propagation delay of the register. Inputted data propagates through the combinational logic circuit. Then, the processed data arrive at the register of the next processing iteration after a certain delay in the critical path. Here, the critical path is that which provides the maximum delay in the circuit. After the period of setup time, which is necessary to secure correct latching of the arrived data by flip-flop, the processed data are written in the register by inputting a clock signal. Thereby, the processing iteration terminates. Consequently, one cycle time, which is equal to the time interval between two successive clock signal inputs, must be longer than the sum of the propagation delay, the delay in the critical path of the combinational logic circuit, and the setup time. This sum represents the upper bound of the maximum operation frequency.

Pipelining a circuit boosts the throughput by increasing the maximum operation frequency. Figure 6.2b depicts an example of a circuit with four pipeline stages: $n = 4$. A circuit can be divided into stages by inserting pipeline registers. With them, the combinational logic circuits on which the data must propagate are shortened. Even the critical paths of combinational logic circuit could be perfectly divided into uniform circuits, the cycle time might not be shortened to $1/n$ because of the existence of clock skew, which is the misalignment of clock signals supplied to each register, and because of the propagation delay and setup time in pipeline registers. Furthermore, dividing a combinational logic circuit into n uniform stages is sometimes impossible. Such a case is illustrated by the third pipeline stage in Fig. 6.2b. In this situation, one stage usually has a longer delay time than others and becomes the critical path, which happens quite often. Consequently, $n = 4$ might not increase the operation frequency by four times because the maximum delay is longer than one-fourth.

These difficulties generally become more deleterious with an increasing number of stages. Although an operation frequency improvement can be obtained by increasing the number of stages of shallow pipelining with a few stages, fine divisions such as

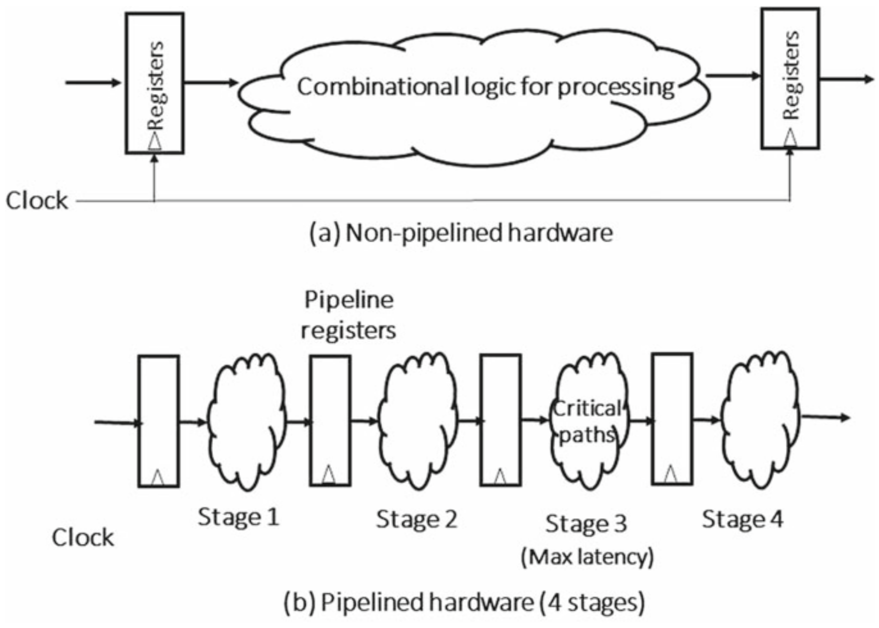


Fig. 6.2 Hardware structure for pipelining

dozens or hundreds might halt the operation frequency augmentation and might even degrade the performance due to clock skew and other factors. However, adding few pipeline stages instead of finely dividing the already existing ones is different. Actually, the increase in the entire circuit's latency by pipelining when compared to non-pipelining-based systems should be carefully investigated. Such delay can be caused by overhead related to pipeline registers or non-uniform staging, as shown in Fig. 6.2b.

6.2 Parallel Processing and Flynn's Taxonomy

6.2.1 Flynn's Taxonomy

To design high-performance hardware, we should consider processing parallelism. The taxonomy of architectures which Flynn proposed in 1965 is useful for considering parallelism for hardware [2, 3]. Hereafter, we refer to this as Flynn's taxonomy. Flynn's taxonomy classifies general-purpose computer architectures in terms of the concurrency degree in an instruction stream for control and a data stream to be processed (Fig. 6.3). It includes Single Instruction stream Single Data stream (SISD), Single Instruction stream Multiple Data stream (SIMD), Multiple Instruction stream

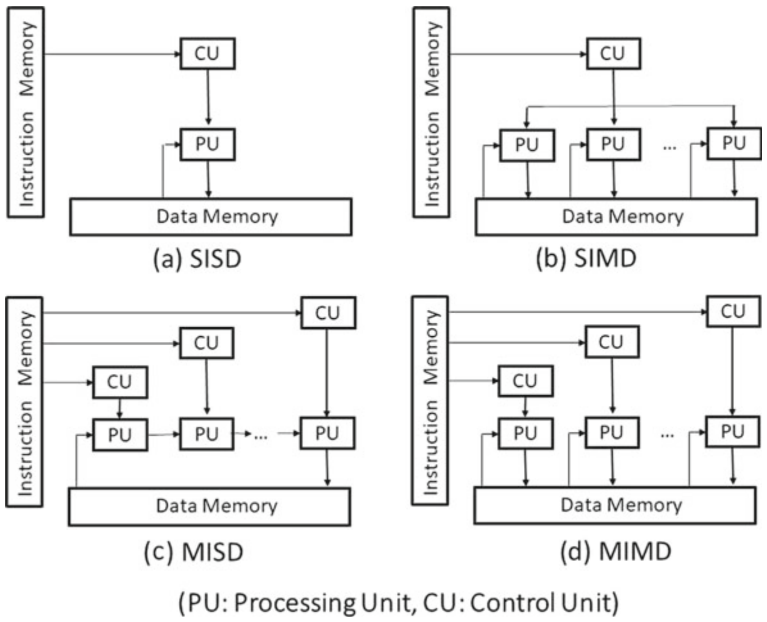


Fig. 6.3 Flynn's taxonomy

Single Data stream (MISD), and the Multiple Instruction stream Multiple Data stream (MIMD). Although this taxonomy is oriented to architectures of general-purpose processors that execute a sequence of instructions intrinsically, it is useful for classifying more general architectures of parallel processing hardware if we recognize the instruction stream as control in a broader sense.

In Flynn's taxonomy, computer architecture comprises the four components of the processing unit (PU), control unit (CU), data memory, and instruction memory. In SISD shown in Fig. 6.3a, one CU controls one PU based on the instruction stream read from the instruction memory. PU executes processing for a single data stream read from the data memory based on the controls directed by CU. Consequently, SISD does not perform parallel processing and represents the architecture of general and sequential processor without parallel processing. In the following, the other three items in the taxonomy are explained.

6.2.2 *SIMD Architecture*

In SIMD in Fig. 6.3b, a single CU reads out an instruction stream and controls multiple PUs simultaneously. Each PU executes common processing based on common controls but for different instruction streams. Consequently, SIMD is an architecture making use of data parallelism. The data memory can be accessed by all the PUs as local memories of the PUs or a single-shared memory common to all the memory. Because the SIMD architecture is suitable to process numerous data synchronously with a single sequence of instructions, it is used as the designated processor for image processing.

In addition, a microprocessor can incorporate SIMD instructions to provide itself with the function of parallel data processing. For example, Intel Corp., aiming at speeding up 3D graphics, designed a microprocessor, the Pentium MMX, with SIMD-type extended instructions. It was commercially produced in 1997 [4, 5]. The MMX Pentium can perform four 16-bit integer operations simultaneously based on one SIMD instruction. Furthermore, AMD Corp. introduced a new product of K6-2 processors equipped with 3DNow! Technology, which provides SIMD-type extended instructions for floating-point operations in 1998. Later, Intel processors were augmented with Streaming SIMD Extensions (SSE), an SIMD-type instruction set for floating-point operation. Then, Pentium III included SSE extended instructions and Pentium IV received extended instructions SSE2 and SSE3. Currently available microprocessors have been augmented with instructions of 128-bit integers and double precision floating-point operations in addition to other operations for compression of video images, as outlined above. They are prevailing as mainstream microprocessors. Now it is indispensable to make use of these SIMD-type extended instructions to take full advantage of the operational performance of microprocessors [5].

6.2.3 MISD Architecture

In the MISDs shown in Fig. 6.3c, multiple CUs read out instruction streams that differ from each other and which control multiple PUs. Although each PU works based on different controls, MISD processes a single data stream successively, regarding it as a whole. It is difficult to find a commercial product of this type in the market. A coarse-grained pipeline, in which in-line PUs work as stages of the pipeline and one provides each stage with different controls, might seem to be an MISD architecture. Because we recognize that CUs provide different functions to PUs in performing parallel processing with MISD, it earns the designation of architecture for functional parallelism. Application-specific hardware such as an image processor array that executes different processes of conversion of pixel values, edge detection, cluster classification, and others form each stage of pipeline corresponds to MISD-type architecture if each processing iteration is controlled by an instruction stream [6, 7].

6.2.4 MIMD Architecture

Regarding MIMD in Fig. 6.3d, multiple CUs read out instruction streams that differ from each other and which control multiple PUs independently. Different from MSID, each PU performs parallel processing based on different controls and for different data streams. Accordingly, MMID has an architecture simultaneously accommodating data parallelism and functional parallelism. It might perform different processes for multiple data based on different instruction streams. A tightly coupled multiprocessor like a symmetric multiprocessor (SMP), for which multiple processors and multiple processor-cores are connected on a common memory system, is an example of MIMD-type architectures. A cluster-type computer in which computation nodes made of memory and microprocessor are connected by the interconnection network is another example of MIMD-type architectures.

6.3 Systolic Algorithm

6.3.1 Systolic Algorithm and Systolic Array

Systolic algorithm is a general name for algorithms in which a systolic arrays [8, 9] are used to realize parallel processing. A systolic array is a regular array of many processing elements (PEs) for simple operations. It has the following characteristics:

1. PEs are arrayed in a regular fashion: they have the same structure or a few different structures.
2. The neighboring PEs are connected allowing the data movement to be localized with the connection. If a bus connection is used in addition to the local connection, then the array is designated as a semi-systolic array [8].
3. PE repeats simple operations and related data exchange.
4. All PEs perform operations in synchronization with a single clock signal.

Each PE performs its own operation in synchronization with the data exchange between neighboring PEs. Data to be computed flow into an array periodically, and the pipeline and parallel processing are performed while the data propagate in the array. Operations in the PE and the data stream caused by the data exchange between neighboring PEs resemble a bloodstream driven by the rhythmic systolic movement of the heart, hence the name systolic array. A PE is also designated as a cell.

Because the systolic array can scale the performance according to the array size by arranging PEs with simple structures and localized data movement, it is suitable for the implementation in an integrated circuit. Many applications were proposed for systolic arrays in 1980s and 1990s. Figure 6.4 portrays typical systolic arrays and systolic algorithms [9]. Systolic arrays have three types: 1D arrays having a linear array, 2D arrays having a lattice-like array, or a tree structure array with a tree-like connection. Many systolic algorithms for these arrays have been suggested, including signal processing, matrix operation, sorting, image processing, stencil calculation, and calculation in fluid dynamics.

Although the initial systolic array assumed hardwired implementation of fixed structures and functions, a general-purpose systolic array with a programmable or reconfigurable structure was proposed later [10]. Classification of systolic arrays in

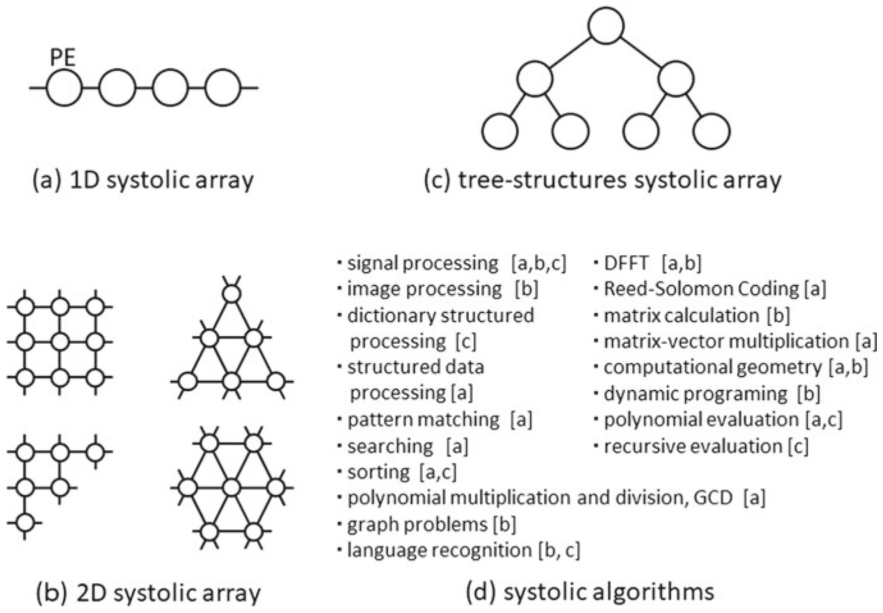

PE
(a) 1D systolic array
(c) tree-structures systolic array
• signal processing [a,b,c]
• image processing [b]
• dictionary structured processing [c]
• structured data processing [a]
• pattern matching [a]
• searching [a]
• sorting [a,c]
• polynomial multiplication and division, GCD [a]
• graph problems [b]
• language recognition [b, c]
• DFFT [a,b]
• Reed-Solomon Coding [a]
• matrix calculation [b]
• matrix-vector multiplication [a]
• computational geometry [a,b]
• dynamic programing [b]
• polynomial evaluation [a,c]
• recursive evaluation [c]
(b) 2D systolic array
(d) systolic algorithms

Fig. 6.4 Representative systolic arrays and systolic algorithms [9]

terms of general versatility is shown in Table 6.1 [10]. In this table, "Programmable" denotes the capability to dynamically change the function of the circuit by programming the fixed circuit. The "Reconfigurable" is the capability to statically change the circuit function by circuit reconfiguration. With the increasing scale of the systolic array, high-frequency operations with globally maintained synchronization become difficult sue to the propagation delay of the clock signal and other factors. Kung proposed the wave-front array, introducing data-flow to a systolic array to cope with this difficulty [11]. In his method, a PE, designed as an asynchronous circuit, operates with its own speed without the synchronization to a single clock signal. Furthermore, the data exchange between neighboring PEs is done using the handshake method.

Examples of systolic arrays and algorithms for 1D and 2D systolic arrays are introduced in the following sections.

6.3.2 Partial Sorting by 1D Systolic Array

An operation of rearranging a given data row according to a given order is referred to as sorting. Sorting operations are important and are used in many applications. Here, we introduce a systolic algorithm that rearranges a given data row of n numerical elements in the descending order and returns the upper N data. Figure 6.5a depicts a 1D systolic array and its PEs, which partially rearrange the upper N data [12].

Table 6.1 Classification of systolic arrays for versatility [10]

Class	General-purpose									Special-purpose
Type	Programmable			Reconfigurable			Hybrid			Hardwired
Organization	SIMD or MIMD			VFIMD						VFIMD
Topology	Programmable		Fixed	Reconfigurable		Fixed	Hybrid		Fixed	Fixed
Intercon. Network	Static	Dynamic	Fixed	Static	Dynamic	Fixed	Static	Dynamic	Fixed	Fixed
Dimensions	n-dimensional ($n > 2$ is rare due to complexity)									n-dimensional

VFIMD Very-few-instruction stream, multiple data streams

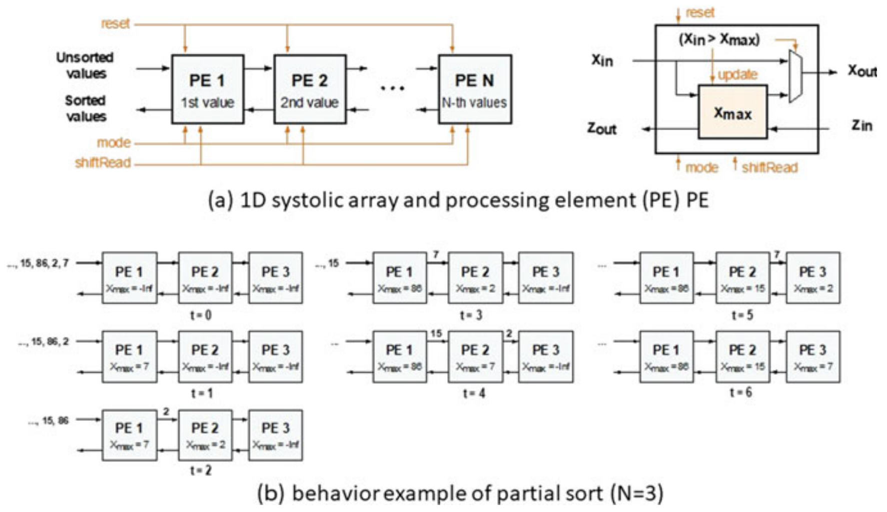


Fig. 6.5 Systolic algorithm of partial sort

Each of N PEs arrayed in the one-dimensional row has a register keeping the temporary maximum value, Xmax. Furthermore, if the input Xin is greater than Xmax, then Xmax is replaced with Xin. Consequently, the temporary maximum value is updated. If the update is made, the former temporary maximum value is sent to the right PE. If not, then Xin is sent to the right PE. When the last input data are sent to the Nth PE repeating the procedure described above, the upper maxima N are kept in the registers beginning with the left-end PE. An example of these operations is illustrated in Fig. 6.5b. Partial sorting of n data takes (N + n – 1) steps using a systolic array with N PEs.

Figure 6.5a shows that three control signals of reset, mode, and shiftRead are inputted to the systolic array. When the reset signal is asserted, the temporary maximum numbers in all PEs are reset to the possible maximum negative value. The mode control signal specifies a request: either the sorting operation or the reading out of the sorting result. In the former, 1 is inputted, whereas 0 is inputted in the latter. The shiftRead control signal asks to read out the result of sorting one by one in the descending order. As presented at t = 6 in Fig. 6.5b, the maximum values are lined up in a descending order starting with the left-end PE which has input and output ports. These values are read out by a systolic array used as the shift register through Zout and Zin connections. The control signal for shift is shiftRead.

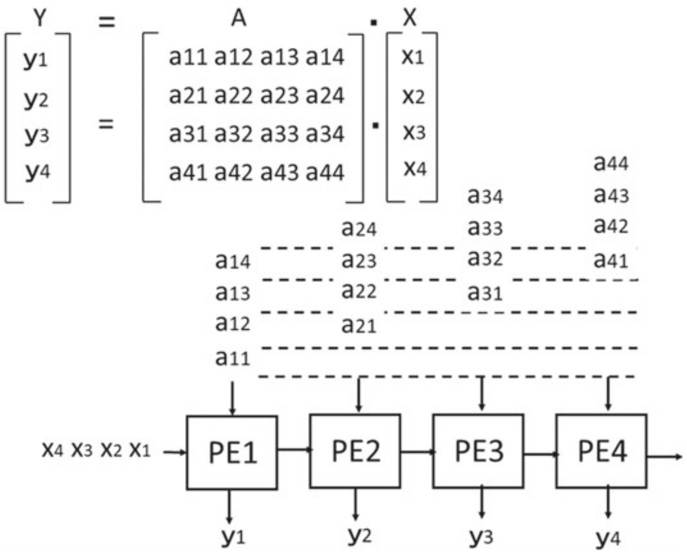


Fig. 6.6 Systolic algorithm of matrix-vector multiplication

6.3.3 Vector Product of Matrices by 1D Systolic Array

A 1D systolic array can perform the operation of vector product **Y = AX**. The operation of an $N \times N$ matrix requires N PEs. The systolic algorithm used for vector product of $N = 4$ matrices is depicted in Fig. 6.6. In PEs arrayed in the 1D row, the operation proceeds as follows: elements in **X** enter the left-end PE, whereas elements in matrix **A** enter each PE from above, both successively. Each PE has a register y_i to keep a temporary value of the element in the **Y** vector in addition to input x element of vector **X** and an element of matrix A. All PEs execute the calculation of $y_i = y_i + ax$ at each step and output x to the right neighbored PE.

At the beginning of the operation, y_i of each register is initialized to 0. Then, $y_1 = 0 + a_{11}x_1$ is calculated in PE1. In the next step, $y_1 = y1 + a_{12}x_2$ is executed in PE1 and $y_2 = 0 + a_{21}x_1$ in PE2, both being done in parallel. The inputs to PE1 through PE4 are done in a manner where the input to a PE occurs one step later than that to the left neighboring PE. This sends the data to the PE in a timely manner. When these operations are repeated until PE4 keeps the last matrix element, all elements of vector **Y**, from y_1 to y_4, are stored in the PE array. The columns of matrix **A** are inputted one element after another. Therefore, the total steps of operations become $2N - 1$.

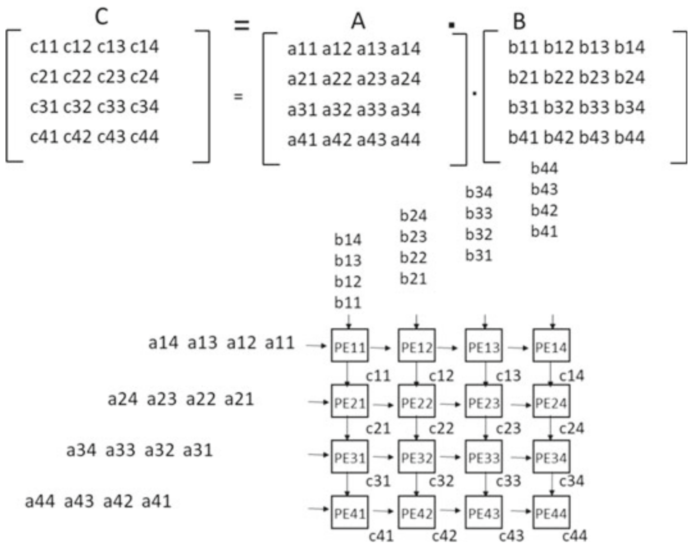


Fig. 6.7 Systolic algorithm of product of matrices

6.3.4 Product of Matrices by 2D Systolic Array

By extending the 1D systolic array discussed in the previous section to a 2D systolic array (with a lattice of PEs), it is possible to perform product operations of two matrices: **C = AB**. An $N \times N$ matrix multiplication requires an $N \times N$ systolic array with N^2 PEs. Figure 6.7 is an example of such an operation with $N = 4$. As in the vector product of matrices done by 1D systolic array, row and column elements are input from left and above top, shifting rows and columns. The function of a PE is the same as that in the previous section. All the internal registers are to be initialized to 0 at the beginning of the operation. When the last matrix elements of PE_{44}, a_{44}, and b_{44}, are input, all the resulting elements of Matrix C are in all the PEs. The number of required steps is $3N - 2$.

6.3.5 Programmable Systolic Array for Stencil Computation and Fluid Simulation

Although the examples introduced in the previous sections were those of simple PE operations, programmable systolic arrays oriented to many stencil computations, and applications for computational fluid dynamics (CFD) and others have been proposed [13–15].

The structure of a systolic computational-memory array and its PE designed for stencil computation is shown in Fig. 6.8. This array has a 2D scheme of vertically and

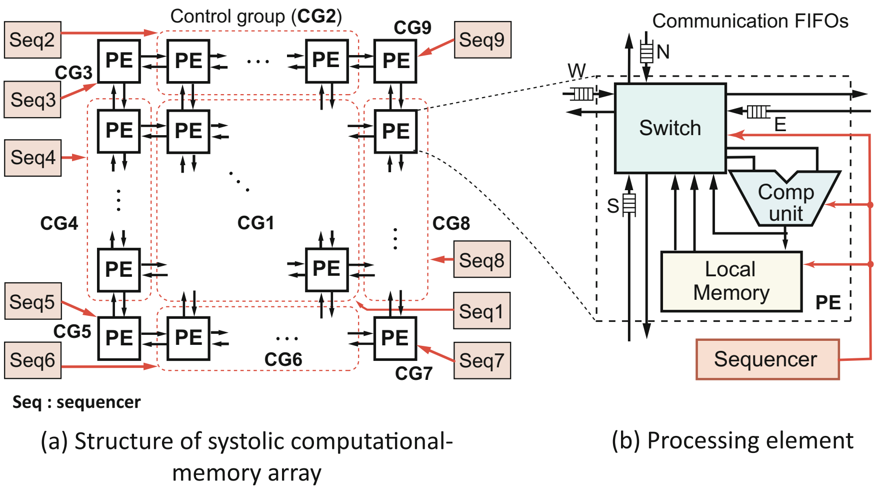


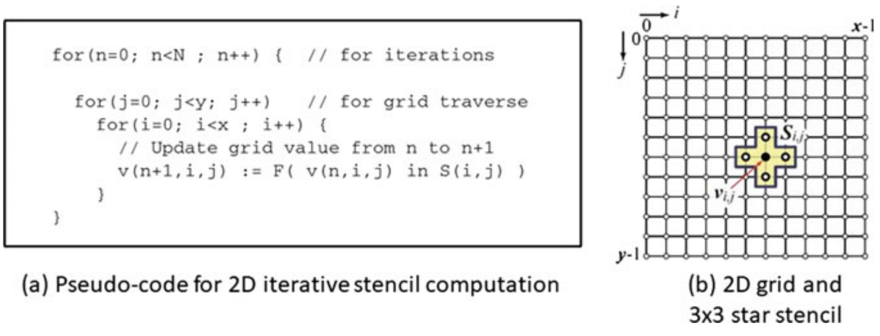
Fig. 6.8 Systolic computational-memory array and processing element

horizontally connected PEs. As shown in Fig. 6.8b, a PE comprises a computation unit, a local memory, a switch to send data to all directions of (W, E, S, and N) and a sequencer to control these elements using a microprogram. Because each PE has a large local memory and because the whole array is a memory not only for operations but also for data storage, this systolic array is considered as a computational memory. The computation unit can execute floating-point multiplications and additions. The computational sub-lattice data are stored in the local memory. A PE can perform various computations with a microprogram by repeating data read operations from the local memory or the neighboring PEs.

As shown in Fig. 6.8a, the systolic computational-memory array operation is made from many control groups (CGs). PEs in the same CG are controlled by a common sequence. They perform parallel processing of SIMD style. In this example, there are nine CGs inside the 2D array, with four sides and four corners because, in fluid dynamic computation and others, the computation in a regular pattern is done inside, whereas computation of different types is done for the boundary condition.

Figure 6.9 depicts pseudo-codes for stencil computation and an example of a 2D lattice with a 3 × 3 star-stencil computation of. As shown in Fig. 6.9b in stencil computation, an operation for a lattice point is done using data from neighboring points and the data on which the lattice point is renewed. The neighboring domain, where the data is referred, is designated as a stencil. The 3 × 3 star stencil shown in the figure, a fundamental one, is widely used. In 2D operations, the data at all lattice points are renewed after the same operations with the same stencil are performed over the lattice.

Figure 6.9a shows pseudo-codes for the 2D stencil computation. It has a triple loop structure consisting of loops for vertical and horizontal directions, and one for iterating them in a time step n. Function F(), the loop body, represents any



```
for(n=0; n<N ; n++) {   // for iterations

  for(j=0; j<y; j++)    // for grid traverse
    for(i=0; i<x ; i++) {
      // Update grid value from n to n+1
      v(n+1,i,j) := F( v(n,i,j) in S(i,j) )
    }
}
```

(a) Pseudo-code for 2D iterative stencil computation
i
0
j
x-1
y-1
$S_{i,j}$
$v_{i,j}$
(b) 2D grid and 3x3 star stencil

Fig. 6.9 2D stencil computation [13–15]

operation using data stored in a stencil. The product-sum operation using a weighting coefficient, shown below, is commonly used as F().

$$v(i, j) := c0 + c1v(i, j) + c2v(i-1, j) + c3v(i+1, j) + c4v(i, j-1) + c5v(i, j+1)$$

Here ":=" stands for the value update after the operation is written at the right-hand side. The PE in Fig. 6.8b performs the operation shown above for the partial lattice stored in the local memory, by using a microprogram in a sequencer. Sano et al. have derived the stencil algorithm by a fractional-step method for fluid dynamical phenomena. There, they repeated stencil calculations with different coefficients shown above and proved the calculations execution with the systolic computational-memory array. Further details can be found in [13–15].

6.3.6 Data-flow Machine

A data-flow machine [16, 17] is a computer architecture that directly contrasts the traditional von Neumann architecture or control flow architecture. It does not have a program counter (at least conceptually), and the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable. Figure 6.10 compares the data-flow machine with the von Neumann one. Since the data-flow machine has no instructions, it can eliminate the bottleneck caused by the instruction memory fetch of the modern computer execution time.

Although no commercially successful general-purpose computer hardware has used a data-flow architecture, it has been successfully implemented in specialized hardware such as in digital signal processing, network routing, graphic processing, telemetry, and more recently in data warehousing.

A data-flow machine executes the data-flow graph for a given program as shown in Fig. 6.11. "Fork" copies the given data, "Primitive Operation" outputs the result of

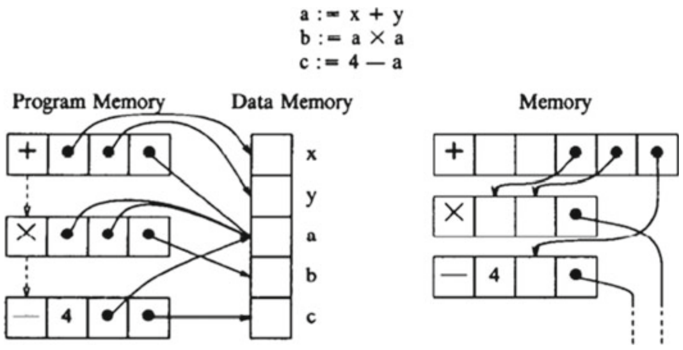


Fig. 6.10 A Comparison Von Neumann machine (Left) with a data flow machine (Referred by [16])

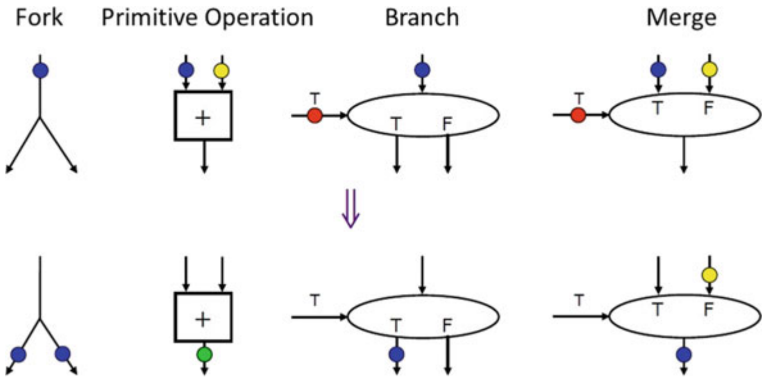


Fig. 6.11 Examples of a data flow node

the executed operation, "Branch" executes the conditional jump corresponding to the signal (True or False), and "Merge" selects the signal corresponding the conditional signal. Figure 6.12 shows the data-flow graph which executes the operations shown in Fig. 6.10. In Fig. 6.12, "◯" denotes the data to be executed, and it is called by the "Token." Here we show the value of token in the circle. First, the adder starts the operation, since two tokens have their values. Then, it generates the execution result. Next, the latter multiplier and subtractor starts the operation since it has received the input token. In this manner, it is possible to easily know the inherent data parallelism in the program to be processed.

Similar to the von Neumann machine, the data-flow machine can realize the conditional jump and loop operations. Figure 6.13 shows an example of the conditional jump. It can be realized by the Branch and the Merge nodes. When the token arrives at the Branch node, it executes the branch operation, then, it sends the token to the selected operational node. Finally, the Merge node selects the output corresponding conditional signal. Figure 6.14 shows an example of the loop operation. While

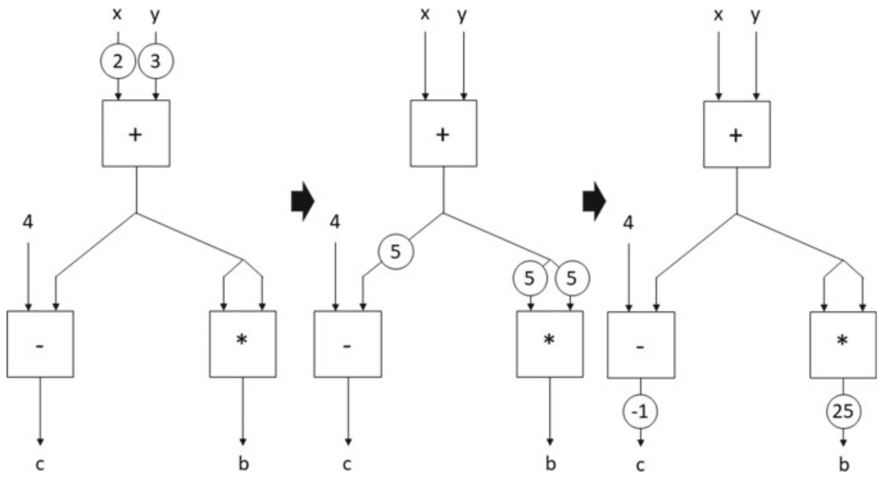
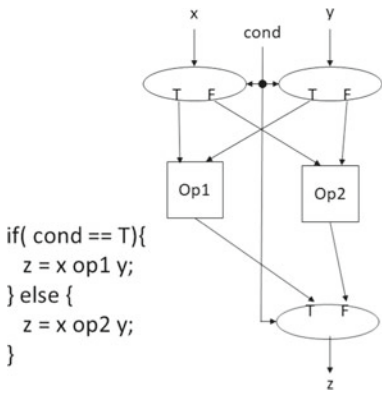


Fig. 6.12 An example of a data flow node

Fig. 6.13 A branch operation for a data flow graph



```
if( cond == T){
  z = x op1 y;
} else {
  z = x op2 y;
}
```

updating the initial value at the Merge node, it repeats the execution node through the Branch when the condition is satisfied. There are two types of data-flow machines that realize programs including loops. One is a static data-driven method, and the other is a dynamic data driven one. The static one expands all the loops and represents them as a flatten data-flow, while the dynamic one shares the executing unit with the processing of the subsequent loops using the data-flow of the loop body. The static data-driven method is based on a concept of a pure-driven method with a static data-flow graph. However, in many cases, since the size of the data-flow graph becomes too large to realize it with a data-flow machine which has a limited hardware resources. On the other hand, in the dynamic data-driven method, execution nodes in the loop body are shared by computing units, such as adders, subtractors, and multipliers. Thus, a control circuit must be provided. Otherwise, if multiple tokens between iterations exist, it is not possible to guarantee the computation result.

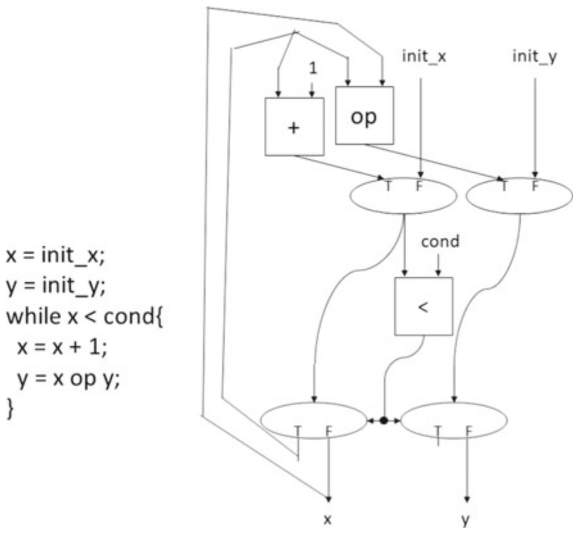


Fig. 6.14 A loop operation for a data flow graph

Figure 6.13 and 6.14 explains such a scenario. If y is updated before updating x in the loop, the computation result will differ from the expected one.

In the dynamic data-driven method, a tag is attached to the token in order to distinct present/next tokens in the iteration. It is called a tagged token method, or a colored (tokenized) one. By using tagged tokens, operations on tokens with the same tag can be guaranteed.

6.3.7 Static Data-Driven Machine

In the static data-driven method, it is often used to represent the operation and the operand of a node in a mixed manner. In the data-driven machine proposed by Dennis [18], the node information of the data-flow graph has the necessary data for the operation, the storage type of the calculation, and the destination for the storage. By using this node information as a token packet, the data-driven processing is realized. Note that, the token does not have tag information. It focuses on the static data-flow which does not include the loop processing. Figure 6.15 shows a hardware structure and each operand cell for the static data-driven architecture. In Fig. 6.15a, the operand cell stores the above information, and it represents the data-flow graph as an instruction of the data-driven machine in the whole instruction having valid information.

Hereafter, we explain the processing steps. When the operands are complete, the operation packet is sent through the arbitration network (ANET). This packet includes all information of the instruction cell, the operation type, and the storage destination for the result. The operation's result is transformed as a data token through the

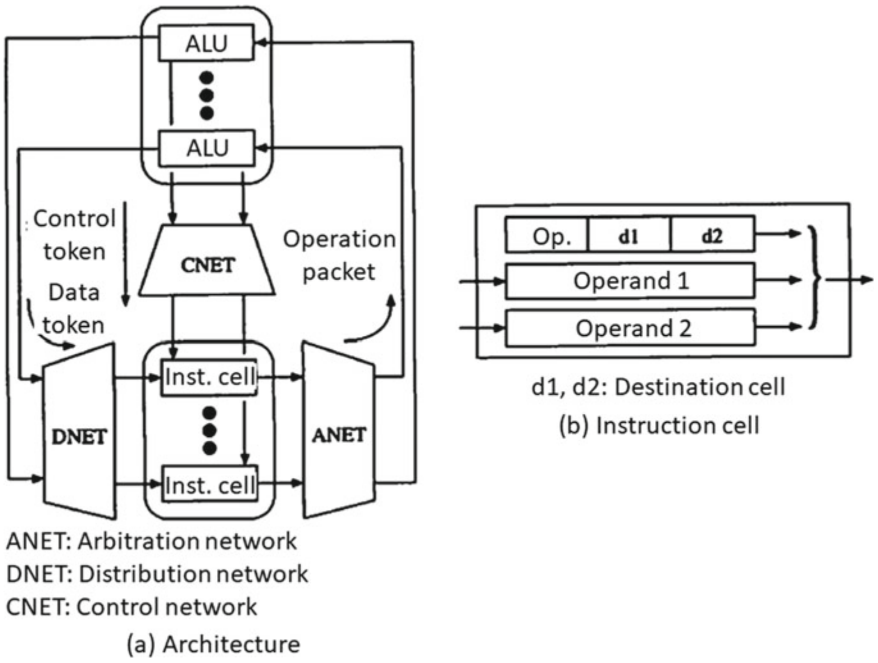


Fig. 6.15 A static data flow machine (Referred by [17])

distribution network (DNET). Then, it is written to the operand part in the instruction according to the storage destination (d1, d2, in Fig. 6.15b). Next, the instruction with the operands sends the operation packet at any time. By following these steps, a series of data driving driven processes is done.

6.3.8 Dynamic Data-Driven Machine

In the dynamic data-driven method, it separately represents the execution unit and the operation of the node. This representation can separate the flow graphs and data. Therefore, there is an advantage that loop processing can be considered by using a tagged token.

As shown in Fig. 6.16a, Arvind proposed a data-driven machine [19], which consists of N PEs with an $N \times N$ crossbar. Figure 6.16b shows the operand for the machine; “op” denotes an operation, “nc” denotes the number of constant to be stored, “nd” denotes the number to destinations, “constant 1” and “constant 2” denote the destination address, “s” denotes a statement number for the destination operand, “p” denotes the input port number for the destination operand, “nt” denotes the number of tokens for the destination operand, and “af” denotes an assignment function to be

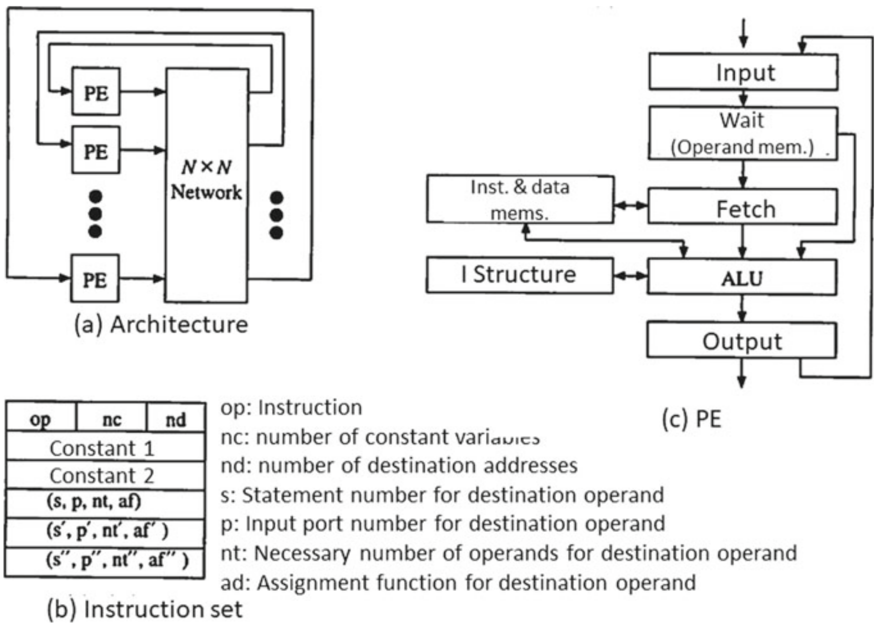

PE
PE
PE
N×N Network
(a) Architecture
Input
Wait (Operand mem.)
Inst. & data mems.
Fetch
I Structure
ALU
Output
(c) PE
op
nc
nd
Constant 1
Constant 2
(s, p, nt, af)
(s′, p′, nt′, af′)
(s″, p″, nt″, af″)
op: Instruction
nc: number of constant variables
nd: number of destination addresses
s: Statement number for destination operand
p: Input port number for destination operand
nt: Necessary number of operands for destination operand
ad: Assignment function for destination operand
(b) Instruction set

Fig. 6.16 A dynamic data flow machine (Referred by [17])

used for the determination of a PE assignment. The af has four parameters. Operands only represent a data-flow, while the execution data are represented by data tokens. That is, the program (data-flow) and the data are separated. The data token consists of the statement number for the destination operand, the tag (color), the input port number, and the destination operand data. By changing the tag, the data-flow graph can execute the loop operation.

Figure 6.16c shows the structure of the PE and the execution steps are conducted as follows. The input unit receives the data token from the interconnection network or the output of its own PE. It is executed until receiving the operand data which are necessary for the execution of the operation. That is, associative search is performed on the operand memory for the statement number and tags of the data token. Two operands are necessary considering the case of performing binomial operation. If one operand has already been received, it is stored in the register. By associative search, it is possible to check whether the necessary operands for the operation are received. When the reception of the necessary operands is completed, the next instruction fetch unit reads the operation information from the instruction memory using the statement number of the storage destination. The newly arrived operand is read from the waiting part. At the same time, another received operand is read from the operand memory. As a result, the necessary information is completed. Then, the calculation is performed in the ALU. Finally, the operation result as a data token is sent to the destination storage following the instruction.

Note that, the "I structure," which is similar to an array, provides a queuing function for a simple data structure. In array access in data-driven method, the data reading of certain array elements may occur before the data generation of ones. After data writing, a 1-bit existence (presence) bit is used for each element of the memory in order to guarantee reading. If its value is 1, it indicates that it has been written; otherwise, it is unwritten. If the presence bit is zero at the reading, it is suspended until the writing is completed. Thus, it is possible to guarantee synchronous data access with hardware.

The above is an overview of two data-driven methods and architectures, i.e., static and dynamic ones. Both of them are different when it comes to arithmetic operation control. This chapter outlined the earliest first-generation data-driven machines; however, regarding the second-generation machines, refer to [20].

6.4 Petrinet

A petrinet (also known as a place/transition (PT) net) is a directed bipartite graph, in which the nodes represent transitions and places. For example, events may occur, and they are represented by bars, while conditions are represented by circles. The directed arc denotes pre- and/or post-conditions for the transitions specified by arrows. The petrinets were introduced in 1939 by Carl Adam Petri for the purpose of describing chemical processes. A variation of the petrinet is a signal transition graph (STG), which is used to describe parallel or asynchronous systems.

By converting the data-flow graph shown in Fig. 6.17 to a petrinet, we have a new graph as shown in Fig. 6.18. The place corresponding to the input data has a token which represents the data. A condition to activate the transition, which represents the operation, is that there are at least more than one token on all the input places connecting the transition. After the activation, it generates a token to the place corresponding to the output. This means that the petrinet can easily represent both the data flow and its operation for the data-driven method. Figure 6.19 shows basic operations for the petrinet, e.g., parallel operation and synchronization. For more details about the petrinet and its parallel description, please refer to [21].

6.5 Stream Processing

6.5.1 Definition and Model

A processing method by which successive operations are done for successively inputted rows of data is referred to as stream processing [22–24]. The data element might be a single scalar data or a vector data including several words. Although the processing time is proportional to the number of elements (stream length), because

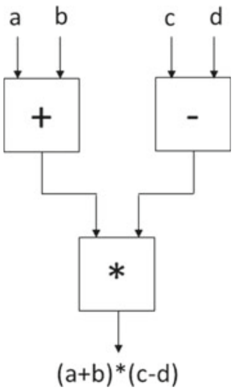


Fig. 6.17 An example of a data flow graph

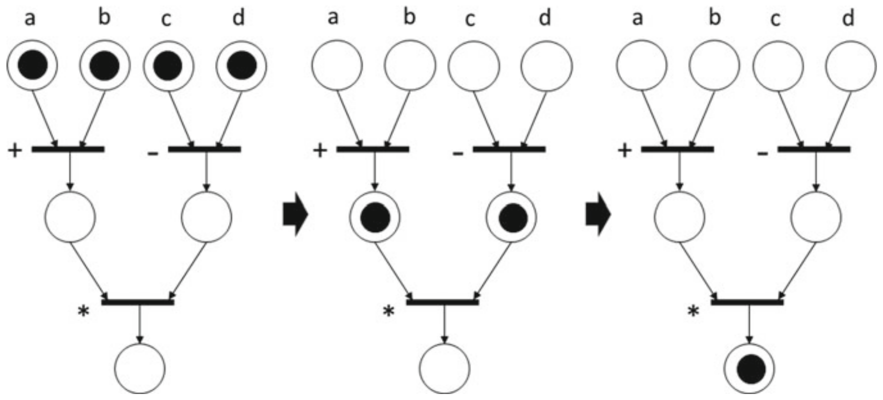


Fig. 6.18 An example of a Petrinet

only one iteration of the processing is executed at a given time, it can process any giant dataset in a sufficient time. The device for stream processing is not equipped with memory for all stream data. Data are supplied usually from an external memory, a network, or sensors. The device is used to process data that are too large to be stored in its own memory. A use case can be found in statistical information where inquiries are coming to a server through the Internet. In addition, when using stream data stored in external memory, an efficient use of data bandwidth by reading out data regularly with continuous addresses might be expected.

The processing of one element of stream data is designated as a processing kernel. Figure 6.20 depicts a model of stream processing by a single kernel. Here, the input is a data stream, but the output might be a data stream or might not be dependent on processing. Additionally, stream processing might be done by connecting processing

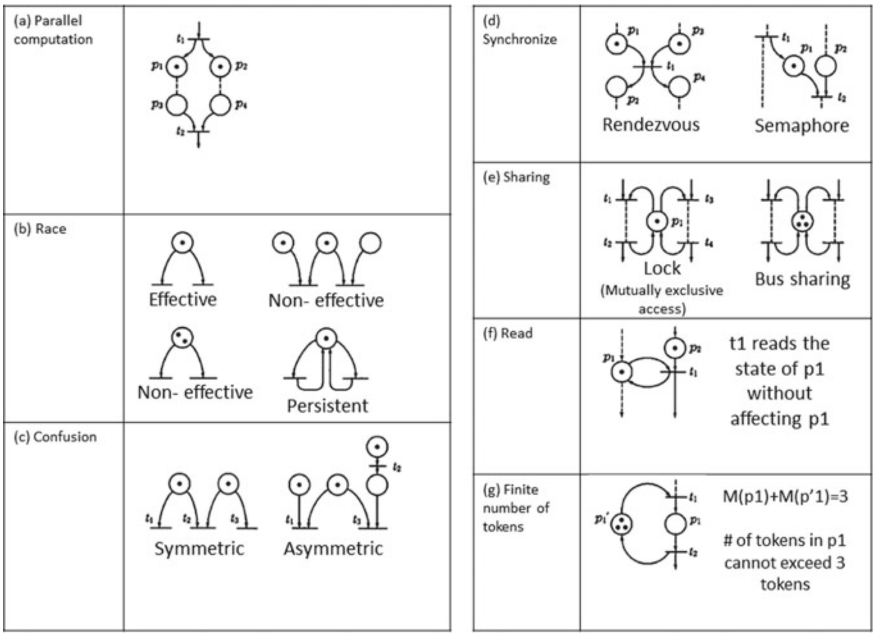


Fig. 6.19 Basic operations for a Petrinet (Referred by [21])

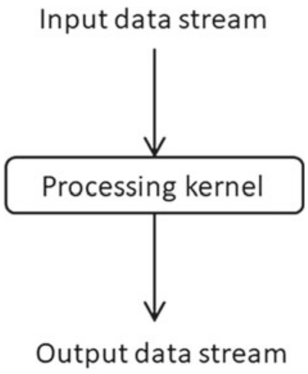


Fig. 6.20 Stream processing with a single kernel

kernels according to their dependency, as shown in Fig. 6.21. This is an expression of stream processing by a data-flow graph where the processing kernel is a node in the graph.

6.5.2 Hardware Implementation

Several methods exist to realize the stream-processing concept. As a means to accomplish stream processing of high throughput by implementing software, it is possible to

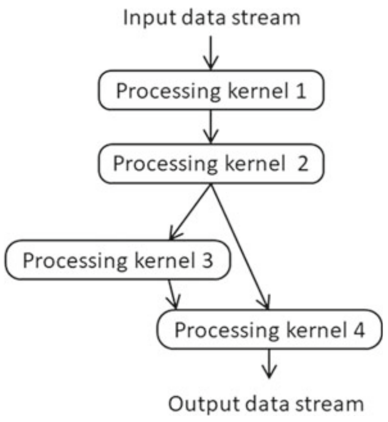


Fig. 6.21 Stream processing with multiple kernels

incorporate vector instruction or SIMD instruction into a general-purpose microprocessor. These instructions can rapidly process vector data of definite length. However, a general-purpose microprocessor presents many limitations on parallelism and inefficient input and output data streaming caused by deep memory hierarchy. Consequently, high-performance stream processing generally requires hardware implementation.

Designing a high-throughput stream-processing hardware usually depends on a structure that performs many operations included in a unit processes in parallel. This structure requires a hardware design based on a parallel processing model such as pipelining, systolic algorithm, and data-driven approaches. As shown in Fig. 6.21, to realize stream processing for multiple kernels, given sufficient hardware resources, kernels designed as pipeline modules can be connected to each other and thereby statically implement a giant pipeline. If sufficient hardware resources are available to implement all necessary kernels simultaneously, high-speed stream computing can be achieved where we can input and output stream elements at each cycle.

What should be managed if sufficient hardware resources are not available? In such a case, not all the necessary processing kernels can be implemented. Hardware designers might opt for a design where different kernels share the same hardware resources and the processing of one stream data element is performed over a longer time. One discussing an example of such a case, shown in Fig. 6.21, the question that could be asked is: Where can we implement only half of the hardware resources for processing kernels? In this case, we can implement a module for kernel 1 and kernel 2 and another module for kernel 3 and kernel 4, and then we switch the mode, as shown in Fig. 6.22a. This resembles folding the original data-flow graph to get a smaller one and making the hardware mapping on it. Therefore, this method is referred to as folding. When using folding, the hardware works in time-sharing mode, as shown in Fig. 6.22b. First, the processing of kernel 1 for the input data occurs. Then, the mode is switched to execute kernel 2. Similar processing is done for kernels 3

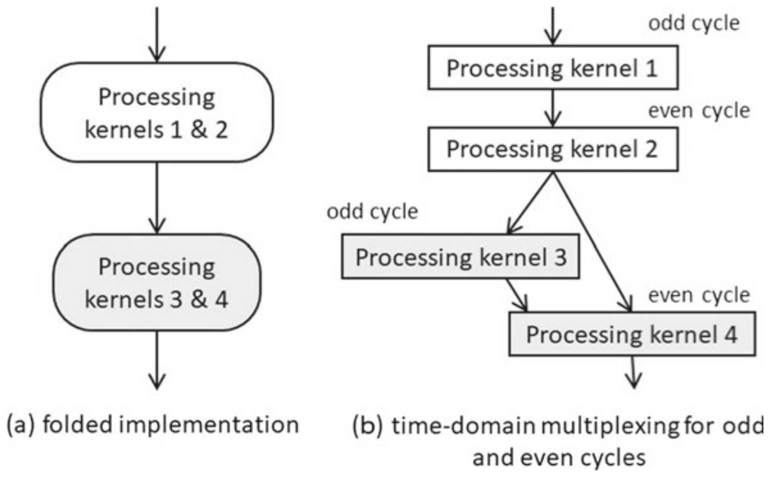


Fig. 6.22 Resource-saved implementation by folding kernels

and 4. Although the cycle time is doubled and the throughput decreases by half, with a small amount of hardware resources, stream processing can be accomplished.

In the example given above, all kernels work all the time and the operation rate is 100%. In this case, the operation rate of the folded processing kernel node is greater than 100%. The throughput decreases as a result. However, as in the case of conditional branching processing, the rate of operation is naturally less than 100%. Some kernels might have an operation rate less than 100%, even if aggregated with the operations of other kernels. Under these circumstances, the hardware resource consumption might be reduced without lowering the rate of operation in expense of a complicated control. Figure 6.23 demonstrates a simple example of folding with a rate of operation which is less than 100 Fig. 6.23a including conditional branching, two operations $x + y * z$ and $x * y + z$ are performed simultaneously. The signal selector outputs the results of the operation in two kernels with proportions of 90% and 10%, respectively. Accordingly, the real operation rates of both kernels are less than 100%. For this case, a module that can process both formulae, while considering common operations, can be implemented. The computation unit is shared in this design. Figure 6.23b illustrates an example. A multiplier and an adder are implemented for each formula, and one selector is inserted to switch the operations of the formulae. Only one operation is necessary for one formula. Therefore, the consumption of hardware resources can be reduced without increasing the number of cycles for one processing iteration of one element of stream data.

In addition, a stream-processing iteration, where some dependence relations exist between successive elementary processing iterations, might be implemented by inserting a delay buffer memory. This buffer sends intermediate data of stream

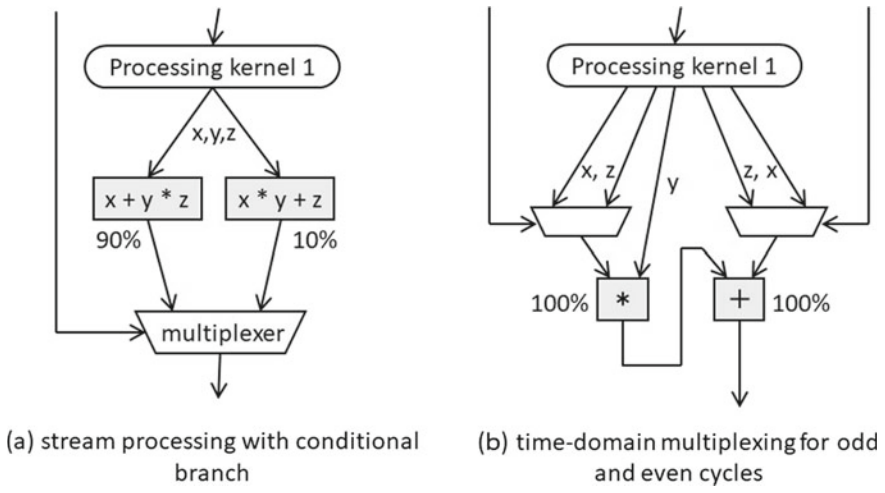

Processing kernel 1
x,y,z
x + y * z
x * y + z
90%
10%
multiplexer
(a) stream processing with conditional branch
Processing kernel 1
x, z
y
z, x
100%
*
+
100%
(b) time-domain multiplexing for odd and even cycles

Fig. 6.23 Folded implementation of underutilized stream processing

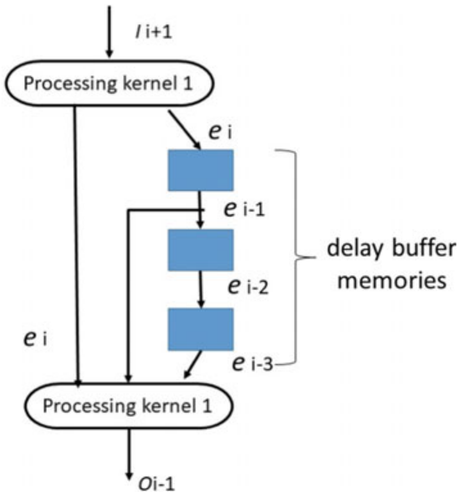


Fig. 6.24 Folded implementation of underutilized stream processing

processing to the next elementary processing iteration in a delayed fashion. In the example presented in Fig. 6.24, in addition to the current output result from processing kernel 1, delay buffer memories to send past data of to processing kernel 2 are inserted. Stream processing of stencil calculation applying delay buffer memory is introduced in the next section.

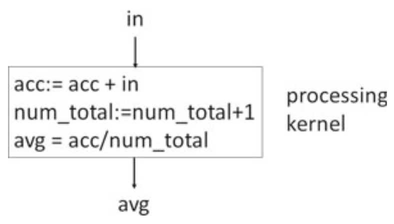


Fig. 6.25 Stream-processing hardware for averaging a series of scalars

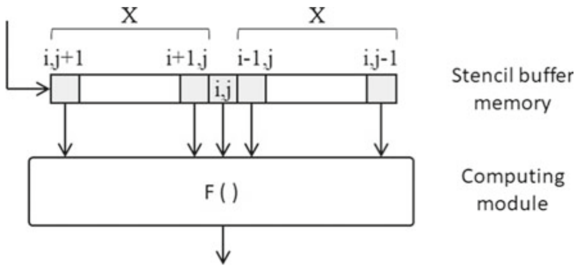


Fig. 6.26 Stream-processing hardware for 3 × 3 star-stencil computation (Fig. 6.9)

6.5.3 *Examples of Stream Processing*

An example of stream processing to find the average of a scalar array is depicted in Fig. 6.25. The processing kernel has two registers: acc and num_total. They are initialized to zero at the beginning of the operation. Whenever scalar data are inputted, it is added to acc and num_total is incremented by 1. At the end of one cycle, acc is divided by num_total to get avg; it is outputted as the average of the input data up to that moment.

Figure 6.26 displays an example of hardware for stream processing of 2D iterative stencil computation. This example is for the 2D iterative stencil computation shown in Fig. 6.9. It generates data stream of lattice data $v_{i,j}$ by traversing the 2D computational lattice, as shown in Fig. 6.9b in the x direction. In the processing kernel, the value at lattice point (i, j) is evaluated by function $F()$ using 5 data of $v_{i,j+1}$, $v_{i+1,j}$, $v_{i-1,j}$, $v_{i,j-1}$ in a 3 × 3 star stencil. In this case, a delay buffer memory is used because the previous and subsequent elements are necessary in addition to the current input element of input data [25]. This buffer memory is referred to as the stencil buffer memory.

Let X be the width of a 2D computational lattice. The stencil buffer is a 2X + 1 long shift register with a five readout ports for $v_{i,j+1}$, $v_{i+1,j}$, $v_{i-1,j}$, $v_{i,j-1}$ as shown in Fig. 6.26. After X cycles, after inputting the data at the current lattice point (i, j), the data appear exactly at the center of the stencil buffer. The five data in the star-stencil become readable simultaneously. The operation module makes use of this data and outputs calculated values at lattice point (i, j). A stencil computation for 2D lattice requires buffers that are proportional to the lattice width. Although the 3D lattice requires buffers that are proportional to the cross-sectional area of the lattice, it needs more buffer memory than the 2D one does. Therefore, on-chip memory cannot

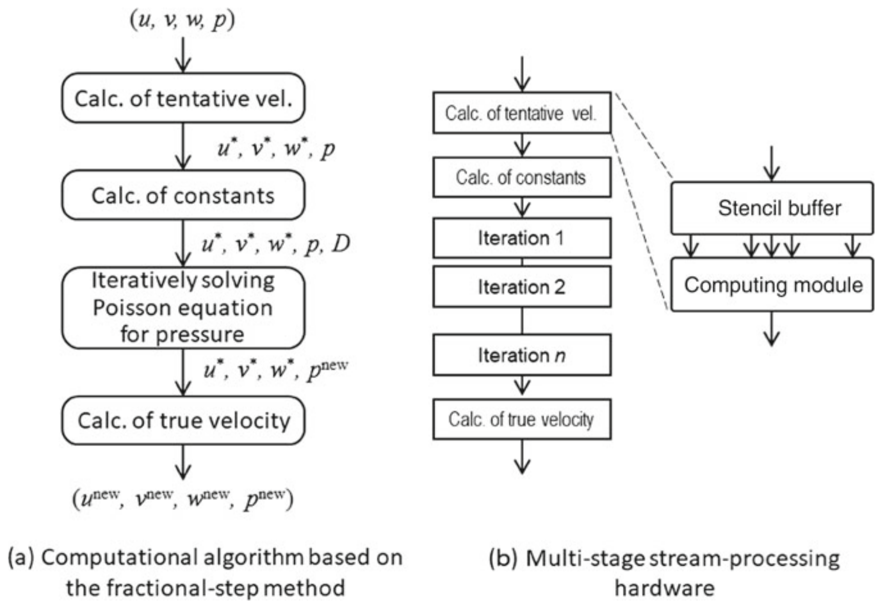


Fig. 6.27 Computational algorithm of incompressible fluid dynamics and its stream-processing hardware

provide sufficient capacity for a large 3D lattice. In this case, the calculation lattice can be divided into smaller partial lattices which are then computed one by one.

A hardware example of multiple computing stages for stream processing of incompressible fluid dynamics is shown in Fig. 6.27. As depicted in Fig. 6.27a, the algorithm, based on the fractional-step method used in CFD, comprises four stages of operations [13, 26]. It is known that each stage is of stencil computation, which refers to neighbors of each lattice point in the orthogonal lattice. For this reason, each step is implemented by stream-processing hardware which is composed of the stencil buffer memory and the computation module, as shown in Fig. 6.26. It is possible to construct stream-processing hardware to perform fluid computation for a single time step by the in-line connection of hardware for each computing stage [26]. Part of the iterative solution of a Poisson equation is implemented using a fixed construction with an *n* element array of hardware for one-iteration stencil calculation. This result is based on the experience that, although sufficient implementation has been done with conventional iteration methods to reduce the residual to a sufficiently small magnitude, the iterations were normally less than a definite number. The description above is a proposed solution to the case in which there is no definite number of iterations. However, the method and conditions should be thoroughly examined to address any practical problem.

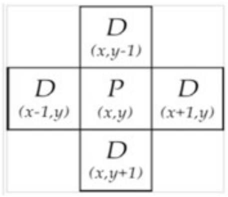


Fig. 6.28 Von Neumann neighborhood

6.6 Cellular Automaton

A cellular automaton is a discrete model studied in computability theory, mathematics, physics, complexity science, theoretical biology, and microstructure modeling. The concept was originally discovered in the 1940s by von Neumann [27]. It consists of a regular grid of cells, each in one of a finite number of states, such as *on* and off [28]. The grid can be in any finite number of dimensions. For each cell, a set of cells (referred to as neighborhood) is relatively defined to the specified cell. An initial state (time t = 0) is elected by assigning a state for each cell. A new generation is created according to some fixed rules that determine the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood. Typically, the rule for updating the state of cells is the same for each cell and does not change over time and is applied to the whole grid simultaneously, though exceptions are known.

Figure 6.28 shows an example of the neighborhood. Let K be the number of states for each cell. Then, five cells including the center have K5 states. Thus, there are KK^5 rules of cellular automaton based on the neighborhood. The well-known cellular automaton is a life game which can be specified with the following rules for K = 2.

- Born: If there are more than or equal to three living cells around the dead cell, then it lives in the next step.
- Living: If there are two or three living cells around a living cell, then it survives in the next step.
- Dead: Otherwise, it dies in the next step.

A finite cell is widely used to simulate the cellular automaton. Generally, although it is implemented assuming a finite rectangle, an implementation of the boundary becomes a problem. There is a method of treating all boundaries as a constant; however, the disadvantage is that the number of rules increases. Another way is to make it as a torus [29], which simulates an infinite rectangle by connecting upper, lower, left, and right respectively, and filling an infinite plane with the same rectangle in the same plane. Figure 6.29 shows a cellular automaton circuit using a torus connection by placing PEs in a 3 × 3 grid pattern. In the case of the game of life, the PE has the state of each cell, and it executes the above rules for each step. Then, it updates the state of the cell in the next step.

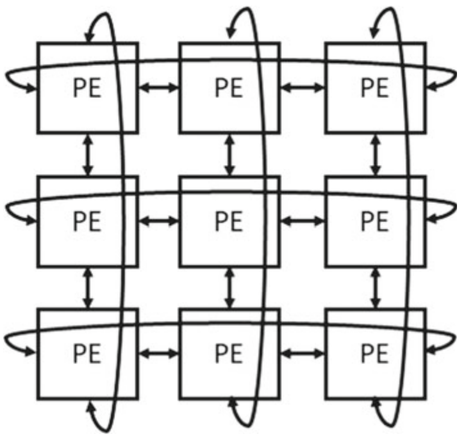


Fig. 6.29 Cellular automata circuit using a 3 × 3 PE

Since all the inputs and outputs in a cellular automaton circuit are parallel, it is highly compatible with FPGA implementation. Also, it has the possibility to surpass the existing von Neumann architecture. In practice, as the number of rules to be computed increases, the throughput can be improved using pipeline circuits. In recent years, an attempt has been applied to build more physical cellular automata from the viewpoint of materials, rather than circuits or devices [30]. It is expected to replace the existing von Neumann-type architecture by applying these realizations to FPGAs.

6.7 Hardware Sorting

In computer science, a sorting is an algorithm that puts n elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. FPGA-based hardware accelerations for sorting are widely used for database, image processing, and data compression. Here, we introduce a sorting network and a merge sort tree that are suitable for hardware implementation.

The simplest sorting algorithm for hardware implementation is the sorting network [31], based on the bubble sort. This algorithm sorts the neighboring elements in parallel. Figure 6.30 shows the sorting network of four elements. It consists of the exchange units (EUs) which sort the neighboring elements. In this circuit, the number of wires is n. Each element passes at most n − 1 EUs. In this case, since it can be performed in parallel, we can realize a fully pipelined EU circuit to increase the throughput. Note that, since it requires n-parallel wires, the amount of hardware tends to be large. The known sorting network on FPGA is the Batcherfs odd–even merge sort [32].

The other type of hardware sorting algorithms is the merge sort tree [33], which is based on the binary tree structure where each vertex is realized by the EU. This circuit has many FIFOs in the input and output, and performs the sorting in parallel.

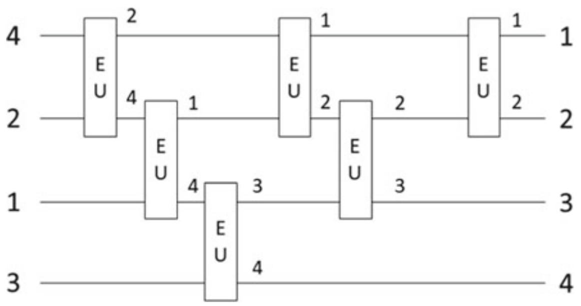


Fig. 6.30 A sorting network for four elements

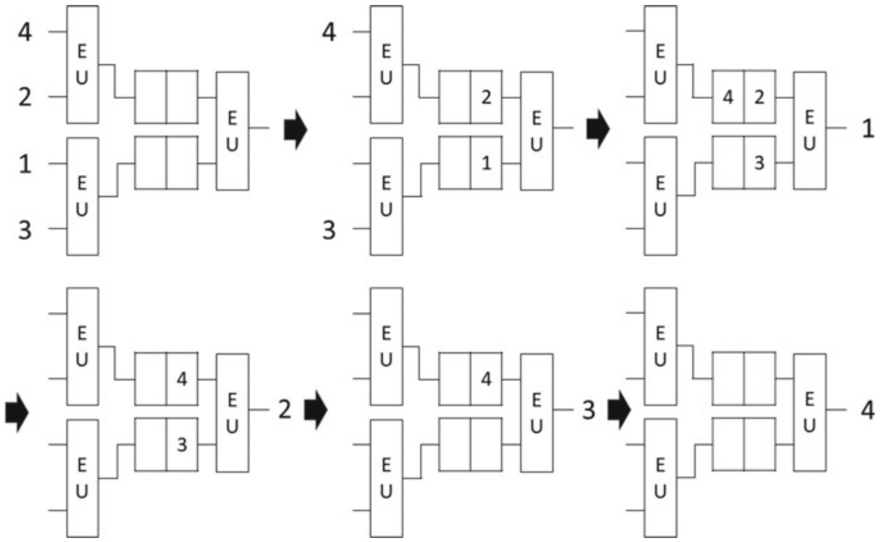


Fig. 6.31 An example of a marge sort tree for four elements

Figure 6.30 shows an example of the merge sort tree for four elements. In the merge sort tree, the input is a sequence to be sorted, and in each level of the tree, the sorted sequence is sent to the next level through the FIFO. Therefore, it is possible to increase the throughput by inserting a pipeline register for each level. To realize a high speed and small area on the FPGA, a combination of sorting network and merge sort tree shown in Fig. 6.31 has been proposed [34].

6.8 Pattern Matching

One of the killer applications for FPGA is the pattern matching which finds a given pattern in the data. Pattern matching algorithms are roughly categorized to an exact matching, a regular expression matching, and an approximate matching. In this section, we introduce the different algorithms for these matchings.

6.8.1 Exact Matching

An exact matching finds a fixed pattern; however, each element of the pattern takes three values, i.e., one, zero, and don't-care which can take both zero and one. Typically, the exact matching can be realized by a content addressable memory (CAM) [35]. Here, we introduce the index generation unit (IGU) N6.8-3 which is a CAM emulator, then we implement it on FPGA.

Let us suppose that the index generation function f is as shown in Figs. 6.32 and 6.34 shows the decomposition chart for f. In Fig. 6.34, the label in the right side denotes X_1 = (x2, x3, x4, x5), the label in the left side denotes X_2 = (x1, x6), and the entry denotes the function value. Note that, each column has at most one nonzero element. Thus, f can be realized by a main memory whose input is only X_1. The main memory maps a 2^n sets to k + 1 sets. This is an ideal case; however, in most cases, we must check X_2 since f may cause mismatch. To do this, first, we store the correct X_2 to an auxiliary (AUX) memory. Then, we use a comparator to generate a correct f when f is equal to X_2; otherwise, it generates zero. Figure 6.33 shows the IGU. First, we read the q from the main memory corresponding to p bit X_1. Then, $X0_2$ is read from the AUX memory corresponding to q. Next, q is generated if $X0_2$ is equal to X_2; otherwise, it generates zero.

Figure 6.35 shows an example of the IGU realizing the index generation function shown in Fig. 6.32. When (x1, x2, x3, x4, x5, x6) = (1, 1, 1, 0, 1, 1), the index "g6" corresponding to X_1 = (x2, x3, x4, x5) = (1, 1, 0, 1) is read. Then, $X0_2$ = (x1, x6) = (1, 1) corresponding to $X0_2$ = (x1, x6) = (1, 1) is read. Next, the correct signal is sent

Fig. 6.32 An example of an index generation function

x_1	x_2	x_3	x_4	x_5	x_6	f
0	0	0	0	1	0	1
0	1	0	0	1	0	2
0	0	1	0	1	0	3
0	0	1	1	1	0	4
0	0	0	0	0	1	5
1	1	1	0	1	1	6
0	1	0	1	1	1	7

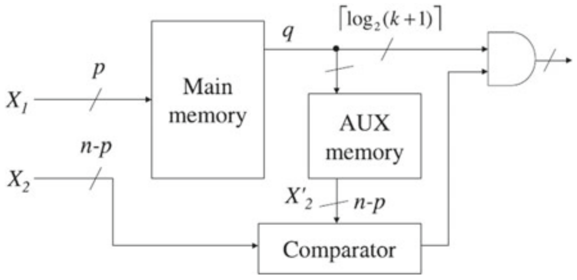
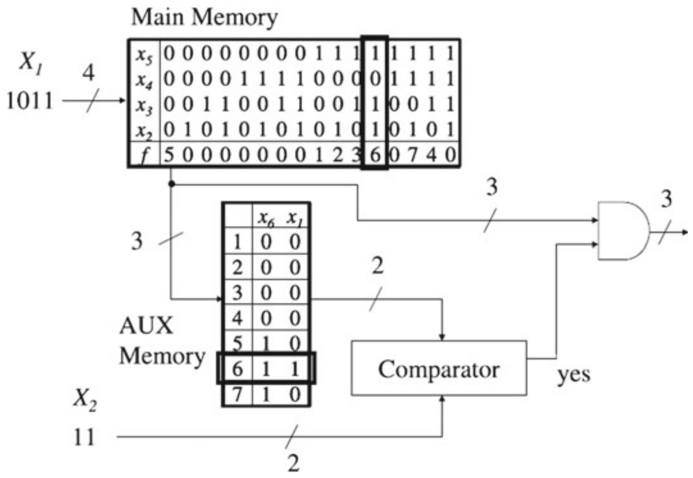


Fig. 6.33 An index generation unit (IGU)

	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_5	
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_4	X_1
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_3	
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	x_2	
00	0	0	0	0	0	0	0	0	1	2	3	0	0	0	4	0		
01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
10	5	0	0	0	0	0	0	0	0	0	0	0	0	7	0	0		
11	0	0	0	0	0	0	0	0	0	0	0	6	0	0	0	0		
x_6, x_1 X_2																		

Fig. 6.34 An example of a decomposition chart for an index generation function



x_5	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x_4	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x_3	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x_2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
f	5	0	0	0	0	0	0	0	1	2	3	6	0	7	4	0

	x_6	x_1
1	0	0
2	0	0
3	0	0
4	0	0
5	1	0
6	1	1
7	1	0

Fig. 6.35 Operation for an IGU

to the AND gate, then "g6" is generated. Since the IGU realizes a mapping which generates $k + 1$ sets from given $2n$ sets, its memory size is drastically reduced from $O(2^n)$ to $O(2^p)$. The theoretical analysis of the IGU is introduced in [36], and the applications for the IGU are presented in [37, 38].

6.8.2 Regular Expression Matching

A regular expression consists of a character and a metacharacter which represents a set of a strings. Various network applications use regular expression matchings to detect malicious data in incoming packets. Regular expression matchings spend a considerable fraction of the total computation time for these applications. The throughput using the Perl compatible regular expression (PCRE) library on a general-purpose MPU is up to hundreds of megabits per second (Mbps), which is too slow for most applications. Thus, a dedicated circuit for regular expression matchings is required. For network applications, since the high-mix low-volume production and the frequent update for new protocols are required, FPGAs are widely used. With the advent of FPGAs embedding dedicated high-speed transceivers for high-speed networks, we expect extensive use of FPGAs in the future.

Regular expressions are detected by finite automata (FA). In a deterministic finite automaton (DFA), for each state and each input, there is a unique transition. While in a non-deterministic finite automaton (NFA), for each state and each input, multiple transitions may exist. In an NFA, there exist ϵ-transitions to other states without consuming input characters.

Most of the proposed regular expression matching circuits are based on finite automata. An Aho–Corasick DFA (AC-DFA) [39] is a known algorithm. A combination of the bit-partitioned AC-DFA and the MPU is proposed [40]. Also, Baeza-Yates proposed the NFA algorithm based on a shift and bitwise AND operations [41], and its hardware realization on an FPGA is proposed [42]. A resource-efficient FPGA realization by prefix and postfix sharing of regular expressions [43], and mapping repeatedly appearance parts of regular expressions into a Xilinx FPGA primitive (SRL16) [44] have been also published.

Hereafter, we introduce the NFA-based regular expression matching algorithm which is suitable for FPGA realization. Figure 6.36 shows a conversion from a regular expression to NFA. In Fig. 6.36, ϵ denotes the ϵ-transition, and the gray circle denotes the accept state. Figure 6.37 shows the NFA representing the regular expression "abc(ab) * a", and shows the state transition for the input string "abca". For each element in the vector corresponding to the state in the NFA, and '1' denotes the active state. Figure 6.38 shows the circuit for the NFA shown in Fig. 6.37. To emulate this NFA, a memory is used to detect a corresponding character, and the detection signal is sent to the matching element (ME). The ME emulates the state transition, and generates the match signal. In the ME, the FFs store the vector shown in Fig. 6.37, where i denotes the transition signal from the previous state; *o* denotes the transition signal to next state; *c* denotes the character detection signal from the memory; *ei* and *eo* denote the in/out signals from the ϵ-transition.

Figure 6.39 compares the NFA [42] with the DFA [40] with respect to the complexity of the parallel execution hardware. Even if we apply the bit-partition which reduces the amount of hardware, the complexity still holds $O(\Sigma^{sm})$. When the number of rules for the regular expression increases, the amount of memory tends to be exponentially large for the DFA based realization, while the NFA-based one does

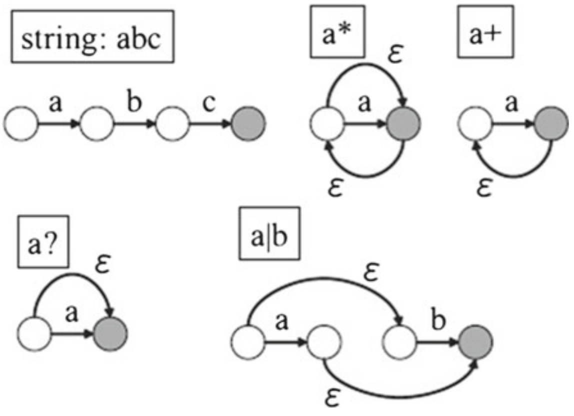
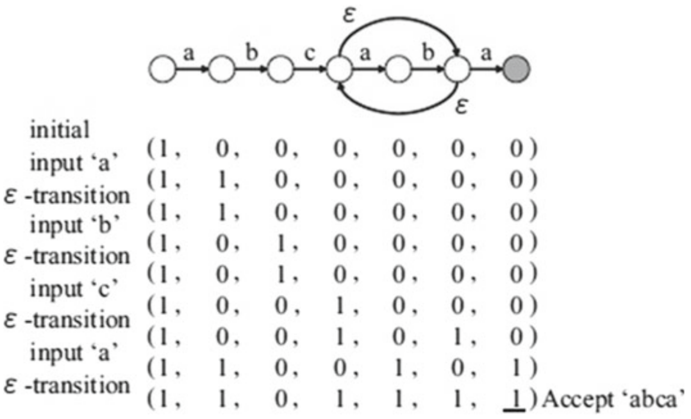
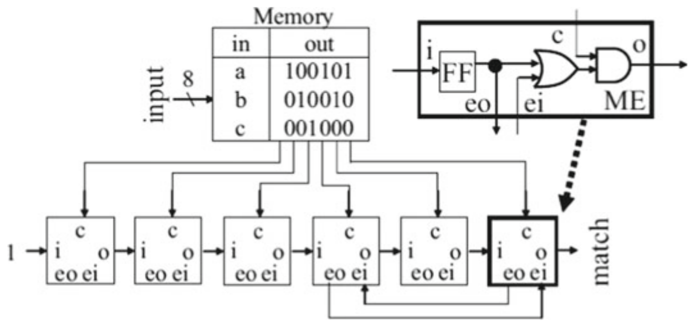


Fig. 6.36 Conversion from a regular expression to an NFA



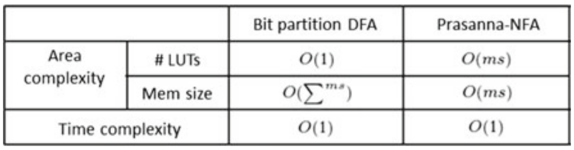
initial	(1,	0,	0,	0,	0,	0,	0)
input ‘a’	(1,	1,	0,	0,	0,	0,	0)
ε -transition	(1,	1,	0,	0,	0,	0,	0)
input ‘b’	(1,	0,	1,	0,	0,	0,	0)
ε -transition	(1,	0,	1,	0,	0,	0,	0)
input ‘c’	(1,	0,	0,	1,	0,	0,	0)
ε -transition	(1,	0,	0,	1,	0,	1,	0)
input ‘a’	(1,	1,	0,	0,	1,	0,	1)
ε -transition	(1,	1,	0,	1,	1,	1,	1) Accept ‘abca’

Fig. 6.37 An NFA accepting ‘abc(ab) * a’



in	out
a	100101
b	010010
c	001000

Fig. 6.38 Circuit for an NFA



		Bit partition DFA	Prasanna-NFA
Area complexity	# LUTs	$O(1)$	$O(ms)$
	Mem size	$O(\sum^{ms})$	$O(ms)$
Time complexity		$O(1)$	$O(1)$

Fig. 6.39 Comparison of complexities

not increase the amount of memory. Thus, the NFA-based one is suitable for FPGA implementations.

6.8.3 *Approximate Matching*

An approximate matching consists of finding an edited pattern in a text. It finds a corresponding pattern in the text while deleting, replacing, and inserting a character. Many approximate matchings are based on dynamic programming. Approximate matching is used in bioinformatics to evaluate similarity between the DNA sequences.

Let "ACG" be a text, and "TGG" be a pattern. Then, we compute the edit distance:

1. Delete "A" from the text "ACG," then we have "CG."
2. Delete "C" from "CG," then we have "G."
3. Insert "G" to "G," then we have "GG."
4. Insert "T" to "GG," then we have the text "TGG," which corresponds to the given pattern.
5. Terminate.

In this example, we set the editing score for both the insertion and deletion to 1. Since the replacement includes insertion and deletion, its score is 2. The above example showed that the editing score between "ACG" and "TGG" is 4.

Figure 6.40 shows a system for an approximate matching. The host PC sends a text and a pattern. The matching system reads the text from the buffer memory, and

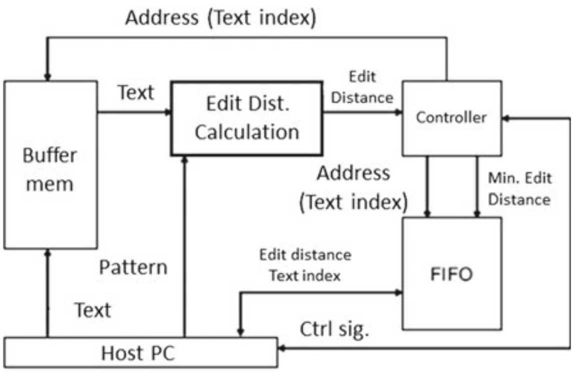


Fig. 6.40 System for an approximate matching

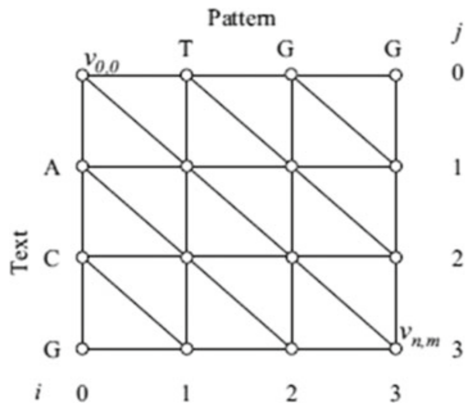


Fig. 6.41 An example of an approximate matching graph

then the editing calculating circuit computes the editing score for a part of the text and the pattern. The controller stores the address whose position of the text and its minimum editing score to the FIFO. The system shifts the text for calculating the score. When all the text is matched, the host PC reads the position of the matched pattern and its minimum score from the FIFO. Then, the edited pattern is computed. For approximate matching, since the calculation time for the editing score is dominant, FPGA accelerators are desired.

The editing score between two strings can be calculated using dynamic programming. The Needleman–Wunsch (NW) algorithm [45] computes the minimum value of the editing score of the entire text and a pattern, while the Smith–Waterman (SW) algorithm [46] computes the editing score of a part of text and a pattern. Here, we introduce the basic algorithm for calculating the minimum value of the editing score between two strings using dynamic programming.

Let P = (p1, …, pn) be a pattern, and T = (t1, …, tm) is a text. Suppose that the matching graph for an approximate string, which has $(n+1) \times (m+1)$ vertices labeled by each column and row. For a coordinate (i, j), a vertex v_i, g_j is placed. We assume that the upper-left vertex is set to (0,0), and the coordinates (i, j) increase toward the lower-right vertex (n, m). For $0 <= i <= n-1$ and $0 <= j <= m-1$, there are edges connecting $v_{i,j}$ to $v_{i+1,j}$ and others connect $v_{i,j}$ to $v_{i,j+1}$. Also, there are diagonal edges connecting $v_{i,j}$ to $v_{i+1,j+1}$. Figure 6.41 shows an example of the approximate matching graph for the text ACG and the pattern TGG.

Let s_{del} be an editing score for deletion, sins the score for insertion, and s_{sub} the score for replacement. Here, we set s_{del} = 1, s_{ins} = 1, and s_{sub} = 2. For each vertex $v_{i,j}$, we must make sure to edit score of a subpattern $P^t = (p_1, p_2, \ldots, p_i)$ and subtext $T^t = (t_1, t_2, \ldots, t_j)$. We define that the vertex score denotes the editing score for each vertex. The minimum vertex score for $v_{i,j}$ is obtained by the following expression:

$$v_{i,j} = min \begin{cases} v_{i-1,j-1} \\ v_{i-1,j} + s_{ins} + \begin{cases} 0 & if \ \ p_i = t_j \\ s_{sub} & otherwise \end{cases} \\ v_{i,j-1} + s_{del} \end{cases} \tag{6.1}$$

By applying the above expression recursively from the vertex $v_{0,0}$ to $v_{n,m}$, we can obtain the minimum editing score. The following algorithm shows how to obtain the minimum editing score:

Algorithm

Input: Text T with length m, and pattern P with length n.
Output: Minimum editing score at vertex $v_{n,m}$.

```
1: $v_{i,0} \leftarrow i$, $(i = 0, 1, \ldots, n)$, $v_{0,j} \leftarrow j$, $|(j = 0, 1, \ldots, m)$
2: **for** $j \leftarrow 1$ **until** $j \leq m + n - 1$ **begin**
3:     **for** $i \leftarrow 1$ **until** $i \leq n$ **begin**
4:         **if** $0 < j - i + 1 \leq m$, compute $v_{i,j-i+1}$ with Eq. 6.1.
5:         $i \leftarrow i + 1$
6:     **end**
7:     $j \leftarrow j + 1$
8: **end**
9: Let $v_{n,m}$ be edit distance and stop.
```

Here, we assume that $n \ll m$. For example, for an alignment in bioinformatics, $n = 10^3$ and $m = 10^9$. An algorithm, which calculates the minimum editing score by dynamic algorithm, is called the Naive method. It uses a processing element (PE) [47] to calculate each column of the approximate matching in parallel. Figure 6.42 shows the architecture of the PE for the Naive method. In this figure, s denotes the number of bits for each character, n denotes the number of characters in the pattern. To directly perform recurrent expression, the naive method receives a text (tin) and a pattern (p in). Then, it selects the replacement score or not through the corresponding detection circuit. At that time, for each vertex, it calculates the editing score. Then, it selects the minimum score selector to generate the minimum one.

Each PE calculates the score corresponding vertex, then outputs it in parallel. Note that, t denotes the time stamp. We consider the data dependency to compute the vertex $v_{i,j}$ by the PEi. To compute $v_{i,j}$, three pieces of data for $v_{i,j-1}$, $v_{i-1,j}$, and $v_{i-1,j-1}$ are necessary. At time (t – 1), since $v_{i,j-1}$ is the output value of the PE_i, it is obtained by the feedback loop. Also, $v_{i,j-1}$ is obtained by the output of PE_{i-1}. At time (t – 2), $v_{i-1,j-1}$ is obtained by PE_{i-1} and is retained by a register. The cascaded PEs shown in Fig. 6.42 computes the approximate matching graph in parallel. In other words, it performs the naive method shown in steps 3–6 of the above algorithm. Thus, its computation complexity becomes $O(m)$. More details of the circuit on FPGA have been demonstrated in [48].

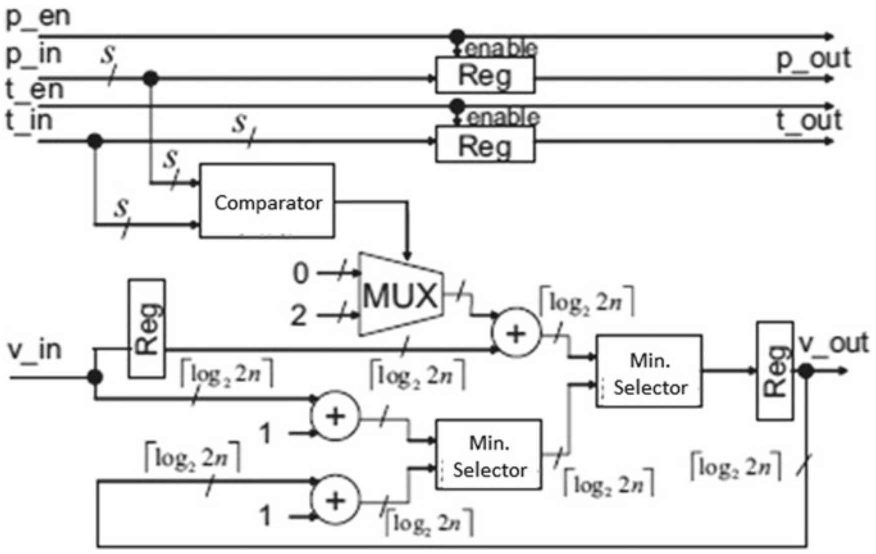


Fig. 6.42 A processing element (PE)

References

1. D.A. Patterson, J.L. Hennessy, *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface* (Morgan Kaufmann Publishers Inc., 2008)
2. H.S. Stone, *High-Performance Computer Architecture* (Addison-Wesley Publishing Company, 1990)
3. M.J. Flynn, Some computer organizations and their effectiveness. IEEE Trans. Comput. **21**(9), 948–960 (1972)
4. A. Peleg, U. Weiser, MMX technology extension to the intel architecture. IEEE Micro **16**(4), 42–50 (1996)
5. M. Hassaballah, S. Omran, Y.B. Mahdy, A review of SIMD multimedia extensions and their usage in scientific and engineering applications. Comput. J. **51**(6) 630–649 (2008)
6. A. Downton, D. Crookes, Parallel architectures for image processing. Electron. Commun. Eng. J. **10**(3), 139–151 (1998)
7. A.P. Reeves, Parallel computer architectures for image processing. Comput. Vis. Gr. Image Process. **25**(1), 68–88 (1984)
8. H.T. Kung, Why systolic architecture? IEEE Comput. **15**(1), 37–46 (1982)
9. J. MaCanny, *Systolic Array Processors* (Prentice Hall, 1989)
10. K.T. Johnson, A.R. Hurson, B. Shirazi, General-purpose systolic arrays. IEEE Comput. **26**(11), 20–31 (1993)
11. S.-Y. Kung, K.S. Arun, R.J. Gal-Ezer, D.V. Bhaskar Rao, Wavefront array processor: language, architecture, and applications. IEEE Trans. Comput. **C-31**(11), 1054–1066 (1982)
12. K. Sano, Y. Kono, FPGA-based connect6 solver with hardware-accelerated move refinement. Comput. Archit. News **40**(5), 4–9 (2012)
13. K. Sano, T. Iizuka, S. Yamamoto, Systolic architecture for computational fluid dynamics on FPGAs, in *Proceeding of IEEE Symposium on Field-Programmable Custom Computing Machines* (2007), pp. 107–116
14. K. Sano, W. Luzhou, Y. Hatsuda, T. Iizuka, S. Yamamoto, FPGA-array with bandwidth-reduction mechanism for scalable and power-efficient numerical simulations based on finite

difference methods. ACM Trans. Reconfig. Technol. Syst. **3**(4), Article No. 21, (2010), https://doi.org/10.1145/1862648.1862651
15. K. Sano, FPGA-based systolic computational-memory array for scalable stencil computations, in *High-Performance Computing Using FPGAs* (Springer, 2013), pp. 279–304
16. A.H. Veen, Dataflow machine architecture. ACM Comput. Surv. **18**(4), 365–396 (1986)
17. K. Hwang, F.A. Briggs, *Computer Architecture and Parallel Processing* (McGraw-Hill, Inc., 1984)
18. J.B. Dennis, Dataflow supercomputer. IEEE Comput. **13**(4), 48–56 (1980)
19. A.V. Kathail, A multiple processor dataflow machine that supports generalized procedures, in *Proceeding of ISCA81*, pp. 291–296, May 1981
20. G.L. Gaudiot, *Advanced Dataflow Computing* (Prentice Hall, 1991)
21. J.L. Perterson, *Petrinet Theory and the Modeling of Systems* (Prentice Hall, 1981)
22. S. Hauck, A. DeHon, *Reconfigurable Computing* (Morgan Kaufmann Publishers Inc., 2008)
23. R. Stephens, A survey of stream processing. Acta Inform. **34**(7), 491–541 (1997)
24. A. Das, W.J. Dally, P. Mattson, Compiling for stream processing, in *Proceeding International Conference on Parallel Architectures and Compilation Techniques* (2006), pp. 33–42
25. K. Sano, Y. Hatsuda, S. Yamamoto, Multi-FPGA accelerator for scalable stencil computation with constant memory-bandwidth. IEEE Trans. Parallel Distrib. Syst. **25**(3), 695–705 (2014)
26. K. Sano, R. Chiba, T. Ueno, H. Suzuki, R. Ito, S. Yamamoto, FPGA-based custom computing architecture for large-scale fluid simulation with building cube method. Comput. Archit. News **42**(4), 45–50 (2014)
27. J. von Neumann, The general and logical theory of automata, in *Cerebral Mechanisms in Behavior? The Hixon Symposium* ed. by L.A. Jeffress (Wiley, New York, 1951), pp. 1–31
28. S. Wolfram, Statistical mechanics of cellular automata. Rev. Mod. Phys. **55**(3), 601–644 (1983)
29. J. von Neumann, A.W. Burks, *Theory of Self Reproducing Automata* (University of Illinois Press, 1966)
30. A. Bandyopadhyay et al., Massively parallel computing on an organic molecular layer. Nat. Phys. **6**, 369–375 (2010)
31. D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Addison Wesley Longman Publishting Co., Inc., Redwood City, CA, USA, 1998)
32. K.E. Batcher et al., *Sorting Networks and Their Applications* (Spring Joint Computer Conference, AFIPS, 1968), pp. 307–314
33. D. Koch et al., FPGA sort, in *Proceedings of FPGA* (2011), pp. 45–54
34. J. Casper, K. Olukotun, Hardware acceleration of database operations, in *Proceedings of FPGA* (2014), pp. 151–160
35. T. Kohonen, *Content-Addressable Memories*, vol. 1, Springer Series in Information Sciences (Springer, Berlin Heidelberg, 1987)
36. H. Nakahara, T. Sasao, M. Matsuura, A regular expression matching circuit: decomposed non-deterministic realization with prefix sharing and multi-character transition. Microprocess. Microsyst. **36**(8), 644–664 (2012)
37. H. Nakahara, T. Sasao, M. Matsuura, H. Iwamoto, Y. Terao, A memory-based IPv6 lookup architecture using parallel index generation units. IEICE Trans. Inf. Syst. **E98-D**(2), 262–271 (2015)
38. H. Nakahara, T. Sasao, M. Matsuura, A virus scanning engine using an MPU and an IGU based on row-shift decomposition. IEICE Trans. Inf. Syst. **E96-D**(8), 1667–1675 (2013)
39. A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
40. L. Tan, T. Sherwood, A high throughput string matching architecture for intrusion detection and prevention, in *Proceedings of 32nd Int'l Symposium on Computer Architecture (ISCA 2005)* (2005), pp. 112–122
41. R. Baeza-Yates, G.H. Gonnet, A new approach to text searching. Commun. ACM **35**(10), 74–82 (1992)
42. R. Sidhu, V.K. Prasanna, Fast regular expression matching using FPGA, in *Proceedings of the 9th Annual IEEE Symposium on Field-programmable Custom Computing Machines (FCCM 2001)* (2001), pp. 227–238

43. C. Lin, C. Huang, C. Jiang, S. Chang, Optimization of regular expression pattern matching circuits on FPGA, in *Proceeding of the Conference on Design, Automation and Test in Europe (DATE 2006)* (2006), pp. 12–17
44. J. Bispo, I. Sourdis, J.M.P. Cardoso, S. Vassiliadis, Regular expression matching for reconfigurable packet inspection, in *Proceeding IEEE International Conference on Field Programmable Technology (FPT 2006)* (2006), pp. 119–126
45. T.F. Smith, M.S. Waterman, Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
46. S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the Amino-Acid sequence of two Proteins. J. Mol. Biol. **48**, 443–453 (1970)
47. L.J. Guibas, H.T. Kung, C.D. Thompson, Direct VLSI implementation of combinatorial algorithms, in *Proceedings of the Conference VLSI: Architecture, Design, Fabrication* (1979), pp. 509–525
48. Y. Yamaguchi, T. Maruyama, A. Konagaya, High speed homology search with FPGAs, in *Proceedings of Pacific Symposium on Biocomputing* (2002), pp. 271–282

Chapter 7
Programmable Logic Devices (PLDs) in Practical Applications

Tsutomu Maruyama, Yoshiki Yamaguchi and Yasunori Osana

Abstract Until the 2000s, FPGAs were mostly used for prototyping of ASIC chips or small-quantity products for limited application areas. Nowadays, FPGAs are used in various applications: high-performance computing, network processing, big data processing, genomics, and high-frequency trading. This chapter picks up the most exciting applications of FPGAs.

Keywords HPC · Network processing · Big data processing · Genomics
High-frequency trading

7.1 Introduction

7.1.1 History Summary of PLD

The concept of a programmable logic device (PLD) appeared in the late 1960s. The programmability of a chip called "reconfigurability" attracted a lot of engineers and academics and placed big hopes on PLDs. However, it was still early to start discussing the shift from application-specific integrated circuits (ASICs) to PLDs because PLDs did not reach the level that satisfies industrial demands such as size, computational speed, power consumption, reliability, and chip cost.

The quick growth of semiconductor industries [1, 2] has moved PLDs from inadequate to good in terms of size, computational speed, and power consumption. For example, the transistor count of the latest and largest PLDs reaches over 20 billion,

T. Maruyama · Y. Yamaguchi
University of Tsukuba, Tsukuba, Japan
e-mail: maruyama@darwin.esys.tsukuba.ac.jp

Y. Yamaguchi
e-mail: yoshiki@cs.tsukuba.ac.jp

Y. Osana (✉)
University of the Ryukyus, Ryukyus, Japan
e-mail: osana@eee.u-ryukyu.ac.jp

H. Amano (ed.), *Principles and Structures of FPGAs*,
https://doi.org/10.1007/978-981-13-0824-6_7

and therefore, PLD engineers can freely design their circuits on PLDs [3, 4]. Regarding operating frequency, high-speed PLDs, whose working frequency reaches 1.5 GHz, were introduced to the market [5]. The power consumption of PLDs has been reduced by bias voltage reduction, new technology node, thin-oxide gates, and so on.

Besides, some coarse-grained reconfigurable devices [6–11] and dynamic reconfiguration [12–14] have been proposed to contribute to the improvement of the energy performance ratio of PLDs [15, 16]. The increase of reliability enables PLDs to be used in finance [17], aerospace [18], and automobile systems [19]. The number and kinds of PLD products have been increasing, and the unit price has been dramatically decreased. Moreover, some nonvolatile technologies such as Spin-Transfer-Torque-Switching MOSFET (STS-MOSFET) [20] and atomic switch [21] begin to come into the practical use of PLDs. Thus, many industries start to reaffirm the increasing significance of PLDs furthermore.

7.1.2 PLD Market Size and Future Prospects

The estimate of the size of future PLD markets is a good measure to evaluate the growth potential of PLDs. Engineers and academics expect that PLDs will be widely accepted in various fields in the future.

The average annual growth rate of PLD markets is about 8.1%, and experts estimate that the market size will reach 10 billion USD in 2020 [22]. Indeed, PLDs are spreading in the Internet of Things (IoT), artificial intelligence (AI), big data, autonomous driving, and robot applications. If we consider not only PLDs (chips) but also PLD products, the market size including those PLD-related products will be beyond 100 billion USD. It is inevitable that these estimates enhance the significance of the research and development of PLDs to lead to future IT systems.

The remaining parts of this chapter examine the needs and features of FPGAs in modern applications. Section 7.2 presents high-performance computing (HPC) systems, including the HA-PACS/TCA. FPGAs have been used as communication acceleration chips to improve the effective performance dramatically. Section 7.3 deals with an FPGA-based network switch and discusses the advantages and limitations. Sections 7.4 and 7.5 present FPGA-based systems for search engine and genome informatics, respectively. Section 7.6 is dedicated to the acceleration for high-frequency trading in electronic trading (e-trading), and Sect. 7.7 shows how an FPGA-based system finds space debris efficiently.

7.2 PLDs/FPGAs in High-Performance Computing (HPC)

7.2.1 High-Performance Computing (HPC) Overview

The TOP500 and Green500 lists represent the ranking of the 500 fastest and energy-efficient supercomputers in the world since 1986 [23] and 2007 [24], respectively. Green500 requires not only the power-performance efficiency but also the absolute performance. Therefore, the list requires that the system must be ranked in TOP500. These rankings have been updated twice a year: in June and November. Figure 7.1 shows the TOP500 performance development between 1993 and 2015. In this figure, FLOP/s means floating-point operations per second.

The TOP500 list is often compared to Formula One car racing, and therefore, it conjures the image of a computational speed competition between supercomputers. Although it may remind us of the development of an acceleration chip such as GPU, a parallel computing system, called supercomputer, requires not only the computational performance but also how to connect computation units tightly. To be clear, the TOP500 is a three-legged race for computers; the total performance decreases rapidly when the performance of one system is slow. A supercomputer requires not only a high-speed accelerator but also a high-speed communication framework. For example, the K computer, manufactured by Fujitsu, adopts a highly dimensional communication topology [25]. The interconnect, called Torus fusion (Tofu), is a three-dimensional torus three-dimensional mesh interconnect. Using the Tofu interconnect, the K computer efficiently uses more than 88,000 processors simultaneously. The following section describes an HPC system with a tightly coupled architecture on FPGAs.

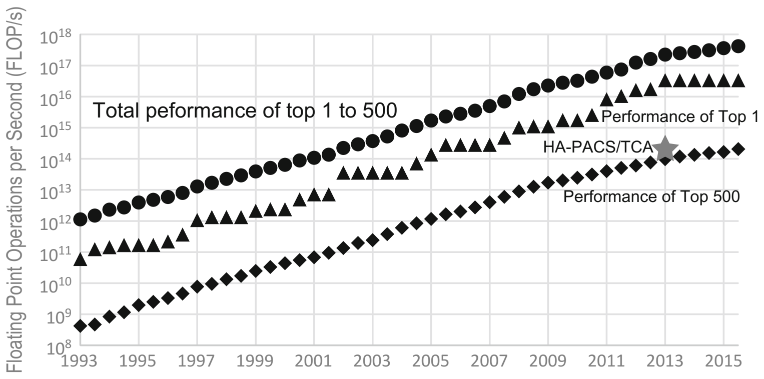


Fig. 7.1 TOP500 performance development (1993–2015) [23]

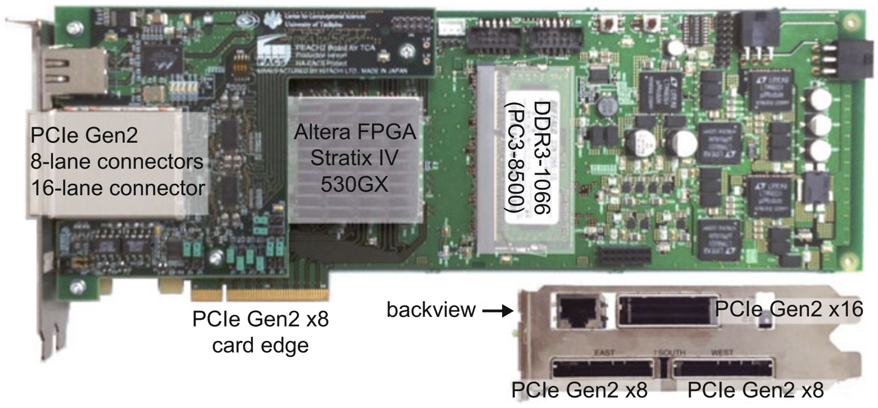


Fig. 7.2 PEACH2 board (Altera Stratix IV GX530)

7.2.2 HPC System with CPU, GPU, and FPGA

The HPC system series, of the Parallel Advanced system for Computational Sciences (PACS/PAX), is a supercomputer family manufactured by the University of Tsukuba. The PACS/PAX series has pursued a tightly coupled architecture between CPU and memory for more than 30 years. For example, the performance ratio of CPU, memory, network achieved 1:4:1 on the sixth generation called CP-PACS, which brought the fastest performance in the TOP500 in November 1996.

The eighth generation, HA-PACS, adopted GPUs as computational nodes [26], and it was the first attempt in this family. In the early GPU period, GPUs could not share the data efficiently. For example, each GPU had to move the shared data twice between main memory and local memory[1] for each computation. Moreover, the data transfer required protocol translation for the communication, and it made the latency larger. Thus, for efficient parallel GPU computing, HA-PACS tried to implement a high-speed communication architecture between GPUs.

To solve these problems, tightly coupled architecture (TCA) was proposed, based on PCI Express Adaptive and Reliable Link (PEARL) [30], and then PCI Express Adaptive Communication Hub (PEACH) [31] was developed and implemented on an FPGA. Figures 7.2 and 7.3 show the PEACH2 board and HA-PACS/TCA, respectively.

FPGAs have sufficiently brought performance improvement for HA-PACS. The actual and ideal performances of the HA-PACS base cluster were 421.6 TFlop/s and 778.128 TFlop/s, respectively. The actual and ideal performances of the HA-PACS/TCA were 277.1 TFlop/s and 364.3 TFlop/s, respectively. Thus, PEACH2 could improve the performance efficiency from 0.541 to 0.761. In November 2013,

[1]Current GPUs have proposed many solutions for parallel computing such as GPU direct communication (GPUDirect) [27], high-speed interconnect (NVLink) [28], and large cache/local memory such as adopting HBM [29].

Fig. 7.3 HA-PACS/TCA system

the HA-PACS/TCA took the 134th place at the TOP500 and the third place at the Green500 since it can achieve 3.52 GFlops/W. Also, in June 2014, the HA-PACS/TCA took the 164th place at the TOP500 and the third place at the Green500.

In the HA-PACS/TCA, PEACH2 provides the framework which enables a GPU to communicate with another GPU directly. To be more specific, PEACH2 extended the PCIe link to GPU communication.[2] Technically, PEACH2 is a router that extends the communication between PCIe root complex and endpoints to GPU communication, as shown in Fig. 7.4. Thus, as illustrated in Fig. 7.5, the GPU data transfer was simplified from “GPGPU mem $\rightarrow$ CPU mem $\rightarrow$ (InfiniBand/MPI[3]) $\rightarrow$ CPU mem $\rightarrow$ GPGPU mem” to “GPGPU mem $\rightarrow$ (PCIe/PEACH2) $\rightarrow$ GPGPU mem”. Moreover, PEACH2 could also achieve low-latency communication thanks to using only PCIe protocol.

The question that may arise now is: how efficient was the communication performance of PEACH2? PEACH2 has four PCIe Gen2 ports which has eight lanes.[4] The number of PCIe ports is limited by the used FPGA and not from PEACH/PEARL.[5] In the ping-pong latency, it was 2.0 µs when GPU-to-GPU direct communication was used in PEACH2. It is sufficiently small because the ping-pong latency of CPU-to-CPU communication is 1.8 µs. Thus, it has enough competitive power compared to MVAPICH2 v2.0-GDR whose latency with GDR and without GDR are 4.5 and 19 µs, respectively [35]. In the ping-pong bandwidth, the CPU-to-CPU performance

[2]Some switch products adopt the same concept, and Bonet Switch [32] is a PCIe Gen2 off-the-shelf product.

[3]Message Passing Interface (MPI) is a parallelization application programming interface (API) for distributed memory-based architectures. Please refer to [33].

[4]The link speed is equivalent to InfiniBand 4$\times$ QDR (32 [Gbit/s] = 4 $\times$ 8 [Gbit/s]).

[5]PEACH3 project is running [34], and it supports PCI Express Gen3.

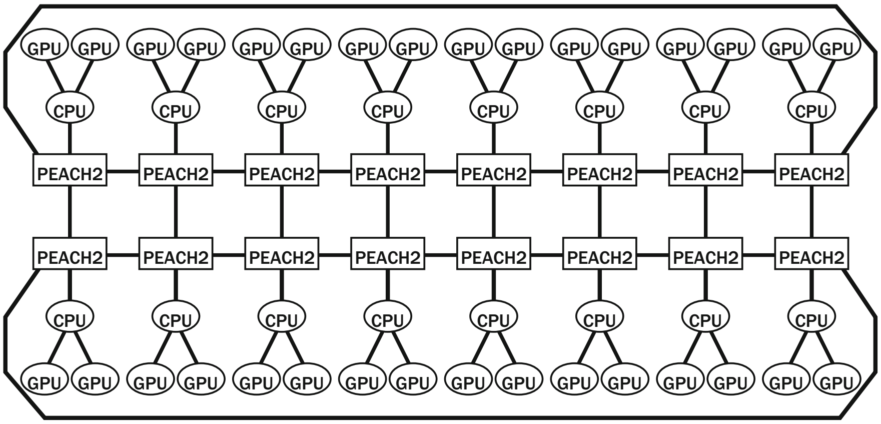
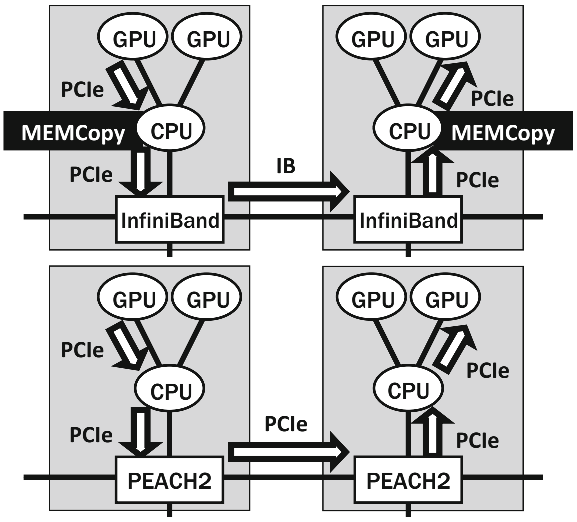


Fig. 7.4 HA-PACS/TCA (the half is not depicted)

Fig. 7.5 Network simplification



on PEACH2 is around 3.5 GBytes/s and the performance efficiency achieves about 0.95 of the ideal performance, 3.66 GBytes. Also, the GPU-to-GPU performance on PEACH2 is around 2.8 GBytes/s; however, the performance of MVAPICH2 v2.0-GDR becomes higher than PEACH2 if the payload size becomes larger than 512 KBytes. In the operation of HA-PACS/TCA, both PEACH2 and MVAPICH2 v2.0-GDR are used on a case-by-case basis [36]. To make a breakthrough in the HPC field, such technological trials are further required.

7.3 PLD/FPGA in a Network

7.3.1 *Role of Switches in a Network*

In a computer network, a network switch is a computer networking device that establishes the communication among computers and network switches. Specifically, it carries small data blocks called packets that include the source, destination, data. These network switches can be classified into two broad categories: layer-2 switches and layer-3 switches from the viewpoint of functionality.

Figure 7.6 shows an example where a packet is transported from PC0 to PCz by a layer-2 switch. In this figure, a packet sent by PC0 is transported to PCz according to process ① to ⑥. The network bandwidth increases when switches that construct a network can use MAC address tables efficiently. For example, multiple packets can be transported by switches simultaneously if all destinations and all sources were registered on tables.

The question is: how can we further increase the network bandwidth? Embedded CPUs can be replaced by high-end CPUs for the packet control. However, the processing overhead is more than 1 μs [37], and thus not software implementation, but hardware implementation is required. PLD/FPGA is one good candidate for the reduction of the network latency. An efficient hardware implementation can expect further reduction of the switching latency [38].

7.3.2 *FPGA Performance Upgrade as a Network Chip*

FPGAs have introduced high-speed transceivers for high-speed serial interfaces since around 2000. Table 7.1 shows the FPGA performance upgrade from the standpoint of a network chip. In Table 7.1, the speed per transceiver has increased by about ten times in the past decade, reaching 32.75 Gbit/s in 2015. However, it is hard

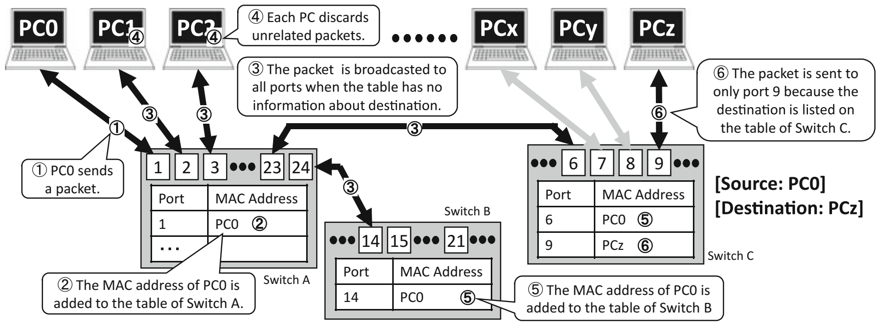

PC0
PC1
PC2
④
④ Each PC discards unrelated packets.
PCx
PCy
PCz
③ The packet is broadcasted to all ports when the table has no information about destination.
⑥ The packet is sent to only port 9 because the destination is listed on the table of Switch C.
①
③
③
③
③
⑥
① PC0 sends a packet.
1 2 3 ●●● 23 24
Port MAC Address
1 PC0 ②
...
Switch A
② The MAC address of PC0 is added to the table of Switch A.
Switch B
●●● 14 15 ●●● 21 ●●●
Port MAC Address
14 PC0 ⑤
⑤ The MAC address of PC0 is added to the table of Switch B
●●● 6 7 8 9 ●●●
Port MAC Address
6 PC0 ⑤
9 PCz ⑥
Switch C
[Source: PC0]
[Destination: PCz]

Fig. 7.6 L2 packet transportation by using MAC address

Table 7.1 Performance upgrade of high-speed transceivers on Xilinx FPGAs

Year	Family	Transceiver (maximal I/O speed, # of transceivers)
2002	Virtex 2 Pro	Rocket IO (3.125 Gbit/s, ×24)
2004	Virtex 4	Rocket IO MGT (6.5 Gbit/s, ×24)
2006	Virtex 5	Rocket IO GTX (6.5 Gbit/s, ×48)
2009	Virtex 6	GTX (6.6 Gbit/s, ×48) + GTH (11.18 Gbit/s, ×24)
2010	Virtex 7	GTH (13.1 Gbit/s, ×72) + GTZ (28.05 Gbit/s, ×16)
2013	UltraScale	GTH (16.3 Gbit/s, ×60) + GTY (30.5Gbit/s, ×60)
2015	UltraScale+	GTY (32.75 Gbit/s, ×128)

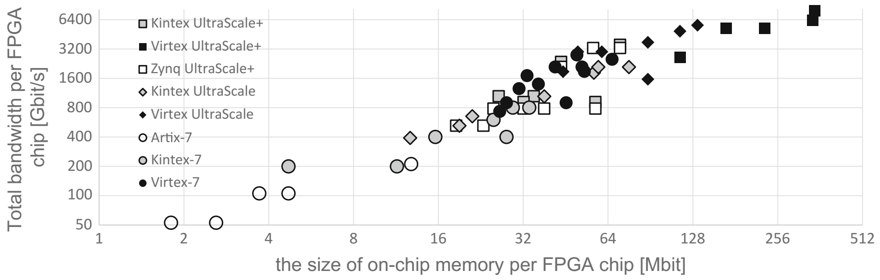
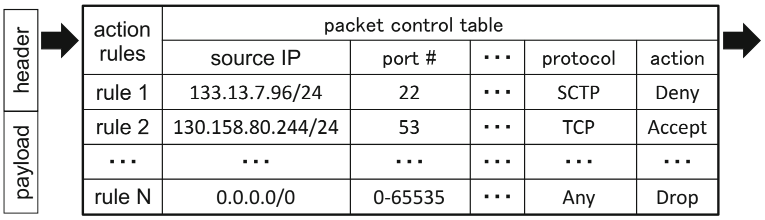


Fig. 7.7 Relationship between total bandwidth and the size of on-chip memory [39, 40]

to keep the high signal integrity of high-speed transceivers at a higher frequency. Therefore, it is difficult to increase the transceiver speed at the same pace as before in the next decade. Toward high-speed and low-latency switching, the number of transceivers per FPGA chip has to increase, and efficient hardware IP cores should also be supported.

Figure 7.7 shows the relationship between the total bandwidth and the size of on-chip memory per FPGA, where the memory size of UltraScale+ is the sum of BRAM and UltraRAM. The size of on-chip memory affects the performance of not only the bandwidth but also the packet filtering/classification, and it is positively correlated with the total bandwidth. Thus, it is not an exaggeration to say that FPGA is a network chip because it has been equipped with the latest transceivers and sufficient on-chip memory in every generation. There are many network products with FPGAs such as Exablaze [38], Arista [41], Cisco [42], Mellanox [43], Simplex [44].



action rules	packet control table				
	source IP	port #	···	protocol	action
rule 1	133.13.7.96/24	22	···	SCTP	Deny
rule 2	130.158.80.244/24	53	···	TCP	Accept
···	···	···	···	···	···
rule N	0.0.0.0/0	0-65535	···	Any	Drop

Fig. 7.8 Packet classification control

7.3.3 PLD/FPGA and Software-Defined Networking (SDN)

First, a network flow is composed of network packets that include a packet header and a payload. A packet header contains the information of communication type such as destination, used network port, and protocol. If a network packet were a postal package, the packet header would express the address and the postal type of the package. Thus, by using network switches, we can control the network flow.

During the packet transfer, network switches autonomously collect routing information based on the information of packets already carried. The collected data can be summarized in Fig. 7.6, and a cache table can be used to reduce the control processes for the same header that has been already sent. However, it is hard to realize network virtualization, dynamic rule change, and network simulation for secure communication based on the current framework. Thus, a new framework called software-defined networking (SDN)[6] has been proposed [45].

SDN has been proposed since the 1990s [46] and widely accepted since OpenFlow [47] protocol were established by Open Networking Foundation [48]. FPGA vendors and third-party companies are also interested in its hardware implementation as is the case for SDNet by Xilinx [49] and SDN CodeChip by Arrive [50]. There are a lot of research of SDN on FPGAs [51–57]. The packet classification problem, as shown in Fig. 7.8, is one of the main issues.

A network switch receives a packet and transfers it to an appropriate port, though it might be discarded by the rule stored on the switch [58]. Thus, the packet classification problem is how efficient a switch can complete these controls. To realize high-speed and highly reliable networks, we need to consider latency, bandwidth, and rule complexity. FPGAs are expected to overcome these problems by direct hardware computing and reconfigurability.

[6]Software-defined networking (SDN) is also known as software-defined infrastructure (SDI) or software-defined data center (SDDC).

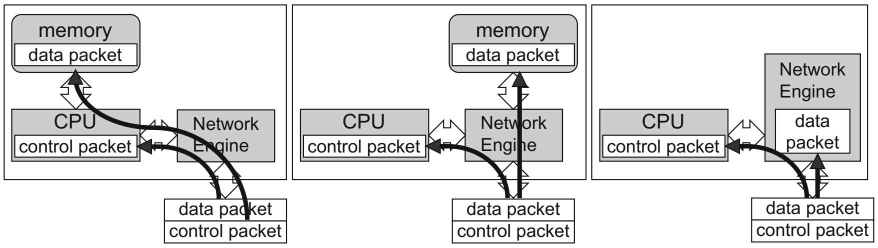


Fig. 7.9 Difference in packet control by system configuration

7.3.4 Packet Classification and Its System Configuration

This section gives an overview of system configuration with FPGA. A network switch must store received packets on a storage such as on-chip or off-chip memory during the analysis of a packet. Figure 7.9 shows the difference between off-chip and on-chip memories. The current off-the-shelf system is commonly composed of memory chips, a CPU, and a network engine.

Figure 7.9 follows this current situation, but SoC-FPGA may soon change it.

The left portion of Fig. 7.9 shows a CPU-based implementation where the CPU analyzes and classifies the received packets. All the packets are stored in the memory, at least once. Here, the computation based on network protocol difference (such as Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Internet Control Message Protocol (ICMP), and Address Resolution Protocol (ARP)) is a bit complicated. Thus, the computation on a CPU can provide the answer and the feasibility of CPU implementation can be evaluated by how it solves the memory bottleneck [59]. The center and right sides of Fig. 7.9 show the system configuration where a network engine (NE) is the main computation unit of packet classification. In these cases, the main unit is implemented on ASICs or FPGAs. It enhances further the performance improvement, but the system cost also increases.

Table 7.2 shows a particular flow of computing a packet classification. In the left side of Fig. 7.9, a network packet comes to a network port, and it passes through the NE ($L_{cpu}0$), CPU ($L_{cpu}1$), and memory ($L_{cpu}2$). CPU reads the packet from the memory ($L_{cpu}3$) and identifies the packet header. Then, the packet goes back to the NE ($L_{cpu}4$) and is forwarded to the network through the appropriate network port ($L_{cpu}5$).

If the latency of a packet transfer between two chips is 500 ns, the latency of the left, center, and right sides in Fig. 7.9 are 3, 2, and 1 μs, respectively. The network packets go through many switches, and the latency should be minimized for efficient communication. The latency of each switch should be within 1 μs in high-speed networks. Thus, the type of current off-the-shelf network switches is the center and right architectures in Fig. 7.9.

Intel has announced a new FPGA that integrates a Xeon E5-2600v2 (Ivy Bridge) and a Stratix V FPGA [60]. Furthermore, FPGAs start to support high-speed memo-

Table 7.2 Comparison of latency in Fig. 7.9

	Left	Center	Right
Network → network engine	$L_{cpu}0$	$C_{cpu/ne}0$	$R_{cpu/ne}0$
Network engine → CPU	$L_{cpu}1$	$C_{cpu}1$	$R_{cpu}1$
CPU → memory	$L_{cpu}2$	N/A	N/A
Memory → CPU	$L_{cpu}3$	N/A	N/A
CPU → network engine	$L_{cpu}4$	N/A	N/A
Network engine → memory	N/A	$C_{ne}1$	N/A (on-chip)
Memory → network engine	N/A	$C_{ne}2$	N/A (on-chip)
Network engine → network	$L_{cpu}5$	$C_{ne}3$	$R_{ne}1$

ries [61, 62] such as Hybrid Memory Cube (HMC) [63] and High Bandwidth Memory (HBM) [29]. This movement accelerates the shift of network switches to the right side of Fig. 7.9.

7.3.5 Content-Addressable Memory (CAM) and FPGA

The packet classification on FPGAs has applied some approaches, such as hash table implementation [64] and n-branch tree search [65]. This section introduces another implementation, called content-addressable memory (CAM) [66, 67], which is highly compatible with FPGAs.[7]

The behavior of CAM differs from that of standard memories. For example, CAM returns address(es) but not data onto the memory. Figure 7.10 illustrates such a difference. In packet classification, a network switch compares a destination IP with the IP list on the CAM, and then the appropriate port is selected by using the returned address and some rule sets stored in standard memories.

In general, IP address search is little to use an exact match search; instead, a part of the target address is used, such as subnet. Thus, CAM can be roughly classified as binary CAM (B-CAM) and ternary CAM (T-CAM). As shown in Fig. 7.3, T-CAM can treat don't-care ("x") in the comparison. Here, T-CAM may propose multiple address candidates due to the presence of the "x". In general, T-CAM outputs the

[7]The history of CAM can be traced back to the 1950s when memory appeared. Since it is a fundamental concept of memory implementation, it has always been discussed in FPGA companies [68, 69] and academics [70]. Besides, the latest CAM chip [71] is attractive to implement network applications.

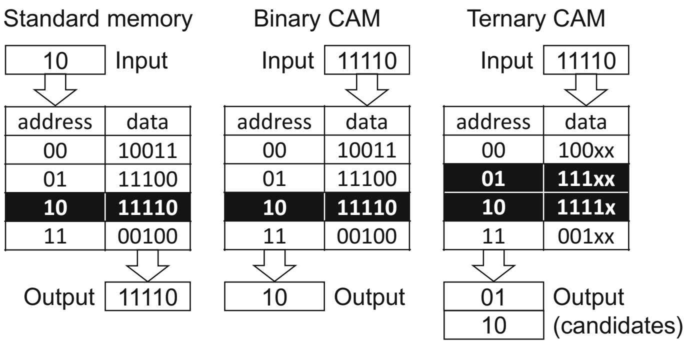


Fig. 7.10 Difference between standard memory and CAM ("x" means don't-care.)

lowest address among these candidates. For instance, in Fig. 7.10, T-CAM has two candidates "00", and the lower address "01" which is output. Efficient algorithms of data store and read will be a challenging research topic if the lowest address is not appropriate for a given application. However, the question that remains is: how fast is a CAM in search problem?

In a network search engine, R8A20686BG-G by Renesas [72, 73], that includes T-CAM, the performance achieves two billion searches per second. The theoretical performance is 300 Gbit/s because it has two Interlaken LA ports. One search request runs a single CAM access, and the search time is constant regardless of the bit width of the search word and the number of registered entries. The latest Xilinx FPGA has nine Interlaken IP cores, and some studies have proposed a proper implementation for CAM [64, 65]. Although CAMs require a larger number of transistors compared to a standard memory, as discussed in [74], the advantage of CAM seems to outweigh the drawbacks soon. The implementation with FPGA and CAM will be one option to implement network processing.

7.4 Big Data: Web Search

Web search engines have enabled easy search for basic and essential knowledge, by allowing access to numerous documents distributed in different Web sites in all over the world. Once a search keyword is entered, huge databases are scanned to find the Web pages containing this keyword. Then, the Web pages are scored and sorted in terms of their relevance with the keyword, before the list is displayed on the user screen.

Originally, search engines were built on PC-based server clusters. But now, they are facing the "power wall" that limits the scale of data centers. So, to avoid stopping the continuous evolution of Web search engines, breakthrough in power efficiency is an urgent requirement. This section briefly reviews Microsoft's Catapult FPGA accelerator in Bing search engine.

7.4.1 Overview of Bing Search

Queries (or search terms) from user is received by the front-end of the Bing engine. The front-end searches the query cache and then transfers the query to another server called Top Level Aggregator (TLA) on cache miss. Then the TLA transfers the query to 20–40 Mid Level Aggregator (MLA) servers again. Each server rack has one MLA server, then MLA queries 48 Index File Managers (IFM) nodes in the same rack. Each IFM performs rank calculation of the query on about 10,000–20,000 documents, and the results are aggregated by MLAs and TLA on their way back to the front-end [75].

The IFM first picks up documents containing all words in the query and then chooses the final candidates. During this process, a vector stream called "Hit Vector" represents the locations query terms in the generated document. After Hit Vectors are generated, the documents' scores for ranking are calculated based on the Hit Vectors. This ranking process is computation intensive; thus, an FPGA-based accelerator can be introduced.

7.4.2 Ranking Engine Acceleration

The first stage of ranking is feature extraction (FE), to calculate some "feature" scores from the Hit Vector. In the FPGA accelerator, custom scoring FSMs are implemented for each feature and all FSMs work on a given Hit Vector in parallel (as stated in [76], 43 feature extraction FSMs extract 4,484 features). Because multiple pipelines work on a Hit Vector, it is called a "Multiple Instruction, Single Data (MISD)" architecture [76]. The FE process takes 600 μs in software, but the FPGA implementation requires only 4 μs.

The next stage is to calculate the "hybrid feature" scores, using feature scores from FE stage. This process is called Free-Form Expression (FFE). As the name implies, this process consists of various arithmetic operations including floating-point arithmetic. Because this stage requires high flexibility, custom-designed soft processor cores are implemented. The soft processor cores form a many-core architecture, with 60 cores, on a Stratix V FPGA.

Hybrid features from the FFE stage are sent to the Machine-Learning Scoring (MLS) to calculate the final, single floating-point score of the document. The details of this stage are not open, but the algorithm seems to be frequently updated. Such frequent updates are a good use of FPGA's flexibility.

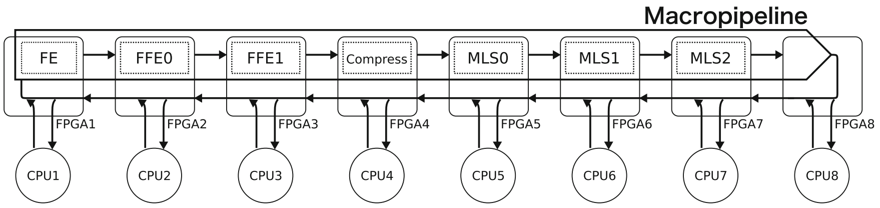


Fig. 7.11 Macropipeline structure of Catapult

7.4.3 Organization of Catapult Accelerator

Microsoft developed the Catapult FPGA board as the ranking accelerator described above. The board has an Altera's Stratix V FPGA, an 8 GB DDR3 SDRAM SO-DIMM, and PCI Express interface toward the server host. In addition to PCI Express interface to the host, Catapult board has 10 Gbps bidirectional serial links to form 6×8 torus network. With this network, a "Macropipeline" spanning multiple FPGAs is formed.

As illustrated in Fig. 7.11, eight FPGA boards construct a ranking Macropipeline. The FE stage is implemented on the first FPGA, and the FFE and MLS stages are on the FPGAs located at the downstream. Because these eight FPGAs are connected to each different IFM nodes, the FPGA-to-FPGA link transports not only datastream in Macropipeline, but also datastream between IFM–FE or MLS–IFM.

In Fig. 7.11, the datastream in Macropipeline flows from left to right. On the other hand, IFM–FE and MLS–IFM datastreams flow from right to left. So, the traffic always goes clockwise; thus, the bidirectional link bandwidth is fully utilized in all segments.

Furthermore, the 6×8 torus can contain other accelerator macropipelines for several different applications, such as computer vision [77]. These macropipelines are also available in IFM through the FPGA-to-FPGA links.

7.5 Genomics: Assembly and Mapping of Short Reads

Any organism has its own genome, composed by only four DNA molecules denoted by: A (adenine), T (thymine), C (cytosine), and G (guanine). Different species have much different in genome size: 1.5×10^6 base pairs for several microorganisms, 3.2×10^9 for human, or 17×10^9 for bread wheat.

Because a genomic DNA sequence consists of only four molecules, various string processing algorithms can be applied for genome analysis. FPGAs are suitable for the analysis acceleration by using hardware implementation of custom state machine. Especially for genomic DNA sequence, the FPGA implementation is advantageous when compared to the CPU implementation because the component can be coded in a smaller number of bits than 8. Thus, various FPGA-based accelerators for genomics have been implemented since Splash 2 [78] in the early 1990s.

For a long time, genomes had been sequenced using DNA sequencers based on the "Sangar sequencing method", invented in 70s. With the emergence of next-generation sequencing (NGS) technology in the early 2000s, the throughput of genome sequencing has dramatically improved. NGS reads short fragments of DNAs in a massively parallel way (the result of sequencing of each fragment is usually called just a "read").

To sequence a genomic DNA, the DNA chain is randomly fragmented by mechanically using supersonic wave, or chemically. Most of the randomly fragmented DNA chains usually have overlaps to some other fragments in both ends. So, the whole original DNA chain can be "assembled" by coalescing the fragmented DNA chains together. However, this process requires a vast string matching workloads to find the overlaps between two or more reads. Especially with NGS, the assembled workload is much heavier than with the Sangar sequencer since NGS reads a massive number of very short reads.

NGS is also useful in personal genome sequencing, a promising technology in tailor-made medical treatments. In personal genome sequencing, sequenced fragments are "mapped" on already known human genome (or, "reference") sequence. Everyone has a slightly different genome sequence, and sometimes the small differences attract medical interests. Although the read mapping is a lesser computation-intensive task than the genome assembly, it still is a large workload. Hereafter, FPGA implementation examples for short read assembly and read mapping to reference genome are presented.

7.5.1 De Novo Genome Assembly from Short Reads

De novo sequencing or assembly consists of assembling the whole genome without any previously known sequences, but just using short reads. With the emergence of NGS technology, several de novo assembler softwares specialized in using short reads have appeared. Every short read assembler requires a long computation time to find overlaps between the reads.

FAssem [79] is an FPGA-accelerated implementation of Velvet [80], a well-known short read assembler. Velvet finds read overlaps first and then generates De Bruijn graph to obtain the final assembly. FAssem does the first stage on an FPGA and then executes the second stage using the Velvet's original software implementation. With the FPGA acceleration on a Xilinx's Virtex-6 LX130T, 2.2x to 8.4x speedup is achieved compared to the software-only version on a 2.6 GHz Core 2 Duo E4700.

7.5.2 Short Read Mapping on a Reference Genome

We are all *Homo sapiens*, but everyone (except identical twins) have a slightly different genome sequence. These personal variations are attracting medical interests to evaluate personal risks in specific diseases, or to make tailor-made medicine possible. For example, many single nucleotide polymorphisms (SNPs) are already known as "markers" of personal genetic variations.

To obtain a personal genome sequence, the DNA reads are mapped on the reference human genome sequence. Because this human genome sequence is already available, it is not necessary to assemble the whole genome sequence again, but just mapping the reads on the already known reference sequence is enough. The reads will be different from the reference sequence at several locations: This is the personal genome variation.

Basically, mapping is much faster than assembling, and faster is always better because this technology is expected to be used in the medical domain. The mapping process is not simple since there will be mismatches in molecules, insertion or deletion of sequences between the sequencer reads and reference sequence.

Several read mapping tools have been developed and published, such as Bowtie [81] and BWA [82]. The work in [83] is an FPGA-accelerated implementation of BWT, using Burrows–Wheeler transform. The FPGA implementation on Altera's Stratix V board achieved 21.8x speedup compared to a 4-core microprocessor. Both FAssem- and FPGA-accelerated BWA have an array of many string matching modules, and parallel string matching on these modules contributes to their high throughput.

7.6 High-Frequency Trading (HFT)

High-frequency trading is a prominent part of electronic trading (e-Trading) and requires low-latency computing. Thus, from the standpoint of low-latency computing, this section introduces stock trading and then explains how FPGAs are used there.

7.6.1 Stock Trading Overview

The earliest roots of stock markets can be traced back to the sixteenth century.[8] The stock has been introduced to reduce the risk of funders and to gather capitals efficiently and widely. Figure 7.12 shows how to efficiently raise capitals and distribute the benefits based on stocks. In that era, stocks have not been traded so frequently.

[8]The company's name is Vereenigde Oost-Indische Compagnie, called the United East India Company in English, and was founded in 1602.

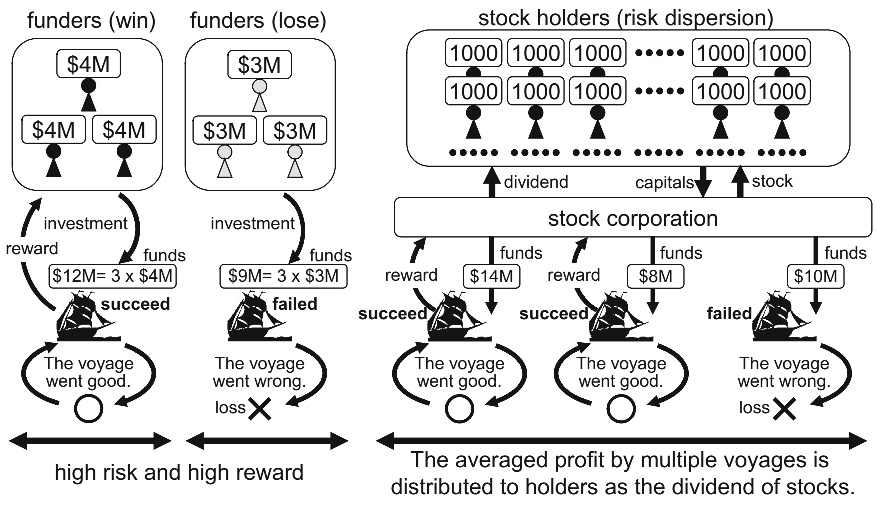


Fig. 7.12 Risk dispersion by stocks and a flow of capitals, funds, and rewards

Table 7.3 Rough classification of stock tradings

		Time priority (first-in first-out trading)	
		Yes	No
Price priority	Yes	Continuous limit order during a trading session	Limit order before/after a trading session
	No	Market order during a trading session	N/A

The holders needed to have their stocks for several months or more because their purpose was to obtain the dividend brought by each voyage.

As the number and kinds of stocks have been increasing, the holders began to choose a better stock carefully. This has increased the stock trading, and some holders start to focus on the margin of stock transactions. Hence, stock markets and various trading methods have been generated based on a wide range of requests. Although trading methods are becoming more complex, they inherit the same core principles, “price” and “time”.

Table 7.3 shows a rough classification of stock tradings from the viewpoint of price and time. This table does not consider all trading methods using other conditions, such as the conditional order.

In Table 7.3, a *limit order* is an order to buy or sell a stock. When buying, each order can only be executed at the limit price or lower. When selling, each order can only be executed at the limit price or higher. A *market order* is an order to be executed immediately at the current market price. It is enabled while a trading

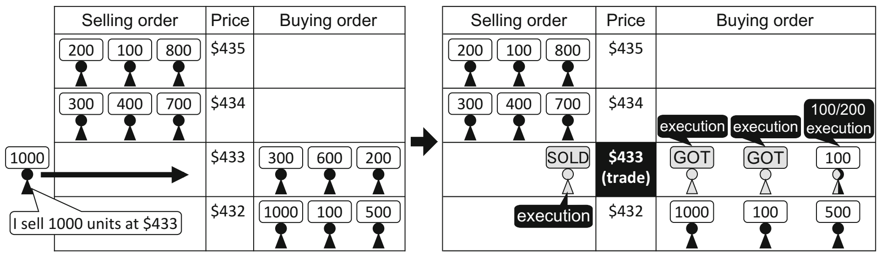


Fig. 7.13 Double auction: both buyers and sellers bid a price that each wants

session is working because the price is decided by the market. During the continuous trading session, any orders can continuously be executed under price and time priority conditions.

7.6.2 Continuous Limit Order to High-Frequency Trading (HFT)

Fig. 7.13 illustrates the overview of a stock trading under limit orders. Both buyers and sellers order prices that each wants, and then the information is published. This trading is one of the open-outcry double auctions.

Nothing is done when there is no overlap between selling orders and buying orders. When an overlap happens, the deal starts. In Fig. 7.13, an overlap happens at the price of $433, and transactions are made based on the first-in-first-out queue of the $433 buying order.

In stock trading, a person who has a higher priority will obtain a larger margin in a transaction. Thus, the most important thing is how to get the highest priority. From the perspective of price, a good stock value is expected in the future. Stock trading can apply value prediction algorithms, and artificial intelligence approaches have received a lot of attention recently. From the perspective of time, it is important to shorten the time for sending an order. This requires to accelerate the recognition of the current market situation and to send the trading request as quickly as possible. Advances in information and communications technology (ICT) have enabled to dramatically reduce the trading time since the 1990s. For example, the order response time in the London Stock Exchange, Tokyo Stock Exchange, and Singapore Stock Exchange are around 690 μs [84, 85], 500 μs [17], and 90 μs [86], respectively.[9, 10] Consequently, high-frequency trading (HFT) has emerged as a new market to obtain the significant margin within a short duration. Table 7.4 shows an example of the speed and volume of HFT.

[9] μs (microsecond): one millionth of a second.

[10] In 2017, the New York Stock Exchange has announced the introduction of a delay mechanism, and the speed bump is 350 μs [87]. The concept has been introduced in [88].

Table 7.4 Average time per order and the number of price changes (June 26, 2008)

	Average time (ms/order)	Price changes in one day
Citigroup	2.238	12,499
General electric	4.244	7862
General motors	7.843	9016

This table was made by reference to Table 1 in [89]

7.6.3 HFT Speed Bump and the Value of Latency

The latency in computer systems is a top priority issue in the battle against the opportunity loss in ICT world. Viraf Reporter [90] reports that at least one percent of the reward could be lost if a trading order is 5 ms behind the competition. Also, Google loses 20% of its traffic if a search page load requires additional 500 ms, and Amazon loses 1% of their sales if an extra 100 ms is required [91].

However, the fact remains that the core process of HFT is straightforward, and therefore, the speed of a software implementation such as C language may be sufficient. In [92], an HFT approach, called tradeHFT, was implemented in C language, and the execution time was around 750 ns.[11] If the current situation continues, a lot of hardware-based systems will go back to a CPU-based system [93].

On August 1, 2012, Knight Capital Americas LLC experienced a significant error and lost over $460 million from unwanted positions [94]. Stock exchanges start to re-evaluate HFT after this accident, and the New York Stock Exchange decided to implement a delay mechanism in 2017 [87]. The speed bump directly reduces ultra-high-frequency trading, but complex and middle-to-high-frequency trading will increase. Consequently, future HFT will employ new algorithms customized to the limitations of each market. At that time, for sophisticated computation, external accelerators such as GPUs and FPGAs can be adopted since the current speed bump, 350 μs, in NYSE is sufficiently long.

The semiconductor industry cycle between standardization and customization is known as the Makimoto's Wave [95]. Also in HFT, the era of FPGAs will come again soon after the current CPU era.

7.6.4 HFT System Integration on an FPGA

Figure 7.14 shows the evolution of FPGA systems.

All the computation are processed on a CPU of a standard system, as shown in Fig. 7.14. In this implementation, high-level programming languages such as C language are used for implementing HFT approaches. If lightweight and simple processes with less data transfer are treated, this configuration can be sufficient [92].

[11] ns (nanosecond): one billionth of a second.

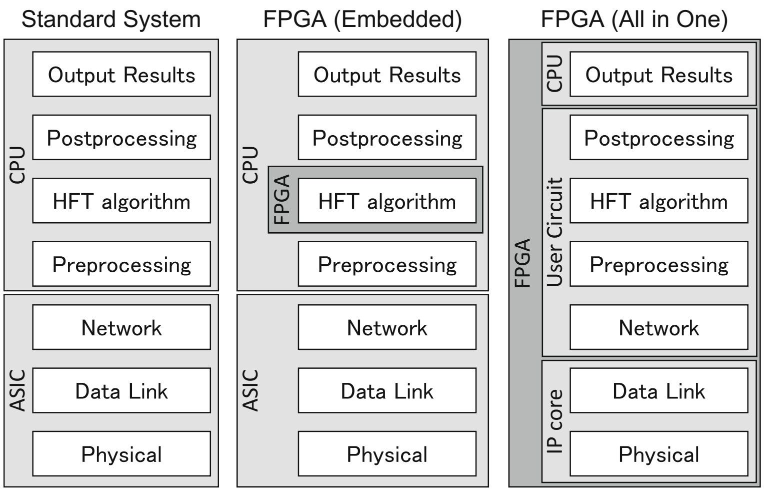


Fig. 7.14 Evolution of FPGA systems

An acceleration chip will be required when there is high computation for a targeted approach. Then, FPGA can be a good candidate of low-latency streaming computing, such as HFT. This is because the latency is smaller compared to other systems, including GPUs.[12] However, the communication latency cannot be sufficiently reduced in this implementation. In fact, one-way latencies of a standard system, a system with embedded FPGAs, and an FPGA-based all-in-one system are around 12.8 µs, 4.1 µs, and 2.6 µs, respectively [96]. Furthermore, newer FPGAs with 10GbE ports can reduce the latency within 1 µs [97, 98]. Eventually, all components will be integrated on an FPGA as done in [99]. The latest FPGAs have powerful processors. For example, both Xilinx Zynq Ultrascale+ [4] and Intel Stratix10 FPGAs [3] have a quad-core ARM processor. Intel has also integrated an FPGA fabric into an Intel Xeon package [60]. In addition, FPGA vendors and third parties start to support high-level synthesis such as [100–102]. Finally, the Arrowhead Systems Inc. in the Tokyo Stock Exchange applies FPGAs, called SimplexBLAST FPGA [44], and also network vendors like [103] are interested in the acceleration by FPGAs.

7.7 Image Processing: Space Debris Detection

Since the first satellite has been launched in 1955, a large number of rockets sent satellites and space probes around the earth. As a result, thousands of artificial objects are on their orbit. Many of them fulfilled their role or are already broken. The velocity

[12]A bottleneck of GPU is the communication between a host PC and a GPU board through PCIe bus. See Sect. 7.2.

of objects are as high as 8 km/s in low orbit; thus, such objects have large kinetic energy. Because even a very small fragment has enough kinetic energy, collision with satellites or spacecraft can cause a critical accident. Moreover, collisions make more fragments in the orbit.

This kind of fragments are called "space debris," and they are observed and tracked by the space exploration agencies of many countries since they pose a large threat to the operations of satellites, spacecraft, and space stations. For example, North American Aerospace Defense Command (NORAD) is always tracking over 8000 debris in low orbit. There is no efficient way to collect space debris safely for now. So, high precision detection, tracking, and collision avoidance are required for the safe operation in outer space.

This section introduces an FPGA-accelerated approach, developed by Japan Aerospace eXploration Agency (JAXA), to detect space debris with high-resolution optical telescopes.

7.7.1 The Method Overview

JAXA developed this system with optical telescopes in Nyugasa-yama Optical Observatory in Ina city, Nagano. The telescopes have 2K2K CCD and 4K4K CCD cameras. These cameras serve to take a series of pictures of the sky with a constant interval. The trajectory of "known" objects such as known stars, satellites, or the International Space Station (ISS) can be calculated and mapped onto the pictures. After mapping known objects, "unknown" objects remain on the picture. These objects may be space debris or newly found asteroids.

However, because unknown objects have unknown trajectories, it is difficult to detect and track them. To make the problem worse, they are often dark so they are hard to find. JAXA resolved these problems by an image processing algorithm called "stacking method" [104].

This method detects dark, unknown objects from noisy telescopic images by stacking them with a constant offset, then taking the median value of each pixel. When the images are stacked with no offset, as shown in Fig. 7.15a, the trajectory of known objects can be traced. However, we have no knowledge on how to trace other unknown objects.

To detect unknown objects, images are stacked with various offsets in the X and Y directions, as depicted in the example of Fig. 7.15b, c. Because the direction and speed of debris or asteroids are constant, they appear as a bright dots when the stacking offset matches with the object's speed.

However, performing the stacking for all possible directions and speeds is a heavy task. For example, to find an object that moves within 256×256 pixels between two images, $256 \times 256 = 65{,}536$ stacking patterns must be investigated. Fortunately, the number of stacking patterns remains constant even with more telescopic images, because the direction and speed of an object are basically constant.

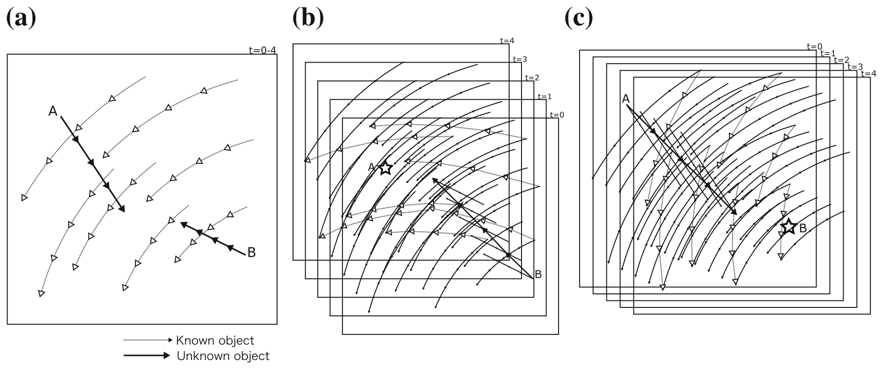


Fig. 7.15 Detection of space debris with stacking method **a** Stacking with no offset **b** Stacking with offset, to match with trajectory of A **c** Stacking with offset, to match with trajectory of B

According to JAXA's report in 2001, analyzing 16-bit grayscale images, by performing the stacking in 65,536 ways, requires 280 h on a CPU. 280 h of image processing is not realistic for application use, so an accelerator has been developed using an FPGA board.

7.7.2 FPGA Acceleration

To easily handle the image data on an FPGA, the 16-bit grayscale image is binarized to make the dataset compact. In general, binarization makes S/N ratio worse; but, the result of this method does not degrade. This is because the stacking effectively reduces the noise. Image binarization also enables the replacement of the gray-level median calculation with bright dot counting. This makes the FPGA implementation simple and fast, but controlling the threshold values is crucial for both image binarization (grayscale to bright/dark) and object detection (counter value of object/background).

The FPGA version of this method is implemented on Nallatech H101-PCMXM board with Xilinx Virtex-4 LX100 FPGA and Nallatech's C-based high-level synthesis tool. As results, 1200x speedup is achieved when compared to the original software, and the method can be used in practice. This speed constitutes a large contribution to high precision and high sensitivity detection of space debris with optical telescopes.

References

1. G.E. Moore, Cramming more components onto integrated circuits. Proc. IEEE **86**(1), 82–85 (1998)
2. Xilinx Staff, Celebrating 20 years of innovation. Xcell J. **48**, 14–16 (2004)
3. Altera Corp., http://www.altera.com
4. Xilinx Inc., http://www.xilinx.com

5. Achronix Semiconductor Corp., Speedster 22i HD FPGA Platform, 2.7 edn. (June 2014). Product Brief, http://www.achronix.com/
6. K. Compton, S. Hauck, Totem: custom reconfigurable array generation, in *IEEE Symposium on Field-Programmable Custom Computing Machines* (March 2001), pp. 111–119
7. D.C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, C. Ebeling, Architecture design of reconfigurable pipelined datapaths, in *20th Anniversary Conference on Advanced Research in VLSI* (March 1999), pp. 23–40
8. P. Heysters, G. Smit, E. Molenkamp, A flexible and energy-efficient coarse-grained reconfigurable architecture for mobile systems. J. Supercomput. **26**(3), 283–308 (2003)
9. C. Mei, P. Cao, Y. Zhang, B. Liu, L. Liu, Hierarchical pipeline optimization of coarse grained reconfigurable processor for multimedia applications, in *IEEE International Parallel Distributed Processing Symposium Workshops* (May 2014), pp. 281–286
10. T. Miyamori, K. Olukotun, REMARC: reconfigurable multimedia array coprocessor. IEICE Trans. Inf. Syst. **E82-D**(2), 389–397 (1999)
11. H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, R.R. Taylor, Piperench: a virtualized programmable datapath in 0.18 micron technology, in *IEEE Custom Integrated Circuits Conference* (May 2002), pp. 63–66
12. B. Mei, S. Vernalde, D. Verkest, H.D. Man, R. Lauwereins, *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix* (Springer, Berlin, Heidelberg, 2003), pp. 61–70
13. M. Motomura, STP engine, a C-based programmable HW core featuring massively parallel and reconfigurable PE array: its architecture, tool, and system implications, in *Proceedings of Cool Chips XII* (2009), pp. 395–408
14. T. Sugawara, K. Ide, T. Sato, Dynamically reconfigurable processor implemented with IPFlex's DAPDNA technology. IEICE Trans. Inf. Syst. **E87-D**(8), 1997–2003 (2004)
15. S.M.A.H. Jafri, S.J. Piestrak, K. Paul, A. Hemani, J. Plosila, H. Tenhunen, Energy-aware fault-tolerant CGRAs addressing application with different reliability needs, in *Euromicro Conference on Digital System Design* (September 2013), pp. 525–534
16. K. Kinoshita, Y. Yamaguchi, D. Takano, T. Okamura, T. Yao, Energy efficiency improvement by dynamic reconfiguration for embedded systems. IEICE Trans. Inf. Syst. **E98-D**(2), 220–229 (2015)
17. Japan Exchange Group, New, Enhanced TSE arrowhead cash equity trading system—for a safer, more convenient market (September 2015). News Release, http://www.jpx.co.jp/english/corporate/news-releases/0060/20150924-01.html
18. NASA Electronic Parts and Packaging, Military and aerospace FPGA and applications (MAFA) meeting (November 2007), https://nepp.nasa.gov/mafa/
19. Audi selects Altera SoC FPGA for production vehicles with 'piloted driving' capability (January 2015). Intel News Release, https://newsroom.intel.com/news-releases/audi-selects-altera-soc-fpga-production-vehicles-piloted-driving-capability/
20. T. Tanamoto, H. Sugiyama, T. Inokuchi, T. Marukame, M. Ishikawa, K. Ikegami, Y. Saito, Scalability of spin field programmable gate array: a reconfigurable architecture based on spin metal-oxide-semiconductor field effect transistor. J. Appl. Phys. **109**(7), 1–4 (2011), 07C312
21. S. Kaeriyama, T. Sakamoto, H. Sunamura, M. Mizuno, H. Kawaura, T. Hasegawa, K. Terabe, T. Nakayama, M. Aono, A nonvolatile programmable solid-electrolyte nanometer switch. IEEE J. Solid-State Circuits **40**(1), 168–176 (2005)
22. Markets and Markets, FPGA Market by Architecture (Sram, Fuse, Anti-Fuse), Configuration (High End, Mid-Range, Low End), Application (Telecommunication, Consumer Electronics, Automotive, Industrial, Military and Aerospace, Medical, Computing and Data Centers), and Geography—Trends and Forecasts From 2014–2020 (January 2015)
23. Top500—performance development (November 2015), http://www.top500.org/statistics/perfdevel/
24. Green500, http://www.green500.org

25. Y. Ajima, T. Inoue, S. Hiramoto, S. Uno, S. Sumimoto, K. Miura, N. Shida, T. Kawashima, T. Okamoto, O. Moriyama, Y. Ikeda, T. Tabata, T. Yoshikawa, K. Seki, T. Shimizu, *Tofu Interconnect 2: System-on-Chip Integration of High-Performance Interconnect* (Springer International Publishing, 2014). pp. 498–507
26. HA-PACS project, https://www.ccs.tsukuba.ac.jp/research/research_promotion/project/ha-pacs
27. NVIDIA GPUDirect, https://developer.nvidia.com/gpudirect
28. NVIDIA Corporation, NVIDIA NVlink high-speed interconnect: application performance. Technical report, NVIDIA Corporation (November 2014), whitepaper
29. J. Kim, Y. Kim, HBM: Memory solution for bandwidth-hungry processors, in *Hot Chips: A Symposium on High Performance Chips* (August 2014), HC26.11-310
30. T. Hanawa, T. Boku, S. Miura, M. Sato, K. Arimoto, PEARL: power-aware, dependable, and high-performance communication link using PCI express, in *Proceedings of IEEE/ACM International Conference on Green Computing and Communications and IEEE/ACM International Conference on Cyber, Physical and Social Computing* (December 2010), pp. 284–291
31. Y. Kodama, T. Hanawa, T. Boku, M. Sato, PEACH2: an FPGA-based PCIe network device for tightly coupled accelerators. ACM SIGARCH Comput. Archit. News **42**(4), 3–8 (2014)
32. AKIB Networks, INC, Bonet switch, http://www.akibnetworks.com/product2.html
33. P.S. Pacheco, *Parallel Programming with MPI* (Morgan Kaufmann, 1996)
34. T. Kuhara, T. Kaneda, T. Hanawa, Y. Kodama, T. Boku, H. Amano, A preliminarily evaluation of PEACH3: a switching hub for tightly coupled accelerators, in *Proceedings of International Symposium on Computing and Networking* (December 2014), pp. 377–381
35. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RDMA over Converged Ethernet
36. K. Matsumoto, T. Hanaway, Y. Kodama, H. Fujiiz, T. Boku, Implementation of CG method on GPU cluster with proprietary interconnect TCA for GPU direct communication, in *Proceedings of Accelerators and Hybrid Exascale Systems in Conjunction with IEEE International Parallel & Distributed Processing Symposium* (May 2015), pp. 647–655
37. F. Cerqueira, B.B. Brandenburg, A comparison of scheduling latency in Linux, PREEMPT-RT, and LITMUS-RT (July 2013)
38. EXABLAZE, EXALINKFUSION (September 2015), product brochure
39. Xilinx: 7 Series FPGAs Overview (DS890), 2.2 edn. (August 2015), preliminary Product Specification
40. Xilinx: UltraScale Architecture and Product Overview, 2.6 edn. (December 2015), product Specification (DS890)
41. ARISTA: 7124FX Application Switch (April 2014), datasheet
42. CISCO: Cisco Nexus 7000 Series FPGA/EPLD Upgrade, release 4.1 edn. (April 2009), Release Notes
43. MELLANOX: Programmable ConnectX-3 Pro Adapter Card, rev 1.0 edn. (November 2014), product Brief 15-4369PB
44. Simplex Inc., Equities Solution SimplexBLAST (January 2014), http://www.simplex.ne.jp/en/
45. S. Scott-Hayward, S. Natarajan, S. Sezer, A survey of security in software defined networks. IEEE Commun. Surv. Tutor. **17**(4), 2317–2346 (2015)
46. N. Mihai, G. Vanecek, New generation of control planes in emerging data networks, in *Proceedings of First International Working Conference*, vol. 1653 (1999), pp. 144–154
47. N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. **38**(2), 69–74 (2008)
48. Open networking foundation (2011), https://www.opennetworking.org/
49. SDNet development environment, http://www.xilinx.com/products/design-tools/software-zone/sdnet.html
50. Arrive technologies, http://www.arrivetechnologies.com/
51. A. Bitar, M. Abdelfattah, V. Betz, Bringing programmability to the data plane: packet processing with a NoC-enhanced FPGA, in *Proceedings of International Conference on Field-Programmable Technology* (2015), pp. 1–8

52. K. Guerra-Perez, S. Scott-Hayward, OpenFlow multi-table lookup architecture for multi-gigabit software defined networking (SDN), in *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research* (2015), pp. 1–2
53. W. Jiang, V.K. Prasanna, N. Yamagaki, Decision forest: a scalable architecture for flexible flow matching on FPGA, in *Proceedings of International Conference on Field Programmable Logic and Applications* (2010), pp. 394–399
54. H. Nakahara, T. Sasao, M. Matsuura, A packet classifier using LUT cascades based on EVMDDS (k), in *Proceedings of International Conference on Field Programmable Logic and Applications* (September 2013), pp. 1–6
55. J. Naous, D. Erickson, G.A. Covington, G. Appenzeller, N. McKeown, Implementing an Openflow switch on the NetFPGA platform, in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2008), pp. 1–9
56. S. Pontarelli, M. Bonola, G. Bianchi, A. Caponey, C. Cascone, Stateful openflow: hardware proof of concept, in *Proceedings of International Conference on High Performance Switching and Routing* (2015), pp. 1–8
57. Y.R. Qu, H.H. Zhang, S. Zhou, V.K. Prasanna, Optimizing many-field packet classification on FPGA, multi-core general purpose processor, and GPU, in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2015), pp. 87–98
58. P. Gupta, N. McKeown, Algorithms for packet classification. IEEE Netw. Mag. Global Internetwork. **15**(2), 24–32 (2001)
59. Toshiba develops NPEngineTM, the world's first hardware engine that directly streams video content from SSD to IP networks (April 2012). Toshiba Press Release, https://www.toshiba.co.jp/about/press/2012_04/pr0901.htm
60. P. Gupta, Xeon+FPGA platform for the data center, in *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic in Conjunction with International Symposium on Computer Architecture*, http://www.ece.cmu.edu/calcm/carl/
61. Altera: Hybrid Memory Cube Controller IP Core (May 2016), user Guide (UG-01152)
62. Xilinx: The Rise of Serial Memory and the Future of DDR, 1.1 edn. (March 2015), white Paper (WP456)
63. Hybrid Memory Cube Consortium, http://www.hybridmemorycube.org/
64. B. Yang, R. Karri, An 80Gbps FPGA implementation of a universal hash function based message authentication code, in *The DAC/ISSCC Student Design Contest* (2004) pp. 1–7, Operational Category: 3rd Place Winner
65. Y. Qu, V.K. Prasanna, High-performance pipelined architecture for tree-based IP lookup engine on FPGA, in *Proceedings of IEEE International Parallel and Distributed Processing Symposium, Workshops and Ph.D. Forum, Reconfigurable Architectures Workshop* (May 2013), pp. 114–123
66. K.E. Grosspietsch, Associative processors and memories: a survey. Micro, IEEE **12**(3), 12–19 (1992)
67. T. Kohonen, *Associative Memory* (Springer, 1977)
68. Altera Staff, Designing switches & routers with APEX CAM. White Paper M-WP-APEXCAM-02, Altera Corporation (October 2000)
69. Xilinx Staff, Content Addressable Memory (CAM) in ATM Applications. Application Note XAPP202 (v1.2), Xilinx Inc. (January 2001)
70. S.A. Guccione, D. Levi, D. Downs, A reconfigurable content addressable memory, in *Proceedings of IPDPS Workshops on Parallel and Distributed Processing* (2000), pp. 882–889
71. TCAMs and BCAMs: Ternary and Binary Content-Addressable Memory Compilers, https://www.esilicon.com/services-products/products/custom-memory-ip-and-ios/specialty-memories/tcam-and-bcam-compilers/
72. 80 Mbit Dual-Port Interlaken-LA TCAMs Achieve 2BSPS Deterministic Lookups, https://www.renesas.com/ja-jp/media/products/memory/network-search-engine/r10cp0002eu0000_tcam.pdf
73. Network Search Engine, R8A20686BG-G

74. W. Jiang, K. Viktor, Prasanna, A FPGA-based parallel architecture for scalable high-speed packet classification, in *Proceedings of IEEE International Conference on Application-specific Systems, Architectures and Processors* (July 2009), pp. 24–31
75. D. Burger, Transitioning from the era of multicore to the era of specialization, in *Keynote Speech at SICS Multicore Day* (2015), https://www.sics.se/sites/default/files/pub/sics.se/doug_burger_catapult_-_sics.pdf
76. A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, D. Burger, A Reconfigurable fabric for accelerating large-scale datacenter services, in *Procceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)* (June 2014), pp. 13–24
77. A. Putnam, A. Caulfield, E. Chung, et al., Large-scale reconfigurable computing in a Microsoft datacenter, in *Proceedings of Hot Chips 26* (August 2014)
78. D.A. Buell, J.M. Arnold, W.J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*. (Wiley-IEEE Computer Society Press, 1996)
79. B.S.C. Varma, K. Paul, M. Balakrishnan, D. Lavenier, FAssem: FPGA based acceleration of de novo genome assembly, in *Proceeding of the Annual International IEEE Symposium on Field-Programmable Custom Computing Machines* (April 2013), pp. 173–176
80. D.R. Zerbino, E. Birney, Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res. **18**(5), 821–829 (2008)
81. B. Langmead, C. Trapnell, M. Pop, S.L. Salzberg, Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. **10**(3), R25 (2009)
82. H. Li, R. Durbin, Fast and accurate short read alignment with burrowswheeler transform. Bioinformatics **25**(14), 1754–1760 (2009)
83. H.M. Waidyasooriya, M. Hariyama, Hardware-acceleration of short-read alignment based on the burrows-wheeler transform. IEEE Trans. Parallel Distrib. Syst. 1–8 (2015), http://www.computer.org/csdl/trans/td/preprint/07122348-abs.html
84. London Stock Exchange Group, Turquoise derivatives: FTSE 100 Index Futures (June 2013). Factsheet, https://www.lseg.com/sites/default/files/content/documents/LSEG_FTSE_100_Futures_Factsheet.pdf
85. London Stock Exchange Group, Turquoise derivatives: FTSE 100 Index Options (June 2013). Factsheet, https://www.lseg.com/sites/default/files/content/documents/LSEG_FTSE_100_Options_Factsheet.pdf
86. Singapore Exchange, SGX invests $250 million to offer fastest access to Asia (June 2010). News Release, http://investorrelations.sgx.com/releasedetail.cfm?releaseid=590607
87. The New York Stock Exchange, NYSE MKT transition to NYSE American (March 2017), https://www.nyse.com/publicdocs/nyse/markets/nyse-american/Pillar_Update_NYSE_American_March_2017.pdf
88. M. Lewis, *Flash Boys: A Wall Street Revolt* (W. W. Norton & Company Inc., 2014)
89. R. Cont, Statistical modeling of high-frequency financial data. IEEE Signal Process. Mag. **28**(5), 16–25 (2011)
90. Viraf Reporter, The value of a millisecond: finding the optimal speed of a trading infrastructure. Technical report, TABB Group (April 2008), https://community.rti.com/sites/default/files/archive/V06-007_Value_of_a_Millisecond.pdf
91. G. Linden, Make data useful. CS345, Stanford University (December 2006), first version
92. jcl, Hacking a HFT system. The Financial Hacker (July 2017). The Financial Hacker: a new view on algorithmic trading, http://www.financial-hacker.com/hacking-hft-systems/
93. M. O'Hara, FPGA and hardware accelerated trading, part five the view from Intel (August 2012). The Trading Mesh, http://www.thetradingmesh.com/pg/blog/mike/read/60770/fpga-hardware-accelerated-trading-part-five-the-view-from-intel
94. E.M. Murphy, Order instituting administrative and cease-and-desist proceedings, pursuant to sections 15(b) and 21c of the securities exchange act of 1934, making findings, and imposing remedial sanctions and a cease-and-desist order. The Securities and Exchange Commission,

https://www.sec.gov/litigation/admin/2013/34-70694.pdf (October 2013), release #70694, File #3-15570
95. T. Makimoto, Implications of Makimoto's wave. Computer **46**(12), 32–37 (2013)
96. C. Leber, B. Geib, H. Litz, High frequency trading acceleration using FPGAs, in *International Conference on Field Programmable Logic and Applications* (September 2011), pp. 317–322
97. M. Dvořák, J. Kořenek, Low latency book handling in FPGA for high frequency trading, in *International Symposium on Design and Diagnostics of Electronic Circuits Systems* (April 2014), pp. 175–178
98. J.W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, K. Vissers, A low-latency library in FPGA hardware for high-frequency trading (HFT), in *IEEE Annual Symposium on High-Performance Interconnects* (August 2012), pp. 9–16
99. J.W. Lockwood, A. Gupte, N. Meh, M. Blott, T. English, K. Visser, A low-latency library in FPGA hardware for high-frequency trading (HFT), in *Proceedings of IEEE 20th Annual Symposium on High-Performance Interconnects* (August 2012), pp. 9–16
100. OpenCL running on FPGAs accelerates Monte Carlo analysis of Black-Scholes financial market model by 10x, https://forums.xilinx.com/t5/Xcell-Daily-Blog/OpenCL-running-on-FPGAs-accelerates-Monte-Carlo-analysis-of/ba-p/435490
101. Altera Staff, Implementing FPGA design with the OpenCL standard. White Paper WP-01173-3.0, Altera Corporation (November 2013)
102. D.B. Thomas, Acceleration of financial Monte-Carlo simulations using FPGAs, in *IEEE Workshop on High Performance Computational Finance* (November 2010), pp. 1–6
103. M. O'Hara, Accelerating transactions through FPGA-enabled switching: an interview with John Peach of Arista networks. HFT Review Ltd. (June 2012)
104. T. Yanagisawa, H. Kurosaki, A. Nakajima, Activities of JAXA's innovative technology center on space debris observation, in *Advanced Maui Optical and Space Surveillance Technologies Conference* (2009)

Chapter 8
Advanced Devices and Architectures

Masato Motomura, Masanori Hariyama and Minoru Watanabe

Abstract The last chapter of this book is for advanced devices and brand new architectures around FPGAs. Since the basic logic blocks of FPGAs are consisting of LUTs, they are called fine-grained reconfigurable architectures. In contrast, coarse-grained reconfigurable architectures use processing elements to improve the performance per power for computation-centric applications. Dynamic reconfiguration is also easily done in such an architecture, and the configuration data set is called a hardware context. By switching hardware context frequently, they can achieve better usage of semiconductor area. The next part is asynchronous FPGA which can be a breakthrough of high-performance operation with low-power consumption. The handshake mechanism, a key component of such architectures, is explained in detail. 3D implementation is another new trend, while 2.5D is now in commercial use. The last part of this chapter is for activities of optical techniques around FPGAs for drastic improvement I/O and reconfiguration performance.

Keywords CGRA · Hardware context · Asynchronous FPGAs
Optical I/O · Optical reconfiguration

8.1 Coarse-Grained Reconfigurable Architecture

As was explained in Chap. 1, FPGAs started as devices for prototyping small-scale logic circuits. As they become larger in accordance with the shrink in transistor size, the idea to use FPGAs as acceleration devices is getting more popular, as

M. Motomura
Hokkaido University, Sapporo, Japan
e-mail: motomura@ist.hokudai.ac.jp

M. Hariyama (✉)
Tohoku University, Sendai, Japan
e-mail: hariyama@tohoku.ac.jp

M. Watanabe
Shizuoka University, Shizuoka, Japan
e-mail: tmwatan@ipc.shizuoka.ac.jp

H. Amano (ed.), *Principles and Structures of FPGAs*,
https://doi.org/10.1007/978-981-13-0824-6_8

demonstrated in Chap. 7. This approach, known as reconfigurable computing or reconfigurable systems, is becoming more important as CPUs performance improvement are slowing down. It is natural to use an array of LUTs when the main purpose of an FPGA is prototyping.

As a device for reconfigurable computing, however, it may make more sense to use other primitive elements. To fit better for the acceleration of computing functions, such elements might be less versatile, but they should be more efficient in computing than LUTs. This is how coarse-grained reconfigurable architectures (CGRAs) have been proposed and investigated.

8.1.1 CGRA Basics and History

Starting from the 1980s, CGRAs have been mostly presented by universities and startups. The well-known ones are PipeRench from CMU and XPP from PACT, in addition to others [1]. Recently proposed good examples of such architectures, both in Japan, are CMA from Keio University [2] and LAPP from NAIST [3]. As shown in Fig. 8.1, a CGRA can be represented as an array of operation units and memories, associated with a network structure connecting them. As for the operation units granularity, there are varieties such as: 4, 8, 16, and 32 bits. The finer the architecture is the more it becomes like an FPGA. On the other hand, the coarser it is the more it becomes like a traditional parallel processor. As for the instruction set, traditional arithmetic logic operations are commonly found, as well as extended instructions for customized acceleration of target applications. The array configuration may not necessarily be a two-dimensional one as in FPGAs, but also a one-dimensional array when a target application is sufficient with linear processing. Either dynamic switching on-chip routing networks or static switching interconnection fabrics are used for the network in Fig. 8.1.

8.1.2 CGRA Design Space

Since CGRAs are equipped with operation units customized for given applications, they surpass FPGAs, in general, in processing performance and density. High density means more parallel operation units can be integrated into the same area. This also translates into better processing performance. In addition, configuration information can become much smaller compared to FPGAs. It is known that a major portion of configuration information is spent on interconnections: FPGA architectures require specifying bit-level interconnections. While in the CGRA case, interconnections are bundled to the coarse granularity specified by the architecture. Another important CGRA merit to note is its familiarity to design tools. This is very important since software programmers, in the reconfigurable computing field, are the users who map target applications onto the CGRA architecture.

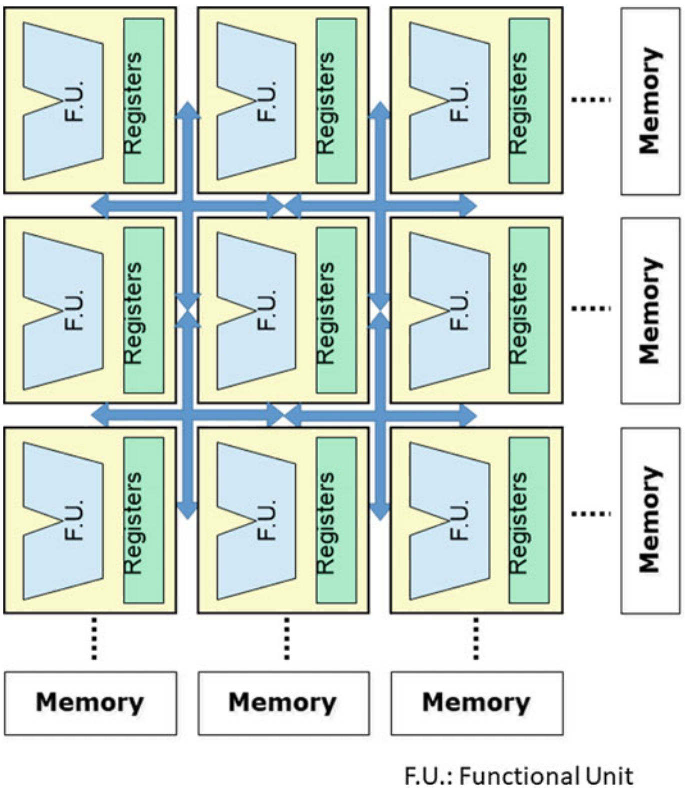


Fig. 8.1 Generalized CGRA architecture

There are also drawbacks in the CGRA approach. First, they become less general purpose when compared to FPGAs because of their structure. Considering that FPGAs have occupied a large portion of the market, because it is a general-purpose device (at least from the HW prototyping point of view), this is a major issue to be carefully considered. Many CGRA architectures ended up being just research prototypes because of this reason.

Another issue to carefully consider when defining the CGRA architecture is to efficiently utilize the hardware-based computation as much as possible. For example, when choosing a single bit from an 8-bit data, an FPGA needs just to wire the desired bit. Whereas in the CGRA case, an 8-to-1 selector or shifter is required. This means that an overhead is incurred both in performance and area.

Based on considerations such as the ones above, CGRA architectures are mainly considered as a tightly coupled accelerator to a CPU core, but not a replacement to FPGA. For example, CMA and LAPP allow the CGRA core to directly access CPU registers so that it can accelerate sub-tasks of a CPU (such as frequently executed loops).

8.1.3 Dynamically Reconfigurable Architecture

When an FPGA is used as a computation device, a resource limitation issue may rise; i.e., what should be done when the required computation does not fit within the hardware resources of a given FPGA? This problem is easy to solve when FPGAs are considered for prototyping: Just increase the number of FPGA chips and make the connection between them. If only computation is concerned, just like software does, a reconfigurable computation solution should robustly account for variously sized applications. This is the reason why dynamically reconfigurable architectures were investigated.

One of the earliest works in this area is WASMII from Keio University, Japan [4]. This work proposed quite an advanced concept where hardware is considering as a set of pages, which can swap in and out (Fig. 8.2). In conventional operating systems, a virtual memory allows a large memory space that does not fit in a physically available memory, to be allocated to applications using a page-by-page swapping method. Similarly to virtual memory, in WASMII, the page-oriented hardware architecture allows virtual hardware, where a virtually large hardware can be put on a physically existing small reconfigurable device.

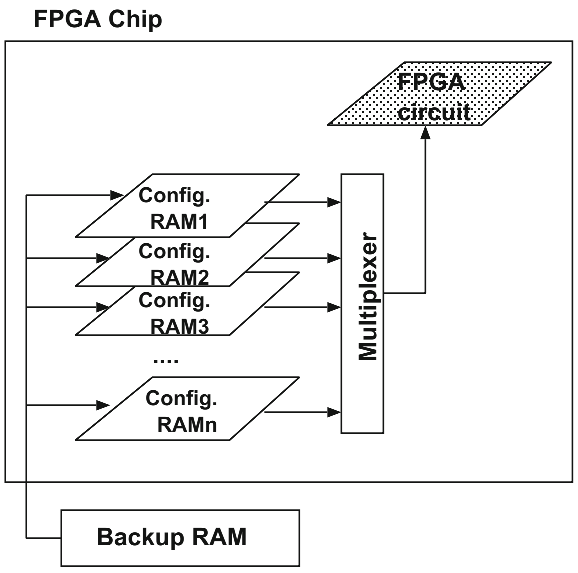


Fig. 8.2 WASMII execution model

8.1.4 Case Study: DRP

Dynamically Reconfigurable Processor (DRP) is a coarse-grained, dynamically reconfigurable architecture proposed by NEC Corporation in 2002 [5]. The architecture was mainly developed for an IP core integrated into an SoC. This section overviews DRPs as an example of dynamically reconfigurable architectures (they are also an example of a CGRA described in the previous section).

Figure 8.3 shows its basic architecture. A processing element (PE), that constitutes a two-dimensional array, is composed of two general-purpose 8-bit ALU, register file, and instruction memory. The two ALUs have bit-manipulation instructions such as bit mask/select, so that it can cover bit-level operations that FPGA can handle well. PEs are interconnected to each other with an 8-bit-width hierarchical bus. Bus selectors connect those ALUs and a register file with vertical/horizontal buses.

An instruction memory stores a set of hardware configurations, from which one configuration is selected, i.e., hardware dynamic reconfiguration. Each instruction includes operation codes for the two ALUs, as well as control bits for the bus selectors. For example, it is possible to bypass the register file in a PE and connects the ALU outputs to inputs of other PEs in a flow-through manner. Ordinary processors store outputs produced at one cycle into registers and then read those registers for following cycles. The PEs in a DRP, on the other hand, spatially connect plural PEs for constructing a customized datapath.

The PE array is associated with an state transition controller (STC), which is responsible for managing the dynamic reconfiguration. A basic role of an STC is to dispatch instruction pointer to the array: Each PE receives the pointer, then selects, and reads a specified instruction. The STC has a sequencer which keeps track of the state transitions of a given application. When a new pointer is dispatched, all the instructions of PEs, as well as all the interconnections in the PE array, change at once. This operation can be interpreted such that the hardware configuration changes among the datapath contexts stored in the memory. This makes it similar to the WASMII's concept explained in Fig. 8.2. Conditional state transitions require branches which need to examine branch conditions. Information required for this examination is returned back from the datapath as event signals to the STC.

A DRP core is made up of multiple tiles of PEs. Each tile has its own STC to control the reconfiguration. Hence, multi-tiled DRPs can run multiple state machines in parallel. There is also a mechanism to connect multiple tiles and control them by a single state machine.

The execution model of DRPs is usually learned from high-level synthesis tool which is a tool to compile a program written in high-level language like C to an executable hardware. Generally speaking, a program has control flows which are composed of conditional branches and loops, etc., and data flows which are trees of data handling operations. High-level synthesis tools compile control flows to finite-state machines, and data flows to hardware datapaths. DRP architectures feature clear one-to-one correspondence with this generic high-level synthesis model: STCs manage finite state machines, and PE arrays handle datapath. Here, a datapath asso-

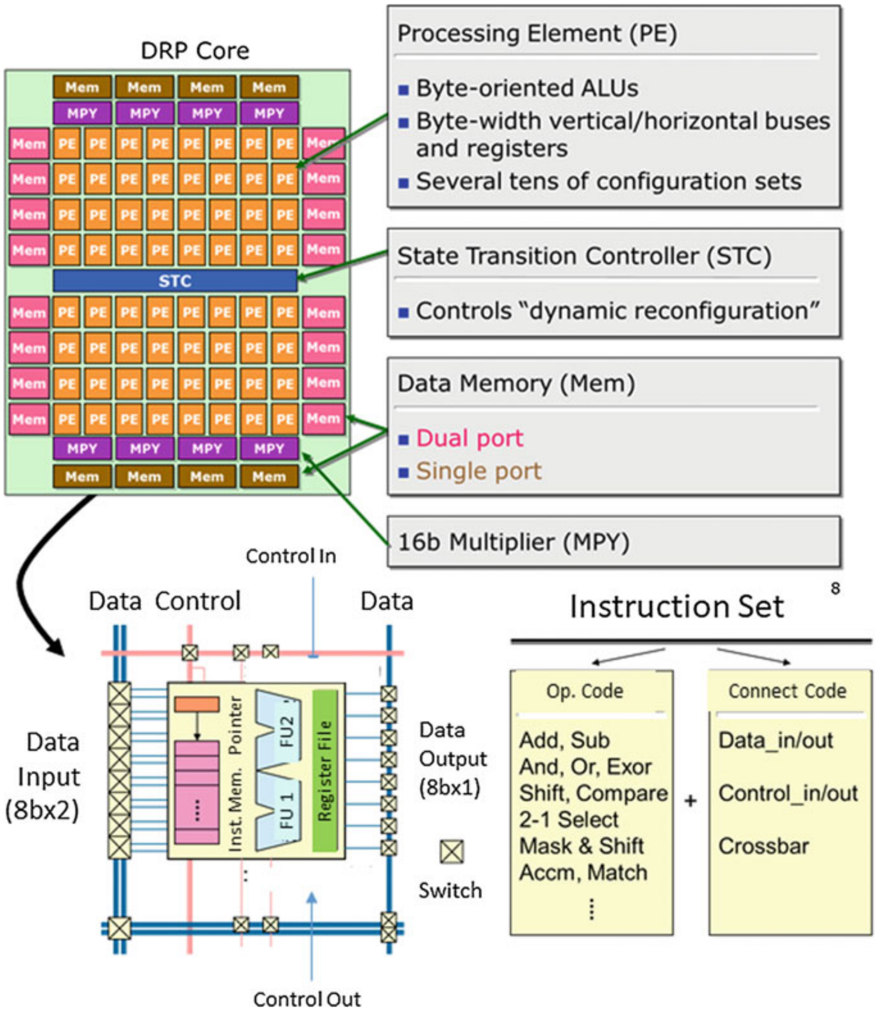


Fig. 8.3 DRP architecture

ciated with each state is called "context." Contexts are generated when (1) they are associated with states in the control flow, or (2) when there is a resource limitation and a context should be divided into multiple ones.

DRPs feature GUI-based high-level synthesis-oriented tool flow (Fig. 8.4). Context generation is all handled by this tool, and designers do not have to worry about how to decompose hardware datapaths. The DRP core, which is now owned by Renesas Electronics and a commercially used dynamically reconfigurable architecture, has been used in products such as video cameras and digital cameras.

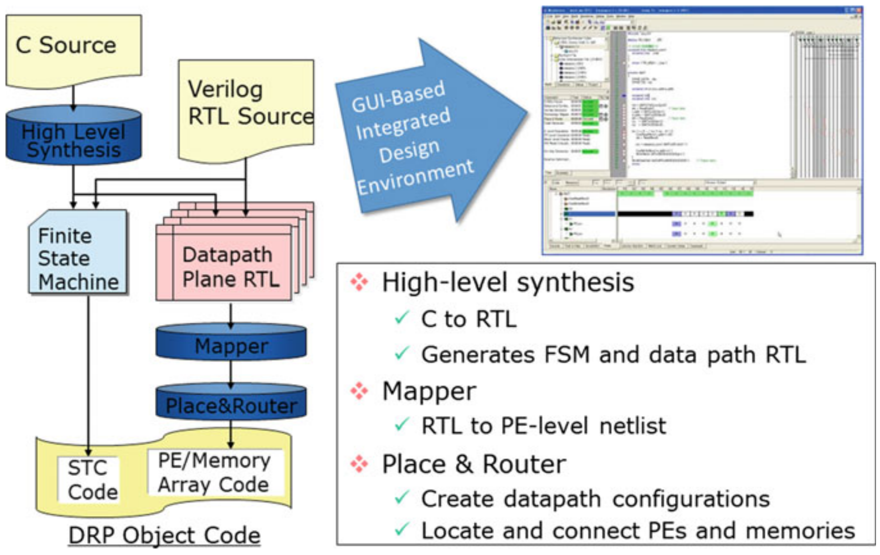


Fig. 8.4 DRP design tool

8.1.5 Relation to Parallel Processors

An important consideration in conducting dynamic hardware reconfiguration is the need to load large amount of configuration information at once. Typically, dynamically reconfigurable architectures feature one to several clock cycle latencies for this hardware context switch (for DRP, it is less than a single cycle). This is the reason why such architectures, including DRPs, adopt CGRAs in order to reduce configuration information.

Dynamically reconfigurable CGRA hardware may look very similar to on-chip many-core parallel processors (such as Xeon Phi from Intel). The difference becomes clear when their execution models are examined:

- **Dynamically reconfigurable CGRA**: A block of instructions are first spatially mapped on an array of processing elements (it constitutes a hardware context). Then, the hardware contexts are multiplexed in time (space to time order).
- **On-chip many-core parallel processor architecture**: It first assigns a block of instructions to a single processor as a thread. Then, multiple threads are mapped onto a processor array among which the synchronization will take place from time to time (time to space order).

8.1.6 Other Architecture Examples

FPGA-based (i.e., fine-grained) dynamically reconfigurable hardware architectures were proposed in Tabula [1]. Tabula exploits dynamic reconfiguration for speeding up FPGA operational frequency. That is, it divides the critical path into several segments and maps them to different hardware contexts. Tabula was proposed for realizing over-GHz range FPGA for high-end applications (the project was suspended in 2015).

8.2 Asynchronous FPGA

8.2.1 Problems of Conventional Synchronous FPGAs

In conventional synchronous circuits, some serious problems become obvious as the miniaturization of semiconductor process continues. Figure 8.5 shows the global clock network of synchronous FPGAs. The global clock network is connected to the clock inputs of all registers. A register loads data only at the rising edge of the clock pulse. As soon as the data is loaded, it appears on the output. The data on the register output is used as the input of the logic circuits. The result of these circuits is used as the input of the following register. FPGAs usually have much larger circuits and have much more registers than application-specific integrated circuits (ASICs). Therefore, conventional synchronous FPGAs have larger parasitic capacitance of the global clock network and require much more clock buffers to reduce clock skews. This causes the following problems:

- The clock network consumes larger power.
- The speed is limited by the clock skews.

As for conventional synchronous FPGAs, lowering the power consumption is not so easy compared to ASICs due to the following reasons:

Difficulty to use clock gating: Clock gating is a major technique to reduce the power consumption in ASICs. It prevents the input of a circuit from causing unnecessary signal transitions when the circuit is unused. When using clock gating for ASICs, designers should carefully design the customized clock network to avoid clock skews, and the clock network is fixed at the manufacturing phase. As for FPGAs, the clock network cannot be customized for a certain circuit since various circuits are implemented on FPGAs. Moreover, it is not recommended to use the clock network that can be reconfigured for the clock gating, since reconfiguring the clock network causes faults due to clock skews.

Difficulty to use power gating: Power gating is another major method to reduce the power consumption in ASICs. It turns the circuits power off when they are not in use and wake them up just before being used. The power gating requires control circuits

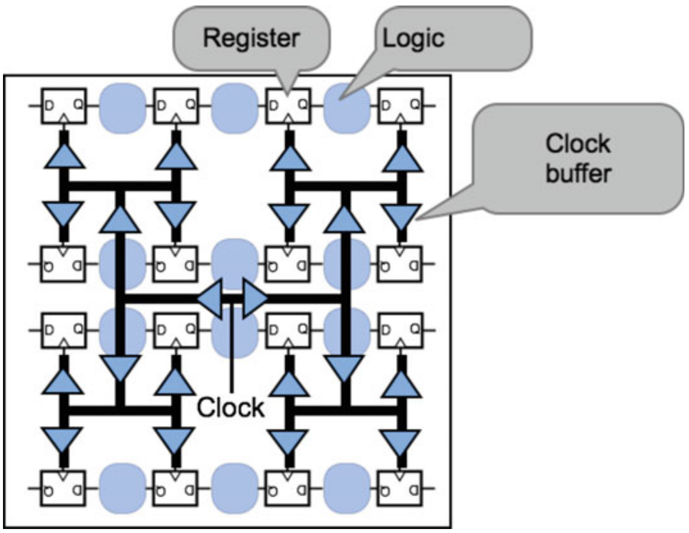


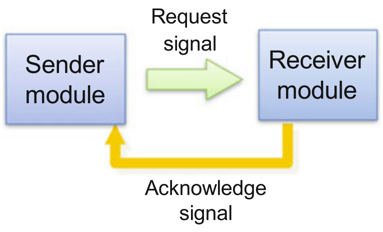
Fig. 8.5 Global clock network of synchronous FPGAs

and dedicated connections to distribute these control signals. Especially in FPGAs, these overheads are significantly large due to the flexibility of FPGAs.

8.2.2 Overview of Asynchronous FPGAs

In order to solve the problems of the synchronous FPGAs, FPGAs based on asynchronous circuits are proposed. Figure 8.6 shows the basic behavior of an asynchronous circuit. Data transfers between processing modules are done using a handshake protocol as follows. At first, the sender sends the data and a request signal to the receiver. The request signal is used to inform the receiver of the data arrival. After receiving the request signal, the receiver takes the data in and sends the acknowledge signal to the sender. The acknowledge signal is used to inform the sender that the

Fig. 8.6 Basic behavior of an asynchronous circuit



receiver has completed receiving the data. After the acknowledge signal reception, the sender sends a new data in the same manner.

The advantages of asynchronous circuits over synchronous ones are summarized as follows:

No dynamic power consumption in the inactive state: An asynchronous circuit does not consume power consumption when not processing data. This is because it does not have the global clock network that always transfers the clock pulse.
Low peak power or peak current: In asynchronous circuits, processing modules start to process data after they receive it. Since the data arrival times vary from each other, the durations of the power consumption peaks (and current peaks) of the modules vary from each other as well. As a result, the average power consumption of the whole circuit becomes low.
Low-Level electromagnetic radiation: Since the peak current is low as described before, the level of the electromagnetic radiation is also low.
Robust to the fluctuation of the supply voltage: Even when the supply voltage decreases slightly, it is guaranteed that the output of the circuit is correct thanks to its clock-less operation.

The major disadvantage of asynchronous circuits is their larger amount of hardware for control, e.g., circuits that detect data arrival, and the additional wires for acknowledge and request signals.

In asynchronous circuits, there are three major types of handshake protocols [6]:

(1) Bundled data protocol,
(2) Four-phase dual-rail protocol, and
(3) Level-encoded dual-rail (LEDR) protocol.

The bundled data protocol is also called single-rail protocol. It represents one bit of data by using a single-rail-like synchronous circuits. A word to be transferred consists of multiple data bits and 1-bit request signal. Hence, the overhead for control is only 1-bit per word, which can be considered as very small. The disadvantage of the bundled data protocol is that it requires a timing constraint for the request signal; the request signal must arrive at the receiver module after the data. In order to ensure this, a delay buffer is usually inserted into the request signal wire as shown in Fig. 8.7.

Figure 8.8 shows the data transfer and the encoding of the four-phase dual-rail protocol. In this protocol, a word to be transferred from the sender has 2 bits: 1 bit for data and 1 bit for the request signal, as depicted in Fig. 8.8a. The receiver sends the 1-bit acknowledge signal back to the sender. In general, 1-bit acknowledge signal is enough for multiple words. Figure 8.8b illustrates the encoding. Data “0” and “1” is represented by code words $(D_t, D_f) = (0, 1)$ and $(D_t, D_f) = (1, 0)$, respectively. As a separator between data, the spacer $(D_t, D_f) = (0, 0)$ is used. Note that $(D_t, D_f) = (1, 1)$ is invalid. Let us consider an example of data transfer represented in Fig. 8.8c, where data “1”, “1”, “0”, and “0” are transferred. Since the sender sends a code word and the spacer alternatively, the receiver can detect the data. In the four-phase dual-rail protocol, the racing problem caused by the arrival timings of data and the request signal does not occur. In other words, D_t and D_f do

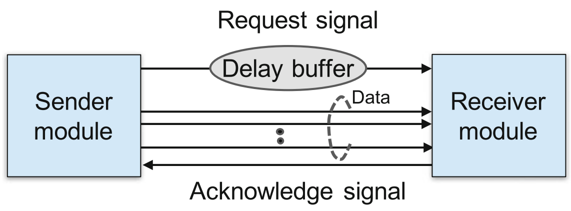


Fig. 8.7 Bundled data protocol

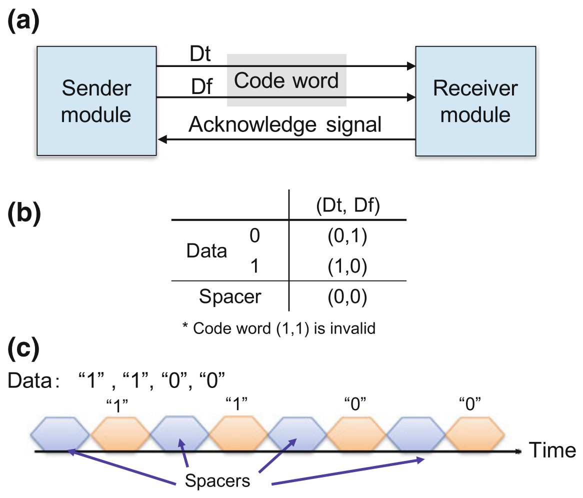


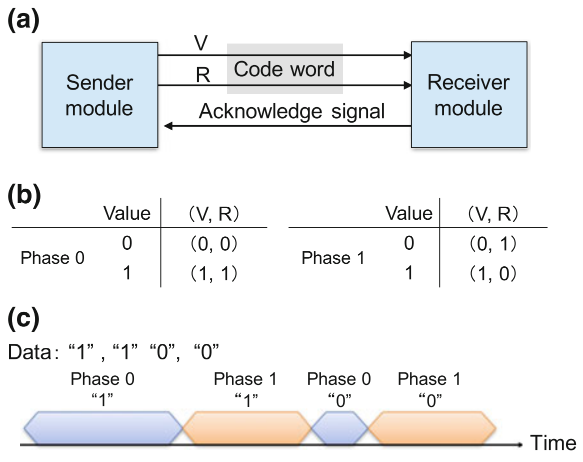
Fig. 8.8 Four-phase dual-rail protocol

not change at the same time. This is because the code words are designed such that the Hamming distance between any two code words is one. Hence, the four-phase dual-rail protocol is more robust for timing variations than the single-rail protocol.

The circuit of the four-phase dual-rail protocol is simpler than that of the LEDR protocol, described below, since the data corresponds to a single code word. The disadvantage of the four-phase dual-rail protocol over the LEDR protocol is its lower throughput due to the insertion of the spacer.

The LEDR protocol is suitable for high-throughput data transfer. Figure 8.9 shows the data transfer and the encoding of the LEDR protocol. The way of data transfer is the same as the four-phase dual-rail protocol, as presented in Fig. 8.9a. The big difference between the four-phase dual-rail and LEDR protocols is the encoding, as demonstrated in Fig. 8.9b. Data "0" is encoded by two different code words: $(V, R) = (0, 0)$ in phase 0 and $(V, R) = (0, 1)$ in phase 1. Data "1" is also encoded by two different code words: $(V, R) = (1, 1)$ in phase 0 and $(V, R) = (1, 0)$ in phase 1. Let us consider an example of a data transfer using the phases, as shown in Fig. 8.9c, where data "1", "1", "0", and "0" are transferred. In the LEDR protocol, the code word in phase 0 and the code word in phase 1 are alternatively transferred. The receiver can detect the change of data by detecting the change of phases. Since the LEDR protocol dose not need spacers, it can achieve high throughput. The disadvantage of this protocol is that it requires larger circuits since one data value has two different code words.

Fig. 8.9 LEDR protocol



Hereafter, we explain asynchronous FPGAs. Asynchronous FPGAs using the bundled data protocol have been proposed [7, 8]. Although they benefit from the used small circuits, the main disadvantage is their low performances due to the large delay buffers inserted in the request signal wire to ensure the correct behavior for various datapaths.

As for asynchronous FPGAs protocol, the dual-rail protocol is ideal since it can avoid the racing problem mentioned above without any timing constraints on the data nor the request signal. Figure 8.10 shows a basic architecture based on the dual-rail

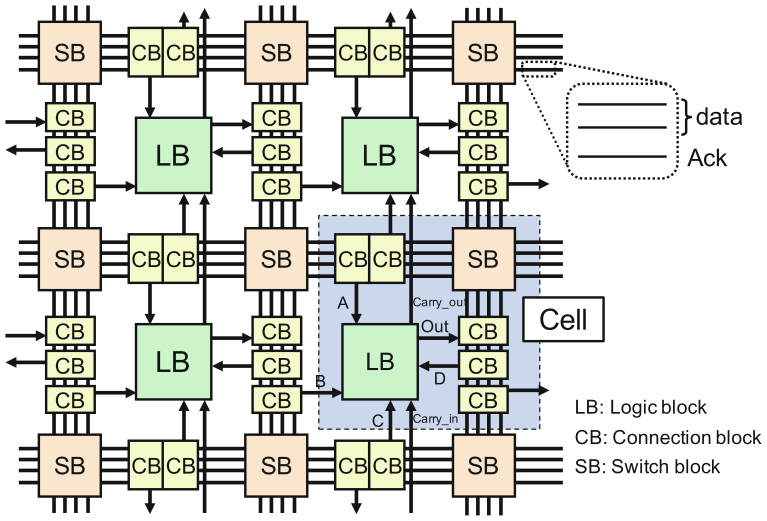


Fig. 8.10 Asynchronous FPGA based on the dual-rail protocol

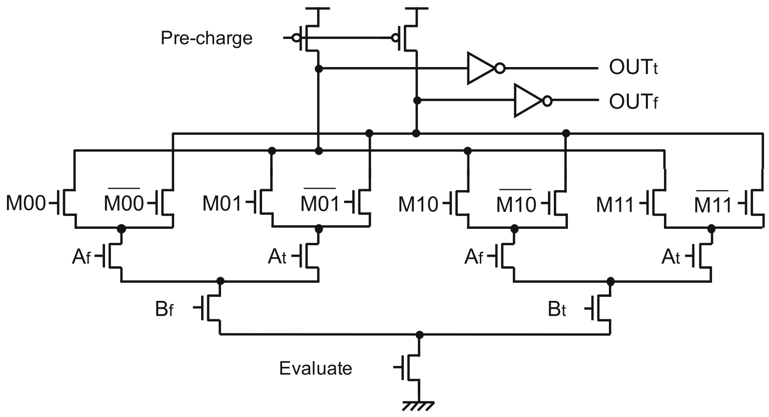


Fig. 8.11 LUT for the four-phase dual-rail protocol (2-input LUT)

protocol [9]. Similar to conventional synchronous FPGAs, logic blocks (LBs) are connected to each other via connection blocks (CBs) and switch blocks (SBs). An interconnection unit consists of wires for code words and the acknowledge signal.

Among the dual-rail protocols, the four-phase dual-rail protocol is employed to implement small circuits [10, 11]. Figure 8.11 shows the structure of an look-up table (LUT), where, for simplicity, the numbers of inputs and outputs are limited to two and one, respectively. The four-phase dual-rail protocol is suitable for dynamic circuits which are used for area-efficient design. This is because the pre-charge signals and the evaluation signal are easily generated from the spacer. The number of bits of the configuration memory is 2^N for an N-input LUT like conventional synchronous FPGAs. In the case of Fig. 8.11, the 2-input LUT has a 4-bit configuration memory (M00, M01, M10, M11). The output (OUT_t, OUT_f) is determined according to the configuration memory and the external inputs (A_t, A_f) and (B_t, B_f).

8.2.3 *Design for Low Power, High Throughput, and Modularity*

For low power, asynchronous circuits can provide some design information, and intelligent control is realized based on this information. For example, the receiver module can detect the data arrival by using the request signal; the sender module can know whether the receiver module is ready to load data. In [12], fine-grained adaptive control of the supply voltage is proposed to reduce the dynamic power consumption. According to the state of the receiver module, the sender module adaptively controls its supply voltage and processing speed. In [13], fine-grained power gating is proposed to reduce the static power consumption caused by the leakage current of transistors. Each module detects the arrival of its inputs data by using the request signal that is sent from the sender module. If the inputs do not arrive within a predefined time, the module automatically turns the supply voltage of the core circuit off. When the inputs come, the module wakes its core circuit up [13].

In order to achieve high throughput, fine-grained pipelining is frequently employed for high throughput datapaths [11, 14–16]. From the point of view of data transfer, the protocol hybrid architecture is proposed, where the LEDR protocol is used for high-throughput data transfer and the four-phase dual-rail protocol is used for simple datapaths [9, 17]. Moreover, the hybridization of synchronous circuits and asynchronous circuits is proposed [18]. When a significant amount of input data continuously arrives, synchronous circuits are considered to be efficient in terms of power consumption. On the other hand, asynchronous circuits are efficient for the case when the data arrival is less uniform. Based on this observation, an LUT is designed to be used in both asynchronous circuit and synchronous circuit while sharing the circuit. Depending on the used applications, the blocks of LUTs are configured as asynchronous or synchronous circuits. Note that both of asynchronous and synchronous circuits can coexist on a single FPGA. High throughput and low power can be optimally achieved by combining the asynchronous and synchronous circuits based on their aptitudes for different applications.

One major problem in asynchronous circuits is their difficulty to program. The reason is that the design for modularity is not easy in asynchronous circuit. To solve this problem, design methods using handshake components are proposed for general asynchronous circuits [19, 20]. Handshake components are basic building blocks to describe the data flow and control flow, including arithmetic/logic operations, conditional branch, and sequence control. Designing circuits is easily done by connecting such handshake modules. In [21], an asynchronous FPGA is proposed whose logic blocks are suitable to implement the handshake components.

8.3 3D FPGA

As described before, FPGAs consist of a configuration memory, programmable interconnection units, and programmable logic circuits to achieve a high degree of flexibility. Such redundant resources lead to a lower area efficiency compared to ASICs. Moreover, the complex interconnection causes a large delay and degrades the performance. These problems will be more serious in the near future since the miniaturization of the semiconductor manufacturing process nears the physical limit.

Based on this background, applying 3D integration technologies such as TSV (Through Silicon Vias) [22–24] to FPGAs is strongly desired. 3D FPGAs are classified into two types: heterogeneous and homogeneous.

Figure 8.12 shows the conventional 2D FPGA architecture and the 3D heterogeneous architecture. As shown in Fig. 8.12 (bottom), different resources such as logic blocks, routing blocks, and a configuration memory are distributed into different layers; the resources in different layers are connected by using interconnections such as TSV. Therefore, the heterogeneous architecture can increase the resource density per

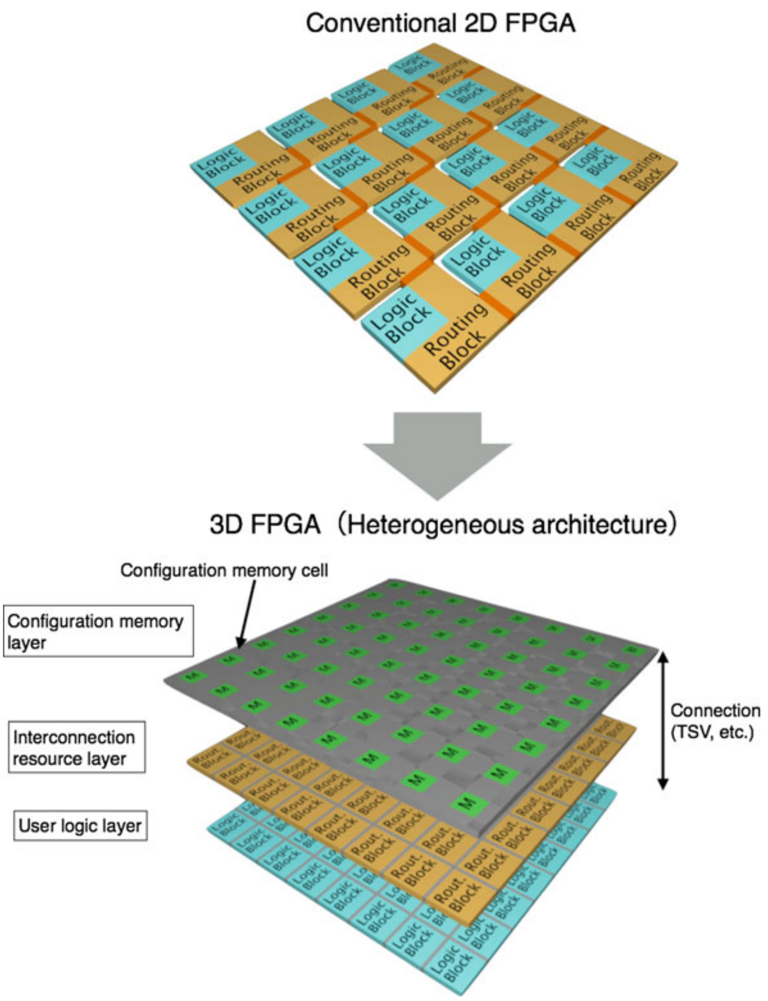

Conventional 2D FPGA
Logic Block
Routing Block
3D FPGA (Heterogeneous architecture)
Configuration memory cell
Configuration memory layer
Interconnection resource layer
User logic layer
Connection (TSV, etc.)
M
Rout. Block
Logic Block

Fig. 8.12 3D FPGA (heterogeneous architecture)

footprint [25–28]. The scalability along the vertical direction of the heterogeneous architecture is lower than that of the homogeneous architectures, described below, since the number of layers is limited by the number of resource types.

Figure 8.13 shows the 3D homogeneous architecture. Each layer has the same functions as a 2D FPGA, that is, logic blocks, routing blocks, a configuration memory. The routing block is designed such that it connects the logic blocks in the same layer and also connects the routing blocks in different layers via vertical connections [29–32]. Hence, the homogeneous architecture is an extension of the 2D FPGA architecture to the third dimension. When the number of stacked layers increases in accordance with the progress of the 3D integration technology, the total circuit

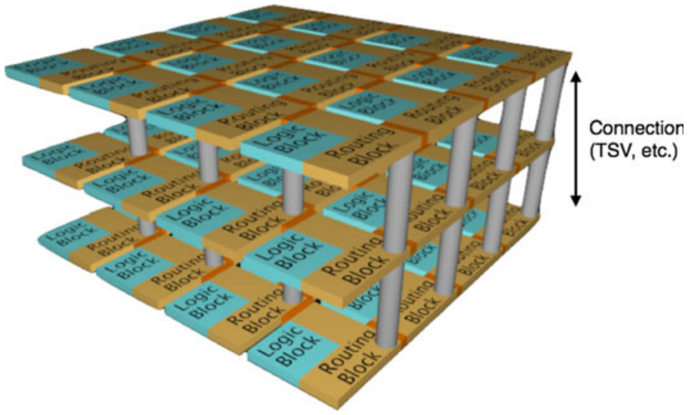


Fig. 8.13 3D FPGA (homogeneous architecture)

size of a 3D FPGA can also increase linearly. Moreover, the performance could be improved compared to 2D FPGAs. When mapping circuits with complex topology onto a conventional 2D FPGA, the connected circuit blocks are not always mapped onto near logic blocks. As a result, it may result in mapping with long wires. 3D FPGAs can map such circuits onto near logic blocks by using different layers.

Hereafter, the issues of 3D FPGAs are summarized. The first issue is the challenge facing the technologies to use for the vertical connections, which should be cheap and highly reliable. Moreover, when increasing the number of layers in the homogeneous architecture, the thermal radiation can be critical as well as CAD support [33–39].

8.4 High-Speed Serial I/O

Microsoft recently announced a server using the Stratix V FPGAs to be used in its data center for Bing search engine [40]. Although the introduction of FPGAs has increased the power consumption by 10%, it enhanced the throughput by 95% when compared to the software implementation. Thus, this has emphasized the effectiveness of FPGAs. In this implementation, the 10-Gbps high-speed communication port of the FPGA is used for a mutual network that is indispensable in a data center. In recent years, this case underscores the further emerging importance of networks that can leverage FPGA applications.

As described in Chap. 3, recent FPGAs provide many general-purpose inputs/outputs (GPIOs) that can accommodate various devices such as memories. GPIOs readily realize an interface with various devices connected to an FPGA at a high bandwidth. Recent FPGAs are equipped with serial I/Os that allow high-speed communications of Gbps order in addition to GPIOs, as highlighted in the data center case described above. Accordingly, short-distance communications between FPGA chips, middle-distance communications between systems including FPGA chips, and long-distance network communications between systems including FPGA chips have been implemented at higher speeds. Xilinx Inc. and Altera Corp. mutually compete in terms of performance, and they are locked in a development race to mount more high-speed serial I/Os on their own cutting-edge FPGAs. As a result, FPGAs communication performance has improved rapidly in recent years. The importance of serial I/Os in FPGAs is anticipated to further increase in the future. Therefore, this section describes these high-speed serial I/Os in the Stratix family devices, as an example.

8.4.1 LVDS

The Stratix family supports differential interfaces of small amplitude such as Low Voltage Differential Signaling (LVDS) [41], Mini-LVDS [42], and Reduced Swing Differential Signaling (RSDS) [43]. Mini-LVDS and RSDS are standards derived from LVDS for computer displays, formulated, respectively, by Texas Instruments Inc. and National Semiconductor. This section describes LVDS, which has been standardized by ANSI/TIA/EIA-644. The Stratix family adopts LVDS that satisfies this standard [44, 45].

LVDS is a one-way signal transmission standard under which a signal is transmitted from a transmitting side to a receiving side using two lines, as indicated in Fig. 8.14. For example, in cases where a signal “1” is transmitted from the transmitting side, transistors (1) and (2) in Fig. 8.14 are turned ON for transmission. In this event, the current flows from the current source on the transmitting circuit to the upper line via transistor (1). A terminator is mounted on the receiving circuit. The great portion of the current flows into the terminator and returns to the transmitting circuit via the other line to flow into VSS via transistor (2). At this time, the potential between both terminals of the terminator on the receiving side rises to about +350 mV. A differential amplifier on the receiving circuit detects this state. Then, the receiving circuit determines it as a signal of “1”.

On the other hand, when transmitting a signal of “0”, transistors (3) and (4) on the transmitting circuit are turned ON. Thereby, the current flows from the current source through the lower line. Similarly to the description above, the current passes through the terminator, returns to the transmitting terminal via the upper line, and flows into VSS via transistor (3). At this time, a potential of −350 mV occurs at the terminator. Consequently, the current flows in an opposite direction according to the transmitted value of “1” or “0”, and a potential of +350 mV occurs on the receiving circuit. The receiving circuit judges whether the transmitted value is “0” or “1” by detecting this potential. This small amplitude allows high-speed and low-power communications.

The Stratix IV GX 40-nm FPGAs are equipped with 28–98 LVDS ports that support high-speed communications up to 1.6 Gbps [46]. The number of ports described above is expressed by the number of full-duplex channels through which transmission and reception are conducted simultaneously; e.g.,“28 ports” denote that there

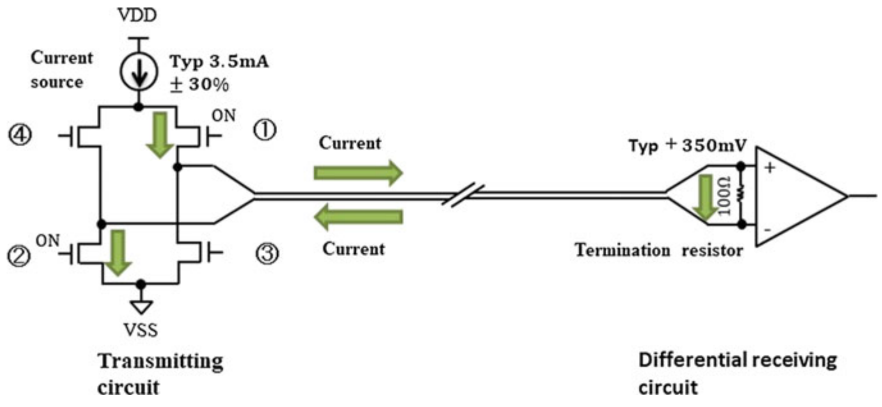


Fig. 8.14 Schematic view of LVDS transmitting and receiving circuits

are 28 LVDS ports for transmission and 28 LVDS ports for reception. The number of available ports might be different depending on the package, even for FPGAs of the same size. Meanwhile, the Stratix V manufactured by the 28-nm TSMC process supports LVDS ports up to 1.4 Gbps [44]. The Stratix V GX device is equipped with 66–174 full-duplex LVDS ports [45].

The Stratix family I/Os for high-speed communications have a built-in hard macro serializer/deserializer (SerDes) circuit up to 10 bit. It is difficult to build a communication circuit that can directly operate as fast as 1.4–1.6 Gbps inside an FPGA. However, the hard macro of the serializer can easily convert a signal from a parallel transmitting circuit using, for example, a 10-bit FIFO operating at a low clock frequency into a high-speed serial signal as fast as 1.4–1.6 Gbps. The block diagram of a transmitting circuit is presented in Fig. 8.14. The deserializer at a receiving circuit can convert a 1-bit high-speed serial signal to a 10-bit parallel signal in the same way, so that a receiving circuit can be constructed with a FIFO operating at a low clock frequency. Furthermore, a resistance of 100 Ω that terminates the differential signal at the receiving side of an LVDS is programmable in the Stratix V, so that high-speed communications between FPGA chips can be easily implemented without extra parts just with the board design considering the impedance. A guideline for board design is provided from Altera. One report of the relevant literature is particularly useful [46] in describing the board design (Fig. 8.15).

8.4.2 28-Gbps High-Speed Serial I/O

Stratix still supports more high-speed serial I/O in addition to LVDS. For instance, Stratix V GX FPGA and Stratix V GS FPGA are equipped with up to 66 high-speed communication ports that operate at 12.5 Gbps. The Stratix V GT FPGA is equipped with four 28-Gbps high-speed communication ports in addition to 32 14.1-Gbps communication ports.

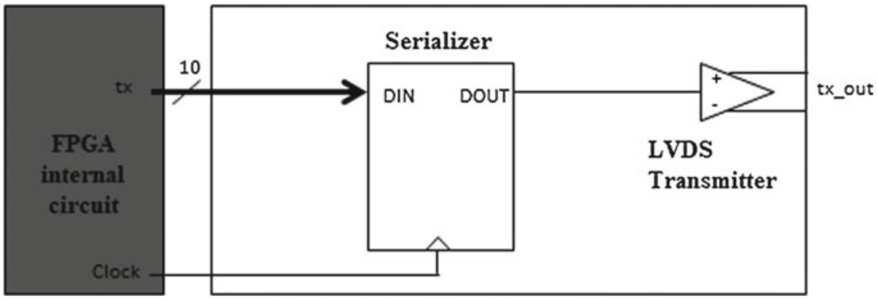


Fig. 8.15 Schematic block diagram of LVDS transmitting circuit of Stratix V

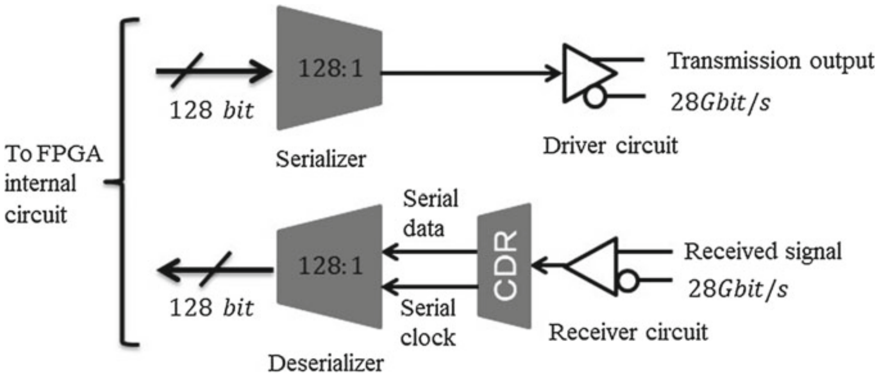


Fig. 8.16 Stratix V 28-Gbps transmitting circuit

The Stratix has an embedded hard macro of a serializer/deserializer between 128 and 1 bit, so that 128-bit parallel data (provided via FIFO) is converted into a 1-bit high-speed serial signal by the hard macro of the serializer. Finally, the signal is transmitted via a driver circuit in the same manner as the LVDS, previously described in Fig. 8.16. Similarly, at the receiving side, the deserializer parallelizes the received 28-Gbps high-speed serial signal into 128 bits to pass it through low-speed FIFO. An error-free receiver circuit includes a clock data recovery (CDR) circuit which can detect a phase shift between the internal clock and received data and can correct it continually. Xilinx also supplies FPGAs that support such high-speed serial communications. For example, the Virtex-7 HT FPGA has 28-Gbps communication ports, which provide excellent communication performance.

8.4.3 FPGA with 120-Gbps Optical I/O

As described above, a transfer rate of the order of Gbps can be implemented even with metal wiring. However, optical communications are beneficial in terms of power

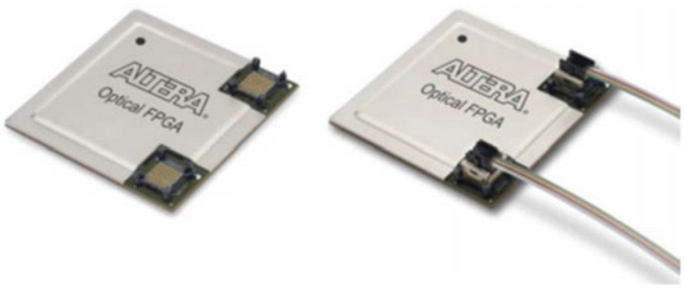


Fig. 8.17 Appearances of 0.7424-mm pitch LGA sockets mounted on Stratix IV package (left) with MicroPOD optical modules (Avago Technologies Ltd.) (right)

consumption for a distance of 10 m or more, as indicated in a report by Altera. Optical modules have been adopted by Xilinx and Altera and have been mounted on FPGA boards in recent years to support optical communications on the board level. However, Altera and Avago Technologies developed and announced a more pioneering optical FPGA with an optical communication interface mounted on an FPGA chip in March, 2011 [47]. Although it is only a trial chip, and no plans for marketing have been announced, its advanced architecture is introduced hereafter.

This optical FPGA is prototyped based on Stratix IV GT FPGA with 11.3-Gbps I/Os for high-speed communications. The salient difference in this optical FPGA from conventional FPGAs is that two of the four corners on its package are provided with sockets of a 0.7424-mm pitch land grid array (LGA), as presented in Fig. 8.17: one for transmission and the other for reception. Each socket is plugged with a dedicated optical module for optical communications supplied by Avago.

This Stratix IV GT FPGA has 32 full-duplex I/O ports for high-speed communications, 12 of which are allocated to these optical I/Os. Twelve 11.3 Gbps high-speed serial I/Os on the FPGA are connected to the sockets for transmission, and 12 11.3 Gbps high-speed serial I/Os are connected to the sockets for reception. The optical communication module is as small and compact as 8.2 mm $\times$ 7.8 mm, as illustrated in Fig. 8.18.

Twelve vertical cavity surface emitting lasers (VCSELs) are embedded in optical communication modules of the transmitting side, whereas 12 GaAs PIN photodiodes are mounted in the optical communication module of the receiving side. Optical communications are conducted through a 12-core fiber cable. The VCSEL lasers can be aligned in two dimensions like common transistors on an integrated circuit. The VCSEL can build a compact laser array [48]. This optical module allows 10.3125-Gbps data transfer per channel consisting of one VCSEL and one GaAsPIN photodiode. In all, 12 channels in the module realize an overall transmission speed of 120 Gbps. In spite of such high-speed communications, a multimode fiber of OM4 grade accommodates long-distance transmissions as far as 150 m. It is highly likely that such optoelectronics will become indispensable when ultra-high-speed I/Os over 28 Gbps become necessary in the future.

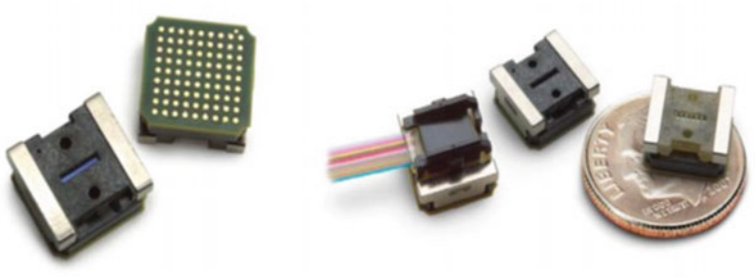


Fig. 8.18 A MicroPOD optical module is mounted on an FPGA package with an LGA socket. Its packaging area is 8.2 mm × 7.8 mm

8.4.4 *Optically Reconfigurable Architecture*

Optically Reconfigurable Architecture by Caltech: Caltech announced an optically reconfigurable gate array (ORGA) using a holographic memory in April, 1999 [49]. This ORGA, the world's first FPGA that can be reconfigured optically, consists of a holographic memory, a laser array, a photodiode array, and an FPGA component. Because its gate array component has a fine-grained gate array structure that is identical to that of conventional FPGAs, its fundamental function appears to be the same as that of the existing FPGAs to device users. However, unlike conventional FPGAs, its programming method is optical reconfiguration. The holographic memory of this optically reconfigurable gate array is used as a read only memory (ROM). Multiple circuit information can be stored in the holographic memory in advance. Then, this circuit information is addressed by the laser array. It is read out as a two-dimensional diffraction pattern. This diffraction pattern is then recognized by the photodiode arrays, transferred serially to the FPGA, and reconfigured. Studies at Caltech have demonstrated the benefits of this optically reconfigurable gate array: It can use large-scale properties of the holographic memory, it can carry multiple circuit information, and its circuit information is programmable within 16–20 μs (Fig. 8.19).

8.4.5 *Japanese-Made ORGA*

Research on ORGAs was also started at Kyushu Institute of Technology in Japan in January 2000. The research base was later moved to Shizuoka University. The research is still in progress. Since Caltech has reported no research on optically reconfigurable devices since, Japan is presumably the only research base for ORGAs in the world at present. Japanese ORGAs under development are introduced hereafter.

Several types of ORGAs are undergoing research and development in Japan, including ORGAs that adopt an electrically rewritable spatial light modulation

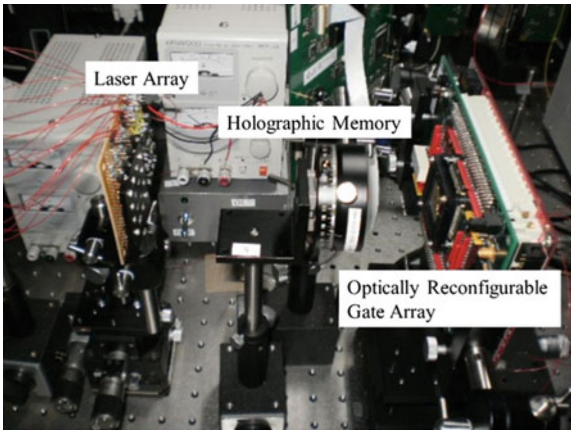


Fig. 8.19 Optically reconfigurable gate array (Shizuoka University)

element as a holographic memory [50, 51] and ORGAs that employ a laser array and microelectromechanical systems (MEMSs) together to address a holographic memory [52]. An ORGA of a simple architecture, similar to that by Caltech, consisting of a holographic memory, a laser array, and a gate array VLSI is introduced here.

ORGAs under development in Japan adopt a fine-grained gate array like Caltech's ORGA, so that the function of the gate array is the same as the existing FPGAs. However, Japanese devices employ a fully parallel configuration, different from Caltech's, where the gate array has many photodiodes. Two-dimensional light patterns generated by the holographic memory are read in a fully parallel mode by these photodiodes. This optical reconfiguration approach allows dynamic reconfiguration of the gate array in a cycle of 10 ns using large amounts of circuit information stored in advance in the holographic memory. To date, ORGAs with circuit information of 256 types have been developed.

An ORGA stores circuit information in a holographic memory. Theoretically, a holographic memory can store as much as 1 Tbit of information within a volume of one lump of sugar. So, its high capacity is expected to be promising also for the next-generation optical memories [53]. The aim of the ORGA is to implement a virtual large-scale gate array by storing much circuit information in a holographic memory using its high capacity [54, 55].

A holographic memory has no fine structures as it is the case for those of existing SRAMs, DRAMs, or ROMs. It can be made simply by consolidating materials such as photopolymers. Accordingly, its production is extremely simple and inexpensive. Information is written on a holographic memory with a dedicated writer using the interference of light. The writer splits a coherent laser beam into two optical paths, an object light representing the binary pattern of circuit information and a reference light, and records the interference pattern of these two light waves on the holographic memory. Greater amounts of information can be recorded by varying the incident angle of a reference light and the irradiation position on a hologram. The stored information can be read out using a laser beam with the identical coherent light as the reference light. In the case of an ORGA, circuit information is usually written in with a writer before the device starts to operate. Its holographic memory is used as a ROM while the device is in operation. Because a large amount of circuit information can be stored in a holographic memory, it is possible to select it with a laser array and to dynamically conduct the reconfiguration.

The holographic memory has a characteristic that it can be used even if it has been contaminated by impurities or partial defects. Holographic memory is usually irradiated with a coherent laser beam as a reference light when reading information. This light undergoes phase modulation or amplitude modulation in the holographic memory and is read out from it. The intensity of light at an arbitrary point is determined by the phase of the gathered light from the whole holographic memory. A collection of lights in phase brightens the point, whereas a collection of lights of diverse phases darkens it. Because information is read out by the superposition of many light waves, the holographic memory has long been known as a robust memory that is useful even if it has defects. Research on radiation-hardened ORGAs is in progress using this characteristic of robustness of the holographic memory. The optoelectronic device has not been used widely yet. However, it might overcome obstacles that are difficult to resolve solely by using integrated circuits in the far future.

References

1. R. Tessier, K. Pocek, A. DeHon, Reconfigurable computing architectures, in *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–351 (March 2015)
2. K. Masuyama, Y. Fujita, H. Okuhara, H. Amano, A 297MOPS/0.4 mW ultra low power coarse-grained reconfigurable accelerator CMA-SOTB-2, in *Proceedings of The 10th International Conference on Reconfigurable Computing and FPGAs (ReConFig)* (Dec 2015)
3. J. Yao, Y. Nakashima, N. Devisetti, K. Yoshimura, T. Nakada
4. X.-P. Ling, H. Amano, WASMII: a data driven computer on a virtual hardware, in *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 33–42 (April 1993)
5. T. Toi, T. Awashima, M. Motomura, H. Amano, Time and space-multiplexed compilation challenge for dynamically reconfigurable processors, in *Proceedings of the 54th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 31–39 (Aug 2011)
6. T.E. Williams, M.E. Dean, D.L. Dill, Efficient self-timing with level-encoded 2-phase dual-rail(ledr), in *Proceedings University of California/Santa Cruz Conference Advanced Research*, VLSI (1991)
7. R. Payne, Self-timed FPGA systems, in *Proceedings International, Workshop Field Program Logic, Applications* (1995)
8. V. Akella, K. Maheswaran, PGA-STC: programmable gate array for implementing self-timed circuits. Int. J. Electron. **84**(3) (1998)
9. M. Kameyama, Y. Komatsu, M. Hariyama, Anasynchronous high-performance FPGA based on LEDR/four-phase-dual rail hybrid archicture. Proc. 5th Int. Symp. HEART (2014)
10. R. Manohar, Reconfigurable asynchronous logic, in *Proceedings IEEE Custom Integrated, Circuits Conference* (2006)
11. R. Manohar, J. Teifei, An asynchronous dataflow FPGA architecture. IEEE Trans. Comput. **53**(11) (2004)
12. M. Hariyama, M. Kameyama, S. Ishihara, Z. Xie, Evaluation of a self-adaptive voltage control scheme for low-power FPGA. J. Semicond. Tech. Sci. **10**(3) (2010)

13. M. Kameyama, S. Ishihara, M. Hariyama, A low-power FPGA based on autonomous fine-grain power gating. IEEE Trans. VLSI Syst. **19**(8) (2011)
14. Achronix SpeedSter22 HP (2011), http://www.achronix.com/products/speedster22ihp.html
15. B. Devlin, M. Ikeda, K. Asada, A 65 nm gate-level pipelined self-synchronous FPGA for high performance and variation robus operation. IEEE J. Solid-State Circuits **46**(11) (2011)
16. B. Devlin, M. Ikeda, K. Asada, A gate-level pipelined 2.97 GHz self synchronous FPGA in 65 nm FPGA CMOS. Prof. ASP-DAC (2011)
17. M. Kameyama, Y. Komatsu, H. Hariyama, An asynchronous high-performance FPGA based on LADR/four-phase-dual-rail hybrid architecture. Proc. HEART (2014)
18. Y. Tsuchiya, M. Komatsu, H. Hariyama, M. Kameyama, R. Ishihara, Implementation of a low-power FPGA based on synchronous/asynchronous hybrid architecture. IEICE Trans. Electron. **E94-C**(10) (2011)
19. A. Bardsley, Implementation balsa handshake circuits. Ph.D. Thesis (Eindhovan Universithy of Technology, 1996)
20. M. Roncken, R. Saeijs, F. Schalij, K. Berkel, J. Kessels, The VLSI programming language trangram and its translation into handshake circuits, in *Proceedings European Conference in Design Automation, EDAC* (1991)
21. M. Kameyama, Y. Komatsu, H. Hariyama, Architecture of an asynchronous FPGA for handshake-component-based design. IEICE Trans. Fund. **E88-A**(12) (2005)
22. A.W. Topol, D.C. La Tulipe, L. Shi, D.J. Frank, K. Bernstein, S.E. Steen, A. Kumar, G.U. Singco, A.M. Young, K.W. Guarini, M. Ieong, Three-dimensional integrated circuits. IBM J. Res. Develop. **50**(4), 5 (2006)
23. G. Katti, A. Mercha, J. Van Olmen, C. Huyghebaert, A. Jourdain, M. Stucchi, M. Rakowski, I. Debusschere, P. Soussan, W. Dehaene, K. De Meyser, Y. Travaly, E. Beyne, S. Biesmans, B. Swinne, 3D stacked ICs using Cu TSVS and die to wafer hybrid collective bonding. IEEE Int. Electron Dev. Meeting IEDM (2009)
24. K. Banerjee, S.J. Souri, P. Kapur, K.C. Saraswat, 3-D ICs: a novel chip design for improving deep-submicrometer interconnect performance and sisytes-on-chip intergration. Proc. IEEE **89**(5) (2001)
25. M. Lin, A. El Gamal, Y.-C. Lu, S. Wong, Performance benefits of monolithically stacked 3-D FPGA. IEEE Trans. Comput. Aided Design Integr. Circuits Syst. **26**(2) (2007)
26. R. Le, S. Reda, R. Iris Bahar, High-performance, const-effective heterogeneous 3D FPGA Architectures, in *Proceedings the 19th ACM Great Lake Symposium VLSI* (2000)
27. T. Naito, T. Ishida, T. Onoduka, M. Nishigoori, T. Nakayama, Y. Ueno, Y. Ishimoto, A. Suzuki, W. Chung, R. Madurawe, S. Wu, S. Ikeda, H. Oyamatsu, World's first monolithic 3D-FPGA with ifi SRAM over 90 nm 9 layer Cu CMOS, in *Proceedings Symposium VLSI Technology* (2010)
28. Y.Y. Liauw, Z. Zhang, Z. Zhang, W. Kim, A.E. Gamal, S.S. Wong, Nonvolatile 3D-FPGA with monolithically stacked RRAM-based configuration memory. ISSCC (2012)
29. A. Gayasen, V. Narayanan, M. Kandemir, A. Rahman, Designing a 3-D FPGA: switch box architecture and thermal issues. IEEE Trans. VLSI Syst. **16**(7) (2008)
30. F. Furuta, T. Matsumura, K. Osada, M. Aoki, K. Hozawa, K. Takeda, N. Miyamoto, Scalable 3D-FPGA using wafer-to-wafer TSV interconnect of 15 Tbps/w, 33 Tbps/mm^2. IEEE Trans. VLSI Syst. (2013)
31. M.J. Alexander, J.P. Cohoon, J.L. Colflesh, J. Karro, G. Robins, Three-dimensional field-programmable gate arrays, in *Proceedings of 8th Annual IEEE International ASIC Conference and Exhibit* (1995)
32. S.A. Razavi, M.S. Zamani, K. Bazargan, A tileable switch module architecture for homogeneous 3D FPGAs, in *Proceedings IEEE International 3D System Integration* (2009)
33. A. Rahman, S. Das, A.P. Chandrakasan, R. Reif, Wiring requerement and three-dimensional integration technology for field programmable gate arrays. IEEE Trans. VLSI Syst. **11**(1) (2003)
34. C. Ababei, H. Mogal, K. Bazargan, Three-diminsional place and route for FPGAs. IEEE Trans. Comput. Aided Design Integr. Circuits Syst. **25**(6) (2006)

35. M. Amagasaki, Y. Takeuchi, Q. Zhao, M. Iiea, M. Kuga, T. Sueyoshi, Architecture exploration of 3D FPGA to minimize internal layer connection, in *ACM/IEEE International Conference on 3D Systems Integration* (2015)
36. M.J. Alexander, J.P. Cohoon, J.L. Colflesh, J. Karro, E.L. Peters, G. Robins, Placement and routing for three-dimensional FPGAs, in *4th Canadian Workshop Field Programmable Devices* (1996)
37. M. Lin, A. El Gamal, A routing fabric for monolithically stacked 3D FPGA, in *Proceedings ACM/IEEE International Conference on FPGA* (2007)
38. N. Miyamoto, Y. Matsuomto, H. Koike, T. Matsumura, K. Osada, Y. Nakagawa, T. Ohmi, Development of a CAD tool for 3D-FPGAs, in *IEEE International Conference on 3D Systems Integration* (2010)
39. Y. Kwon, P. Lajevardi, A.P. Ch, D.E. Troxel, A 3-D FPGA wire resource prediction model validated using a 3-D placement and routing tool, in *Proceedings of SLIP '05* (2005)
40. A. Putnam, et al., A reconfigurable fabric for accelerating large-scale datacenter services, in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 13–24 (2014)
41. The Telecommunications Industry Association (TIA), Electrical characteristics of low voltage differential signaling (LVDS) interface circuits, PN-4584 (May 2000)
42. National Semiconductor: RSDS Intra-panel Interface Specification (May 2003)
43. Texas Instruments, mini-LVDS Interface Specification (2003)
44. Altera Corporation: Stratix IV Device Handbook, vol. 1 (June 2015)
45. Altera Corporation: Stratix V Device Handbook, vol. 1 (June 2015)
46. Altera Corporation: High speed board design Ver.4.0, Application Note 75 (Nov 2001)
47. M. Peng Li, J. Martinez, D. Vaughan, Transferring high-speed data over long distances with combined FPGA and multichannel optical modules (2012)
48. H. Li, K. Iga, Vertical-cavity surface-emitting laser devices, in *Springer Series in Photonics*, vol. 6 (2003)
49. J. Mumbra, D. Psaltis, G. Zhou, X. An, F. Mok, Optically programmable gate array (OPGA). Opt. Comput. (1999)
50. H. Morita, M. Watanabe, Microelectromechanical configuration of an optically reconfigurable gate array. IEEE J. Quant. Electron. **46**(9), 1288–1298 (Sept 2008)
51. Y. Yamaguchi, M. Watanabe, Liquid crystal holographic configurations for ORGAs. Opt. Comput. **47**(28), 4692–4700 (2008)
52. Y. Yamaji, M. Watanabe, A 4-configuration-context optically reconfigurable gate array with a MEMS interleaving method, in *NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 172–177 (June 2013)
53. A. Ogiwara, M. Watanabe, Optical reconfiguration by anisotropic diffraction in holographic polymer-dispersed liquid crystal memory. Appl Opt **51**(21), 5168–5188 (July 2012)
54. H.J. Coufal, D. Psaltis, G.T. Sincerbox, Holographic data storage, in *Springer Series in Optical Sciences*, vol. 76 (2000)
55. S.-L.L. Lu, P. Yiannacouras, R. Kassa, M. Konow, T. Suh, An FPGA-based Pentium in a complete desktop system, in *ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, pp. 53–59 (2007)