# Chapter 7
# Basics of Field-Programmable Gate Array

**Yota Yamamoto**

**Abstract** A field-programmable gate array (FPGA) is a large-scale integration (LSI) that enables the user to modify the internal circuit structure. Central processing units (CPUs) are also implemented on LSIs and have one or more arithmetic logic units (ALUs). FPGAs, however, can have tens or hundreds of thousands of ALU-equivalent arithmetic cores using their on-board logic resources. CPU ALUs can operate as fast as 3 GHz, whereas FPGAs are nearly an order of magnitude slower at around 500 MHz. To build a high-speed special-purpose computer using FPGAs, we must select suitable algorithms that have less dependence on data, employ low precision, and are easily parallelizable. Effective parallel computation can be attained by taking advantage of the FPGAs' plentiful arithmetic units.

## 7.1 Structure of Field-Programmable Gate Array

**Field-programmable gate arrays** (**FPGA**s) are large-scale integrations (LSIs) that enable the user to modify the internal circuit structure. Figure 7.1 reveals a typical FPGA structure. Their internal circuits, unlike CPUs and graphics processing units (GPUs), are not functionally connected, and they work by loading precise circuit configuration data upon launch. The circuit configuration data configure the different blocks in the FPGA, like the programmable logic blocks (LBs) that implement logic circuits, programmable input-output blocks (IOBs) that offer the interface to external circuits, and programmable routing blocks [connection blocks (CBs) and switching blocks (SBs)], which connect each block. Inside the FPGA, these elements are arranged in a grid.

**Supplementary Information** The online version contains supplementary material available at https://doi.org/10.1007/978-981-99-1938-3_7.

Y. Yamamoto (✉)
Faculty of Engineering, Tokyo University of Science, 6-3-1 Niijuku, Katsushika-ku, Tokyo 125-8585, Japan
e-mail: yy-yamamoto@rs.tus.ac.jp

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2023

T. Shimobaba and T. Ito (eds.), *Hardware Acceleration of Computational Holography*,
https://doi.org/10.1007/978-981-99-1938-3_7

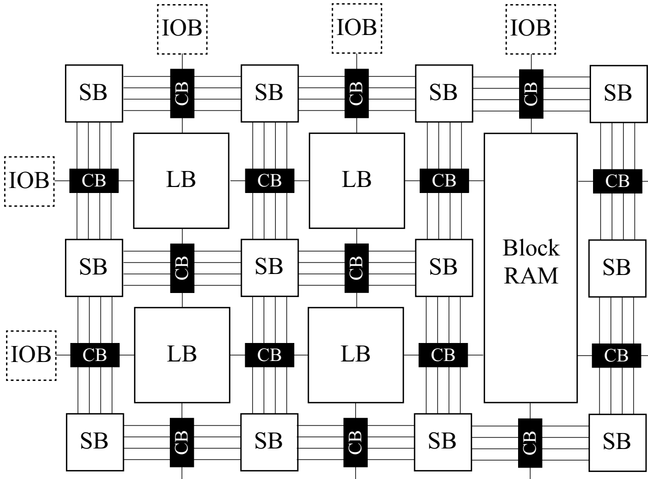



**Fig. 7.1** Structure of FPGA

LBs are based on the lookup table (LUT) and multiplexer (MUX) cells to implement certain logic functions. The LB's name differs among FPGA vendors: Xilinx calls it a configurable LB (CLB) [1], and Intel calls it a logic array block (LAB) [2]. Furthermore, even if it is from the same vendor, the internal structure of the LB varies depending on the family.

There are two primary kinds of LBs: hard logic and soft logic. Hard logic includes a **digital signal processor** (**DSP**) [3] and block RAM [4]. Although it lacks the flexibility of LUT-based soft blocks, it can run predetermined logic functions at a high speed. Soft logic, which comprises LUTs and so on, fulfills any logic functions that are unavailable in hard logic.

The primary difference between CPUs and FPGAs is the arithmetic units' number. CPUs have one or more arithmetic logic units (ALUs), whereas FPGAs can have tens or hundreds of thousands of ALU-equivalent arithmetic cores using their on-board logic resources. CPU ALUs can perform as fast as 3 GHz, while FPGA ALUs are nearly an order of magnitude slower at around 500 MHz.

## 7.2 Hardware Description Language (HDL)

The circuit configuration data are produced by compiling the source code written by **hardware description languages** (**HDLs**) using a tool offered by FPGA vendors (Fig. 7.2). First, the HDL is transformed into an intermediate code called a netlist using a process called **logic synthesis**. The netlist is then mapped to the physical pin assignments and LBs of the actual device by a process called implementation, and the circuit configuration data are produced [5].

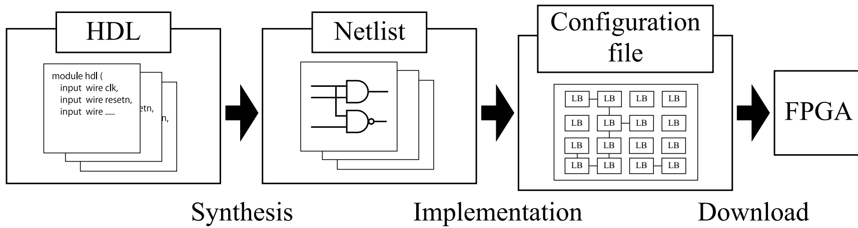
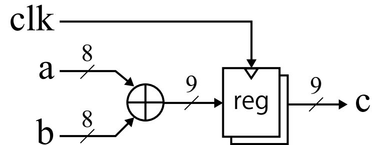



**Fig. 7.2** Programming flow of FPGA

**Fig. 7.3** Block diagram of the adder circuit




There are different types of HDLs, and there are compilers called high-level synthesis tools that produce HDL from abstract descriptions in C language [6]. In this section, we shortly present the **VHDL** [7] and **SystemVerilog** [8]. Listings 7.1 and 7.2 reveals the source code of an adder circuit using VHDL and SystemVerilog, and Fig. 7.3 reveals the block diagram.

In HDL, it is possible to describe the logic circuit to be implemented in FPGA using addition and subtraction of variables, conditional branching, and so on, just as in C programming. However, it is crucial to note that the operations of CPUs and FPGAs are very different. Software programming, including C programming, describes how the CPU operates, and the process is run sequentially. However, FPGA hardware programming explains the logic circuits' structure. All the illustrated logic circuits operate simultaneously.

The defining of input and output signals is the first step in both VHDL and SystemVerilog. The circuit is synchronized with "clk," which is a clock signal that repeats "0" and "1," The circuit conducts the addition of the input values of "a" and "b." The bit width of the CPU and GPU is fixed, whereas that of the FPGA can be freely determined by the user.

**Listing 7.1** Source code for adder circuit using VHDL

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5
6  entity adder_vh is
7      generic (
8          INPUT_WIDTH : integer := 8
9      );
10     port (
11         clk : in std_logic;
```

```
12          a : in std_logic_vector(INPUT_WIDTH−1 downto 0);
13          b : in std_logic_vector(INPUT_WIDTH−1 downto 0);
14          c : out std_logic_vector(INPUT_WIDTH−1+1 downto 0)
15      );
16  end adder_vh ;
17
18  architecture rtl of adder_vh is
19      signal add : std_logic_vector(INPUT_WIDTH−1+1 downto 0);
20  begin
21      c <= add;
22
23      process (clk)
24      begin
25          if clk'event and clk = '1' then
26              add <= ('0'&a) + ('0'&b);
27          end if;
28      end process;
29
30  end architecture;
```

**Listing 7.2** Source code for adder circuit using SystemVerilog

```
1   module adder_sv #(
2       parameter int INPUT_WIDTH = 8
3   )(
4       input wire clk,
5       input wire [INPUT_WIDTH−1:0] a,
6       input wire [INPUT_WIDTH−1:0] b,
7       output wire [INPUT_WIDTH−1+1:0]c
8   );
9
10      logic [INPUT_WIDTH−1+1:0] add;
11
12      assign c = add;
13
14      always_ff @(posedge clk) begin
15              add <= a + b;
16      end
17
18  endmodule
```

## 7.3 Special-Purpose Computation Circuit Using FPGA

To build a high-speed special-purpose computer using FPGAs, it is a must to use tens to hundreds of thousands of arithmetic units. However, FPGAs are about one order of magnitude slower than CPUs in terms of operating frequency, and it is crucial to consider effective data flow to build a high-speed special-purpose computer using FPGAs.

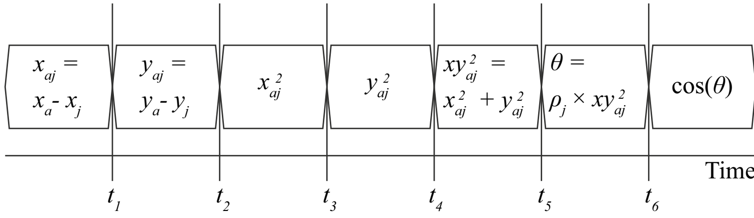



**Fig. 7.4** Sequential execution of Eq. 7.1

There are two important factors for efficient computation: throughput and latency. **Throughput** is the processing capacity per unit of time. **Latency** is the delay time needed for each process. By enhancing throughput and latency, faster computation becomes possible.

To enhance throughput and latency, pipeline and data parallelization can be employed. To explain this, we consider the implementation of Eq. 7.1 to compute a computer-generated hologram (CGH):

$$I\left(x_a, y_a\right) = \sum_{j=0}^{M-1} \cos\left[\rho_j\left\{\left(x_a - x_j\right)^2 + \left(y_a - y_j\right)^2\right\}\right], \tag{7.1}$$

where$(x_a, y_a)$ represents a coordinate on the CGH plane, $\rho_j = \pi/2\lambda z_j$, $(x_j, y_j, z_j)$ are the coordinates of the 3D object's point cloud, $M$ denotes the point-cloud number, and $\lambda$ represents the reference light's wavelength.

For a sequential computation on a CPU, the computation inside $\Sigma$ in Eq. 7.1 is shown in Fig. 7.4.

For example, we consider the computation time $t$ [s] for 1,024 × 1,024-pixel CGH from $M = 100$ object points at the latency shown in Fig. 7.4. Assuming that each operation is run at 250 MHz (4 ns), the computation time is

$$t = \frac{1}{250\ \text{MHz}} \times 7 \times 100 \times 1{,}024 \times 1{,}024 = 2.94\ \text{s}. \tag{7.2}$$

Since $x_{aj}$ and $y_{aj}$ are independent of each other, the computations for them can be parallelized as illustrated in Fig. 7.5. Here, the latency is reduced from 7 to 5, and the computation time can be lowered to

$$t = \frac{1}{250\ \text{MHz}} \times 5 \times 100 \times 1{,}024 \times 1{,}024 = 2.10\ \text{s}. \tag{7.3}$$

Although we have focused on the computation of only a single CGH pixel, the CGH computation can be parallelized for each CGH pixel. Figure 7.6 reveals the five-step computation in Fig. 7.5 parallelized for two CGH pixels:

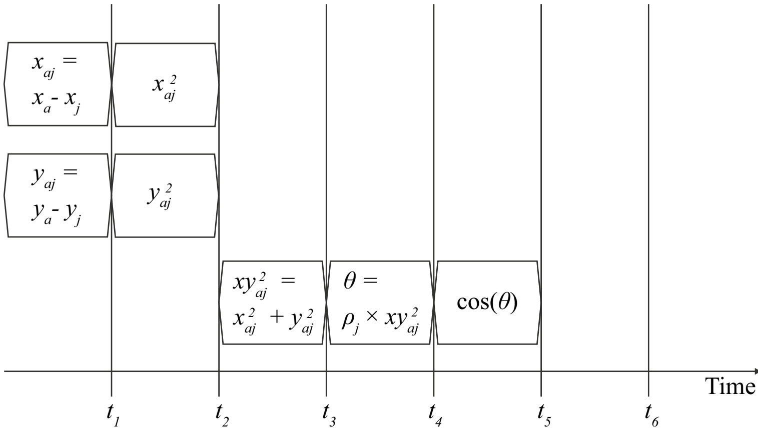



**Fig. 7.5** Parallelization of $x$ and $y$ calculations

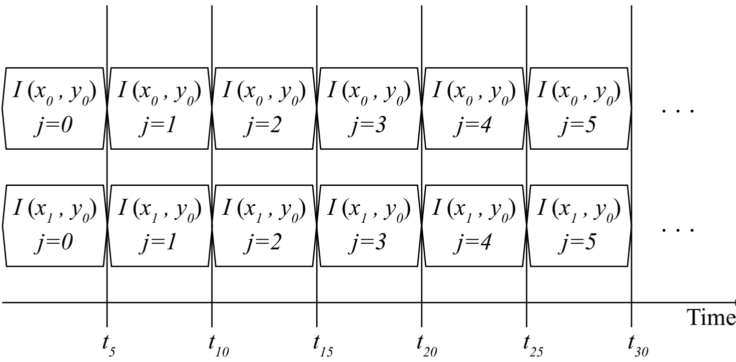



**Fig. 7.6** Pixel-by-pixel parallelization

$$t = \frac{1}{250\text{MHz}} \times 5 \times 100 \times 1{,}024 \times 1{,}024 \div 2 = 1.05 \text{ s}. \tag{7.4}$$

This parallelization approach, which takes advantage of the lack of dependency between data and performs operations in parallel, is called **data parallelization**.

The computation time can be further accelerated using **pipeline parallelization**. Data parallelization is user-controllable not only in FPGAs but also in CPUs and GPUs, whereas pipeline parallelization is a user-controllable parallelization approach only in FPGAs. Here, we denote $x_a - x_j$ and $y_a - y_j$ operations, ${x_{aj}}^2$ and ${y_{aj}}^2$ operations, ${xy_{aj}}^2$ operation, $\theta$ operation, and cos $(\theta)$ operation in Fig. 7.4 as $\text{OP0}_j$, $\text{OP1}_j$, $\text{OP2}_j$, $\text{OP3}_j$, and $\text{OP4}_j$, respectively. In pipeline parallelization, the amount of arithmetic units needed for the entire computation is arranged as illustrated in Fig. 7.7 for Fig. 7.4. Additionally, it is parallelization at the operator level. Here, the latency is the same as that in data parallelization. However, the throughput is enhanced. In

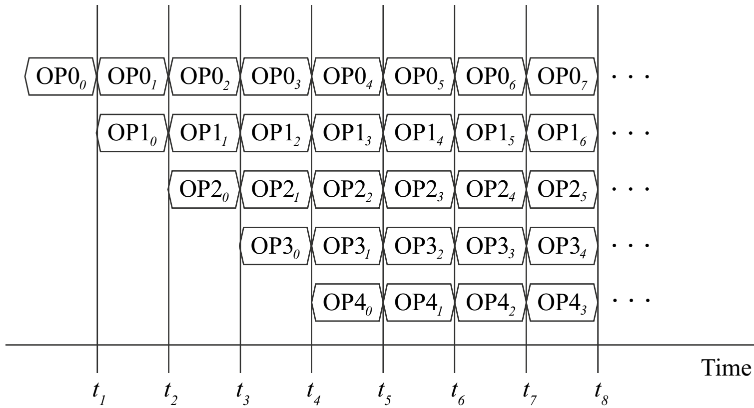



**Fig. 7.7** Pipeline parallelization

the case of data parallelization only, the next data cannot be input to the circuit until all five operations are completed. However, in the case of pipeline parallelization, the following object point data can be input immediately. The computation time can be expressed as follows:

$$t = \frac{1}{250\,\mathrm{MHz}} \times (5 + 100 - 1) \times 1024 \times 1024 = 0.44\,s. \tag{7.5}$$

The computation is nearly five times faster than when neither data parallelization nor pipeline parallelization is employed. Combining pipeline parallelization and data parallelization is also possible. When the two are combined, the computation time in Eq. 7.5 is reduced by the number of parallels. If 10 CGH pixels can be data-parallelized, for example, the computation time can be further accelerated from Eq. 7.5 as

$$t = \frac{1}{250\,\mathrm{MHz}} \times (5 + 100 - 1) \times 1024 \times 1024 \div 10 = 0.044\ \mathrm{s}. \tag{7.6}$$

FPGAs can attain high-speed computation by employing data parallelization and pipeline parallelization, as well as an efficient parallel computation that takes advantage of the reconfigurable resources inside the FPGA. To attain high-speed computation, the algorithm should have less resilience on data, should be able to compute with as low accuracy as feasible, and should be readily parallelized.

## 7.4 Fixed-Point and Floating-Point Arithmetic

**Floating-point arithmetic** are frequently employed in CPUs. Floating-point arithmetic employs exponential representation to denote numerical values, and the IEEE754 [9] standard defines the data format. Although floating-point arithmetic can handle a wide range of values, exponentiation operations are required. However, fixed-point arithmetic is frequently employed for numerical computations in FPGAs. In fixed-point arithmetic, the user places the decimal point's position arbitrarily. Compared with floating-point arithmetic, fixed-point arithmetic has a smaller range of values, but they do not need exponentiation operations and can be computed with simple hardware.

Equation 7.7 is the phase computation part of Eq. 7.1, and we describe how to compute it using fixed-point arithmetic.

$$\theta = \rho_j \left\{ \left(x_a - x_j\right)^2 + \left(y_a - y_j\right)^2 \right\}. \tag{7.7}$$

FPGAs can use any data width, whereas floating-point arithmetic use 32-bit or 64-bit data widths. The smaller the data width, the more resources (gates or transistors) can be employed to construct the arithmetic unit and the more parallelism can be realized.

If $x_a, x_j, y_a, y_j$ in Eq. 7.7 are normalized by the sampling interval of CGH, they are integer values. The normalized values' data width is determined from the minimum and maximum values. Here, $(x_a, y_a)$ is the coordinate on the CGH plane and $x_j$, $y_j$ is the point cloud's coordinate. These coordinates range from -2,048 to 2,047 when using a CGH with 4,096 $\times$ 4,096 pixels; therefore, $x_a$, $x_j$, $y_a$, $y_j$ are denoted by 12 bits. Figure 7.8 reveals the data widths and decimal point positions of fixed-point integer arithmetic. In the fixed-point arithmetic's addition and subtraction between integers, no decimal point change occurs. However, the data width is extended by 1 bit in addition. Also, in multiplication, the sum of the data widths of both operands is extended.

Fixed-point arithmetic in binary numbers is each digit weights units of powers of two as shown in Fig. 7.9. Figure 7.9 reveals an example of an unsigned binary number; in a signed binary number represented in two's complement, the most significant bit's weight is $-2^3$ as in the case of Fig. 7.9.

Figure 7.10 shows the data width of two fixed-point arithmetics and how multiplication moves the decimal point. The multiplication of integer and decimal fraction fixed-point arithmetics can also be computed in a straightforward manner. However, the decimal point is shifted, and the decimal point's position in the computation finding $\theta$ becomes the 32nd bit position.

Here, $\theta$ is represented as a fixed-point number with a 32-bit decimal part. Here, the decimal part's minimum value is 0.000000000232 ($2^{-32}$). In other words, we must treat $\theta$ as estimated values with some error. This error is known as quantization error. The quantization error may have a large influence on some computations, so it is necessary to assess the effect of the quantization error in advance by simulation.

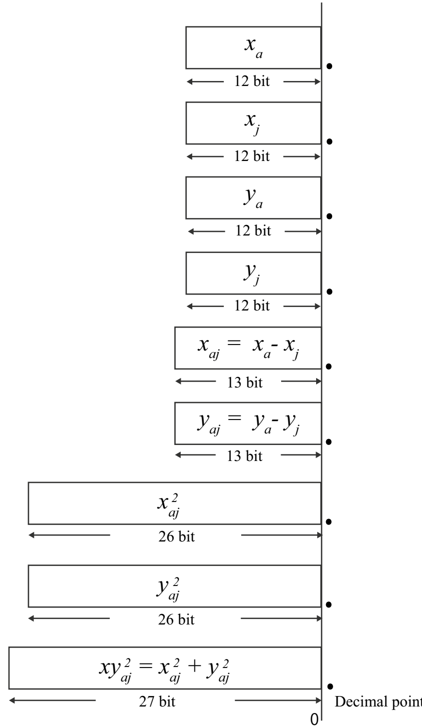



**Fig. 7.8** Integer fixed-point arithmetic

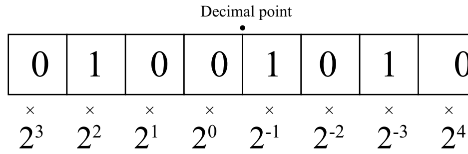



**Fig. 7.9** Fixed-point arithmetic representation. Here, the decimal number is 8.625

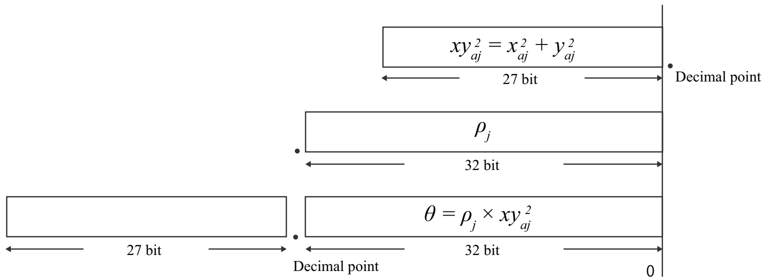



**Fig. 7.10** Fixed-point arithmetic of decimals

## 7.5 Communication Between FPGAs and CPUs

A special-purpose computer using FPGAs is not employed alone but is connected to CPUs (FPGA embedded or on a PC) that send and pre- and post-process the data. There are different communication protocols, including Universal Serial Bus (USB) and PCI Express. In Xilinx FPGAs, the **advanced extensible interface** (**AXI**) [10] is employed for communication between a CPU (hard logic embedded on an FPGA) and programmable logic (Fig. 7.11) [11]. Even if the FPGA is communicated with a host PC through PCI Express or Zynq [12, 13] with a built-in CPU, we can employ AXI communication by developing auxiliary circuits.

## 7.6 AXI Communication

AXI is an inter-module communication protocol created by ARM Ltd [10]. There are three AXI communications: AXI(-Full), AXI-Lite, and AXI-Stream. AXI Lite is employed for small-scale data communication (e.g., control signals), whereas AXI(-Full) and AXI-Stream are used for large-scale data communication.

A circuit that requests data is called a "requester," and a circuit that sends and receives data in response to the request is called a "responder." The requester retains complete control over the data's transmission and reception. In AXI(-Full) and AXI-Lite, the data may be sent from the requester to the responder or from the responder to the requester in this chapter. AXI-Stream always sends data from the requester to the responder. Figure 7.12 reveals a diagram of the basic communication.

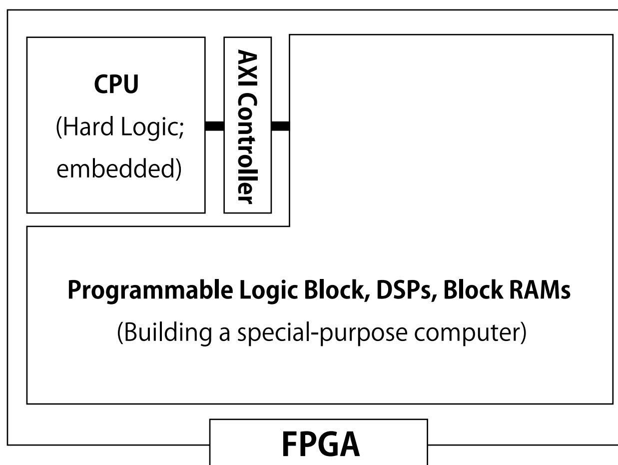



**Fig. 7.11** Outline of the circuit connected by AXI

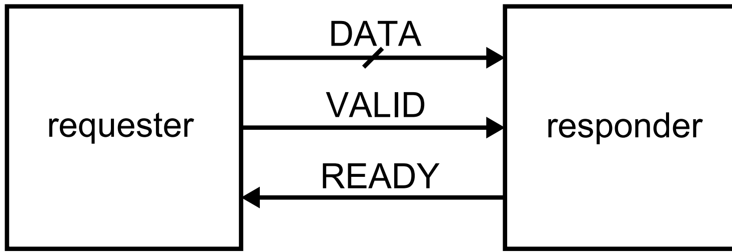



**Fig. 7.12** AXI basic communication

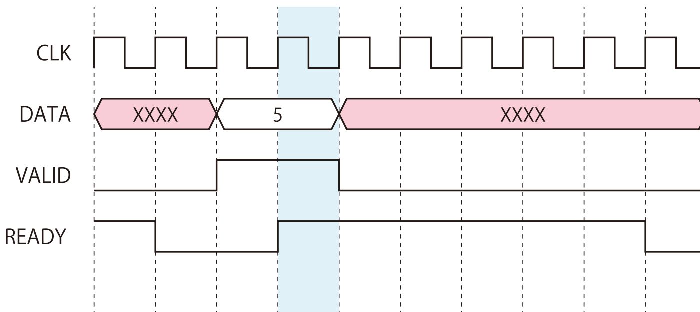



**Fig. 7.13** The VALID-READY communication

When both VALID and READY of AXI signals are set to 1, the transfer is complete. The form of communication in which VALID shows that valid data are being introduced and READY illustrates that the data can be received is known as VALID-READY communication. The AXI protocol can employ several channels based on VALID-READY communication to improve communication capacity (Fig. 7.13).

Figure 7.14 shows a block diagram of a communication circuit using the AXI protocol (the signal's description can be found in Tables 7.1 and 7.2). In Fig. 7.14, regulating to write and read data are implemented as state machines, which transition the internal state depending on the input and current state. Figure 7.15 reveals the state transition diagrams for reading and writing data. Listing 7.3 indicates the implementation of Fig. 7.13 written by SystemVerilog.

The READ state machine, which is the data read from a CPU to an FPGA, comprises R_IDLE (read wait state) and R_READ (read response). After the start (e.g., assertion of the reset signal), the circuit begins in the R_IDLE state. A transition is made to the R_READ state when the signal S_AXI_ARVALID, which shows that a valid address is an output from the requestor (CPU), becomes 1. In the R_READ state, the FPGA maintains the signal S_AXI_RVALID as 1, indicating that it is outputting valid data, and returns to the R_ILDE state after receiving the read response (S_AXI_BVALID set as 1).

The WRITE state machine, which is the data written from the CPU to the FPGA, comprises the W_IDLE state, which is the write wait state, and the W_RESP state,

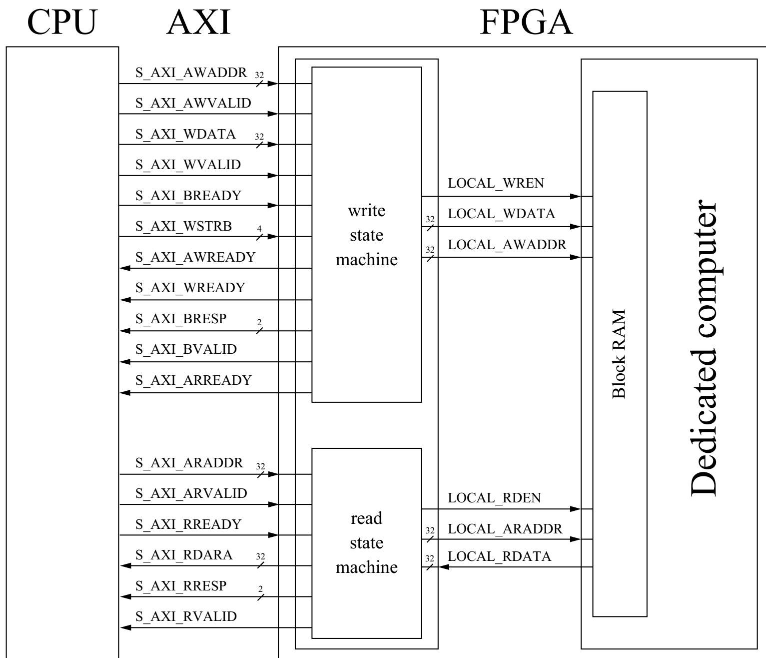



**Fig. 7.14** Block diagram of the communication circuit (excluding clock and reset signals). The shaded lines on the signal lines in the figure show the bit width

**Table 7.1** Description of AXI signals

| Signal name | Description |
|---|---|
| S_AXI_AWADDR | Write start address |
| S_AXI_AWVALID | Write address valid |
| S_AXI_WDATA | Write data |
| S_AXI_WVALID | Write data valid |
| S_AXI_BREADY | Acceptable |
| S_AXI_WSTRB | Byte enable |
| S_AXI_AWREADY | Write address can be accepted |
| S_AXI_WREADY | Writing can be accepted |
| S_AXI_BRESP | Write response |
| S_AXI_BVALID | Write response enabled |
| S_AXI_ARREADY | Readable address can be accepted |
| S_AXI_ARADDR | Read start address |
| S_AXI_ARVALID | Read address valid |
| S_AXI_RREADY | Read data can be accepted |
| S_AXI_RDATA | Read data |
| S_AXI_RRESP | Read response |
| S_AXI_RVALID | Read data valid |

**Table 7.2** Description of local signals. These signals are defined by the author

| Signal name | Description |
|---|---|
| LOCAL_WREN | Write data and address valid |
| LOCAL_WDATA | Write data |
| LOCAL_AWADDR | Write start address |
| LOCAL_RDEN | Read data and address response |
| LOCAL_ARADDR | Read start address |
| LOCAL_RDATA | Read data |

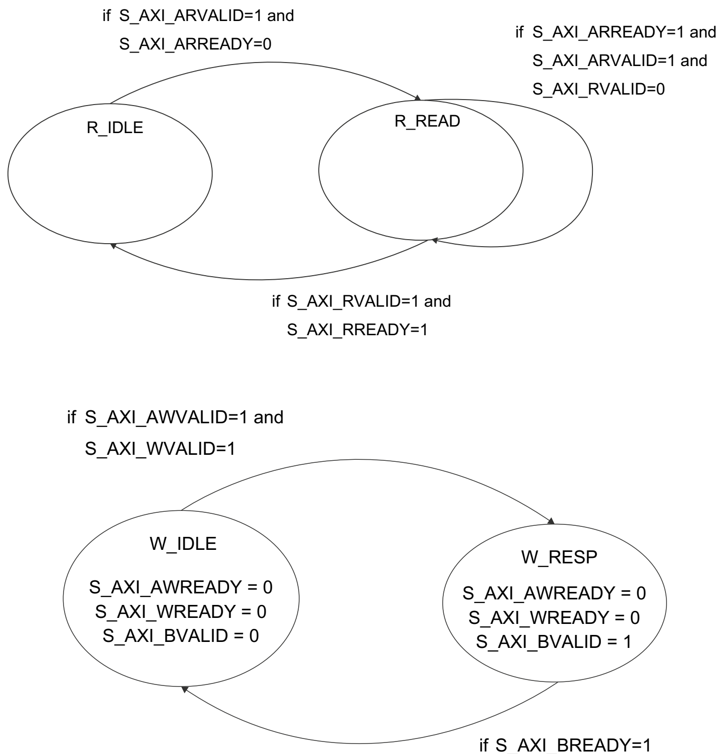



**Fig. 7.15** State transition diagram for AXI Lite. The upper and bottom figures show READ and WRITE state machines, respectively

which is the write response state. The state machine starts in W_IDLE after initialization. When both the signal S_AXI_AWVALID, showing that the address is generating a valid value, and the signal S_AXI_WVALID, indicating that the data are valid, are set to 1 by the requestor (CPU), a transition to the W_RESP state happens. The FPGA returns to the W_IDLE state after a successful read response in W_RESP.

Listing 7.3 reveals the sample source code for the AXI Lite response side.

**Listing 7.3** Source code for AXI Lite

```
1  module axi_lite_s # (
2    parameter integer C_S_AXI_DATA_WIDTH = 32,
3    parameter integer C_S_AXI_ADDR_WIDTH = 32
4  ) (
5    // Users to add ports here
6    output wire       local_wren,
7    output wire [C_S_AXI_DATA_WIDTH-1 : 0]  local_wdata,
8    output wire [C_S_AXI_ADDR_WIDTH-1 : 0]  local_awaddr,
9    output wire       local_rden,
10   input wire [C_S_AXI_DATA_WIDTH-1 : 0]  local_rdata,
11   output wire [C_S_AXI_ADDR_WIDTH-1 : 0]  local_araddr,
12   output wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0]  local_wstrb,
13
14   // Ports of Axi S Bus Interface S_AXI
15   input  wire       s_axi_aclk,
16   input  wire       s_axi_aresetn,
17   input  wire [C_S_AXI_ADDR_WIDTH-1 : 0]  s_axi_awaddr,
18   input  wire [2 : 0]      s_axi_awprot,
19   input  wire       s_axi_awvalid,
20   output  wire       s_axi_awready,
21   input  wire [C_S_AXI_DATA_WIDTH-1 : 0]  s_axi_wdata,
22   input  wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0]  s_axi_wstrb,
23   input  wire       s_axi_wvalid,
24   output  wire       s_axi_wready,
25   output  wire [1 : 0]      s_axi_bresp,
26   output  wire       s_axi_bvalid,
27   input  wire       s_axi_bready,
28   input  wire [C_S_AXI_ADDR_WIDTH-1 : 0]  s_axi_araddr,
29   input  wire [2 : 0]      s_axi_arprot,
30   input  wire       s_axi_arvalid,
31   output  wire       s_axi_arready,
32   output  wire [C_S_AXI_DATA_WIDTH-1 : 0]  s_axi_rdata,
33   output  wire [1 : 0]      s_axi_rresp,
34   output  wire       s_axi_rvalid,
35   input  wire       s_axi_rready
36 );
37
38   localparam  W_IDLE = 2'd0, W_RESP = 2'd1;
39   localparam  R_IDLE = 2'd0, R_READ = 2'd1;
40
41   logic [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_awaddr;
42   logic       axi_awready;
43   logic [C_S_AXI_DATA_WIDTH-1 : 0]  axi_wdata;
44   logic       axi_wready;
45   logic       axi_bvalid;
46   logic [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_araddr;
47   logic       axi_arready;
48   logic       axi_rvalid;
49   logic [(C_S_AXI_DATA_WIDTH/8)-1 : 0] axi_wstrb;
50
```

```
51  logic [1:0] w_state, r_state;
52
53  // I/O Connections assignments
54  assign s_axi_awready = axi_awready;
55  assign s_axi_wready  = axi_wready;
56  assign s_axi_bresp  = 2'b0; // 'OKAY' response
57  assign s_axi_bvalid  = axi_bvalid;
58  assign s_axi_arready = axi_arready;
59  assign s_axi_rresp  = 2'b0; // 'OKAY' response
60  assign s_axi_rvalid  = axi_rvalid;
61
62  always_ff @( posedge s_axi_aclk ) begin
63    if ( s_axi_aresetn == 1'b0 ) begin
64      w_state  <= W_IDLE;
65      axi_awready <= 1'b0;
66      axi_awaddr  <= 0;
67      axi_wready  <= 1'b0;
68      axi_wdata  <= 0;
69      axi_wstrb <= 0;
70      axi_bvalid <= 1'b0;
71    end else begin
72      case ( w_state )
73        W_IDLE: begin
74          if ( ~axi_awready && ~axi_wready && s_axi_awvalid && s_axi_wvalid ) begin
75            axi_awready <= 1'b1;
76            axi_awaddr  <= s_axi_awaddr;
77            axi_wdata  <= s_axi_wdata;
78            axi_wstrb <= s_axi_wstrb;
79            axi_wready  <= 1'b1;
80            w_state  <= W_RESP;
81          end else begin
82            axi_awready <= 1'b0;
83            axi_wready  <= 1'b0;
84            axi_bvalid  <= 1'b0;
85          end
86        end
87        W_RESP: begin
88          if ( s_axi_bready && axi_bvalid ) begin
89            axi_bvalid  <= 1'b0;
90            w_state  <= W_IDLE;
91          end else begin
92            axi_awready <= 1'b0;
93            axi_wready  <= 1'b0;
94            axi_bvalid  <= 1'b1;
95          end
96        end
97        default: begin
98          w_state <= W_IDLE;
99        end
100     endcase
101   end
102 end
103
```

```
104  always_ff @( posedge s_axi_aclk ) begin
105    if ( s_axi_aresetn == 1'b0 ) begin
106      axi_arready <= 1'b0;
107        axi_araddr <= 0;
108      axi_rvalid <= 1'b0;
109      r_state <= R_IDLE;
110    end else begin
111      case ( r_state )
112        R_IDLE: begin
113          if ( ~axi_arready && s_axi_arvalid ) begin
114              axi_arready <= 1'b1;
115              axi_araddr <= s_axi_araddr;
116            r_state <= R_READ;
117            end else begin
118              axi_arready <= 1'b0;
119            axi_rvalid <= 1'b0;
120            end
121        end
122        R_READ: begin
123            if ( axi_arready && s_axi_arvalid && ~axi_rvalid ) begin
124              axi_rvalid <= 1'b1;
125            axi_arready <= 1'b0;
126            end else if ( axi_rvalid && s_axi_rready ) begin
127            axi_rvalid <= 1'b0;
128            axi_arready <= 1'b0;
129            r_state <= R_IDLE;
130          end
131        end
132        default: begin
133          r_state <= R_IDLE;
134        end
135      endcase
136    end
137  end
138
139  assign local_wren = axi_wready && s_axi_wvalid && axi_awready && s_axi_awvalid;
140  assign local_rden = axi_arready && s_axi_arvalid && ~axi_rvalid;
141  assign local_araddr = axi_araddr;
142  assign local_awaddr = axi_awaddr;
143  assign local_wdata = axi_wdata;
144  assign s_axi_rdata = local_rdata;
145  assign local_wstrb = axi_wstrb;
146
147 endmodule
```

## 7.7 Communication Program Between CPU and FPGA

Figure 7.15 shows that the CPU and FPGA are connected to communicate data, and it is crucial to create a dedicated driver. However, since creating a driver is outside the scope of this book, we will present an approach using /dev/mem [15], which

is slow but easy to read and write data from/to an FPGA. As a prerequisite, we examine a situation where a Linux OS, including Petalinux (Linux manufactured by Xilinx) [14], is operating on the CPU embedded in Zynq. In Linux, reading and writing data to a device (here, an FPGA) can be replaced by reading and writing to a special file called /dev/mem. Listing 7.4 shows a sample program.

**Listing 7.4** Source code for AXI Lite

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <sys/mman.h>
6
7  FPGA_ADDR_START=0xA0000000;
8  FPGA_ADDR_SIZE=0x8000;
9
10 int main()
11 {
12     uint32_t *uio;
13     int fd;
14
15     // open "/dev/mem"
16     fd = open("/dev/mem", O_RDWR | O_SYNC);
17     if (fd < 1) {
18         perror("Failed to open devfile");
19         return -1;
20     }
21
22     // map FPGA physical address into user space
23     uio = (uint32_t *)mmap(NULL, FPGA_ADDR_SIZE, PROT_READ|PROT_WRITE,
           MAP_SHARED, fd, FPGA_ADDR_START);
24
25     // write "5" to FPGA
26     uio[0] = 0x5;
27
28     // cleanup
29     munmap((void*)address, 0x1000);
30     close(fd);
31
32     return 0;
33 }
```

Listing 7.4 reveals an example where the start address for writing data to the FPGA is 0xA0000000. This address is determined by vendors (refer to the datasheet for details). The Linux system commands (C language functions) “write” and “read” are employed to send data as if reading and writing to a file.

Using the /dev/mem technique removes the need for building a driver, but it should be noted that this is not a permanent approach from the viewpoint of security and speed. This is only for confirmation purposes. To improve the communication speed, device drivers need to be created.

## 7.8 Discussion

In this study, we presented a communication scheme between CPU and FPGA abstracted by AXI in Xilinx FPGAs. Data communication is a barrier to parallelization in FPGAs and CPUs and GPUs: in the CGH computation example, if the data size that can be sent by the communication circuit is 128 bits, the number of pixels that can be sent at a time (assumed to be 8 bits) is 16. Here, there will be a delay in data transmission if more than 16 are parallelized. If data transmission and reception are slow, the communication time becomes a barrier that lowers the circuit's arithmetic efficiency and makes it impossible to produce an arithmetic speed commensurate with parallelization.

A possible countermeasure is to create high-speed communication circuits that can transmit and receive numerous data at high speeds using direct memory access (DMA); DMA allows asynchronous communication so that the computation circuit can operate while sending and receiving data. Pipeline parallelization is completed at the operator level, so if all needed data can be stored in the FPGA, there are no communication constraints during computation. The longer the pipeline, the higher the pipeline parallelization's speed-up rate. By integrating pipeline parallelization with data parallelization, special-purpose computers that are unaffected by communication barriers can be built.

**Fundings** This work was supported by JSPS KAKENHI Grant Number JP21K21294.

## References

1. Xilinx 7 Series FPGAs: The Logical Advantage, Xilinx. https://docs.xilinx.com/v/u/en-US/wp405-7Series-Logical-Advantage
2. Intel MAX 10 FPGA Device Architecture, intel. https://www.intel.com/content/www/us/en/docs/programmable/683105/current/logic-array-block.html
3. DSP Solution, Xilinx. https://www.xilinx.com/products/technology/dsp.html
4. Memory Solution, Xilinx. https://www.xilinx.com/products/technology/memory.html
5. Implementation, Xilinx. https://www.xilinx.com/products/design-tools/vivado/implementation.html
6. Introduction to FPGA Design with Vivado High-Level Synthesis, Xilinx. https://docs.xilinx.com/v/u/en-US/ug998-vivado-intro-fpga-design-hls
7. IEEE Standard for VHDL Language Reference Manual, IEEE 1076-2019. https://standards.ieee.org/ieee/1076/5179/
8. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language, IEEE 1800-2017. https://standards.ieee.org/ieee/1800/6700/
9. ISO/IEC/IEEE 60559:2011. https://www.iso.org/standard/57469.html
10. AMBA AXI and ACE Protocol Specification Version E, ARM. https://developer.arm.com/documentation/ihi0022/e/
11. AXI Interconnect, Xilinx. https://www.xilinx.com/products/intellectual-property/axi_interconnect.html#overview
12. SoCs with Hardware and Software Programmability, Xilinx. https://japan.xilinx.com/products/silicon-devices/soc/zynq-7000.html

13. Zynq UltraScale+ MPSoC, Xilinx. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html
14. PetaLinux Tools, Xilinx. https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html
15. The Linux Programming Interface, man7.org. https://man7.org/linux/man-pages/man4/mem.4.html