Embedded Systems Design with Platform FPGAs

Principles & Practices

Chapter -2 The Target

Dr. Bassam Jamil
Adopted and updated from
Ron Sass and Andrew G. Schmidt

Outline

- From Transistor to FPGA
  - CMOS
  - Programmable logic devices
  - FPGA
- From HDL to a bitstream
  - HDL
  - HDL to bitstream

Big Picture

- a little knowledge can be a very bad thing
  - often, a very reasonable and legitimate specification written in an HDL language produces a horrible FPGA design
  - despite the efforts of many very talented researchers, high-level synthesis is imperfect
  - real-world consequence: write HDL idiomatically
- make conscious effort to think about the end result:
  - when I write this ...
  - the synthesis tool sees this ...
  - and, ultimately, this is what is implemented ...

Result: must have intimate details of FPGAs structures

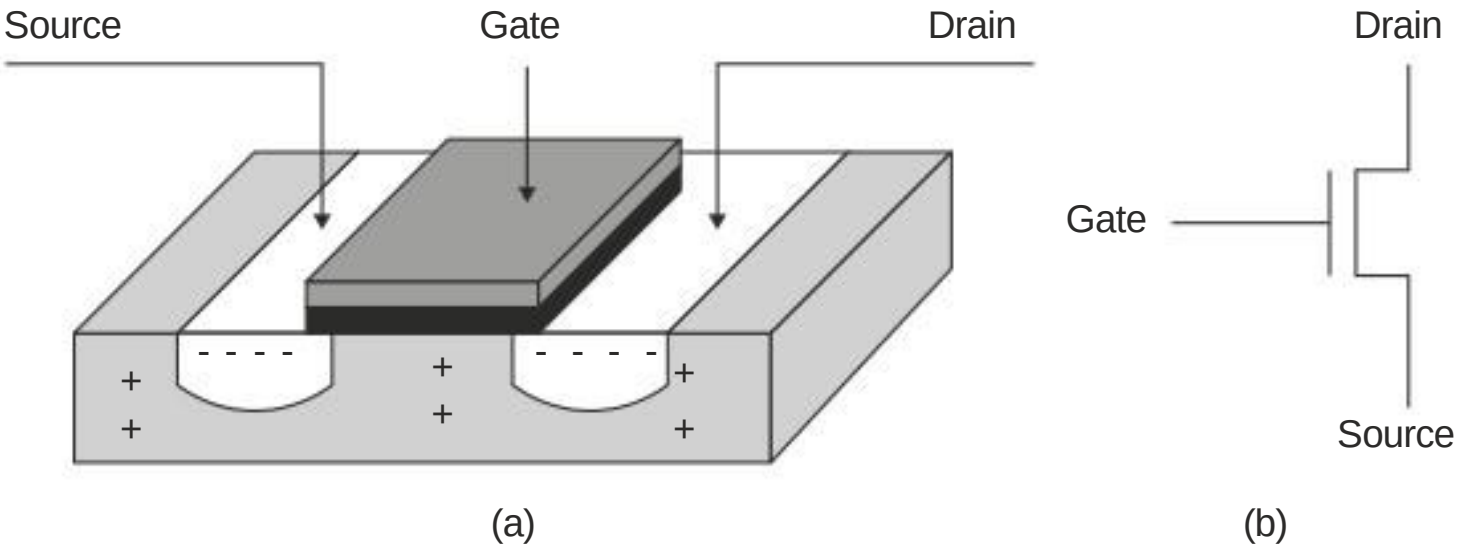
Transistors

- FPGAs rely up CMOS transistors
- important to understand their construct and functionality
- improves design decisions
- making for better designs
- NMOS/PMOS/CMOS transistors

Transistors

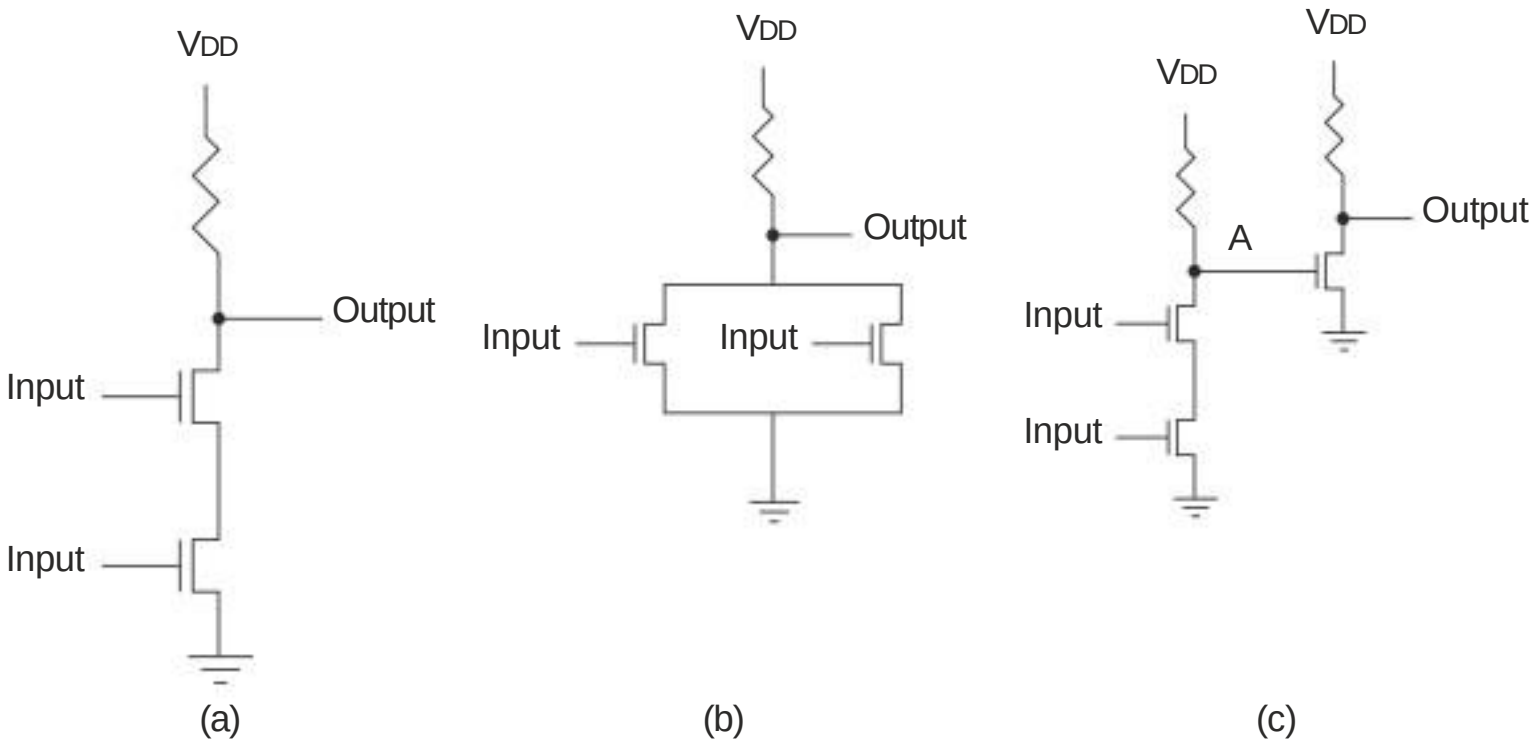
- focus on use as a voltage-controlled switch
- Metal Oxide Semiconductor Field Effect Transistor (MOSFET)
  - created from 2-D regions in a slab of silicon substrate
  - constructed with excess of positive or negative charge
  - a layer of silicon dioxide placed over substrate to insulate it from metal layer placed above silicon dioxide

N-Channel MOSFET



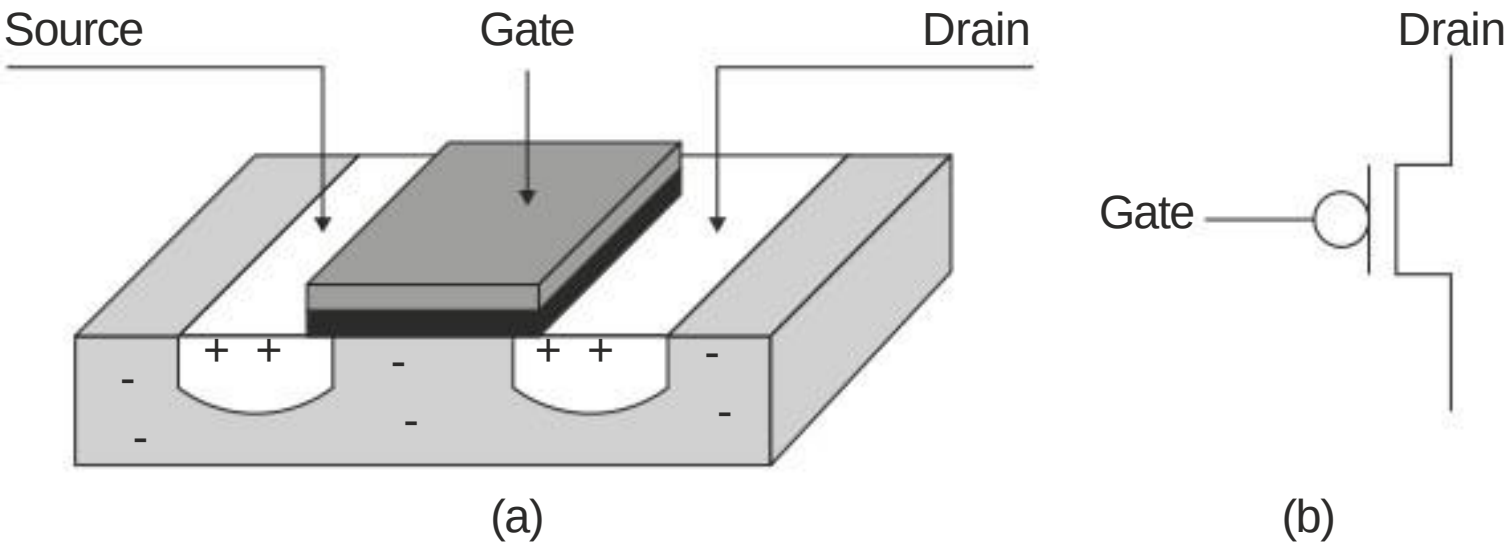
(a) an NMOS transistor in silicon substrate
(b) schematic of NMOS transistor

N-Channel MOSFET



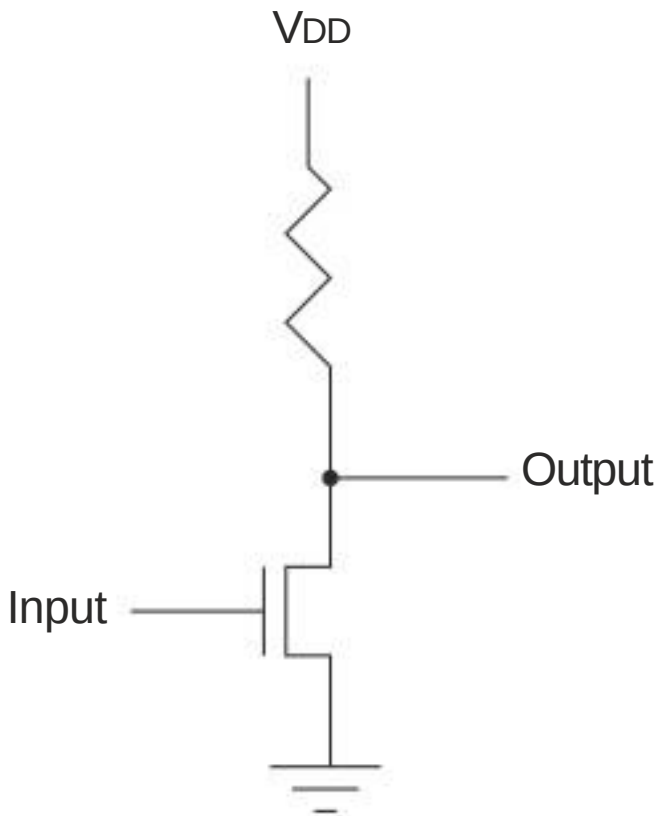
an NMOS (a) NAND gate (b) NOR gate (c) AND gate

P-Channel MOSFET



(a) an PMOS transistor in silicon substrate
(b) schematic of PMOS transistor

Inverter

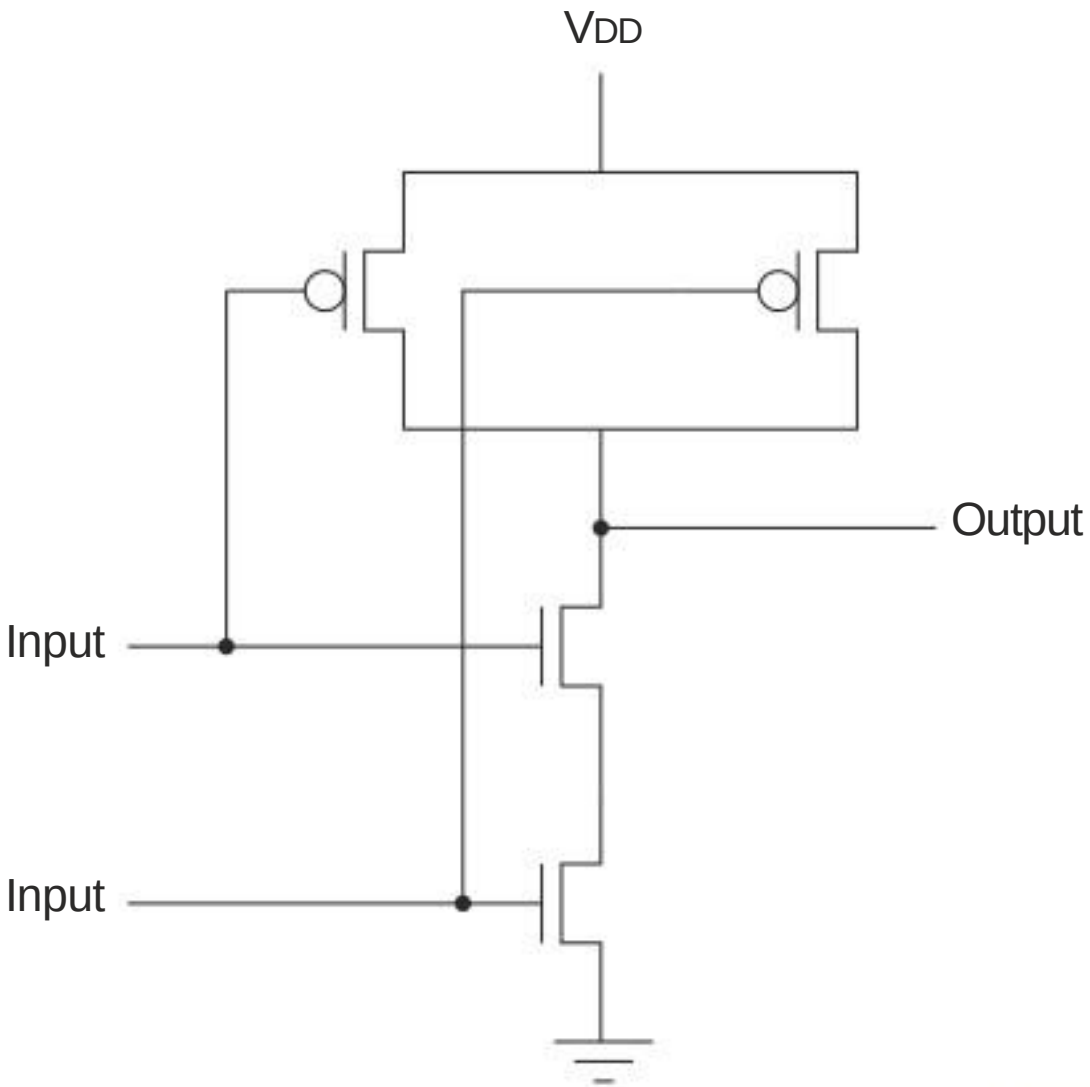


schematic symbol of an NMOS inverter

Complimentary MOS (CMOS) Transistor

- combine NMOS and PMOS transistor
- in steady-stead no appreciable current is flowing
- **rule-of-thumb:** most energy in a design is consumed when a CMOS gate is changing state

CMOS NAND



simple CMOS NAND circuit

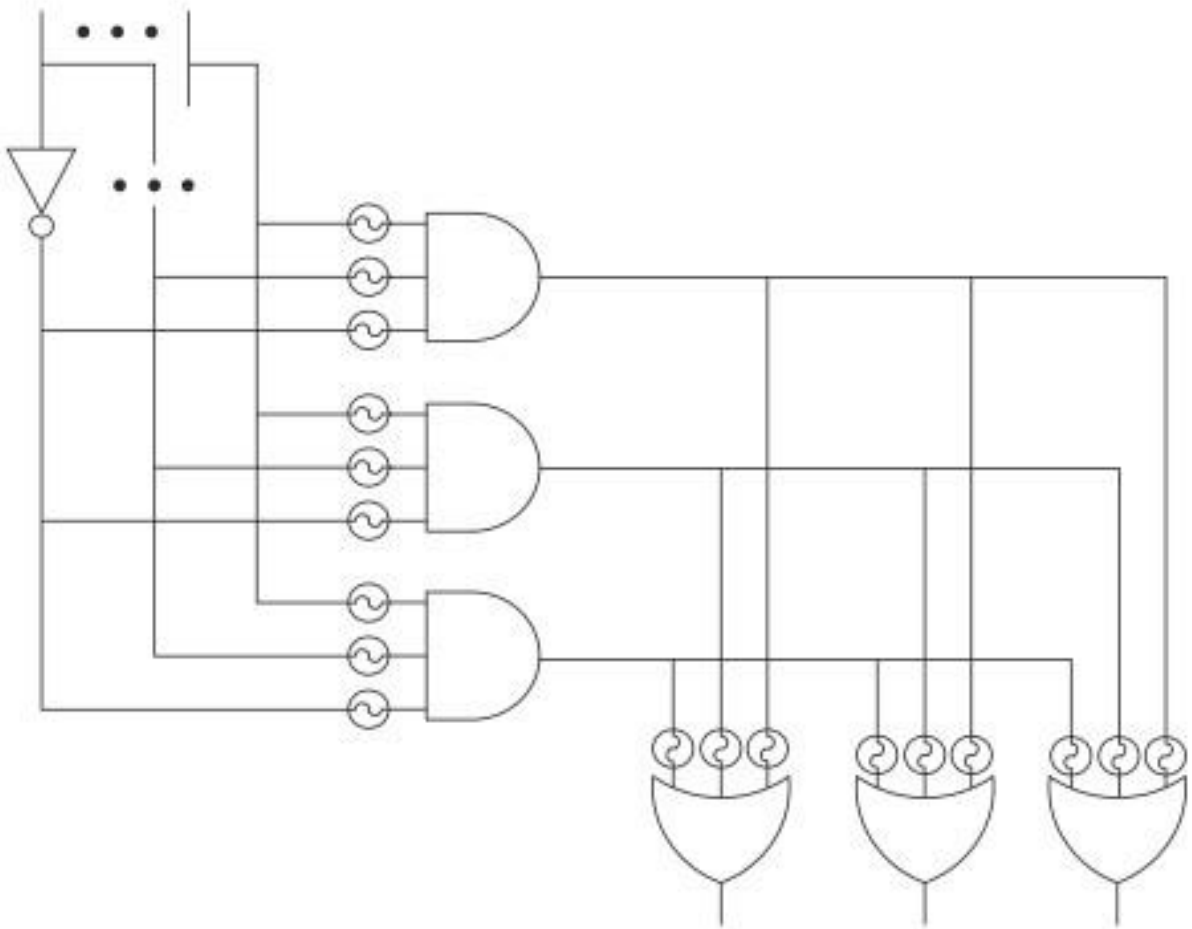
Programmable Logic Devices

- p- and n-type location/functionality fixed at fabrication
- how to build devices to be configured after fabrication?
- need a structure capable of implementing arbitrary combinational circuits, a storage cell (memory), and a mechanism to connect these resources.

Programmable Logic Arrays

- sum-of-products formulation of a Boolean function
- array of multi-input AND gates
- array of multi-input OR gates
- device programmed by breaking unwanted connections at the inputs to the AND and OR gates
- for manufacturing reasons, approach fell out of favor

Programmable Logic Devices



PLA device organization

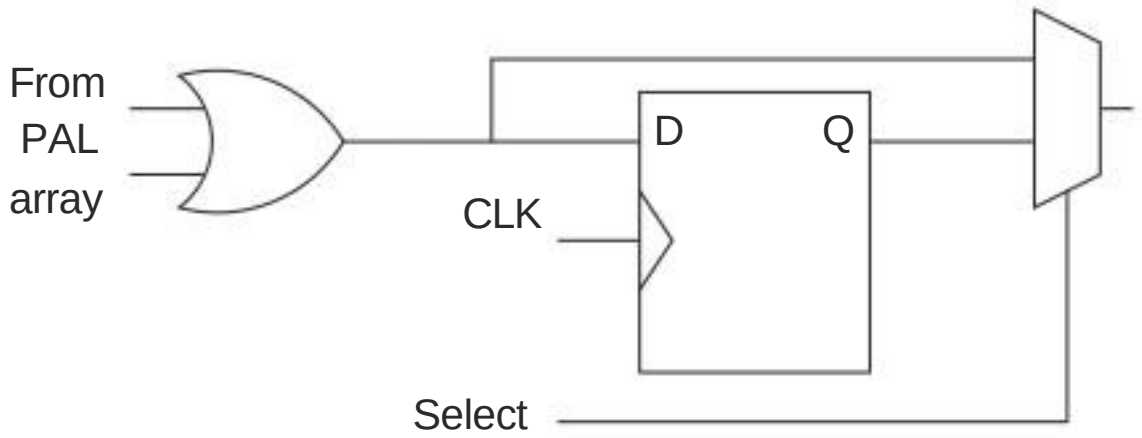
Programmable Logic Arrays

PLA functionality increased to include:

- memory elements, such as D-type Flip-Flops
- a MUX

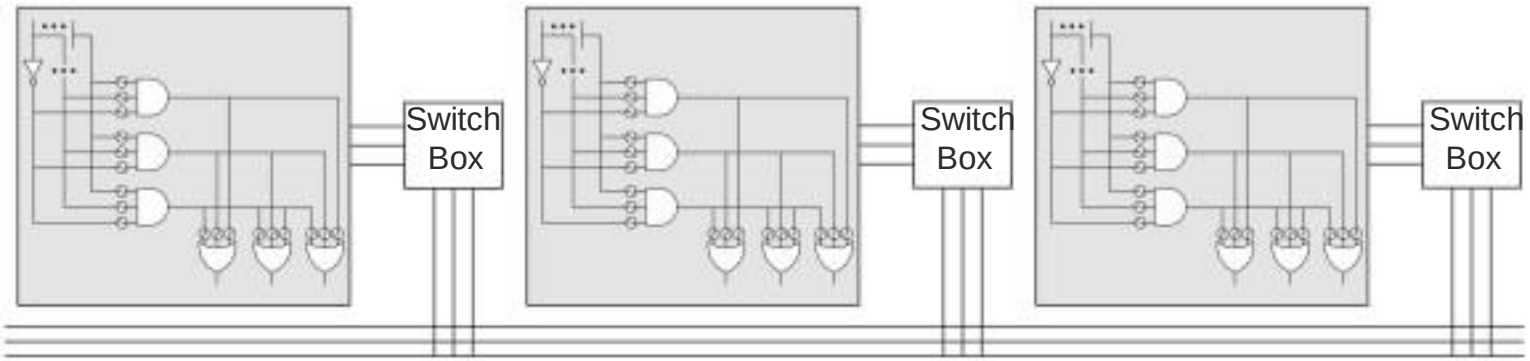
Programmable Logic Devices

Adding storage to a PLA logic cell:



Complex Programmable Logic Devices

PLAs with routing networks:



Field-Programmable Gate Array

Modern FPGA consists of:

- a 2-D array of programmable logic blocks,
- fixed-function blocks,
- and routing resources

implemented in CMOS technology

Field-Programmable Gate Array

Function generators

Field-Programmable Gate Array

Function generators

- Boolean logic functionality vs. physical gates

Field-Programmable Gate Array

Function generators

- Boolean logic functionality vs. physical gates
- implemented as an X -input lookup table

Field-Programmable Gate Array

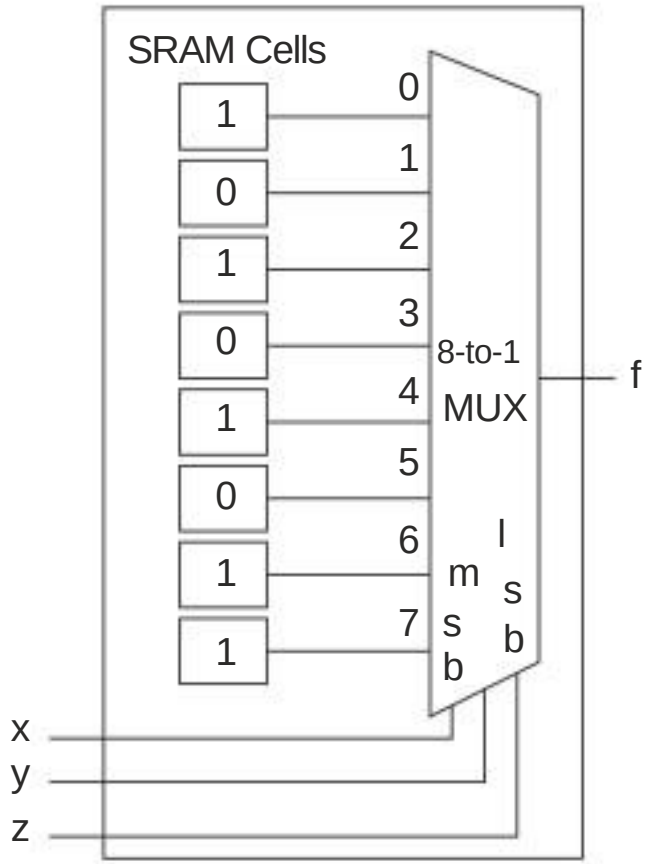
Function generators

- Boolean logic functionality vs. physical gates
- implemented as an X -input lookup table
- have a fixed propagation delay through the LUT

Function Generators Example

x	y	z	xy + z’
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

(a)



(b)

3-input function generator in (A) truth table and (B) look-up table form for

$f(x, y, z) = xy + z'$

Function Generators

To generalize, basic n-input function generator consists of:

- a 2^n-to-1 multiplexer (MUX)
- 2^n SRAM (static random access memory) cells
- i.e. a 3-LUT is a 3-input function generator
- 4-LUTs and 6-LUTs are more common
  (3-LUTs are easier to draw by hand!)

Function Generators

What about more complex functions?

- use multiple LUTs
- function decomposed into sub-functions
- sub-function has subset of inputs
- each sub-function assigned to one LUT
- LUTs combined via routing resources to form function

Field-Programmable Gate Array

To **configure** the FPGA:

Field-Programmable Gate Array

To **configure** the FPGA:

- stream configuration data bit-by-bit

Field-Programmable Gate Array

To **configure** the FPGA:

- stream configuration data bit-by-bit
- bits specify logic functionality

Field-Programmable Gate Array

To **configure** the FPGA:

- stream configuration data bit-by-bit
- bits specify logic functionality
- SRAM is volatile — reconfigure if power lost

Storage Elements

Most circuits need:

- function generators (LUTs)
- storage elements (flip-flops)

Can now create sequential circuits!

Logic Cells

A **logic cell** combines:

Logic Cells

A **logic cell** combines:

- a function generator (LUT), and

Logic Cells

A **logic cell** combines:

- a function generator (LUT), and
- a D flip-flop

Logic Cells

- low-level building block upon which FPGA designs are built
- combinational/sequential logic built from logic cells
- used to compare capacity of different FPGA devices
- **slice**

Logic Cells

- low-level building block upon which FPGA designs are built
- combinational/sequential logic built from logic cells
- used to compare capacity of different FPGA devices
- **slice** Xilinx term for primitive with LUTs and FFs (quantity depends on FPGA device)

Logic Blocks

A **logic block** combines:

Logic Blocks

A **logic block** combines:

- several logic cells

Logic Blocks

A **logic block** combines:

- several logic cells
- with special-purpose circuitry (add/sub carry chain)

Logic Blocks

A **logic block** combines:

- several logic cells
- with special-purpose circuitry (add/sub carry chain)
- connected by routing network for complex circuits

Logic Blocks

A **logic block** combines:

- several logic cells
- with special-purpose circuitry (add/sub carry chain)
- connected by routing network for complex circuits
- close logic cells have short communication paths

Logic Blocks

A **logic block** combines:

- several logic cells
- with special-purpose circuitry (add/sub carry chain)
- connected by routing network for complex circuits
- close logic cells have short communication paths
- **Configurable Logic Block** Xilinx term for primitive with Slices (quantity depends on FPGA device)

Logic Blocks

A **switch box** is:

Logic Blocks

A **switch box** is:

- used to route between the inputs/outputs of a logic block to the general on-chip routing network

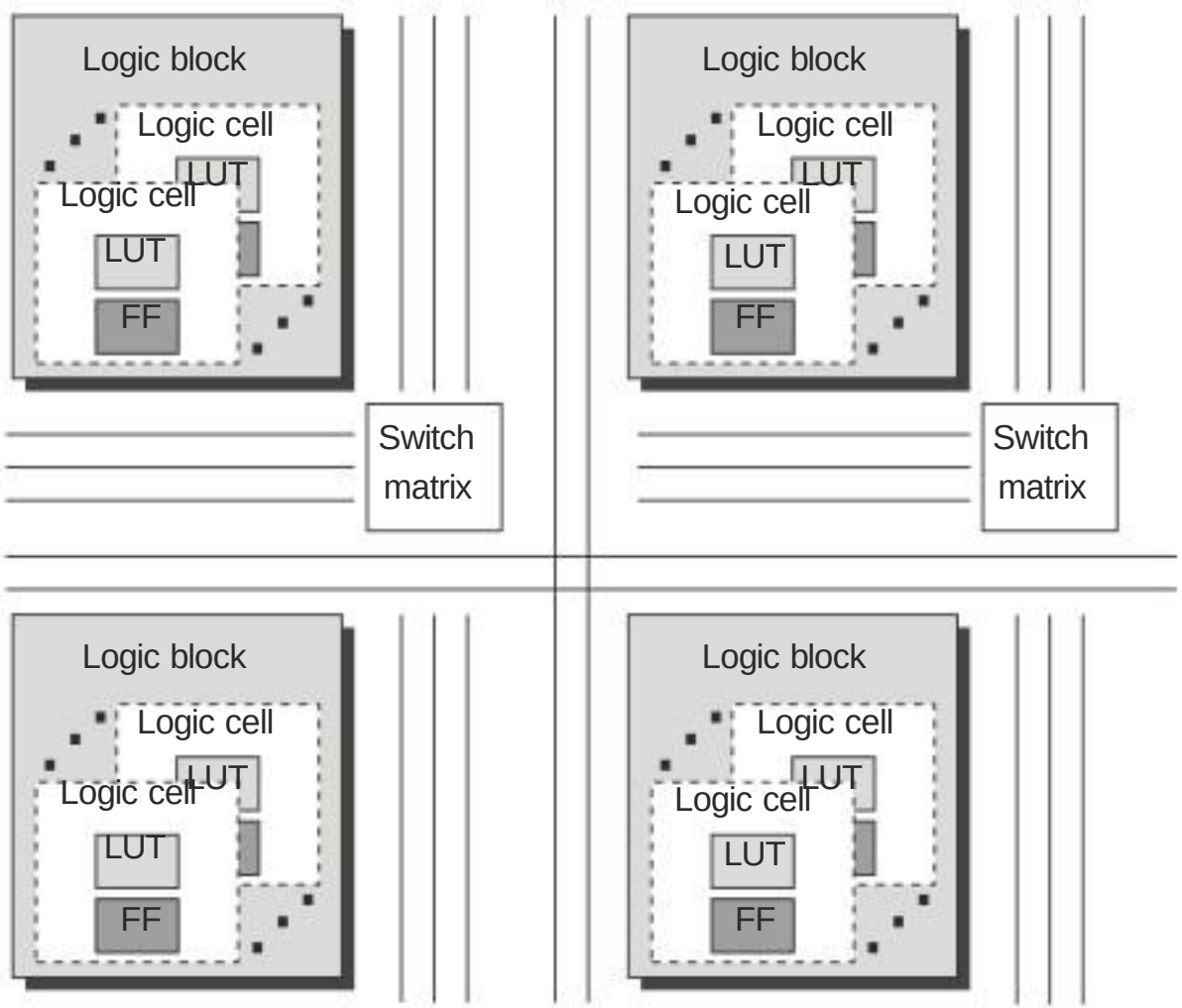
Logic Blocks

A **switch box** is:

- used to route between the inputs/outputs of a logic block to the general on-chip routing network
- responsible for passing signals between wire segments

Field-Programmable Gate Array

Taken all together, a high-level structure of a simple FPGA:



Special-Purpose Function Blocks

FPGAs combine programmable logic with additional resources embedded into FPGA fabric. Why?

- embedded resources consume less resources
- embedded resources consume less power
- embedded resources can operate at a higher frequency

Special-Purpose Function Blocks

Such as:

- Block RAM
- Digital Signal Processing Blocks
- Processors
- Digital Clock Manager
- Multi-Gigabit Transceivers

Block RAM

- many designs require use of on-chip memory
- possible to use using logic cells to build memory elements
- logic resources quickly consumed as demand grows
- **Block RAM** on-chip memory in FPGA fabric

Digital Signal Processing Blocks

Digital signal processing (DSP) quickly consumes resources:

- implement necessary components in FPGA fabric:
  - multiplier
  - accumulator
  - adder
  - bitwise logical operations (AND, OR, NOT, NAND, etc.)
- combine DSP blocks to perform larger operations:
  - single/double precision floating point
  - addition, subtraction, multiplication
  - division, and square-root

Processors

Simplify designs, reducing resource and power consumption

- IBM PowerPC 405 and 440
  (in Virtex 4 and 5 FX FPGAs, respectively)
- conventional RISC processors
- implement PowerPC instruction set
- no hardware floating point functions
- level-1 instruction and data caches
- memory management unit
- translation look-aside buffer
- can connect to FPGA fabric through system bus
- not all FPGAs have embedded processors

Digital Clock Manager

How to support designs with multiple clock domains?
Digital Clock Manager (DCM):

- generate different clocks from a reference clock
- divide reference clock
- clocks have lower jitter and phase relationship

Multi-Gigabit Transceivers

High Speed Serial Transceivers are devices that:

Multi-Gigabit Transceivers

High Speed Serial Transceivers are devices that:

- serialize and deserialize parallel data over a serial channel

Multi-Gigabit Transceivers

High Speed Serial Transceivers are devices that:

- serialize and deserialize parallel data over a serial channel
- are capable of baud rates from 100 Mb/s to 11.0 Gb/s

Multi-Gigabit Transceivers

High Speed Serial Transceivers are devices that:

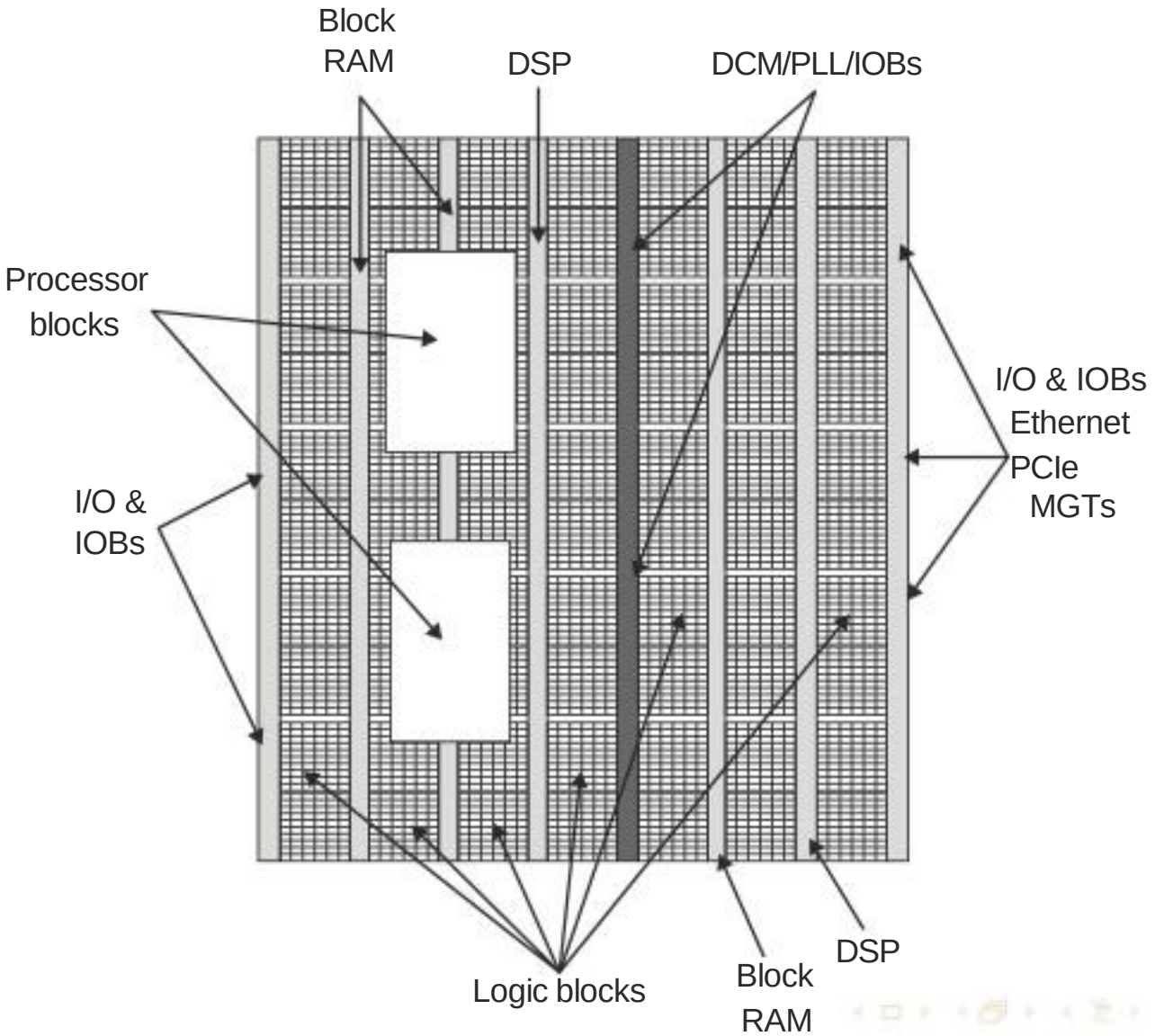
- serialize and deserialize parallel data over a serial channel
- are capable of baud rates from 100 Mb/s to 11.0 Gb/s
- can implement variety of standards: Ethernet, InfiniBand

Multi-Gigabit Transceivers

High Speed Serial Transceivers are devices that:

- serialize and deserialize parallel data over a serial channel
- are capable of baud rates from 100 Mb/s to 11.0 Gb/s
- can implement variety of standards: Ethernet, InfiniBand
- can be bonded together to increase bandwidth

Platform FPGA



Writing HDL for Platform FPGAs

Assume you are moderately familiar with HDL (VHDL or Verilog)

- **Question:** how to write HDL for FPGAs?
- **Answer:** Carefully!
- when writing HDL — be careful, not everything is synthesizable
- briefly cover VHDL next

VHSIC Hardware Description Language

- describe digital circuits
- major styles of writing VHDL:
  - Structural circuits described as logic/function units (AND/OR/NOT, etc.) and signals
  - Data-Flow type of structural has syntactic support to make expressing Boolean logic easier
  - Behavioral circuits use imperative language to describe how outputs relate to inputs via a process
- a third style is a mix between structural and behavioral
- behavioral form of VHDL most convenient for programmers
- structural is useful for black box designs

VHDL Syntax

- entity declaration:
  describes the component's interface
- followed by one or more declared architectures:
  the implementation of the interface
- best to learn by example

1-bit Full Adder in VHDL

```
library ieee;
use ieee . std_logic_1164 . all ;

entity fadder is port (
    a, b   : in std_logic;
    cin    : in std_logic;
    sum    : out std_logic;
    cout   : out std_logic );
end fadder ;

architecture beh of fadder is
begin
    sum  <= a xor b xor cin ;
    cout <= (a and b) or (b and cin ) or (a and cin );
end beh ;
```

2-bit Adder in VHDL (1/3)

```
library ieee;
use ieee.std_logic_1164.all;

entity fadder2bit is port (
        a, b    : in  std_logic_vector(1 downto 0);
        cin     : in  std_logic;
        clk     : in  std_logic;
        rst     : in  std_logic;
        sum     : out std_logic_vector(1 downto 0);
        cout    : out std_logic );
end fadder2bit;

architecture beh of fadder2bit is
  -- 2-bit Register Declaration for A, B, Sum
   signal a_reg, b_reg, sum_reg : std_logic_vector(1 downto 0);
  -- 1-bit Register Declaration for Cin, Cout
   signal cin_reg, cout_reg     : std_logic;
  -- 1-bit Internal signal to connect carry-out to carry-in
   signal carry_tmp             : std_logic;
  -- 1-bit Full Adder Component Declaration
   component fadder is
   port (
      a, b  : in  std_logic;
      cin   : in  std_logic;
      sum   : out std_logic;
      cout  : out std_logic );
   end component fadder;
```

2-bit Adder in VHDL (2/3)

```
begin
  -- VHDL Process to Register Inputs A, B and Cin
   register_proc      : process    (clk) is
   begin
      if   (( clk ' event ) and      (clk    =   '1')) then
        -- Synchronous Reset
         if   (rst    =   '1') then
           -- Initialize Input Registers
            a_reg        <= "00" ;
            b_reg        <= "00" ;
            cin_reg      <=  '0';
           -- Initialize Outputs
            sum          <= "00" ;
            cout         <=  '0';
         else
           -- Register Inputs
            a_reg      <= a ;
            b_reg      <= b ;
            cin_reg    <= cin ;
           -- Register Outputs
            sum        <= sum_reg ;
            cout       <= cout_reg ;
         end if;
      end if;
   end process register_proc ;
```

2-bit Adder in VHDL (3/3)

```
-- Instantiate Bit 1 of Full Adder
 fa1   : fadder port map       (
    a       => a_reg (1) ,
    b       => b_reg (1) ,
    cin     => carry_tmp ,
    sum     => sum_reg (1) ,
    cout    => cout_reg      );
-- Instantiate Bit 0 of Full Adder
 fa0   : fadder port map       (
    a       => a_reg (0) ,
    b       => b_reg (0) ,
    cin     => cin_reg ,
    sum     => sum_reg (0) ,
    cout    => carry_tmp     );
end beh ;
```

Finite State Machines in VHDL

- useful for a variety of purposes
- writing style can result in different netlists
- recommended to follow synthesis guide
  (i.e. Xilinx Synthesis Technology (XST) User Guide)
- simple 8-bit Adder with FSM example

8-bit Adder in VHDL with FSM (1/4)

```
-- Traditional Library and Packages used in a hardware core
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adderFSM is
   generic (
     C_DWIDTH : in natural := 8);
   port (
   a, b          : in  std_logic_vector(C_DWIDTH-1 downto 0);
   a_we, b_we    : in  std_logic;
   clk, rst      : in  std_logic;
   result        : out std_logic_vector(7 downto 0);
   valid         : out std_logic);
end adderFSM;

architecture beh of adderFSM is
  -- 8-bit Register/Next Declaration for A and B
   signal a_reg, b_reg     : std_logic_vector(7 downto 0);
   signal a_next, b_next   : std_logic_vector(7 downto 0);
  -- Finite State Machine Type Declaration
   type FSM_TYPE is (wait_a_b, wait_a, wait_b, add_a_b);
  -- Internal signals to represent the Current State (fsm_cs)
  -- and the Next State (fsm_ns) of the state machine
   signal fsm_cs  : FSM_TYPE := wait_a_b;
   signal fsm_ns  : FSM_TYPE := wait_a_b;
```

8-bit Adder in VHDL with FSM (2/4)

```
begin
  -- Finite State Machine Process to Register Signals
   fsm_register_proc        : process     (clk) is
   begin
     -- Rising Edge Clock to infer Flip Flops
      if   (( clk ' event ) and      (clk    =   '1')) then
        -- Synchronous Reset       /  Return to  Initial          State
         if   (rst     =   '1') then
            a_reg           <=  (others    =>   '0 ');
            b_reg           <=  (others    =>   '0 ');
            fsm_cs          <=  wait_a_b ;
         else
            a_reg           <=  a_next ;
            b_reg           <=  b_next ;
            fsm_cs          <=  fsm_ns ;
         end  if ;
      end  if ;
   end  process  fsm_register_proc ;
```

8-bit Adder in VHDL with FSM (3/4)

```
-- Finite State Machine Process for Combination Logic
fsm_logic_proc : process ( fsm_cs , a , b , a_we, b_we, a_reg, b_reg) is
begin
  -- Infer Flip-Flop rather than Latches
   a_next  <= a_reg ;
   b_next  <= b_reg ;
   fsm_ns  <= fsm_cs ;

   case (fsm_cs) is
     -- State 0: Wait for either A or B Input
     --  denoted by a_we =  '1' and/or b_we = '1'
      when wait_a_b =>
        -- Clear Initial Context of Registers/Outputs
         a_next   <= (others => '0');
         b_next   <= (others => '0');
         result   <= (others => '0');
         valid    <= '0';
         if ((a_we and b_we) = '1') then
            a_next  <= a ;
            b_next  <= b ;
            fsm_ns  <= add_a_b ;
         elsif (a_we = '1') then
            a_next  <= a ;
            fsm_ns  <= wait_b ;
         elsif (b_we = '1') then
            b_next  <= b ;
            fsm_ns  <= wait_a ;
         end if ;
```

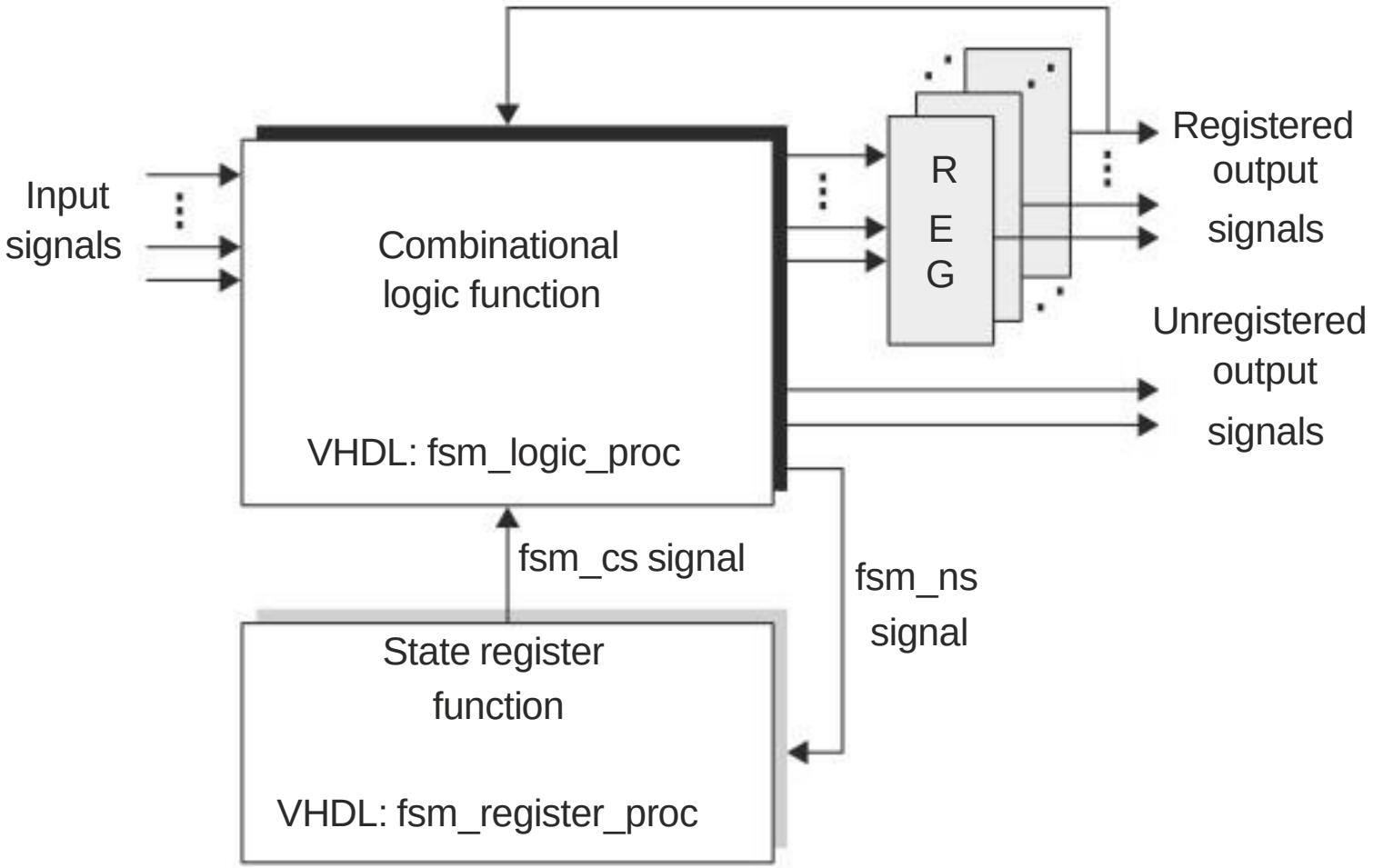
8-bit Adder in VHDL with FSM (4/4)

```
        -- State 1: Wait for A Input
        --  Being in this state means B      already    arrived
         when wait_a =>
            result          <= (others   =>  '0');
            valid           <=  '0';
            if   (a_we =    '1') then
               a_next       <= a ;
               fsm_ns       <= add_a_b ;
            end if;

        -- State 2: Wait for B Input
        --  Being in this state means A      already    arrive
         when wait_b =>
            result          <= (others   =>  '0');
            valid           <=  '0';
            if   (b_we =    '1') then
               b_next       <= b ;
               fsm_ns       <= add_a_b ;
            end if;

        -- State 3: Perform Add Operation and return to wait_a_b
        when add_a_b =>
            result          <= a_reg   + b_reg;
            valid           <=  '1';
            fsm_ns          <= wait_a_b ;
      end case ;
   end process fsm_logic_proc ;
end beh ;
```

Sequential Logic Diagram



From HDL to Bitstream

Overview of process:

synthesis the process of generating the **logic** configuration for the specified device from HDL source

netlist is a computer representation of the a collection of logic units and how they are to be connected

mapping (also known as map) the process of grouping gates and assigning functionality to FPGA **primitives**

placement after map the next step in the back-end tool flow is to **assign** the FPGA primitives in the netlist to **specific blocks** on a particular FPGA

From HDL to Bitstream

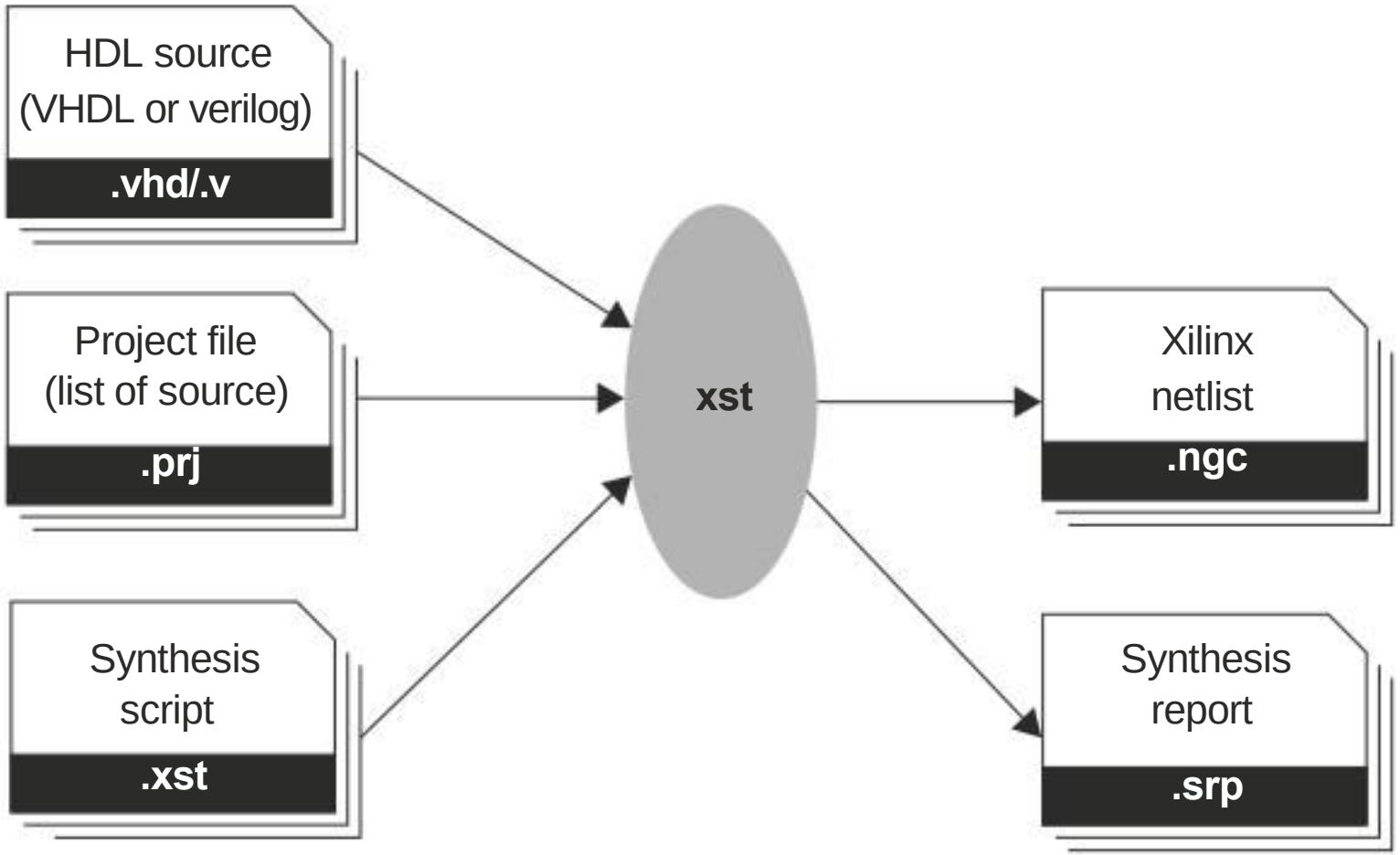
Overview of process:

routing (also known as route) is the process after placement where during which **wire segments are selected** and passing transistors are set

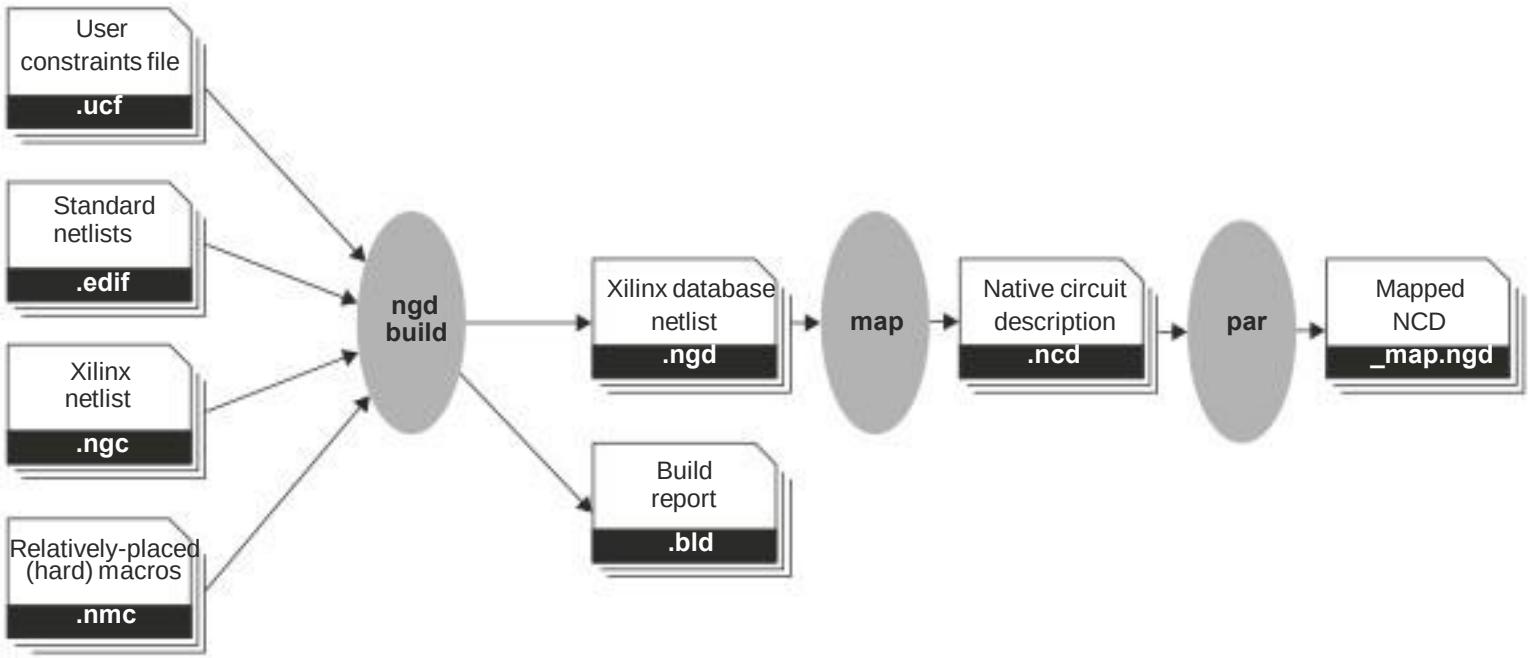
bitgen the process after routing which takes a placed and routed netlist and sets generates a configuration file known as a **bitstream that will be used to configure the FPGA device**

bitstream is the file that includes a header and frames of configuration information needed to set the SRAM cells of the FPGA in order to set the functionality of the device

Xilinx Synthesis Process



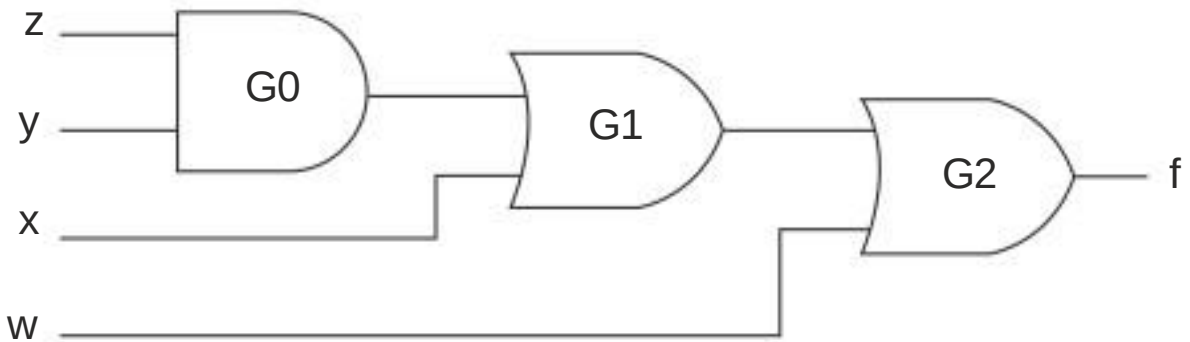
Xilinx Synthesis Process



From HDL to Configuration Bitstream

Let's start with an example, consider the Boolean function:

$$f(w, x, y, z) = w + x + yz$$



From HDL to Configuration Bitstream

Circuit expressed in a netlist.
Nets describe how the cells are “wired up”

- each gate becomes a “cell”
- each primitive cell includes:
  - type of gate
  - a name for this cell
  - and a set of ports
- ports have names and direction
- complex cells formed with instances of cells and nets
- hierarchy formed from complex cells with top-level cell encompassing all other cells
- top-level port cells associated with external pins of FPGA

From HDL to Configuration Bitstream

Excerpt of netlist:

```
(edif gates
   (edifVersion 2 0 0)
   (external UNISIMS
      (cell LUT4
         (cellType GENERIC)
         (interface
            (port I0
               (direction       INPUT)
            )
            (port I1
               (direction       INPUT)
            )
            (port I2
               (direction       INPUT)
            )
            (port I3
               (direction       INPUT)
            )
            (port O
               (direction       OUTPUT)
            )
         )
      )
      )
)
```

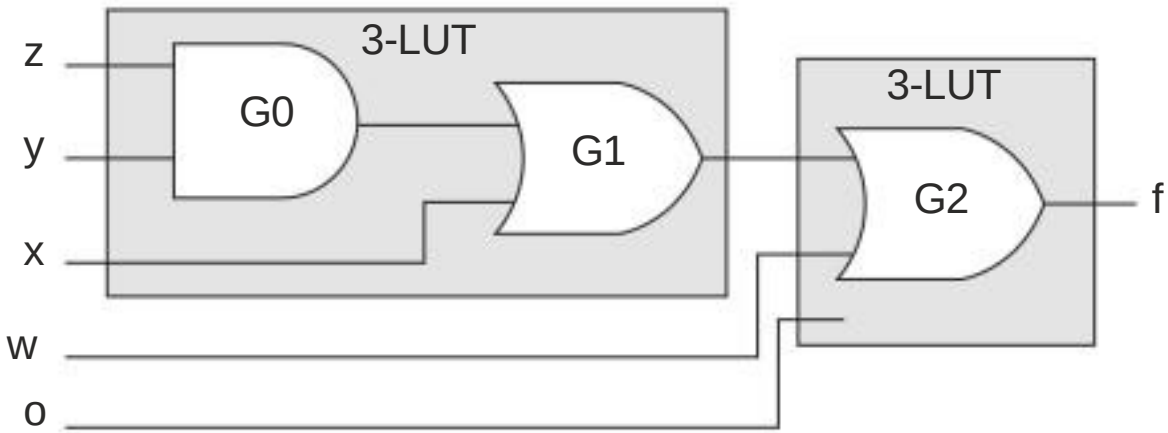
From HDL to Configuration Bitstream

Digital circuit to sequence of bits using previous example:

- with 4-inputs, cannot implement f in a single 3-LUT
- solution: two 3-LUTs and routing resources
- f can be decomposed into two functions:
  - $f(w, x, y, z) = f_2(w, f_1(x, y, z))$
  - $f_1(x,y,z) = x + yz$
  - $f_2(w,t) = w + t$
- now f_1 and f_2 can be used to configure two 3-LUTs
- the output of f_1 is routed to the second input of f_2

From HDL to Configuration Bitstream

Example of a 4-input function mapped to two 3-LUTs:



From HDL to Configuration Bitstream

After generating the netlist:

- **mapping** (MAP) process of grouping gates and assigning functionality to FPGA primitives
- after MAP another netlist is generated
- cells now refer to FPGA primitives (LUTs,FFs, etc.)

From HDL to Configuration Bitstream

After MAP:

- **placement** assign FPGA primitives in netlist to specific blocks on particular FPGA device
- our example, f uses two 3-LUTs
- during placement tools decide which two 3-LUTs to use and add that information to the netlist
- try to place cells close by for lowest propagation delay
- **routing** process of picking wire segments and setting passing transistors
- place and route often combined and called **PAR**

From HDL to Configuration Bitstream

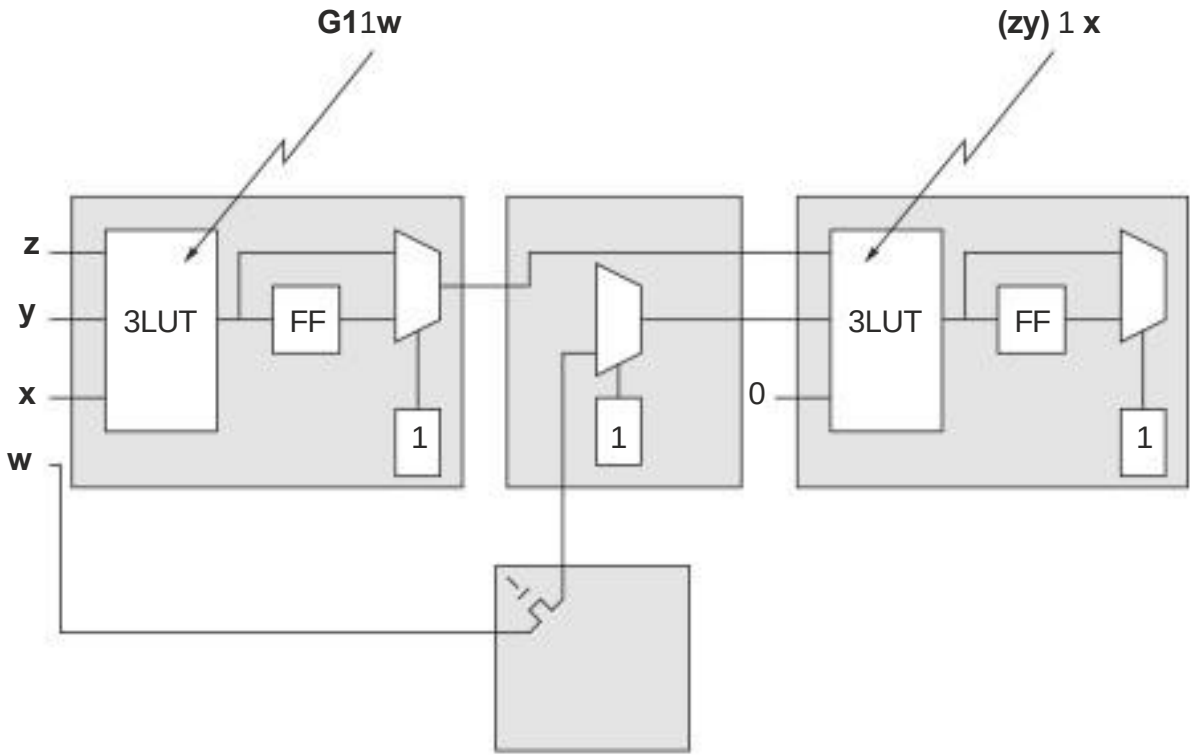


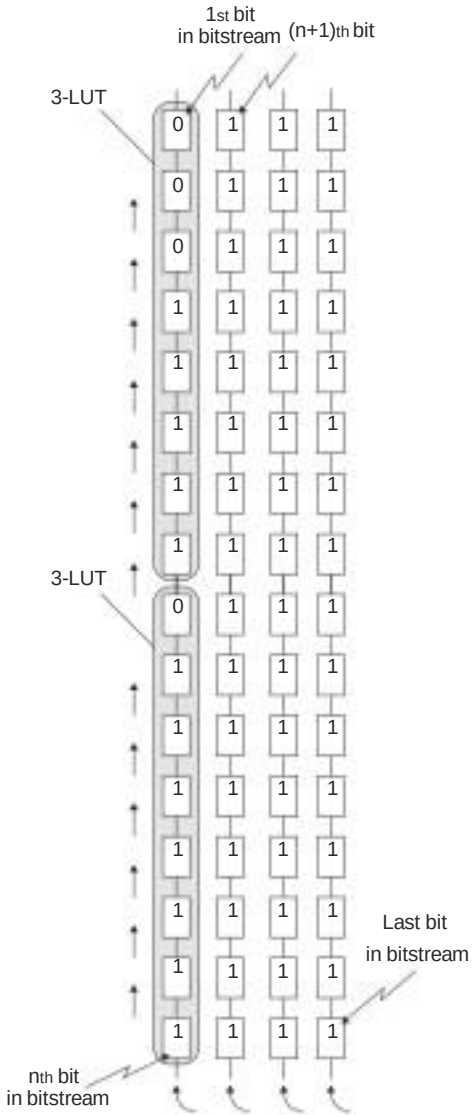
illustration of f (w , x , y , z) placed and routed

From HDL to Configuration Bitstream

After PAR, final step:

- convert SRAM cell settings into a linear stream of bits
- imagine all cells arrayed into two dimensions
- one column might be some 3-LUTs on the chip
- the column actually is all SRAM cells of 3-LUTs
- all column's SRAM cells form giant shift register
- **bitgen** process of taking a placed and routed netlist and sets the SRAM cells of this large 2-D array accordingly
- along with a header, each of the columns is written out sequentially to a binary file called a **bitstream**

From HDL to Configuration Bitstream



Chapter-In-Review

- transistors: NMOS,PMOS,CMOS
- programmable logic devices
- writing VHDL for Platform FPGAs
- Platform FPGA properties
- from HDL to a bitstream

Chapter 2 Terms

logic cell is a low-level building block of an FPGA which consists of a look-up table and a D flip-flop storage element

logic block is a group of several logic cells along with special-purpose circuitry, such as an adder/subtractor carry chain

slice is the Xilinx defined term for a group of logic cells

configurable logic block (CLB) is the Xilinx defined term for a logic block, which is a group of slices interconnect circuitry

switch box is used to route between the inputs/outputs of a logic block to the general on-chip routing network along with passing signals from wire segment to wire segment

Chapter 2 Terms

Block RAM (BRAM) a Xilinx defined term for a fix amount of on-chip memory within the FPGA fabric

high speed serial transceivers are devices that serialize and deserialize parallel data over a serial channel

simulation the process of translating an HDL source into a form suitable for a general-purpose processor to mimic the hardware

synthesis the process of generating the logic configuration for the specified device from HDL source

netlist is a computer representation of the a collection of logic units and how they are to be connected

mapping the process of grouping gates and assigning functionality to FPGA primitives

Chapter 2 Terms

placement after map the next step in the back-end tool flow is to assign the FPGA primitives in the netlist to specific blocks on a particular FPGA

routing is the process after placement where during which wire segments are selected and passing transistors are set

bitgen the process after routing which takes a placed and routed netlist and sets generates a configuration file known as a bitstream that will be used to configure the FPGA device

bitstream is the file that includes a header and frames of configuration information needed to set the SRAM cells of the FPGA in order to set the functionality of the device