Digital Logic Circuit (SE273 – Fall 2020)

Lecture 6: Hardware Description Languages

Some slides from ICL, Peter Chung

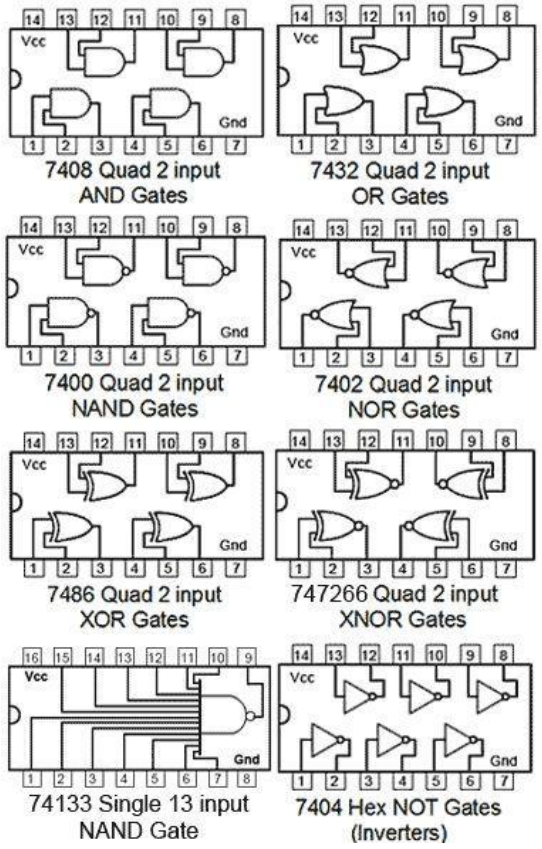
Jaesok Yu, Ph.D. (jaesok.yu@dgist.ac.kr)

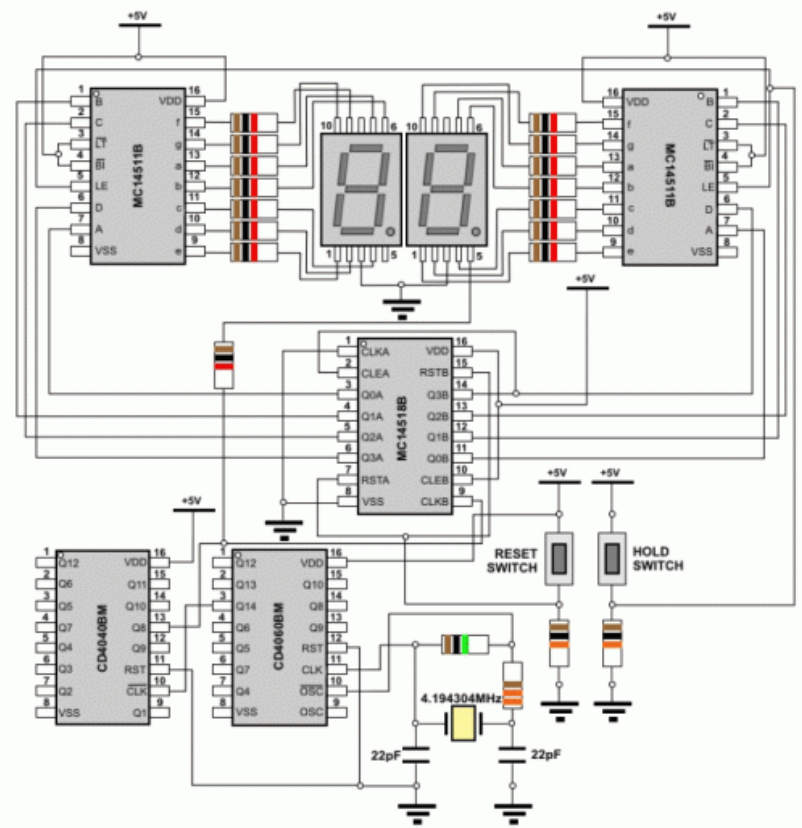
Assistant Professor

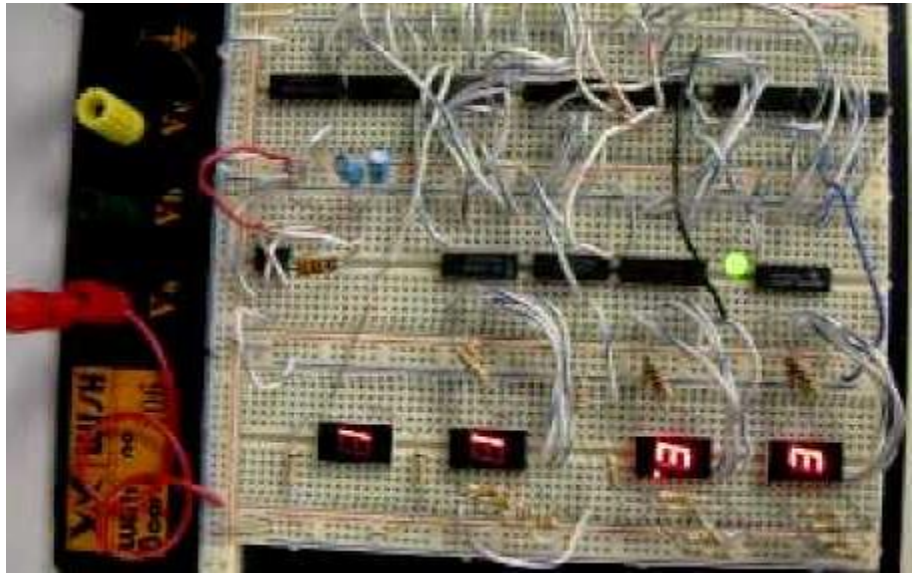
Department of Robotics Engineering, DGIST

Traditional ways of implementing digital circuit

- Discrete logic based on gates or small silicon
- Tedious, Slow, Expensive, Low Efficiency, Low Readability

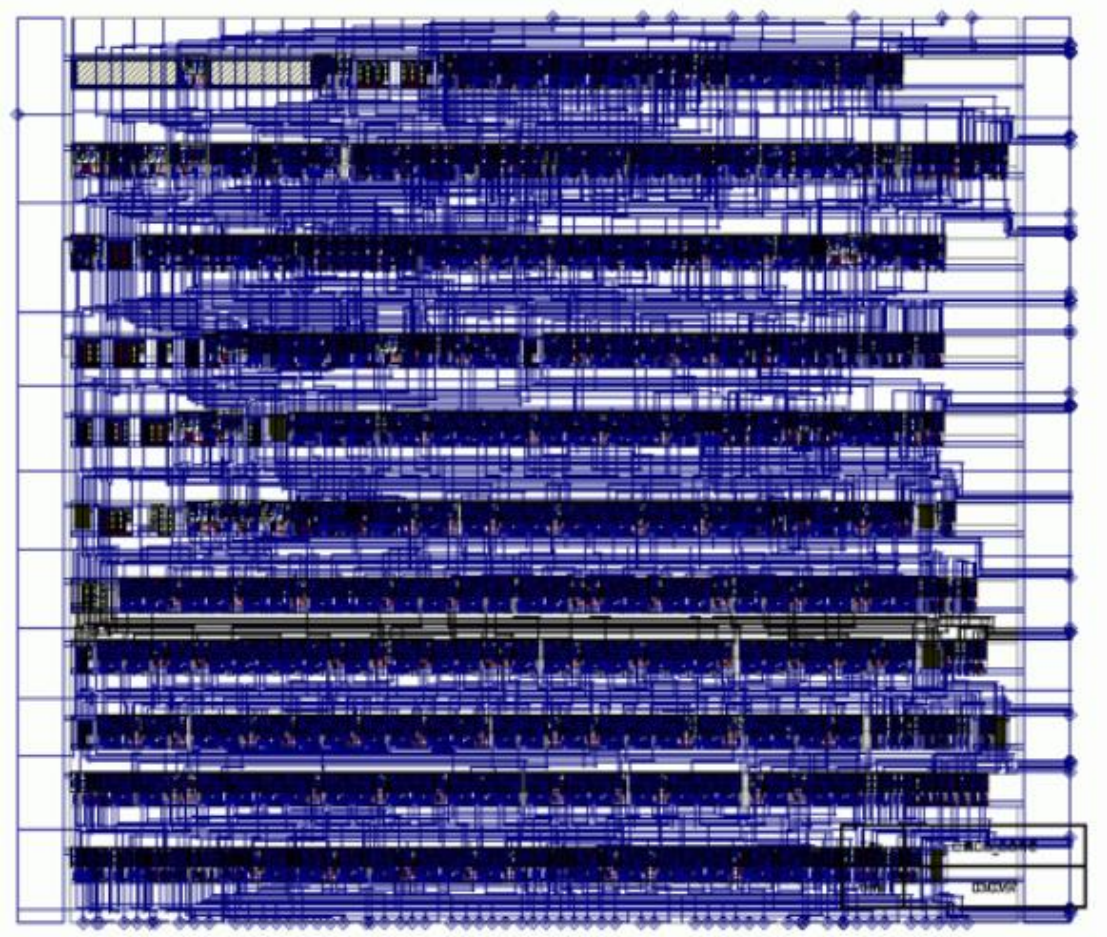






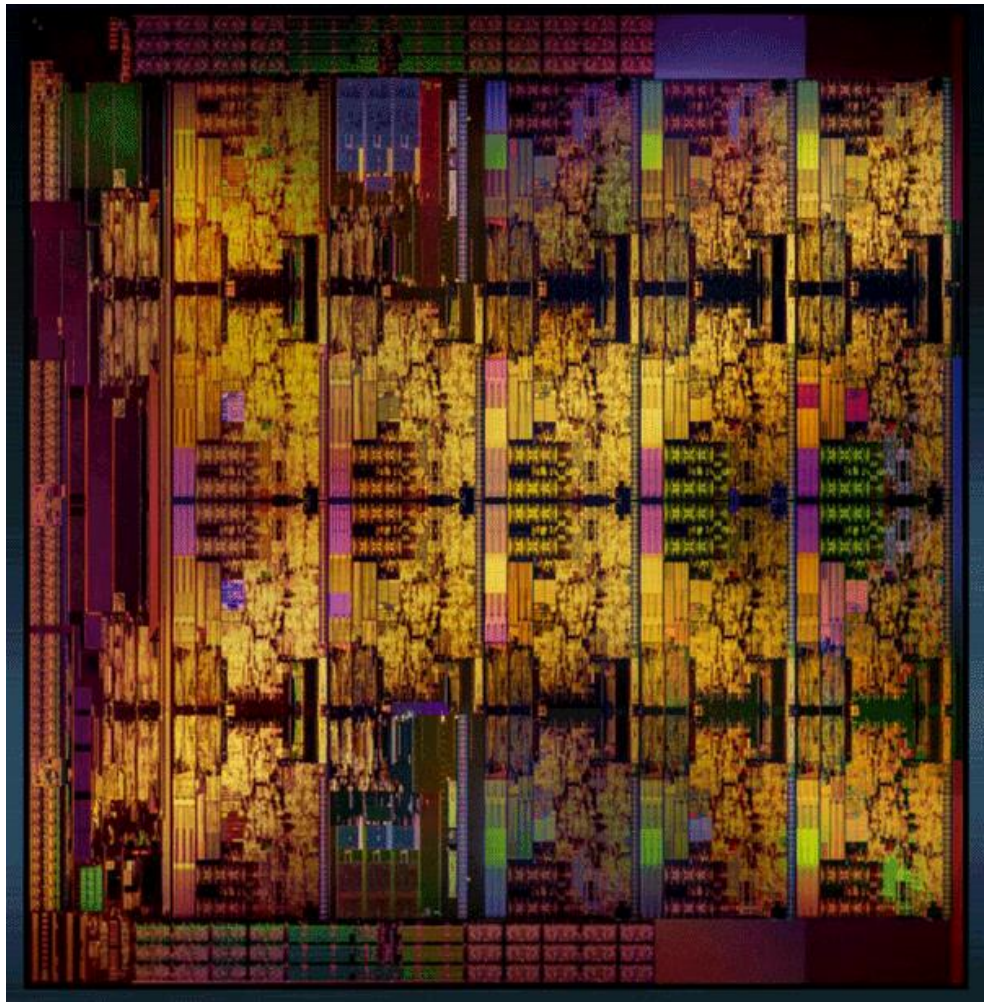
Early Integrated Circuits based on Gate Arrays

- Row of Standard gates – Connect or Disconnect between gates to form customer specific circuits



- Can be full-custom
  - Completely fabricated from scratch
- Can-be semi-custom
  - Customization on the metal layers only
- Once fabricated, the design is fixed.

Modern Digital Design – Full custom Integrated Circuits



- Intel Core I9 Die Shot
- >20 billion transistors
- Expensive to design and manufacture
- Highly risky
- Hard to change the design
- Not viable unless the market is huge

- Most applications cannot afford to embark on such a design

> FPGA can be solution.

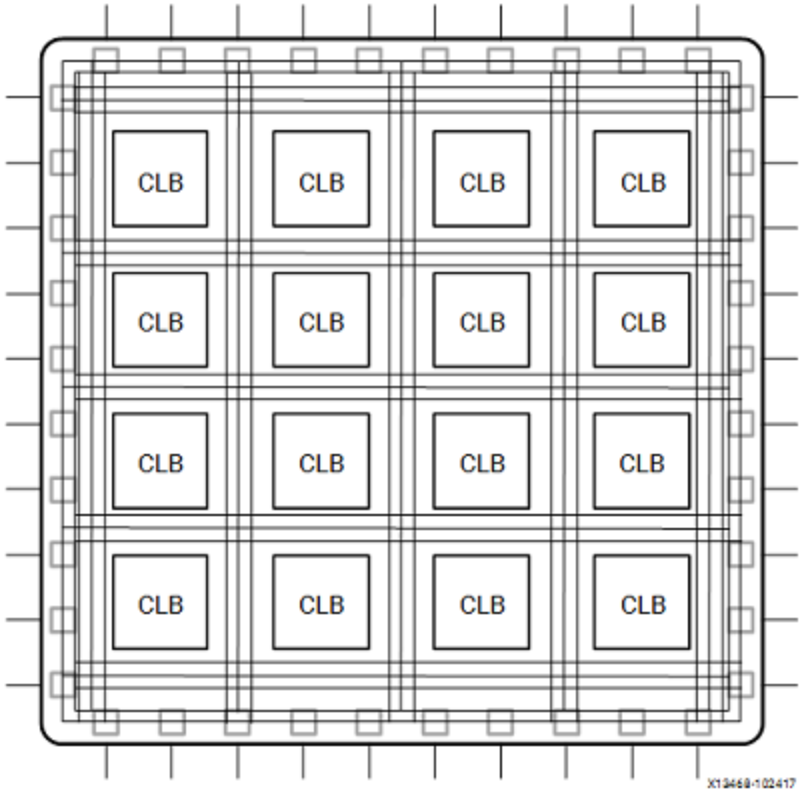
Field Programmable Gate Arrays (FPGAs)

- Combining idea from Programmable Logic Devices (PLDs) and gate arrays
- First introduced by Xilinx (1985)
- **Pros**
  - Fast (Parallelism)
  - Flexibility
  - Power-efficient
  - Low prototyping cost

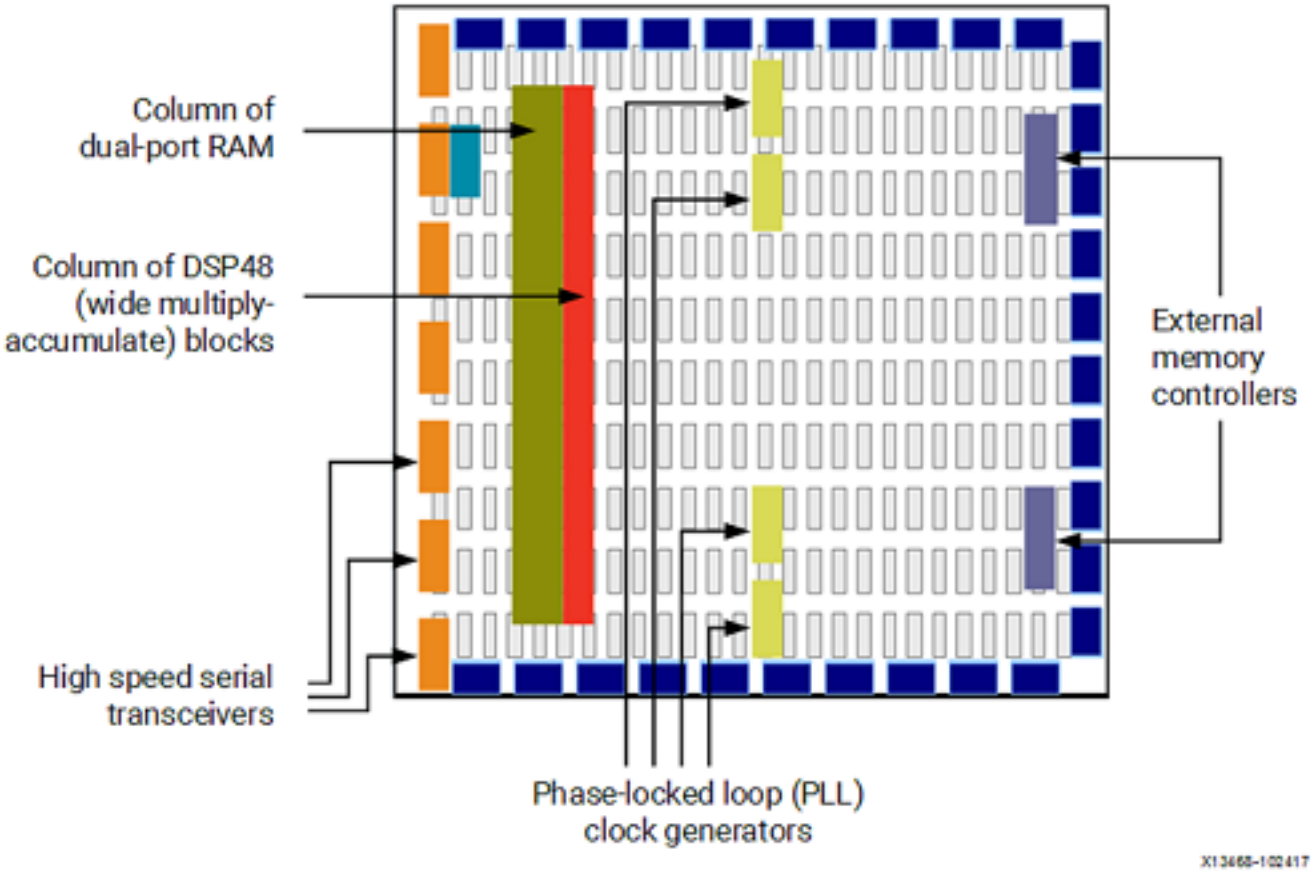
- **Applications**
  - ASIC/DSP prototyping (Simulation > FPGA Hardware > ASIC/DSP Hardware)
  - A.I. accelerator: **FPGA** vs GPGPU
    - Pros: Performance / Power efficiency / Heat
    - Cons: Requiring hardware design knowledge.
    - (Recently, compiler from C language to HDL was released but you still need to know hardware design skill to achieve high performance.)

Field Programmable Gate Array (FPGA)

- Architecture



Basic FPGA Architecture



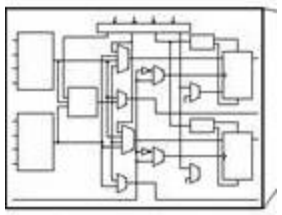
Modern FPGA Architecture

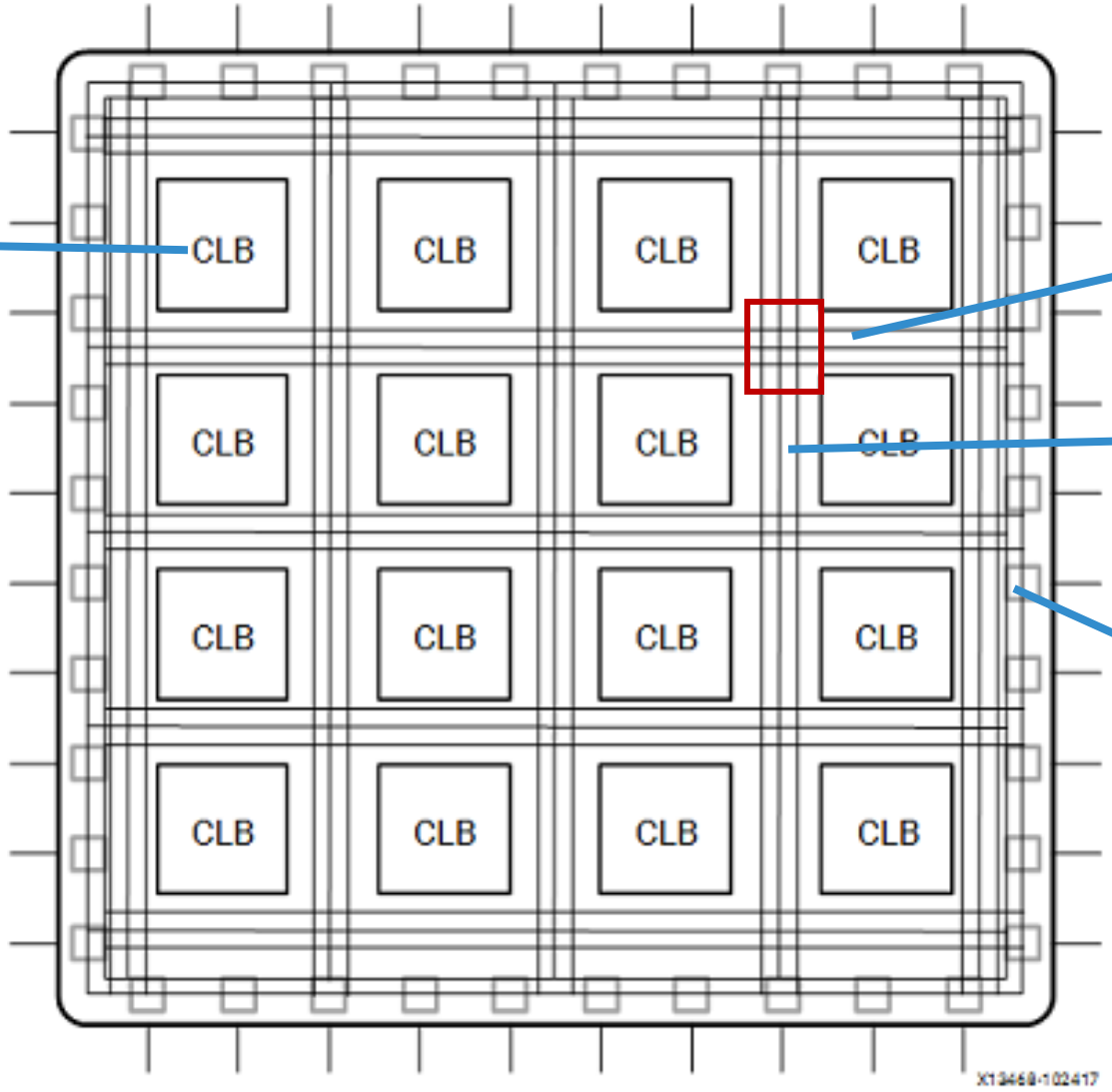
Field Programmable Gate Array (FPGA)

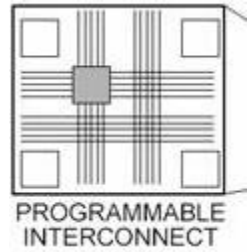
- Architecture

Configurable Logic Block (CLB)

- LUT + Flip-flops



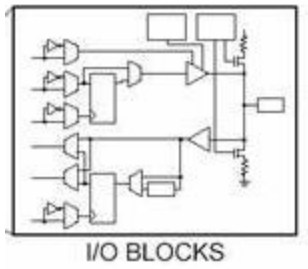




Switching Matrix

Routing Channels

I/O Pad

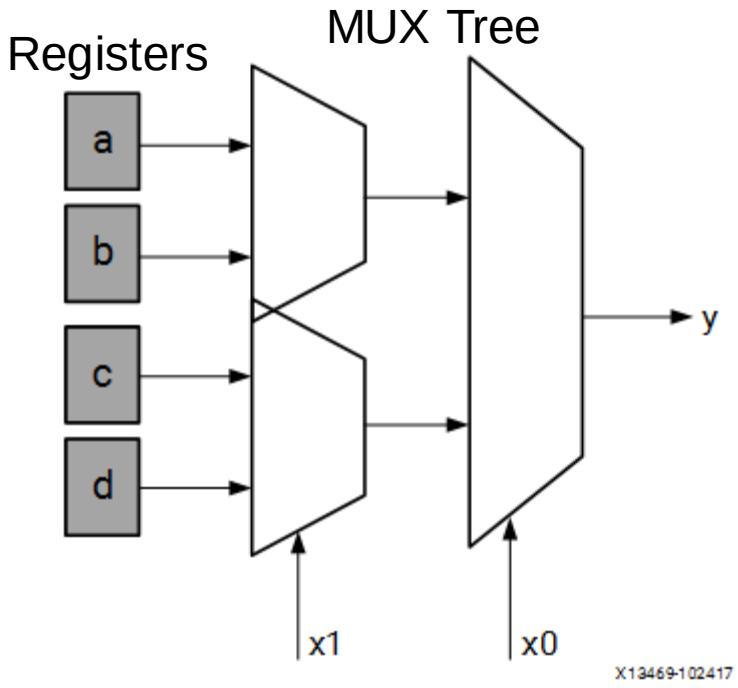


Field Programmable Gate Array (FPGA)

- Look-up table (LUT) - This element performs logic operations.
- Flip-Flop (FF) - This register element stores the result of the LUT.
- Wires - These elements connect elements to one another.
- Input/Output (I/O) pads - These physical ports get data in and out of the FPGA.

Look-up Table (LUT)

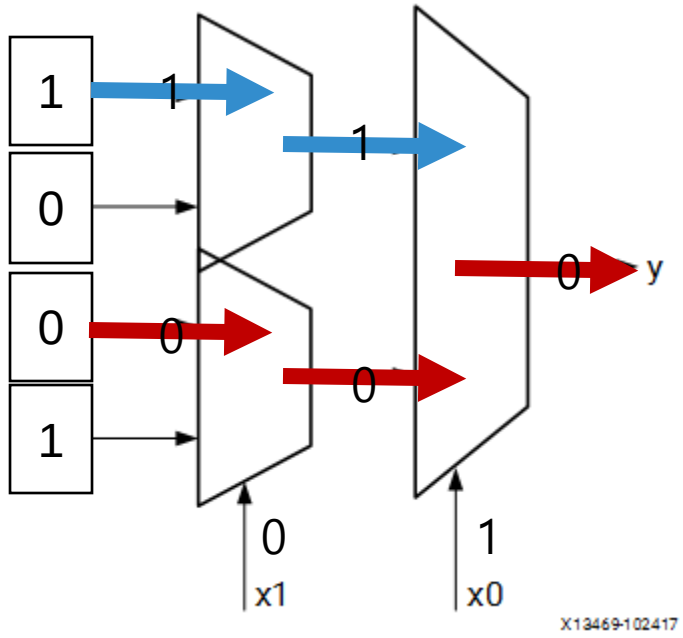
- LUT: The basic building block of an FPGA and is capable of implementing any logic function of N Boolean variables. (In modern Xilinx FPGA, N = 6)
- Truth Table! (Download a BITSTREAM, which is not a PROGRAM)
- NOTE: Discrete logic gates do not actually exist inside of an FPGA!



2-input LUT Structure

x0	x1	y
0	0	1
0	1	0
1	0	0
1	1	1

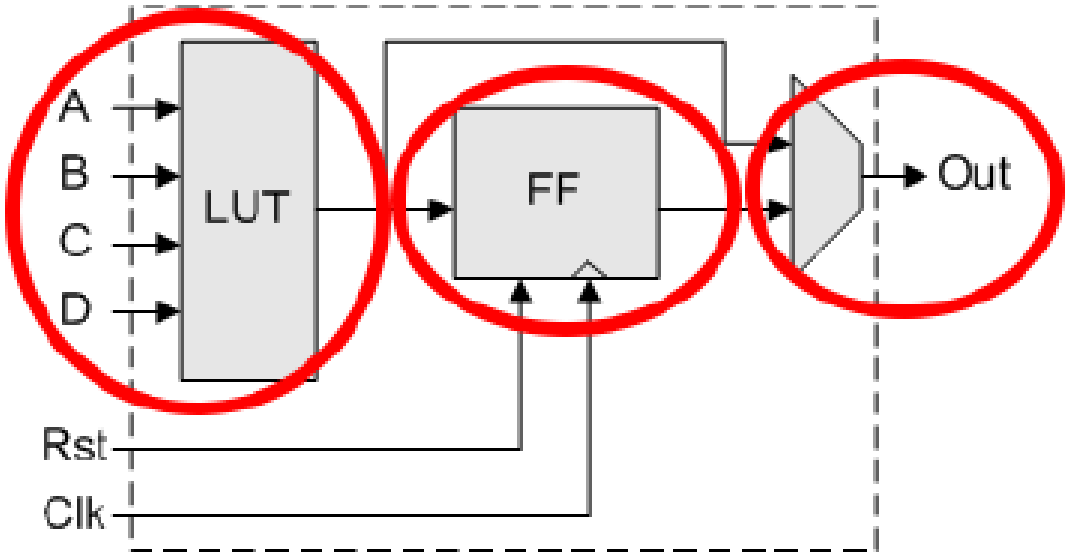
XNOR Truth Table



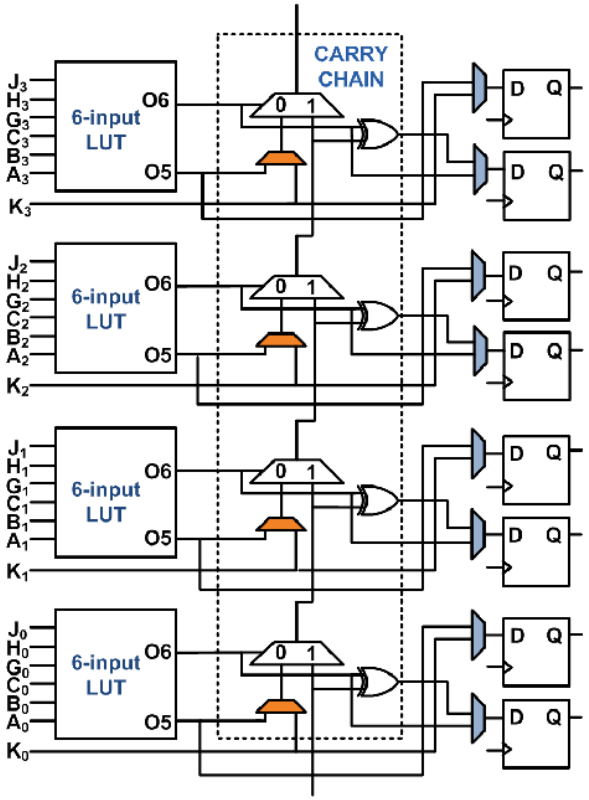
ex) LUT implementation of XNOR gate

Configurable Logic Block (CLB)

- Based around Look-up Tables
- Optional D-Flipflop at the output of the LUT
- Special Circuit for cascading logic blocks (ex, carry-chain)

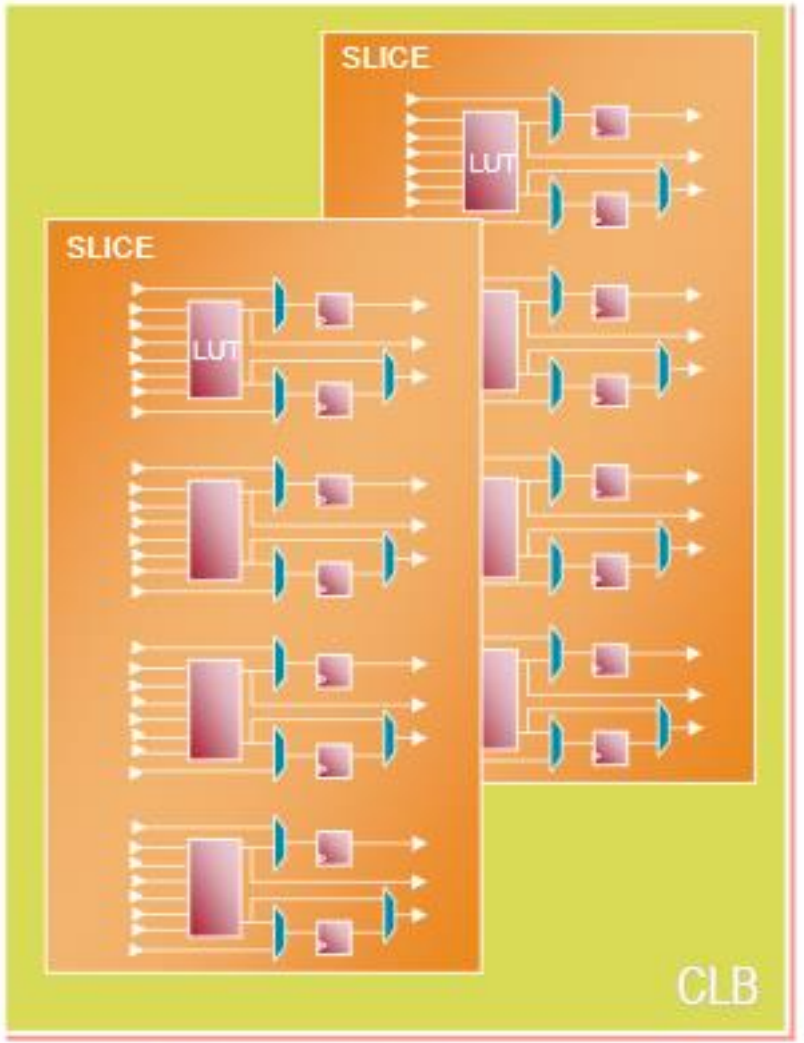


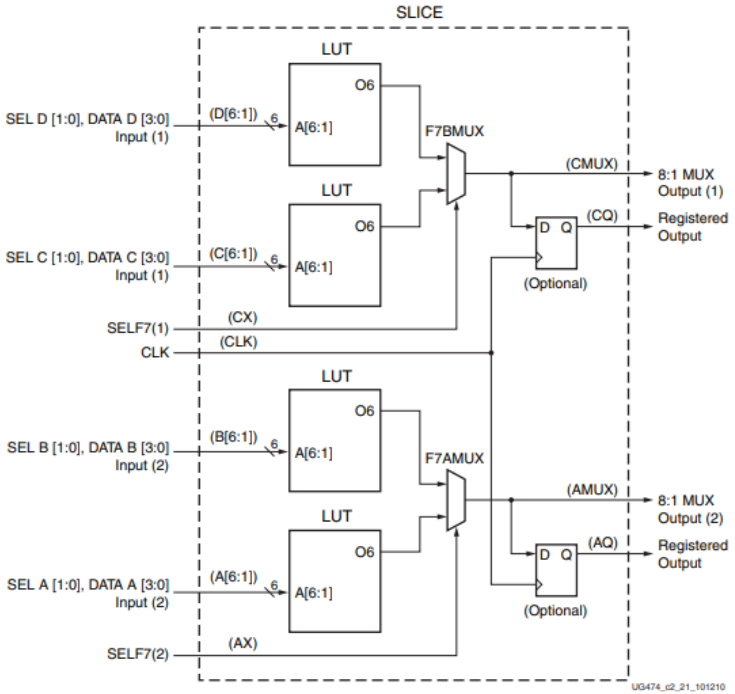
CLB with 4-input LUT and D-flipflop



Configurable Logic Block (CLB)

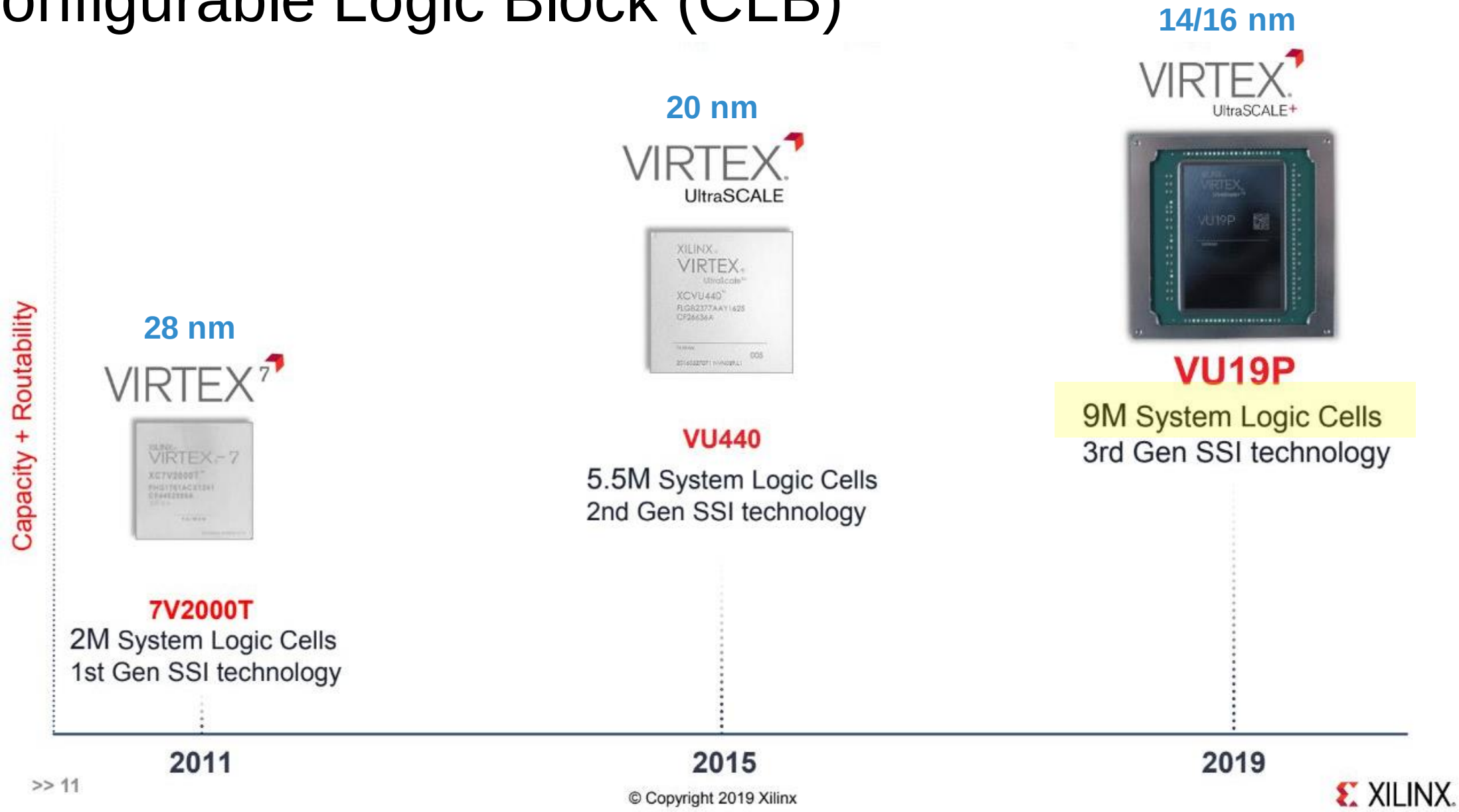
- Two side-by-side slices per CLB
  - Slice_M are memory-capable
  - Slice_L are logic and carry only
- Four 6-input LUTs per slice
  - Consistent with previous architectures
  - Single LUT in Slice_M can be a 32-bit shift register or 64 x 1 RAM
- Two flip-flops per LUT
  - Excellent for heavily pipelined designs





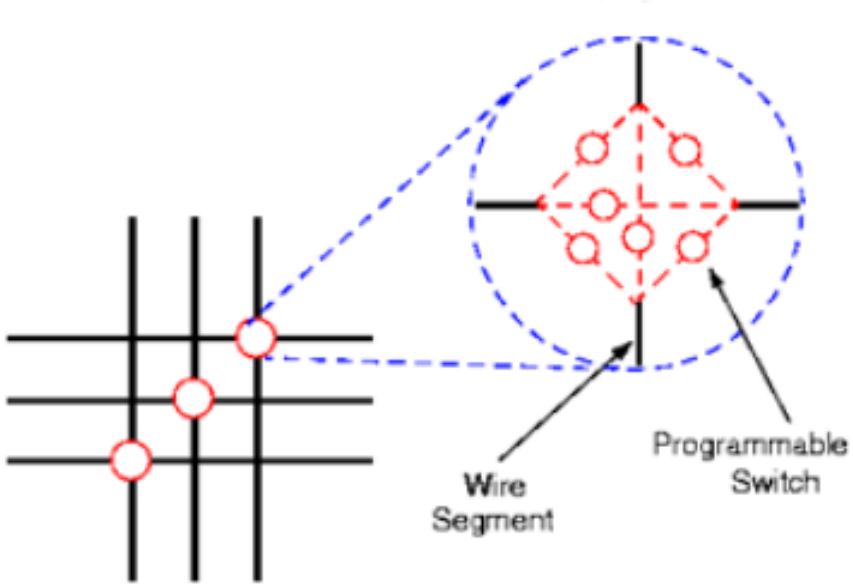
XILINX ALL PROGRAMMABLE

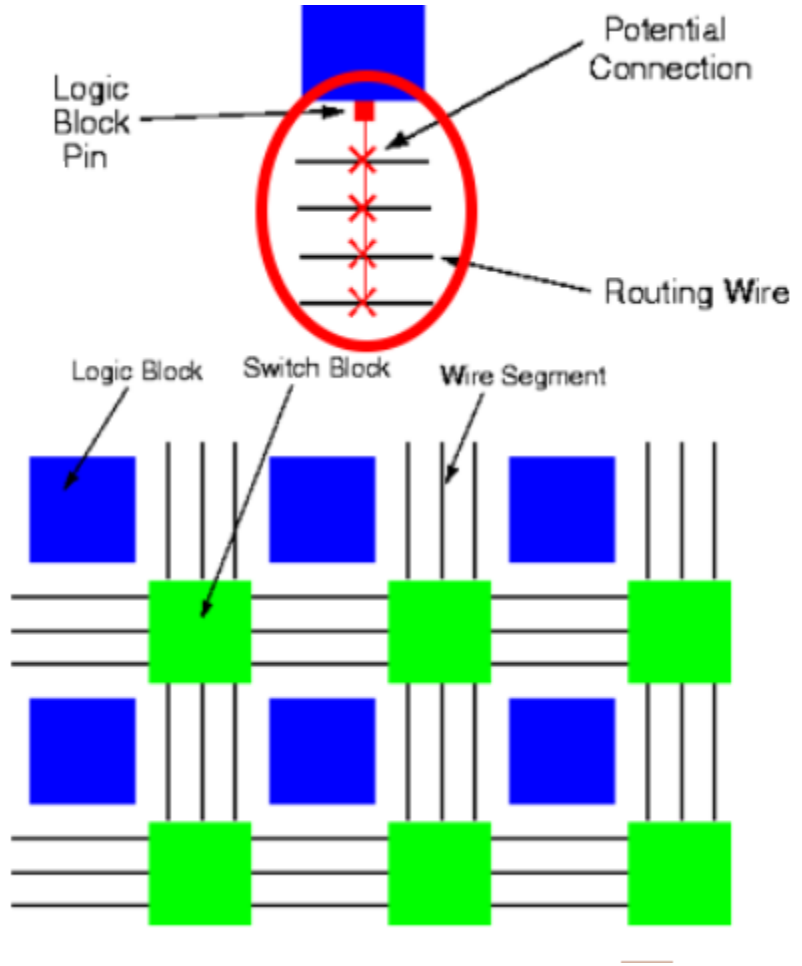
Configurable Logic Block (CLB)



Programmable Routing (Switching Matrix)

- Between rows and columns of logic blocks are wiring channels
- Programmable – a logic block pin can be connected to one of many wiring tracks through programmable switch
- Each wire segment can be connected in one of many ways

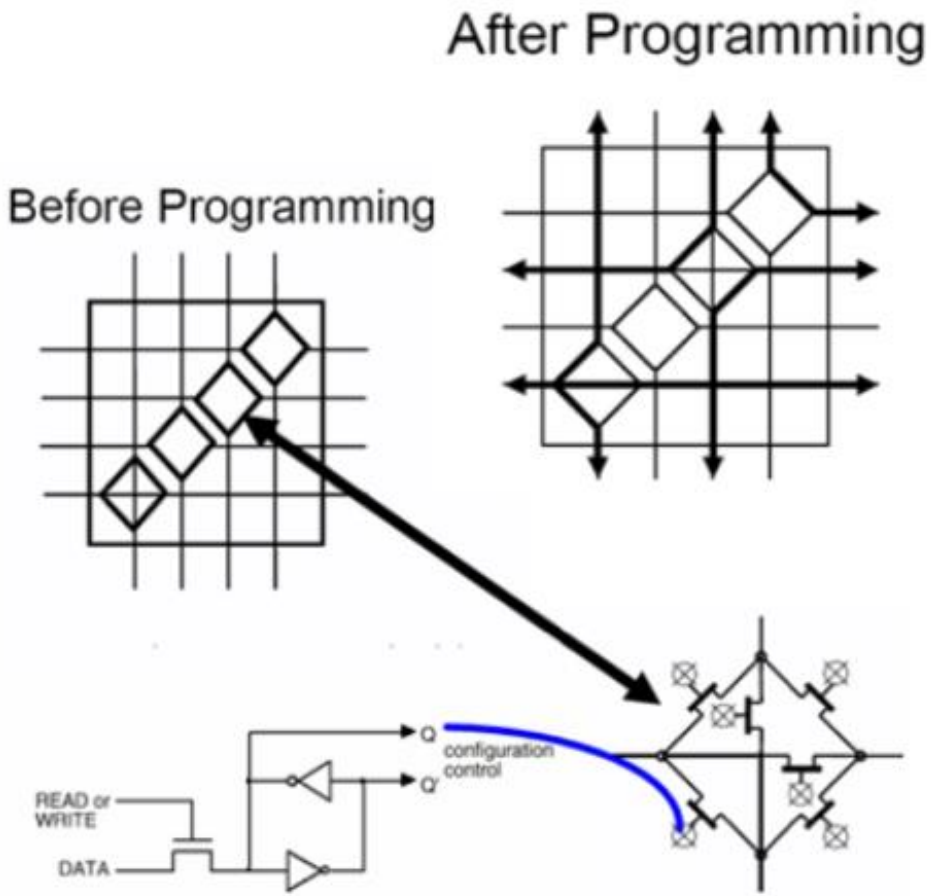




Programmable Routing (Switching Matrix)

- At each interconnect site, there is a transistor switch which is default OFF (not conducting)
- Each switch is controlled by the output of a 1-bit configuration register
- Configuring the routing is simply to put a ‘1’ or ‘0’ in this register to control the routing switches

- Configuration is volatile
- “Bitstream” is either stored on local flash memory or download via computer
- Configuration happens on Power-up



Modern FPGA

Kintex-7 FPGA Feature Summary

Table 6: **Kintex-7 FPGA Feature Summary by Device**

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP Slices(2)	Block RAM Blocks(3)			CMTs(4)	PCIe(5)	GTXs	XADC Blocks	Total I/O Banks(6)	Max User I/O(7)
		Slices(1)	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)						
XC7K70T	65,600	10,250	838	240	270	135	4,860	6	1	8	1	6	300
XC7K160T	162,240	25,350	2,188	600	650	325	11,700	8	1	8	1	8	400
XC7K325T	326,080	50,950	4,000	840	890	445	16,020	10	1	16	1	10	500
XC7K355T	356,160	55,650	5,088	1,440	1,430	715	25,740	6	1	24	1	6	300
XC7K410T	406,720	63,550	5,663	1,540	1,590	795	28,620	10	1	16	1	10	500
XC7K420T	416,960	65,150	5,938	1,680	1,670	835	30,060	8	1	32	1	8	400
XC7K480T	477,760	74,650	6,788	1,920	1,910	955	34,380	8	1	32	1	8	400

Notes:
1. Each 7 series FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM or SRLs.
2. Each DSP slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator.
3. Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks.
4. Each CMT contains one MMCM and one PLL.
5. Kintex-7 FPGA Interface Blocks for PCI Express support up to x8 Gen 2.
6. Does not include configuration Bank 0.
7. This number does not include GTX transceivers.

Table 7: **Kintex-7 FPGA Device-Package Combinations and Maximum I/Os**

Package(1)	FBG484			FBG676(2)			FFG676(2)			FBG900(3)			FFG900(3)			FFG901			FFG1156		
Size (mm)	23 x 23			27 x 27			27 x 27			31 x 31			31 x 31			31 x 31			35 x 35		
Ball Pitch (mm)	1.0			1.0			1.0			1.0			1.0			1.0			1.0		
Device	GTX(4)	I/O		GTX(4)	I/O		GTX	I/O		GTX(4)	I/O		GTX	I/O		GTX	I/O		GTX	I/O	
		HR(5)	HP(6)		HR(5)	HP(6)		HR(5)	HP(6)		HR(5)	HP(6)		HR(5)	HP(6)		HR(5)	HP(6)		HR(5)	HP(6)
XC7K70T	4	185	100	8	200	100															
XC7K160T	4	185	100	8	250	150	8	250	150												
XC7K325T				8	250	150	8	250	150	16	350	150	16	350	150						
XC7K355T																24	300	0			
XC7K410T				8	250	150	8	250	150	16	350	150	16	350	150						
XC7K420T																28	380	0	32	400	0
XC7K480T																28	380	0	32	400	0

Notes:
1. All packages listed are Pb-free (FBG, FFG with exemption 15). Some packages are available in Pb option.
2. Devices in FBG676 and FFG676 are footprint compatible.
3. Devices in FBG900 and FFG900 are footprint compatible.
4. GTX transceivers in FB packages support the following maximum data rates: 10.3Gb/s in FBG484; 6.6Gb/s in FBG676 and FBG900. Refer to *Kintex-7 FPGAs Data Sheet: DC and AC Switching Characteristics* (DS182) for details.
5. HR = High-range I/O with support for I/O voltage from 1.2V to 3.3V.
6. HP = High-performance I/O with support for I/O voltage from 1.2V to 1.8V.

Hardware Description Languages (HDLs)

Verilog:

```
1
2
3  always @({S0,S1}, A, B, C, D)
4        case ({S0,S1})
5           2'b00: Y = A;
6           2'b01: Y = B;
7           2'b10: Y = C;
8           2'b11: Y = D;
9        endcase
10
```

Verilog	VHDL
ASIC Designs	FPGA Designs
Weakly Typed	Strongly Typed
Low Verbosity	High Verbosity
Partially Deterministic	Very Deterministic
More “C” like	Non “C” like

VHDL:

```
2  process ({S0,S1},A,B,C,D)
3  begin
4     case {S0,S1}, is
5        when "00" => Y <= A;
6        when "01" => Y <= B;
7        when "10" => Y <= C;
8        when "11" => Y <= D;
9        when others => Y <= A;
10    end case;
11 end process;
```

IEEE standard (1364)
(https://standards.ieee.org/standard/1364-2005.html)

IEEE standard (1076)
(https://standards.ieee.org/standard/1076-2019.html)

Originated from Cadence (Industry)

Originated from USA DoD

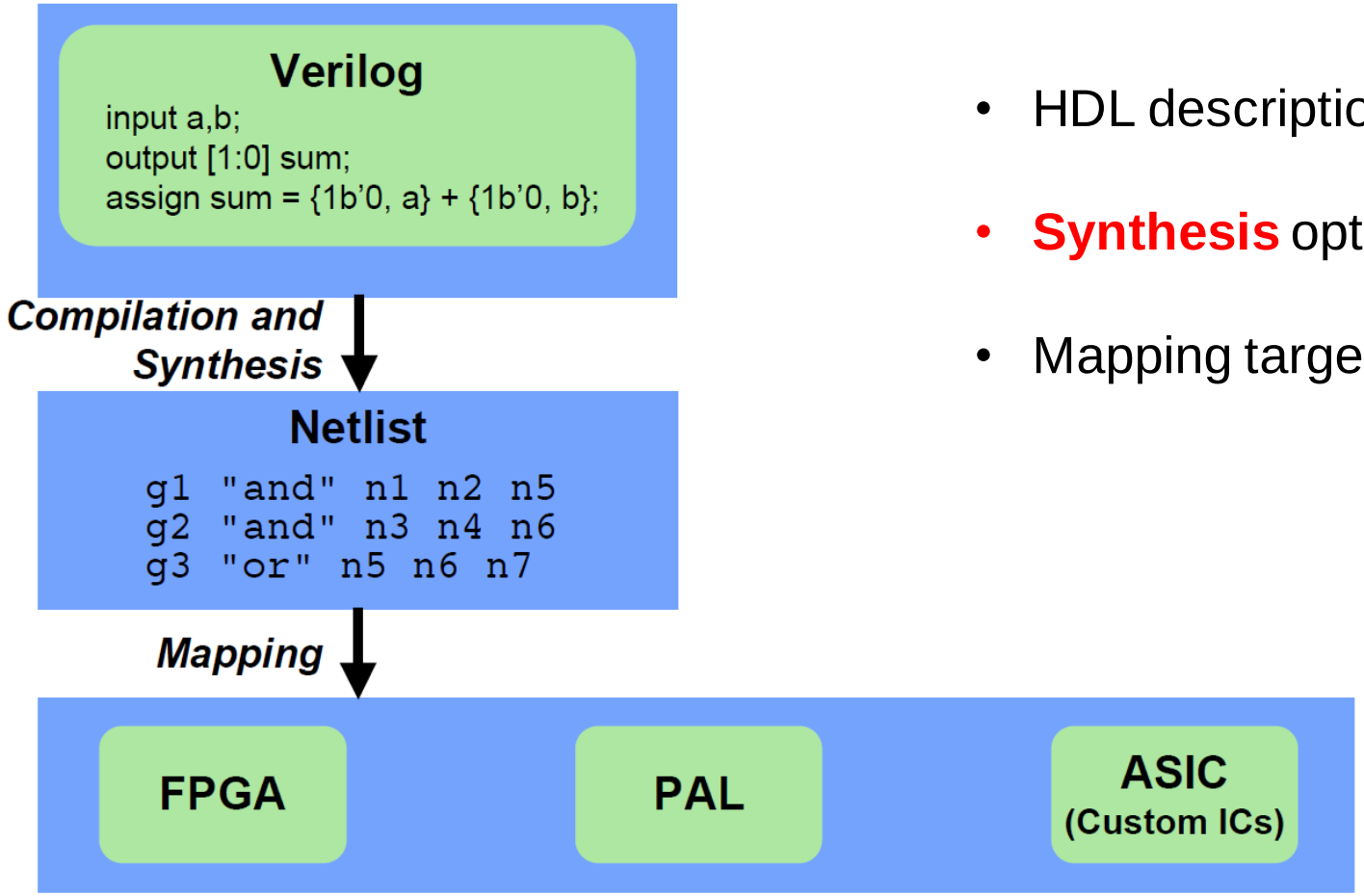
Other HDL..

- *ABEL (Advanced Boolean Expression Language)*
- *AHDL (Altera HDL)*
- *JHDL (Java HDL)*
- *MyHDL (Python-based HDL).*
- *Etc.*

V = Very-High-Speed Integrated Circuits

HDLs and Synthesis

- Hardware Description Language (HDL) is a convenient, device-independent representation of digital logic



- HDL description is compiled into a **netlist**
- **Synthesis** optimizes the logic
- Mapping targets a specific hardware platform

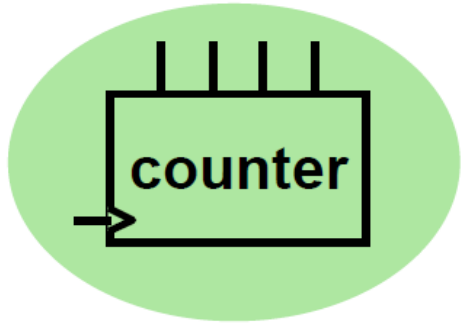
Synthesis and Mapping

- Infer macros: choose FPGA macros that efficiently implement various parts of the HDL code

```
...
always @ (posedge clk)
begin
    count <= count + 1;
end
...
```

HDL Code

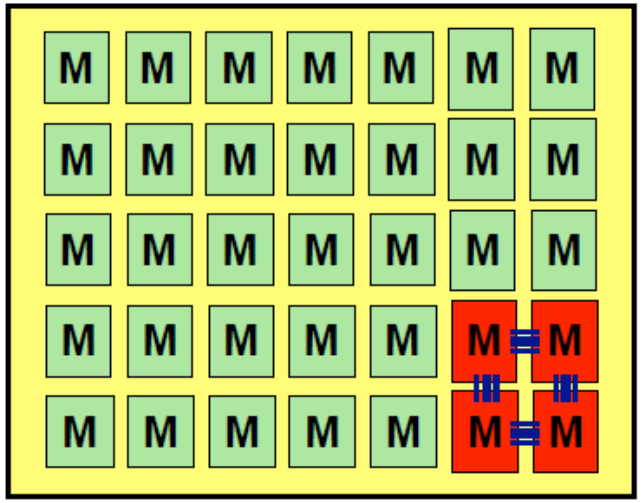
"This section of code looks like a counter. My FPGA has some of those..."



Inferred Macro

Synthesis and Mapping

- Place-and-Route: with area and/or speed in mind, choose the needed macros by location and route the interconnect

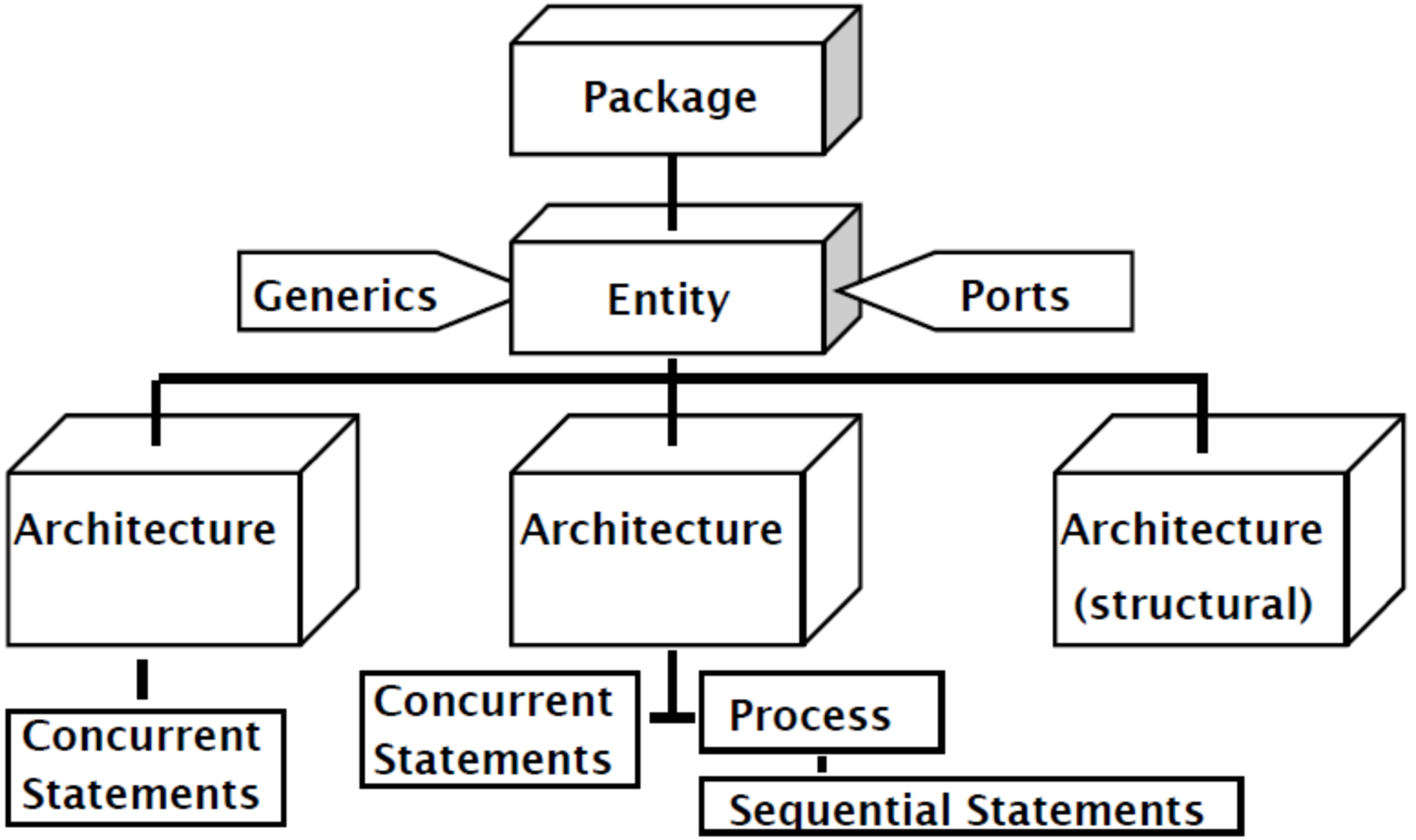


"This design only uses 10% of the FPGA. Let's use the macros in one corner to minimize the distance between blocks."

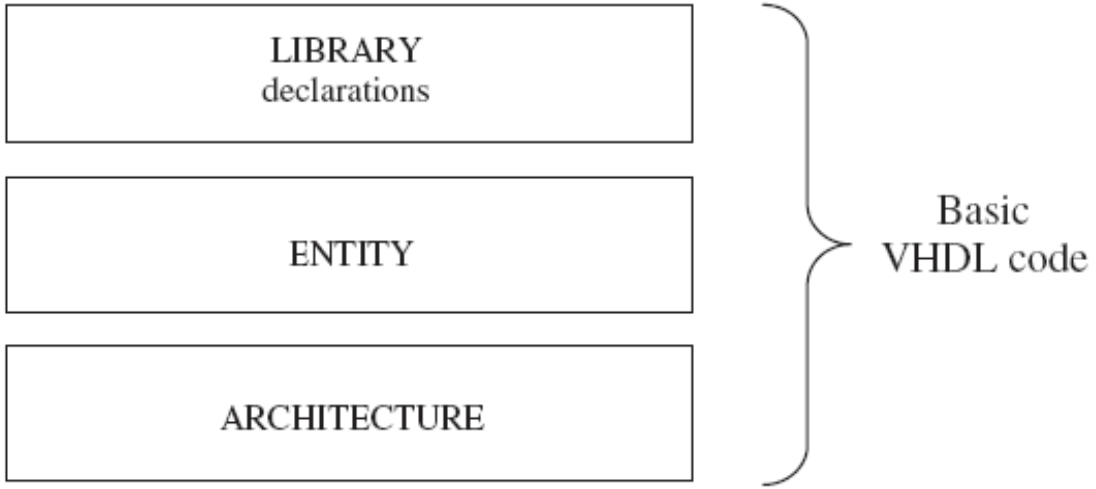
VHDL

- VHDL = VHSIC Hardware Description Language
  - VHSIC = Very High-Speed Integrated Circuit
- Developed in 1980's by US DOD
- ANSI/IEEE Standard 1076
- VHDL descriptions can be synthesized and implemented in programmable logic
  - NOTE: Synthesis tools can accept only subset of VHDL
- Some useful rules
  - VHDL is not case sensitive
  - Identifier must start with a letter
  - All statements end with a semi-colon
  - Comments precede with (--)
  - "<= " - signal assignment
  - ":=" - variable assignment
  - Signals have a one-time value set associated to it at any time.

VHDL Hierarchy



VHDL structure




```
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity encoder is
    Port (    a: IN std_logic;
              b: in std_logic;
              c: in std_logic;
              d: in std_logic;
              y: out std_logic_vector(8 downto 0));
end encoder;

architecture Behavioral of encoder is

begin
    y <=  "000000000" WHEN (a = '0' and b = '0' and c = '0' and d = '0') ELSE
          "000000001" WHEN (a = '0' and b = '0' and c = '0' and d = '1') ELSE
          "000000010" WHEN (a = '0' and b = '0' and c = '1' and d = '0') ELSE
          "000000100" WHEN (a = '0' and b = '0' and c = '1' and d = '1') ELSE
          "000001000" WHEN (a = '0' and b = '1' and c = '0' and d = '0') ELSE
          "000010000" WHEN (a = '0' and b = '1' and c = '0' and d = '1') ELSE
          "000100000" WHEN (a = '0' and b = '1' and c = '1' and d = '0') ELSE
          "001000000" WHEN (a = '0' and b = '1' and c = '1' and d = '1') ELSE
          "010000000" WHEN (a = '1' and b = '0' and c = '0' and d = '0') ELSE
          "100000000" WHEN (a = '1' and b = '0' and c = '0' and d = '1') ELSE
          "ZZZZZZZZZ";

end Behavioral;
```

- **LIBRARY declarations**
  - **Contains a list of all libraries to be used in the design. (ex:ieee,std,work.etc)**
- **ENTITY**
  - **Specifies the I/O pins of the circuit**
- **ARCHITECTURE**
  - **Contains the VHDL code proper, which describes how the circuit should behave(function)**

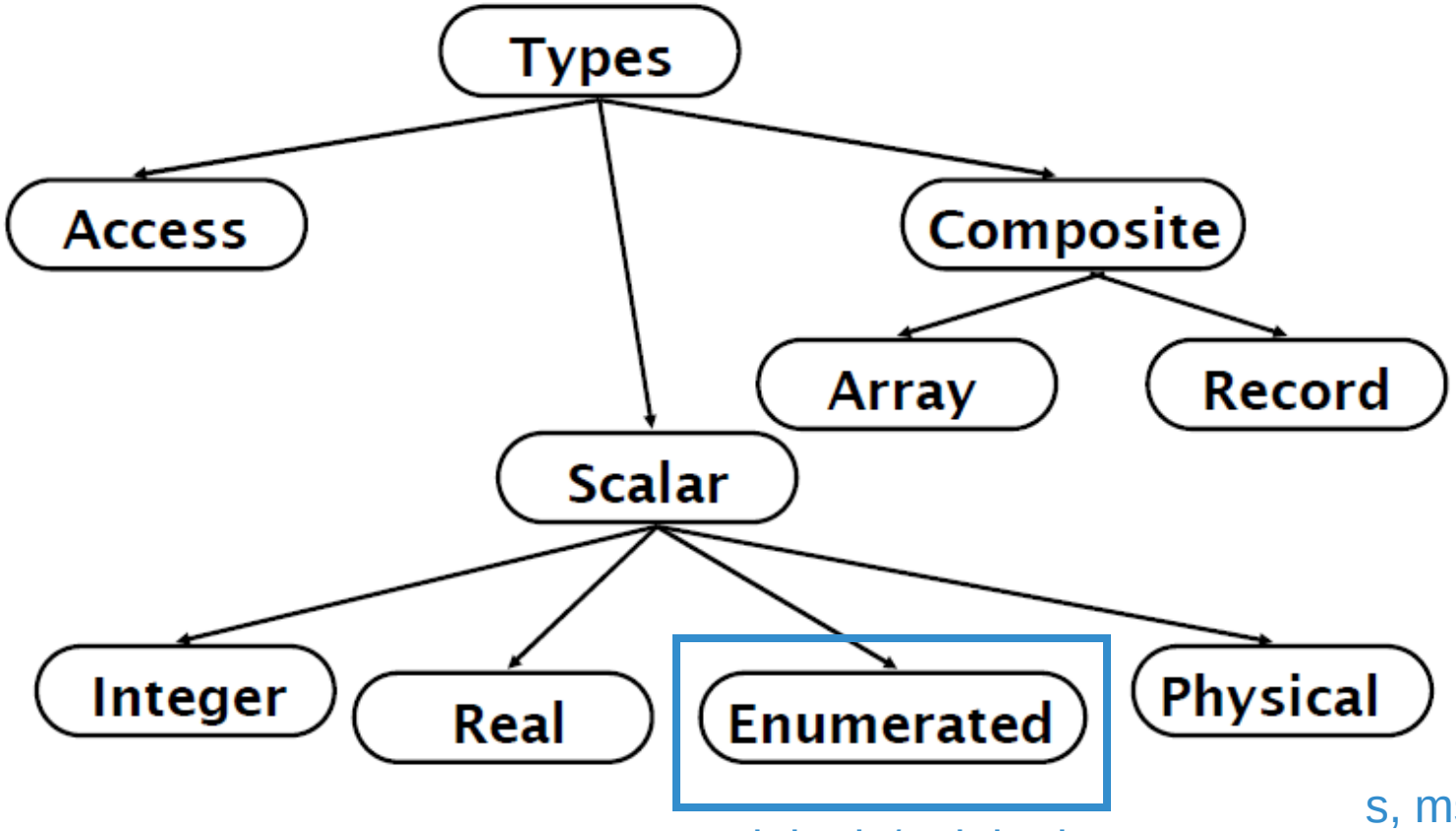
VHDL structure

- LIBRARY declarations (like stdio.h in C language)
  - usually needed in a design
    - LIBRARY ieee;
      - USE ieee.std_logic_1164.all; ← ***Semi-colon indicates the end of a statement declaration***
    - LIBRARY std;
      - USE std.standard.all;
    - LIBRARY work;
      - USE work.all;

VHDL structure 1: Library Packages

- **'std_logic_1164' package**
  - **std_logic (BIT)**
  - **std_logic_vector (MULTI-BITS)**
  - **integer**
- 'numeric_std' package
  - Mathematical operation & Signed/Unsigned
- 'textio' and 'std_logic_textio'
  - Used in testbench to write and read the data to the file
- Others
  - Fixed point & floating point…

Data Type



std_logic/std_logic_vector
boolean, bit, etc..

s, ms, us, ns, ps

Enumerated type

- type std_logic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'); *Represent "Wire"*
  - 1: Logic 1
  - 0: Logic 0
  - U: Uninitialized
  - X: Unknown
  - Z: High-Z
  - W: Weak signal, (can't tell if 0 or 1)
  - L: Weak 0, pull down
  - H: Weak 1, pull up
  - -: Don't care
- type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
- type state is (STATE1, STATE2, STATE 3,..);
- type boolean is (false,true); / type bit is ('0', '1');

std_logic vs std_logic_vector

- std_logic
  - 1 bit / 8 values

Represent "Wire"

- std_logic_vector
  - downto / to

std_logic_vector(7 downto 0)

A(7) MSB	A(6)	A(5)	A(4)	A(3)	A(2)	A(1)	A(0) LSB

std_logic_vector(0 to 7)

A(0) MSB	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7) LSB

Signal assignments

- constant a: integer := 523;

- signal b: std_logic_vector(11 downto 0);

```
b <= “000000010010”;
b <= B”000000010010”;
b <= B”0000_0001_0010”;
b <= X”012”;
b <= O”0022”;
```

Built-in operators

- Logic operators
  - AND, OR, NAND, NOR, XOR, XNOR (XNOR in VHDL'93 only!!)
- Relational operators
  - =, /=, <, <=, >, >=

Operators	Means	Operators	Means
=	Equal to	/=	Not equal to
<	Less than	>	Greater than
<=	Less than or equal to	>=	Greater than or equal to

- Addition operators
  - +, -, &
- Multiplication operators (DON'T USE IN THIS TERM PROJECT)
  - *, /, mod, rem
- Miscellaneous operators
  - **, abs, not

Assign operators

- Assign a value a SIGNAL

<=

- Assign a value a VARIABLE, CONSTANT, or GENERIC, establishing initial values

:=

- Assign a values to individual vector elements or with OTHERS

=>

example

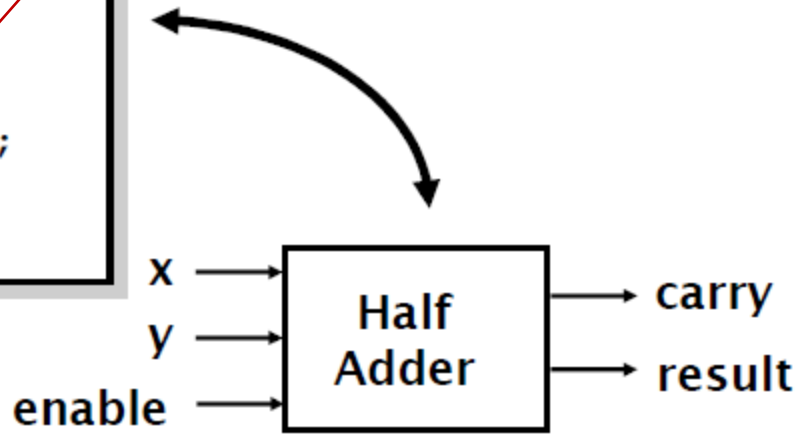
```
19  ----------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_ARITH.ALL;
23  use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25  ---- Uncomment the following library declaration if ins
26  ---- any Xilinx primitives in this code.
27  --library UNISIM;
28  --use UNISIM.VComponents.all;
29
30  entity test1 is
31      Port ( m_a : in STD_LOGIC_VECTOR(3 downto 0);
32             m_b : in STD_LOGIC_VECTOR(3 downto 0);
33             m_c : in STD_LOGIC;
34             m_x : out STD_LOGIC_VECTOR(3 downto 0);
35             m_y : out STD_LOGIC_VECTOR(6 downto 0));
36  end test1;
37
38  architecture Behavioral of test1 is
39
40  signal s_tmp1 : std_logic_vector(3 downto 0);
41  signal s_tmp2 : std_logic;
42
43  begin
44
45  s_tmp1 <= m_a + m_b;
46  s_tmp2 <= not m_c;
47
48  m_y(6 downto 3) <= "0101"; -- or x"5"
49  m_y(2) <= s_tmp2 and s_tmp1(3);
50  m_y(1 downto 0) <= s_tmp1(3 downto 2);
51
52  m_x(3) <= '1';
53  m_x(2 downto 0) <= "010";
54
55
56  end Behavioral;
57
```

VHDL structure 2: Entity Declaration

- An entity declaration describes the interface of the component.
- PORT clause indicates input and output ports.
- An entity can be thought of as a symbol for a component.

FYI, “std_logic” is popular

```
ENTITY half_adder IS

    PORT( x, y, enable: IN bit;
          carry, result: OUT bit);

END half_adder;
```

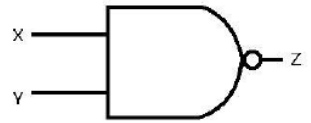


VHDL structure 3: Port Declaration

- PORT declaration establishes the interface of the object to the outside world.
- Three parts of the PORT declaration
  - Name
    - Any identifier that is not a reserved word.
  - Mode
    - In, Out, Inout, Buffer
  - Data type
    - Any declared or predefined datatype.
- Sample PORT declaration syntax:

```
ENTITY test IS
    PORT( name : mode data_type);
END test;
```

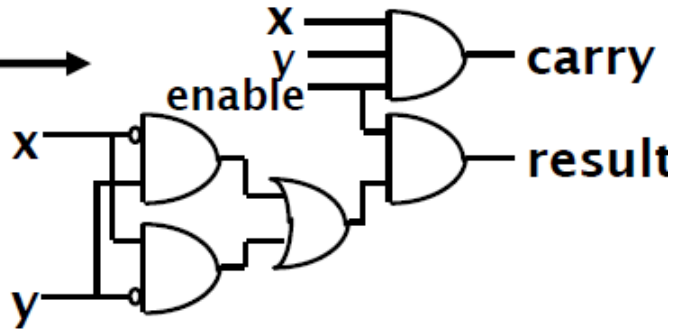
```
ENTITY nand_gate IS
    port (    x, y      : IN STD_LOGIC;
              z         : OUT STD_LOGIC);
END nand_gate;
```



VHDL structure 4: Architecture Declaration

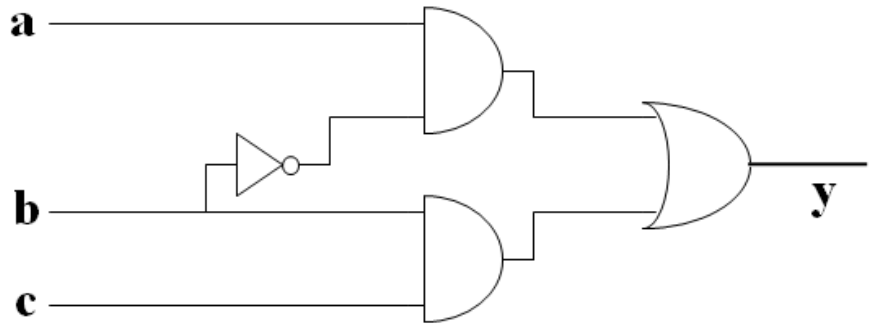
- Architecture declarations describe the operation of the component.
- Many architectures may exist for one entity, but only one may be active at a time.
- An architecture is similar to a schematic of the component.

```
ARCHITECTURE behave OF half_adder IS
BEGIN
    PROCESS (enable, x, y)
    BEGIN
      IF (enable = '1') THEN
        result <= x XOR y;
        carry  <= x AND y;
      ELSE
        carry  <=  '0';
        result <=  '0';
      END IF;
    END PROCESS;
END behave;
```

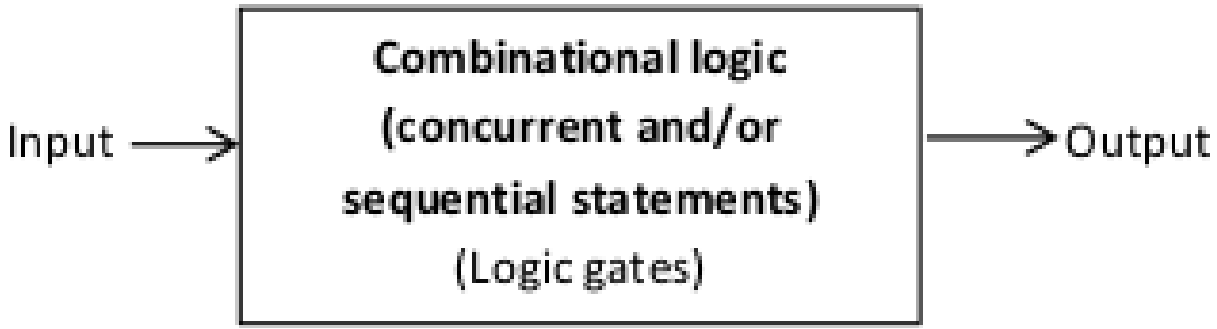


VHDL code structure : example

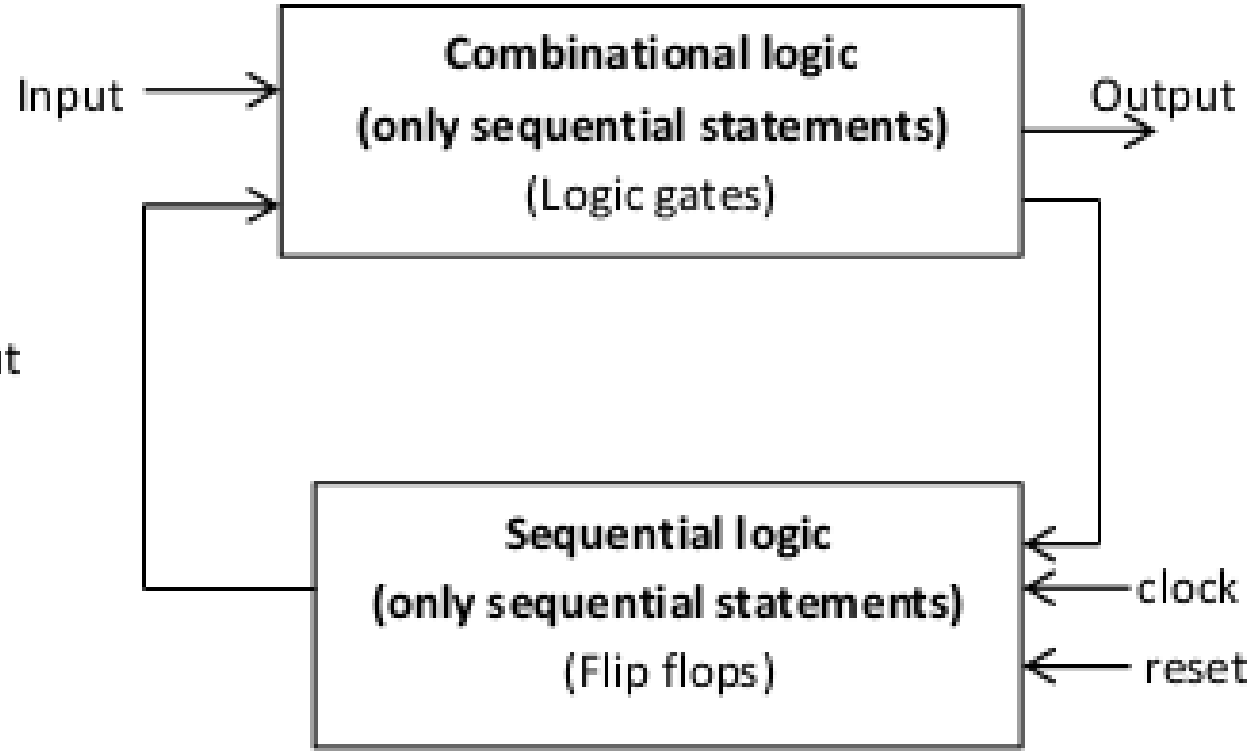
```
15  -- Revision:
16  -- Revision 0.01 - File Created
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_ARITH.ALL;
23  use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25  ---- Uncomment the following library declaration if instantiating
26  ---- any Xilinx primitives in this code.
27  --library UNISIM;
28  --use UNISIM.VComponents.all;
29
30  entity test1 is
31      Port ( m_a : in STD_LOGIC;
32             m_b : in STD_LOGIC;
33             m_c : in STD_LOGIC;
34             m_y : out STD_LOGIC);
35  end test1;
36
37  architecture Behavioral of test1 is
38
39  signal s_tmp : std_logic;
40
41  begin
42
43  s_tmp <= not m_b;
44  m_y <= (m_a and s_tmp) or (m_b and m_c);
45
46  end Behavioral;
47
48
```



Combinational logic vs Sequential Logic



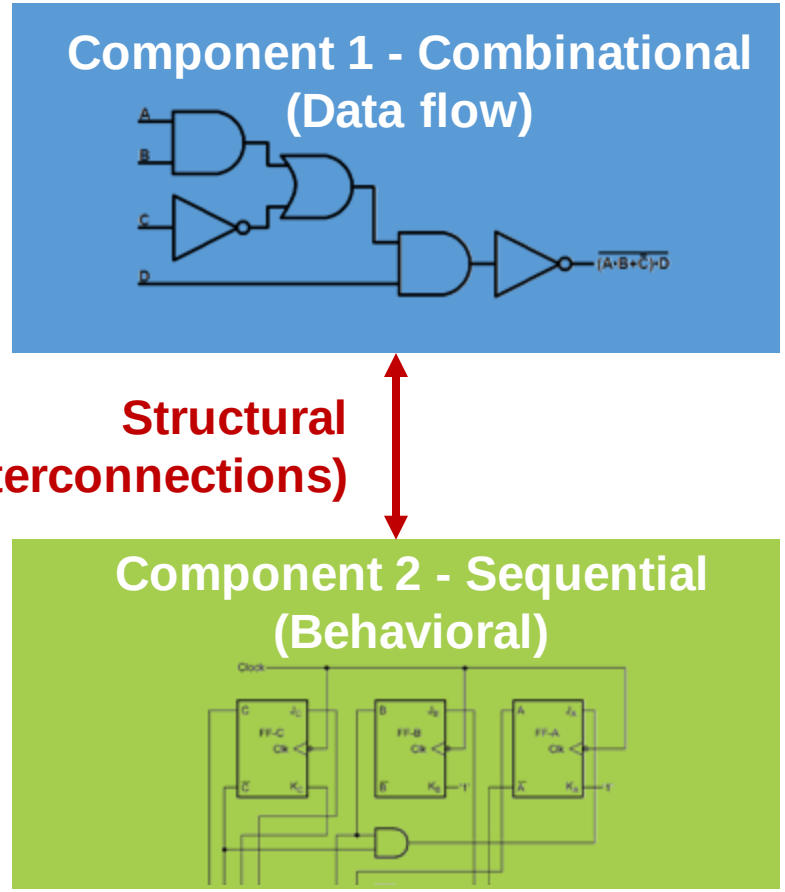
a) Combination design



b) Sequential design

Modeling Styles

- There are three modeling styles:
  - Dataflow (Combinational)
    - LOGIC
  - Behavioral (Sequential)
    - Only PROCESS statement
  - Structural
    - Interconnection between components



Modeling Styles

- Register-Transfer Level (RTL) Design
  - When designing digital integrated circuits with a HDL, the designs are usually engineered at a higher level of abstraction than transistor or logic gate level
  - But, a gate-level logic implementation (RTL design) is sometimes preferred.
  - This level describes the logic in terms of registers and the Boolean equations for the combinational logic between registers
  - For a combinational system there are no registers and the RTL logic consists only of combinational logic

Modeling Styles

Design	Statement	VHDL
Sequential (Flip-flops and Logic gates)	Sequential statements only	if
		case
		loop
		wait
Combinational (Logic gates Only)	Concurrent and Sequential	not/and/or/nand/nor/xor/xnor
		when-else
		with-select
		generate

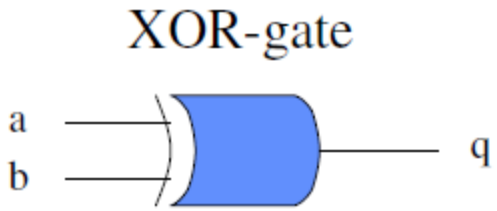
Concurrent vs Sequential Statements in VHDL

- Two different types of execution: sequential and concurrent.
- Different types of execution are useful for modeling of real hardware.
  - Supports various levels of abstraction.

- Sequential statements view hardware from a “programmer” approach.

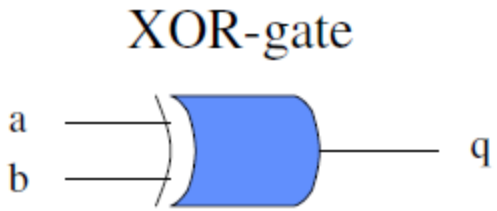
- Concurrent statements are order-independent and asynchronous.

Example: XOR-gate in Sequential Style




```
process(a,b)
begin
        if (a/=b) then
                q <= '1';
        else
                q <= '0';
        end if;
end process;
```

Example: XOR-gate in Dataflow Style

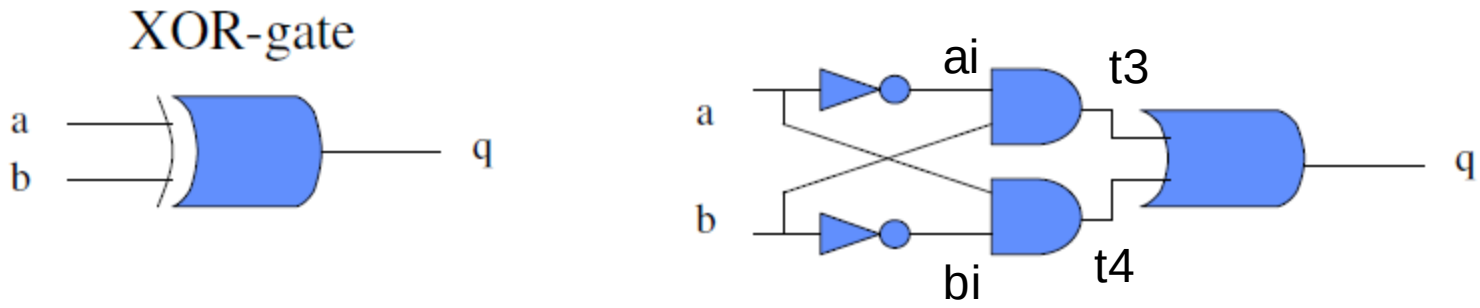



```
q <= a xor b;
```

Or in behavioral data flow style:

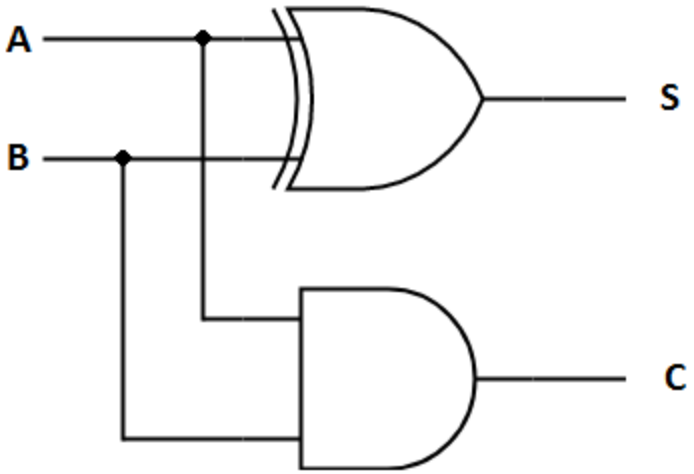
```
q <= '1' when a/=b else '0';
```

Example: XOR-gate in Structural Style



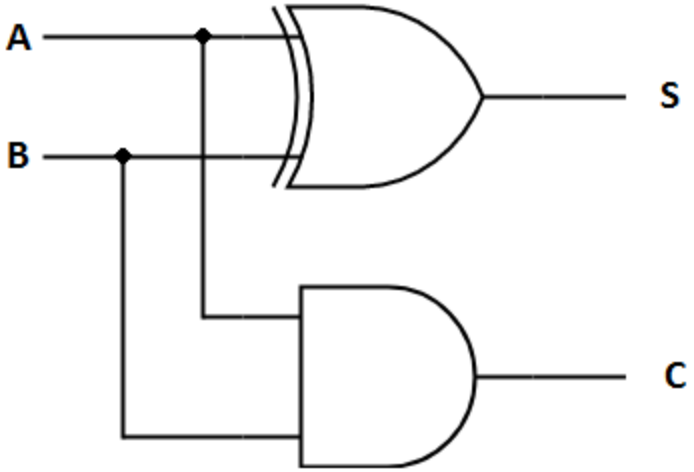

```
u1: inverter port map (a,ai);

u2: inverter port map (b,bi);

u3: and_gate port map (ai,b,t3);

u4: and_gate port map (bi,a,t4);

u5: or_gate port map (t3,t4,q);
```

Example: Half-adder in Sequential Style



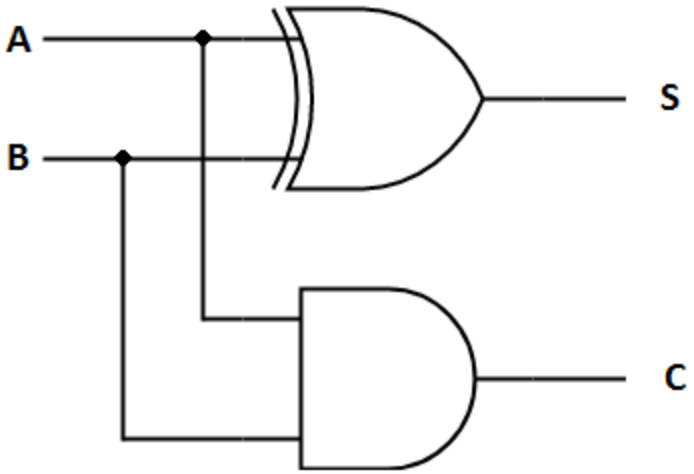

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity half_adder is
  port (a, b: in std_logic;
    sum, carry_out: out std_logic);
  end half_adder;

architecture behavior of half_adder is
  begin
    ha: process (a, b)
    begin
      if a = '1' then
        sum <= not b;
        carry_out <= b;
      else
        sum <= b;
        carry_out <= '0';
      end if;
    end process ha;

end behavior;
```

Example: XOR-gate in Dataflow Style




```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.
4.  entity half_adder is
5.    port (a, b: in std_logic;
6.      sum, carry_out: out std_logic);
7.    end half_adder;
8.
9.  architecture dataflow of half_adder is
10.   begin
11.   sum <= a xor b;
12.   carry_out <= a and b;
13. end dataflow;
```

Example: XOR-gate in Structural Style




```
u1: xor_gate port map (a, b, sum);
u2: and_gate port map (a, b, carry_out);
```

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

entity half_adder is                        -- Entity declaration for half adder
  port (a, b: in std_logic;
    sum, carry_out: out std_logic);
end half_adder;

architecture structure of half_adder is     -- Architecture body for half adder

  component xor_gate                        -- xor component declaration
    port (i1, i2: in std_logic;
      o1: out std_logic);
  end component;

  component and_gate                        -- and component declaration
    port (i1, i2: in std_logic;
      o1: out std_logic);
  end component;

begin
   u1: xor_gate port map (i1 => a, i2 => b, o1 => sum);
   u2: and_gate port map (i1 => a, i2 => b, o1 => carry_out);
-- We can also use Positional Association
--    => u1: xor_gate port map (a, b, sum);
--    => u2: and_gate port map (a, b, carry_out);
end structure;
```

Combinational: WHEN – Simple and Selected

- WHEN is a fundamental concurrent statements

- WHEN / ELSE – Simple WHEN

```
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...;
```

- In WITH / SELECT / WHEN,
  - OTHERS, UNAFFECTED are often useful

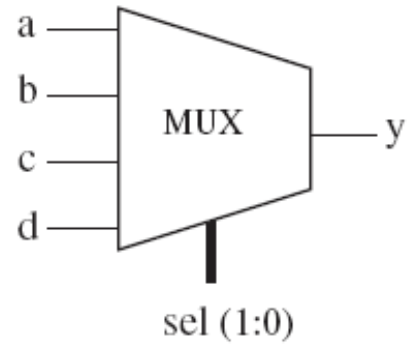
Example:

```
------ With WHEN/ELSE -------------------------
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
        "001" WHEN ctl='1' ELSE
        "010";

---- With WITH/SELECT/WHEN --------------------
WITH control SELECT
   output <= "000" WHEN reset,
             "111" WHEN set,
             UNAFFECTED WHEN OTHERS;
-----------------------------------------------
```

Combinational: WHEN – Simple and Selected

- Example: Multiplexer

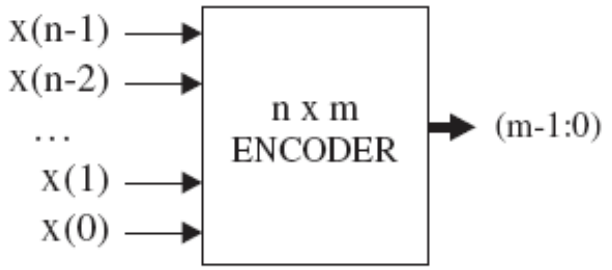
```
1  ------- Solution 1: with WHEN/ELSE --------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN STD_LOGIC;
7            sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8            y: OUT STD_LOGIC);
9  END mux;
10 ----------------------------------------
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13    y <=  a WHEN sel="00" ELSE
14          b WHEN sel="01" ELSE
15          c WHEN sel="10" ELSE
16          d;
17 END mux1;
18 ----------------------------------------
```



Combinational: WHEN – Simple and Selected

- Example: Encoder

```
1  ---- Solution 1: with WHEN/ELSE -------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------------
5  ENTITY encoder IS
6     PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
8  END encoder;
9  ----------------------------------------------
10 ARCHITECTURE encoder1 OF encoder IS
11 BEGIN
12    y <=   "000" WHEN x="00000001" ELSE
13           "001" WHEN x="00000010" ELSE
14           "010" WHEN x="00000100" ELSE
15           "011" WHEN x="00001000" ELSE
16           "100" WHEN x="00010000" ELSE
17           "101" WHEN x="00100000" ELSE
18           "110" WHEN x="01000000" ELSE
19           "111" WHEN x="10000000" ELSE
20           "ZZZ";
21 END encoder1;
22 ----------------------------------------------
```



Sequential Statements

- Sequentially Code are executed sequentially.
  - -With Sequential Code, We can build **Sequential Circuits** as well as **Combinational Circuits**.
  - -Allow only in **PROCESSES**, **FUNCTIONS**, and **PROCEDURES**.

Sequential Statements

- Syntax

```
PROCESS(sensitivity_list)
        declarations;
BEGIN
        sequential statement;
        sequential statement;
        ...
END PROCESS;
```

  - Sensitivity List
    - It is a list of signals to which the process responds. (i.e., is “sensitive to”)
    - If you put all input signals into the sensitivity list, a synthesizer will make it to combinational circuit. (But, it’s easy to make mistake. Don’t recommend.)
    - For Synchronous design, put only “CLOCK” and “reset” (if you want to design asynchronous reset)

Sequential Statements

- Example

```
34  entity top_test is
35     Port( reset : in std_logic;
36           clk : in std_logic;
37           en : in std_logic;
38           cnt_val : out std_logic_vector(3 downto 0));
39  end top_test;
40
41  architecture Behavioral of top_test is
42
43  signal s_cntval : std_logic_vector(3 downto 0);
44
45  begin
46
47  process(reset, clk)
48  begin
49     if reset='0' then
50        s_cntval <= "0000";
51     elsif rising_edge(clk) then
52        if en='1' then
53           s_cntval <= s_cntval + '1';
54        end if;
55     end if;
56  end process;
57
58  cnt_val <= s_cntval;
59
60  end Behavioral;
61
```

Synchronous Design (+ Asynchronous Reset)

Sequential Statements 1: IF

Syntax

```
IF conditions THEN assignments;
ELSIF conditions THEN assignments;
...
ELSE assignments;
END IF;
```

Example

```
IF (x<y) THEN temp <="11111111";
ELSIF (x=y AND w='0') THEN temp <="11110000";
ELSE temp <=(OTHERS =>'0');
```

Sequential Statements 2: CASE

Syntax

```
CASE identifier IS
WHEN value => assignments;
WHEN value => assignments;
...
END CASE;
```

Example

```
CASE control IS
        WHEN "00" => x<=a; y<=b;
        WHEN "01" => x<=b; y<=c;
        WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

Sequential Statements 2: CASE vs IF

- STATEMENT Type
  - CASE : Concurrent Code
  - IF : Sequential Code

- The IF/ELSE statement might infer the construction of an unnecessary priority decoder. (which would never occur with CASE)

  > It cause NEGATIVE CONSEQUENCE.!

Sequential Statements 2: CASE vs IF

Example – The Codes below implement the ~~same~~ **similar** circuit.

```
CASE sel IS
        WHEN "00" => x<=a;
        WHEN "01" => x<=b;
        WHEN "10" => x<=c;
        WHEN OTHERS => x<=d;
END CASE;
```

No priority

```
IF (sel="00") THEN x<=a;
ELSIF (sel="01") THEN x<=b;
ELSIF (sel="10") THEN x<=c;
ELSE x<=d;
END IF;
```

1st priority

↓

Last priority

Sequential Statements 2: CASE vs WHEN

	WHEN	CASE
Statement type	Concurrent	Sequential
Usage	Only outside PROCESSES, FUNCTIONS or PROCEDURES	Only inside PROCESSES, FUNCTIONS or PROCEDURES
All permutations must be tested	Yes for WITH/SELECT/WHEN	Yes
Max # of Assignments per test	1	Any
No-Action Key-Word	UNAFFECTED	NULL

Sequential Statements 2: CASE vs WHEN

Example – Equiv. Code from a **functional** point of view.

```
CASE sel IS
        WHEN "000" => x<=a;
        WHEN "001" => x<=b;
        WHEN "010" => x<=c;
        WHEN OTHERS => NULL;
END CASE;
```

```
WITH sel SELECT
        x <= a WHEN "000",
                b WHEN "001",
                c WHEN "010",
                UNAFFECTED WHEN OTHERS;
```

Sequential Statements 3: WAIT (for testbench)

Syntax

```
PROCESS // No Sensitivity List!
...
WAIT UNTIL signal_condition;
WAIT ON signal1 [, signal2, ... ];
WAIT FOR time;
```

- For use WAIT statement, **No Sensitivity List.**
- **WAIT UNTIL :** accepts only one signal.
- **WAIT ON :** accept multiple signal.
- **WAIT FOR :** waiting for specific time. (simulation only.)

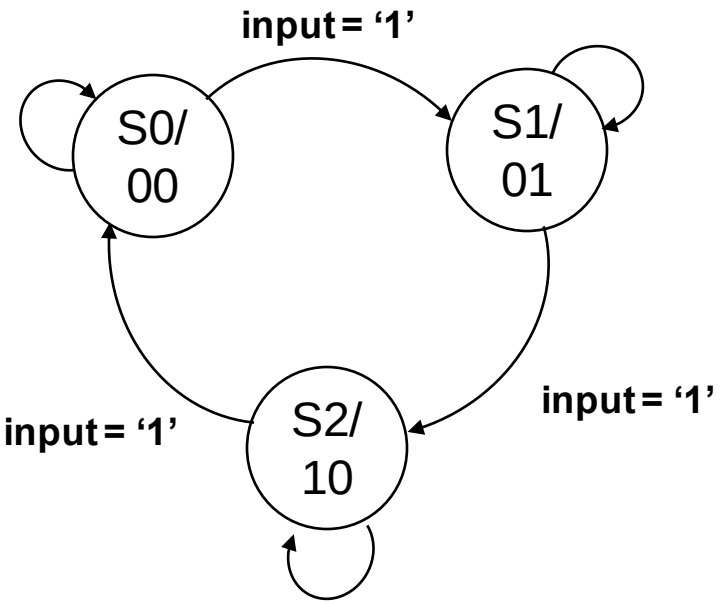
Example.

- **WAIT UNTIL :** WAIT UNTIL (clk'EVENT AND clk='1')
- **WAIT ON :** WAIT ON clk, rst;
- **WAIT FOR :** WAIT FOR 5ns;

Sequential Statements Example: Flip-flop

```
34  entity top_test is
35     Port( reset : in std_logic;
36           clk : in std_logic;
37           en : in std_logic;
38           din : in std_logic_vector(3 downto 0);
39           dout : out std_logic_vector(3 downto 0));
40  end top_test;
41
42  architecture Behavioral of top_test is
43
44  begin
45
46  process(reset, clk)
47  begin
48     if reset='0' then
49        dout <= "0000";
50     elsif rising_edge(clk) then
51        if en='1' then
52           dout <= din;
53        end if;
54     end if;
55  end process;
56
57  end Behavioral;
```

State Machine Example: (Moore)



Declaration

```
ENTITY state_machine IS
   PORT(
      clk      : IN   STD_LOGIC;
      input    : IN   STD_LOGIC;
      reset    : IN   STD_LOGIC;
      output   : OUT  STD_LOGIC_VECTOR(1 downto 0));
END state_machine;
ARCHITECTURE a OF state_machine IS
   TYPE STATE_TYPE IS (s0, s1, s2);
   SIGNAL state   : STATE TYPE;
```

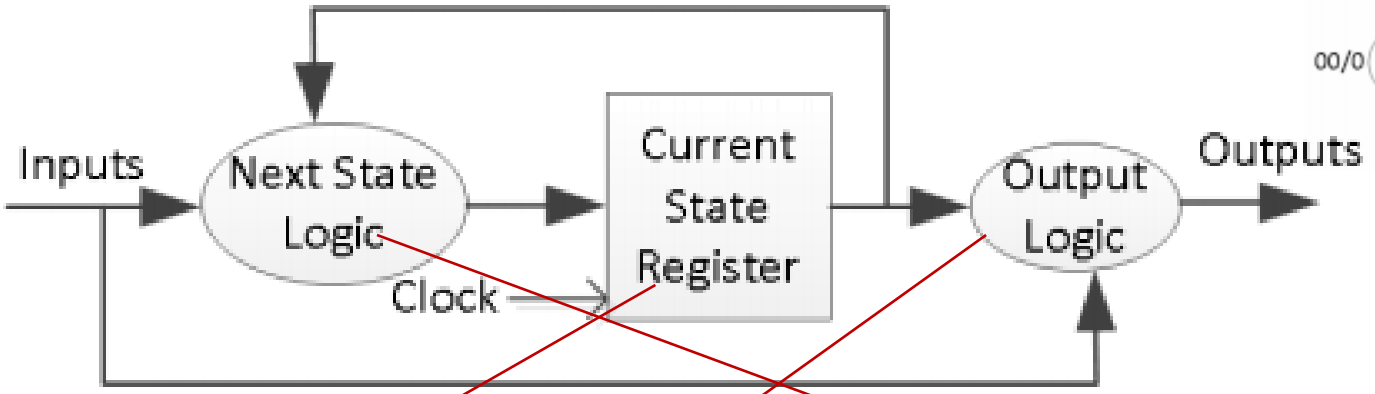
State Machine

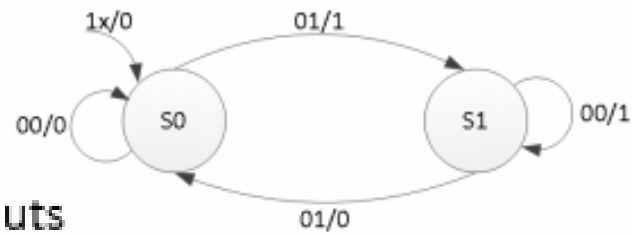
```
BEGIN
   PROCESS (clk, reset)
   BEGIN
      IF reset = '1' THEN
         state <= s0;
      ELSIF (clk'EVENT AND clk = '1') THEN
         CASE state IS
            WHEN s0=>
               IF input = '1' THEN
                  state <= s1;
               ELSE
                  state <= s0;
               END IF;
            WHEN s1=>
               IF input = '1' THEN
                  state <= s2;
               ELSE
                  state <= s1;
               END IF;
            WHEN s2=>
               IF input = '1' THEN
                  state <= s0;
               ELSE
                  state <= s2;
               END IF;
         END CASE;
      END IF;
   END PROCESS;
```

Output Decoder

```
   PROCESS (state)
   BEGIN
      CASE state IS
         WHEN s0 =>
            output <= "00";
         WHEN s1 =>
            output <= "01";
         WHEN s2 =>
            output <= "10";
      END CASE;
   END PROCESS;

END a;
```

Example : Mealy Machine



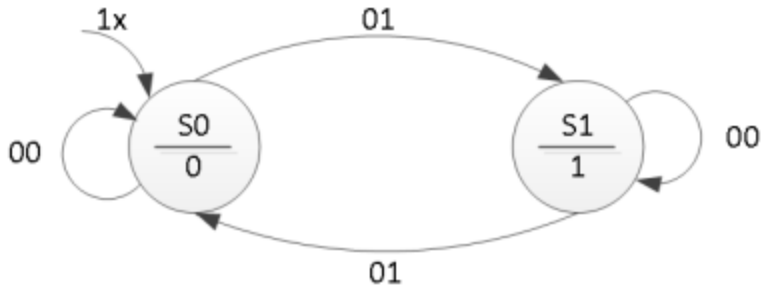


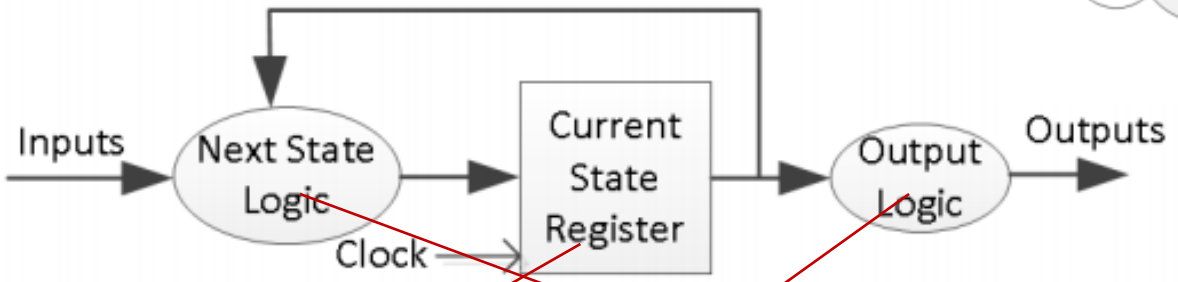

```
type state_type is (S0, S1);
signal state, next_state : state_type

begin

SYNC_PROC : process (clk)
begin
      if rising_edge(clk) then
            if (reset = '1') then
                  state <= S0;
            else
                  state <= next state
            end if;
      end if;
end process;
```

```
OUTPUT_DECODE : process (state, x)
begin
      parity <= '0';
      case (state) is
            when S0 =>
                  if (x = '1') then
                        parity <= '1'
                  end if;
            when S1 =>
                  if (x = '0') then
                        parity <= '1'
                  end if;
            when others =>
                  parity <= '0';
            end case;
end process;
```

```
NEXT_STATE_DECODE : process (state, x)
begin
      next_state <= S0;
      case (state) is
            when S0 =>
                  if (x = '1') then
                        next_state <= S1;
                  end if;
            when S1 =>
                  if (x = '0') then
                        next_state <= S1;
                  end if;
            when others =>
                  next_state <= S0;
            end case;
end process;
```

Example : Moore Machine



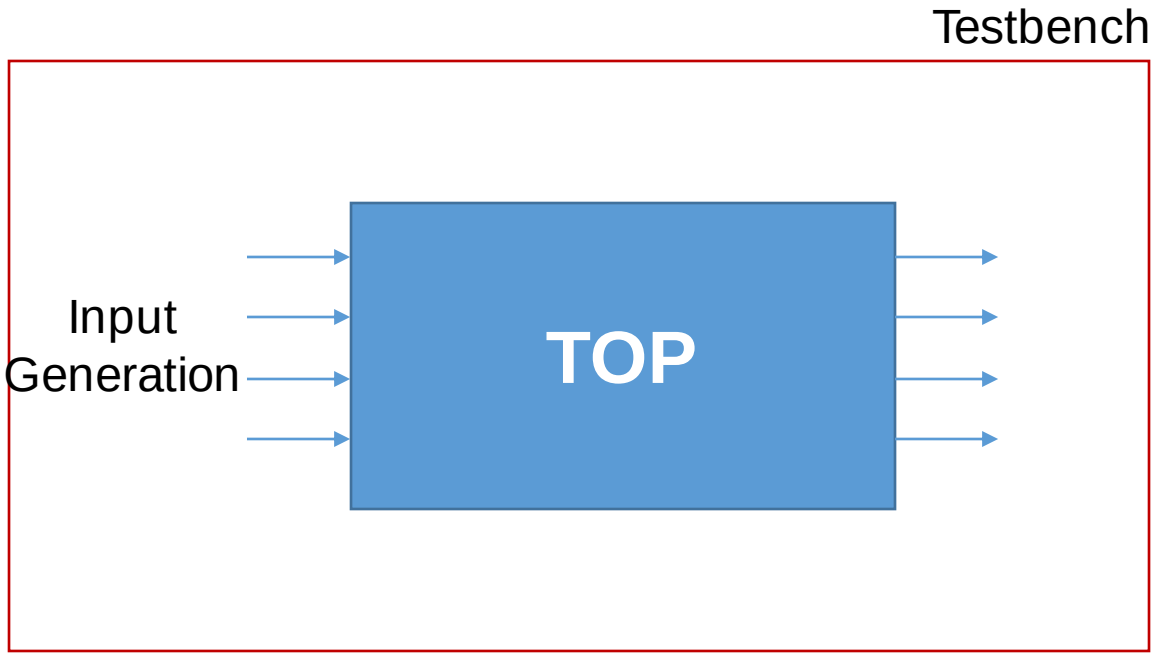



```
type state_type is (S0, S1);
signal state, next_state : state_type;

begin

SYNC_PROC : process (clk)
begin
      if rising_edge(clk) then
            if (reset = '1') then
                  state <= S0;
            else
                  state <= next_state;
            end if;
end process;
```

```
OUTPUT_DECODE : process (state)
begin
      case (state) is
            when S0 =>
                  parity <= '0';
            when S1 =>
                  parity <= '1';
            when others =>
                  parity <= '0';
      end case;
end process;
```

```
NEXT_STATE_DECODE : process (state, x)
begin
      next_state <= S0;
      case (state) is
            when S0 =>
                  if (x = '1') then
                        next_state <= S1;
                  end if;
            when S1 =>
                  if (x = '0') then
                        next_state <= S1;
                  end if;
            when others =>
                  next_state <= S0;
      end case;
end process;
```

Testbenches



Testbenches

```
35  ENTITY tb_lat4 IS
36  END tb_lat4;
37
38  ARCHITECTURE behavior OF tb_lat4 IS
39
40      -- Component Declaration for the Unit Under Test (UUT)
41
42      COMPONENT lat4
43      PORT(
44           reset : IN  std_logic;
45           clk : IN  std_logic;
46           en : IN  std_logic;
47           din : IN  std_logic_vector(3 downto 0);
48           dout : OUT  std_logic_vector(3 downto 0)
49          );
50      END COMPONENT;
51
52
53     --Inputs
54     signal reset : std_logic := '0';
55     signal clk : std_logic := '0';
56     signal en : std_logic := '0';
57     signal din : std_logic_vector(3 downto 0) := (others => '0');
58
59     --Outputs
60     signal dout : std_logic_vector(3 downto 0);
61
62     -- Clock period definitions
63     constant clk_period : time := 10 ns;
64
65  BEGIN
66
67     -- Instantiate the Unit Under Test (UUT)
68     uut: lat4 PORT MAP (
69            reset => reset,
70            clk => clk,
71            en => en,
72            din => din,
73            dout => dout
74          );
```

Initial values

```
75
76     -- Clock process definitions
77     clk_process :process
78     begin
79          clk <= '0';
80          wait for clk_period/2;
81          clk <= '1';
82          wait for clk_period/2;
83     end process;
84
85
86     -- Stimulus process
87     stim_proc: process
88     begin
89          -- hold reset state for 100 ns.
90          wait for 100 ns;
91
92             reset <= '1';
93          wait for 50 ns;
94             din <= "0101";
95             en <= '1';
96
97          wait for 50 ns;
98             din <= "1010";
99
100         wait for 50 ns;
101            en <='0';
102            din <= "0110";
103
104         wait for clk_period*10;
105
106         -- insert stimulus here
107
108         wait;
109    end process;
110
111 END;
```

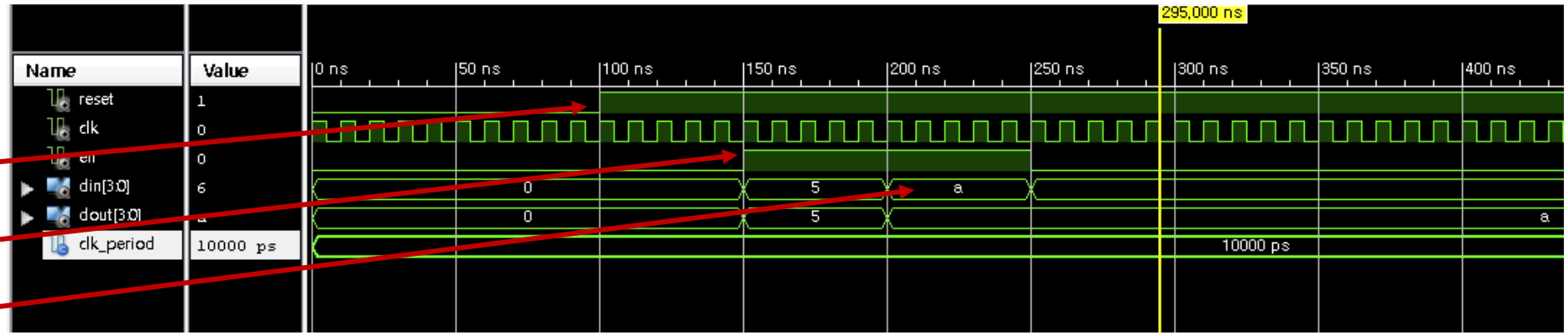
Clock generation

Input timing control

Input

Testbenches - Simulation

```
62    -- Clock period definitions
63    constant clk_period : time := 10 ns;
64
75
76    -- Clock process definitions
77    clk_process :process
78    begin
79       clk <= '0';
80       wait for clk_period/2;
81       clk <= '1';
82       wait for clk_period/2;
83    end process;
84
85
86    -- Stimulus process
87    stim_proc: process
88    begin
89       -- hold reset state for 100 ns.
90       wait for 100 ns;
91
92          reset <= '1';
93       wait for 50 ns;
94          din <= "0101";
95          en <= '1';
96
97       wait for 50 ns;
98          din <= "1010";
99
100      wait for 50 ns;
101         en <='0';
102         din <= "0110";
103
104      wait for clk_period*10;
105
106      -- insert stimulus here
107
108      wait;
109   end process;
110
111 END;
```



Helpful websites

FPGA designs with VHDL

- https://vhdlguide.readthedocs.io/en/latest/index.html

VHDL Tutorial, University of Pennsylvania (Jan Van der Spiegel)

- https://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html