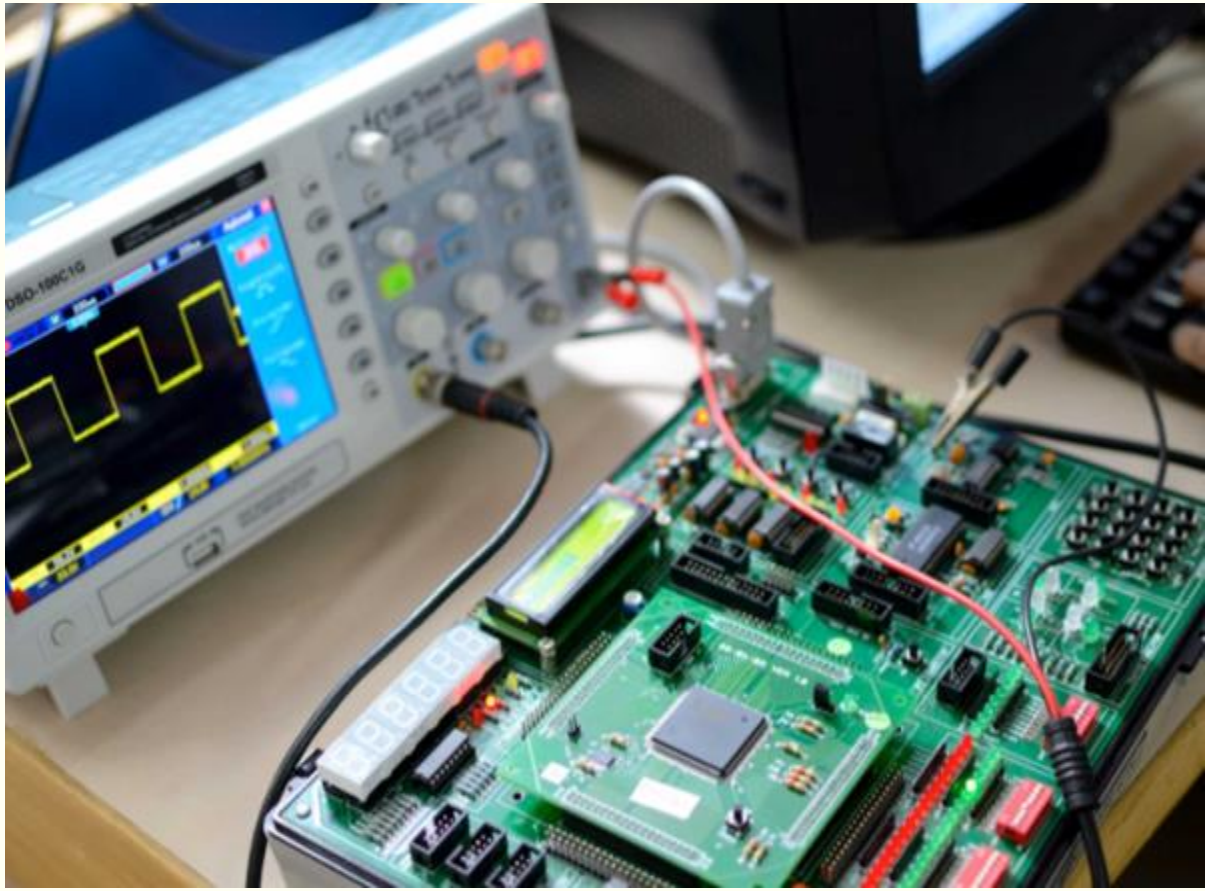
INTRODUCTION TO VHDL



K. Siozios

Section of Electronics and Electronic Computers
Department of Physics
Aristotle University of Thessaloniki (AUTH), Greece

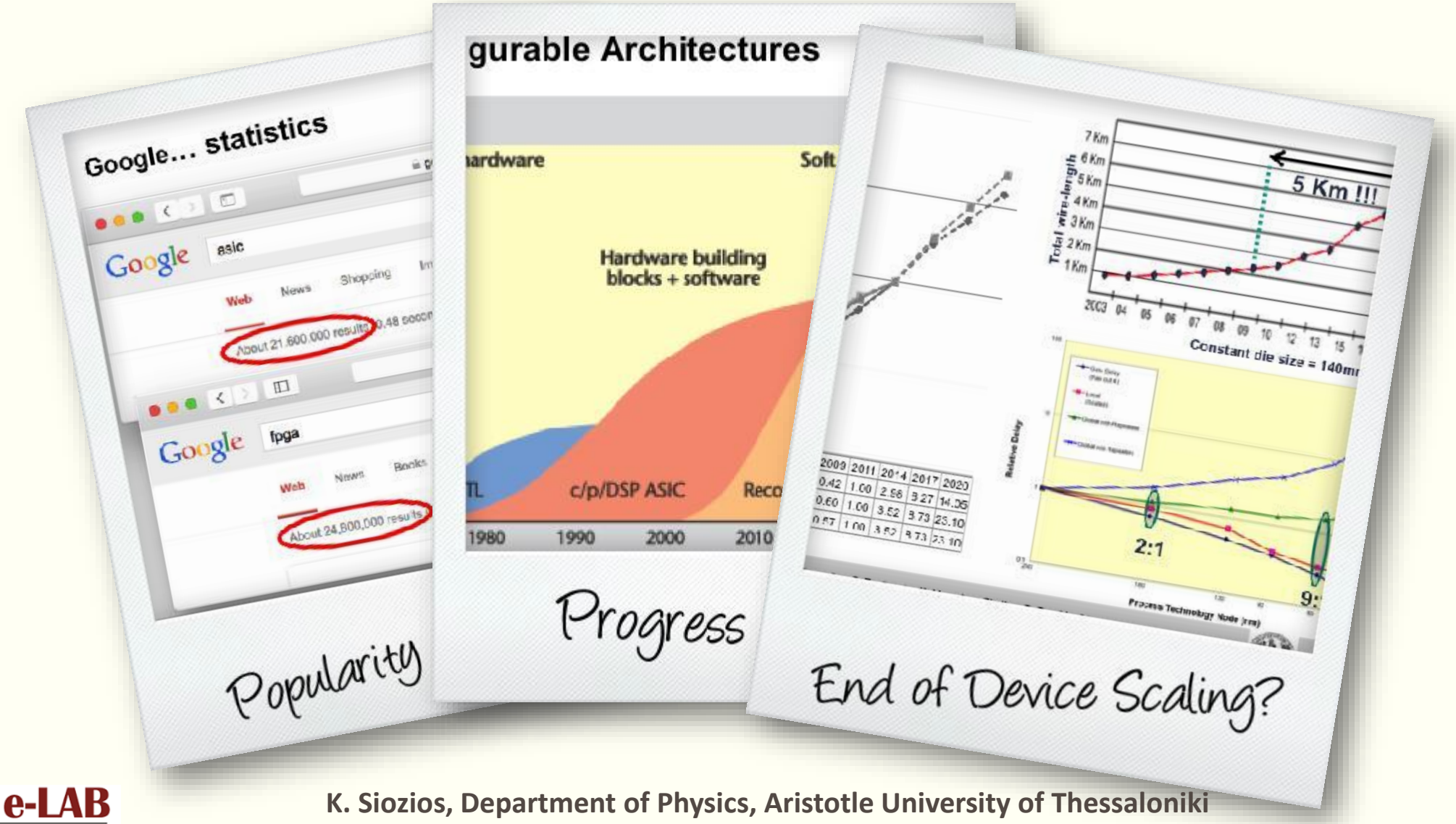
"It is not the strongest of the species that survive,
nor the most intelligent,
but the one most responsive to change"

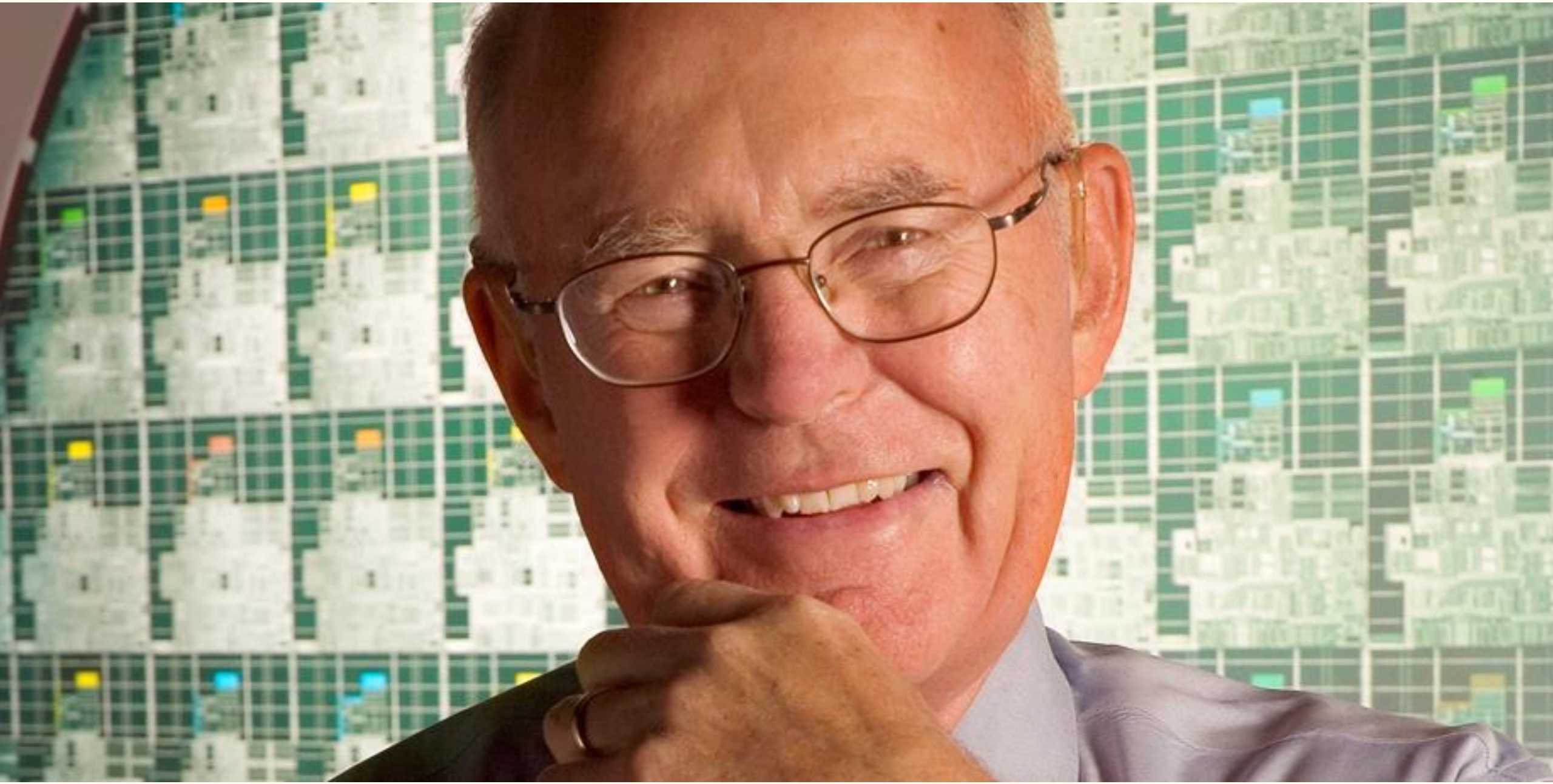
Charles Darwin

It is not the strongest (ASIC) of the species that survive,
nor the most intelligent (CPU),
but the one most responsive to change (FPGA)

Charles Darwin

Reconfigurable Architectures Today





K. Siozios, Department of Physics, Aristotle University of Thessaloniki

Moore's Law

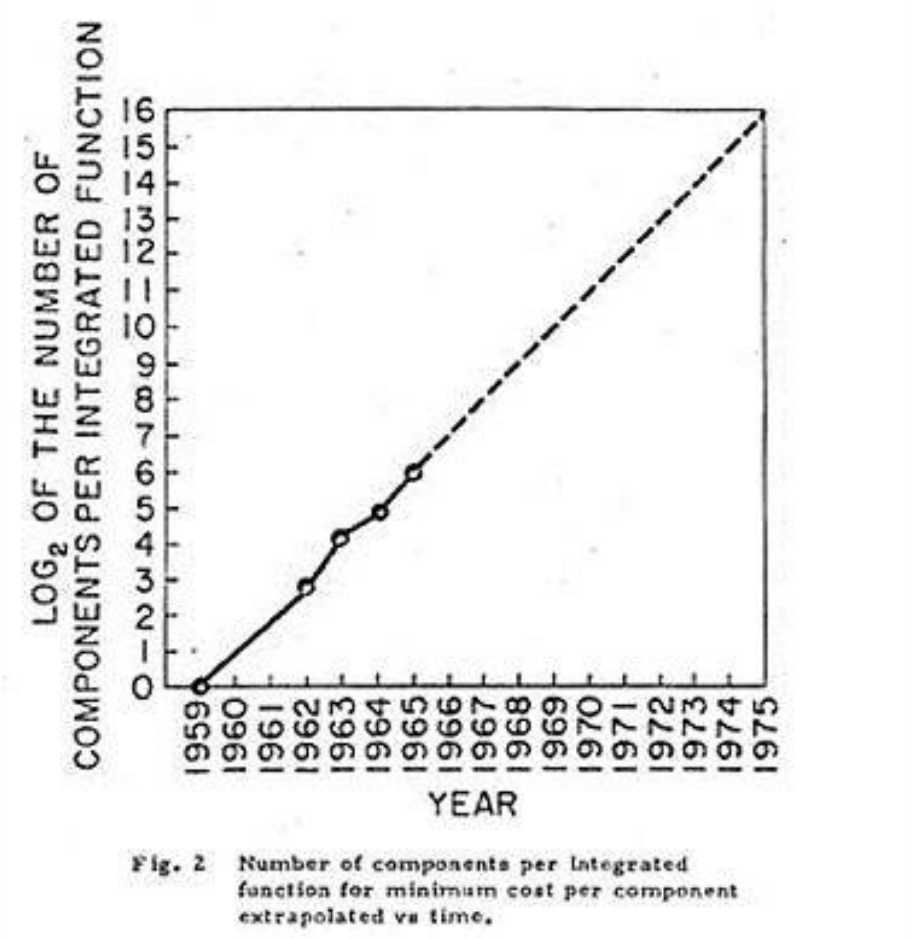
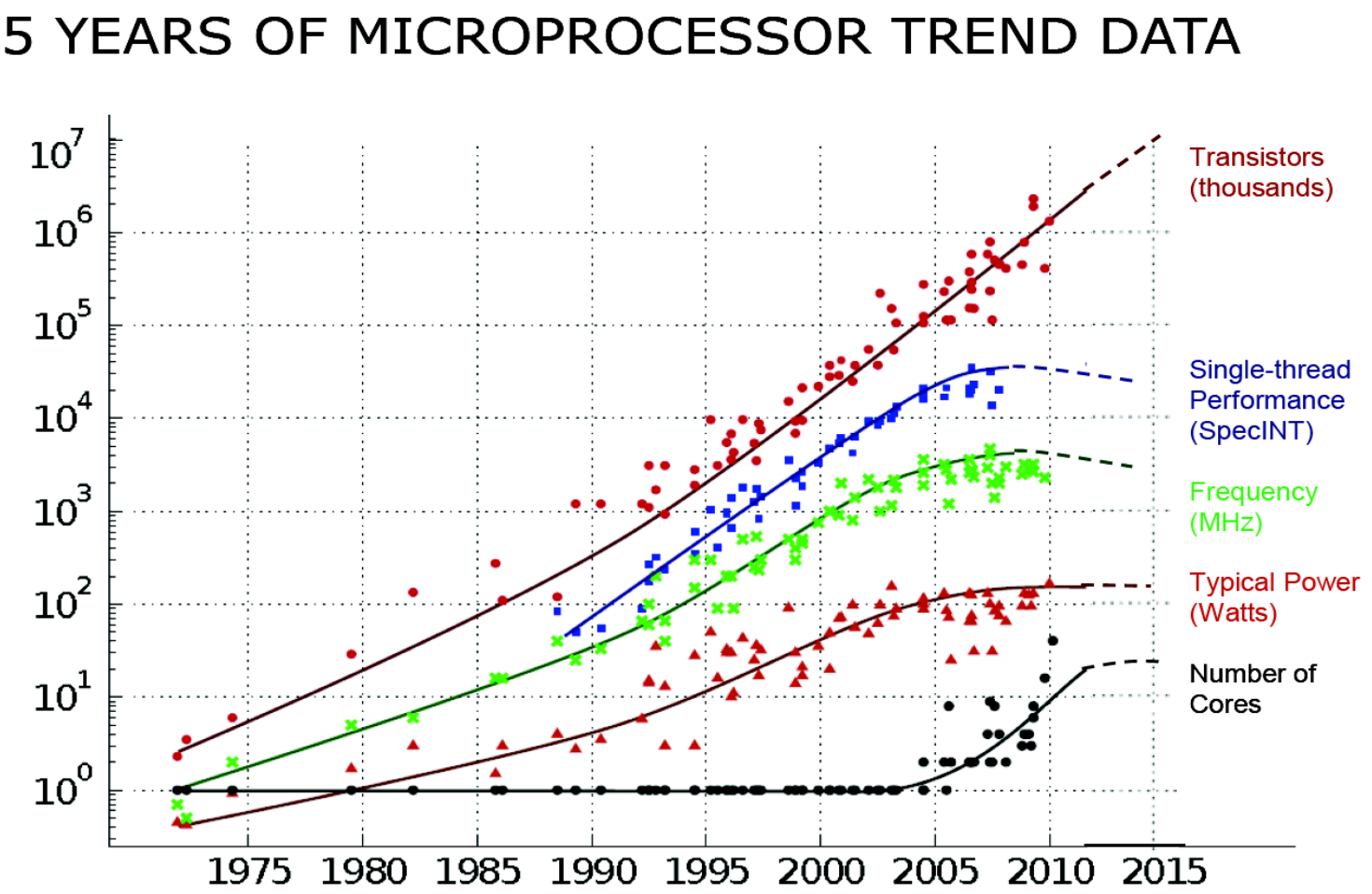


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

"*The number of transistors incorporated in a chip will approximately double every 24 months.*"
—Gordon Moore, Intel Co-Founder

Moore's Law in Microprocessors



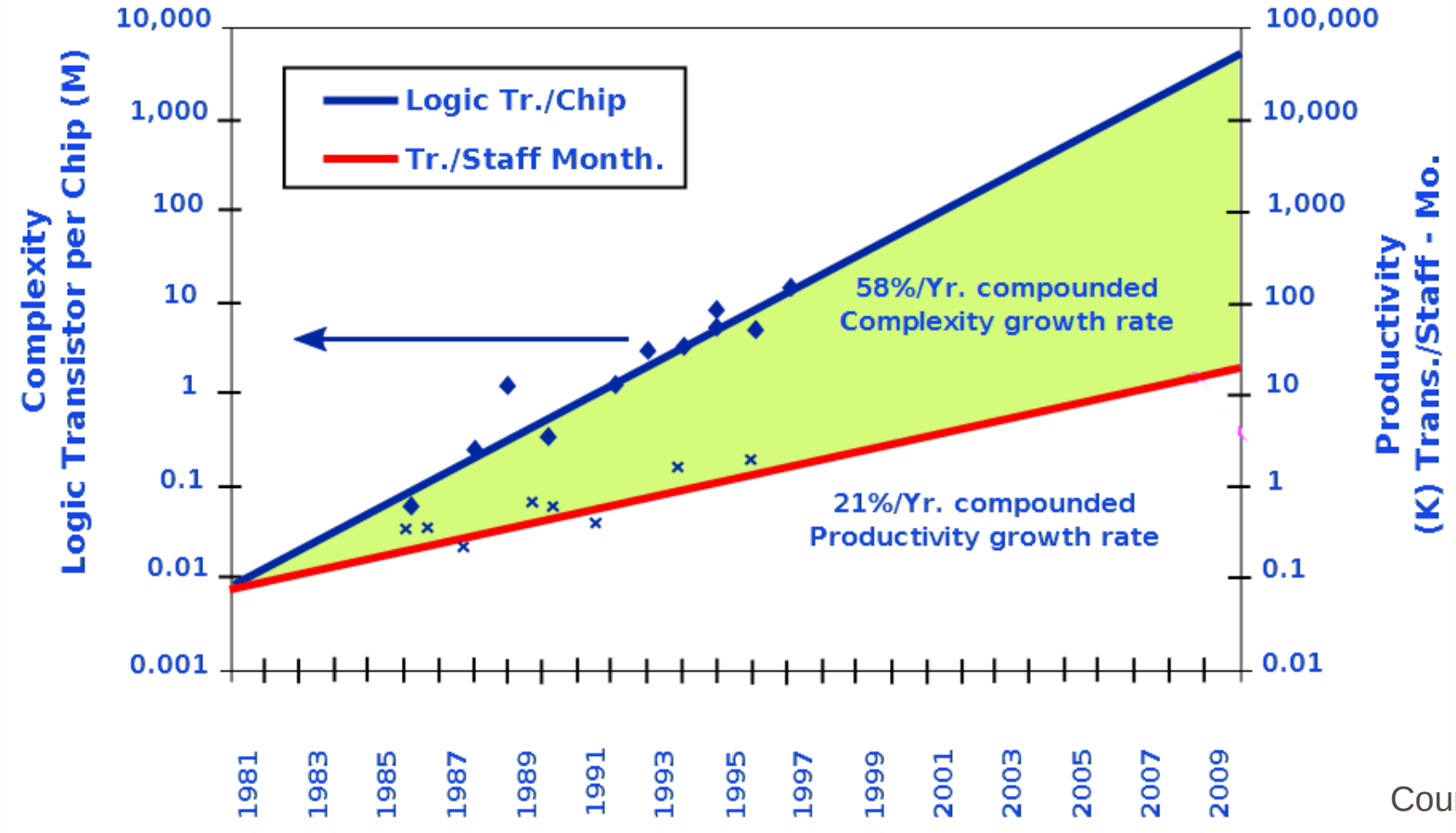
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Moore's Law: CPUs and FPGAs

CPU	Transistor count	Process
Pentium 4	42,000,000	180 nm
Atom	47,000,000	45 nm
AMD K8	105,900,000	130 nm
Itanium 2	220,000,000	130 nm
Cell	241,000,000	90 nm
Core 2 Duo	291,000,000	65 nm
Core i7 (Quad)	731,000,000	45 nm
Six-Core Xeon 7400	1,900,000,000	45 nm
POWER6	789,000,000	65 nm
16-Core SPARC T3	1,000,000,000	40 nm
8-core POWER7	1,200,000,000	45 nm
Quad-core z196	1,400,000,000	45 nm
Dual-Core Itanium 2	1,700,000,000	90 nm

FPGA	Transistor count	Process
Virtex	~70,000,000	
Virtex-E	~200,000,000	
Virtex-II	~350,000,000	130 nm
Virtex-II PRO	~430,000,000	
Virtex-4	1,000,000,000	90 nm
Virtex-5	1,100,000,000	65 nm
Stratix IV	2,500,000,000	40 nm
Stratix V	3,800,000,000	28 nm
Virtex-7	6,800,000,000	28 nm

Productivity Trends

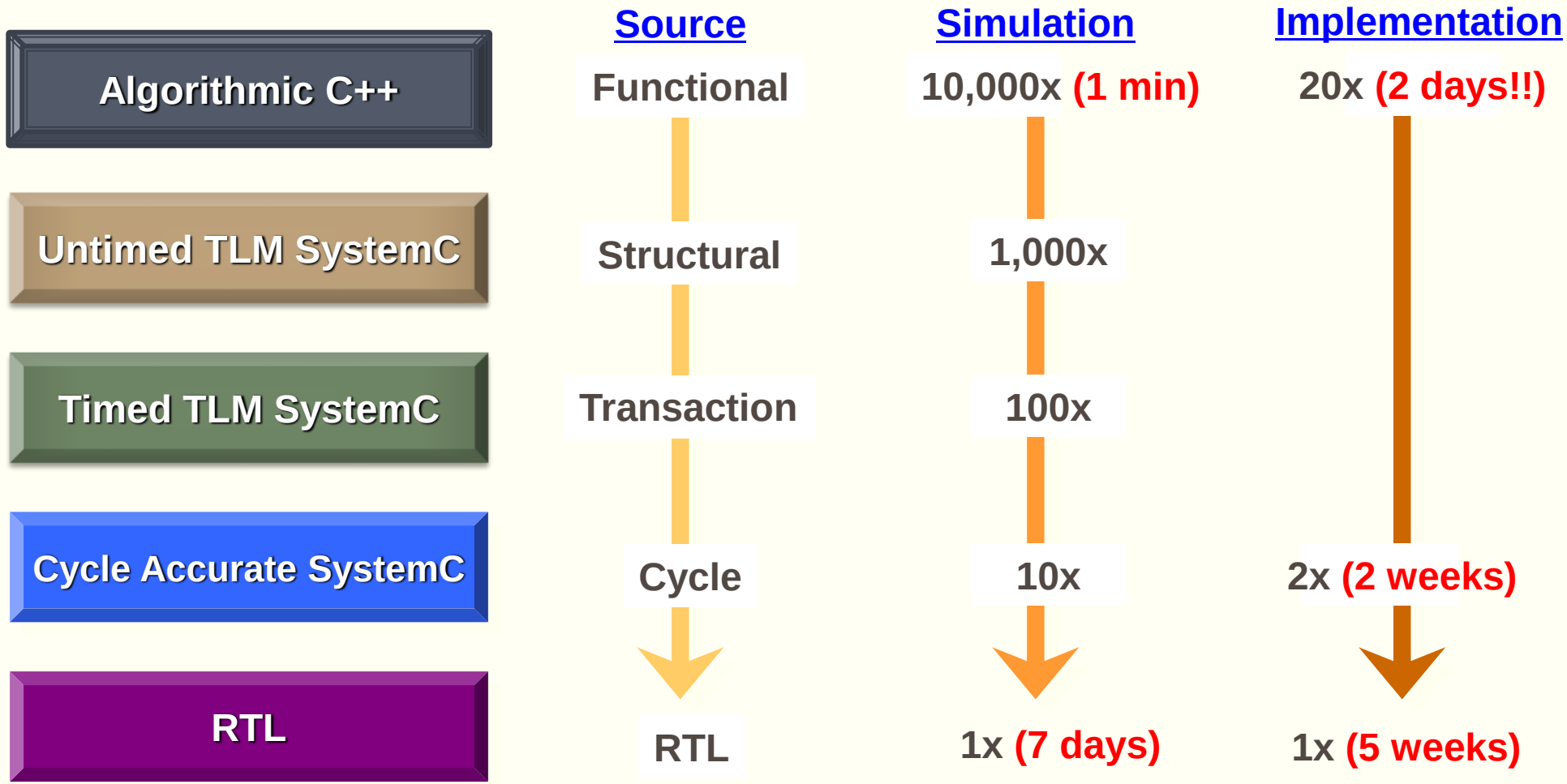


Courtesy, ITRS Roadmap

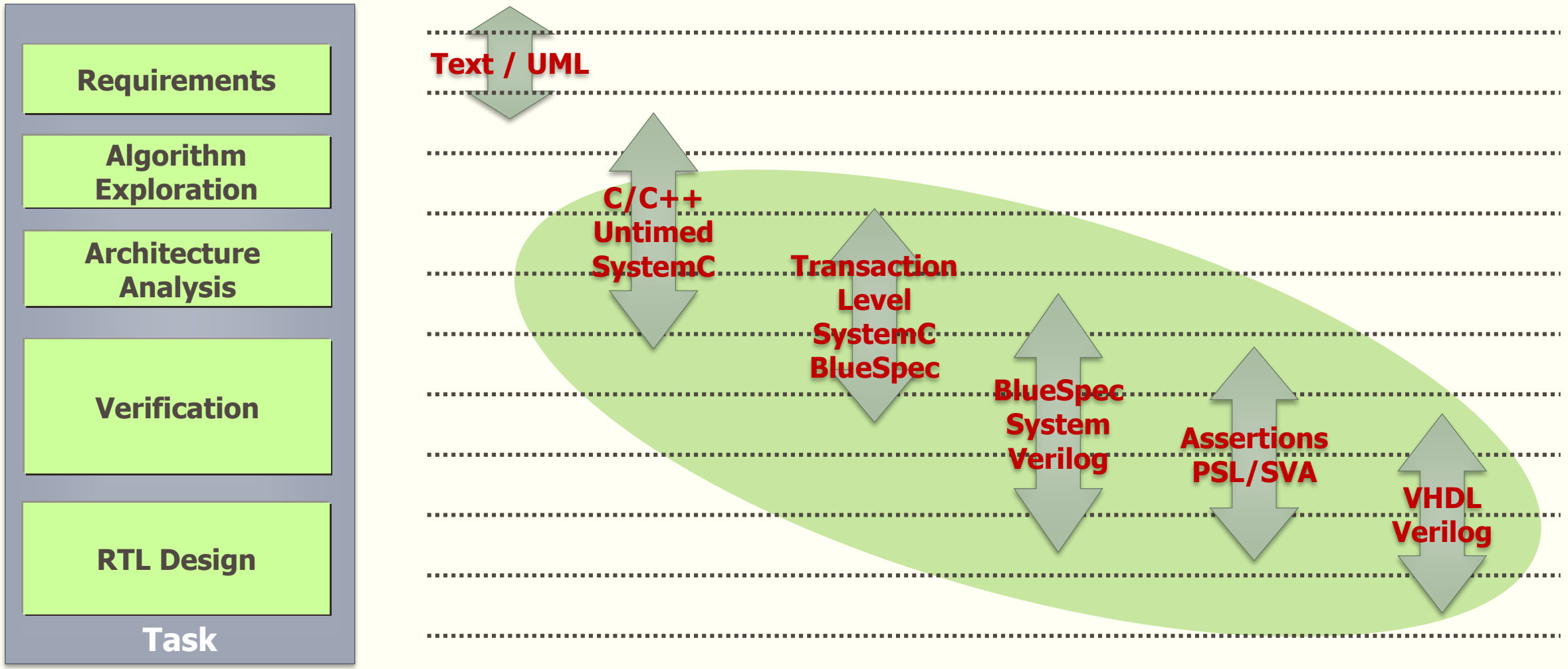
Source: Sematech

Complexity outpaces design productivity

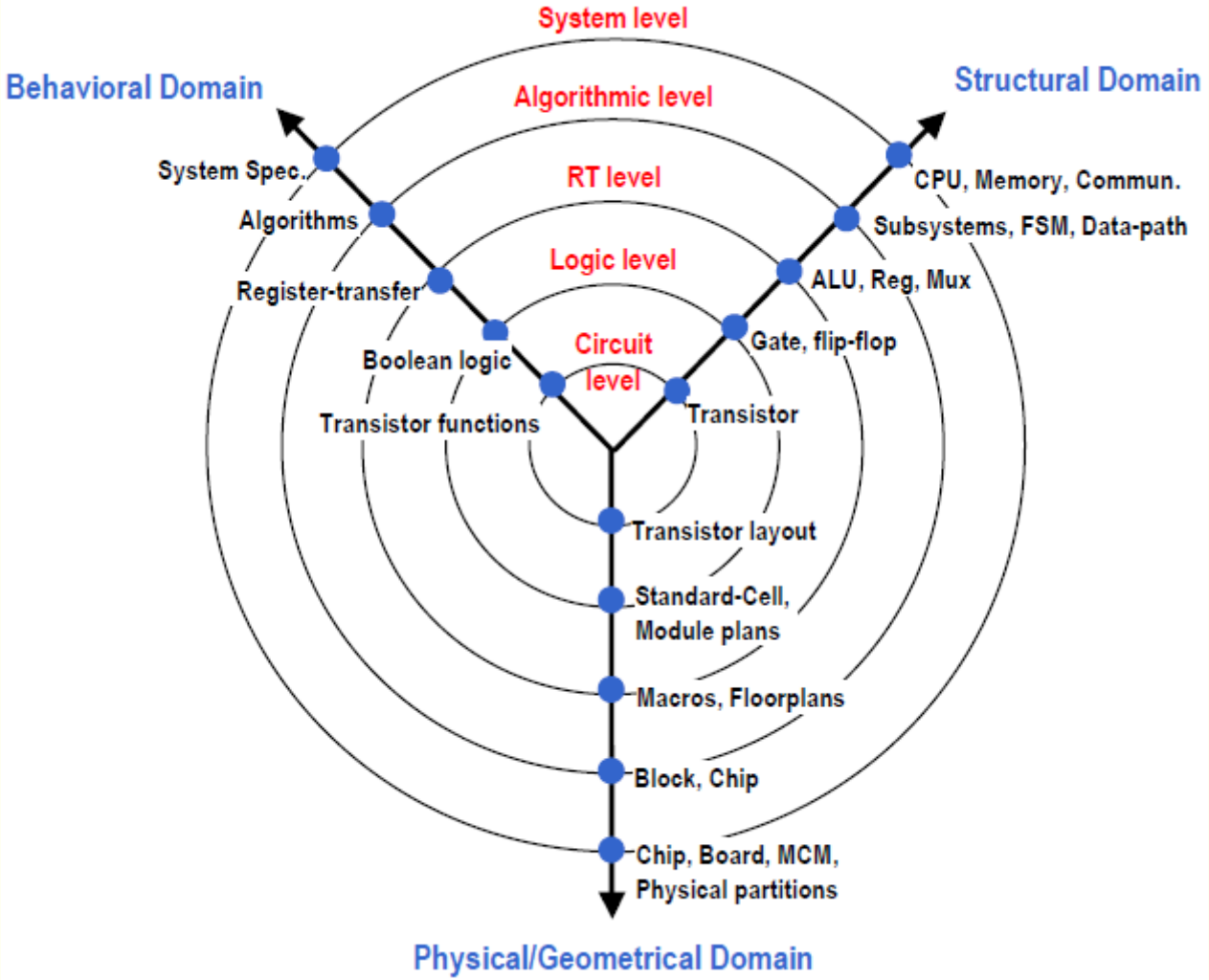
Abstraction Drives Design Productivity



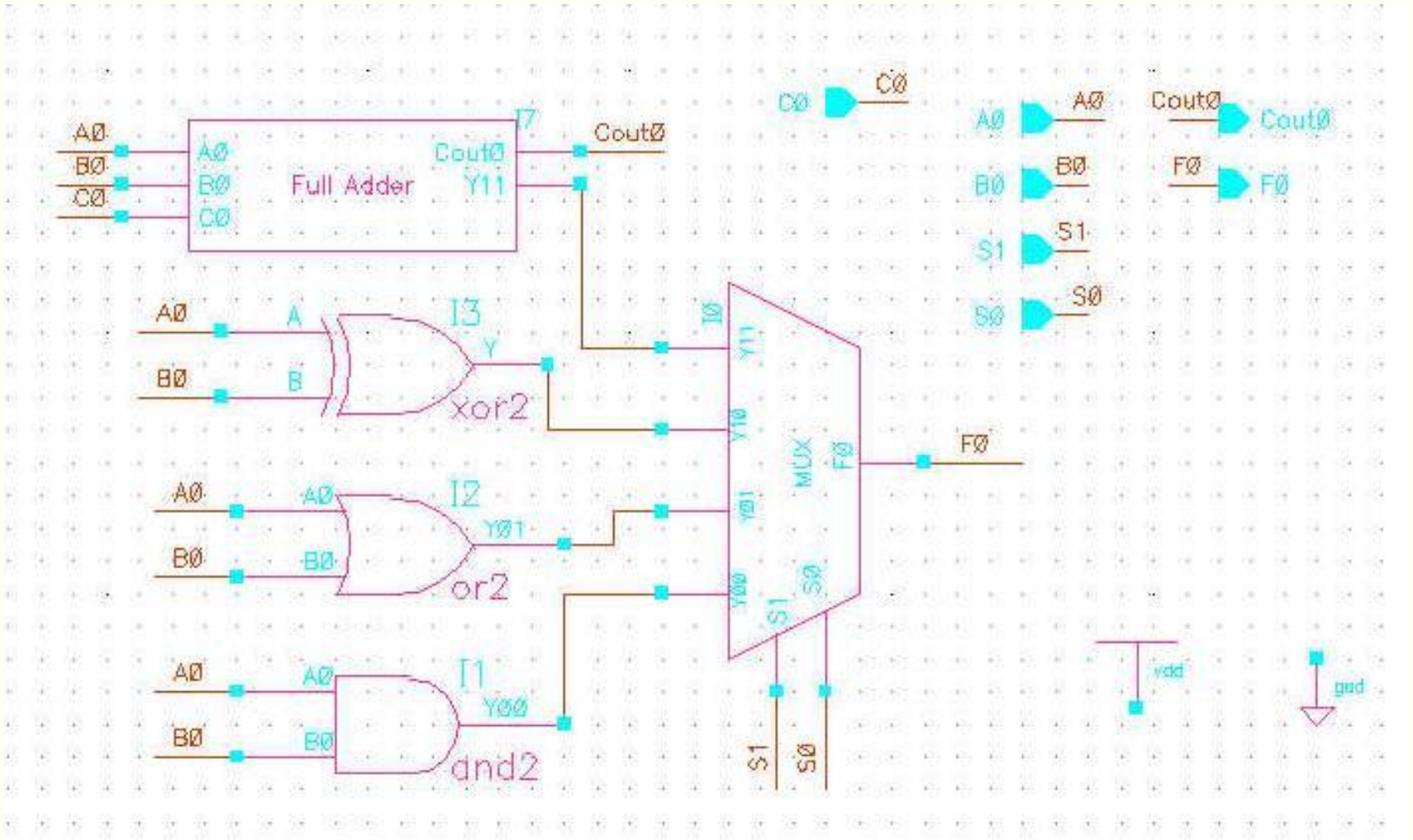
Design Languages & Tasks



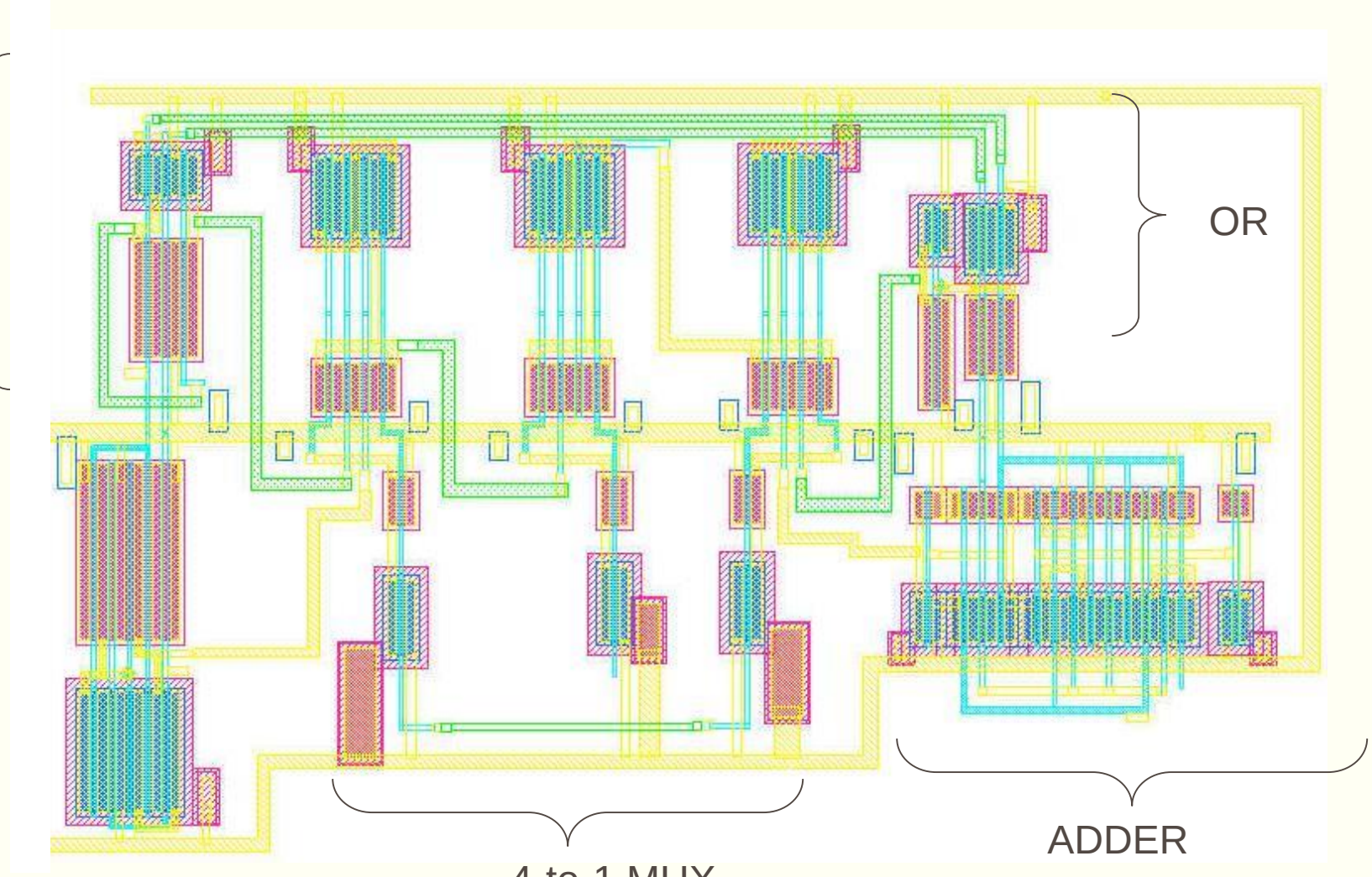
Gajski’s & Khun’s Y-chart



1-bit ALU (schematic)

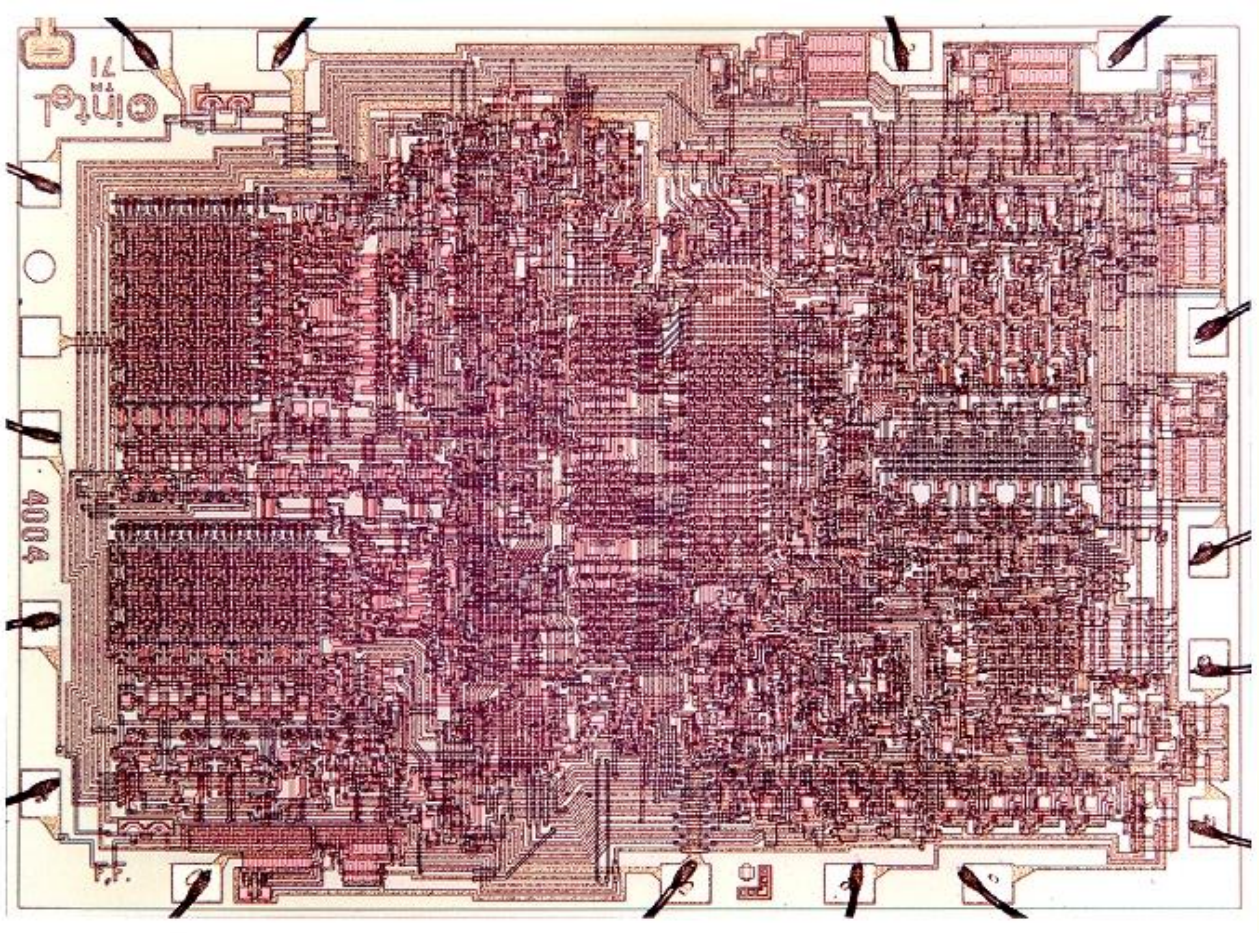


1-bit ALU (layout)

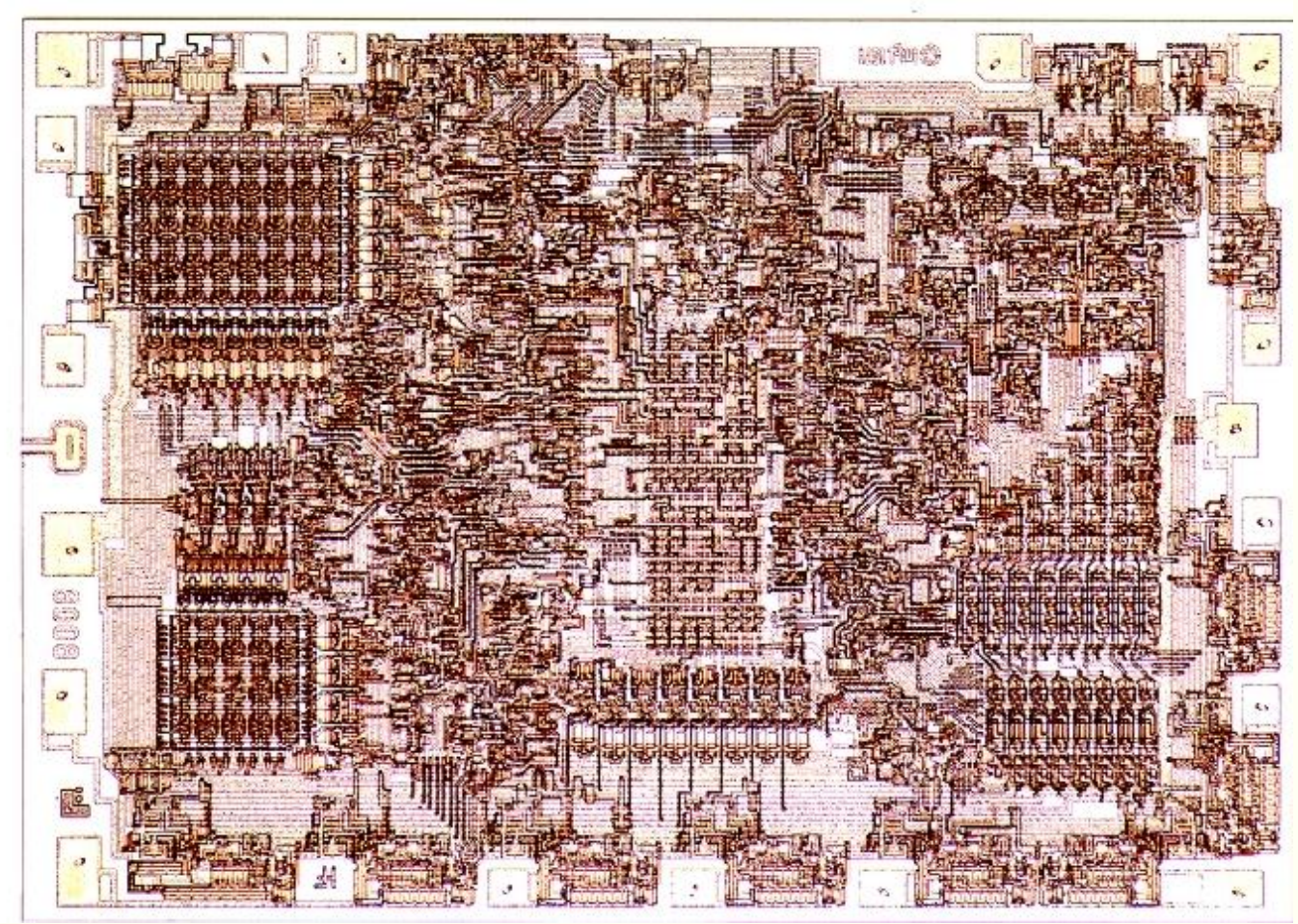


4004 μP (1971)

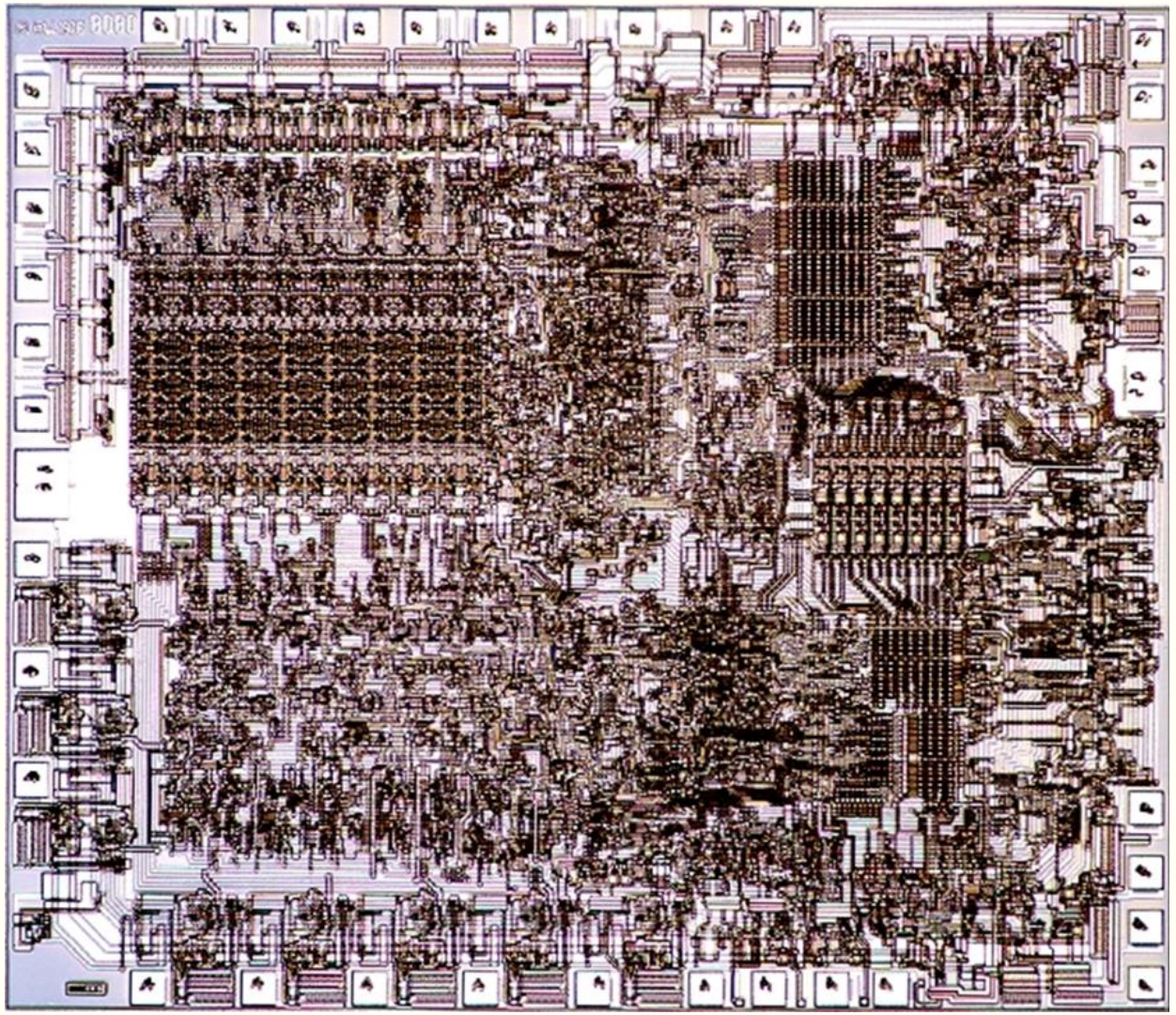
- 4-bit CPU
- First complete CPU on chip
- First commercially available CPU
- 2,300 transistors
- MIPS: 0.092
- Clock speed: 1MHz



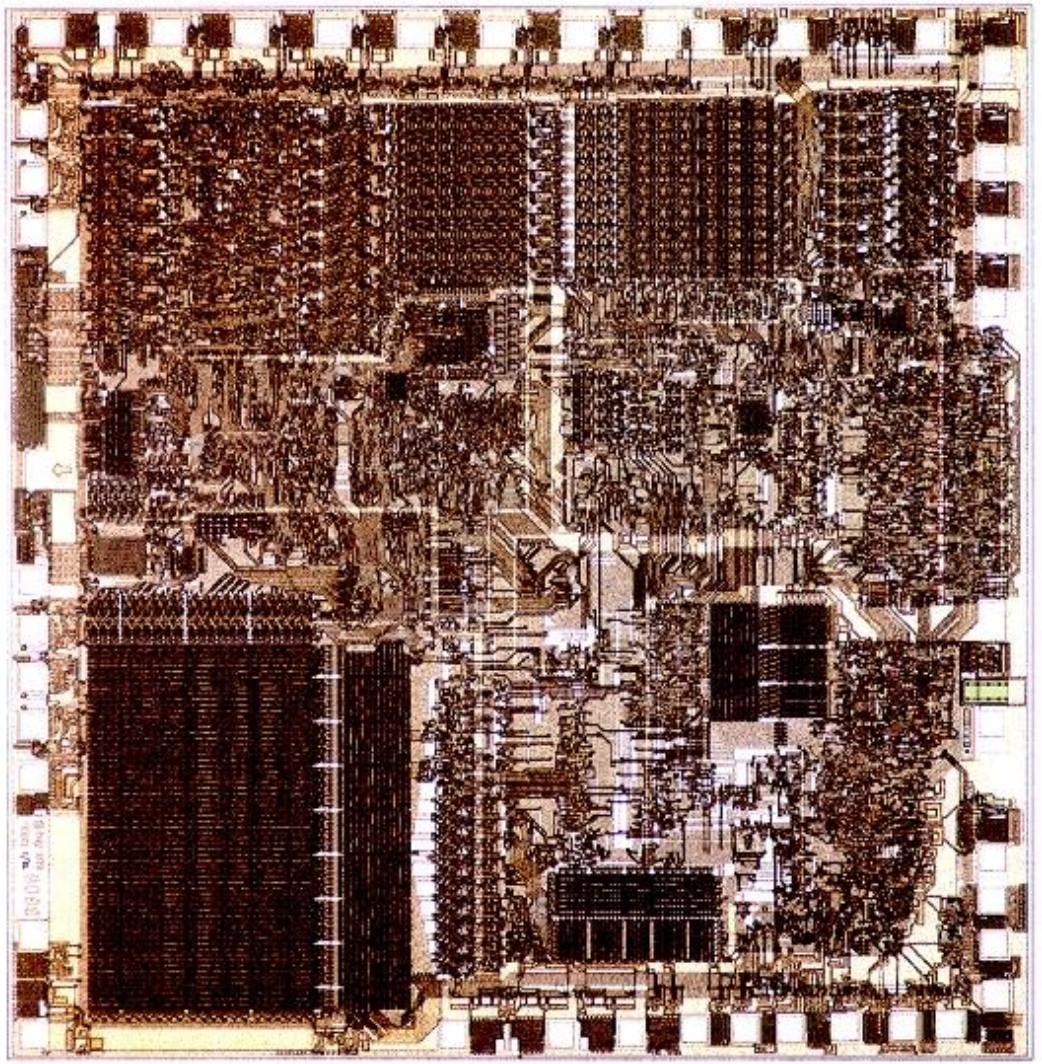
8008 μP (1972)



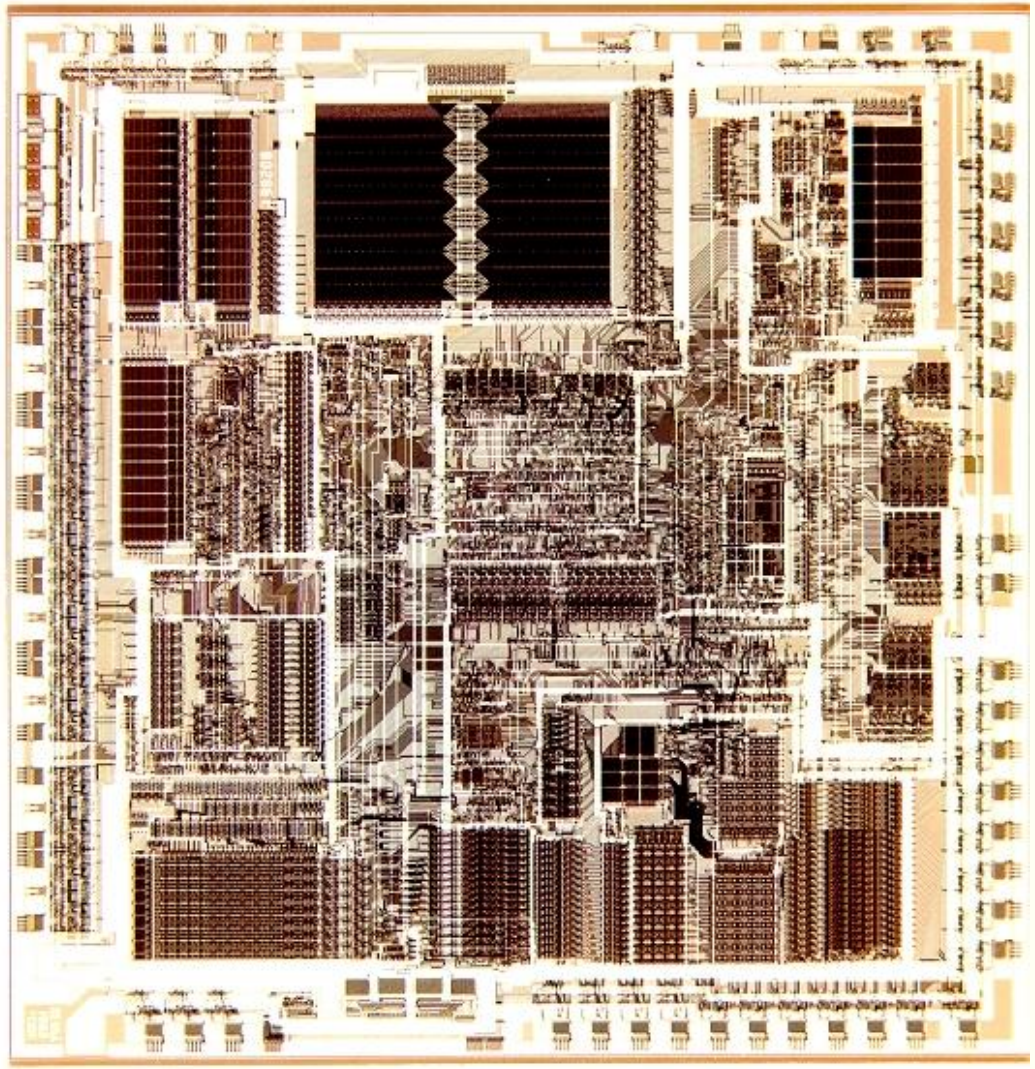
8080 μP (1974)



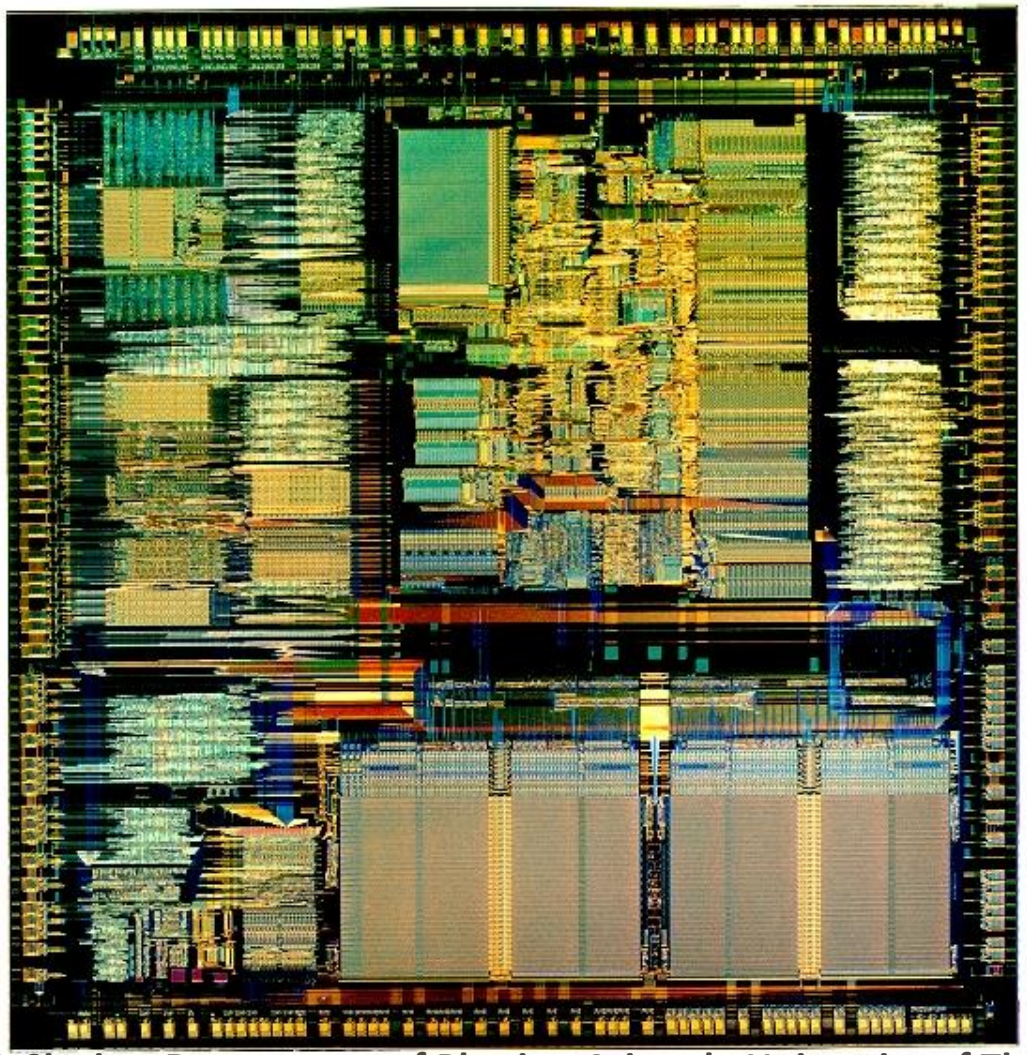
8088 μP (1978)



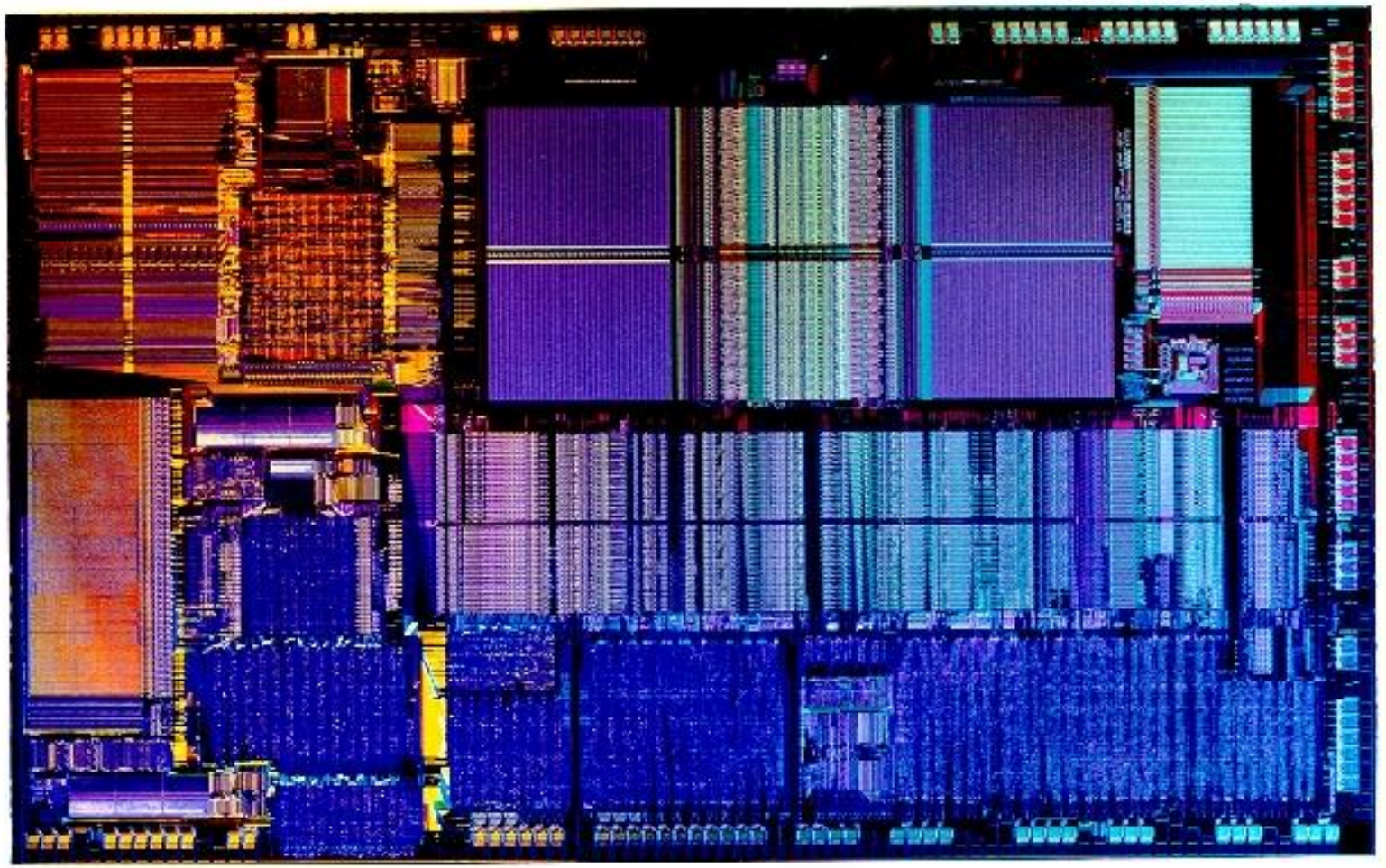
8286 μP (1982)



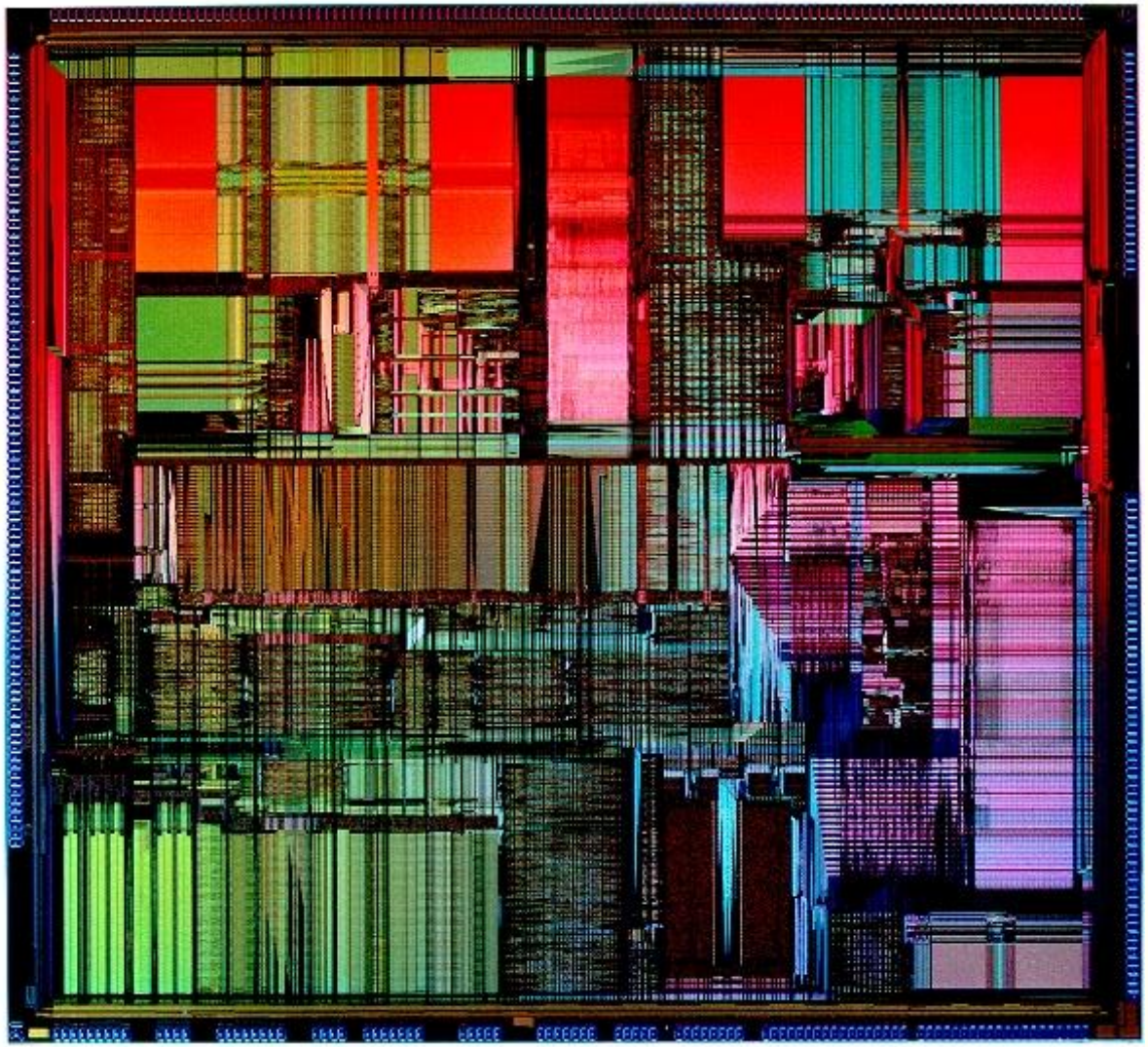
8386 μP (1985)



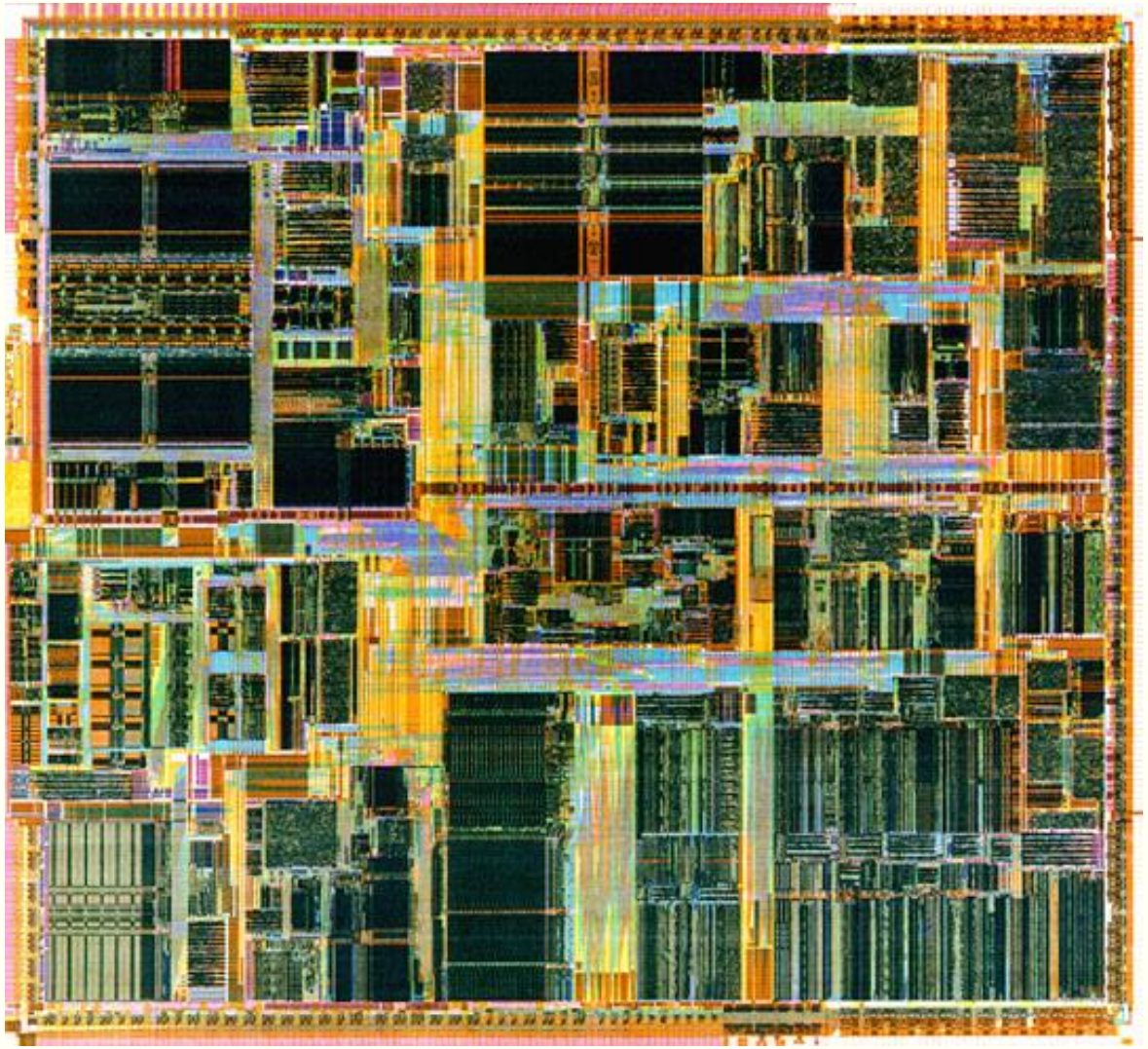
8486 μP (1989)



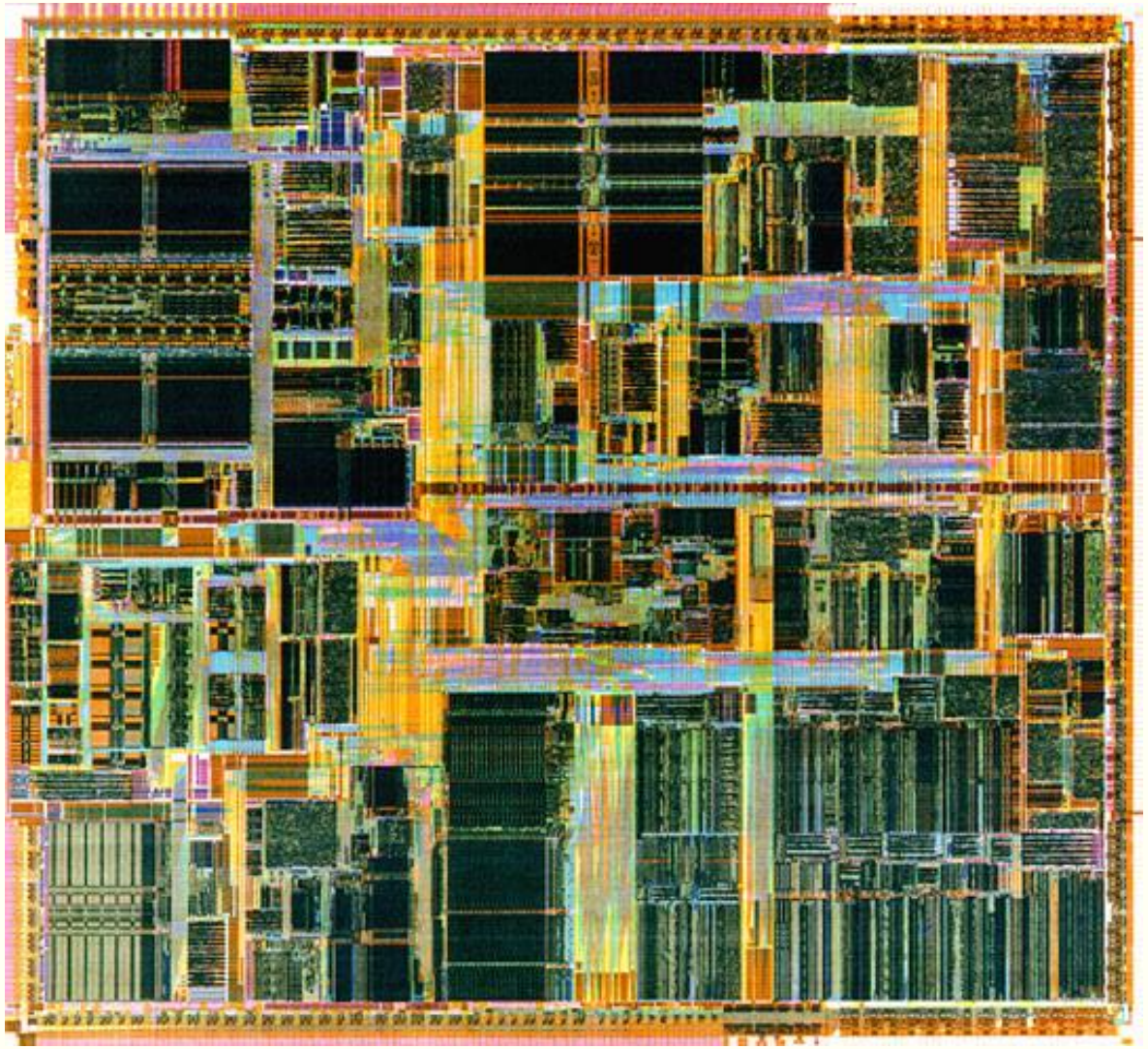
Pentium μP (1993)



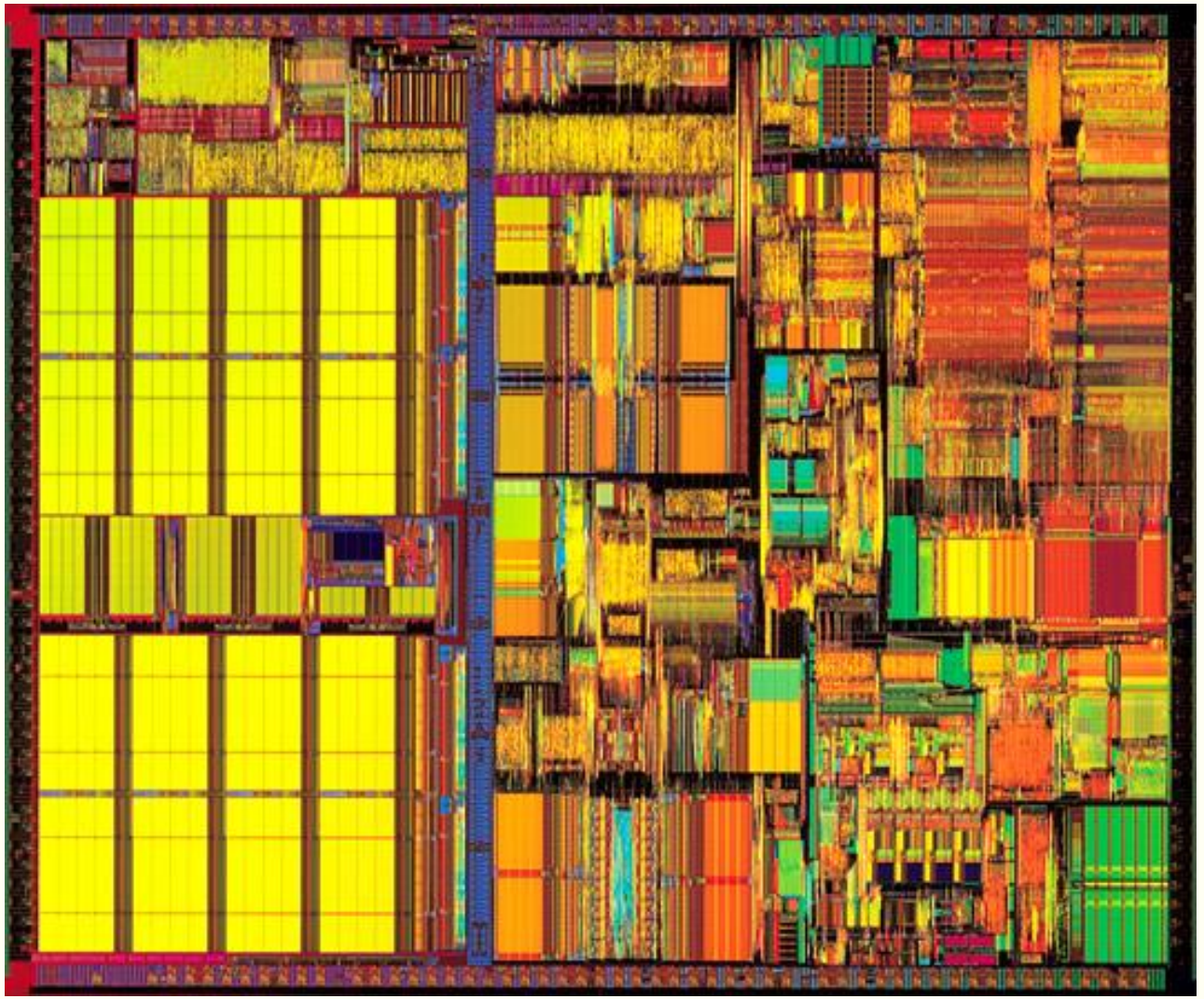
Pentium II μP (1997)



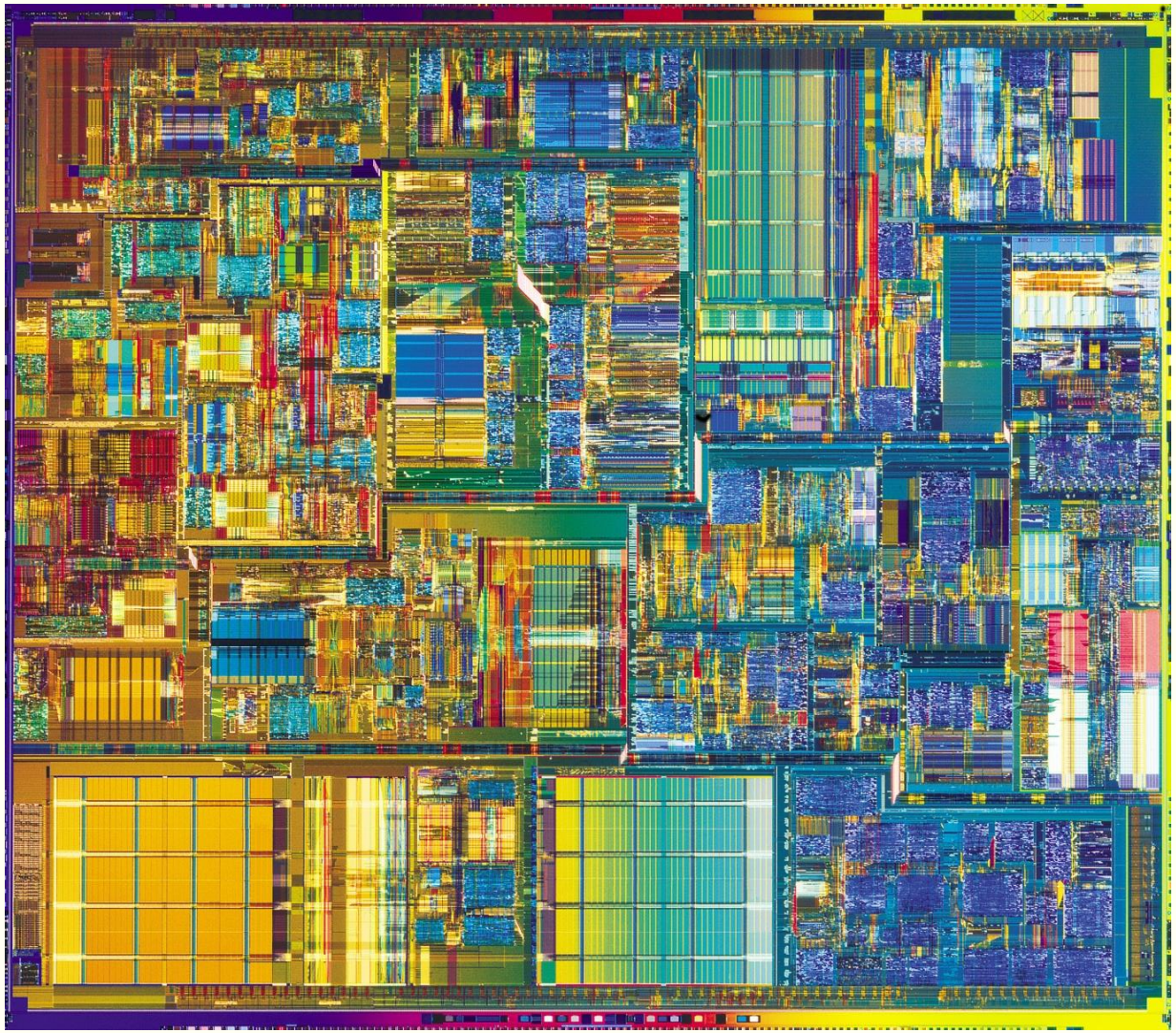
Pentium II XEON μP (1998)

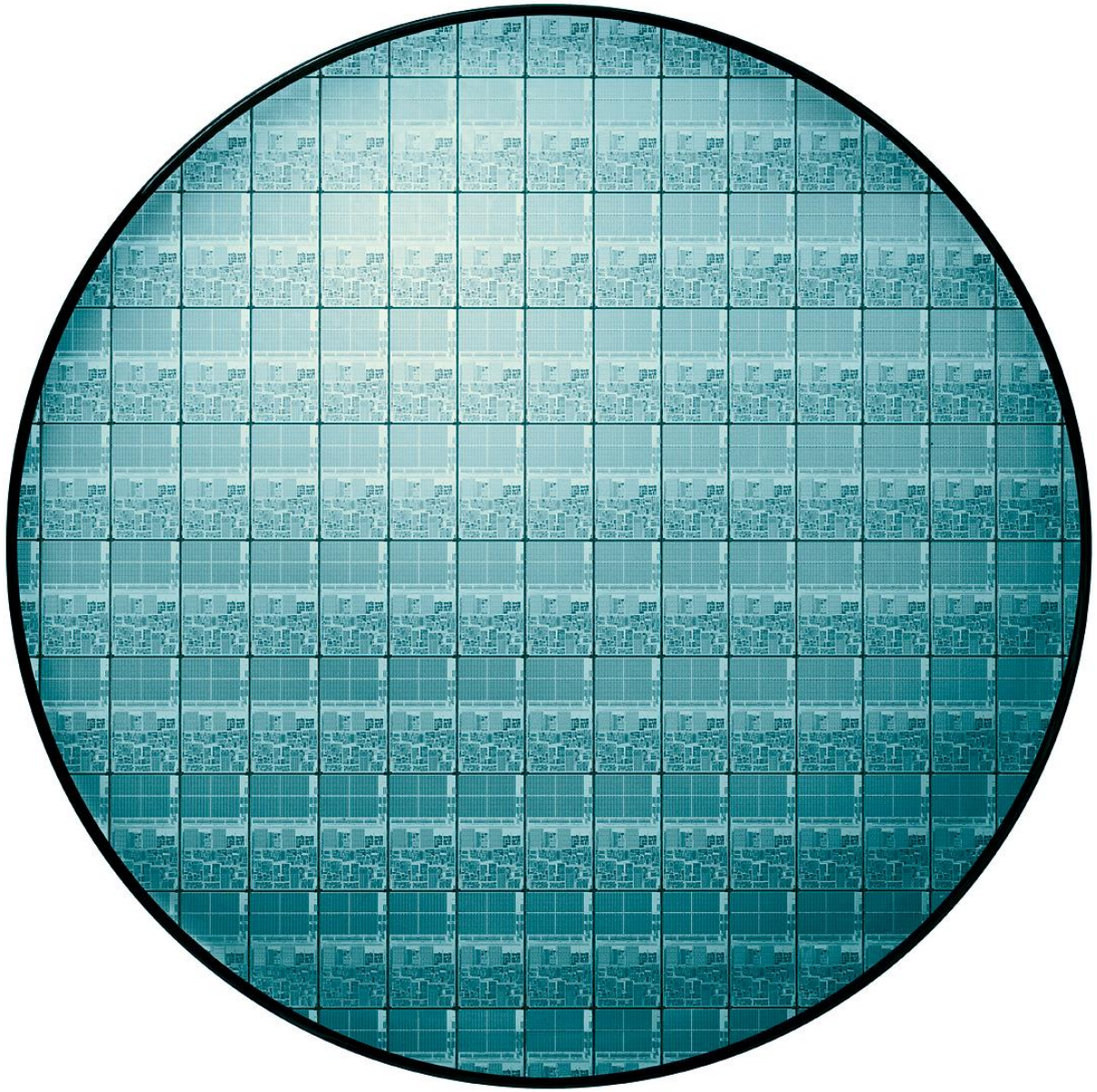


Pentium III μP (1999)

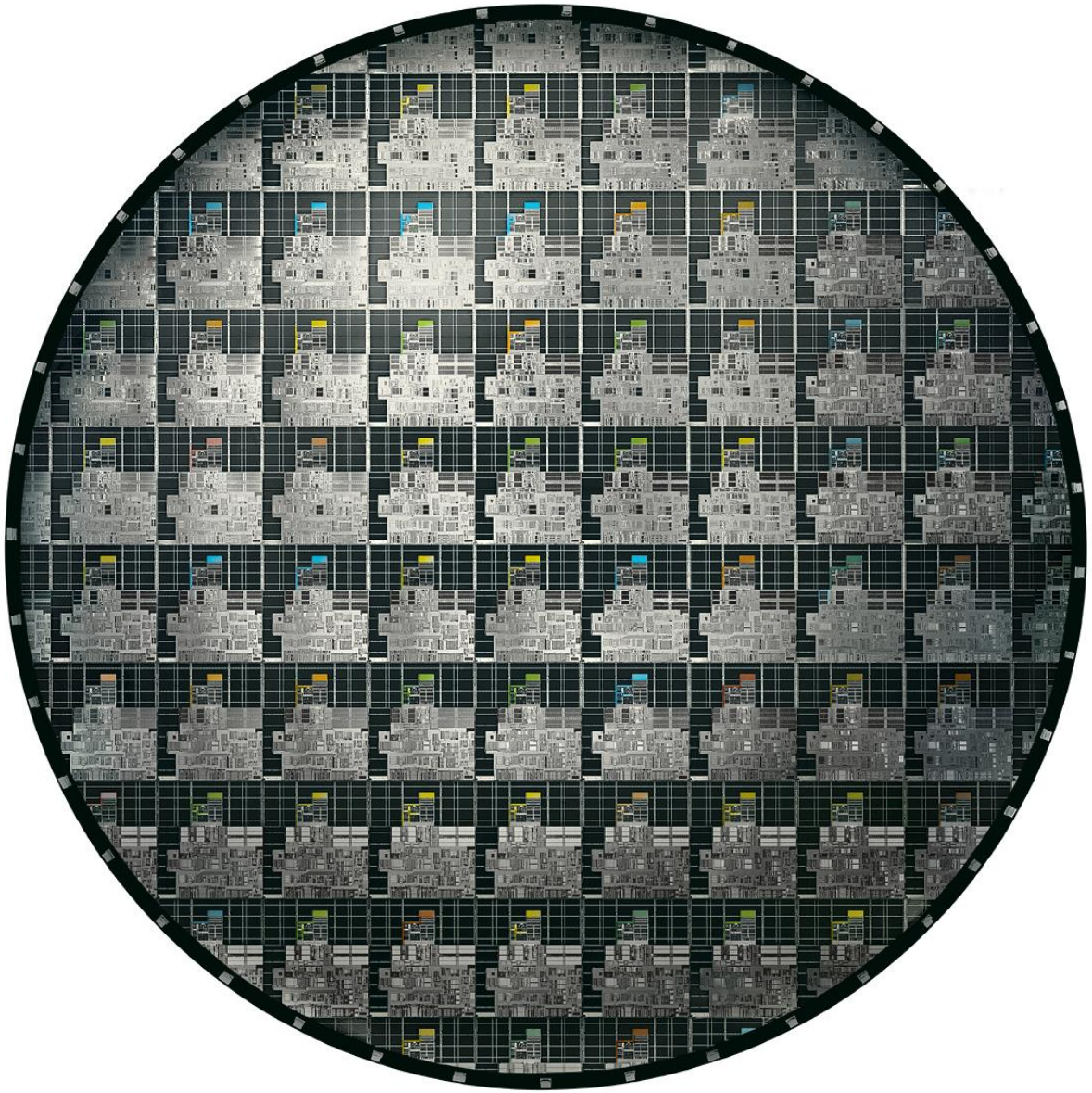


Pentium 4 μP (2000)





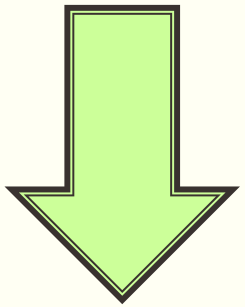
Wafer of Intel® Xeon™ processors



Wafer of Itanium® processors

VHDL: Very Hard Difficult Language

VHDL: Very Hard Difficult Language

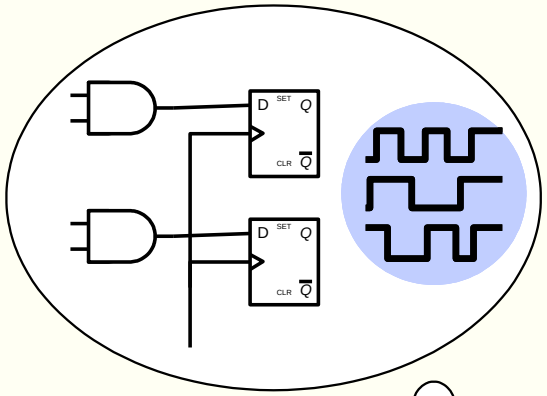


Very High-Speed Integrated Circuit Hardware Description Language

VHDL is a language for describing digital hardware used by industry worldwide

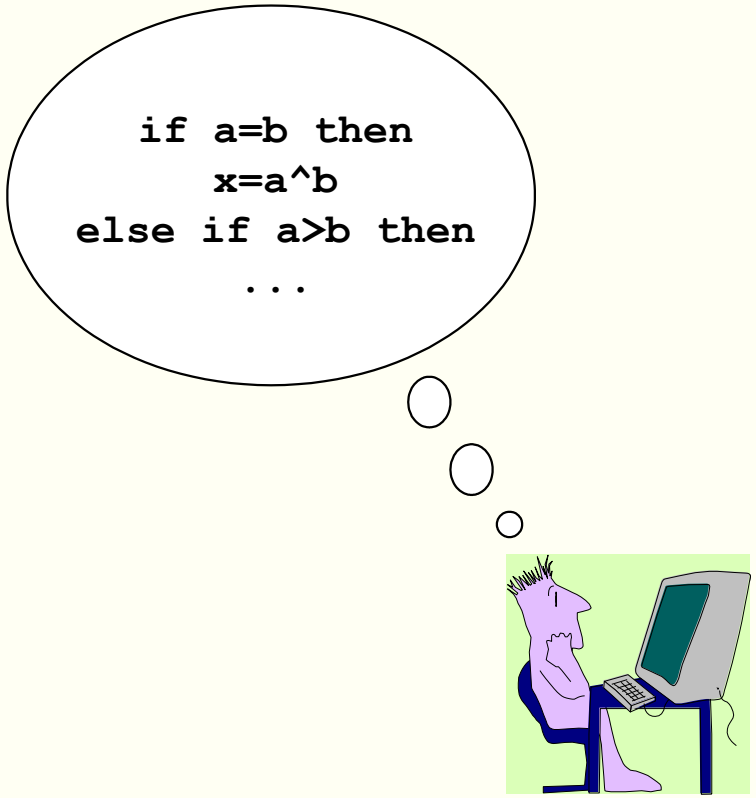
VHDL

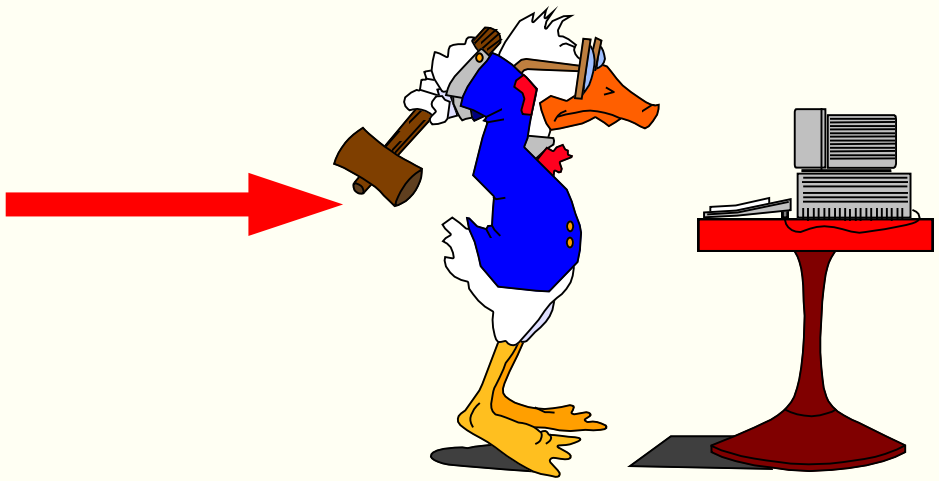
How to write HDL code:



VHDL

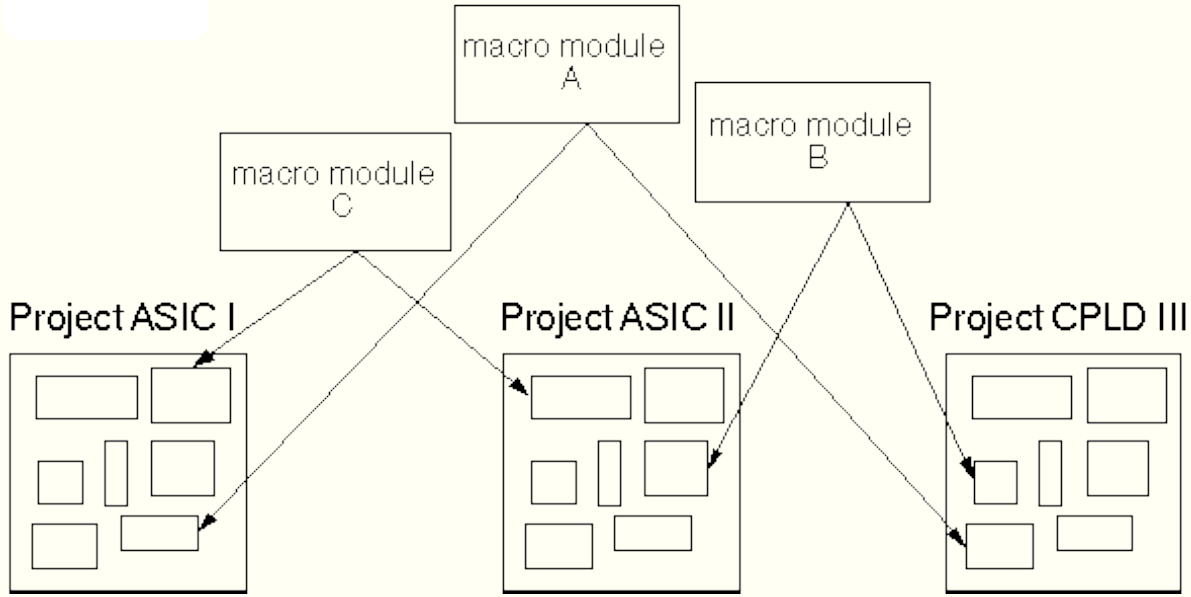
How **NOT** to write HDL code:





Advantages of VHDL

- Technology/vendor independent
- Portable
- Reusable



Genesis of VHDL

- **State of art circa 1980**
  - Multiple design entry methods and hardware description languages in use
  - No or limited portability of designs between CAD tools from different vendors
  - Objective: shortening the time from a design concept to implementation from 18 months to 6 months

History of VHDL

- June 1981: Woods Hole Workshop
- July 1983: contract awarded to develop VHDL
  - Intermetrics
  - IBM
  - Texas Instruments
- August 1985: VHDL Version 7.2 released
- December 1987: VHDL became IEEE Standard 1076-1987 and in 1988 an ANSI standard

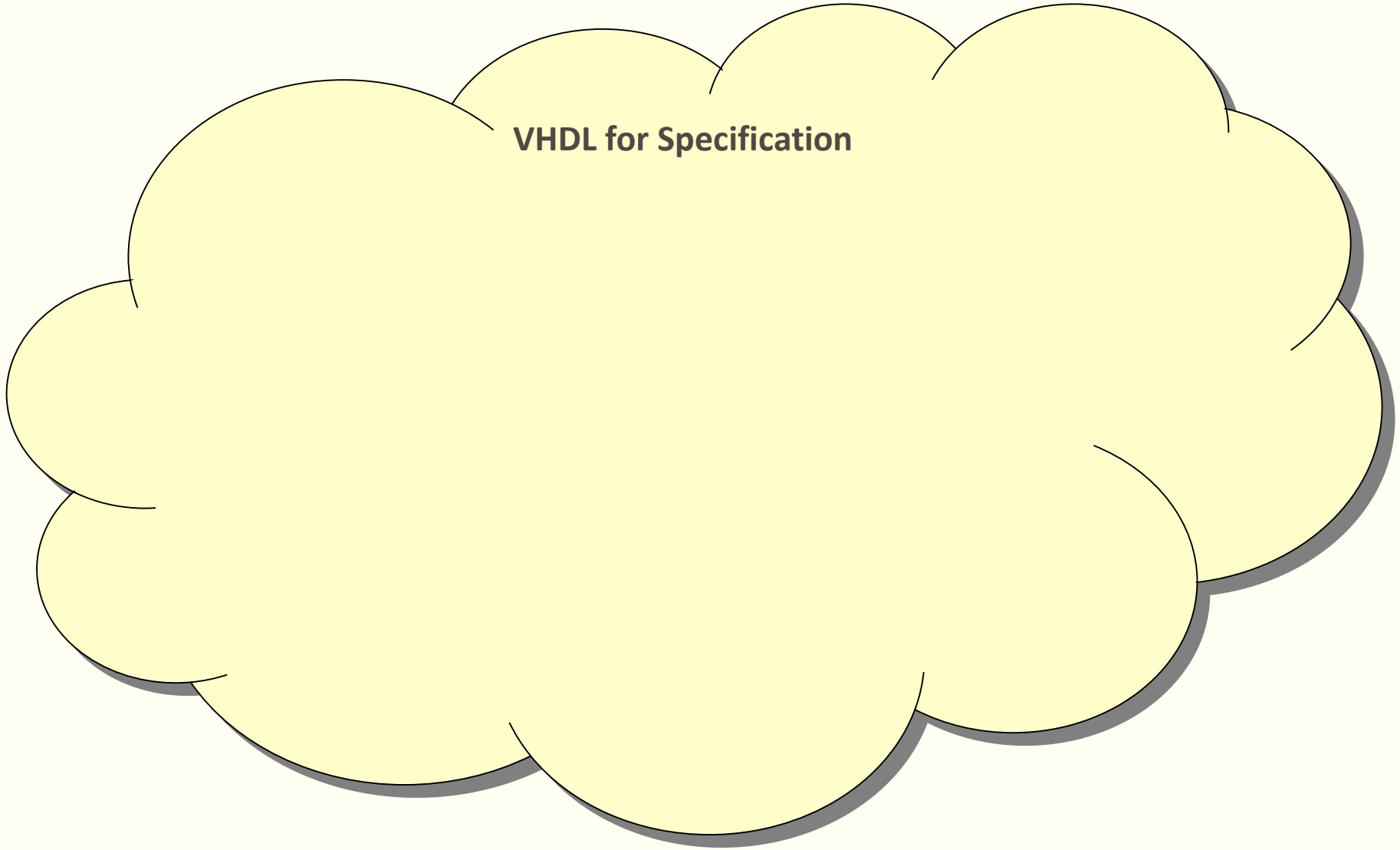
VHDL Standards

- **P1076** Standard VHDL Language Reference Manual (VASG) VHDL-87, VHDL-93, VHDL-01, VHDL-08
- **P1076.1** Standard VHDL Analog and Mixed-Signal Extensions (VHDL-AMS)
- **P1076.1.1** Standard VHDL Analog and Mixed-Signal Extensions - Packages for Multiple Energy Domain Support (StdPkgs) - this group is now part of 1076.1
- **P1076.2** IEEE Standard VHDL Mathematical Packages (math)
- **P1076.3** Standard VHDL Synthesis Packages (vhdlsynth)
- **P1076.4** Standard VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification (VITAL) - This group is now part of 1076.
- **P1164** Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164) (vhdl-std-logic)

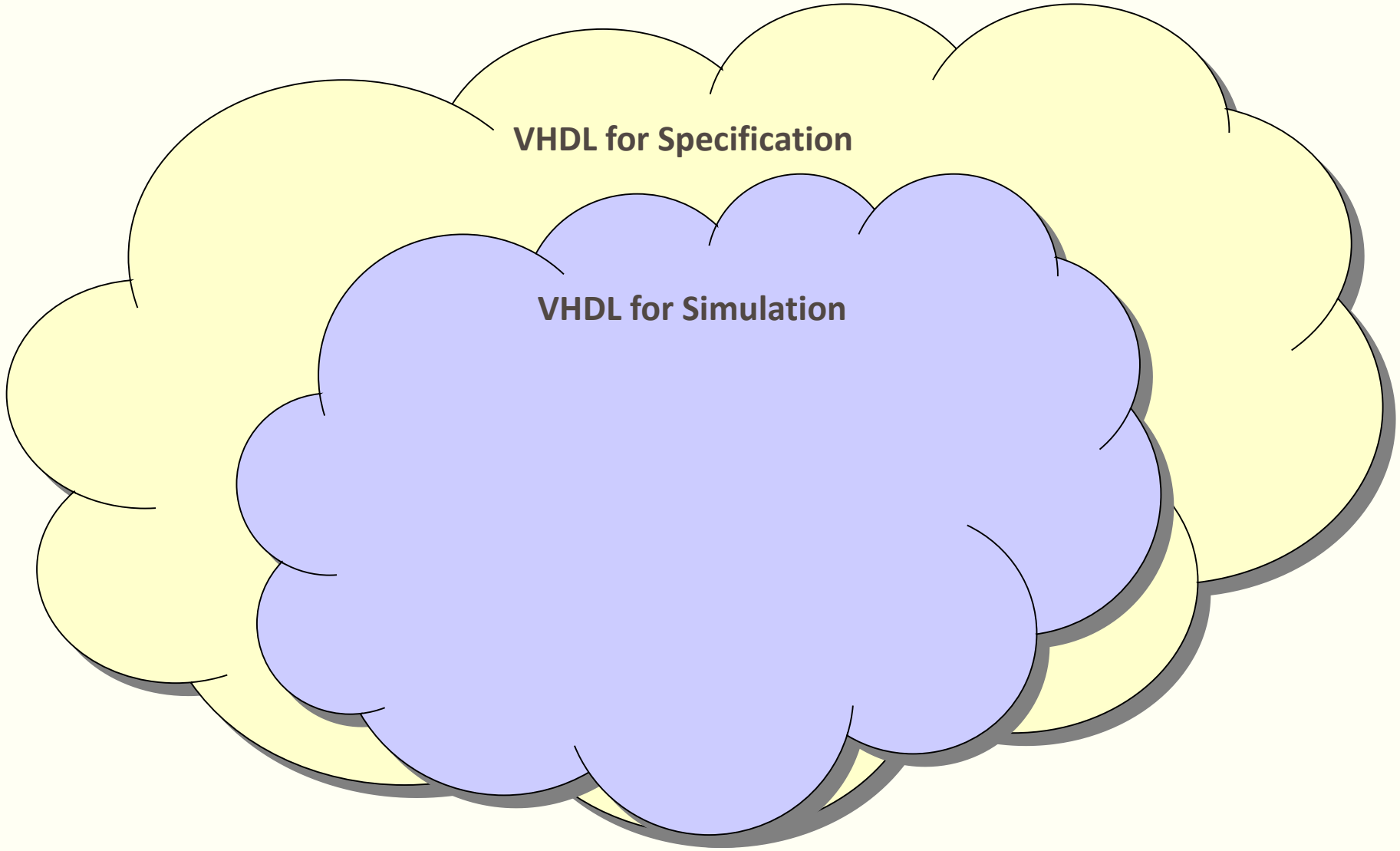
VHDL vs Verilog

VHDL	Verilog
Government Developed	Commercially Developed
Ada based	C based
Strongly Type Cast	Mildly Type Cast
Difficult to learn	Easier to Learn
More Powerful	Less Powerful

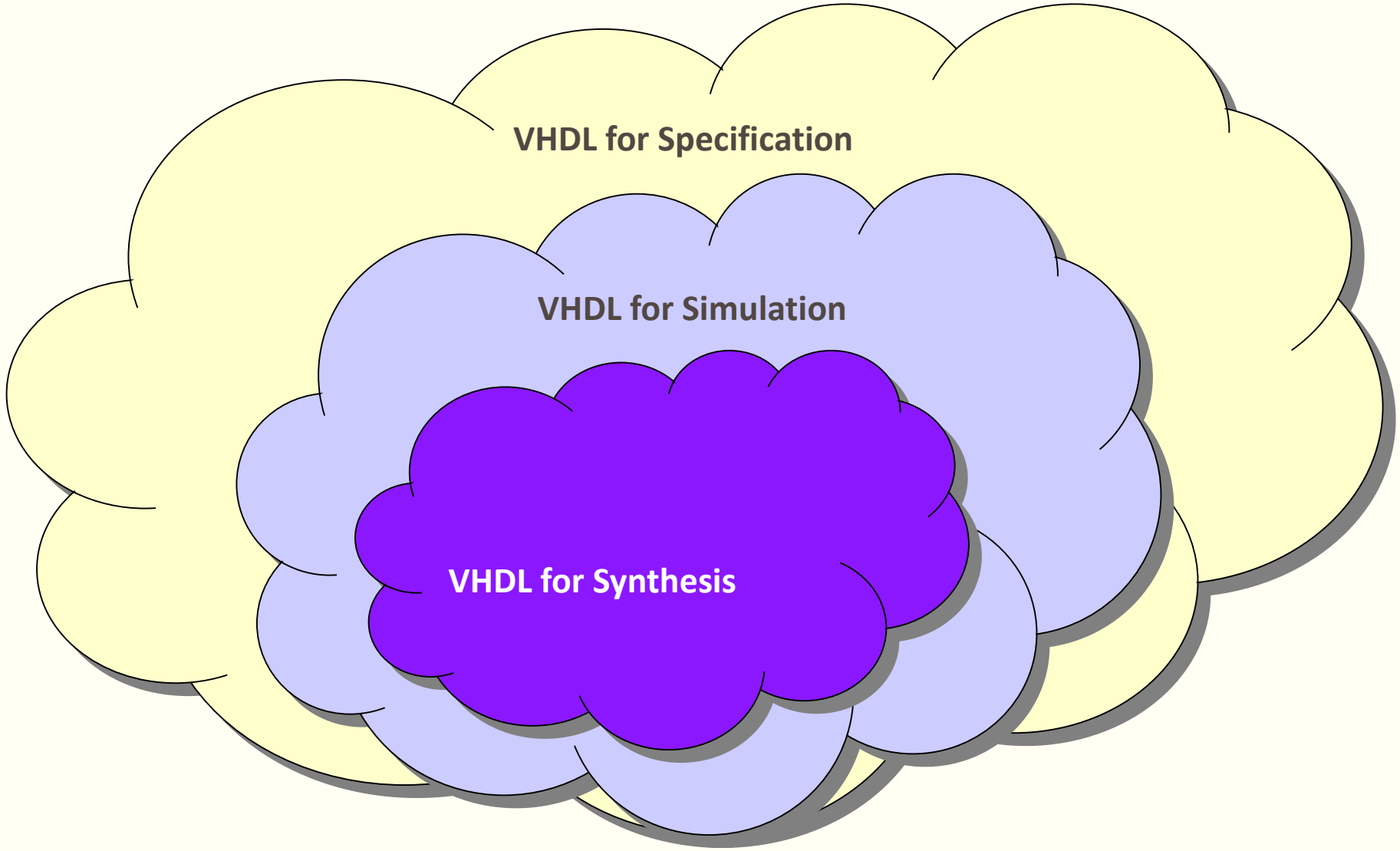
VHDL Subsets



VHDL Subsets



VHDL Subsets



Design Flow

Design and implement a simple unit permitting to speed up encryption with RC5-similar cipher with fixed key set on 8031 microcontroller. Unlike in the experiment 5, this time your unit has to be able to perform an encryption algorithm by itself, executing 32 rounds…..

Specification

Design Flow

Design and implement a simple unit permitting to speed up encryption with RC5-similar cipher with fixed key set on 8031 microcontroller. Unlike in the experiment 5, this time your unit has to be able to perform an encryption algorithm by itself, executing 32 rounds…..

Specification

VHDL description

```
Library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RC5_core is
        port(
            clock, reset, encr_decr: in std_logic;
            data_input: in std_logic_vector(31 downto 0);
            data_output: out std_logic_vector(31 downto 0);
            out_full: in std_logic;
            key_input: in std_logic_vector(31 downto 0);
            key_read: out std_logic;
        );
end AES_core;
```

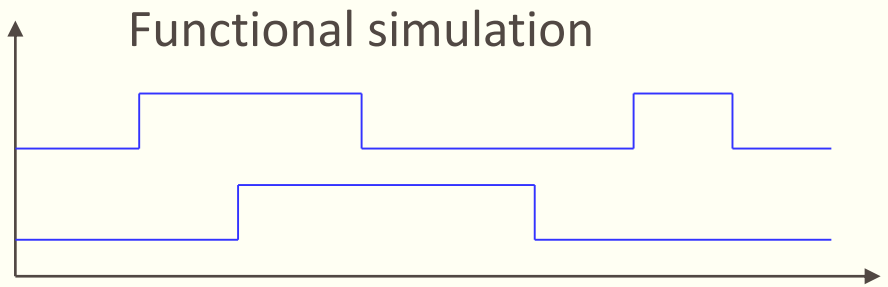
Design Flow

Design and implement a simple unit permitting to speed up encryption with RC5-similar cipher with fixed key set on 8031 microcontroller. Unlike in the experiment 5, this time your unit has to be able to perform an encryption algorithm by itself, executing 32 rounds…..

Specification

VHDL description

```
Library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RC5_core is
        port(
            clock, reset, encr_decr: in std_logic;
            data_input: in std_logic_vector(31 downto 0);
            data_output: out std_logic_vector(31 downto 0);
            out_full: in std_logic;
            key_input: in std_logic_vector(31 downto 0);
            key_read: out std_logic;
        );
end AES_core;
```



Design Flow

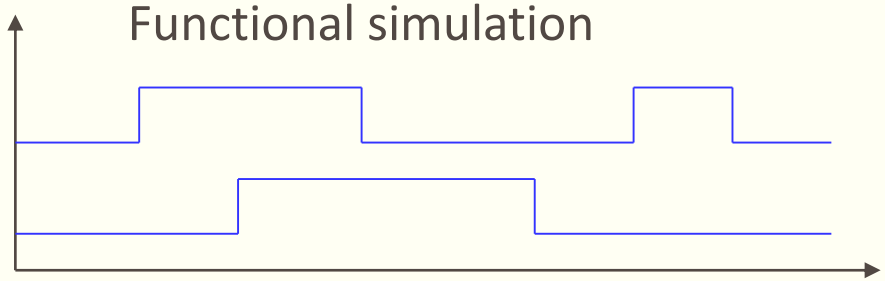
Design and implement a simple unit permitting to speed up encryption with RC5-similar cipher with fixed key set on 8031 microcontroller. Unlike in the experiment 5, this time your unit has to be able to perform an encryption algorithm by itself, executing 32 rounds…..

Specification

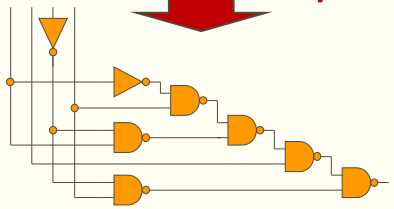
VHDL description

```
Library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RC5_core is
        port(
            clock, reset, encr_decr: in std_logic;
            data_input: in std_logic_vector(31 downto 0);
            data_output: out std_logic_vector(31 downto 0);
            out_full: in std_logic;
            key_input: in std_logic_vector(31 downto 0);
            key_read: out std_logic;
        );
end AES_core;
```

Functional simulation



Synthesis



Design Flow

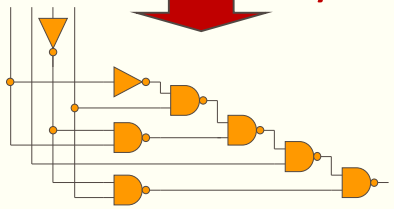
Design and implement a simple unit permitting to speed up encryption with RC5-similar cipher with fixed key set on 8031 microcontroller. Unlike in the experiment 5, this time your unit has to be able to perform an encryption algorithm by itself, executing 32 rounds…..

Specification

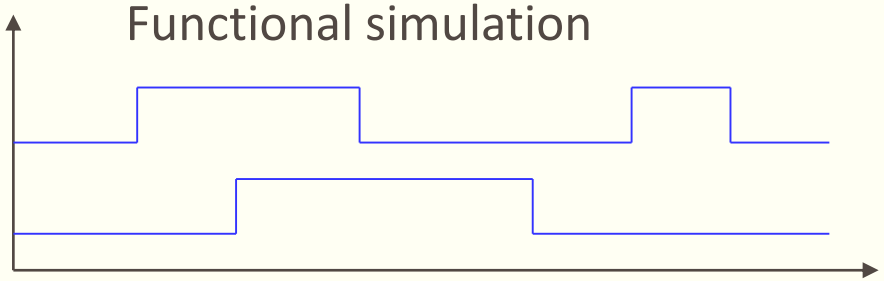
VHDL description

```
Library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RC5_core is
        port(
            clock, reset, encr_decr: in std_logic;
            data_input: in std_logic_vector(31 downto 0);
            data_output: out std_logic_vector(31 downto 0);
            out_full: in std_logic;
            key_input: in std_logic_vector(31 downto 0);
            key_read: out std_logic;
        );
end AES_core;
```

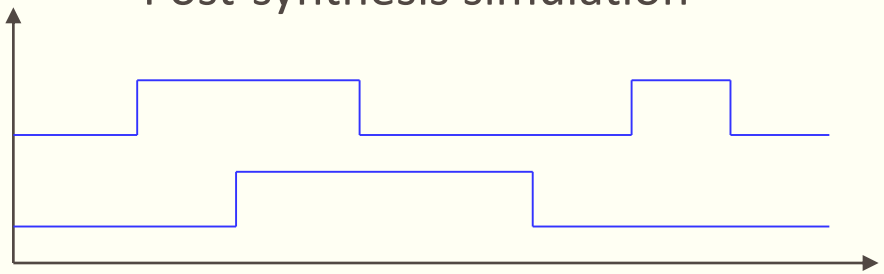
Synthesis



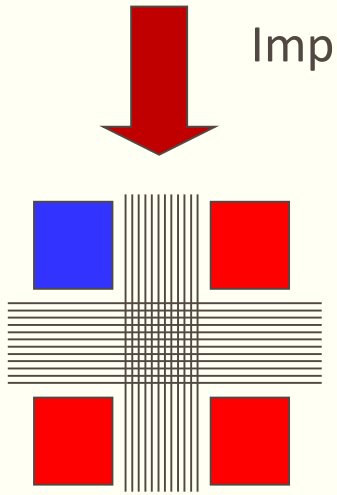
Functional simulation



Post-synthesis simulation

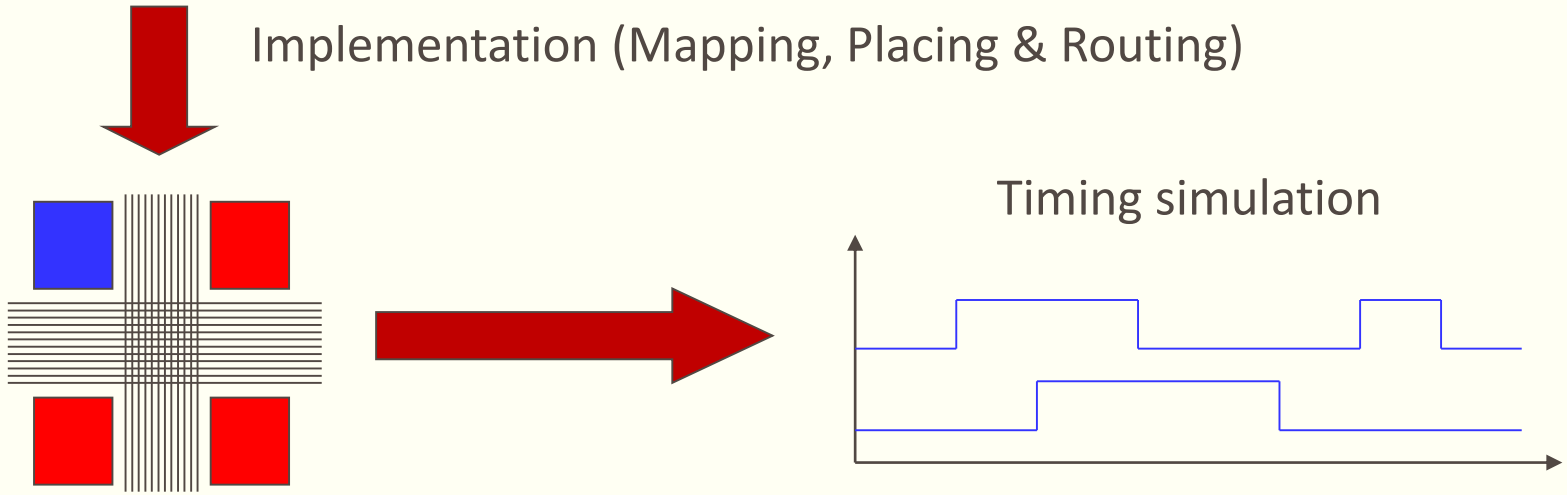


Physical Implementation Flow

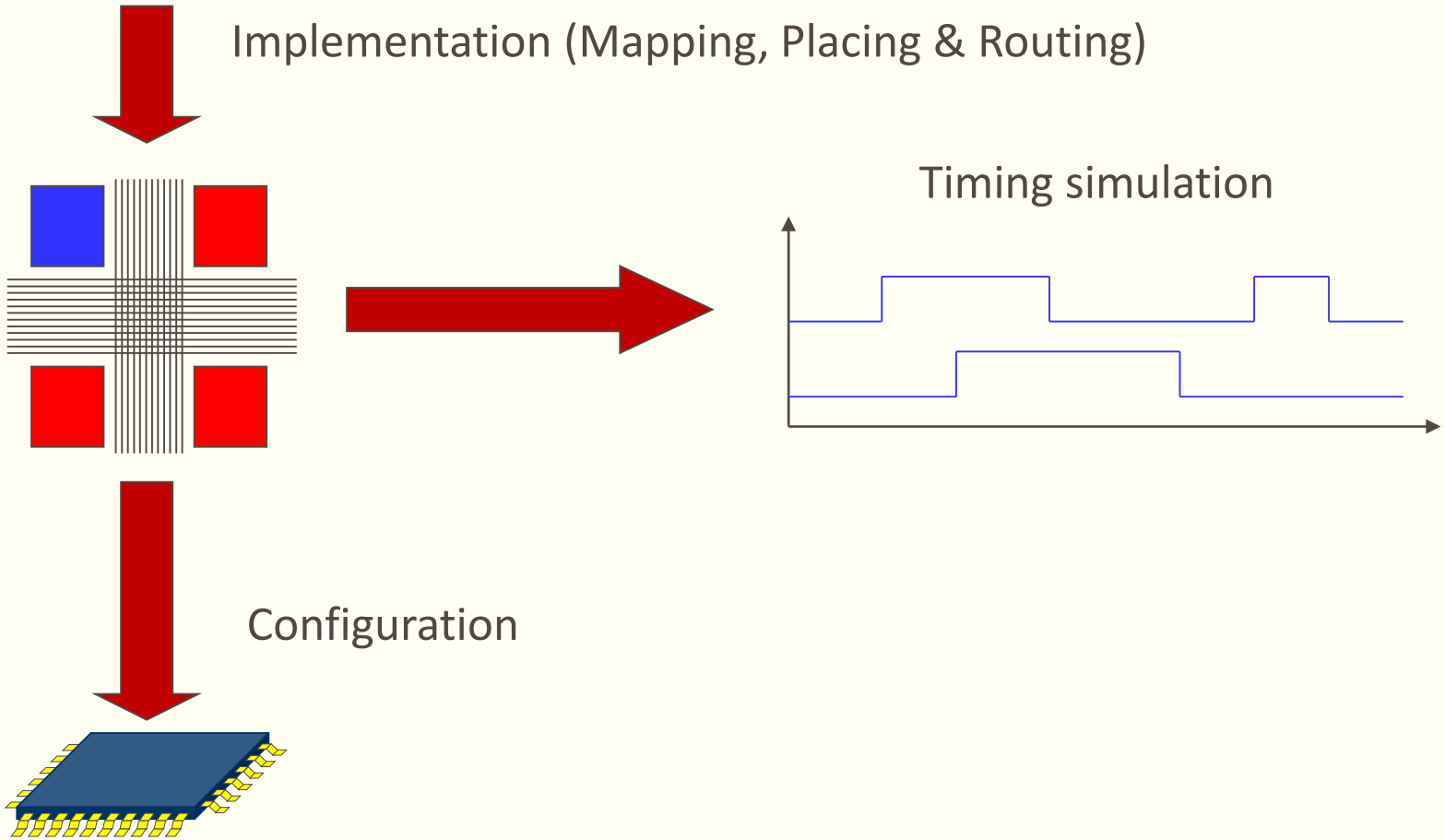


Implementation (Mapping, Placing & Routing)

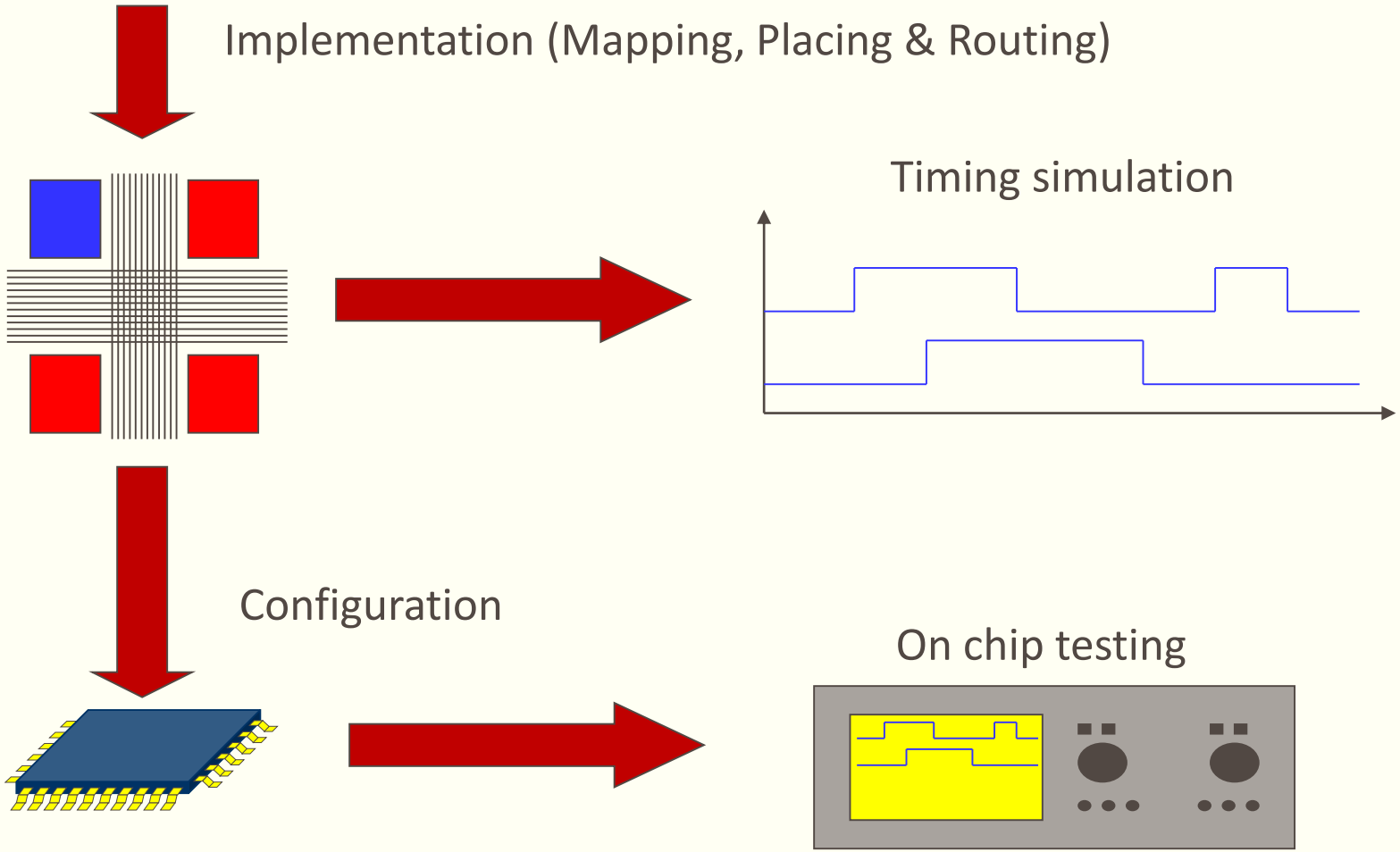
Physical Implementation Flow



Physical Implementation Flow



Physical Implementation Flow



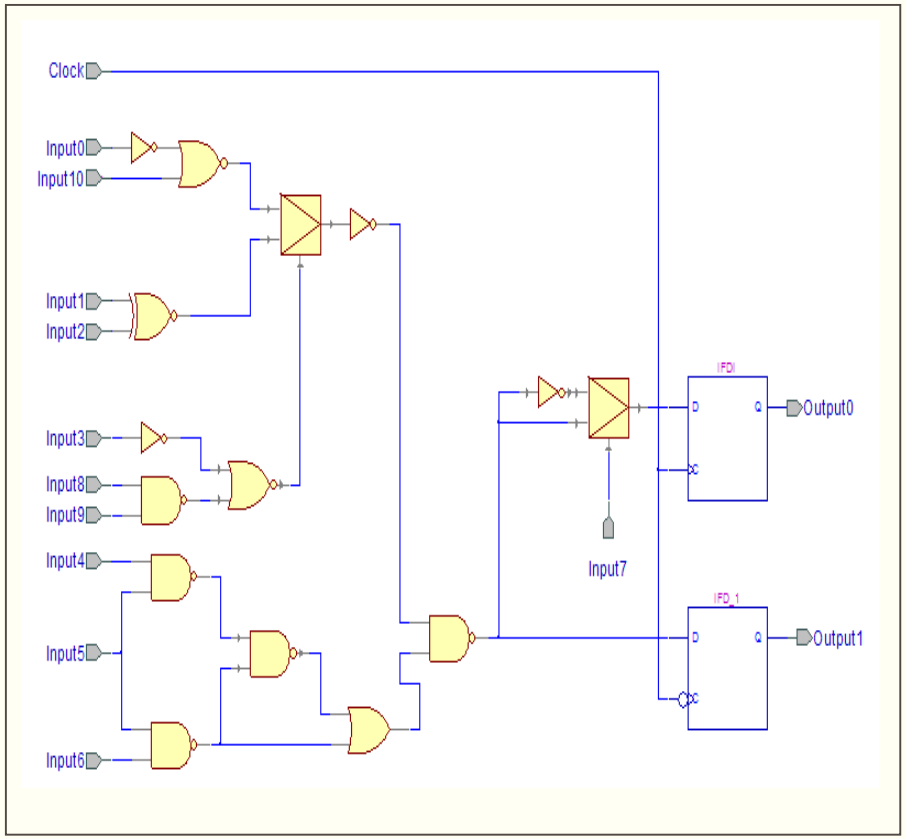
EDA Tools

- ModelSim – www.model.com, www.mentor.com
- ModelSim Xilinx Edition – www.xilinx.com
- Leonardo Spectrum – www.mentor.com
- Xilinx ISE – www.xilinx.com

Translating Code into Circuits

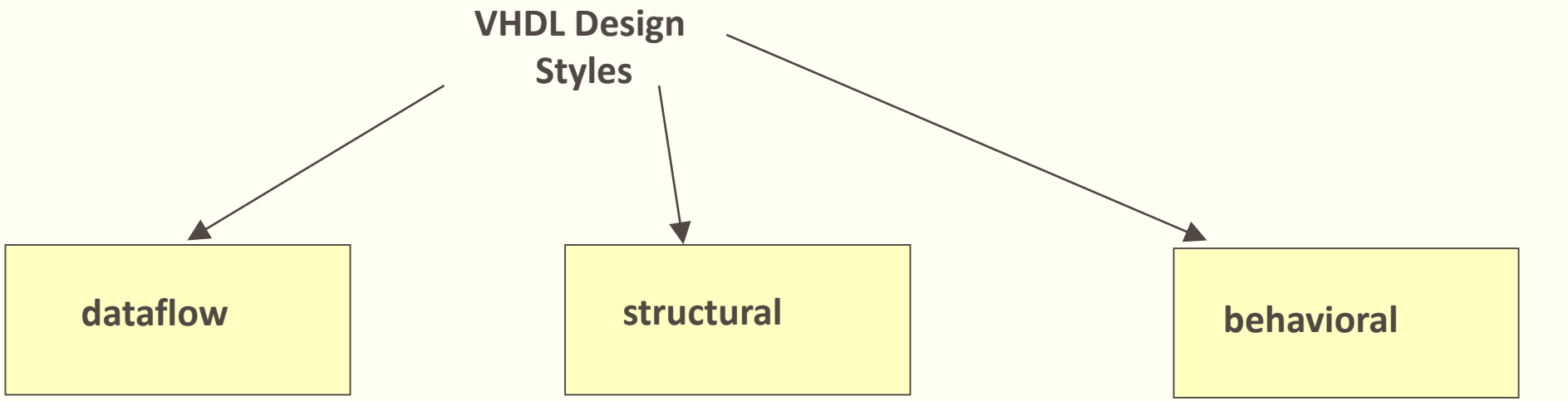
```
architecture MLU_DATAFLOW of MLU is
signal A1:STD_LOGIC;
signal B1:STD_LOGIC;
signal Y1:STD_LOGIC;
signal MUX_0, MUX_1, MUX_2, MUX_3: STD_LOGIC;

begin
   A1<=A when (NEG_A='0') else not A;
   B1<=B when (NEG_B='0') else not B;
   Y<=Y1 when (NEG_Y='0') else not Y1;

   MUX_0<=A1 and B1;
   MUX_1<=A1 or B1;
   MUX_2<=A1 xor B1;
   MUX_3<=A1 xnor B1;

   with (L1 & L0) select
      Y1 <= MUX_0 when "00",
          MUX_1 when "01",
          MUX_2 when "10",
          MUX_3 when others;
end MLU_DATAFLOW;
```

VHDL description



Circuit netlist

VHDL Design Styles



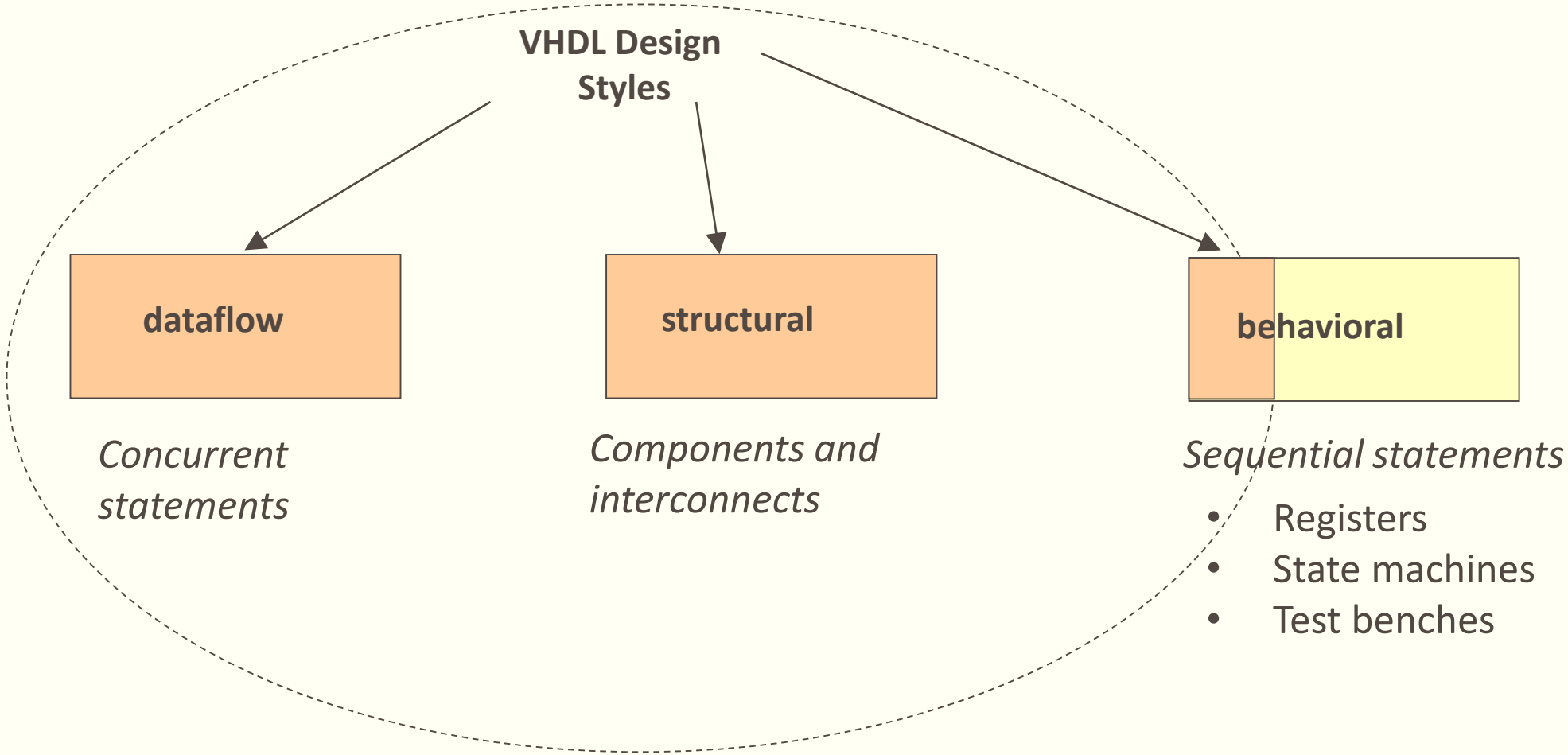
Concurrent statements

Components and interconnects

Sequential statements

- Registers
- State machines
- Test benches

VHDL Design Styles



Dataflow Description

- Describes how data moves through the system and the various processing steps.
- Data Flow uses series of concurrent statements to realize logic.
  - Concurrent statements are evaluated at the same time; thus, order of these statements doesn't matter.
- Data Flow is most useful style when series of Boolean equations can represent a logic.
- Data Flow describes best combinational logic.

Structural Description

- Structural design is the simplest to understand.
  - This style is the closest to schematic capture and utilizes simple building blocks to compose logic functions.
- Components are interconnected in a hierarchical manner.
- Structural descriptions may connect simple gates or complex, abstract components.
- Structural style is useful when expressing a design that is naturally composed of sub-blocks.

Behavioral Description

- It accurately models what happens on the inputs and outputs of the black box (no matter what is inside and how it works).
- This style uses PROCESS and sequential statements to realize logic, which are executed one after the other, like in most programming languages.
- Behavioral descriptions describe best sequential logic.

Naming and Labeling

- VHDL is not case sensitive

  Example:

  Names or labels

  databus

  Databus

  DataBus

  DATABUS

  are all equivalent

Naming and Labeling

General rules of thumb (according to VHDL-87)

1. All names should start with an alphabet character (a-z or A-Z)
2. Use only alphabet characters (a-z or A-Z) digits (0-9) and underscore (_)
3. Do not use any punctuation or reserved characters within a name (!, ?, ., &, +, -, etc.)
4. Do not use two or more consecutive underscore characters (__) within a name (e.g., Sel__A is invalid)
5. All names and labels in a given entity and architecture must be unique

Free Format

- VHDL is a “free format” language
- No formatting conventions, such as spacing or indentation imposed by VHDL compilers. Space and carriage return treated the same way.

Example:

```
if (a=b) then
or
   if      (a=b)     then
or
   if (a =
   b) then
are all equivalent
```

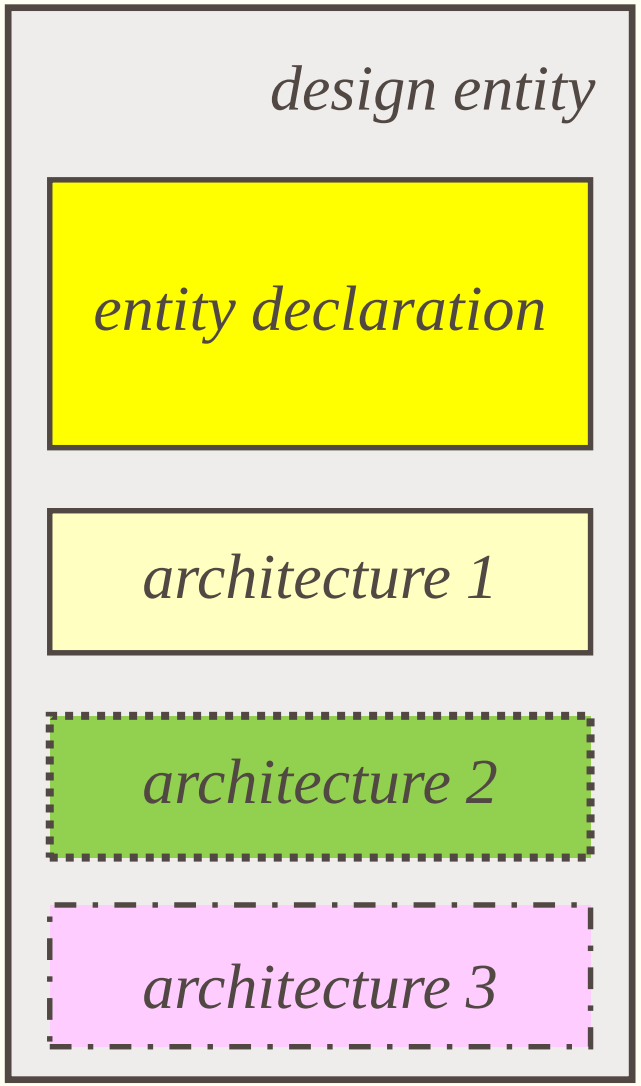
Comments

- Comments in VHDL are indicated with a “double dash”, i.e., “--”
  - Comment indicator can be placed anywhere in the line
  - Any text that follows in the same line is treated as a comment
  - Carriage return terminates a comment
  - No method for commenting a block extending over a couple of lines

Examples:

```
-- main subcircuit
Data_in <= Data_bus;     -- reading data from the input
FIFO
```

Design Entity

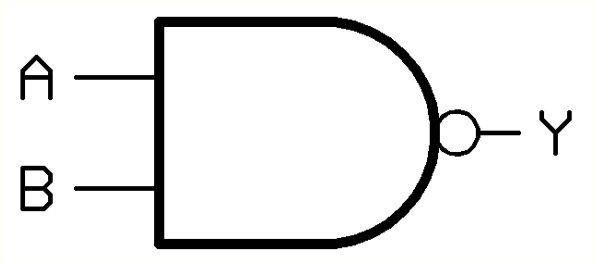


Design Entity - most basic building block of a design

One *entity* can have many different *architectures*

Entity Declaration

Entity Declaration describes the interface of the component, i.e. input and output ports.



Entity name

Port type

Semicolon

Reserved words

No Semicolon

Port names

Port modes (data flow directions)

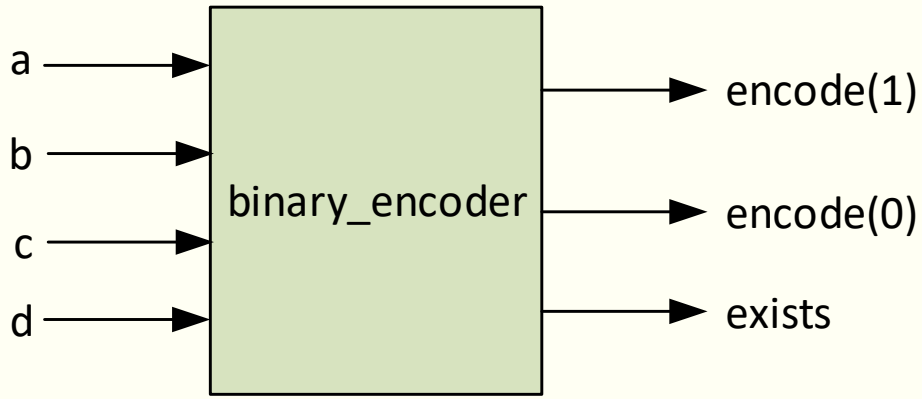
```
ENTITY nand_gate IS
    PORT(
        A    : IN STD_LOGIC;
        B    : IN STD_LOGIC;
        Y    : OUT STD_LOGIC
        );
END nand_gate;
```

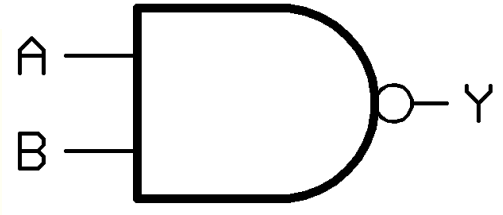
Entity Declaration - Syntax

```
ENTITY entity_name IS
          PORT (
                  port_name :  signal_mode  signal_type;
                  port_name :  signal_mode  signal_type;
                  ……….
                  port_name :  signal_mode  signal_type);
END entity_name;
```

Entity Declaration - Example

Write the entity declaration for the 4-2 binary encoder

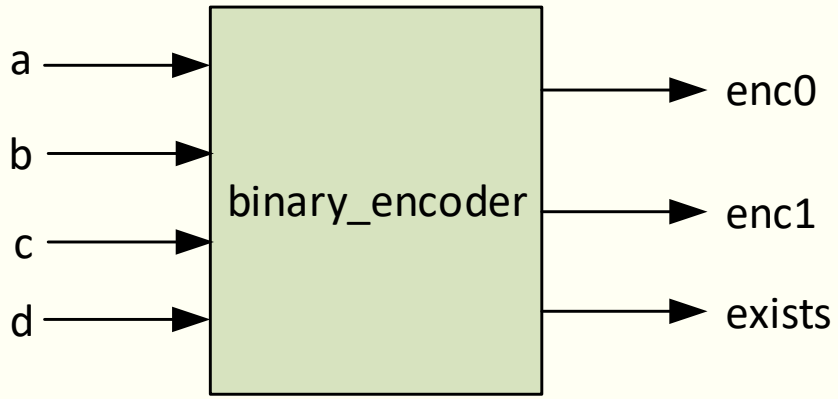





```
ENTITY nand_gate IS
    PORT(
        A    : IN STD_LOGIC;
        B    : IN STD_LOGIC;
        Y    : OUT STD_LOGIC
        );
END nand_gate;
```

Entity Declaration - Example

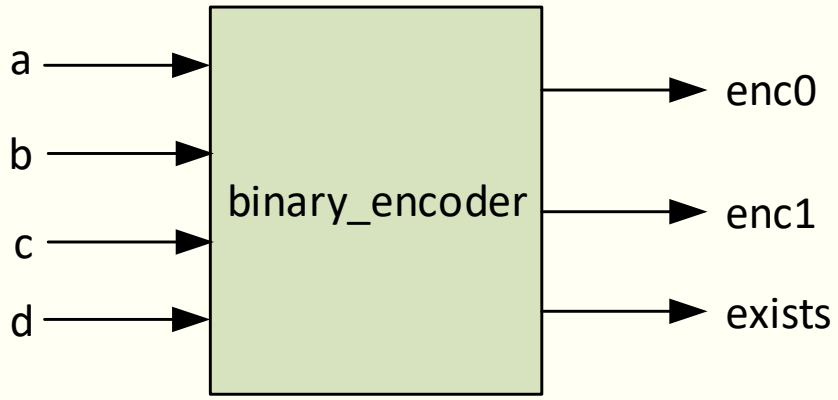
Write the entity declaration for the 4-2 binary encoder (with priority)




```
entity binary_encoder is
   port (
   a: in std_logic;
   b: in std_logic;
   c: in std_logic;
   d: in std_logic;
   enc0: out std_logic;
   enc1: out std_logic;
   exists: inout std_logic
   );
end binary_encoder;
```

Entity Declaration - Example

Write the entity declaration for the 4-2 binary encoder (with priority)



Refinement

```
entity binary_encoder is
   port (
   a: in std_logic;
   b: in std_logic;
   c: in std_logic;
   d: in std_logic;
   enc: out std_logic_vector(1 downto 0);
   exists: inout std_logic
   );
end binary_encoder;
```

Architecture

- Describes an implementation of a design entity.
- Architecture example:

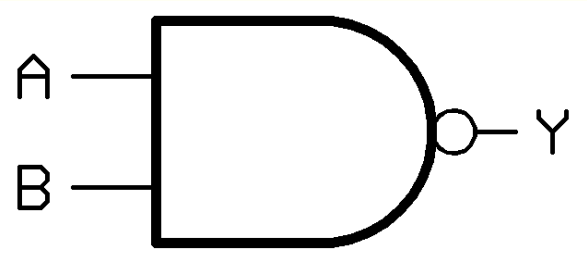
```
ARCHITECTURE model OF nand_gate IS
BEGIN
  z <= a NAND b;
END model;
```

Architecture - Syntax

```
ARCHITECTURE architecture_name OF entity_name IS
    [ declarations ]
BEGIN
     code
END architecture_name ;
```

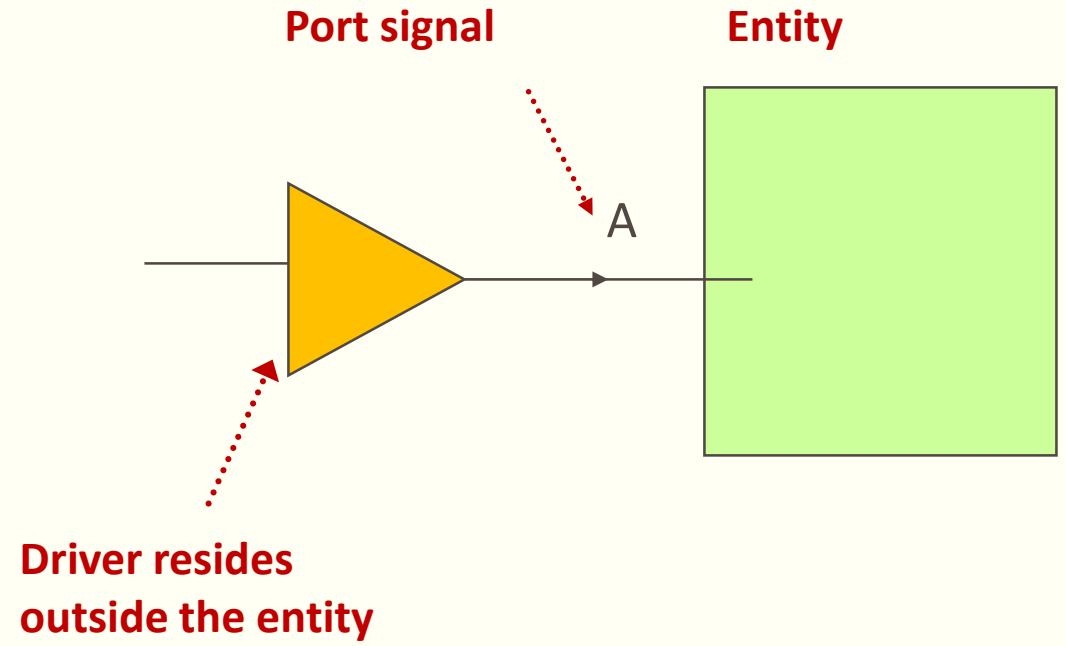
Entity Declaration & Architecture

nand_gate.vhd

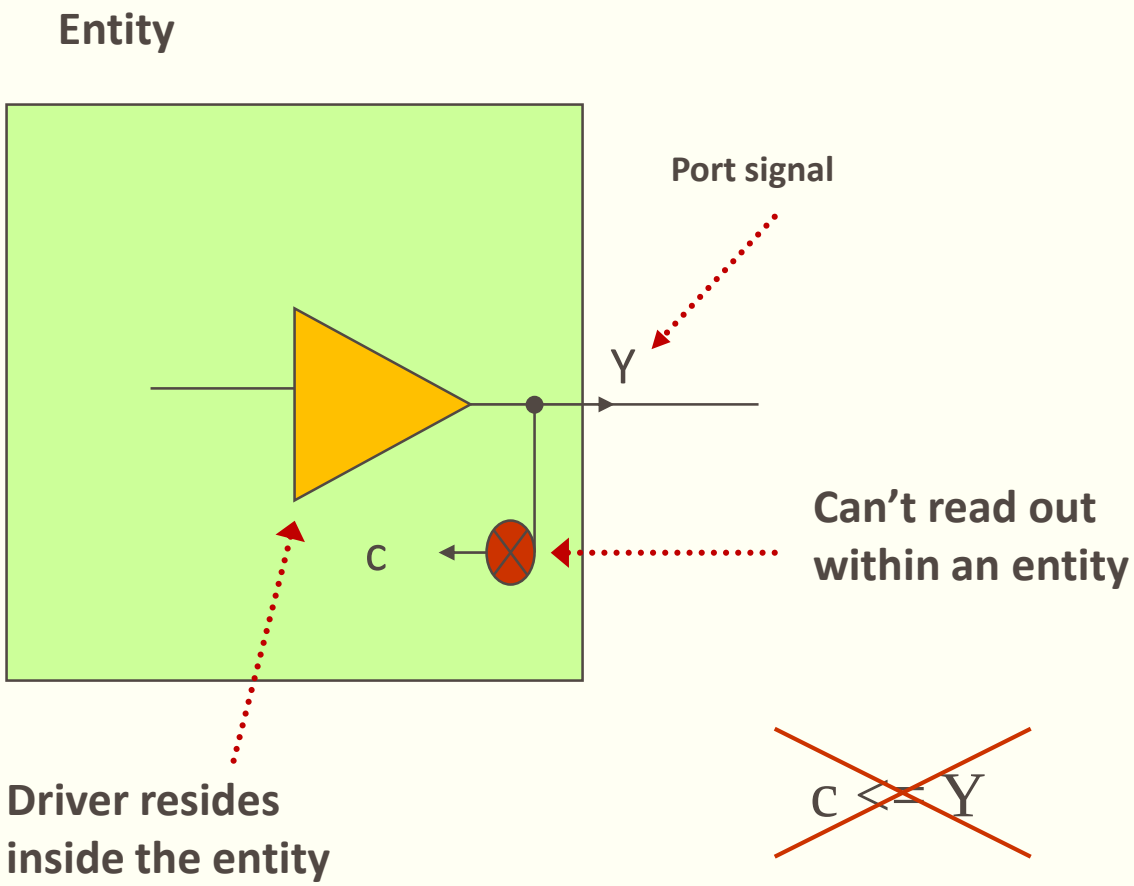



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY nand_gate IS
   PORT(A : IN STD_LOGIC;
        B : IN STD_LOGIC;
        Y : OUT STD_LOGIC);
END nand_gate;

ARCHITECTURE model OF nand_gate IS
BEGIN
  Y <= A NAND B;
END model;
```

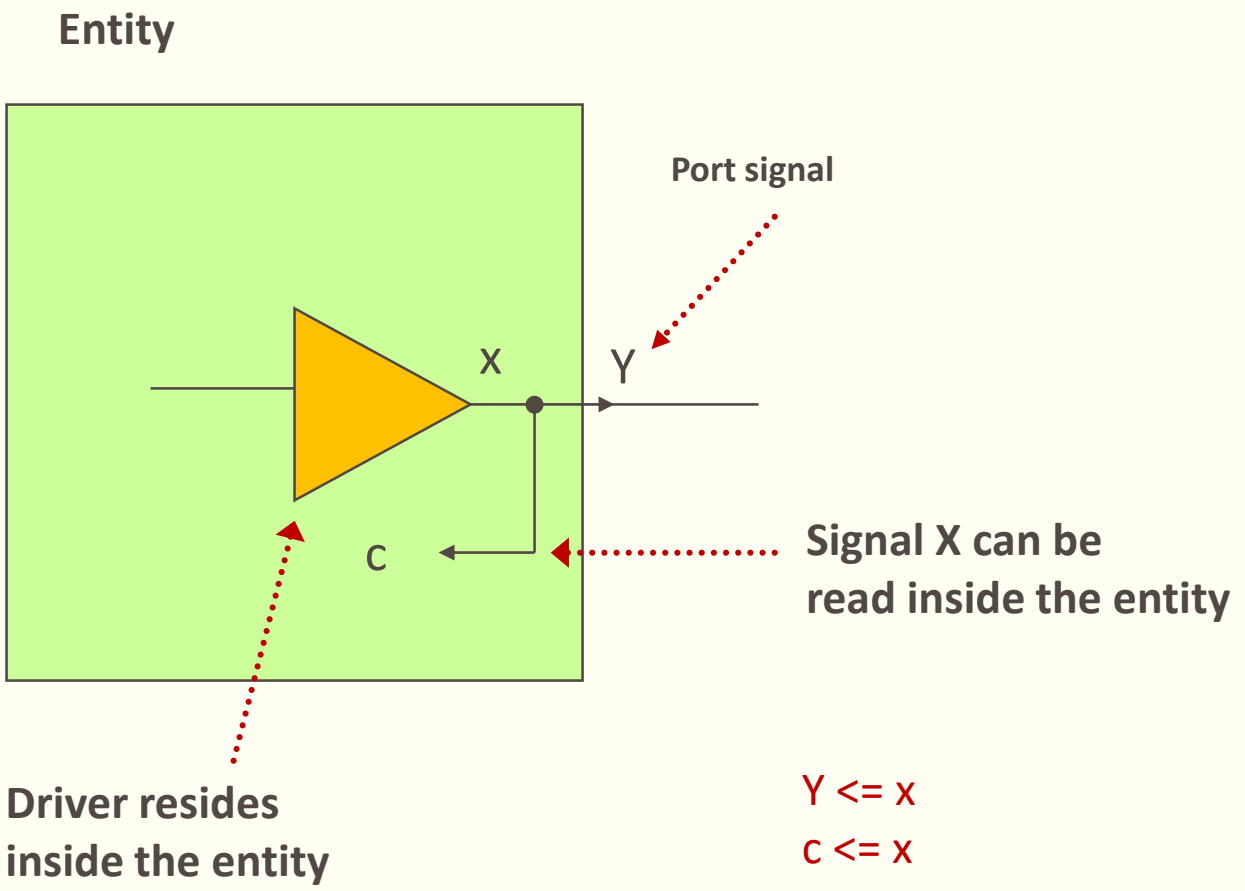
Port Mode In



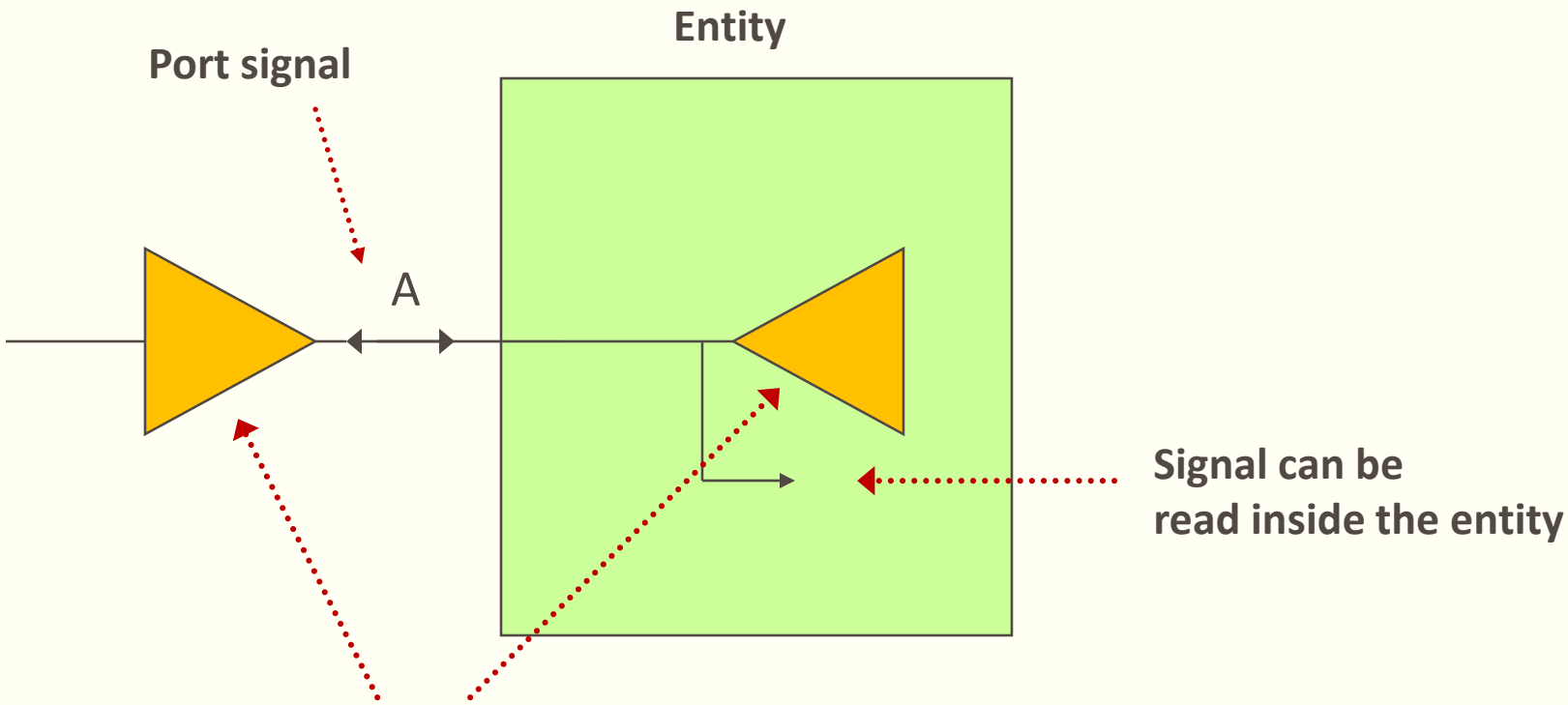
Port Mode Out



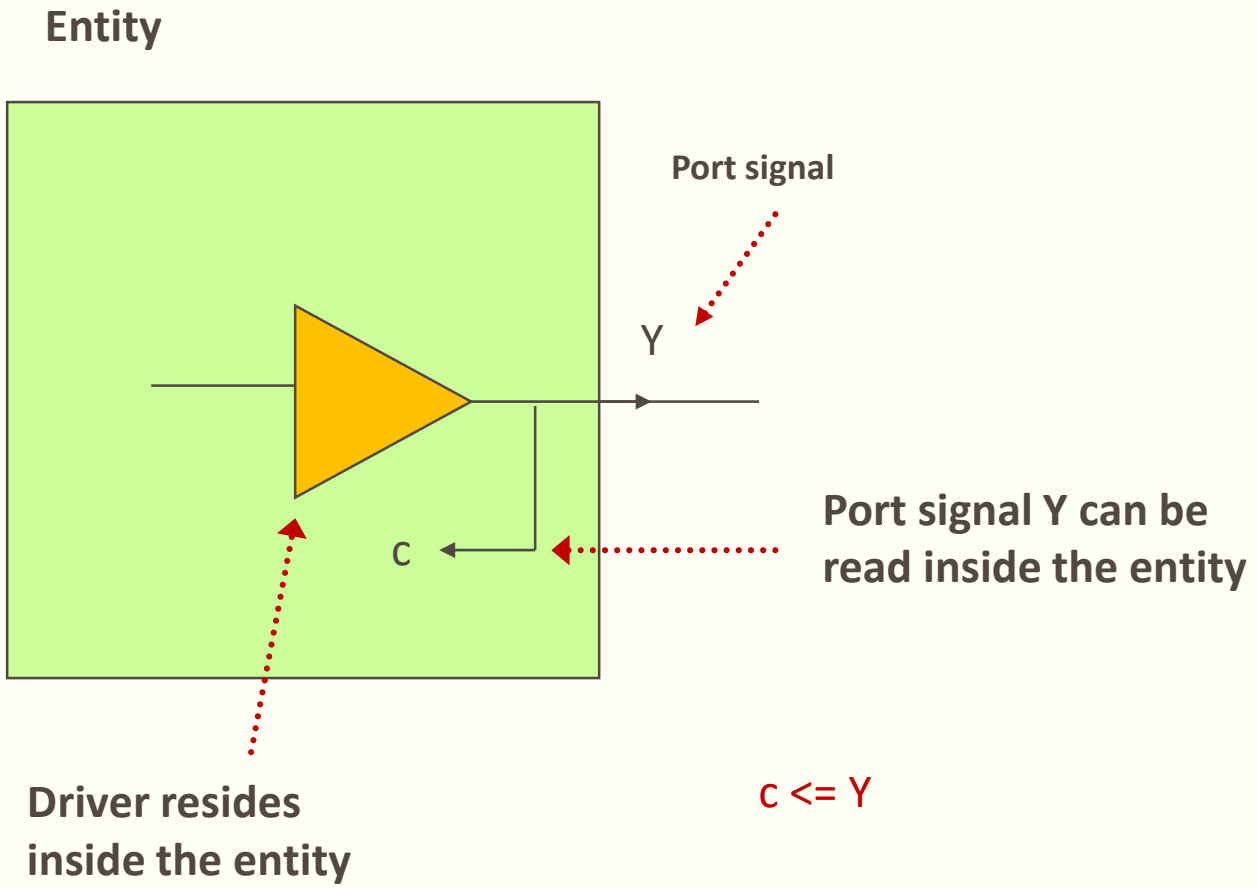
Port Mode Out with Signal



Port Mode Inout



Port Mode Buffer



Port Modes

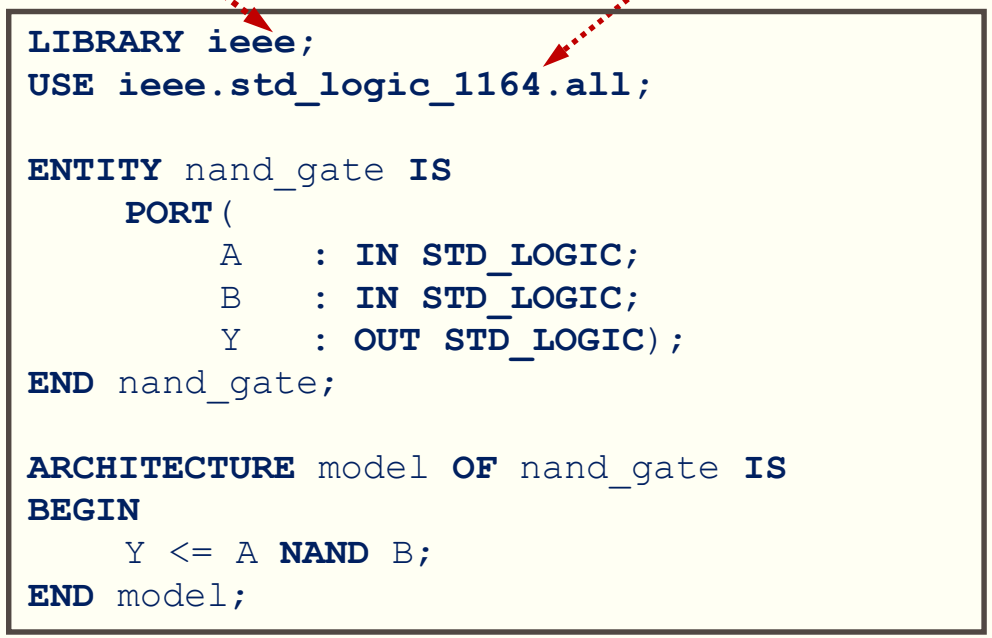
The Port Mode of the interface describes the direction in which data travels with respect to the component

- **In**: Data comes in this port and can only be read within the entity. It can appear **only on the right side** of a signal or variable assignment.
- **Out**: The value of an output port can only be updated within the entity. **It cannot be read**. It can only appear **on the left side** of a signal assignment.
- **Inout**: The value of a bi-directional port can be read and updated within the entity model. It can appear on **both sides** of a signal assignment.
- **Buffer:** Used for a signal that is an output from an entity. The value of the signal can be used inside the entity, which means that in an assignment statement the signal can appear on the left and right sides of the <= operator

Library Declaration

Library declaration

Use all definitions from the package std_logic_1164

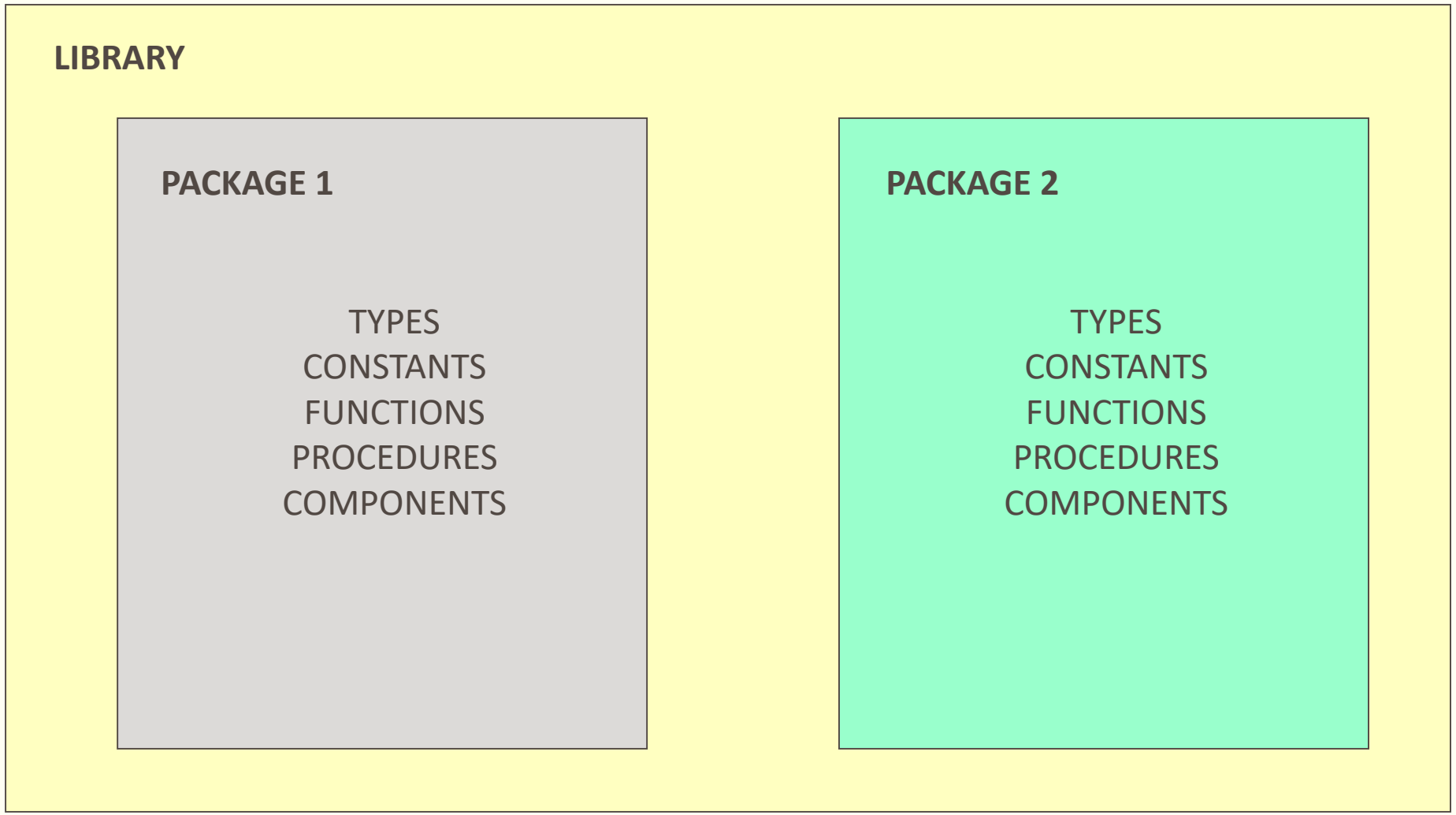


```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY nand_gate IS
     PORT(
          A    : IN STD_LOGIC;
          B    : IN STD_LOGIC;
          Y    : OUT STD_LOGIC);
END nand_gate;

ARCHITECTURE model OF nand_gate IS
BEGIN
     Y <= A NAND B;
END model;
```

Library Declaration - Syntax

```
LIBRARY  library_name;
USE library_name.package_name.package_parts;
```

Parts of a Library



Common Libraries

ieee

Specifies multi-level logic system, including STD_LOGIC, and STD_LOGIC_VECTOR data types.

Need to be explicitly declared

std

Specifies pre-defined data types (BIT, BOOLEAN, INTEGER, REAL, SIGNED, UNSIGNED, etc.), arithmetic operations, basic type conversion functions, basic text i/o functions, etc.

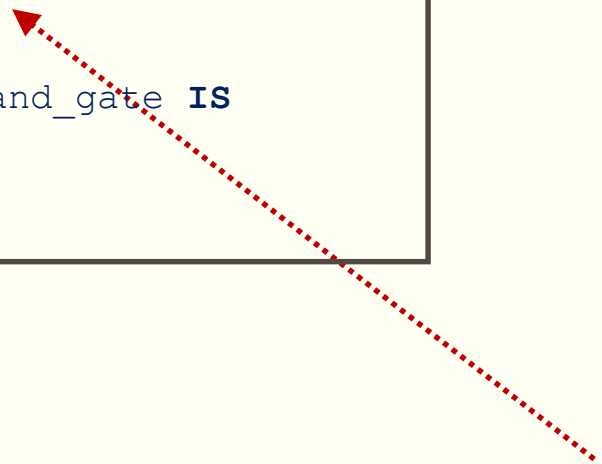
work

Current designs after compilation.

Visible by default

STD_LOGIC

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY nand_gate IS
     PORT(
          A    : IN STD_LOGIC;
          B    : IN STD_LOGIC;
          Y    : OUT STD_LOGIC);
END nand_gate;

ARCHITECTURE model OF nand_gate IS
BEGIN
     Y <= A NAND B;
END model;
```



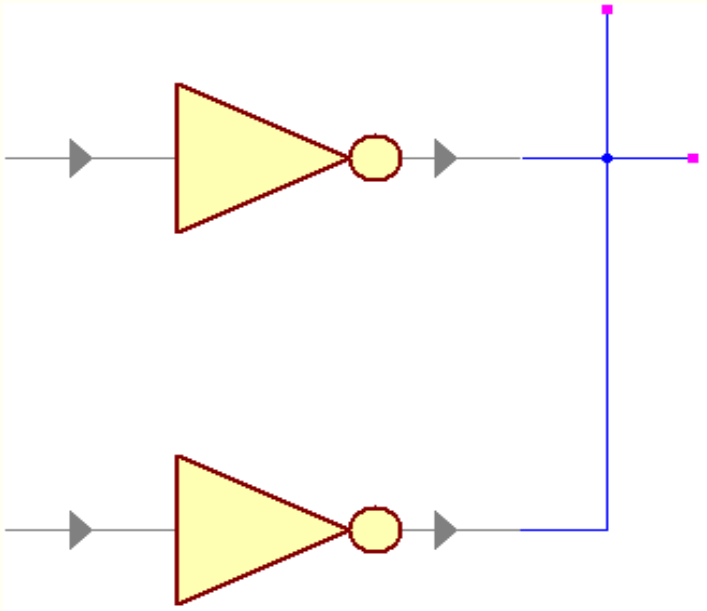
What is **STD_LOGIC** you ask?

STD_LOGIC Type

Value	Meaning
‘X’	Forcing (Strong driven) Unknown
‘0’	Forcing (Strong driven) 0
‘1’	Forcing (Strong driven) 1
‘Z’	High Impedance
‘W’	Weak (Weakly driven) Unknown
‘L’	Weak (Weakly driven) 0. Models a pull down.
‘H’	Weak (Weakly driven) 1. Models a pull up.
‘-’	Don't Care

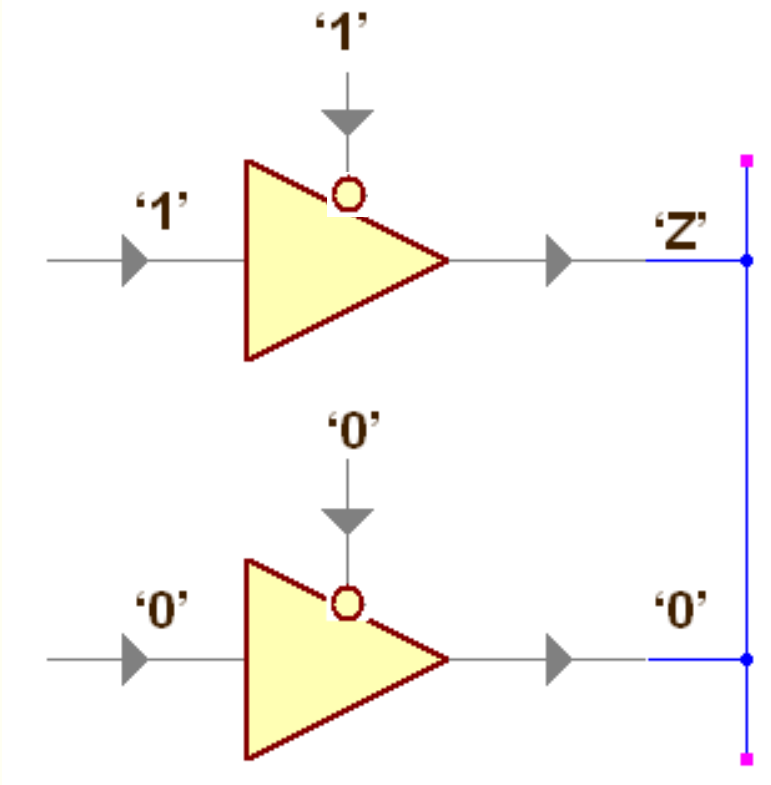
STD_LOGIC Meaning

X

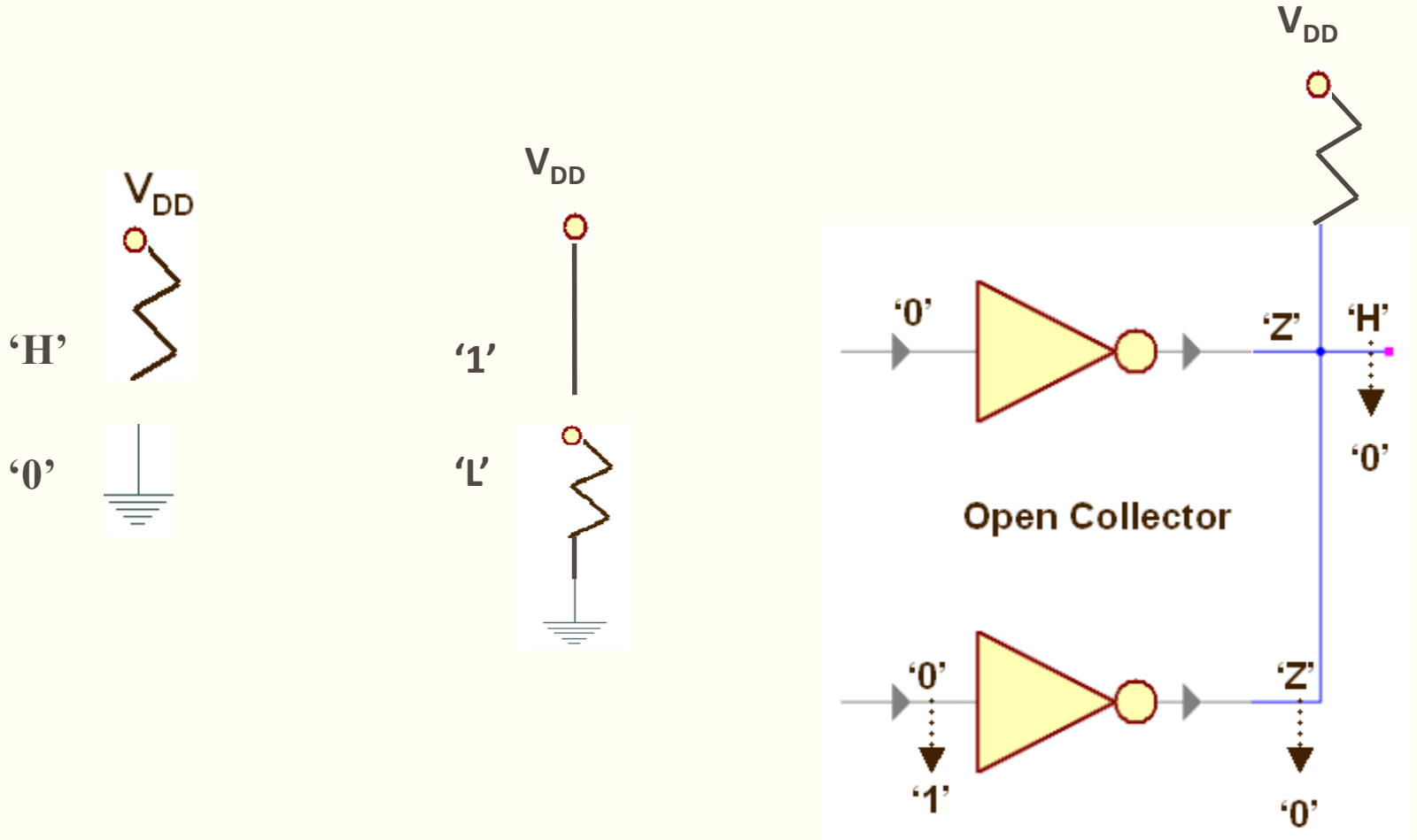


Contention on the bus

STD_LOGIC Meaning



STD_LOGIC Meaning



STD_LOGIC Meaning

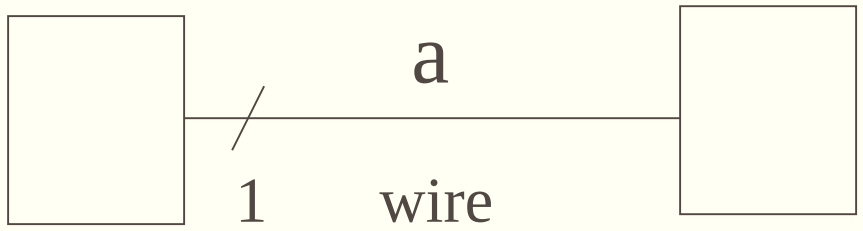
- Do not care
- Can be assigned to outputs for the case of invalid inputs(may produce significant improvement in resource utilization after synthesis).
- Use with caution ‘1’ = ‘-’ give FALSE

Resolving Logic Levels

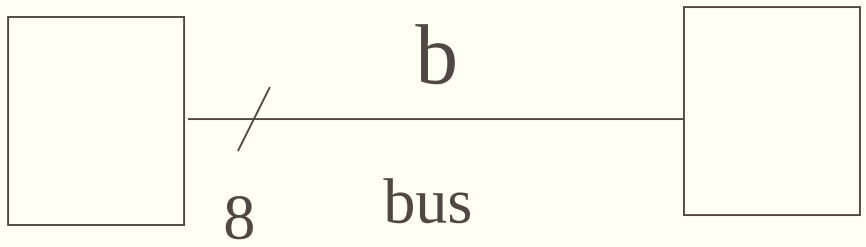
	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

Wires and Buses

SIGNAL a : STD_LOGIC;



SIGNAL b : STD_LOGIC_VECTOR(7 DOWNTO 0);



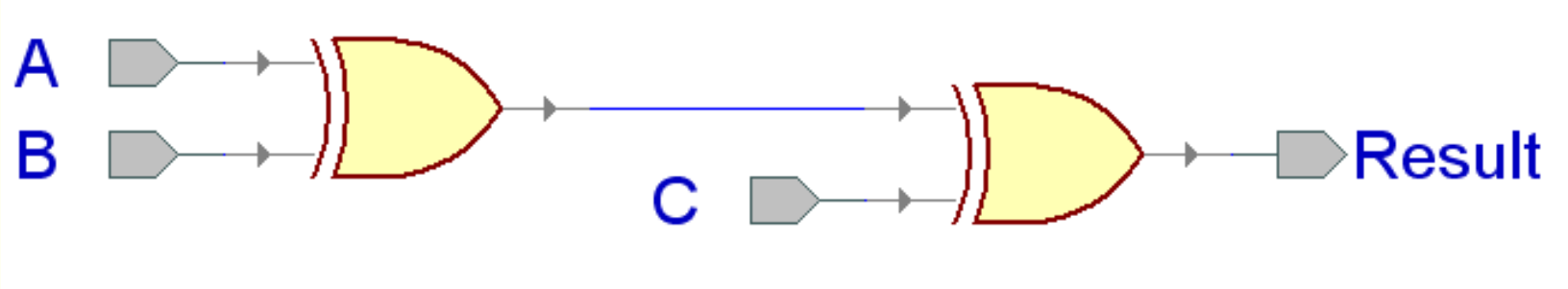
Standard Logic Vectors

```
SIGNAL a: STD_LOGIC;
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL c: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL f: STD_LOGIC_VECTOR(8 DOWNTO 0);

a <= '1';
b <= "0000";                -- Binary base assumed by default
c <= B"0000";               -- Binary base explicitly specified
d <= "0110_0111";           -- You can use '_' to increase readability
e <= X"AF67";               -- Hexadecimal base
f <= O"723";                -- Octal base
```

Concatenation

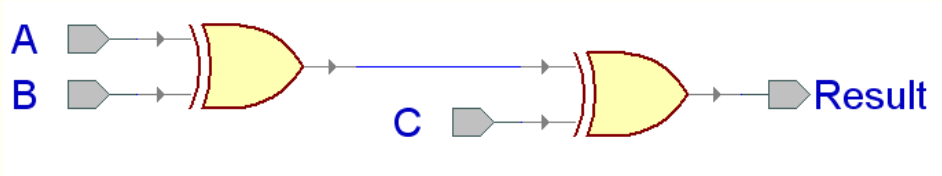
```
SIGNAL a: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL c, d, e: STD_LOGIC_VECTOR(7 DOWNTO 0);

a <= "0000";
b <= "1111";
c <= a & b;                    -- c = "0000_1111"

d <= '0' & "0001111";          -- d <= "0_0001111"

e <= '0' & '0' & '0' & '0' & '1' & '1' & '1' & '1';
                               -- e <= "00001111"
```

XOR3 Example



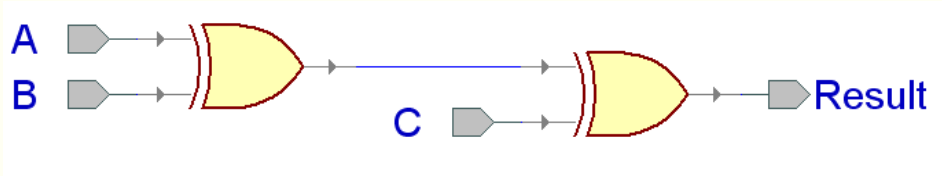
Give the entity declaration for the XOR3 gate

Entity xor3




```
ENTITY xor3 IS
    PORT(
        A : IN STD_LOGIC;
        B : IN STD_LOGIC;
        C : IN STD_LOGIC;
        Result : OUT STD_LOGIC
        );
end xor3;
```

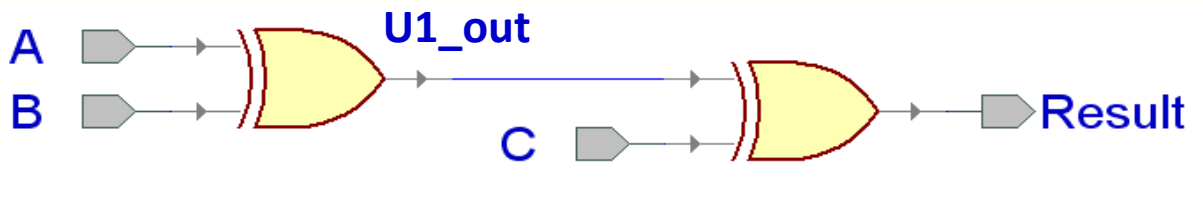
Entity xor3




```
ENTITY xor3 IS
    PORT(
        A : IN STD_LOGIC;
        B : IN STD_LOGIC;
        C : IN STD_LOGIC;
        Result : OUT STD_LOGIC
        );
end xor3;
```

Give the architecture (dataflow) for the XOR3 gate

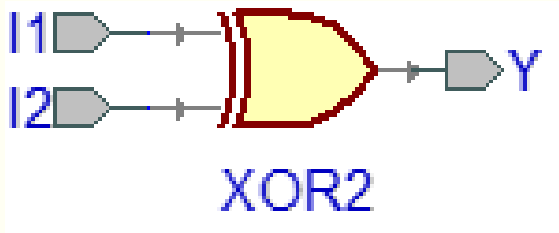
Dataflow Architecture

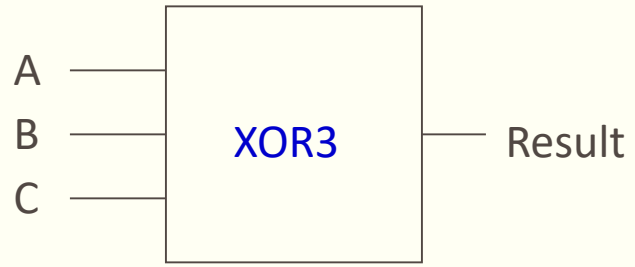


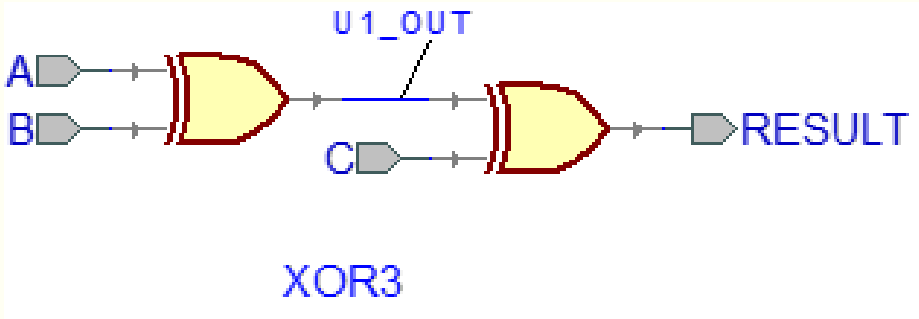

```
ARCHITECTURE dataflow OF xor3 IS
SIGNAL U1_out: STD_LOGIC;
BEGIN
  U1_out <=A XOR B;
  Result <=U1_out XOR C;
END dataflow;
```

Structural Architecture

```
ARCHITECTURE structural OF xor3 IS
SIGNAL U1_OUT: STD_LOGIC;
COMPONENT xor2
    PORT(
     I1 : IN STD_LOGIC;
     I2 : IN STD_LOGIC;
     Y : OUT STD_LOGIC
        );
END COMPONENT;

BEGIN
   U1: xor2 PORT MAP (I1 => A,
                      I2 => B,
              Y  => U1_OUT);

   U2: xor2 PORT MAP (I1 => U1_OUT,
                      I2 => C,
                      Y  => Result);
END structural;
```







Component Instantiation

- Named association connectivity (recommended)

```
COMPONENT xor2 IS
  PORT(
     I1 : IN STD_LOGIC;
     I2 : IN STD_LOGIC;
     Y  : OUT STD_LOGIC
       );
END COMPONENT;

U1: xor2 PORT MAP (I1 => A,
                   I2 => B,
                   Y  => U1_OUT);
```

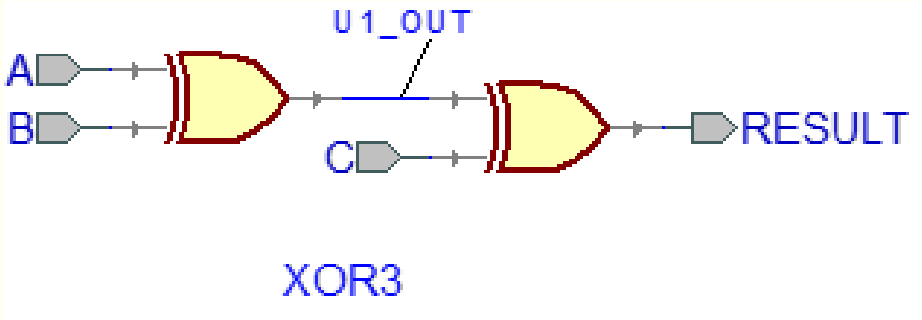
Component Instantiation

- Positional association connectivity (not recommended)

```
COMPONENT xor2 IS
  PORT(
     I1 : IN STD_LOGIC;
     I2 : IN STD_LOGIC;
     Y  : OUT STD_LOGIC
        );
END COMPONENT;

U1: xor2 PORT MAP (A, B, U1_OUT);
```

Behavioral Architecture

```
ARCHITECTURE behavioral OF xor3 IS
BEGIN
xor3_behave: PROCESS (A,B,C)
BEGIN
   IF ((A XOR B XOR C) = '1') THEN
     Result <= '1';
   ELSE
     Result <= '0';
   END IF;
END PROCESS xor3_behave;
END behavioral;
```

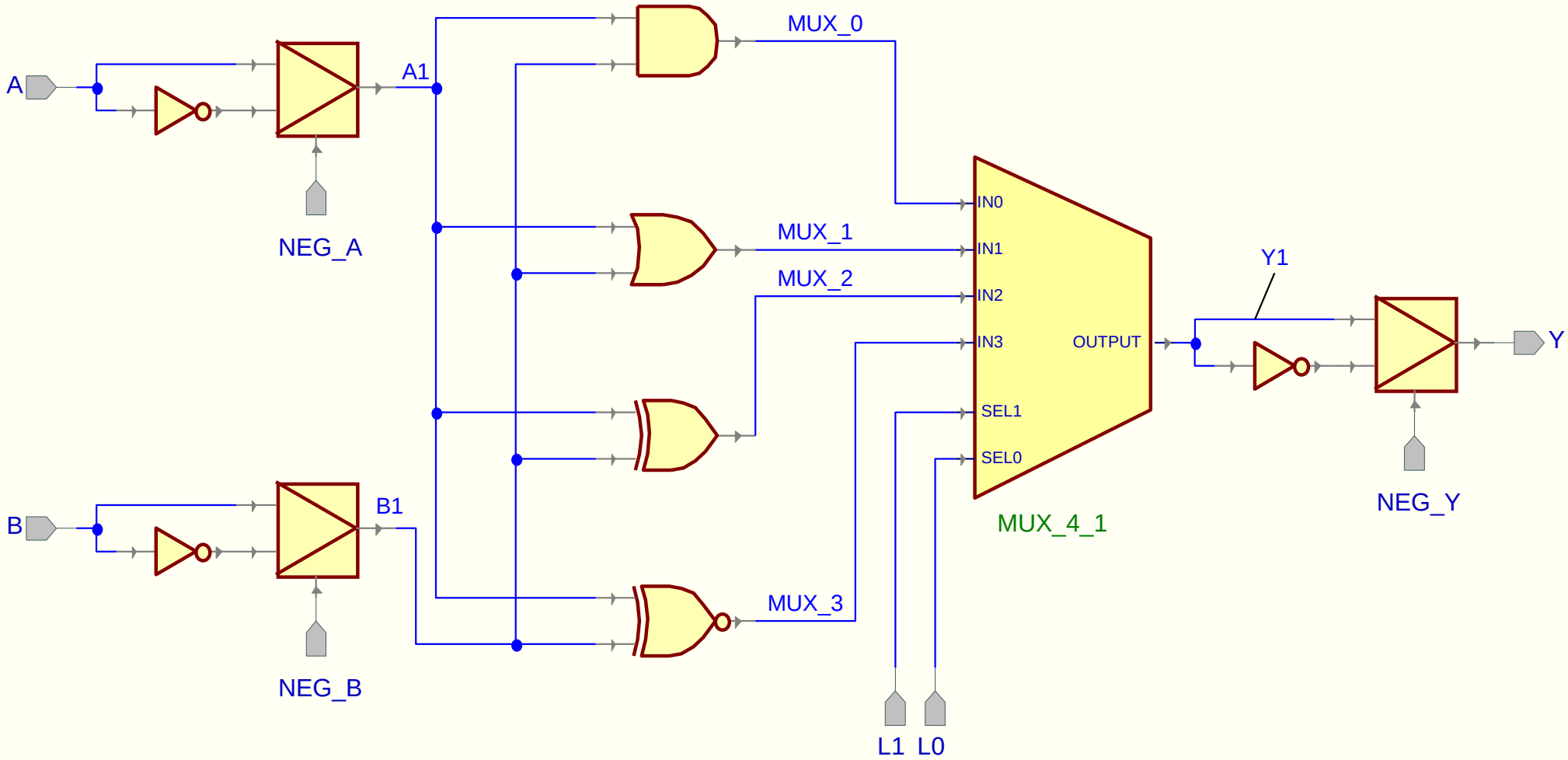


Major instructions

Concurrent statements

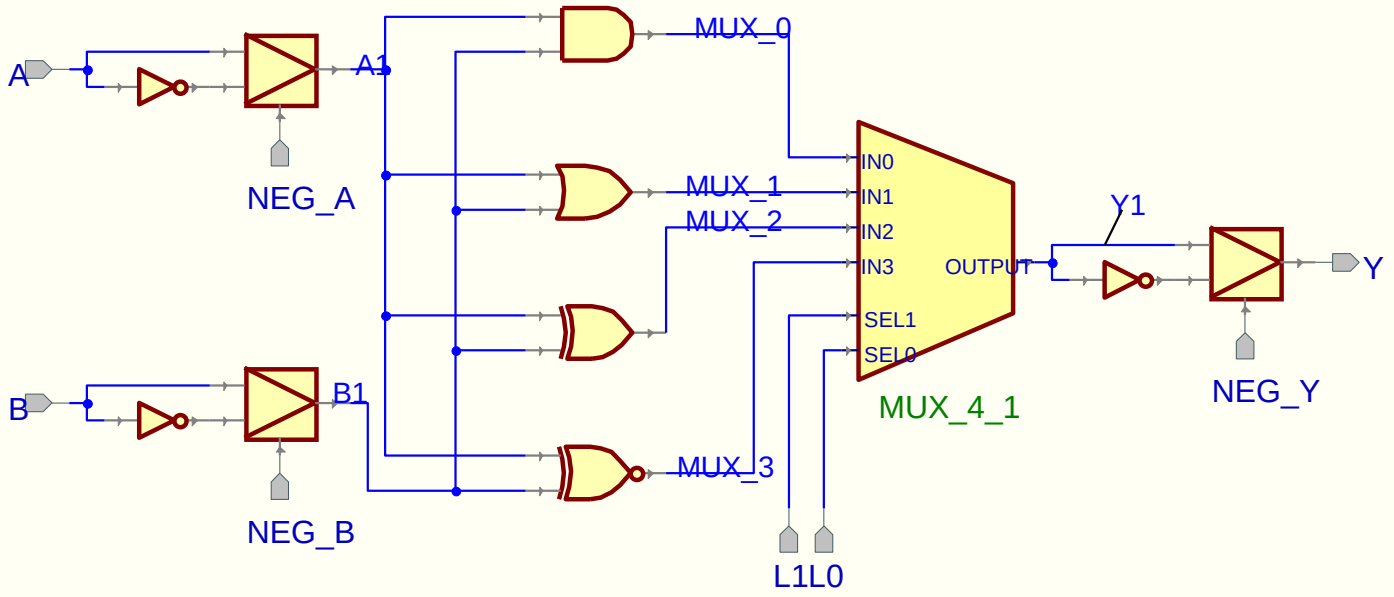
- concurrent signal assignment (⇐)
- conditional concurrent signal assignment (when-else)
- selected concurrent signal assignment (with-select-when)
- generate scheme for equations (for-generate)

MLU Block Diagram



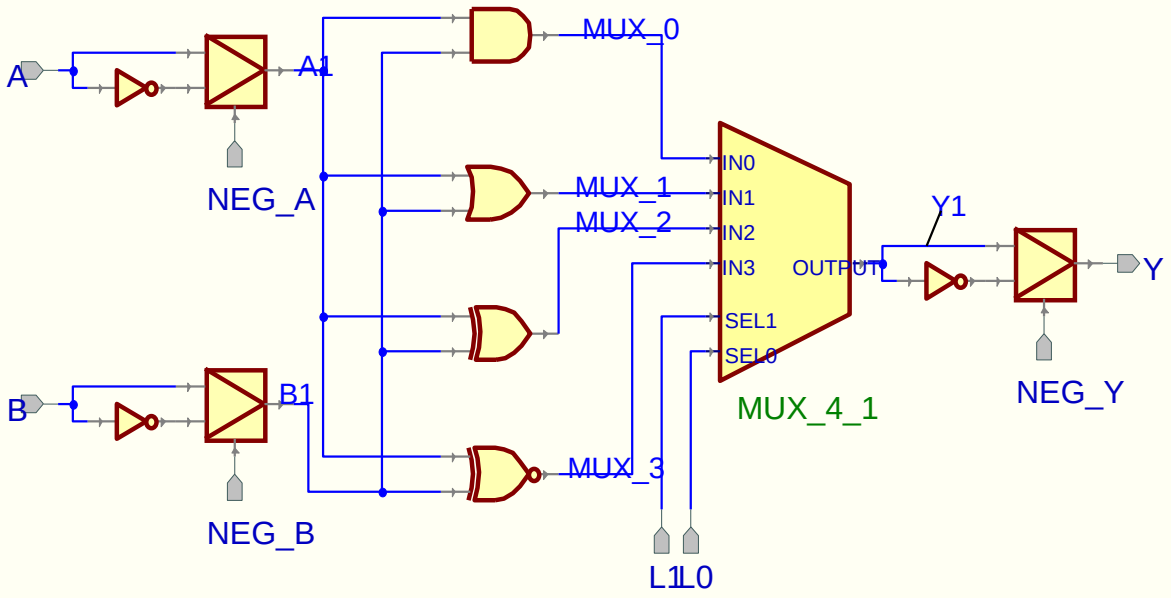
MLU Entity Declaration

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mlu IS
    PORT(
        NEG_A : IN STD_LOGIC;
        NEG_B : IN STD_LOGIC;
        NEG_Y : IN STD_LOGIC;
        A     : IN STD_LOGIC;
        B     : IN STD_LOGIC;
        L1    : IN STD_LOGIC;
        L0    : IN STD_LOGIC;
        Y     : OUT STD_LOGIC
        );
END mlu;
```



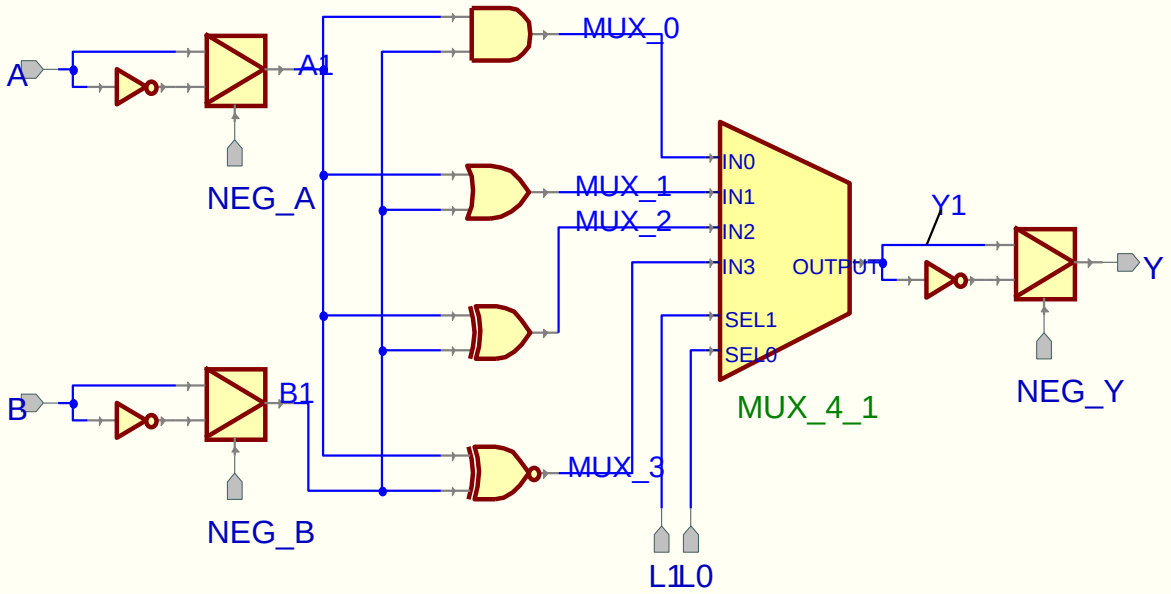
MLU Architecture: Declarations

```
ARCHITECTURE mlu_dataflow OF mlu IS

SIGNAL     A1 :   STD_LOGIC;
SIGNAL     B1 :   STD_LOGIC;
SIGNAL     Y1 : STD_LOGIC;
SIGNAL MUX_0 : STD_LOGIC;
SIGNAL MUX_1 : STD_LOGIC;
SIGNAL MUX_2 : STD_LOGIC;
SIGNAL MUX_3 : STD_LOGIC;
SIGNAL L: STD_LOGIC_VECTOR(1 DOWNTO 0);
```



MLU Architecture: Body

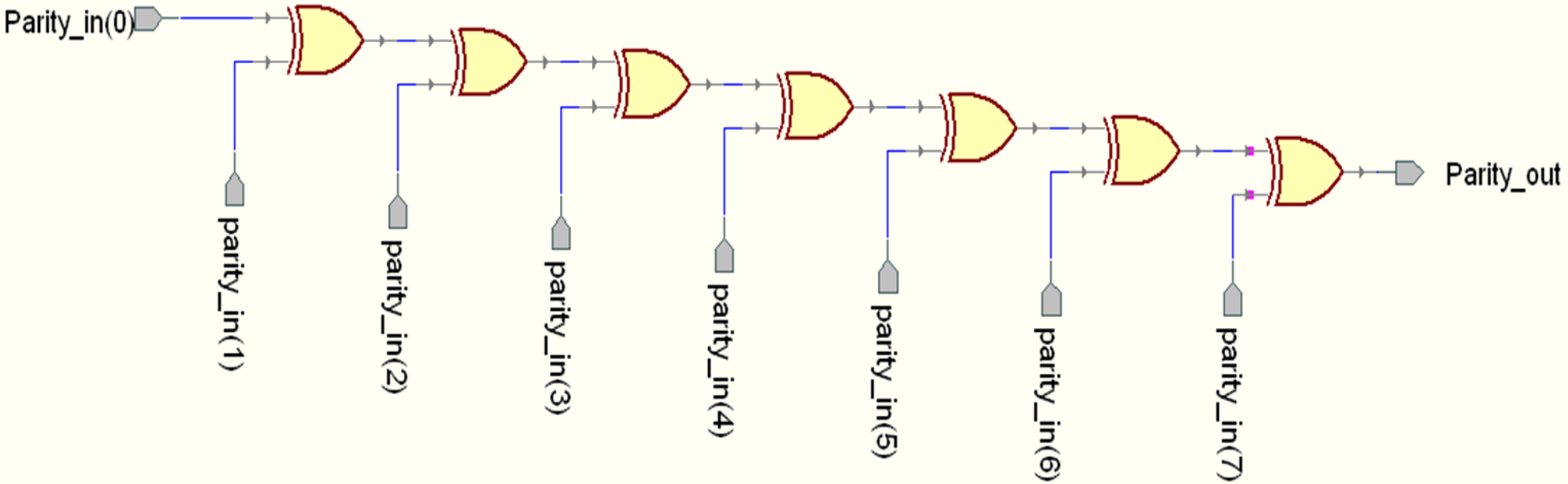
```
BEGIN
   A1<= NOT A  WHEN (NEG_A='1') ELSE A;
   B1<= NOT B  WHEN (NEG_B='1') ELSE B;
   Y <= NOT Y1 WHEN (NEG_Y='1') ELSE Y1;

   MUX_0 <= A1  AND  B1;
   MUX_1 <= A1   OR  B1;
   MUX_2 <= A1  XOR  B1;
   MUX_3 <= A1  XNOR B1;

       L <= L1 & L0;

   with (L) select
        Y1  <=  MUX_0 WHEN "00",
                MUX_1 WHEN "01",
                MUX_2 WHEN "10",
                MUX_3 WHEN OTHERS;

END mlu_dataflow;
```



FOR … GENERATE

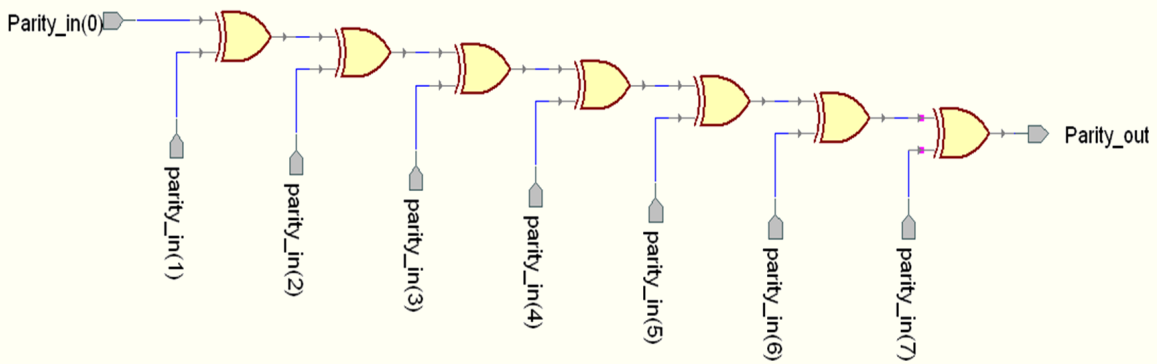
```
label: FOR identifier IN range GENERATE
           BEGIN
             {Concurrent Statements}
           END GENERATE;
```

PARITY Example

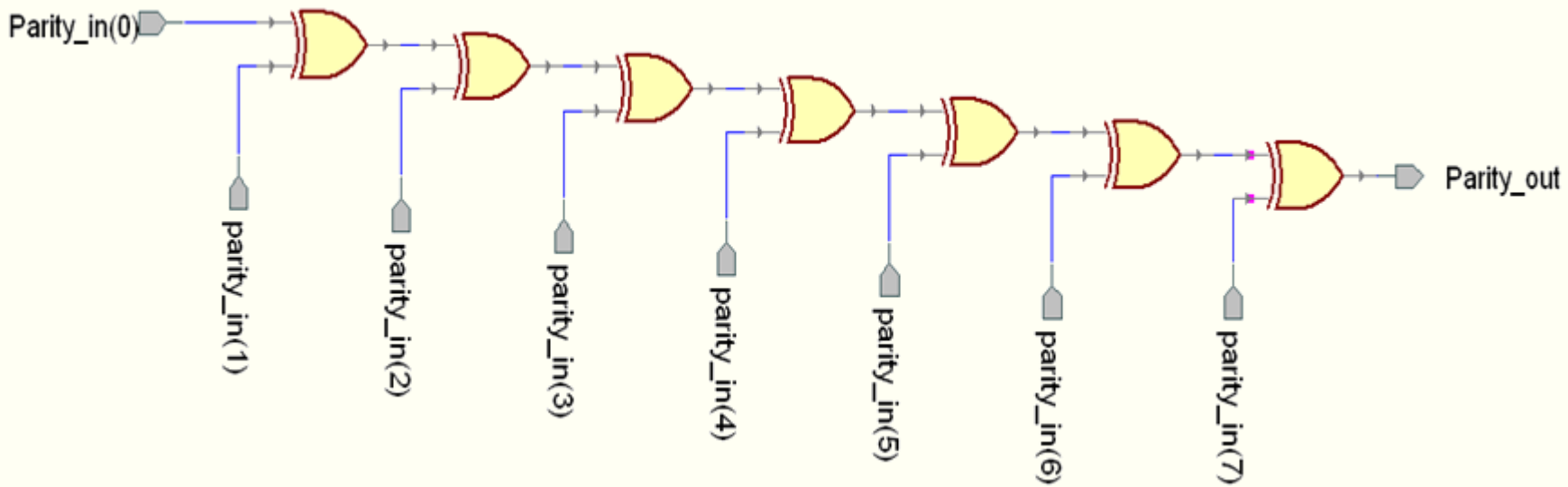


PARITY: Entity Declaration

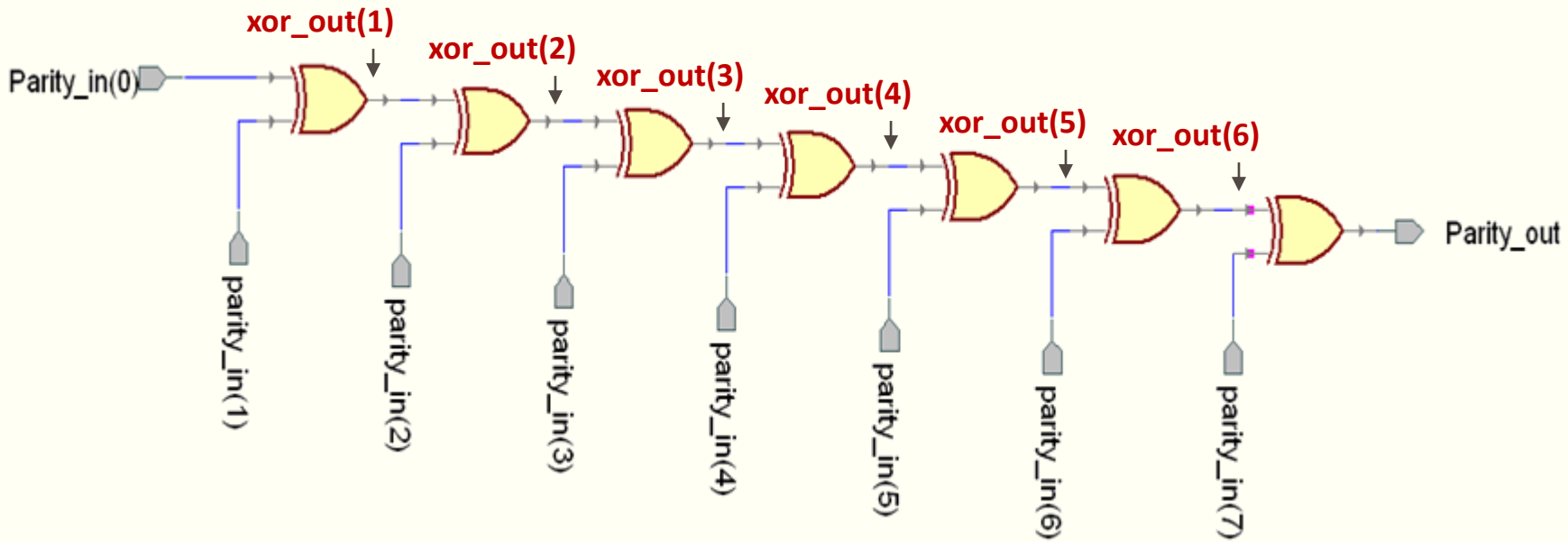
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY parity IS
    PORT(
     parity_in  : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
     parity_out : OUT STD_LOGIC
         );
END parity;
```



PARITY: Block Diagram



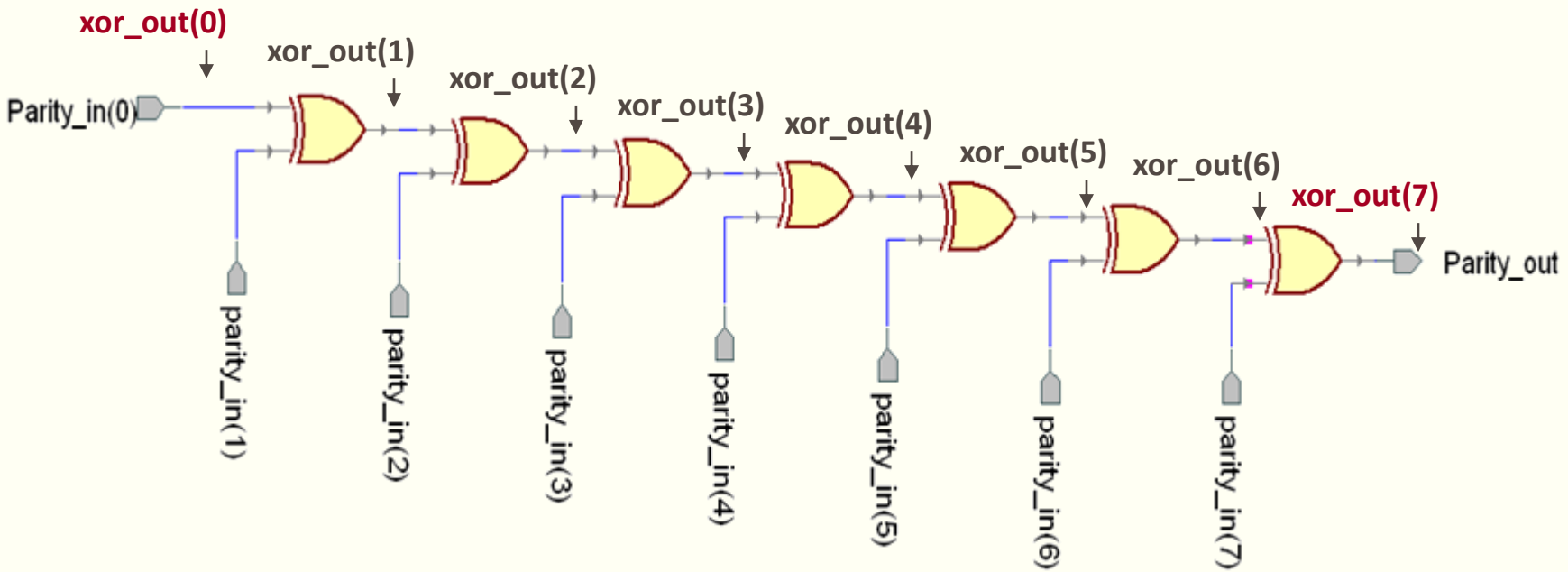
PARITY: Block Diagram



PARITY: Architecture

```
ARCHITECTURE parity_dataflow OF parity IS
SIGNAL xor_out: std_logic_vector (6 downto 1);

BEGIN
   xor_out(1) <= parity_in(0) XOR parity_in(1);
   xor_out(2) <= xor_out(1) XOR parity_in(2);
   xor_out(3) <= xor_out(2) XOR parity_in(3);
   xor_out(4) <= xor_out(3) XOR parity_in(4);
   xor_out(5) <= xor_out(4) XOR parity_in(5);
   xor_out(6) <= xor_out(5) XOR parity_in(6);
   parity_out <= xor_out(6) XOR parity_in(7);

END parity_dataflow;
```

PARITY: Block Diagram 2



PARITY: Architecture 2

```
ARCHITECTURE parity_dataflow OF parity IS

SIGNAL xor_out: STD_LOGIC_VECTOR (7 downto 0);

BEGIN
    xor_out(0) <= parity_in(0);
    xor_out(1) <= xor_out(0) XOR parity_in(1);
    xor_out(2) <= xor_out(1) XOR parity_in(2);
    xor_out(3) <= xor_out(2) XOR parity_in(3);
    xor_out(4) <= xor_out(3) XOR parity_in(4);
    xor_out(5) <= xor_out(4) XOR parity_in(5);
    xor_out(6) <= xor_out(5) XOR parity_in(6);
    xor_out(7) <= xor_out(6) XOR parity_in(7);
    parity_out <= xor_out(7);

END parity_dataflow;
```

PARITY: Architecture 3

```
ARCHITECTURE parity_dataflow OF parity IS

SIGNAL xor_out: STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN
   xor_out(0) <= parity_in(0);

   G2: FOR i IN 1 TO 7 GENERATE
   xor_out(i) <= xor_out(i-1) XOR parity_in(i);
   end generate G2;

   parity_out <= xor_out(7);

END parity_dataflow;
```

Simple Rules

- For combinational logic, use only concurrent statements:
  - concurrent signal assignment (⇐)
  - conditional concurrent signal assignment (when-else)
  - selected concurrent signal assignment (with-select-when)
  - generate scheme for equations (for-generate)

Simple Rules

- For circuits composed of
  - simple logic operations (logic gates)
  - simple arithmetic operations (addition, subtraction, multiplication)
  - shifts/rotations by a constant

use concurrent signal assignment (⇐)

Simple Rules

- For circuits composed of
  - Multiplexers
  - decoders, encoders
  - tri-state buffers

use:

  - conditional concurrent signal assignment (when-else)
  - selected concurrent signal assignment (with-select-when)

Arithmetic Operators

- Synthesizable arithmetic operations:
  - Addition, +
  - Subtraction, -
  - Comparisons, >, >=, <, <=
  - Multiplication, *
  - Division by a power of 2, /2**6 (equivalent to right shift)
  - Shifts by a constant, SHL, SHR

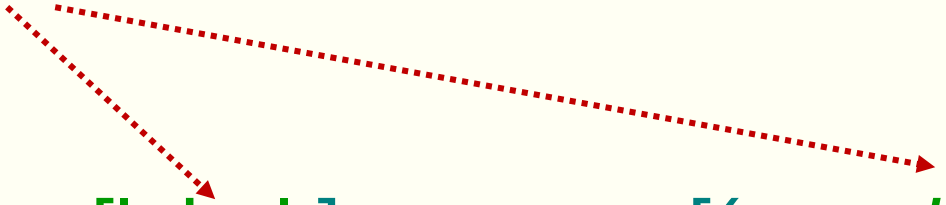
Arithmetic Operators

- The result of synthesis of an arithmetic operation is a
  - combinational circuit
  - without pipelining

The exact internal **architecture** used (and thus delay and area of the circuit) **may depend** on the **timing constraints** specified during synthesis (e.g., the requested maximum clock frequency).

Anatomy of a Process

OPTIONAL



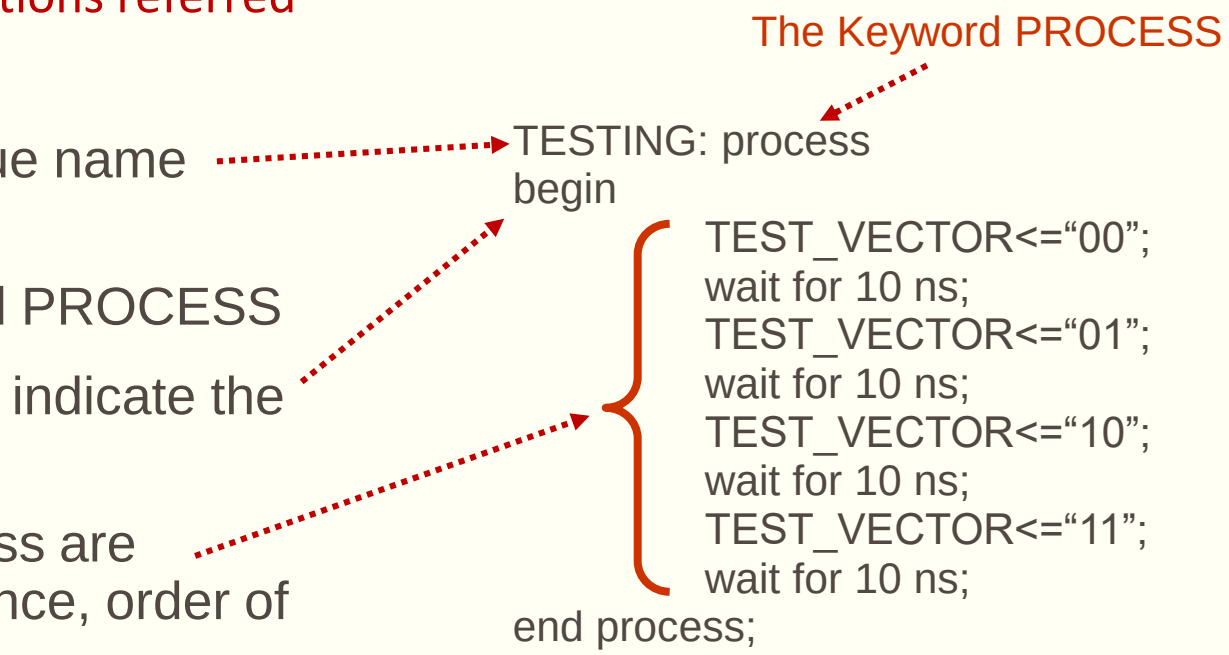
```
[label:] process [(sensitivity list)]
        [declaration part]
begin
        statement part
end process [label];
```

Statement Part

- Contains Sequential Statements to be Executed Each Time the Process Is Activated
- Analogous to Conventional Programming Languages

What is a Process

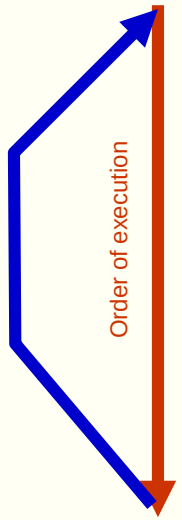
- A process is a sequence of instructions referred to as sequential statements.
- A process can be given a unique name using an optional LABEL
- This is followed by the keyword PROCESS
- The keyword BEGIN is used to indicate the start of the process
- All statements within the process are executed SEQUENTIALLY. Hence, order of statements is important.
- A process must end with the keywords END PROCESS.



Execution of Statements

- The execution of statements continues sequentially till the last statement in the process.
- After execution of the last statement, the control is again passed to the beginning of the process.

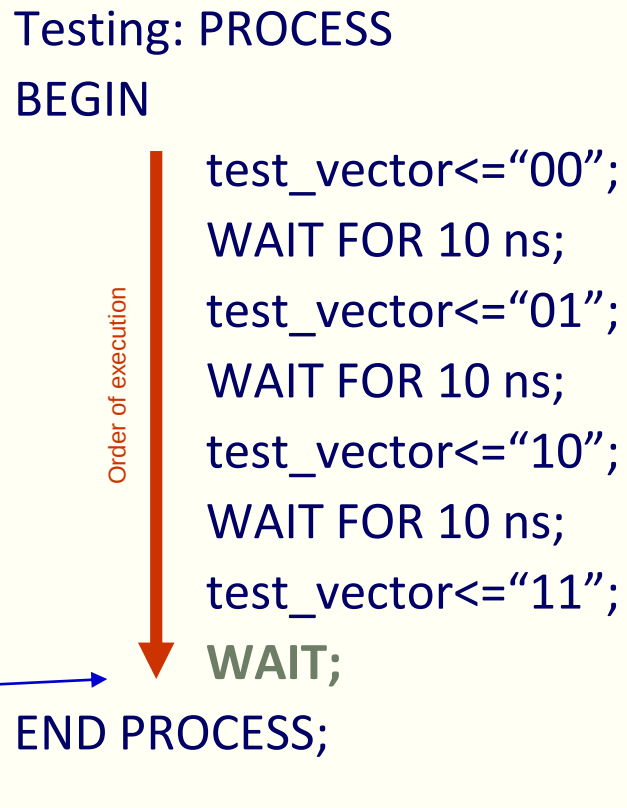
```
Testing: PROCESS
BEGIN
    test_vector<="00";
    WAIT FOR 10 ns;
    test_vector<="01";
    WAIT FOR 10 ns;
    test_vector<="10";
    WAIT FOR 10 ns;
    test_vector<="11";
    WAIT FOR 10 ns;
END PROCESS;
```



Program control is passed to the first statement after BEGIN

WAIT Statement

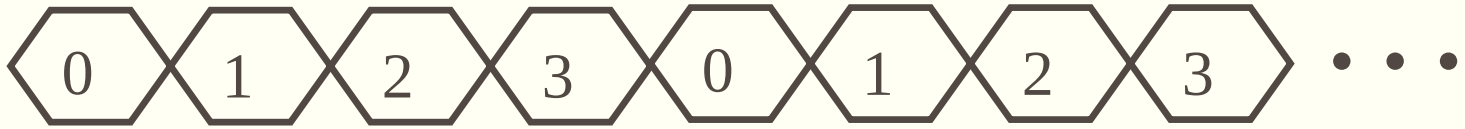
- The last statement in the PROCESS is a WAIT instead of WAIT FOR 10 ns.
- This will cause the PROCESS to suspend indefinitely when the WAIT statement is executed.
- This form of WAIT can be used in a process included in a testbench when all possible combinations of inputs have been tested or a non-periodical signal has to be generated.



Program execution stops here

WAIT FOR vs WAIT

WAIT FOR: waveform will keep repeating itself forever



WAIT : waveform will keep its state after the last wait instruction.

Sensitivity List

- List of signals to which the process is sensitive.
- Whenever there is an event on any of the signals in the sensitivity list, the process fires.
- Every time the process fires, it will run in its entirety.
- **WAIT statements are NOT ALLOWED in a process with SENSITIVITY LIST.**

```
label: process (sensitivity list)
       declaration part
begin
   statement part
end process;
```

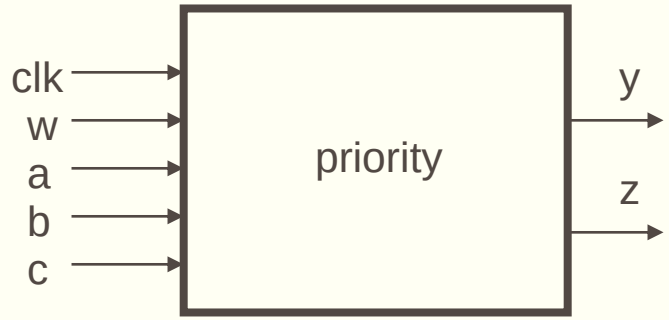
Process Suitability

- Processes Describe Sequential Behavior
- Processes in VHDL Are Very Powerful Statements
  - Allow to define an arbitrary behavior that may be difficult to represent by a real circuit
  - Not every process can be synthesized

- Use Processes with Caution in the Code to Be Synthesized
- Use Processes Freely in Testbenches

Component Equivalent of a Process

```
priority: PROCESS (clk)
BEGIN
  IF w(3) = '1' THEN
      y <= "11" ;
  ELSIF w(2) = '1' THEN
      y <= "10" ;
  ELSIF w(1) = c THEN
      y <= a and b;
  ELSE
      z <= "00" ;
  END IF ;
END PROCESS ;
```



- All signals which appear on the left of signal assignment statement (<=) are outputs e.g. *y, z*
- All signals which appear on the right of signal assignment statement (<=) or in logic expressions are inputs e.g. *w, a, b, c*
- All signals which appear in the sensitivity list are inputs e.g. *clk*
- Note that not all inputs need to be included in the sensitivity list

IF Statement - Syntax

If Statement

```
if boolean expression then
      statements
elsif boolean expression then
      statements
else boolean expression then
      statements
end if;
```

else and elsif are optional

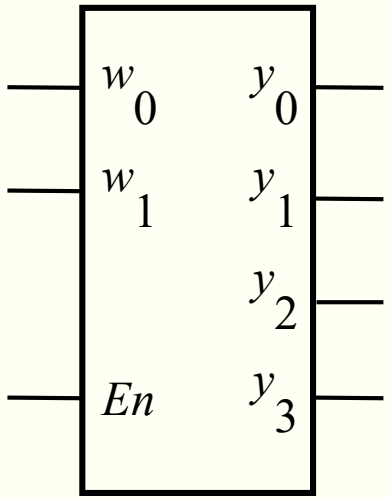
IF Statement: Example

```
SELECTOR: process
begin
    WAIT UNTIL Clock'EVENT AND Clock = '1' ;
      IF Sel = "00" THEN
        f  <= x1;
    ELSIF Sel = "10" THEN
        f  <= x2;
    ELSE
        f  <= x3;
    END IF;
end process;
```

2-to-4 Decoder

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

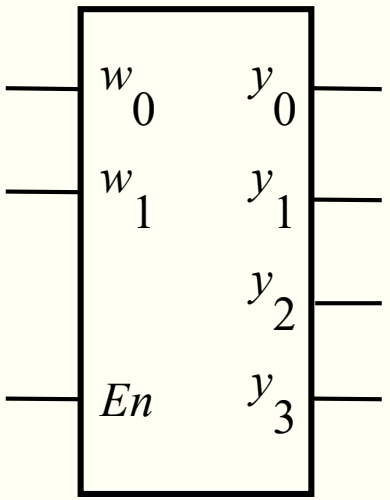
(a) Truth table



(b) Graphical symbol

Dataflow Description (1/2)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END dec2to4 ;

ARCHITECTURE dataflow OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y <= "0001" WHEN "100",
             "0010" WHEN "101",
             "0100" WHEN "110",
             "1000" WHEN "111",
             "0000" WHEN OTHERS ;
END dataflow ;
```



(b) Graphical symbol

Dataflow Description (2/2)

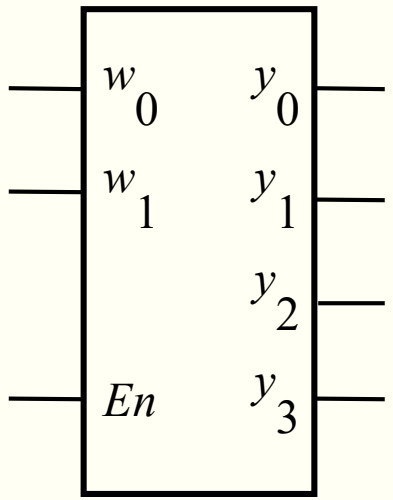
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT ( w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN    STD_LOGIC ;
           y   : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END dec2to4 ;

ARCHITECTURE dataflow OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
           y <= "0001" WHEN "100",
                "0010" WHEN "101",
                "0100" WHEN "110",
                "1000" WHEN "111",
                "0000" WHEN OTHERS ;
END dataflow ;
```

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table

Behavioral Description (1/2)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY dec2to4 IS
    PORT ( w    : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En   : IN    STD_LOGIC ;
           y    : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;
ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
    PROCESS ( w, En )
    BEGIN
        IF En = '1' THEN
            CASE w IS
                WHEN "00" =>    y <= "1000" ;
                WHEN "01" =>    y <= "0100" ;
                WHEN "10" =>    y <= "0010" ;
                WHEN OTHERS =>  y <= "0001" ;
            END CASE ;
        ELSE
            y <= "0000" ;
        END IF ;
    END PROCESS ;
END Behavior ;
```



(b) Graphical symbol

Behavioral Description (2/2)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY dec2to4 IS
     PORT ( w    : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
            En   : IN    STD_LOGIC ;
            y    : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;
ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
     PROCESS ( w, En )
     BEGIN
          IF En = '1' THEN
             CASE w IS
                  WHEN "00" =>       y <= "1000" ;
                  WHEN "01" =>       y <= "0100" ;
                  WHEN "10" =>       y <= "0010" ;
                  WHEN OTHERS =>     y <= "0001" ;
             END CASE ;
          ELSE
             y <= "0000" ;
          END IF ;
     END PROCESS ;
END Behavior ;
```

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table

7 Segment Display

```
LIBRARY ieee ; USE ieee.std_logic_1164.all ;
ENTITY seg7 IS
     PORT (bcd  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           leds : OUT  STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;
ARCHITECTURE Behavior OF seg7 IS
BEGIN
     PROCESS ( bcd )
     BEGIN
          CASE bcd IS                    --    abcdefg
            WHEN "0000"      => leds    <=   "1111110" ;
            WHEN "0001"      => leds    <=   "0110000" ;
            WHEN "0010"      => leds    <=   "1101101" ;
            WHEN "0011"      => leds    <=   "1111001" ;
            WHEN "0100"      => leds    <=   "0110011" ;
            WHEN "0101"      => leds    <=   "1011011" ;
            WHEN "0110"      => leds    <=   "1011111" ;
            WHEN "0111"      => leds    <=   "1110000" ;
            WHEN "1000"      => leds    <=    "1111111" ;
            WHEN "1001"      => leds    <=    "1110011" ;
            WHEN OTHERS      => leds    <=    "-------" ;
          END CASE ;
     END PROCESS ;
END Behavior ;
```

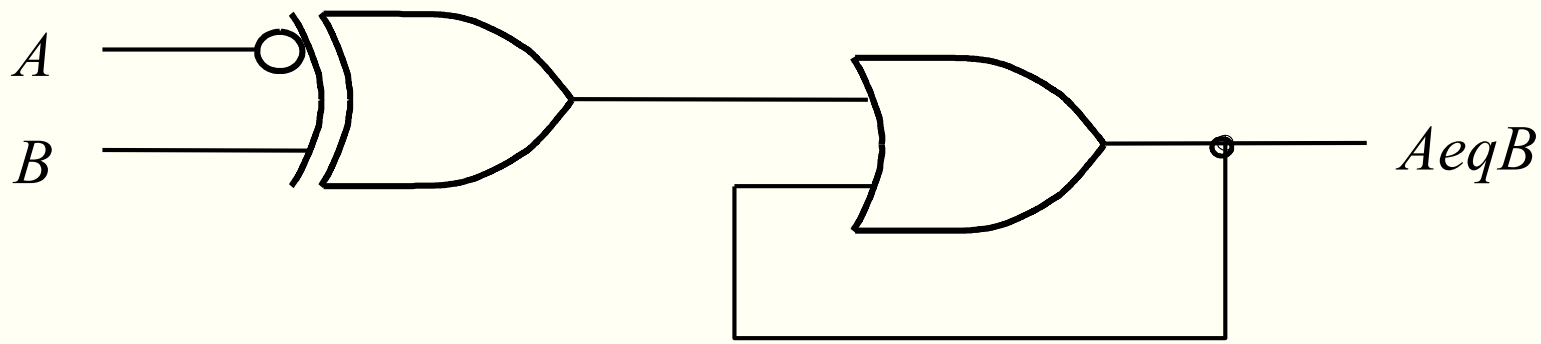
Comparator

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY compare1 IS
    PORT ( A, B   : IN   STD_LOGIC ;
           AeqB   : OUT  STD_LOGIC ) ;
END compare1 ;

ARCHITECTURE Behavior OF compare1 IS
BEGIN
    PROCESS ( A, B )
    BEGIN
        AeqB <= '0' ;
        IF A = B THEN
            AeqB <= '1' ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Latch Inference

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
    PORT ( A, B : IN   STD_LOGIC ;
           AeqB  : OUT  STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
    PROCESS ( A, B )
    BEGIN
       IF A = B THEN
          AeqB <= '1' ;
       END IF ;
    END PROCESS ;
END Behavior ;
```

Latch Inference



Combinational Circuits with Processes

- Rules that need to be followed:
  - All inputs to the combinational circuit should be included in the sensitivity list
  - No other signals should be included in the sensitivity list
  - None of the statements within the process should be sensitive to rising or falling edges
  - All possible cases need to be covered in the internal **IF** and **CASE** statements in order to avoid implied latches

Covering all Combinations with IF

Using ELSE

```
IF A = B THEN
     AeqB <= '1' ;
  ELSE
     AeqB <= '0' ;
```

Using default values

```
AeqB <= '0' ;
IF A = B THEN
     AeqB <= '1' ;
```

Covering all Combinations with CASE

Using WHEN OTHERS

```
CASE y IS
      WHEN S1 => Z <= "10";
      WHEN S2 => Z <= "01";
      WHEN OTHERS => Z <= "00";
END CASE;
```

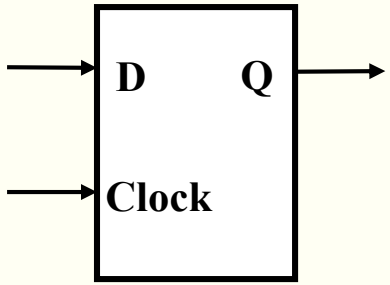
```
CASE y IS
      WHEN S1 => Z <= "10";
      WHEN S2 => Z <= "01";
      WHEN S3 => Z <= "00";
      WHEN OTHERS => Z <= „--";
END CASE;
```

Using default values

```
Z <= "00";
CASE y IS
      WHEN S1 => Z <= "10";
      WHEN S2 => Z <= "10";
END CASE;
```

Sequential Logic: D Latch

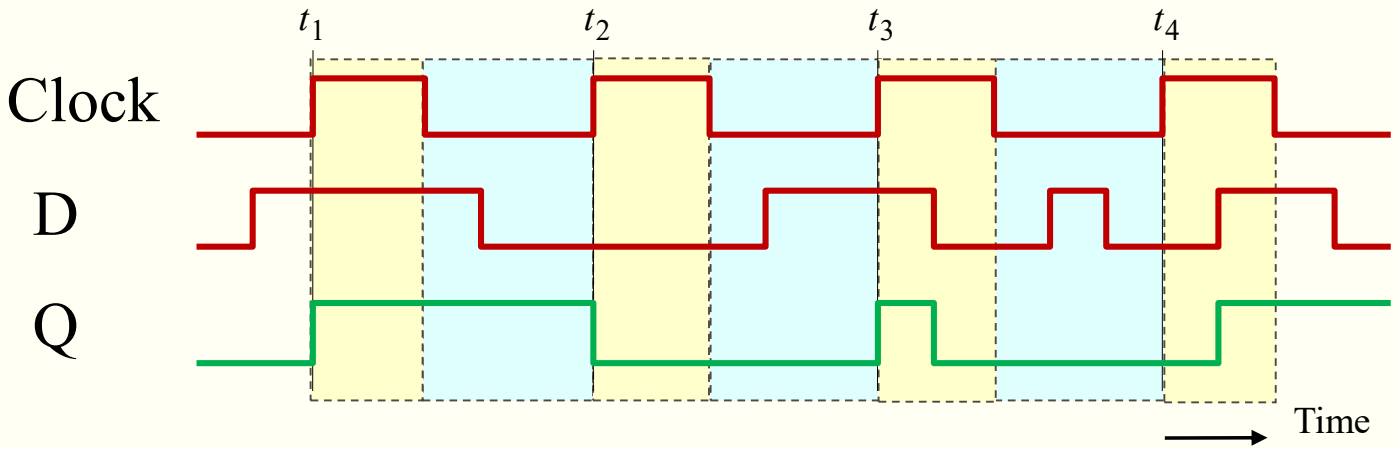
Graphical symbol



Truth table

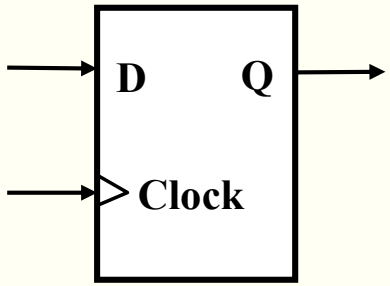
Clock	D	Q(*t*+1)
0	–	Q(*t*)
1	0	0
1	1	1

Timing diagram



Sequential Logic: D Flip-flop

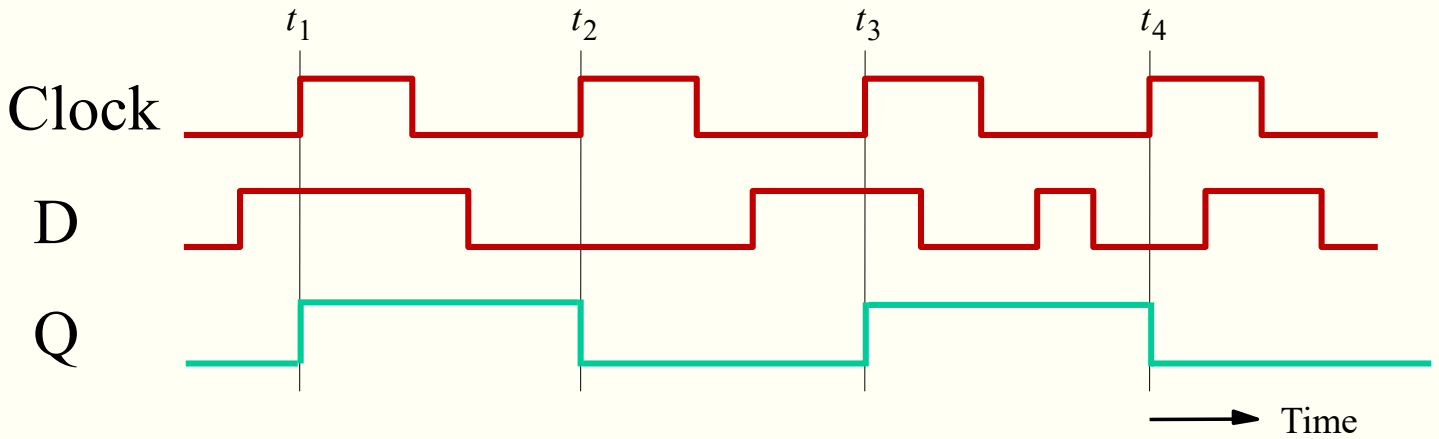
Graphical symbol



Truth table

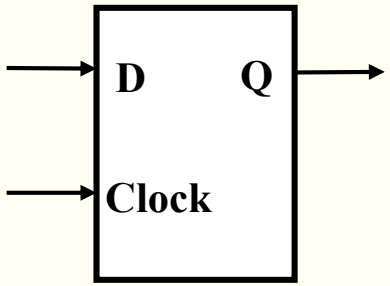
Clk	D	Q(t+1)
↑	0	0
↑	1	1
0	–	Q(t)
1	–	Q(t)

Timing diagram



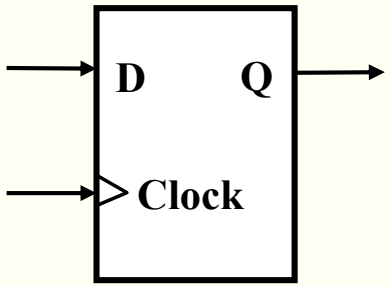
VHDL Code: D Latch

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
    PORT ( D, Clock : IN  STD_LOGIC ;
           Q        : OUT STD_LOGIC) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
    PROCESS ( D, Clock )
    BEGIN
        IF Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior;
```



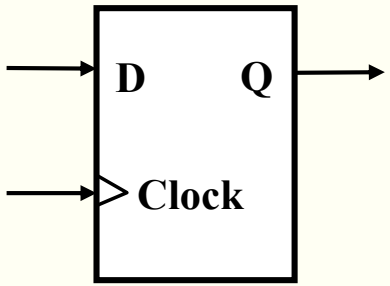
VHDL Code: D Flip-flop

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock : IN  STD_LOGIC ;
           Q        : OUT STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior_1 OF flipflop IS
BEGIN
    PROCESS ( Clock )
    BEGIN
       IF Clock'EVENT AND Clock = '1' THEN
           Q <= D ;
       END IF ;
    END PROCESS ;
END Behavior_1 ;
```



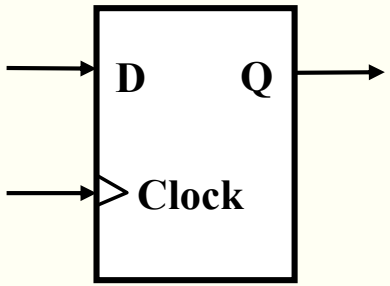
VHDL Code: D Flip-flop

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
   PORT ( D, Clock : IN  STD_LOGIC ;
          Q        : OUT STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior_1 OF flipflop IS
BEGIN
    PROCESS ( Clock )
    BEGIN
       IF rising_edge(Clock) THEN
           Q <= D ;
       END IF ;
    END PROCESS ;
END Behavior_1 ;
```



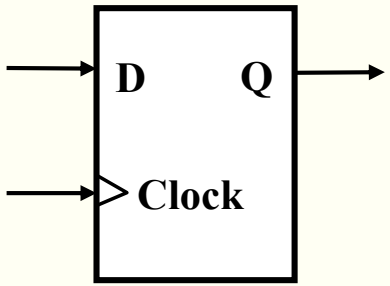
VHDL Code: D Flip-flop

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
   PORT ( D, Clock : IN  STD_LOGIC ;
          Q        : OUT STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior_2 OF flipflop IS
BEGIN
    PROCESS
    BEGIN
       WAIT UNTIL Clock'EVENT AND Clock = '1' ;
       Q <= D ;
    END PROCESS ;
END Behavior_2 ;
```



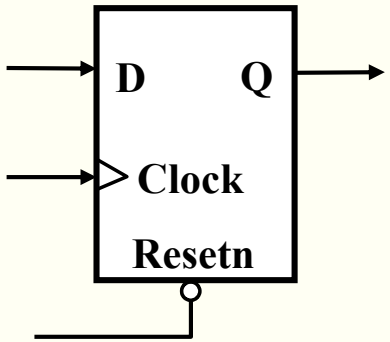
VHDL Code: D Flip-flop

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
   PORT ( D, Clock : IN  STD_LOGIC ;
          Q        : OUT STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior_2 OF flipflop IS
BEGIN
    PROCESS
    BEGIN
       WAIT UNTIL rising_edge(Clock) ;
       Q <= D ;
    END PROCESS ;
END Behavior_2 ;
```



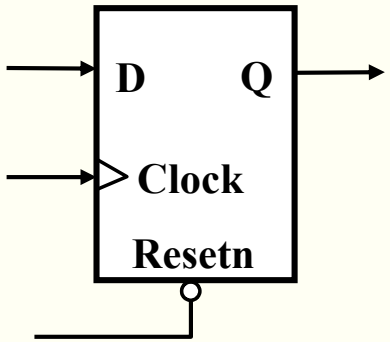
VHDL Code: D Flip-flop Async Reset

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock : IN  STD_LOGIC ;
           Q                : OUT STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```



VHDL Code: D Flip-flop Sync Reset

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY flipflop IS
    PORT ( D, Resetn, Clock : IN  STD_LOGIC ;
           Q                : OUT STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSE
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```



Variables: Example

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Numbits IS
   PORT ( X     : IN   STD_LOGIC_VECTOR(1 TO 3) ;
          Count : OUT  INTEGER RANGE 0 TO 3) ;
END Numbits ;
```

Variables: Example

```
ARCHITECTURE Behavior OF Numbits IS

BEGIN

   PROCESS(X) - count the number of bits in X equal to 1
    VARIABLE Tmp: INTEGER;
   BEGIN
       Tmp := 0;
       FOR i IN 1 TO 3 LOOP
          IF X(i) = '1' THEN
             Tmp := Tmp + 1;
           END IF;
       END LOOP;
   Count <= Tmp;
   END PROCESS;

END Behavior ;
```

Variables: Features

- Can only be declared within processes and subprograms (functions & procedures)
- Initial value can be explicitly specified in the declaration
- When assigned take an assigned value immediately
- Variable assignments represent the desired behavior, not the structure of the circuit
- Should be avoided, or at least used with caution in a synthesizable code

Variable vs Signal – Variable Example

```
ARCHITECTURE Behavior OF Numbits IS

BEGIN

   PROCESS(X) - count the number of bits in X equal to 1
   VARIABLE Tmp: INTEGER;
   BEGIN
        Tmp := 0;
        FOR i IN 1 TO 3 LOOP
             IF X(i) = '1' THEN
                  Tmp := Tmp + 1;
             END IF;
        END LOOP;
   Count <= Tmp;
   END PROCESS;

END Behavior ;
```

Variable vs Signal: NAND Gate

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY NANDn IS
   GENERIC (n: INTEGER := 8)
   PORT (   X: IN STD_LOGIC_VECTOR(1 TO n);
            Y: OUT STD_LOGIC);
END NANDn;
```

Variable vs Signal: NAND Gate - Var

```
ARCHITECTURE behavioral1 OF NANDn IS
BEGIN

  PROCESS (X)
  VARIABLE Tmp: STD_LOGIC;
  BEGIN
    Tmp := X(1);

    AND_bits: FOR i IN 2 TO n LOOP
        Tmp := Tmp AND X( i ) ;
    END LOOP AND_bits ;

    Y <= NOT Tmp ;

  END PROCESS;

END behavioral1 ;
```

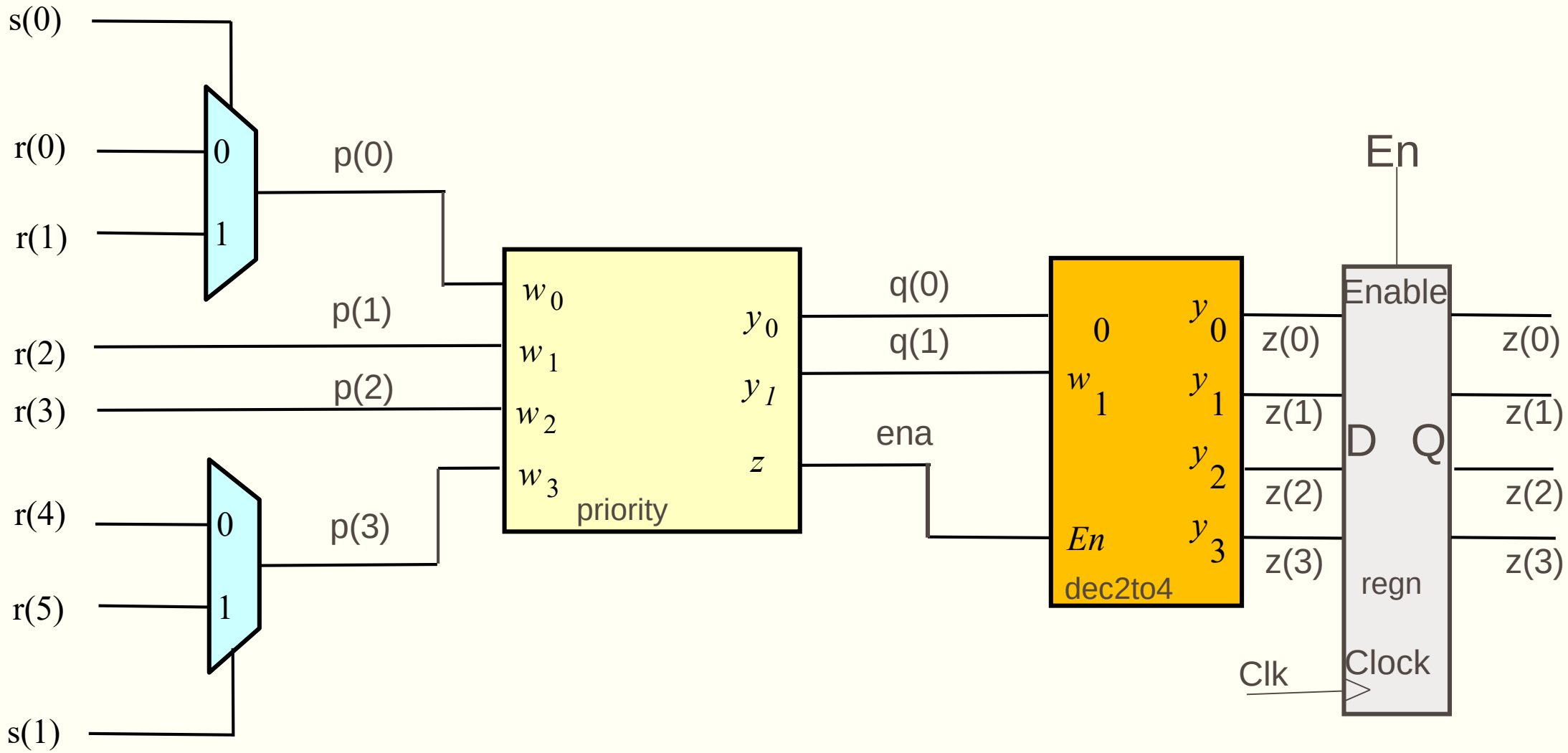
Variable vs Signal: NAND Gate - Signal

```
ARCHITECTURE dataflow1 OF NANDn IS

    SIGNAL Tmp: STD_LOGIC_VECTOR(1 TO n);

BEGIN
   Tmp(1) <= X(1);

    AND_bits: FOR i IN 2 TO n GENERATE
        Tmp(i) <= Tmp(i-1) AND X( i ) ;
    END LOOP AND_bits ;

    Y <= NOT Tmp(n) ;

END dataflow1 ;
```

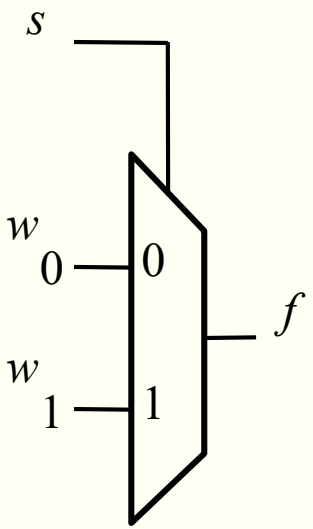
Structural VHDL

- Major instructions:
  - component instantiation (port map)
  - generate scheme for component instantiations (for-generate)
  - component instantiation with generic (generic map, port map)

Structural VHDL: Example



2-to-1 Multiplexer



(a) Graphical symbol

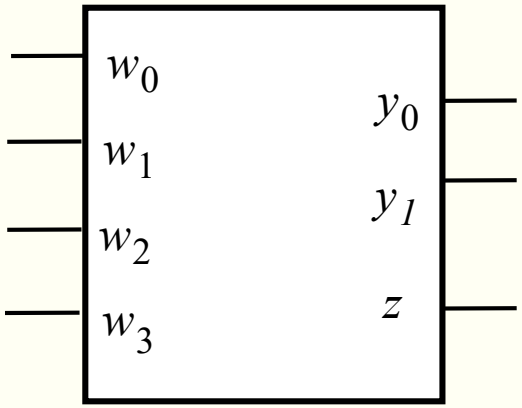
s	f
0	w_0
1	w_1

(b) Truth table

VHDL Code for 2-to-1 Multiplexer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
PORT (   w0, w1, s : IN  STD_LOGIC ;
                   f : OUT STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE dataflow OF mux2to1 IS
BEGIN
   f <= w0 WHEN s = '0' ELSE w1 ;
END dataflow ;
```

Priority Encoder



w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

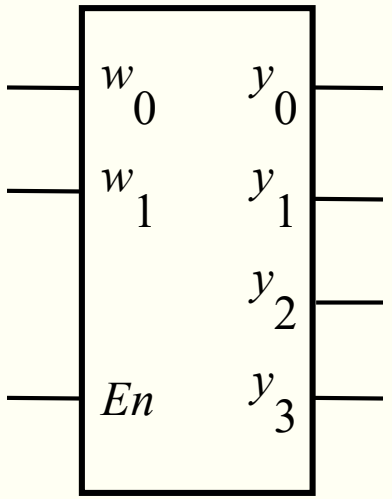
VHDL Code for Priority Encoder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT (w   : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
          y   : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
          z   : OUT STD_LOGIC ) ;
END priority ;

ARCHITECTURE dataflow OF priority IS
BEGIN
    y <= "11" WHEN w(3) = '1' ELSE
         "10" WHEN w(2) = '1' ELSE
         "01" WHEN w(1) = '1' ELSE
         "00" ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END dataflow ;
```

2-to-4 Decoder

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table



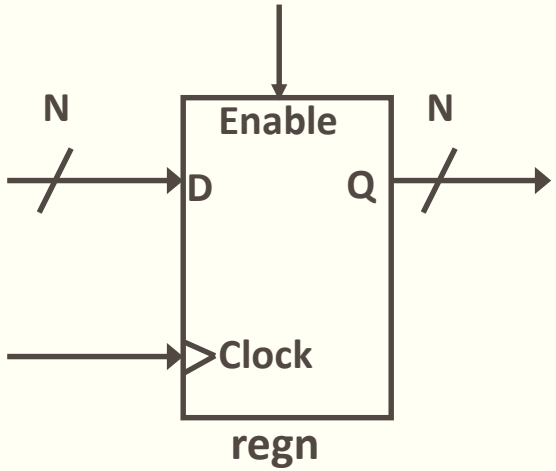
(b) Graphical symbol

VHDL Code for 2-to-4 Decoder

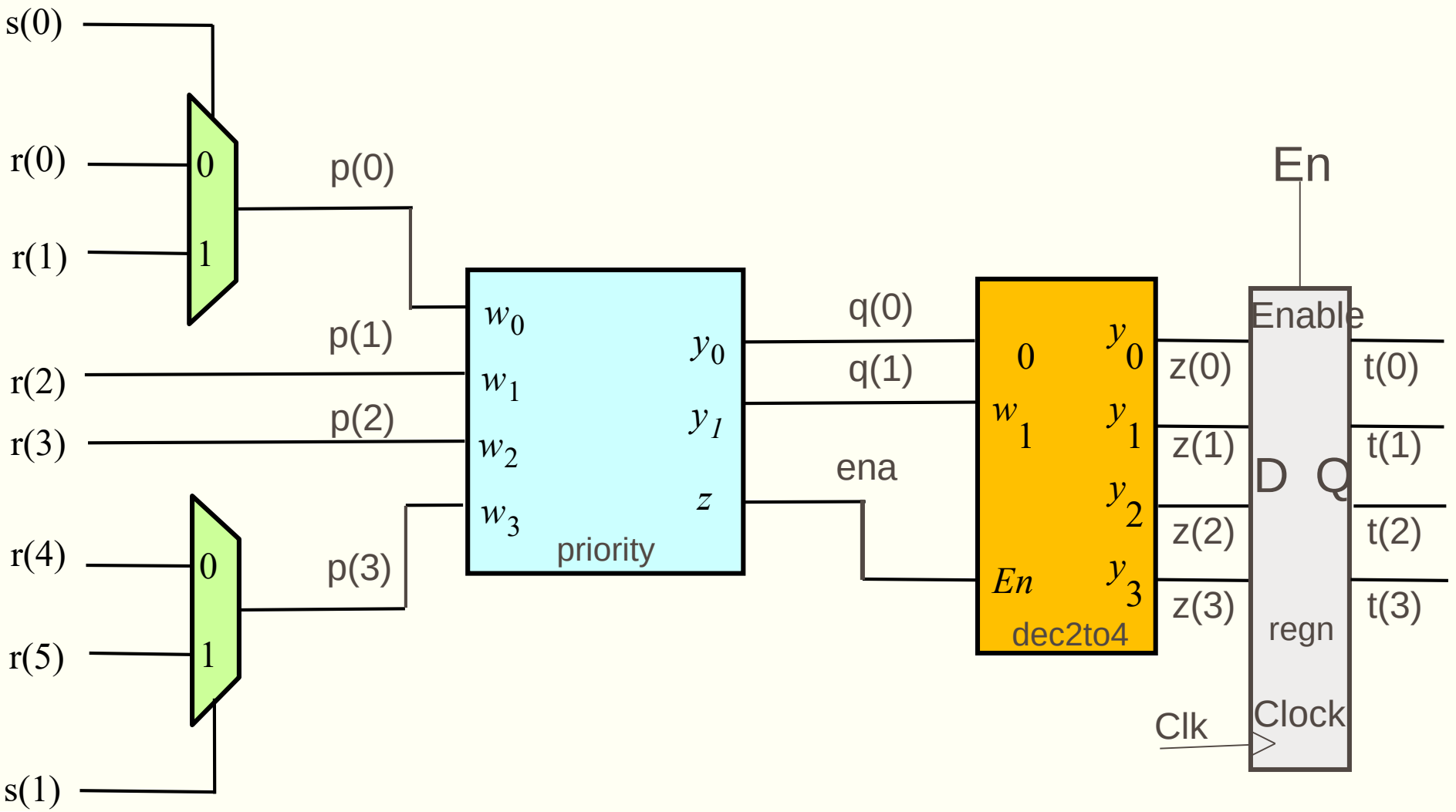
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
    PORT (w : IN        STD_LOGIC_VECTOR(1 DOWNTO 0) ;
          En : IN       STD_LOGIC ;
          y : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END dec2to4 ;

ARCHITECTURE dataflow OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
            y <= "0001" WHEN "100",
                 "0010" WHEN "101",
                 "0100" WHEN "110",
                 "1000" WHEN "111",
                 "0000" WHEN OTHERS ;
END dataflow ;
```

N-bit Register with Enable

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY regn IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT (   D: IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
Enable, Clock : IN  STD_LOGIC ;
            Q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS (Clock)
    BEGIN
        IF (Clock'EVENT AND Clock = '1' ) THEN
            IF Enable = '1' THEN
                Q <= D ;
            END IF ;
        END IF;
    END PROCESS ;
END Behavior ;
```



Circuit Built with Components



Structural Description

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY priority_resolver IS
   PORT (r   : IN  STD_LOGIC_VECTOR(5 DOWNTO 0) ;
         s   : IN  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
         clk : IN  STD_LOGIC;
         en  : IN  STD_LOGIC;
         t   : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END priority_resolver;

ARCHITECTURE structural OF priority_resolver IS

SIGNAL   p: STD_LOGIC_VECTOR (3 DOWNTO 0) ;
SIGNAL   q: STD_LOGIC_VECTOR (1  DOWNTO 0) ;
SIGNAL   z: STD_LOGIC_VECTOR (3 DOWNTO 0) ;
SIGNAL ena: STD_LOGIC ;
```

Structural Description

```
COMPONENT mux2to1
  PORT (w0, w1, s : IN  STD_LOGIC ;
                f: OUT STD_LOGIC ) ;
END COMPONENT ;

COMPONENT priority
  PORT (w   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
        y   : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
        z   : OUT   STD_LOGIC ) ;
END COMPONENT ;

COMPONENT dec2to4
  PORT (w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
        En  : IN    STD_LOGIC ;
        y   : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END COMPONENT ;
```

Structural Description

```
COMPONENT regn
    GENERIC ( N : INTEGER := 8 ) ;
    PORT    ( D : IN    STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
              Enable, Clock : IN STD_LOGIC ;
              Q : OUT   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END COMPONENT ;
```

Structural Description

```
BEGIN

  u1: mux2to1 PORT MAP (w0 => r(0) ,
                        w1 => r(1),
                        s => s(0),
                        f => p(0));
  p(1) <= r(2);
  p(1) <= r(3);

  u2: mux2to1 PORT MAP (w0 => r(4) ,
                        w1 => r(5),
                        s => s(1),
                        f => p(3));

  u3: priority PORT MAP (w => p,
                         y => q,
                         z => ena);

  u4: dec2to4 PORT MAP (w => q,
                        En => ena,
                        y => z);
```

Structural Description

```
u5: regn GENERIC MAP (N => 4)
          PORT MAP (D => z ,
                    Enable => En ,
                    Clock => Clk,
                    Q => t );
 END structural;
```

Structural Description – Pos. Assoc.

```
BEGIN
  u1: mux2to1 PORT MAP (r(0), r(1), s(0), p(0));

  p(1) <= r(2);

  p(1) <= r(3);

  u2: mux2to1 PORT MAP (r(4) , r(5), s(1), p(3));
  u3: priority PORT MAP (p, q, ena);
  u4: dec2to4 PORT MAP (q, ena, z);
  u5: regn GENERIC MAP(4) PORT MAP (z, En, Clk, t);

END structural;
```

Configuration Declaration

```
CONFIGURATION SimpleCfg OF priority_resolver IS

  FOR structural

    FOR ALL: mux2to1
       USE ENTITY work.mux2to1(dataflow);
    END FOR;

    FOR u3: priority
        USE ENTITY work.priority(dataflow);
    END FOR;

    FOR u4: dec2to4
       USE ENTITY work.dec2to4(dataflow);
    END FOR;

  END FOR;

END SimpleCfg;
```

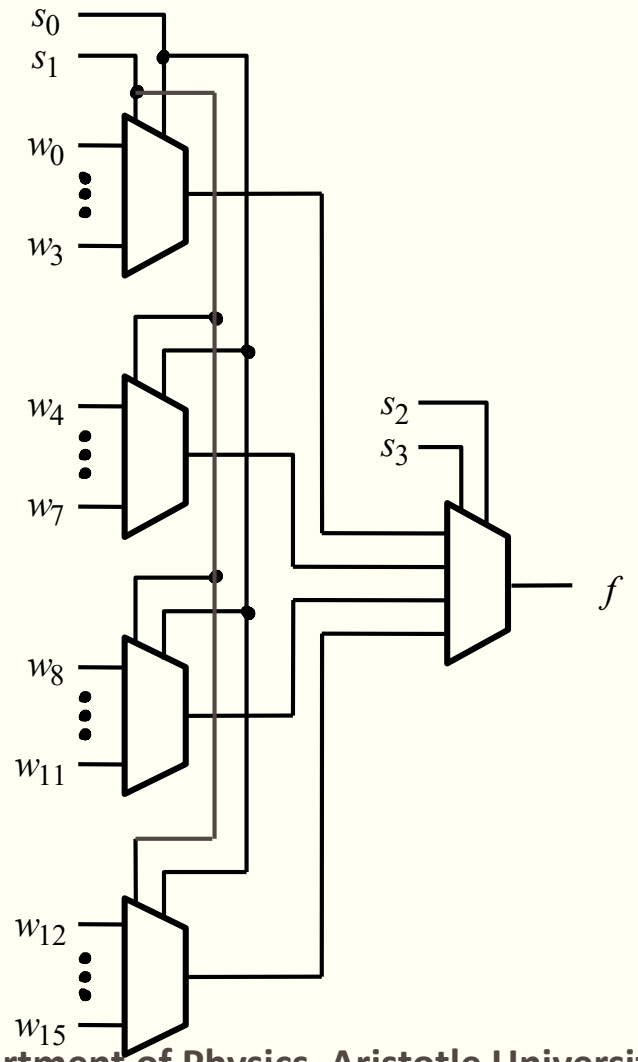
Configuration Specification

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.GatesPkg.all;

ENTITY priority_resolver IS
   PORT (r : IN  STD_LOGIC_VECTOR(5 DOWNTO 0) ;
         s : IN  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
         z : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END priority_resolver;

ARCHITECTURE structural OF priority_resolver IS

SIGNAL   p : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
SIGNAL   q : STD_LOGIC_VECTOR (1  DOWNTO 0) ;
SIGNAL  ena: STD_LOGIC ;

FOR ALL: mux2to1    USE ENTITY work.mux2to1(dataflow);
FOR u3:  priority   USE ENTITY work.priority(dataflow);
FOR u4:  dec2to4    USE ENTITY work.dec2to4(dataflow);
```

Structural VHDL

- Major instructions:
  - component instantiation (port map)
  - component instantiation with generic (generic map, port map)
  - generate scheme for component instantiations (for-generate)

16-to-1 Multiplexer



VHDL Code for 4-to-1 Multiplexer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
   PORT (w0, w1, w2, w3 : IN   STD_LOGIC ;
         s              : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
         f              : OUT STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE Dataflow OF mux4to1 IS
BEGIN
   WITH s SELECT
        f <= w0 WHEN "00",
             w1 WHEN "01",
             w2 WHEN "10",
             w3 WHEN OTHERS ;
END Dataflow ;
```

Straightforward Code for 16-to-1 MUX

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Example1 IS
   PORT (  w    : IN   STD_LOGIC_VECTOR(0 TO 15) ;
           s    : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           f    : OUT  STD_LOGIC ) ;
END Example1 ;
```

Straightforward Code for 16-to-1 MUX

```
ARCHITECTURE Structure OF Example1 IS

   COMPONENT mux4to1
    PORT (w0, w1, w2, w3     : IN   STD_LOGIC ;
          s                  : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
          f                  : OUT  STD_LOGIC ) ;
   END COMPONENT ;

   SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;

BEGIN
   Mux1: mux4to1 PORT MAP ( w(0), w(1), w(2), w(3),   s(1 DOWNTO 0), m(0) ) ;
   Mux2: mux4to1 PORT MAP ( w(4), w(5), w(6), w(7),   s(1 DOWNTO 0), m(1) ) ;
   Mux3: mux4to1 PORT MAP ( w(8), w(9), w(10), w(11), s(1 DOWNTO 0), m(2) ) ;
   Mux4: mux4to1 PORT MAP ( w(12),w(13),w(14), w(15), s(1 DOWNTO 0), m(3) ) ;
   Mux5: mux4to1 PORT MAP ( m(0), m(1), m(2),  m(3),  s(3 DOWNTO 2),   f  ) ;
END Structure ;
```

Modified Code for 16-to-1 MUX

```
ARCHITECTURE Structure OF Example1 IS
    COMPONENT mux4to1
     PORT (w0, w1, w2, w3   : IN  STD_LOGIC ;
           s                : IN  STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           f                : OUT  STD_LOGIC ) ;
    END COMPONENT ;

    SIGNAL m : STD_LOGIC_VECTOR(0 TO 3) ;
      BEGIN
          G1: FOR i IN 0 TO 3 GENERATE
              Muxes: mux4to1 PORT MAP (
                     w(4*i), w(4*i+1), w(4*i+2), w(4*i+3), s(1 DOWNTO 0), m(i) ) ;
          END GENERATE ;
          Mux5: mux4to1 PORT MAP ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
      END Structure ;
```

Array Attributes

Attribute	Description
A'left(N)	left bound of index range of dimension N of A
A'right(N)	right bound of index range of dimension N of A
A'low(N)	lower bound of index range of dimension N of A
A'high(N)	upper bound of index range of dimension N of A
A'range(N)	index range of dimension N of A
A'reverse_range(N)	reversed index range of dimension N of A
A'length(N)	length of index range of dimension N of A
A'ascending(N)	true if index range of dimension N of A is an ascending range, false otherwise

Array Attributes: Example

```
type A is array (1 to 4, 31 downto 0);

A'left(1)      = 1
A'right(2)     = 0
A'low(1)       = 1
A'high(2)      = 31
A'range(1)     = 1 to 4
A'length(2)    = 32
A'ascending(2) = false
```

Attributes of Scalar Types

Attributes	Description
T'left	first (leftmost) value in T
T'right	last (rightmost) value in T
T'low	least value in T
T'high	greatest value in T

Not available in VHDL-87:

Attributes	Description
T'ascending	**true** if T is an ascending range, **false** otherwise
T'image(x)	a string representing the value of x
T'value(s)	the value in T that is represented by s

Attributes of Scalar Types: Examples

```
type index_range is range 21 downto 11;

index_range'left          = 21
index_range'right         = 11
index_range'low           = 11
index_range'high          = 21
index_range'ascending     = false
index_range'image(14)     = "14"
index_range'value("20")   = 20
```

Attributes of Discrete Types

Attributes	Description
T'pos(x)	position number of x in T
T'val(n)	value in T at position n
T'succ(x)	value in T at position one greater than position of x
T'pred(x)	value in T at position one less than position of x
T'leftof(x)	value in T at position one to the left of x
T'rightof(x)	value in T at position one to the right of x

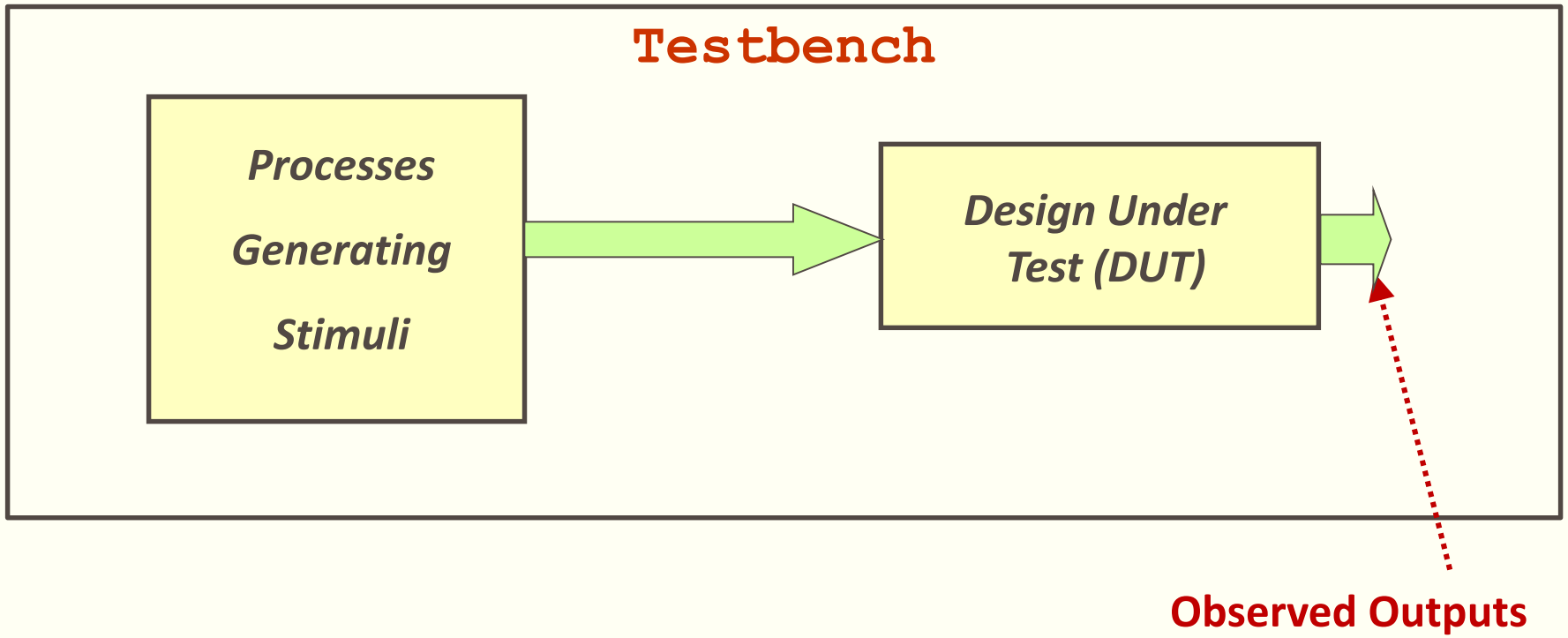
Attributes of Discrete Types:Examples

```
type logic_level is (unknown, low, undriven, high);

logic_level'pos(unknown)       = 0
logic_level'val(3)             = high
logic_level'succ(unknown)      = low
logic_level'pred(undriven)     = low
logic_level'leftof(unknown)    error
logic_level'rightof(undriven) = high
```

Attributes of Signals

- signal_name'**event** returns true if there is a ***change*** in a value of the signal.
- signal_name'**active** returns true if a value was assigned to the signal (even if the new value is the same as the current signal value).
- signal_name'**transaction** bit that returns true if a value was assigned to the signal (even if the new value is the same as the current signal value).
- signal_name'**last_event** returns the elapsed time since the last event that happened to the signal.
- signal_name'**last_active** returns the elapsed time since the last time that the signal was active.
- signal_name'**last_value** returns the last value that was assigned to the signal.

Testbenches



Testbench Definition

- Testbench applies stimuli (drives the inputs) to the Design Under Test (DUT) and (optionally) verifies expected outputs.
- The results can be viewed in a waveform window or written to a file.
- Since Testbench is written in VHDL, it is not restricted to a single simulation tool (portability).
- The same Testbench can be easily adapted to test different implementations (i.e. different architectures) of the same design.

Testbench Template

```
ENTITY tb IS
              --TB entity has no ports
END tb;

ARCHITECTURE arch_tb OF tb IS

              --Local signals and constants

  COMPONENT TestComp --All Design Under Test component declarations
            PORT (  );
  END COMPONENT;
-------------------------------------------------------
BEGIN
  testSequence: PROCESS
                                -- Input stimuli
  END PROCESS;

  DUT:TestComp PORT MAP(        -- Instantiations of DUTs
                        );
END arch_tb;
```

Generating Selected Values

```
SIGNAL test_vector : STD_LOGIC_VECTOR(2 downto 0);

BEGIN
    .......
            testing: PROCESS
            BEGIN
              test_vector <= "000";
              WAIT FOR 10 ns;
              test_vector <= "001";
              WAIT FOR 10 ns;
              test_vector <= "010";
              WAIT FOR 10 ns;
              test_vector <= "011";
              WAIT FOR 10 ns;
              test_vector <= "100";
              WAIT FOR 10 ns;
            END PROCESS;
  ........
END behavioral;
```

Generating all Values

```
SIGNAL test_vector : STD_LOGIC_VECTOR(3 downto 0):="0000";

BEGIN
     .......

   testing: PROCESS
   BEGIN
     WAIT FOR 10 ns;
     test_vector <= test_vector + 1;
   end process TESTING;

   ........
END behavioral;
```

Generating all Values

```
SIGNAL test_ab : STD_LOGIC_VECTOR(1 downto 0);
SIGNAL test_sel : STD_LOGIC_VECTOR(1 downto 0);

BEGIN
    .......
          double_loop: PROCESS
          BEGIN
                test_ab  <="00";
                test_sel  <="00";
                for I in 0 to 3 loop
                     for J in 0 to 3 loop
                         wait for 10 ns;
                             test_ab <= test_ab + 1;
                     end loop;
                     test_sel <= test_sel + 1;
                end loop;
          END PROCESS;
    ........
    END behavioral;
```

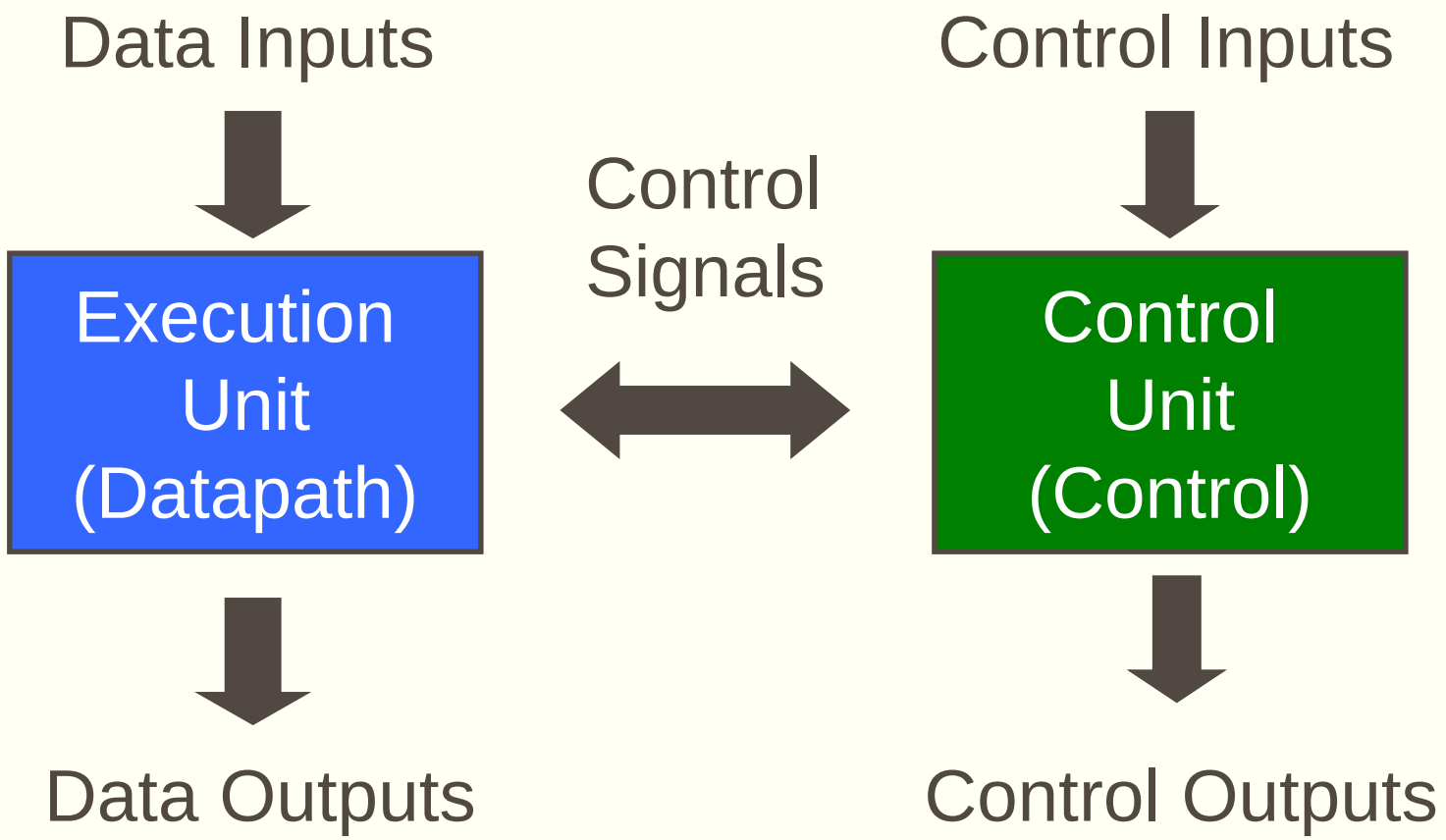
Generating Periodic Signals

```
    CONSTANT clk1_period : TIME := 20 ns;
    CONSTANT clk2_period : TIME := 200 ns;
    SIGNAL clk1 : STD_LOGIC;
    SIGNAL clk2 : STD_LOGIC := '0';

BEGIN
  .......
    clk1_generator: PROCESS
                    clk1 <= '0';
     WAIT FOR clk1_period/2;
     clk1 <= '1';
     WAIT FOR clk1_period/2;
  END PROCESS;

    clk2 <= not clk2  after  clk2_period/2;
  .......
END behavioral;
```

Generating 1 Time Signals

```
     CONSTANT reset1_width : TIME := 100 ns;
     CONSTANT reset2_width : TIME := 150 ns;
    SIGNAL reset1 : STD_LOGIC;
     SIGNAL reset2 : STD_LOGIC := '1';

BEGIN
     .......
     reset1_generator: PROCESS
         reset1 <= '1';
         WAIT FOR reset_width;
         reset1 <= '0';
         WAIT;
     END PROCESS;
     reset2_generator: PROCESS
         WAIT FOR reset_width;
         reset2 <= '0';
         WAIT;
     END PROCESS;
     .......
END behavioral;
```

Structure of a Digital System



Datapath

- Provides all necessary resources and interconnects among them to perform specified ttask

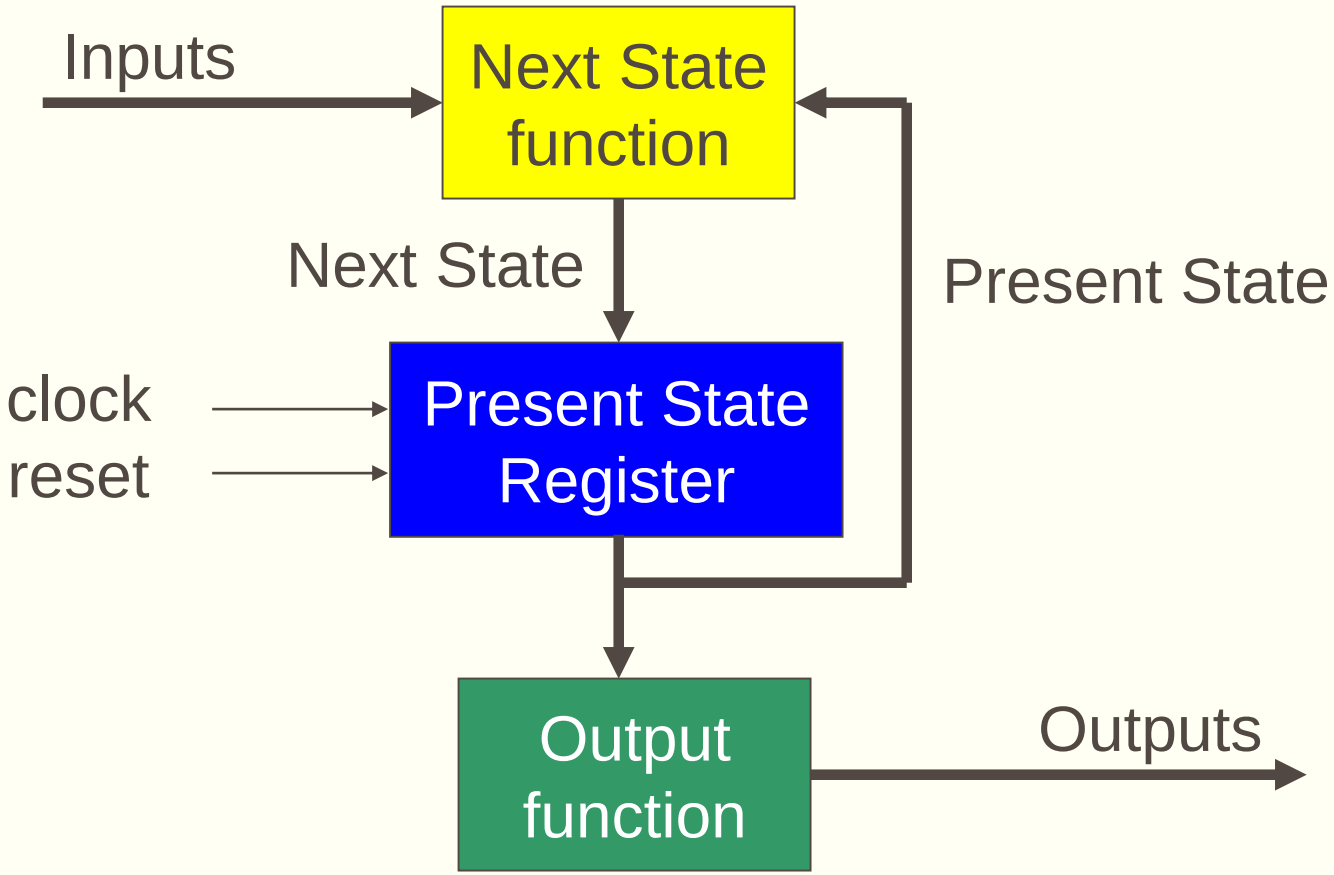
- Examples of resources
  - Adders, Multipliers, Registers, Memories, etc.

Control

- Controls data movements in an operational circuit by switching multiplexers and enabling or disabling resources
- Follows some ‘program’ or schedule
- Often implemented as Finite State Machine (FSM) or collection of Finite State Machines

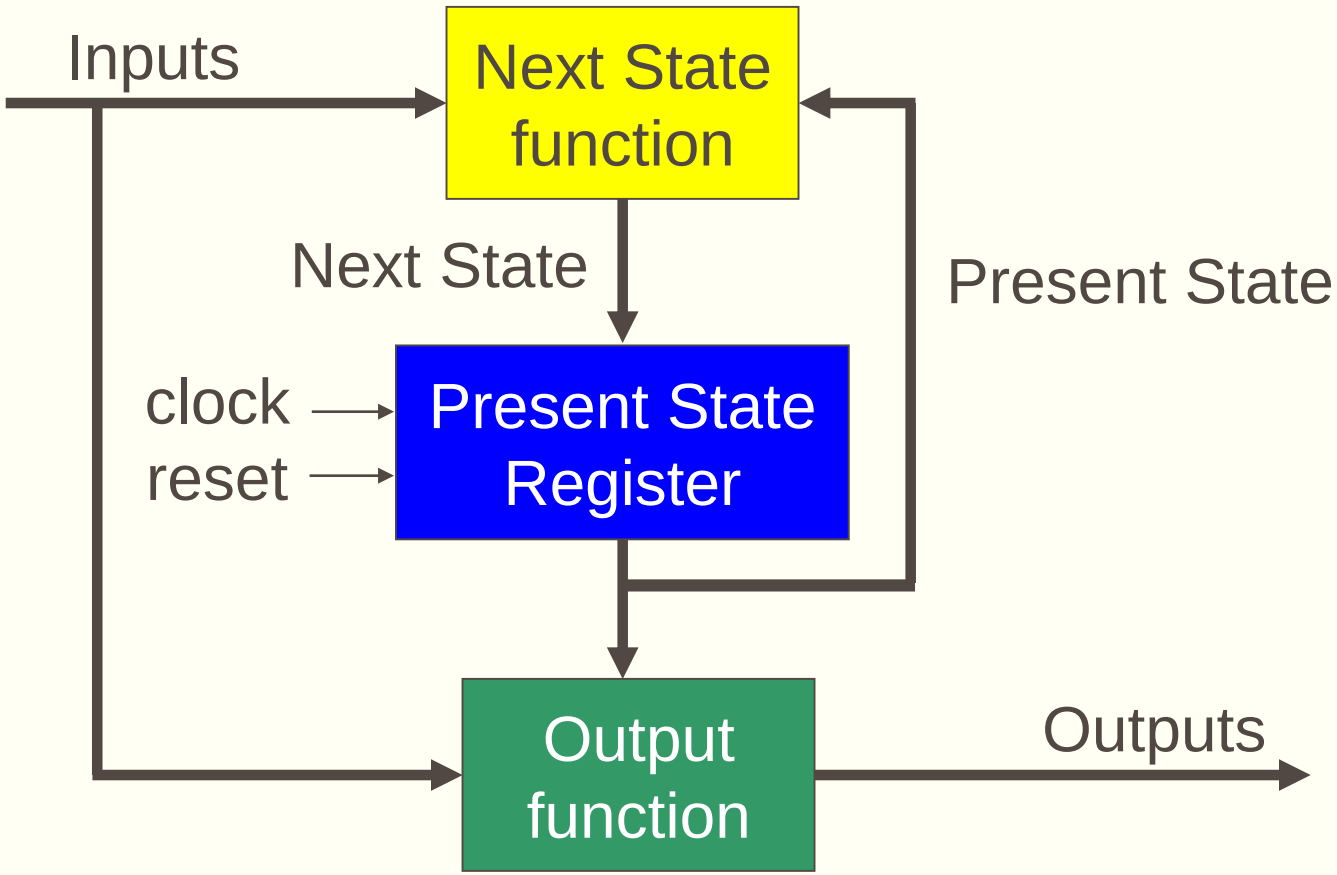
Moore FSM

Output is a function of a present state only

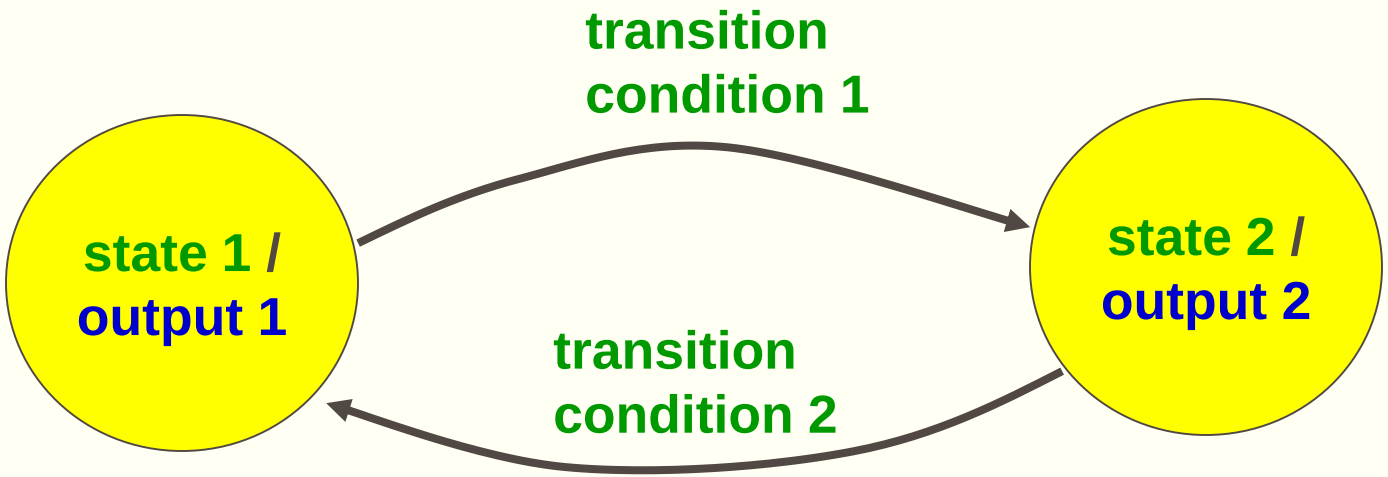


Mealy FSM

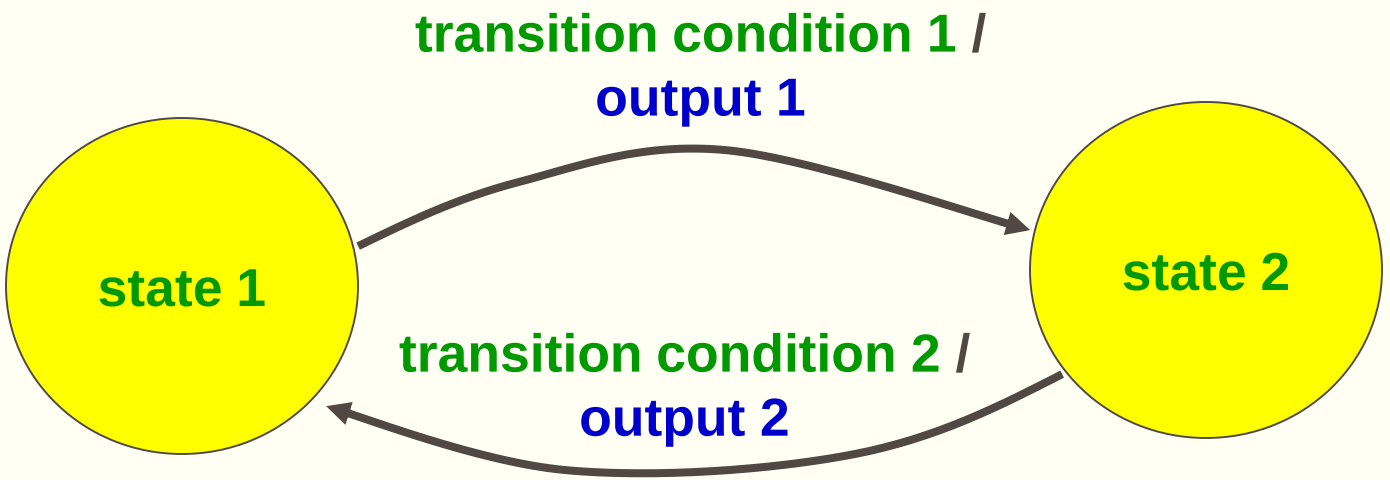
Output is a function of a present state and inputs



Moore FSM


transition
condition 1
state 1 /
output 1
state 2 /
output 2
transition
condition 2

Mealy FSM


transition condition 1 /
output 1
state 1
state 2
transition condition 2 /
output 2

FSMs in VHDL

- Finite State Machines (FSM) can be easily described with processes
- Synthesis tools understand FSM description if certain rules are followed
  - State transitions should be described in a process sensitive to *clock* and *asynchronous reset* signals only
  - Outputs described as concurrent statements outside the process

Moore FSM in VHDL

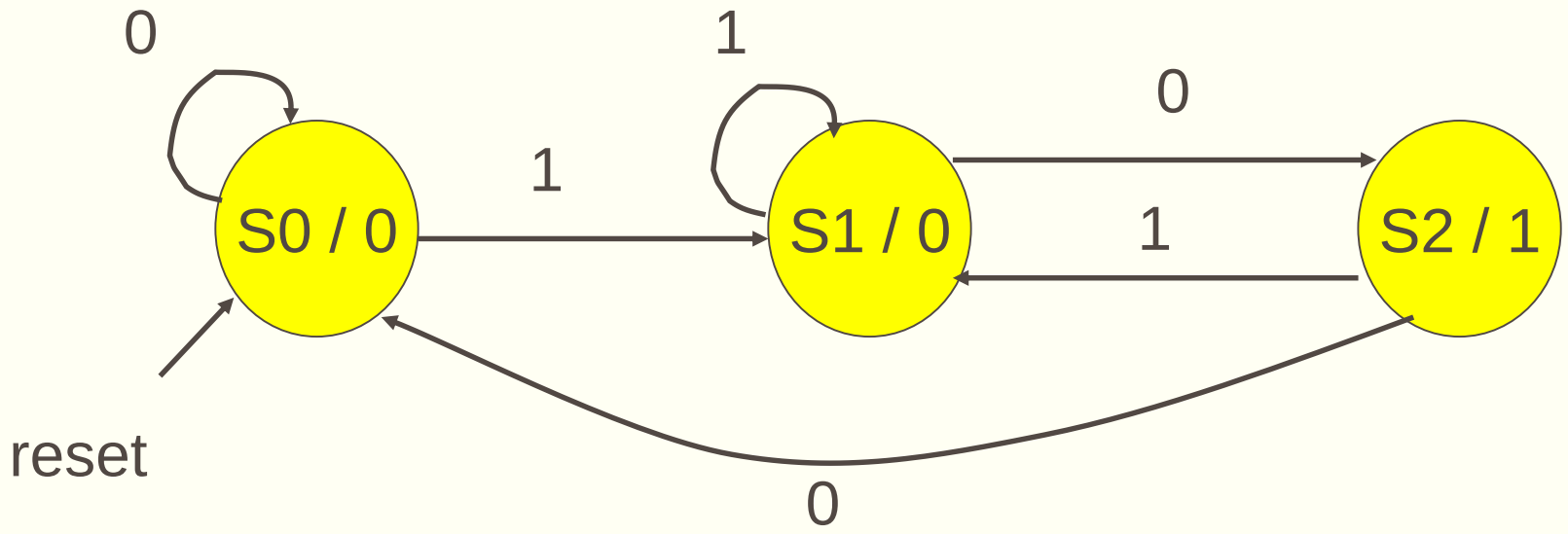
```
TYPE state IS (S0, S1, S2);
SIGNAL Moore_state: state;

U_Moore: PROCESS (clock, reset)
BEGIN
  IF(reset = '1') THEN
    Moore_state <= S0;
  ELSIF (clock = '1' AND clock'event) THEN
    CASE Moore_state IS
       WHEN S0 =>
          IF input = '1' THEN
             Moore_state <= S1;
          ELSE
             Moore_state <= S0;
          END IF;
```

continue…

```
       WHEN S1 =>
          IF input = '0' THEN
             Moore_state <= S2;
          ELSE
             Moore_state <= S1;
          END IF;
       WHEN S2 =>
          IF input = '0' THEN
             Moore_state <= S0;
          ELSE
             Moore_state <= S1;
          END IF;
     END CASE;
   END IF;
END PROCESS;

Output <= '1' WHEN Moore_state = S2
 ELSE '0';
```

State Diagram

Moore FSM that recognizes sequence “10”

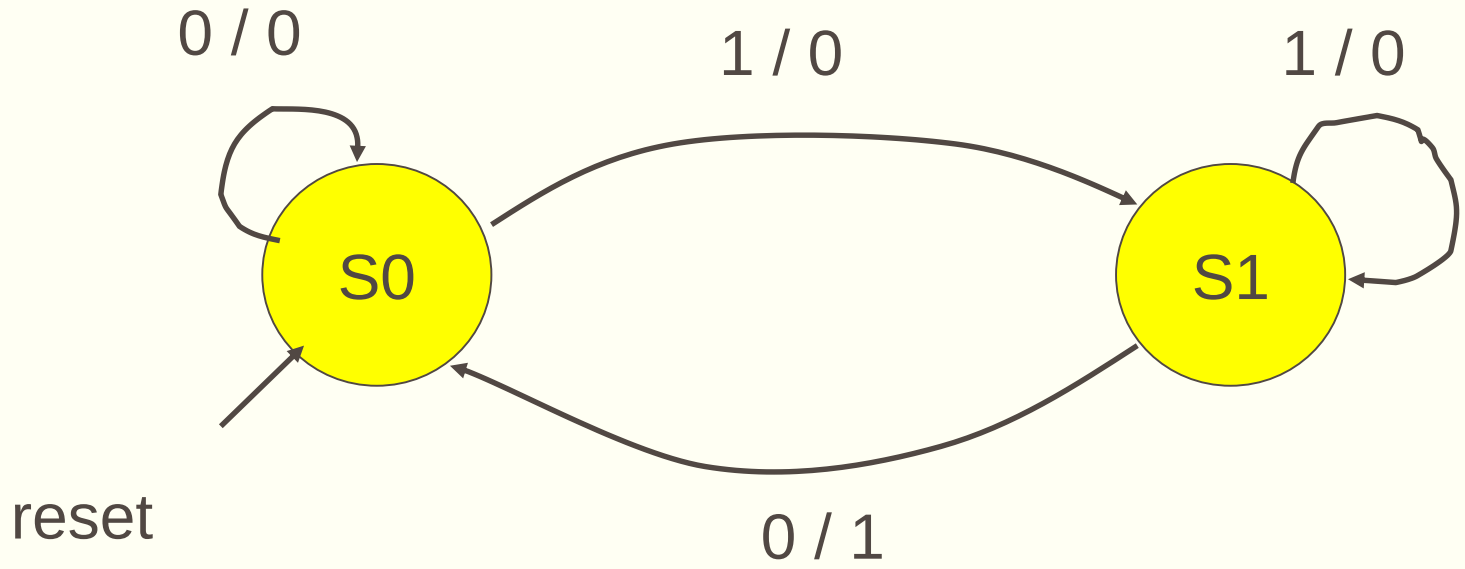


Mealy FSM in VHDL

```
TYPE state IS (S0, S1);
SIGNAL Mealy_state: state;

U_Mealy: PROCESS(clock, reset)
BEGIN
   IF(reset = '1') THEN
      Mealy_state <= S0;
   ELSIF (clock = '1' AND clock'event) THEN
      CASE Mealy_state IS
         WHEN S0 =>
            IF input = '1' THEN
               Mealy_state <= S1;
            ELSE
               Mealy_state <= S0;
            END IF;
         WHEN S1 =>
            IF input = '0' THEN
               Mealy_state <= S0;
            ELSE
               Mealy_state <= S1;
            END IF;
      END CASE;
   END IF;
END PROCESS;

Output <= '1' WHEN (Mealy_state = S1 AND
   input = '0') ELSE '0';
```

State Diagram

Mealy FSM that Recognizes Sequence “10”



Basic Logic Gates

```
-------------------------------------
-- OR gate
--
-- two descriptions provided
-------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-------------------------------------
entity OR_ent is
port(  x: in std_logic;
   y: in std_logic;
   F: out std_logic
);
end OR_ent;
-------------------------------------
```

Basic Logic Gates

```
architecture OR_arch of OR_ent is
begin

    process(x, y)
    begin
        -- compare to truth table
        if ((x='0') and (y='0')) then
            F <= '0';
        else
            F <= '1';
        end if;
    end process;
end OR_arch;

architecture OR_beh of OR_ent is
begin
    F <= x or y;
end OR_beh;
```

Basic Logic Gates

```
-------------------------------------
-- XNOR gate
--
-- two descriptions provided
-------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-------------------------------------
entity XNOR_ent is
port(  x: in std_logic;
   y: in std_logic;
   F: out std_logic
);
end XNOR_ent;
-------------------------------------
```

Basic Logic Gates

```
architecture behv1 of XNOR_ent is
begin
    process(x, y)
    begin
        -- compare to truth table
        if (x/=y) then
            F <= '0';
        else
            F <= '1';
        end if;
    end process;
end behv1;

architecture behv2 of XNOR_ent is
begin
    F <= x xnor y;
end behv2;
```

Combinational Circuits

```
------------------------------------------------------------
-- Combinational Logic Design
-- by Weijun Zhang, 04/2001
-- A simple example of VHDL Structure Modeling we might define
-- two components in two separate files, in main file, we use
-- port map statement to instantiate the mapping relationship
-- between each components and the entire circuit.
------------------------------------------------------------

library ieee;      -- component #1
use ieee.std_logic_1164.all;

entity OR_GATE is
port(X:   in std_logic;
     Y:   in std_logic;
     F2:  out std_logic
);
end OR_GATE;
```

Combinational Circuits

```
architecture behv of OR_GATE is
begin
process(X,Y)
begin
   F2 <= X or Y;      -- behavior des.
end process;
end behv;

-------------------------------------------

library ieee;         -- component #2
use ieee.std_logic_1164.all;

entity AND_GATE is
port(          A:     in std_logic;
   B: in std_logic;
   F1: out std_logic
);
end AND_GATE;
```

Combinational Circuits

```
architecture behv of AND_GATE is
begin
process(A,B)
begin
   F1 <= A and B;      -- behavior des.
end process;
end behv;

-------------------------------------------

library ieee;          -- top level circuit
use ieee.std_logic_1164.all;
use work.all;

entity comb_ckt is
port(input1: in std_logic;
   input2: in std_logic;
   input3: in std_logic;
   output: out std_logic
);
end comb_ckt;
```

Combinational Circuits

```
architecture struct of comb_ckt is component AND_GATE is
port( A:  in std_logic;
          B:  in std_logic;
           F1: out std_logic
    );
    end component;

    component OR_GATE is                -- as entity of OR_GATE
    port( X: in std_logic;
          Y: in std_logic;
         F2: out std_logic
    );
    end component;

    signal wire: std_logic;    -- signal just like wire
```

Combinational Circuits

```
begin

    -- use sign "=>" to clarify the pin mapping

    Gate1: AND_GATE port map (A=>input1, B=>input2, F1=>wire);
    Gate2: OR_GATE port map (X=>wire, Y=>input3, F2=>output);

end struct;

-------------------------------------------------------------
```

Tri-state Driver

```
-------------------------------------------------------
-- VHDL model for tri state driver by Weijun Zhang, 05/2001
-- this friver often used to control system outputs
-------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity tristate_dr is
port( d_in: in std_logic_vector(7 downto 0);
        en: in std_logic;
     d_out: out std_logic_vector(7 downto 0)
);
end tristate_dr;
```

Tri-state Driver

```
architecture behavior of tristate_dr is
begin

  process(d_in, en)
  begin
  if en='1' then
    d_out <= d_in;
  else
    -- array can be created simply by using vector
    d_out <= "ZZZZZZZZ";
  end if;
  end process;

end behavior;
```

Multiplexor

```
------------------------------------------------------------
-- VHDL code for 4:1 multiplexor -- by Weijun Zhang, 04/2001
-- Multiplexor is a device to select different inputs to
   outputs. We use 3 bits vector to describe its I/O ports
------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
------------------------------------------------------------
entity Mux is
port( I3: in std_logic_vector(2 downto 0);
      I2: in std_logic_vector(2 downto 0);
      I1: in std_logic_vector(2 downto 0);
      I0: in std_logic_vector(2 downto 0);
       S: in std_logic_vector(1 downto 0);
       O: out std_logic_vector(2 downto 0)
);
end Mux;
```

Multiplexor

```
architecture behv1 of Mux is
begin
      process(I3,I2,I1,I0,S)
      begin
           -- use case statement
           case S is
            when "00" =>   O <= I0;
            when "01" =>   O <= I1;
            when "10" =>   O <= I2;
            when "11" =>   O <= I3;
            when others => O <= "ZZZ";
  end case;
     end process;
end behv1;
```

Multiplexor

```
architecture behv2 of Mux is
begin
     -- use when.. else statement
        O <= I0 when S="00" else
             I1 when S="01" else
             I2 when S="10" else
             I3 when S="11" else
             "ZZZ";
end behv2;
```

Decoder

```
----------------------------------------------------
-- 2:4 Decoder
-- by Weijun Zhang, 04/2001
-- decoder is a kind of inverse process of multiplexor
----------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
----------------------------------------------------
entity DECODER is
port( I: in std_logic_vector(1 downto 0);
      O: out std_logic_vector(3 downto 0)
);
end DECODER;
----------------------------------------------------
```

Decoder

```
architecture behv of DECODER is
begin
    -- process statement
    process (I)
    begin
        -- use case statement
        case I is
            when "00" => O <= "0001";
            when "01" => O <= "0010";
            when "10" => O <= "0100";
            when "11" => O <= "1000";
            when others => O <= "XXXX";
        end case;
    end process;
end behv;
```

Decoder

```
architecture when_else of DECODER is
begin
    -- use when..else statement
    O <=  "0001" when I = "00" else
    "0010" when I = "01" else
    "0100" when I = "10" else
    "1000" when I = "11" else
    "XXXX";
end when_else;
```

Adder

```
----------------------------------------------------------------
-- VHDL code for n-bit adder by Weujun Zhang, 04/2001
-- function of adder: A plus B to get n-bit sum and 1 bit carry
-- we may use generic statement to set the parameter n of the adder.
----------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
------------------------------------------------
entity ADDER is
generic(n: natural :=2);
port(A:   in std_logic_vector(n-1 downto 0);
     B:   in std_logic_vector(n-1 downto 0);
 carry:   out std_logic;
   sum:   out std_logic_vector(n-1 downto 0)
);
end ADDER;
```

Adder

```
architecture behv of ADDER is
          -- define a temparary signal to store the result

signal result: std_logic_vector(n downto 0);

begin
    -- the 3rd bit should be carry

    result <= ('0' & A) + ('0' & B);
    sum <= result(n-1 downto 0);
    carry <= result(n);

end behv;
```

Comparator

```
------------------------------------------------------------
-- n-bit Comparator by Weijun Zhang, 04/2001
-- this simple comparator has two n-bit inputs &
-- three 1-bit outputs
------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
------------------------------------------------------------

entity Comparator is
generic(n: natural :=2);
port(A:  in std_logic_vector(n-1 downto 0);
     B:  in std_logic_vector(n-1 downto 0);
  less:  out std_logic;
 equal:  out std_logic;
greater: out std_logic
);
end Comparator;
```

Comparator

```
architecture behv of Comparator is
begin
    process(A,B)
    begin
        if (A<B) then
            less <= '1';
            equal <= '0';
            greater <= '0';
        elsif (A=B) then
            less <= '0';
            equal <= '1';
            greater <= '0';
        else
            less <= '0';
            equal <= '0';
            greater <= '1';
        end if;
    end process;
end behv;
```

ALU

```
----------------------------------------------------
-- Simple ALU Module
-- by Weijun Zhang, 04/2001
--
-- ALU stands for arithmatic logic unit.
-- It perform multiple operations according to
-- the control bits.
-- we use 2's complement subraction in this example
-- two 2-bit inputs & carry-bit ignored
----------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
----------------------------------------------------
```

ALU

```
entity ALU is

port( A: in std_logic_vector(1 downto 0);
      B: in std_logic_vector(1 downto 0);
    Sel: in std_logic_vector(1 downto 0);
    Res: out std_logic_vector(1 downto 0)
);

end ALU;

-------------------------------------------------
```

ALU

```
architecture behv of ALU is
begin
    process(A,B,Sel)
    begin
   -- use case statement to achieve different operations of ALU
   case Sel is
       when "00" =>
    Res <= A + B;
       when "01" =>
           Res <= A + (not B) + 1;
             when "10" =>
    Res <= A and B;
       when "11" =>
    Res <= A or B;
       when others =>
    Res <= "XX";
         end case;
    end process;
end behv;
-----------------------------------------------
```

Multiplier

```
-------------------------------------------------------
-- Example of doing multiplication showing
-- (1) how to use variable with in process
-- (2) how to use for loop statement
-- (3) algorithm of multiplication
--
-- by Weijun Zhang, 05/2001
-------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- two 4-bit inputs and one 8-bit outputs
entity multiplier is
port(num1, num2:  in std_logic_vector(1 downto 0);
        product:  out std_logic_vector(3 downto 0)
);
end multiplier;
```

Multiplier

```
architecture behv of multiplier is
begin
process(num1, num2)
  variable num1_reg: std_logic_vector(2 downto 0);
  variable product_reg: std_logic_vector(5 downto 0);
begin
  num1_reg := '0' & num1;
  product_reg := "0000" & num2;
  -- use variables doing computation
  -- algorithm is to repeat shifting/adding
  for i in 1 to 3 loop
    if product_reg(0)='1' then
     product_reg(5 downto 3) := product_reg(5 downto 3) + num1_reg(2 downto 0);
   end if;
   product_reg(5 downto 0) := '0' & product_reg(5 downto 1);
  end loop;
  -- assign the result of computation back to output signal
  product <= product_reg(3 downto 0);
end process;
end behv;
```

Simple Latch

```
-----------------------------------------
-- Simple D Latch
-- by Weijun Zhang, 04/2001
-- latch is simply controlled by enable bit
-- but has nothing to do with clock sigal
-- notice this difference from flip-flops
-----------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
-----------------------------------------
entity D_latch is
port(data_in: in std_logic;
   enable:          in std_logic;
   data_out:  out std_logic
);
end D_latch;
-----------------------------------------
```

Simple Latch

```
architecture behv of D_latch is
begin

    -- compare this to D flipflop

    process(data_in, enable)
    begin
        if (enable='1') then
            -- no clock signal here
       data_out <= data_in;
   end if;
    end process;

end behv;

-----------------------------------------
```

D Flip-flop

```
-------------------------------------------
-- D Flip-Flop
-- by Weijun Zhang, 04/2001
--
-- Flip-flop is the basic component in
-- sequential logic design
-- we assign input signal to the output
-- at the clock rising edge
-------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
use work.all;
-------------------------------------------

entity dff is
port( data_in: in std_logic;
        clock: in std_logic;
     data_out: out std_logic
);
end dff;
```

D Flip-flop

```
-----------------------------------------

architecture behv of dff is
begin

    process(data_in, clock)
    begin

        -- clock rising edge

   if (clock='1' and clock'event) then
       data_out <= data_in;
   end if;

    end process;

end behv;

-----------------------------------------
```

JK Flip-flop

```
-----------------------------------------
-- JK Flip-Flop with reset
-- by Weijun Zhang, 04/2001
--
-- the description of JK Flip-Flop is based
-- on functional truth table
-- concurrent statement and signal assignment
-- are using in this example
-----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------

entity JK_FF is
port ( clock: in std_logic;
         J, K: in std_logic;
        reset:  in std_logic;
      Q, Qbar: out std_logic
);
end JK_FF;
```

JK Flip-flop

```
----------------------------------------------

architecture behv of JK_FF is

    -- define the useful signals here

    signal state: std_logic;
    signal input: std_logic_vector(1 downto 0);
```

JK Flip-flop

```
begin
    -- combine inputs into vector
    input <= J & K;
    p: process(clock, reset) is
    begin
       if (reset='1') then
          state <= '0';
       elsif (rising_edge(clock)) then
       -- compare to the truth table
          case (input) is
             when "11" => state <= not state;
             when "10" => state <= '1';
             when "01" => state <= '0';
             when others => null;
          end case;
       end if;
   end process;
```

JK Flip-flop

```
    -- concurrent statements
    Q <= state;
    Qbar <= not state;

end behv;

-------------------------------------------------
```

Register

```
----------------------------------------------------
-- n-bit Register
-- by Weijun Zhang, 04/2001
----------------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
----------------------------------------------------

entity reg is

generic(n: natural :=2);
port(  I: in std_logic_vector(n-1 downto 0);
   clock: in std_logic;
    load: in std_logic;
   clear: in std_logic;
       Q: out std_logic_vector(n-1 downto 0)
);
end reg;
```

Register

```
-------------------------------------------------
architecture behv of reg is
    signal Q_tmp: std_logic_vector(n-1 downto 0);
begin
    process(I, clock, load, clear)
    begin
      if clear = '0' then     -- use 'range in signal assignment
        Q_tmp <= (Q_tmp'range => '0');
      elsif (clock='1' and clock'event) then
        if load = '1' then
          Q_tmp <= I;
        end if;
      end if;
    end process;
    -- concurrent statement
    Q <= Q_tmp;
end behv;
-------------------------------------------------
```

Shift Register

```
-------------------------------------------------
-- 3-bit Shift-Register/Shifter
-- by Weijun Zhang, 04/2001
-- reset is ignored
-------------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
-------------------------------------------------

entity shift_reg is
port(  I: in std_logic;
   clock: in std_logic;
   shift: in std_logic;
       Q: out std_logic
);
end shift_reg;

-------------------------------------------------
```

Shift Register

```
architecture behv of shift_reg is
    -- initialize the declared signal
    signal S: std_logic_vector(2 downto 0):="111";

begin
    process(I, clock, shift, S)
    begin    -- everything happens upon the clock changing
   if clock'event and clock='1' then
       if shift = '1' then
           S <= I & S(2 downto 1);
       end if;
   end if;
end process;

-- concurrent assignment
Q <= S(0);

end behv;
-------------------------------------------------
```

Counter

```
-------------------------------------------------
-- VHDL code for n-bit counter by Weijun Zhang, 04/2001
-- this is the behavior description of n-bit counter
-- another way can be used is FSM model.
-------------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
-------------------------------------------------
entity counter is

generic(n: natural :=2);
port(clock:   in std_logic;
     clear:   in std_logic;
     count:   in std_logic;
         Q:   out std_logic_vector(n-1 downto 0)
);
end counter;
-------------------------------------------------
```

Counter

```
architecture behv of counter is
    signal Pre_Q: std_logic_vector(n-1 downto 0);

begin
    -- behavior describe the counter
    process(clock, count, clear)
    begin
       if clear = '1' then
          Pre_Q <= Pre_Q - Pre_Q;
       elsif (clock='1' and clock'event) then
          if count = '1' then
             Pre_Q <= Pre_Q + 1;
          end if;
       end if;
    end process;
-- concurrent assignment statement
Q <= Pre_Q;

end behv;
-------------------------------------------------
```

FSM

```
-------------------------------------------------
-- VHDL FSM (Finite State Machine) modeling
-- by Weijun Zhang, 04/2001
--
-- FSM model consists of two concurrent processes state_reg
-- and comb_logic we use case statement to describe the
-- state transistion. All the inputs and signals are
-- put into the process sensitive list.
-------------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
-------------------------------------------------
entity seq_design is
port(a:              in std_logic;
   clock:            in std_logic;
   reset:            in std_logic;
   x:   out std_logic
);
end seq_design;
-------------------------------------------------
```

FSM

```
architecture FSM of seq_design is
-- define the states of FSM model

type state_type is (S0, S1, S2, S3);
signal next_state, current_state: state_type;

begin
   -- cocurrent process#1: state registers
   state_reg: process(clock, reset)
   begin

      if (reset='1') then
          current_state <= S0;
      elsif (clock'event and clock='1') then
          current_state <= next_state;
      end if;

   end process;
```

FSM

```
-- cocurrent process#2: combinational logic
   comb_logic: process(current_state, a)
   begin
 -- use case statement to show the state transition
      case current_state is
         when S0 => x <= '0';
            if a='0' then next_state <= S0;
            elsif a ='1' then next_state <= S1;
            end if;

         when S1 => x <= '0';
            if a='0' then next_state <= S1;
            elsif a='1' then next_state <= S2;
            end if;
```

FSM

```
        when S2 => x <= '0';
           if a='0' then next_state <= S2;
           elsif a='1' then next_state <= S3;
           end if;

        when S3 => x <= '1';
           if a='0' then next_state <= S3;
           elsif a='1' then next_state <= S0;
           end if;

        when others => x <= '0';
           next_state <= S0;
       end case;
    end process;
end FSM;
-------------------------------------------------
```

RAM

```
-------------------------------------------------------------
-- a simple 4*4 RAM module by Weijun Zhang
-------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-------------------------------------------------------------
entity SRAM is
generic( width:  integer:=4; depth: integer:=4; addr: integer:=2);
port(    Clock:  in std_logic;
        Enable:  in std_logic;
          Read:  in std_logic;
         Write:  in std_logic;
     Read_Addr:  in std_logic_vector(addr-1 downto 0);
    Write_Addr:  in std_logic_vector(addr-1 downto 0);
       Data_in:  in std_logic_vector(width-1 downto 0);
      Data_out:  out std_logic_vector(width-1 downto 0)
);
end SRAM;
```

RAM

```
architecture behav of SRAM is
-- use array to define the bunch of internal temparary signals
type ram_type is array (0 to depth-1) of std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type;
begin
    -- Read Functional Section
    process(Clock, Read)
    begin
       if (Clock'event and Clock='1') then
          if Enable='1' then
             if Read='1' then
                -- buildin function conv_integer change the type
                -- from std_logic_vector to integer
                Data_out <= tmp_ram(conv_integer(Read_Addr));
             else
                Data_out <= (Data_out'range => 'Z');
             end if;
          end if;
       end if;
   end process;
```

RAM

```
    -- Write Functional Section
    process(Clock, Write)
    begin
       if (Clock'event and Clock='1') then
          if Enable='1' then
             if Write='1' then
                tmp_ram(conv_integer(Write_Addr)) <= Data_in;
             end if;
          end if;
       end if;
   end process;

end behav;
---------------------------------------------------------------
```

ROM

```
-------------------------------------------------------------
-- 32*8 ROM module by Weijun Zhang, 04/2001
-- ROM model has predefined content for read only purpose
-------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ROM is
port(Clock: in std_logic;
     Reset: in std_logic;
    Enable: in std_logic;
      Read: in std_logic;
   Address: in std_logic_vector(4 downto 0);
  Data_out: out std_logic_vector(7 downto 0)
);
end ROM;
-------------------------------------------------------------
```

ROM

```
architecture Behav of ROM is type ROM_Array is array (0 to 31)
   of std_logic_vector(7 downto 0);

    constant Content: ROM_Array := (
        0 => "00000001",            -- Suppose ROM has
        1 => "00000010",            -- prestored value
        2 => "00000011",            -- like this table
        3 => "00000100",            --
        4 => "00000101",            --
        5 => "00000110",            --
        6 => "00000111",            --
        7 => "00001000",            --
        8 => "00001001",            --
        9 => "00001010",            --
        10 => "00001011",           --
        11 => "00001100",           --
        12 => "00001101",           --
        13 => "00001110",           --
        14 => "00001111",           --
    OTHERS => "11111111"            --
   );
```

ROM

```
begin
    process(Clock, Reset, Read, Address)
    begin
        if( Reset = '1' ) then
           Data_out <= "ZZZZZZZZ";
        elsif( Clock'event and Clock = '1' ) then
           if Enable = '1' then
              if( Read = '1' ) then
                 Data_out <= Content(conv_integer(Address));
              else
                 Data_out <= "ZZZZZZZZ";
              end if;
           end if;
        end if;
    end process;
end Behav;
-------------------------------------------------------------
```

GCD Calculator - Behavioral

```
----------------------------------------------------
-- Behvaior Design of GCD calculator
-- by Weijun Zhang, 05/2001
----------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.all;
----------------------------------------------------
entity gcd1 is
port(   Data_X:    in unsigned(3 downto 0);
        Data_Y:    in unsigned(3 downto 0);
        Data_out:  out unsigned(3 downto 0)
);
end gcd1;
```

GCD Calculator - Behavioral

```
architecture behv of gcd1 is
begin
   process(Data_X, Data_Y)
   variable tmp_X, tmp_Y: unsigned(3 downto 0);
   begin
      tmp_X := Data_X;
      tmp_Y := Data_Y;
      for i in 0 to 15 loop
         if (tmp_X/=tmp_Y) then
            if (tmp_X < tmp_Y) then
               tmp_Y := tmp_Y - tmp_X;
            else
               tmp_X := tmp_X - tmp_Y;
            end if;
         else
            Data_out <= tmp_X;
         end if;
      end loop;
   end process;
end behv;
-------------------------------------------------
```

GCD Calculator - FSMD

```
-------------------------------------------------------
-- GCD design using FSMD by Weijun Zhang, 04/2001
-- GCD algorithm behavior modeling (GCD.vhd)
-- the calculator has two 4-bit inputs and one output
-- NOTE: idle state required to obtain the correct synthesis results
-------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.all;
-------------------------------------------------------
entity gcd is
port(clk: in std_logic;
     rst: in std_logic;
    go_i: in std_logic;
     x_i: in unsigned(3 downto 0);
     y_i: in unsigned(3 downto 0);
     d_o: out unsigned(3 downto 0)
);
end gcd;
```

GCD Calculator - FSMD

```
architecture FSMD of gcd is
begin
    process(rst, clk)
    -- define states using variable
    type S_Type is (ST0, ST1, ST2);
    variable State: S_Type := ST0 ;
    variable Data_X, Data_Y: unsigned(3 downto 0);
    begin
        if (rst='1') then   -- initialization
           d_o <= "0000";
           State := ST0;
        elsif (clk'event and clk='1') then
           case State is
              when ST0 =>    -- starting
                 if (go_i='1') then
                    Data_X := x_i;
                    Data_Y := y_i;
                    State := ST1;
                 else
                    State := ST0;
                 end if;
```

GCD Calculator - FSMD

```
            when ST1 => State := ST2;        -- idle state
            when ST2 =>                      -- computation
               if (Data_X/=Data_Y) then
                  if (Data_X<Data_Y) then
                     Data_Y := Data_Y - Data_X;
                  else
                     Data_X := Data_X - Data_Y;
                  end if;
               State := ST1;
               else
                  d_o <=Data_X;              -- done
                  State := ST0;
               end if;
            when others =>                   -- go back
               d_o <= "ZZZZ";
               State := ST0;
            end case;
         end if;
      end process;
end FSMD;
```

GCD Calculator – FSM+D

```
-------------------------------------------------------------------
-- GCD CALCULATOR by Weijun Zhang, 04/2001
-- we can put all the components in one document(gcd2.vhd)
-- or put them in separate files
-- this is the example of RT level modeling (FSM + DataPath)
-- the code is synthesized by Synopsys design compiler
-------------------------------------------------------------------
-- Component: MULTIPLEXOR ---------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity mux is
    port(rst, sLine: in std_logic;
       load, result: in std_logic_vector( 3 downto 0 );
             output: out std_logic_vector( 3 downto 0 )
    );
end mux;
```

GCD Calculator – FSM+D

```
architecture mux_arc of mux is
begin
    process( rst, sLine, load, result )
    begin
       if( rst = '1' ) then
          output <= "0000";          -- do nothing
       elsif sLine = '0' then
          output <= load;            -- load inputs
       else
          output <= result;          -- load results
       end if;
   end process;
end mux_arc;
```

GCD Calculator – FSM+D

```
-- Component: COMPARATOR ------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity comparator is
   port( rst: in std_logic;
         x, y: in std_logic_vector( 3 downto 0 );
       output: out std_logic_vector( 1 downto 0 )
        );
end comparator;
```

GCD Calculator – FSM+D

```
architecture comparator_arc of comparator is
begin
    process( x, y, rst )
    begin
  if( rst = '1' ) then
      output <= "00";          -- do nothing
  elsif( x > y ) then
      output <= "10";          -- if x greater
  elsif( x < y ) then
      output <= "01";          -- if y greater
  else
      output <= "11";          -- if equivalance
  end if;
    end process;
end comparator_arc;
```

GCD Calculator – FSM+D

```
-- Component: SUBTRACTOR ------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity subtractor is
    port( rst: in std_logic;
          cmd: in std_logic_vector( 1 downto 0 );
        x, y: in std_logic_vector( 3 downto 0 );
   xout, yout: out std_logic_vector( 3 downto 0 )
    );
end subtractor;
```

GCD Calculator – FSM+D

```
architecture subtractor_arc of subtractor is
begin
    process( rst, cmd, x, y )
    begin
       if(rst = '1' or cmd = "00" ) then     -- not active
          xout <= "0000";
          yout <= "0000";
       elsif( cmd = "10" ) then              -- x is greater
          xout <= ( x - y );
          yout <= y;
       elsif( cmd = "01" ) then              -- y is greater
          xout <= x;
          yout <= ( y - x );
       else
          xout <= x;                        -- x and y are equal
          yout <= y;
       end if;
   end process;
end subtractor_arc;
```

GCD Calculator – FSM+D

```
-- Component: REGISTER ----------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity regis is
    port( rst, clk, load: in std_logic;
                  input: in std_logic_vector( 3 downto 0 );
                 output: out std_logic_vector( 3 downto 0 )
    );
end regis;
```

GCD Calculator – FSM+D

```
architecture regis_arc of regis is
begin
    process( rst, clk, load, input )
    begin
       if( rst = '1' ) then
          output <= "0000";
       elsif( clk'event and clk = '1') then
          if( load = '1' ) then
          output <= input;
          end if;
       end if;
   end process;
end regis_arc;
```

GCD Calculator – FSM+D

```
-- component: FSM controller ----------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity fsm is
    port( rst, clk, proceed: in std_logic;
   comparison: in std_logic_vector( 1 downto 0 );
   enable, xsel, ysel, xld, yld: out std_logic
   );
end fsm;
```

GCD Calculator – FSM+D

```
architecture fsm_arc of fsm is
    type states is ( init, s0, s1, s2, s3, s4, s5 );
    signal nState, cState: states;

begin
    process( rst, clk )
    begin
       if( rst = '1' ) then
          cState <= init;
       elsif( clk'event and clk = '1' ) then
          cState <= nState;
       end if;
   end process;
```

GCD Calculator – FSM+D

```
process( proceed, comparison, cState )
   begin
      case cState is
         when init =>
            if( proceed = '0' ) then nState <= init;
            else nState <= s0;
            end if;

         when s0 => enable <= '0';
            xsel <= '0';
            ysel <= '0';
            xld <= '0';
            yld <= '0';
            nState <= s1;

         when s1 => enable <= '0';
            xsel <= '0';
            ysel <= '0';
            xld <= '1';
            yld <= '1';
            nState <= s2;
```

GCD Calculator – FSM+D

```
when s2 => xld <= '0';
   yld <= '0';
   if( comparison = "10" ) then nState <= s3;
   elsif( comparison = "01" ) then nState <= s4;
   elsif( comparison = "11" ) then nState <= s5;
   end if;
when s3 => enable <= '0';
   xsel <= '1';
   ysel <= '0';
   xld <= '1';
   yld <= '0';
   nState <= s2;
when s4 => enable <= '0';
   xsel <= '0';
   ysel <= '1';
   xld <= '0';
   yld <= '1';
   nState <= s2;
```

GCD Calculator – FSM+D

```
            when s5 => enable <= '1';
               xsel <= '1';
               ysel <= '1';
               xld <= '1';
               yld <= '1';
               nState <= s0;

            when others => nState <= s0;
         end case;
      end process;
end fsm_arc;
```

GCD Calculator – FSM+D

```
------------------------------------------------------------------
-- GCD Calculator: top level design using structural modeling
-- FSM + Datapath (mux, registers, subtractor and comparator)
------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.all;

entity gcd is
    port( rst, clk, go_i: in std_logic;
                  x_i, y_i: in std_logic_vector( 3 downto 0 );
                       d_o: out std_logic_vector( 3 downto 0 )
    );
end gcd;
```

GCD Calculator – FSM+D

```
architecture gcd_arc of gcd is
component fsm is
port( rst, clk, proceed: in std_logic;
                comparison: in std_logic_vector( 1 downto 0 );
enable, xsel, ysel, xld, yld: out std_logic
    );
end component;

component mux is
port( rst, sLine: in std_logic;
    load, result: in std_logic_vector( 3 downto 0 );
          output: out std_logic_vector( 3 downto 0 )
    );
end component;

component comparator is
    port( rst: in std_logic;
         x, y: in std_logic_vector( 3 downto 0 );
       output: out std_logic_vector( 1 downto 0 )
    );
end component;
```

GCD Calculator – FSM+D

```
component subtractor is
   port( rst: in std_logic;
         cmd: in std_logic_vector( 1 downto 0 );
       x, y: in std_logic_vector( 3 downto 0 );
  xout, yout: out std_logic_vector( 3 downto 0 )
    );
end component;

component regis is
   port( rst, clk, load: in std_logic;
                  input: in std_logic_vector( 3 downto 0 );
                 output: out std_logic_vector( 3 downto 0 )
    );
end component;

signal xld, yld, xsel, ysel, enable: std_logic;
signal comparison: std_logic_vector( 1 downto 0 );
signal result: std_logic_vector( 3 downto 0 );

signal xsub, ysub, xmux, ymux, xreg, yreg: std_logic_vector( 3 downto 0 );
```

GCD Calculator – FSM+D

```
begin
    -- doing structure modeling here
    -- FSM controller
    TOFSM: fsm port map(    rst, clk, go_i, comparison,
         enable, xsel, ysel, xld, yld );
    -- Datapath
    X_MUX:   mux port map( rst, xsel, x_i, xsub, xmux );
    Y_MUX:   mux port map( rst, ysel, y_i, ysub, ymux );
    X_REG:   regis port map( rst, clk, xld, xmux, xreg );
    Y_REG:   regis port map( rst, clk, yld, ymux, yreg );
    U_COMP:  comparator port map( rst, xreg, yreg, comparison );
    X_SUB:   subtractor port map( rst, comparison, xreg, yreg, xsub, ysub );
    OUT_REG: regis port map( rst, clk, enable, xsub, result );

    d_o <= result;

end gcd_arc;
```

FIR Filter

```
------------------------------------------------------------
-- FIR Digital Filter Design (DSP example)
-- VHDL Data-Flow modeling
-- KEYWORD: generate, array, range, constant and subtype
------------------------------------------------------------
library IEEE;                              -- declare the library
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
------------------------------------------------------------
entity FIR_filter is
port( rst:  in  std_logic;
      clk:  in  std_logic;
  coef_ld:  in  std_logic;
    start:  in  std_logic;
 o_enable:  in  std_logic;
   bypass:  in  std_logic;
    Xn_in:  in  std_logic_vector(3 downto 0);
    Yn_in:  in  std_logic_vector(15 downto 0);
   Xn_out:  out std_logic_vector(3 downto 0);
   Yn_out:  out std_logic_vector(15 downto 0)
); end FIR_filter;
```

FIR Filter

```
architecture BEH of FIR_filter is
    constant K:        integer := 4;          -- circuit has four stages

    -- use type and subtype to define the complex signals
    subtype bit4 is    std_logic_vector(3 downto 0);
    subtype bit8 is    std_logic_vector(7 downto 0);
    subtype bit16 is   std_logic_vector(15 downto 0);

    type klx4  is      array (K downto 0) of bit4;
    type kx8   is      array (K-1 downto 0) of bit8;
    type klx8  is       array (K downto 0) of bit8;
    type kx16  is      array (K-1 downto 0) of bit16;
    type klx16 is      array (K downto 0) of bit16;

    -- define signal in type of arrays
    signal     REG1, REG2, COEF       : klx4;
    signal     MULT8                  : kx8;
    signal     MULT16                 : kx16;
    signal     SUM                    : klx16;
    signal     Xn_tmp                 : bit4;
    signal     Yn_tmp                 : bit16;
```

FIR Filter

```
begin

    -- initialize the first stage of FIR circuit

    REG2(K) <= Xn_in when (start='1')
       else (REG2(K)'range => '0');
    REG1(K) <= (REG1(K)'range => '0');
    SUM(K)  <= Yn_in;
    COEF(K) <= Xn_in;
```

FIR Filter

```
-- start the computation, use generate to obtain the multiple stages
   gen8: for j in K-1 downto 0 generate
       stages: process (rst, clk)
       begin
          if (rst='0') then
             REG1(j) <= (REG1(j)'range => '0');
             REG2(j) <= (REG2(j)'range => '0');
             COEF(j) <= (COEF(j)'range => '0');
             MULT16(j) <= (MULT16(j)'range => '0');
             SUM(j) <= (SUM(j)'range => '0');
          elsif (clk'event and clk ='1') then
             REG1(j) <= REG2(j+1); REG2(j) <= REG1(j);
             if (coef_ld = '1') then COEF(j) <= COEF(j+1); end if;
                MULT8(j) <= signed(REG1(j))*signed(COEF(j));
                MULT16(j)(7 downto 0) <= MULT8(j);
                if (MULT8(j)(7)='0') then MULT16(j)(15 downto 8) <= "00000000";
                else MULT16(j)(15 downto 8)  <=  "11111111";
                end if;
             SUM(j) <= signed(MULT16(j))+signed(SUM(j+1));
             end if;
         end process;
    end generate;
```

FIR Filter

```
    -- control the outputs by concurrent statements

    Xn_tmp <= Xn_in when bypass = '1' else
              REG2(0) when coef_ld = '0' else
             COEF(0);

    Yn_tmp <= Yn_in when bypass = '1' else
              SUM(0);

    Xn_out <= Xn_tmpwhen o_enable = '0' else
             (Xn_out'range => 'Z');

    Yn_out <= Yn_tmpwhen o_enable = '0' else
             (Yn_out'range => 'Z');

end BEH;

-------------------------------------------------------------
```

THANK YOU