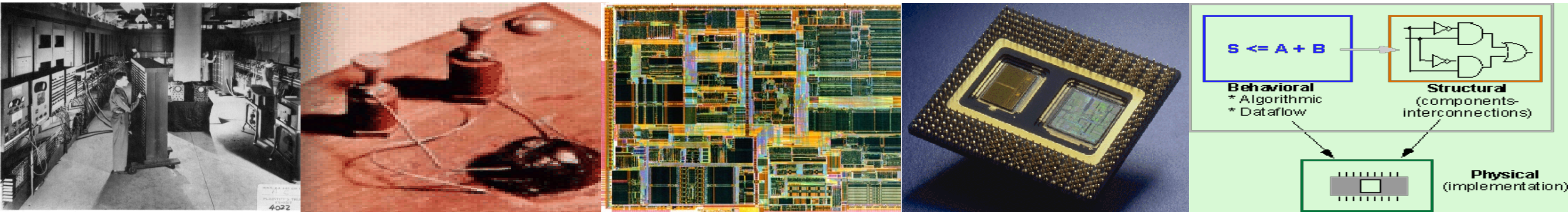
Lecture 3
Introduction to VHDL

Bryan Ackland
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ 07030



Managing Design Complexity

To be successful, designer (or design team) must manage placement and interconnect of up to 10^8 components…

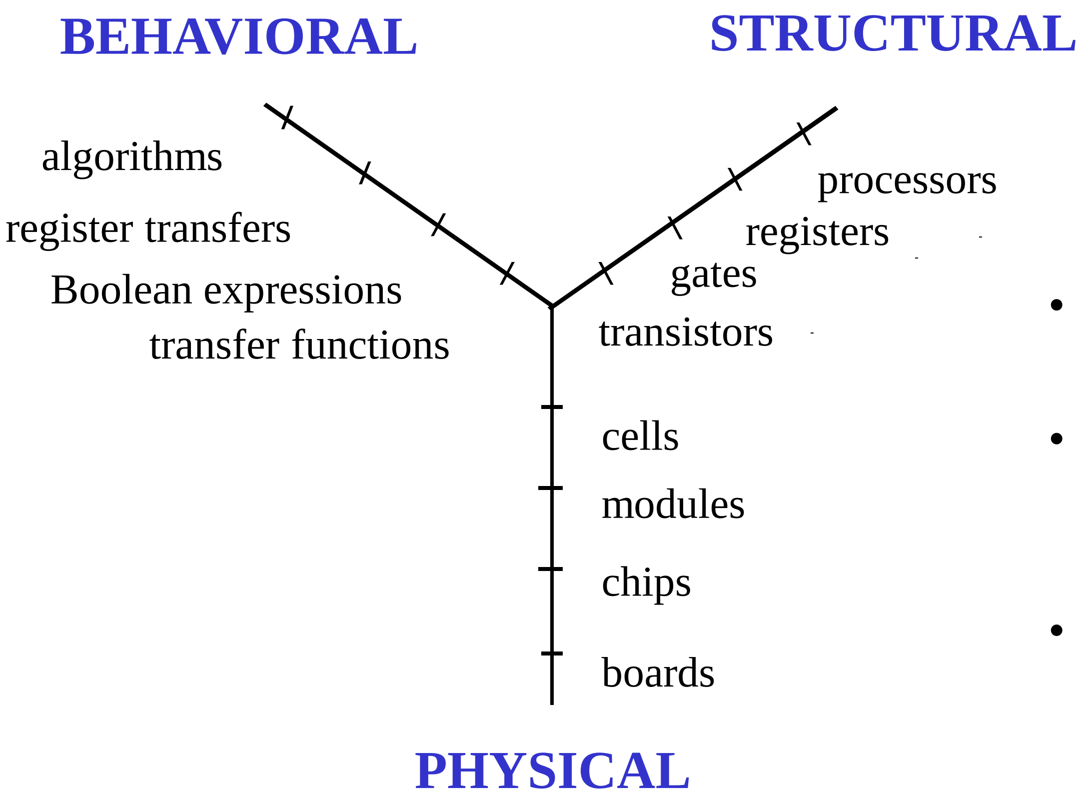
That meet the **original design specification**

i.e. function + performance

while

- Minimizing chip area (die cost)
- Maximizing yield (die cost)
- Minimizing power dissipation (battery life)
- Maximizing reliability (design margin)
- Minimizing test time (product cost)
- Minimizing design cost (**N**on **R**ecurring **E**xpense)
- Minimizing time to market (market share)

Abstraction of Design Space



- Three fundamentally different ways (views) of representing a design
- Each view can be formed at many different levels of abstraction (amount of detail)
- Design process is one of moving from highest behavioral level (specification) to lowest (most detailed) physical level

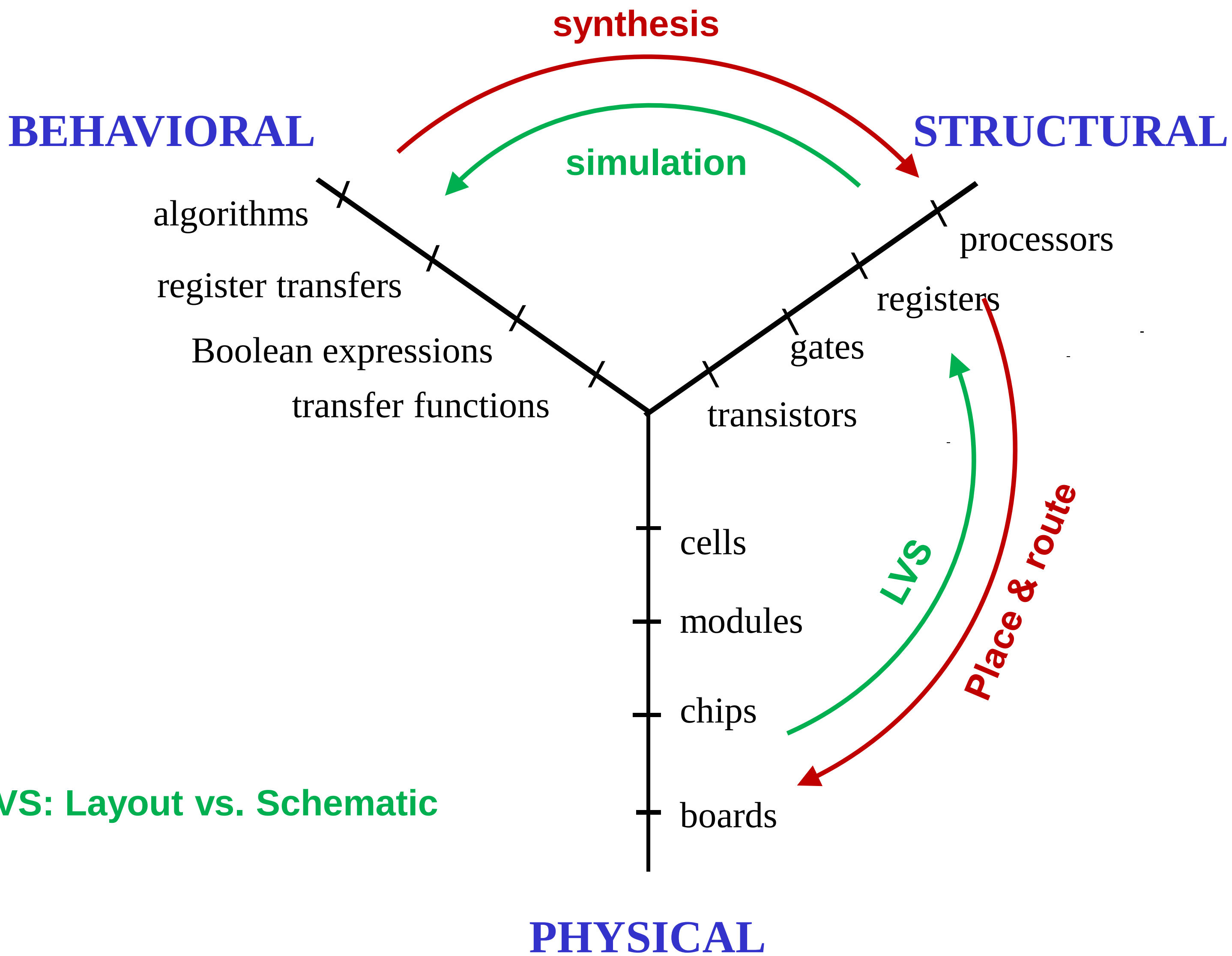
Source: D. Gajski and R. H. Kuhn

Managing Design Process

Taking a complex design from high level behavioral description (spec.) to detailed physical implementation is accomplished using:

- Hierarchy, Modularity & Regularity
  - Break design into manageable pieces
  - Pieces that have well defined functionality and simple interface
  - Pieces that can be re-used elsewhere in the hierarchy
  - Gradually refine design to greater levels of detail
- Set of computer aided design (CAD) tools that
  1. Capture design data (e.g., hardware description languages, text editors, schematic & layout editors)
  2. Translate from one representation to another (e.g., synthesis, component mapping, place & route)
  3. Verify correctness of translation (simulation, timing analysis, design rule check)
- Design Methodology
  - Recipe (or plan) of how to move from one design representation to another, which tools to use and how to rigorously verify each design step

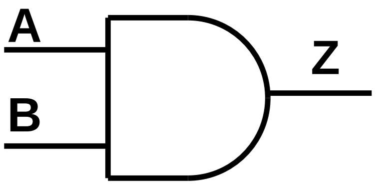
Examples of Computer Aided Design Tools



LVS: Layout vs. Schematic

Digital Design Specification – Boolean Equations

- Function represented by truth tables & logic equations



Z = A.B

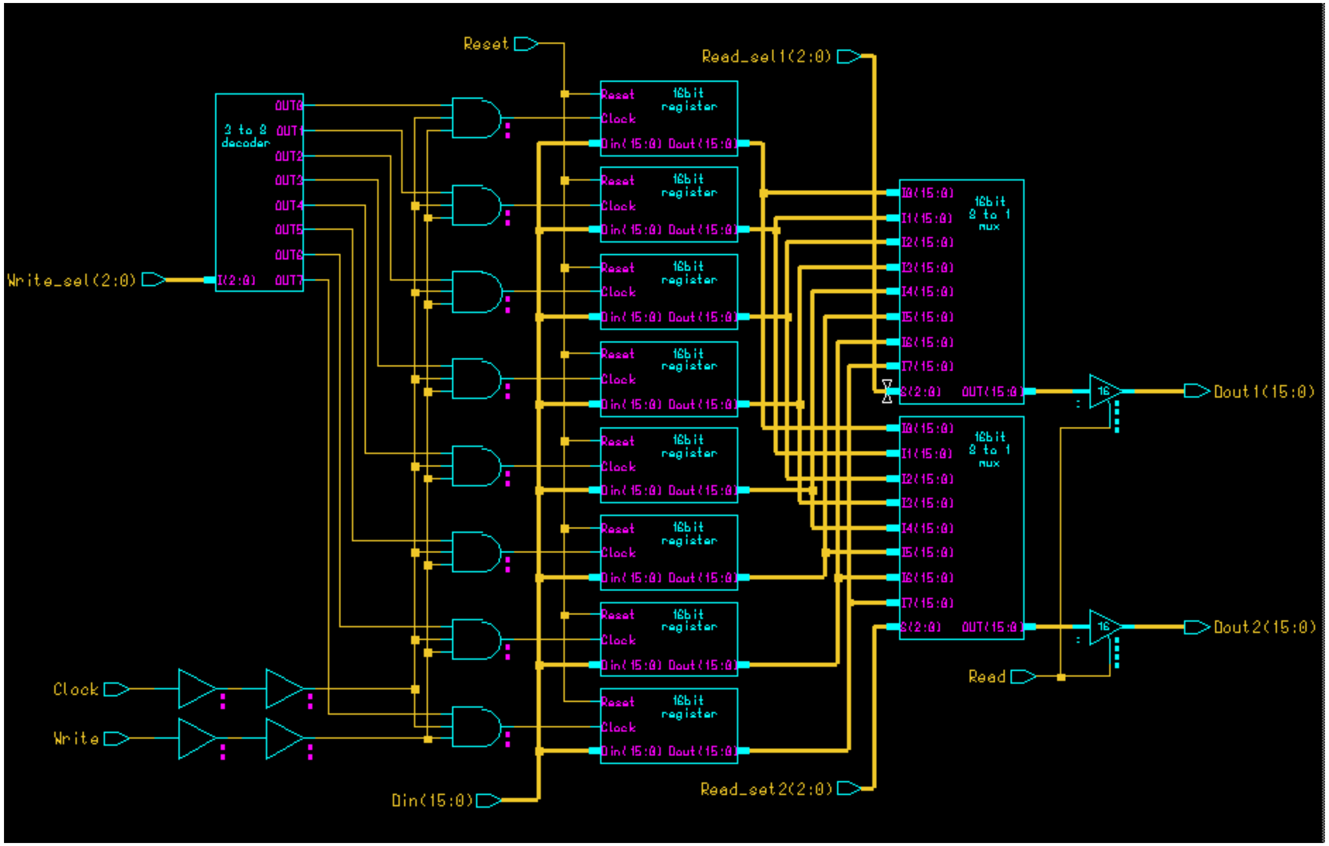
A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

- Function simplified using boolean arithmetic

$$
\begin{aligned}
\boldsymbol{Z} &= \overline{\boldsymbol{B}}.\,(\boldsymbol{A}.\,(\overline{\boldsymbol{B}} + \boldsymbol{C}) + \overline{(\boldsymbol{A} + \boldsymbol{B}.\,\overline{\boldsymbol{C}}}))\\
&= \overline{\boldsymbol{B}}.\,(\boldsymbol{A}.\,(\overline{\boldsymbol{B}} + \boldsymbol{C}) + \overline{\boldsymbol{A}}\,.(\overline{\boldsymbol{B}} + \boldsymbol{C})\\
&= \overline{\boldsymbol{B}}.\,((\boldsymbol{A} + \overline{\boldsymbol{A}}).\,(\overline{\boldsymbol{B}} + \boldsymbol{C}))\\
&= \overline{\boldsymbol{B}}.\,(\overline{\boldsymbol{B}} + \boldsymbol{C})\\
&= \overline{\boldsymbol{B}}
\end{aligned}
$$

- Captures behavior but impractical for more than few hundred gates

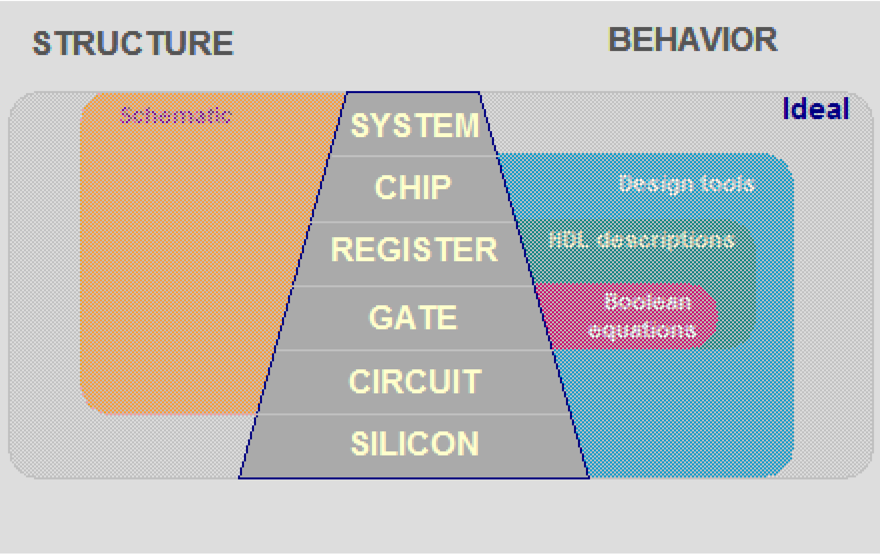
Digital Design Specification – Schematic Capture



- Graphical entry supporting hierarchy, regularity, higher level functions (register, multiplexer, ALU)
- Only captures structure – behavior must be inferred
- Limited to few thousand primitives (gates, registers etc.)

Digital Design – Hardware Description Languages

- Hardware Description Language (HDL) captures behavior and/or structure that can be compiled into simulation or physical implementation (e.g. gate array, FPGA)
- Early Hardware Description languages targeted at Register Transfer or Gate Level behavior



Evita Tutorial

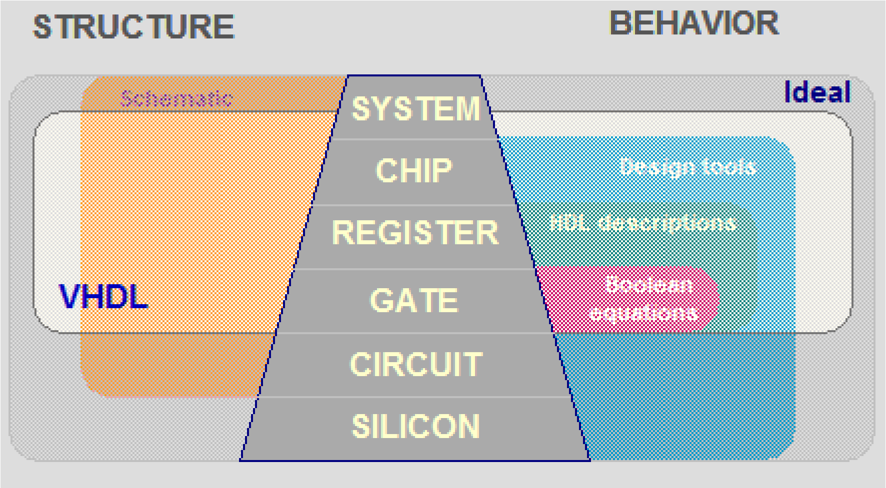
- Different languages used at different levels of abstraction and with different tool vendors leads to lost productivity

Digital Design Specification – VHDL

- **V**HSIC **H**ardware **D**escription **L**anguage

  (VHSIC = Very High Speed Integrated Circuit)

- *Standardized language that can represent behavior and structure at many levels of abstraction*



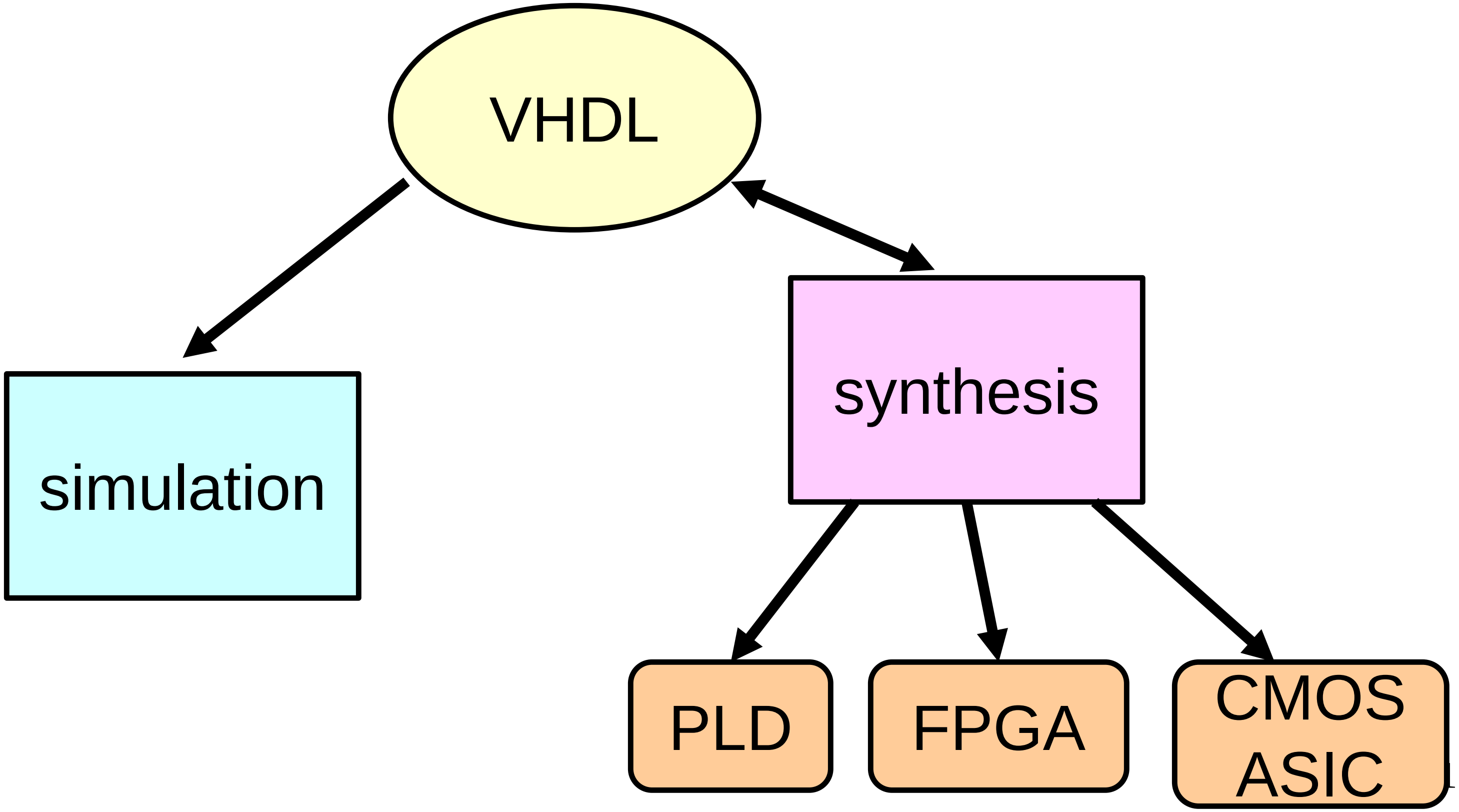
Evita Tutorial

Features of VHDL

- **VHDL can represent:**
  - **behavior** (what the system does) **or**
  - **structure** (how the components are connected) **or**
  - a combination of these.
- **VHDL can be used at different levels of abstraction:**
  - Switch level (switching behavior of transistors)
  - Gate level
  - Register transfer level (registers, multiplexers, alu's etc.)
  - High level architecture (e.g. functional behavior of microprocessor)
- **Technology independent** (ASIC, FPGA, PCB)
- **IEEE Standard** (Interoperability across tool vendors)
- **Provides executable design documentation**

Simulation & Synthesis

- VHDL "program" can be used to drive:
  - **simulation** (functional verification, performance)
  - **synthesis** (translating behavior into physical structure) **or**
  - a combination of these.



VHDL vs. Regular Programming Language

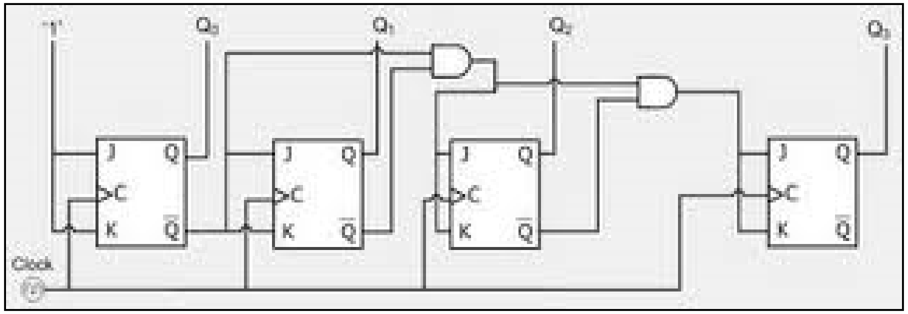
- Procedural programming languages implement an algorithm or recipe
  - for computation & data manipulation
  - essentially single sequential thread (Program Counter)
    - order of statements determines execution sequence
  - no intrinsic concept of time
  - program operates on variables
- VHDL describes a hardware system
  - from different points of view: behavior, structure, dataflow
  - can model highly concurrent operation
  - intrinsic concept of time
    - timed events determines execution sequence
  - program operates on variables and signals

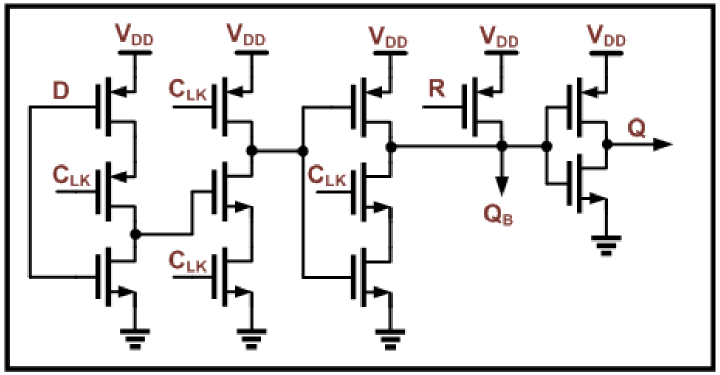
History of VHDL

- Launched in 1980 by Defense Advanced Research Projects Agency (DARPA)
- July 1983
  - Intermetrics, IBM and Texas Instruments were awarded a contract to develop VHDL
- August 1985
  - Release of final version of the language under government contract, VHDL Version 7.2
- December 1987
  - IEEE Standard 1076-1987
- 1988
  - VHDL became an American National Standards Institute (ANSI) standard
- September 1993
  - IEEE VHDL standard revised

Nature of Digital Systems

- *[At all levels of abstraction](#)*, electronic systems are composed of sub-systems interconnected by signals*:





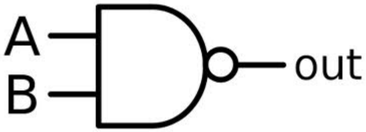
* Signals include wires, optical links & wireless links

VHDL Model of Digital Systems

- *At all levels of abstraction*:
- VHDL names and declares the interface to each (sub-)system using a programming abstraction known as an **entity**.
- VHDL models the operation (i.e. the behavior or the internal structure) of a (sub-)system using an abstraction known as an **architecture**.
- VHDL describes the (timed) information flow between (sub-)systems using an abstraction known as a **signal**.

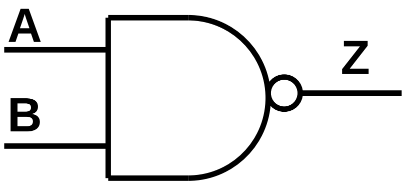
VHDL Entity

- An entity describes ***the*** external view of a system
- An entity specifies:
  - Name of the system
  - Parameters of the system *(get to this later)*
  - Connections to the system (external signals)



- Each of these could be an entity

Entity Example – 2-input NAND gate



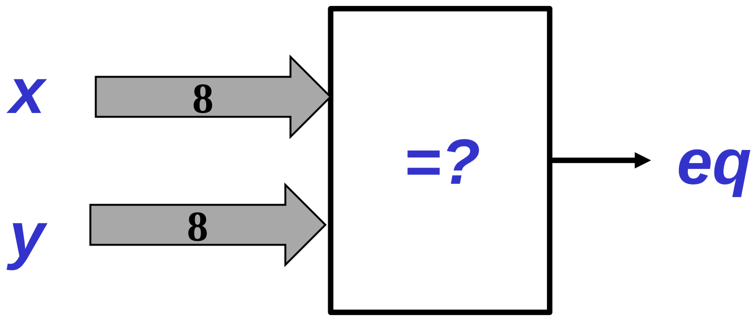
name

external connections

entity nand2 **is**
port(a,b: **in bit**;
        z: **out bit**);
end entity nand2;

Note: words in **bold** are reserved keywords

Entity Example – 8bit comparator



name

entity compare **is**

external connections

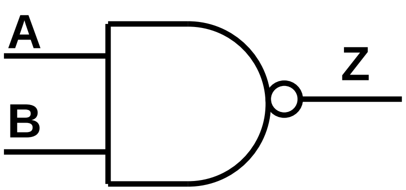
port(x,y: **in bit_vector**(7 **downto** 0);
      eq: **out bit**);

end entity compare;

VHDL Architecture

- An architecture describes *an* internal view of a system
- An architecture describes the behavior or internal structure of a declared entity
- There may be many architectures associated with each entity e.g:
  - structure vs behavior,
  - different levels of abstraction (gate vs RTL)
  - different timing constraints
- There is exactly one entity for each architecture

Architecture Example – 2-input NAND gate



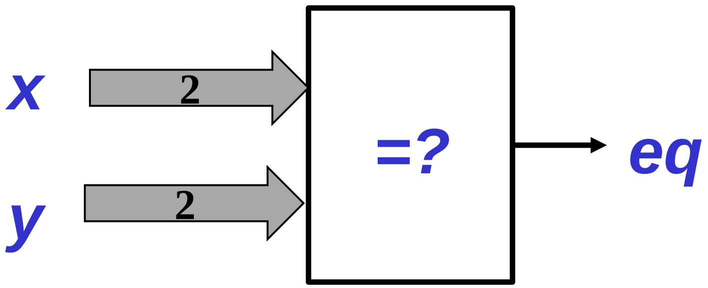
entity nand2 **is**
port(a,b: **in bit**;
        z: **out bit**);
end entity nand2;

architecture name →

architecture ngate **of** nand2 **is**
begin
        z <= **not**(a **and** b) **after** 5 ns;
end architecture ngate;

behavior

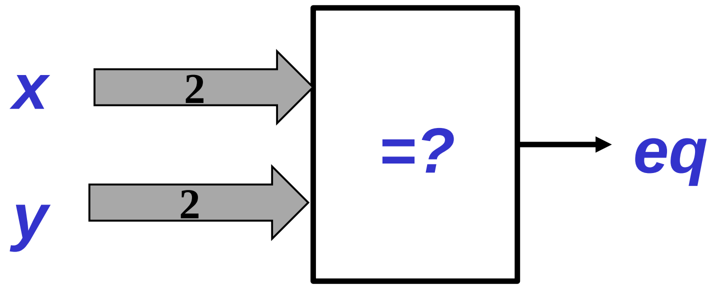
Architecture Example – 2 bit comparator




```
entity compare is
port(x,y: in bit_vector(1 downto 0);
      eq: out bit);
end entity compare;

architecture cmp2 of compare is
begin
   eq <= '1' after 3 ns when (x=y) else
   '0' after 3ns;
end architecture cmp2;
```

Alternative Architecture – 2 bit comparator

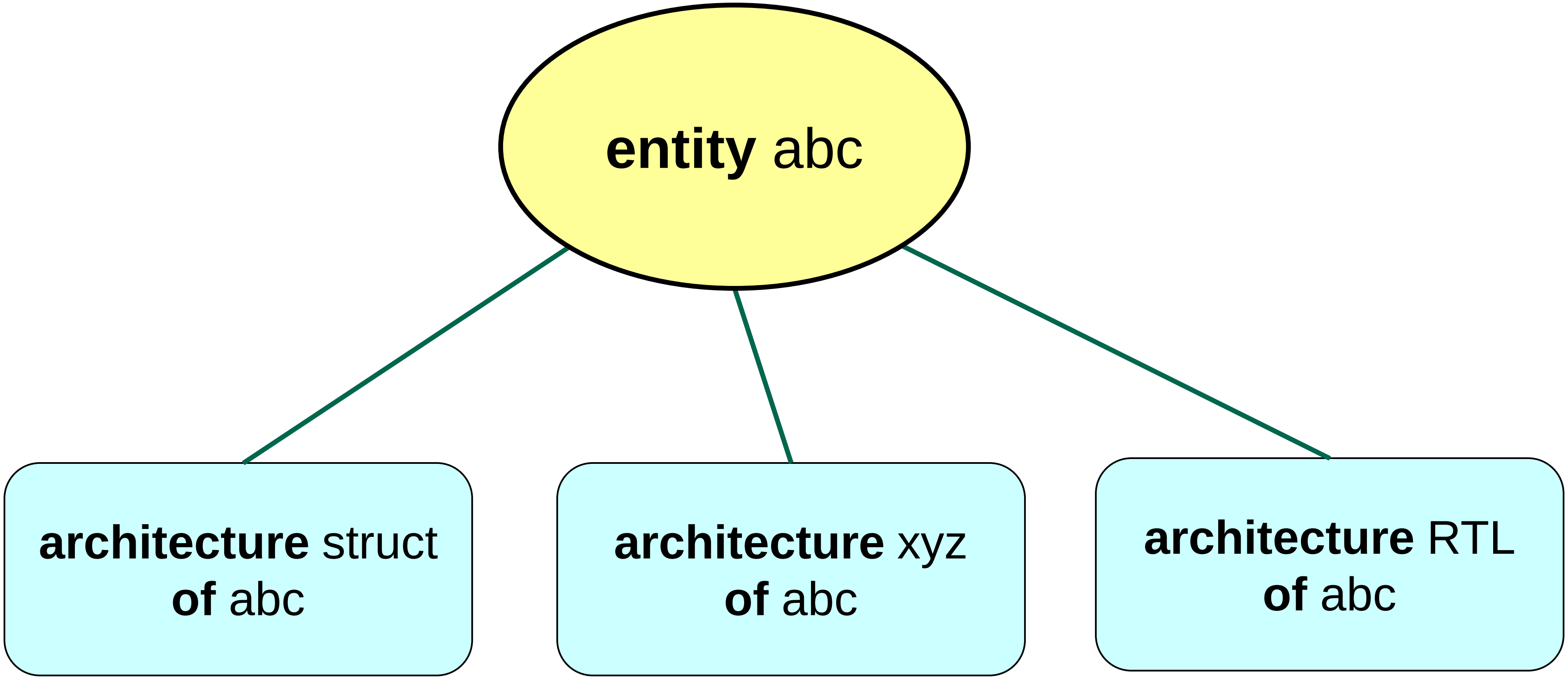



```
entity compare is
port(x,y: in bit_vector(1 downto 0);
      eq: out bit);
end entity compare;

architecture cmp_alt of compare is
signal a1, a0: bit;
begin
   a1 <=  x(1) xor y(1) after 2 ns;
   a0 <=  x(0) xor y(0) after 2 ns;
   eq <=  a0 nor a1 after 3 ns;
end architecture cmp_alt;
```

hints at implementation

One Entity – Multiple Architectures



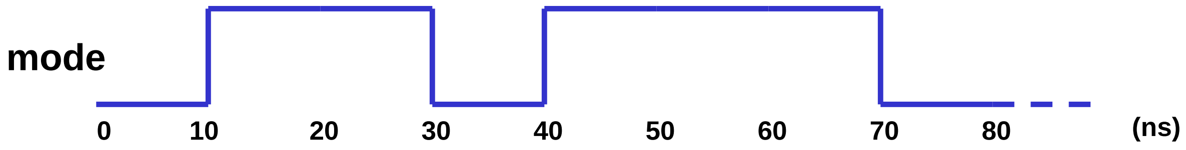
- Different architectures may describe different representations (behavior, structure) of entity at different levels of abstraction

VHDL - Signal

- Like conventional programming languages VHDL manipulates basic objects such as constants and variables.
- VHDL introduces a new class of object: **signal**
- Signal is a sequence of value-time pairs
- A signal will be assigned a value at a specific time
  - It will retain that value until a new value is assigned at a future point in time.

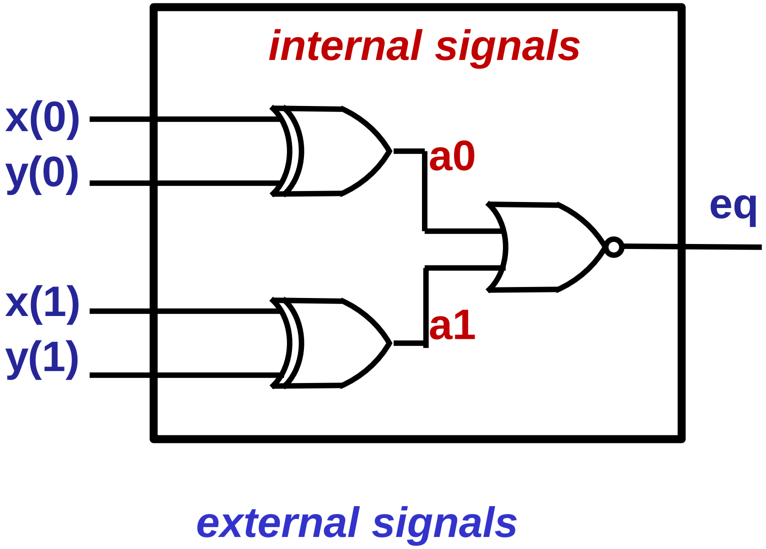
signal mode**: bit;**

mode<=‘0’,‘1’ **after** 10ns,‘0’ **after** 30ns,‘1’ **after** 40ns,‘0’ **after** 70ns;



External & Internal Signals

- Ports are external signals, visible inside & outside the system
- Signals declared in an architecture are internal signals
  - manipulated by programming constructs within the architecture
  - not visible outside the system




```
entity compare is
port(x,y: in bit_vector(1 downto 0);
     eq: out bit);
end entity compare;

architecture cmp_alt of compare is
signal a1, a0: bit;
begin
   a1 <=  x(1) xor y(1) after 2 ns;
   a0 <=  x(0) xor y(0) after 2 ns;
   eq <=  a0 nor a1 after 3 ns;
end architecture cmp_alt;
```

Signals Assignment Statement

- Signal assignment statement assigns a new value to a signal at a specified (future) time

```
abc <= y2 and mode after 2 ns;
```

- *signal to be updated* → `abc`
- *signal assignment operator* → `<=`
- *new value to be assigned* → `y2 and mode`
- *when new value should be assigned* → `after 2 ns`

- Signal assignment statements can be concurrent or sequential
  - describes when they are executed (more on this later)
- If assignment time is not specified, defaults to "**after** 0 ns"

Signal Types

- Like variables, all signals are typed e.g.:
  - boolean *(false, true)*
  - integer *(…,-3,-2,-1,0,1,2,3,…)*
  - character *(…,'A','B','C',…)*
  - bit *('0','1')*
- In digital circuits, often much more convenient to represent a signal as a bus, rather then individual bits.
- Type bit_vector is an array of type bit
  - bit positions are numbered left (msb) to right (lsb)

```
signal abus: bit_vector (0 to 7);        -- 8-bit bus, abus(0) is msb
signal instr: bit_vector (15 downto 0);  -- 16-bit bus, instr(15) is msb
signal opcode: bit_vector (6 downto 3);  -- 4-bit bus, opcode(6) is msb

opcode <= instr(12 downto 9);            -- bit_vector used in assignment statement
```

Sidebar: VHDL - Packages

- Standard VHDL has limited set of types, operators, functions.
- This set can be expanded through the use of packages.
- Packages contain definitions of types, functions & procedures that can be shared by multiple designers.
- A very popular package is IEEE std_logic_1164

```
library IEEE;                          -- IEEE is a library of packages
use IEEE.std_logic_1164.all;           -- load this package from IEEE library

entity half_adder is                   --  half_adder can use std_logic_1164
    •
    •
```

Std_logic type

- Standard bit type can only take on values ‘0’ or ‘1’
- In logic simulation, we often require a richer set of values
- IEEE std_logic type can take on 9 different values:
  - ‘U’ Uninitialized
  - ‘X’ Forcing unknown
  - ‘0’ Forcing 0
  - ‘1’ Forcing 1
  - ‘Z’ High impedance
  - ‘W’ Weak unknown
  - ‘L’ Weak 0
  - ‘H’ Weak 1
  - ‘-’ Don’t care

Using std_logic type

```
library IEEE;
use IEEE.std_logic_1164.all;

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

- Like bit, std_logic has vector extension e.g.:

```
signal addr: std_logic_vector (0 to 7);

signal dataout: std_logic_vector (31 downto 16);
```

More on Types

- VHDL is a strongly typed language
- Every object (signal, variable, constant) has a type
- The type determines which operations can be applied to the object
- In assignment statements, LHS target must be of same type as the RHS expression
  - Type conversion functions (often defined in packages) allow explicit casting from type to another:

```
signal addr: std_logic_vector (0 to 7);
signal index: integer;
begin
    addr <= conv_std_logic_vector(index,8);
```

Some Scalar Types

- Numeric types:
  - **type** integer **is range** -2147483647 **to** +21474843647;
  - **subtype** Positive **is** integer **range** 1 **to** +21474843647;
  - **type** real_voltage **is range** 0.0 to 3.3;
- Enumerated types - explicitly listed set of allowable values
  - **type** BOOLEAN **is** (FALSE, TRUE);
  - **type** BIT **is** (‘0’,’1’);
  - **type** COLOR **is** (red, orange, yellow, green, blue, indigo, violet);
  - **type** STD_ULOGIC is (‘U’,’X’,’0’,’1’,’Z’,’W’,’L’,’H’,’-’);
- Default value is “left-hand-side” of range of object’s type
- Uninitialized objects take on default value at start (t=0) of simulation

Array Types

- An array is a collection of one or more objects of the same type
- For example, *std_logic_vector* is an array of objects of type *std_logic*
- More examples:

```
type data_16 is array (15 downto 0) of std_logic;
type reg_file is array (31 downto 0) of data_16;
signal abc: reg_file;
```

note that: abc (30) is of type data_16

whereas: abc (30) (5) is of type std_logic

```
type row_4  is array (1 to 4) of integer;
type matrix_4x4  is array (1 to 4) of row_4;
```

- Note: If an array is indexed by signal (or variable), the index must be of type integer

Assignments to Arrays

- VHDL provides many different ways to assign values to array objects:

```
signal abc: std_logic_vector (1 to 5);
signal xx, yy: std_logic;

abc <= “01001”;                              -- string literal
abc <= (‘0’, ‘1’, ‘0’, ‘0’, ‘1’);            -- positional
abc <= (2=>’1’, 5=>’1’, others => ‘0’)       -- named

abc <= (others => ’0’) ;                     -- sets all bits to ‘0’

abc <= (1=> xx, 4=> ‘0’, others => yy);      -- other signals
```