Introduction to Digital System

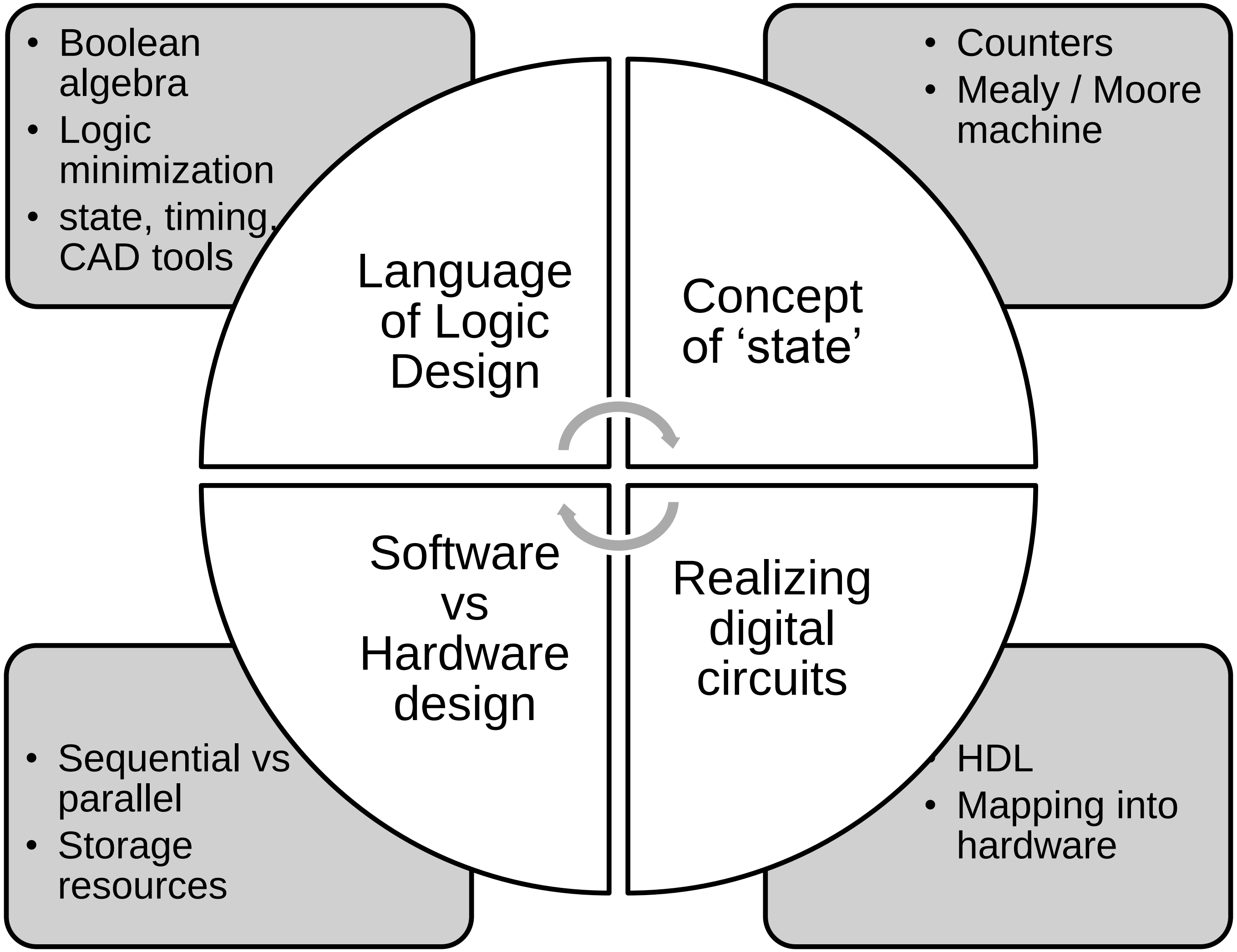
Chapter 1

Why study logic design?

- Obvious reasons
  - this course is part of the EAC requirements
  - it is the implementation basis for all modern computing devices
    - building large things from small components
    - provide a model of how a computer works
- More important reasons
  - the inherent parallelism in hardware is often our first exposure to parallel computation
  - it offers an interesting counterpoint to software design and is therefore
    useful in furthering our understanding of computation, in general

An Era of Technology…

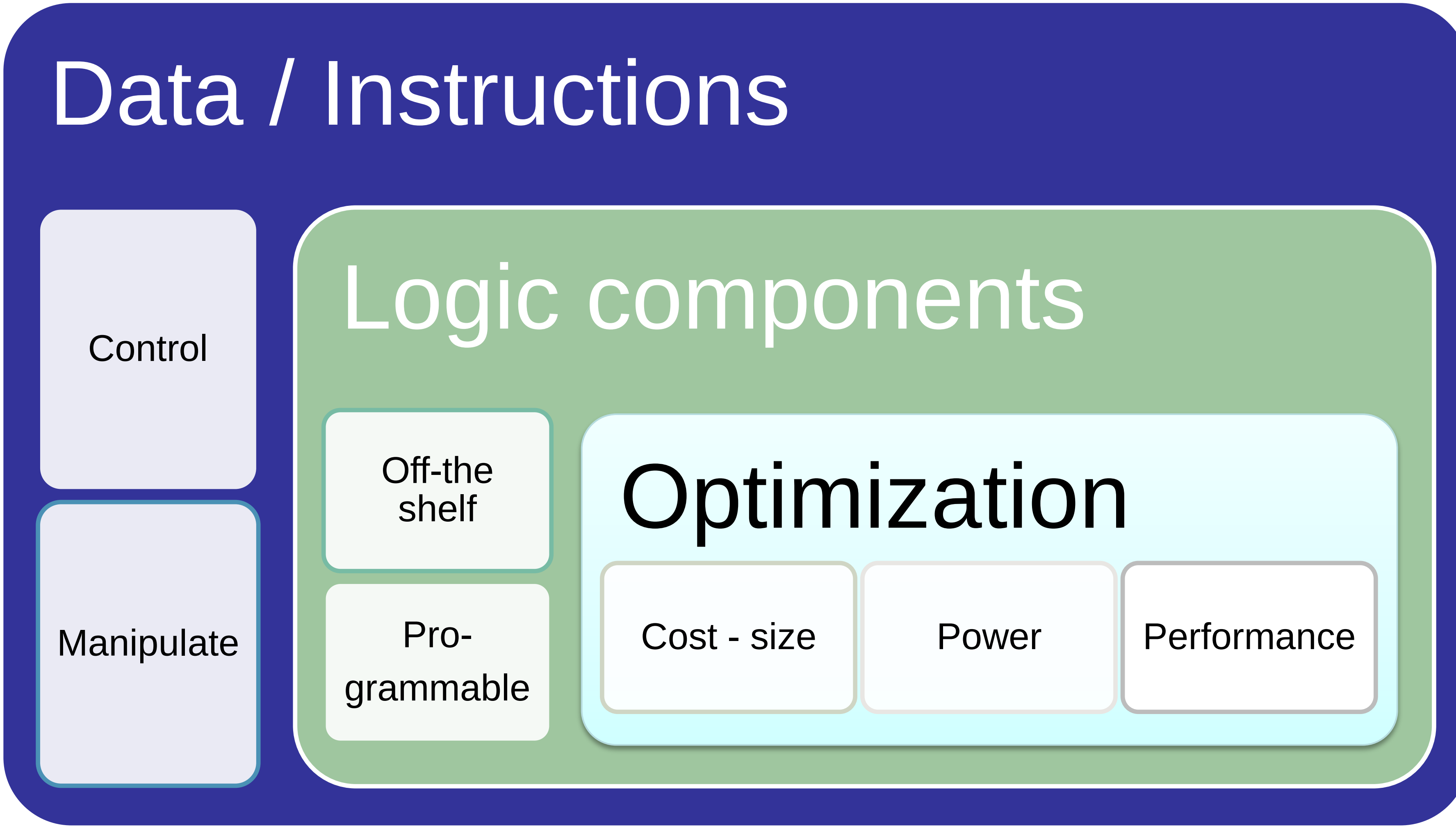
What will we learn in this class?



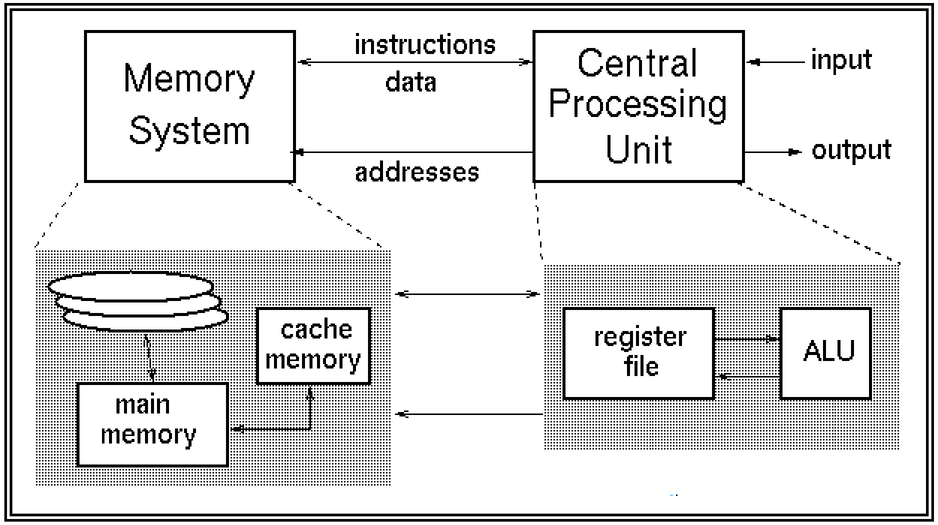
Applications of logic design

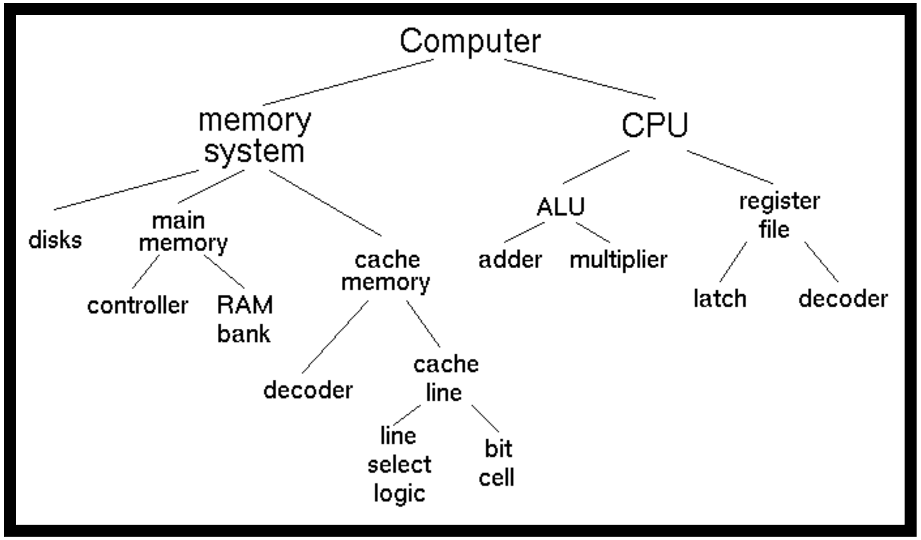
- Conventional computer design
  – CPUs, busses, peripherals
- Networking and communications
  – phones, modems, routers
- Embedded products
  – in cars, toys, appliances, entertainment devices
- Scientific equipment
  – testing, sensing, reporting
- The world of computing is much much bigger than just PCs!

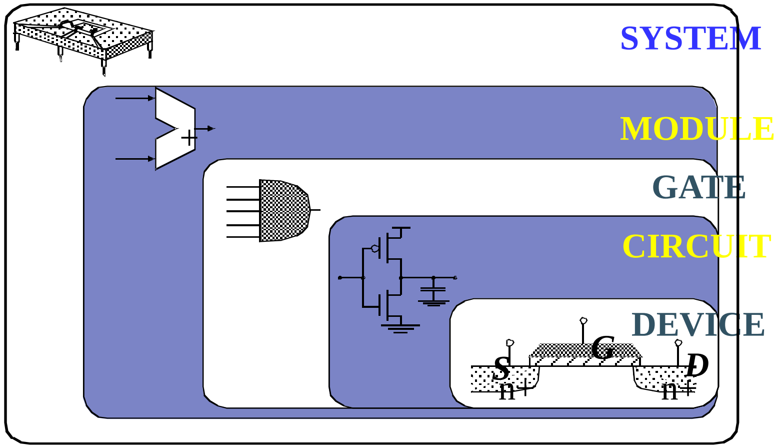
What is logic design?



Design Representation







Hierarchy in Designs

1

Bottom –up

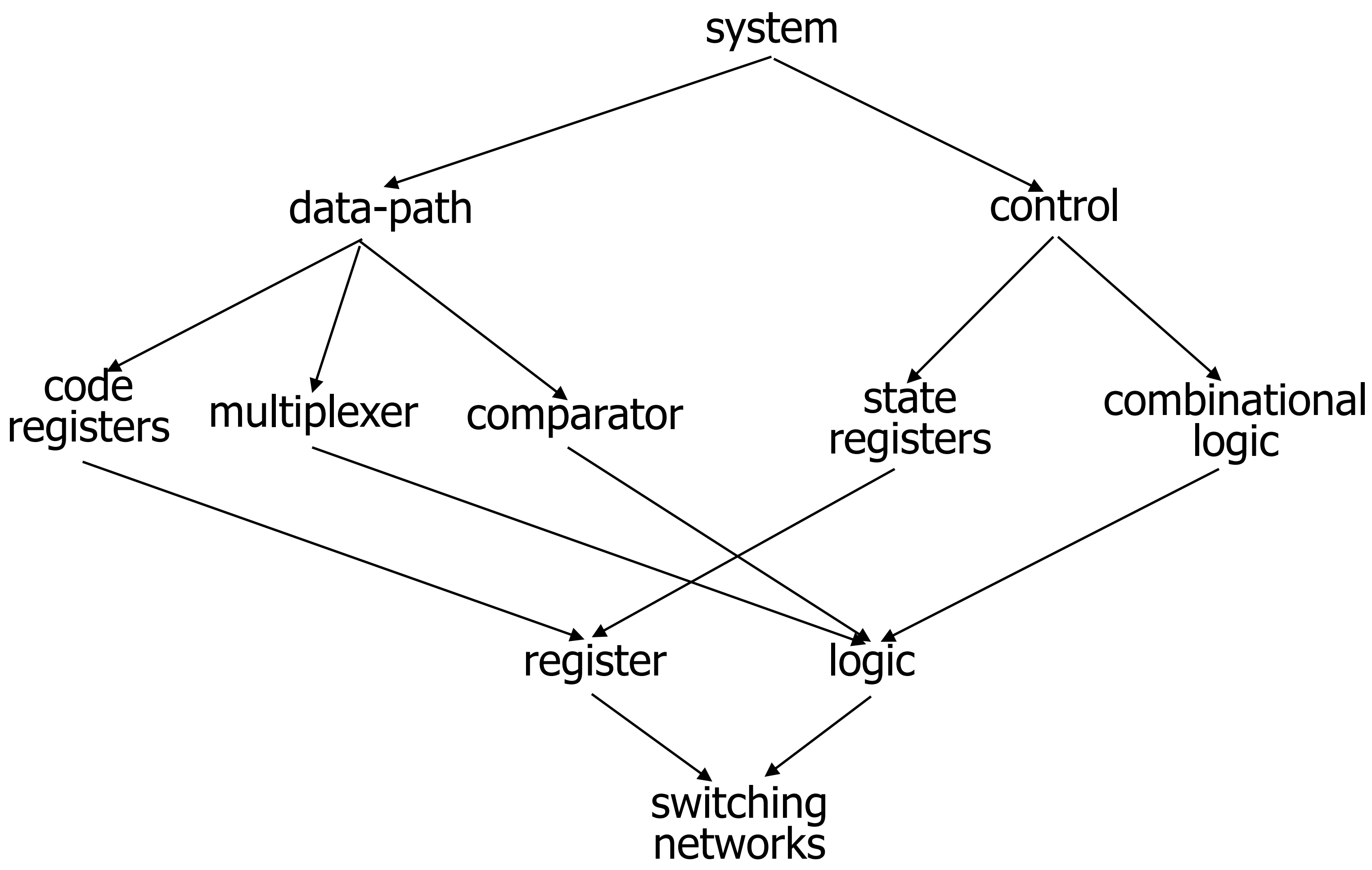
- Start at leaves and put pieces together to build up design

2

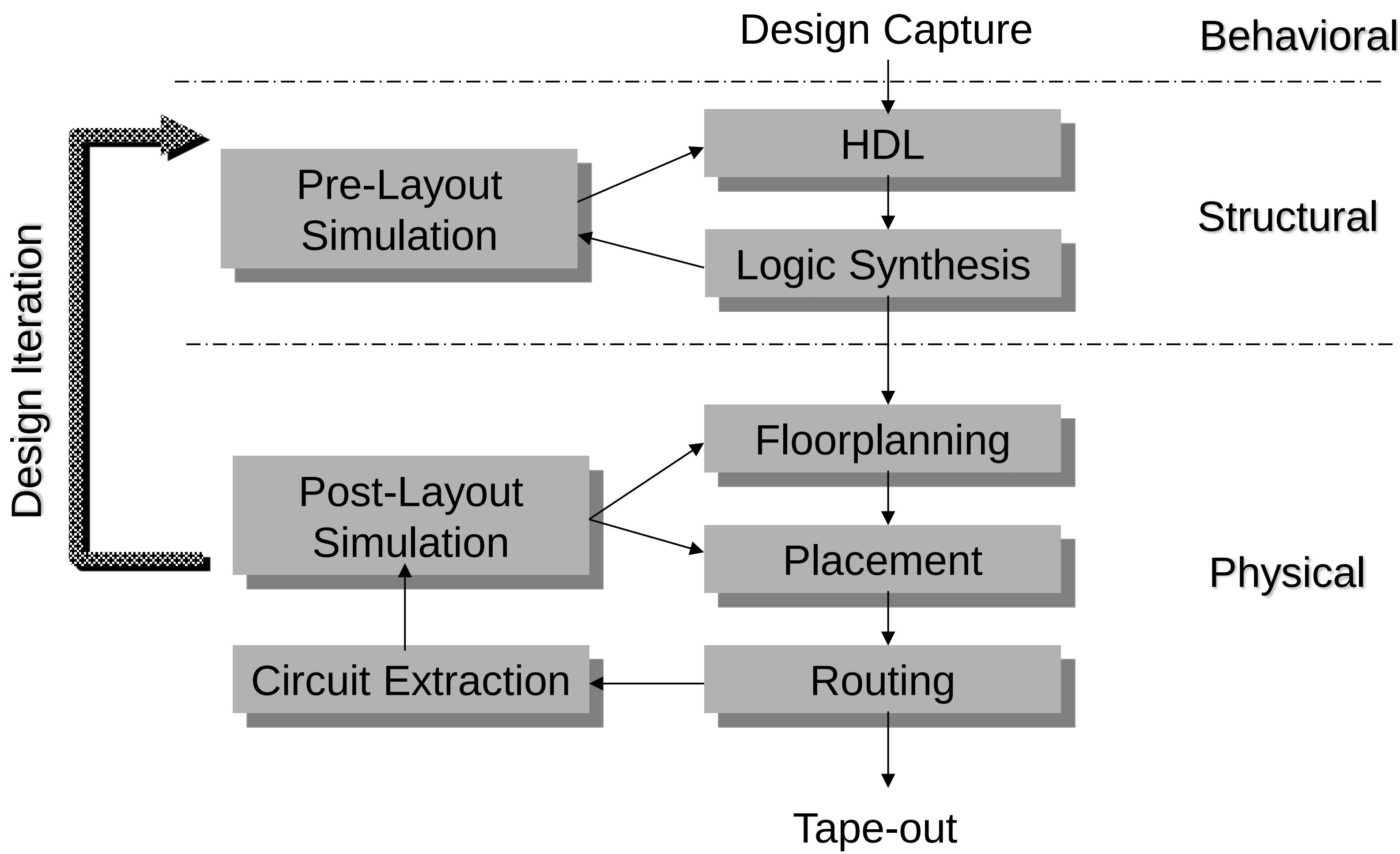
Top-Down

- Start at top and works down by successive refinement

Design hierarchy



Digital Design Flow



NurulHazlina/BEE2243/Intro

BEE2243 (concepts/skills/abilities)

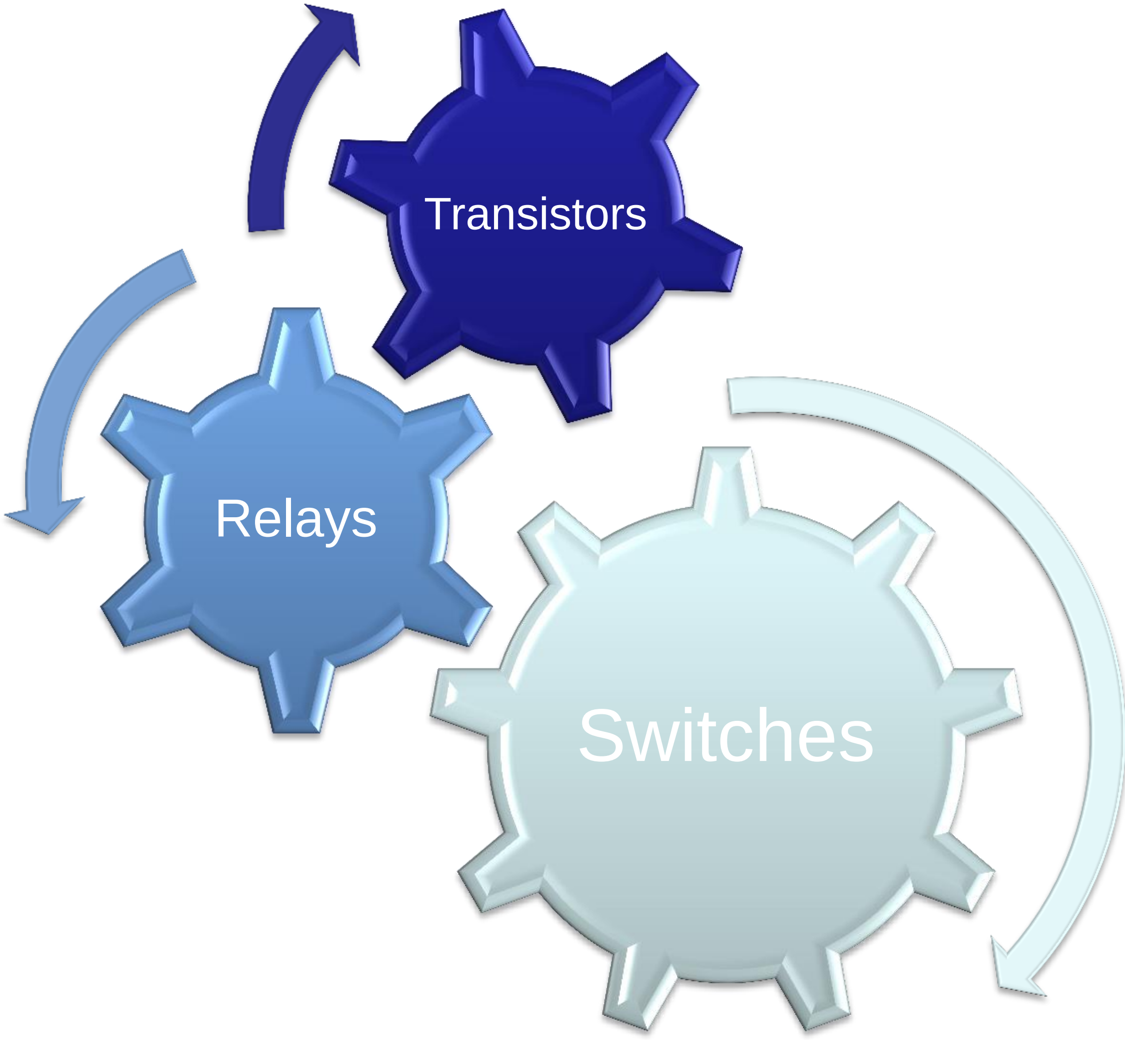
- Understanding the basics of logic design (concepts)
- Understanding sound design methodologies (concepts)
- Modern specification methods (concepts)
- Familiarity with a full set of CAD tools (skills)
- Realize digital designs in an implementation technology (skills)
- Appreciation for the differences and similarities (abilities) in hardware and software design

New ability: to accomplish the logic design task with the aid of computer-aided design tools and map a problem description into an implementation with programmable logic devices after validation via simulation and understanding of the advantages/disadvantages as compared to a software implementation

Computation: abstract vs. implementation

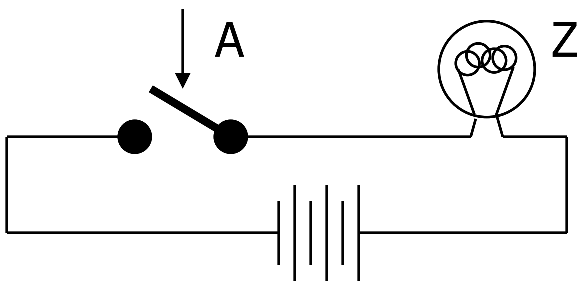
- Up to now, computation has been a mental exercise (paper, programs)
- This class is about physically implementing computation using physical devices that use voltages to represent logical values
- Basic units of computation are:
  - representation: "0", "1" on a wire
    set of wires (e.g., for binary ints)
  - assignment: x = y
  - data operations: x + y – 5
  - control:
    sequential statements: A; B; C
    conditionals: if x == 1 then y
    loops: for (i = 1 ; i == 10, i++)
    procedures: A; proc(...); B;
- We will study how each of these are implemented in hardware and composed into computational structures

Digital Networks

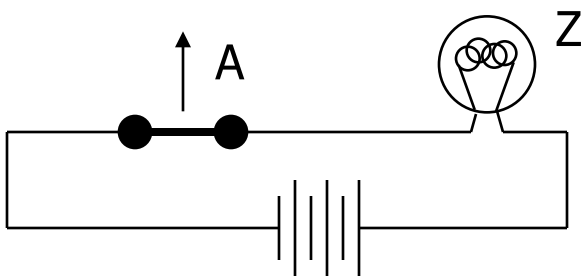


Switches: basic element of physical implementations

- Implementing a simple circuit (arrow shows action if wire changes to "1"):



close switch (if A is "1" or asserted)
and turn on light bulb (Z)



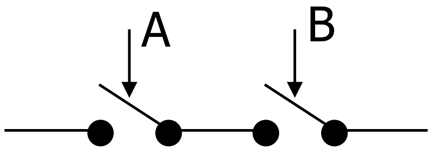
open switch (if A is "0" or unasserted)
and turn off light bulb (Z)

$$Z \equiv A$$

Switches (cont'd)

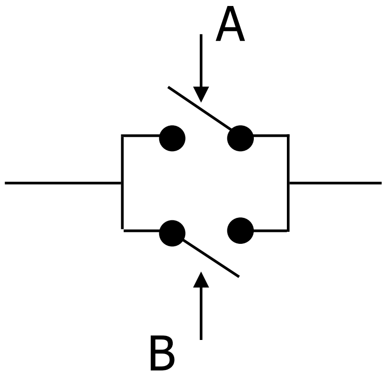
- Compose switches into more complex ones (Boolean functions):

AND



$Z \equiv$ A and B

OR



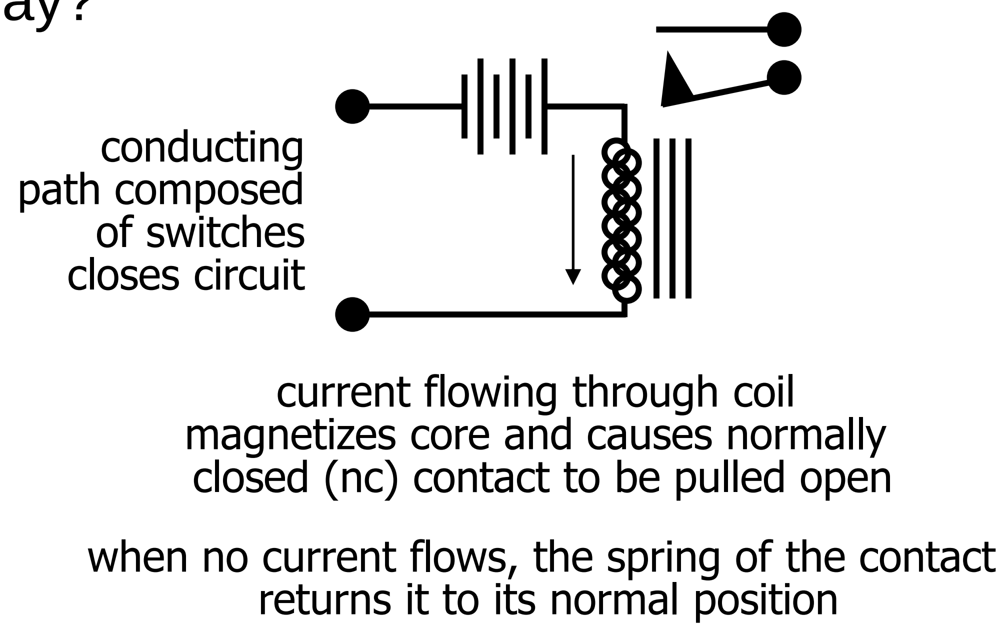
$Z \equiv$ A or B

Switching networks

- Switch settings
  - determine whether or not a conducting path exists to light the light bulb
- To build larger computations
  - use a light bulb (output of the network) to set other switches (inputs to another network).
- Connect together switching networks
  - to construct larger switching networks, i.e., there is a way to connect outputs of one network to the inputs of the next.

Relay networks

- A simple way to convert between conducting paths and switch settings is to use (electro-mechanical) relays.
- What is a relay?



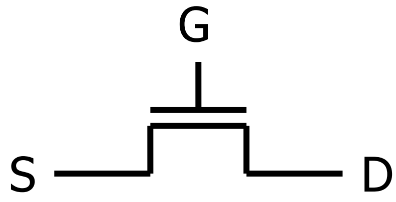
\What determines the switching speed of a relay network?

Transistor networks

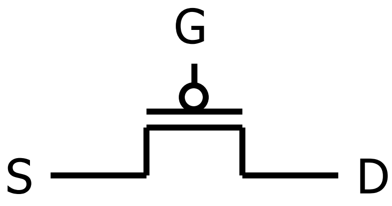
- Relays aren't used much anymore
  - some traffic light controllers are still electro-mechanical
- Modern digital systems are designed in CMOS technology
  - MOS stands for Metal-Oxide on Semiconductor
  - C is for complementary because there are both normally-open and normally-closed switches
- MOS transistors act as voltage-controlled switches
  - similar, though easier to work with than relays.

MOS transistors

- MOS transistors have three terminals: drain, gate, and source
  - they act as switches in the following way: if the voltage on the gate terminal is (some amount) higher/lower than the source terminal then a conducting path will be established between the drain and source terminals

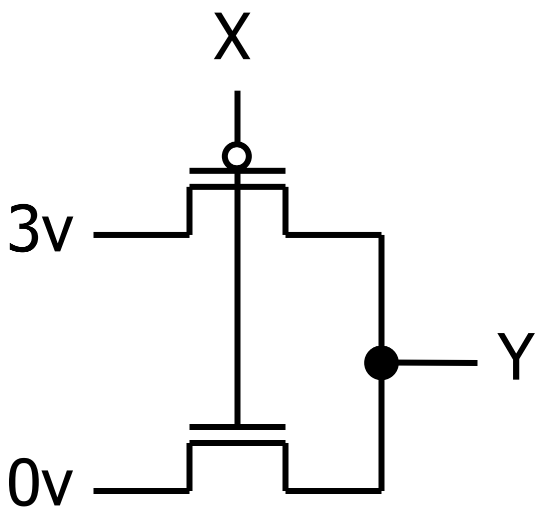


n-channel
open when voltage at G is low
closes when:
voltage(G) > voltage (S) + ε



p-channel
closed when voltage at G is low
opens when:
voltage(G) < voltage (S) – ε

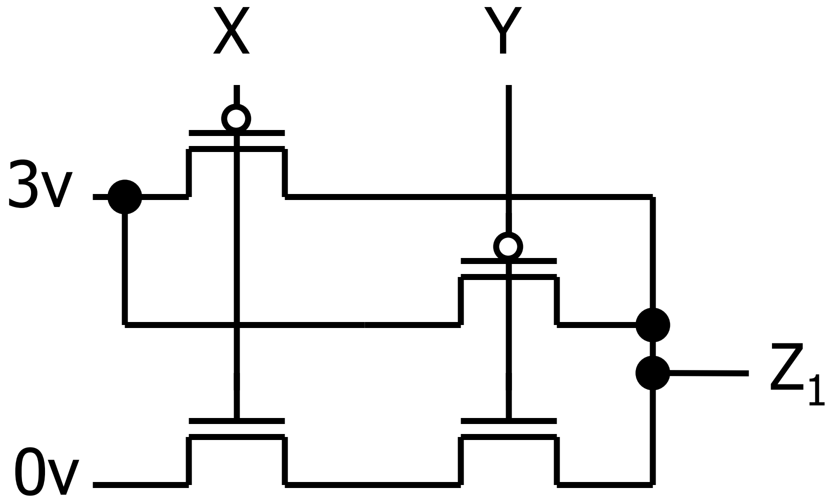
MOS networks

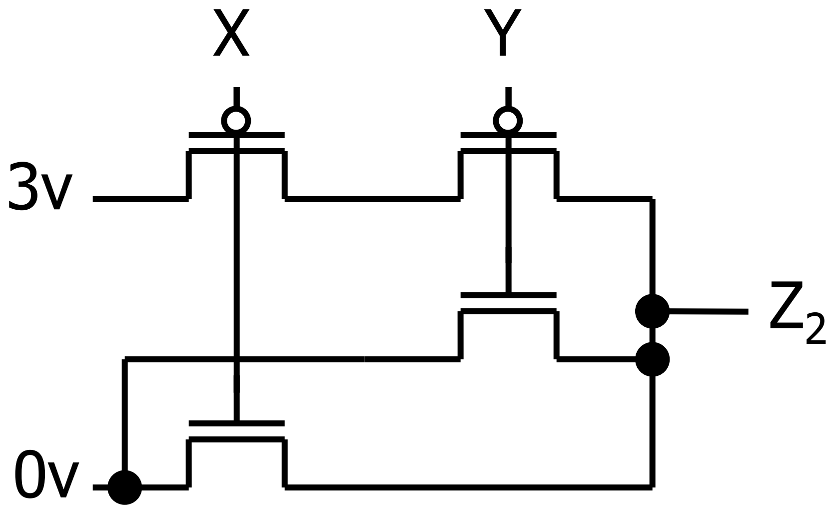


what is the relationship between x and y?

x	y
0 volts	
3 volts	

Two input networks





what is the relationship between x, y and z?

x	y	z1	z2
0 volts	0 volts		
0 volts	3 volts		
3 volts	0 volts		
3 volts	3 volts		

Speed of MOS networks

- What influences the speed of CMOS networks?
  - charging and discharging of voltages on wires and gates of transistors
- Capacitors hold charge
  - capacitance is at gates of transistors and wire material
- Resistors slow movement of electrons
  - resistance mostly due to transistors

Representation of digital designs

- Physical devices (transistors, relays)
- Switches
- Truth tables
- Boolean algebra
- Gates
- Waveforms
- Finite state behavior
- Register-transfer behavior
- Concurrent abstract specifications

scope of BEE2243 (Switches through Register-transfer behavior)

Digital vs. analog

- Convenient to think of digital systems as having only discrete, digital, input/output values
- In reality, real electronic components exhibit continuous, analog, behavior

- Why do we make the digital abstraction anyway?
  - switches operate this way
  - easier to think about a small number of discrete values
- Why does it work?
  - does not propagate small errors in values
  - always resets to 0 or 1

Mapping from physical world to binary world

Technology	State 0	State 1
Relay logic	Circuit Open	Circuit Closed
CMOS logic	0.0-1.0 volts	2.0-3.0 volts
Transistor transistor logic (TTL)	0.0-0.8 volts	2.0-5.0 volts
Fiber Optics	Light off	Light on
Dynamic RAM	Discharged capacitor	Charged capacitor
Nonvolatile memory (erasable)	Trapped electrons	No trapped electrons
Programmable ROM	Fuse blown	Fuse intact
Bubble memory	No magnetic bubble	Bubble present
Magnetic disk	No flux reversal	Flux reversal
Compact disc	No pit	Pit

Digital Circuits

Combinational Circuits

"memory-less"

its output values only depend on its input values

Sequential Circuits

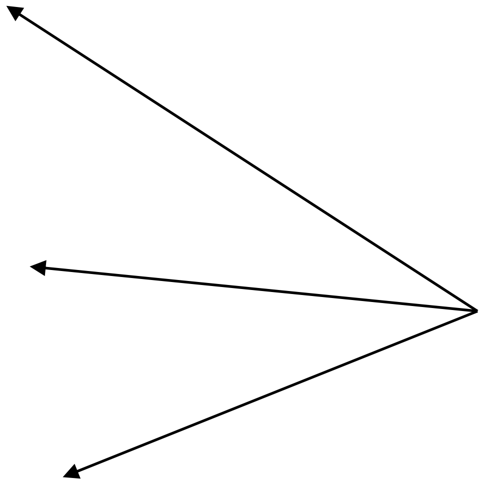
exhibit behaviors (output values) that depend not only on the current input values, but also on previous input values

Combinational logic symbols

- Common combinational logic systems have standard symbols called logic gates
  - Buffer, NOT

  - AND, NAND

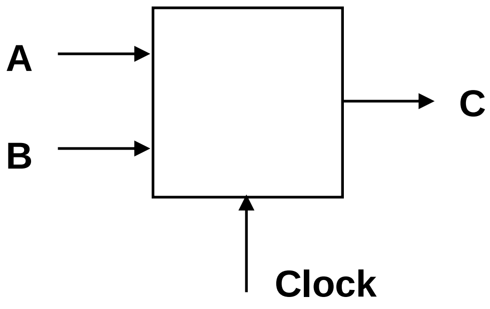
  - OR, NOR



easy to implement with CMOS transistors (the switches we have available and use most)

Example of combinational and sequential logic

- Combinational:
  - input A, B
  - wait for clock edge
  - observe C
  - wait for another clock edge
  - observe C again: will stay the same
- Sequential:
  - input A, B
  - wait for clock edge
  - observe C
  - wait for another clock edge
  - observe C again: may be different



Abstractions

- Some we've seen already
  - digital interpretation of analog values
  - transistors as switches
  - switches as logic gates
  - use of a clock to realize a synchronous sequential circuit
- Some others we will see
  - truth tables and Boolean algebra to represent combinational logic
  - encoding of signals with more than two logical values into binary form
  - state diagrams to represent sequential logic
  - hardware description languages to represent digital logic
  - waveforms to represent temporal behavior

Example – Calendar Subsystem

- Calendar subsystem: number of days in a month (to control watch display)

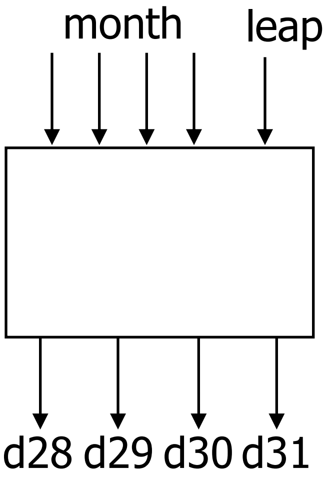
  - used in controlling the display of a wrist-watch LCD screen
  - inputs: month, leap year flag
  - outputs: number of days

Implementation in software

```
integer number_of_days ( month, leap_year_flag) {
    switch (month) {
        case 1: return (31);
        case 2: if (leap_year_flag == 1) then return (29)
                                          else return (28);
        case 3: return (31);
        ...
        case 12: return (31);
        default: return (0);
    }
}
```

Implementation as a combinational digital system

- Encoding:
  - how many bits for each input/output?
  - binary number for month
  - four wires for 28, 29, 30, and 31
- Behavior:
  - combinational
  - truth table specification



month	leap	d28	d29	d30	d31
0000	–	–	–	–	–
0001	–	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	–	0	0	0	1
0100	–	0	0	1	0
0101	–	0	0	0	1
0110	–	0	0	1	0
0111	–	0	0	0	1
1000	–	0	0	0	1
1001	–	0	0	1	0
1010	–	0	0	0	1
1011	–	0	0	1	0
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–

Combinational example (cont’d)

- Truth-table to logic to switches to gates
  - d28 = 1 when month=0010 and leap=0
  - d28 = m8'•m4'•m2•m1'•leap'
  - d31 = 1 when month=0001 or month=0011 or ... month=1100
  - d31 = (m8'•m4'•m2'•m1) + (m8'•m4'•m2•m1) + ... (m8•m4•m2'•m1')
  - d31 = can we simplify more?

symbol for not

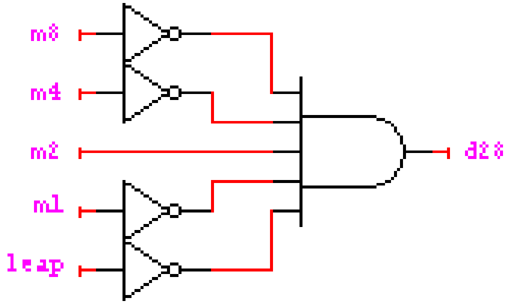
symbol for and

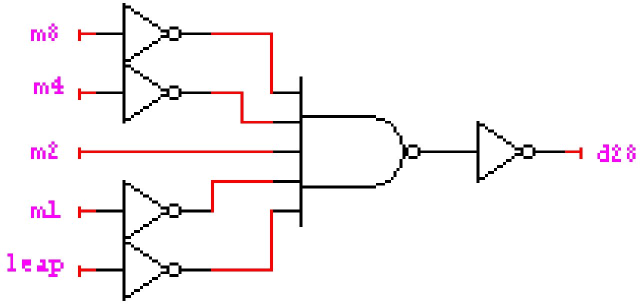
symbol for or

month	leap	d28	d29	d30	d31
0001	–	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	–	0	0	0	1
0100	–	0	0	1	0
...					
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–
0000	–	–	–	–	–

Combinational example (cont'd)

- d28 = m8'•m4'•m2•m1'•leap'
- d29 = m8'•m4'•m2•m1'•leap
- d30 = (m8'•m4•m2'•m1') + (m8'•m4•m2•m1') +
  (m8•m4'•m2'•m1) + (m8•m4'•m2•m1)
  = (m8'•m4•m1') + (m8•m4'•m1)
- d31 = (m8'•m4'•m2'•m1) + (m8'•m4'•m2•m1) +
  (m8'•m4•m2'•m1) + (m8'•m4•m2•m1) +
  (m8•m4'•m2'•m1') + (m8•m4'•m2•m1') +
  (m8•m4•m2'•m1')





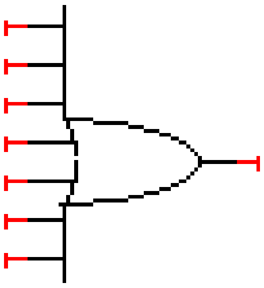
Activity

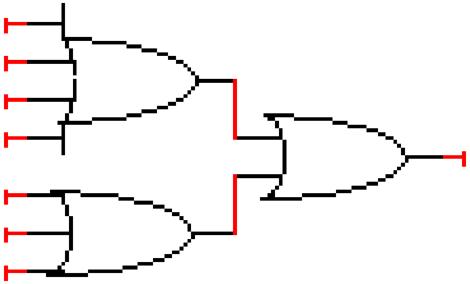
- How much can we simplify d31?

- What if we started the months with 0 instead of 1? (i.e., January is 0000 and December is 1011)

Combinational example (cont'd)

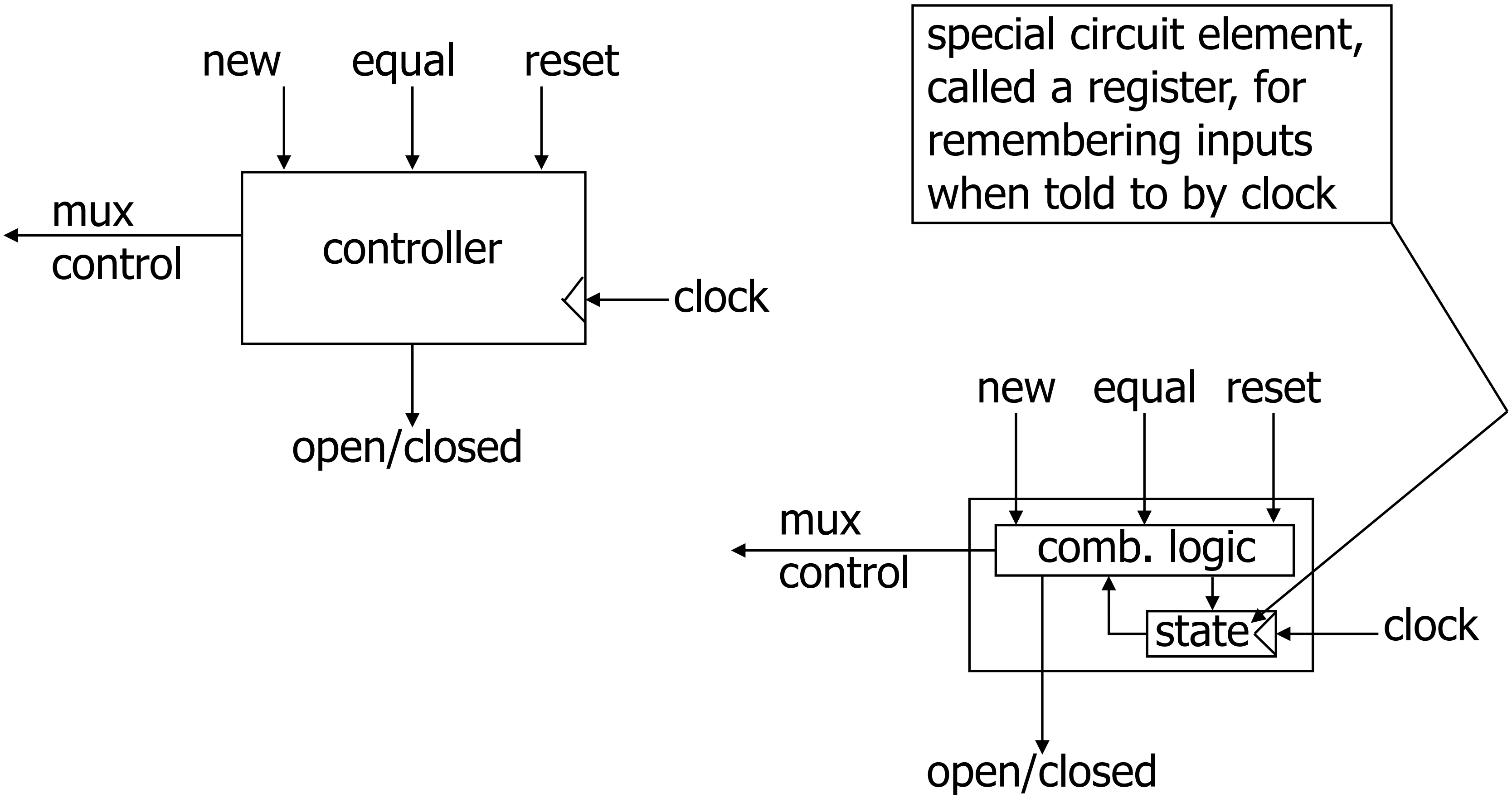
- d28 = m8'•m4'•m2•m1'•leap'
- d29 = m8'•m4'•m2•m1'•leap
- d30 = (m8'•m4•m2'•m1') + (m8'•m4•m2•m1') + (m8•m4'•m2'•m1) + (m8•m4'•m2•m1)
- d31 = (m8'•m4'•m2'•m1) + (m8'•m4'•m2•m1) + (m8'•m4•m2'•m1) + (m8'•m4•m2•m1) + (m8•m4'•m2'•m4') + (m8•m4'•m2•m1') + (m8•m4•m2'•m1')





Sequential example (cont'd): controller implementation

- Implementation of the controller



Summary

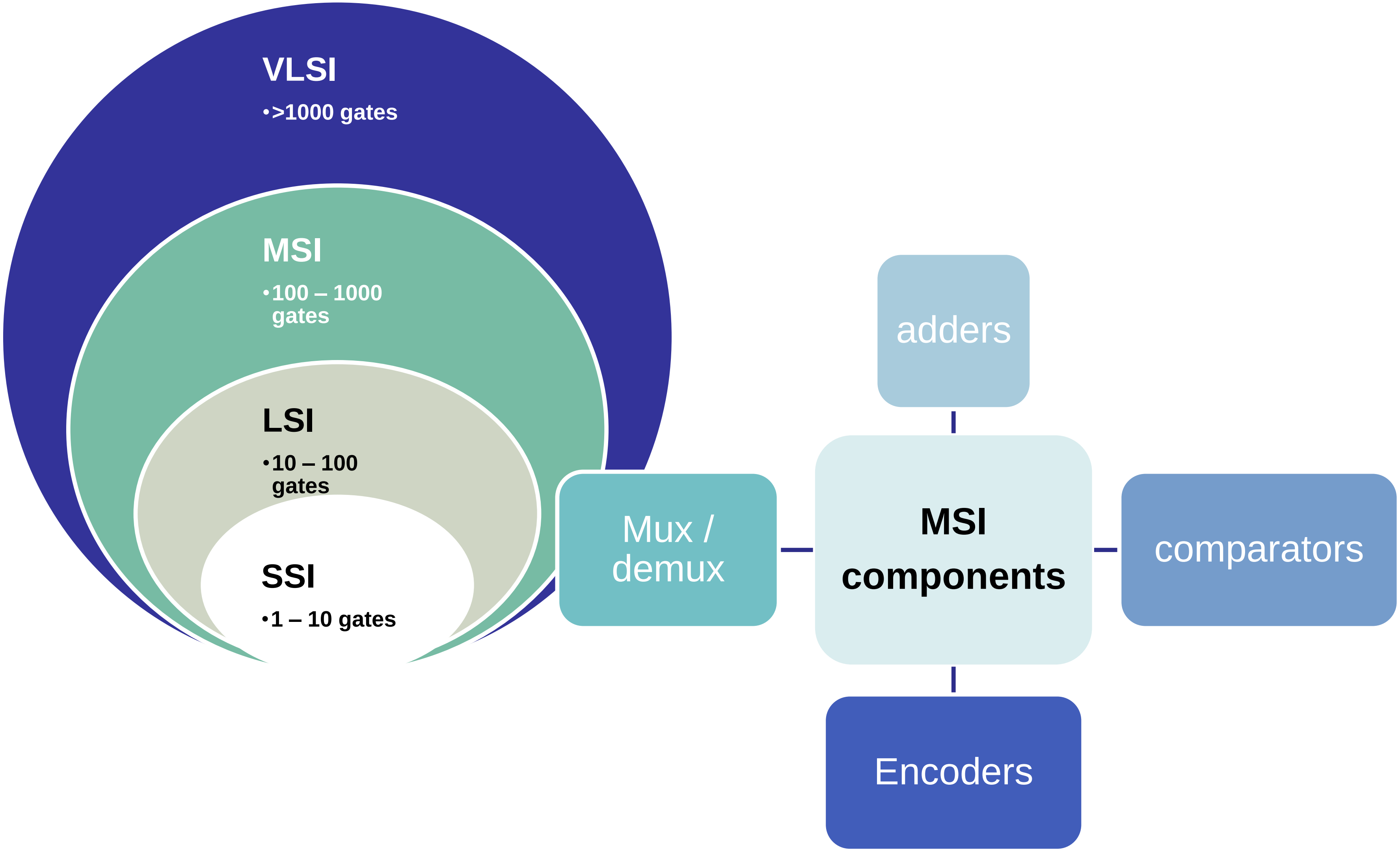
- That was what the entire course is about
  - converting solutions to problems into combinational and sequential networks effectively organizing the design hierarchically
  - doing so with a modern set of design tools that lets us handle large designs effectively
  - taking advantage of optimization opportunities

- Now lets do it again
  - this time we'll take 14 weeks instead of one

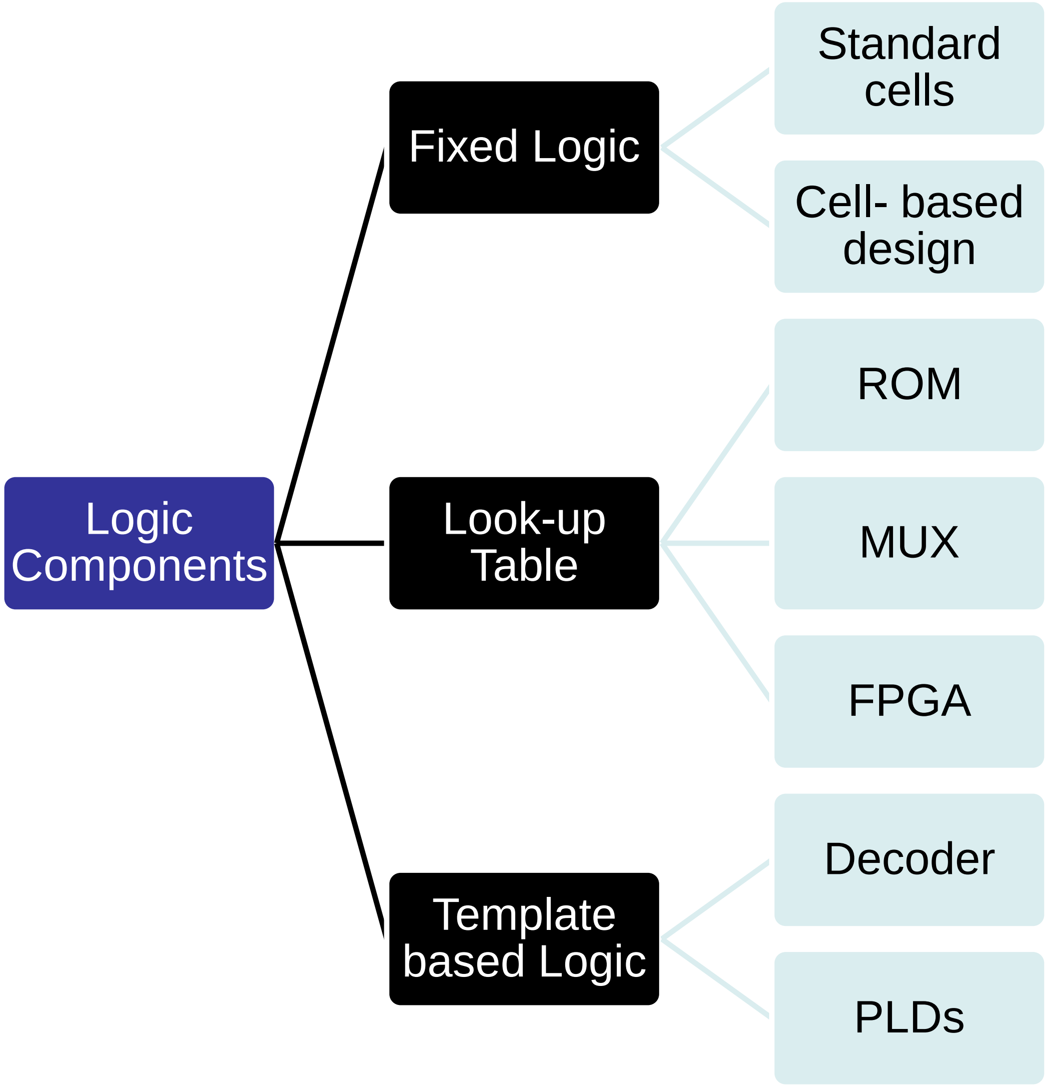
Logic Design Implementation Technologies

1. Programmable Logic Devices (PLD)
   - Programmable Logic Array (PLA)
   - Programmable Array Logic (PAL)
2. Introduction to FPGA & CPLD.
3. Introduction to Hardware Description Language (HDL)

The complexity of a chip

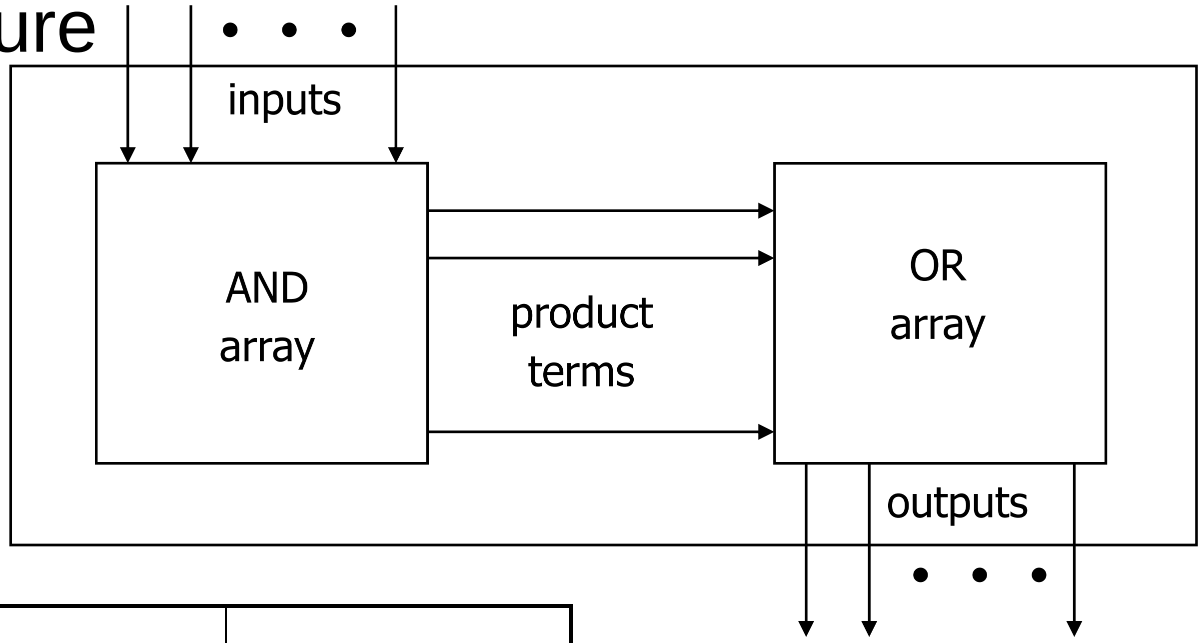


Basic Logic Components



Programmable Logic Devices (PLDs)

General structure



Device	AND-array	OR-array
PROM	Fixed	Programmable
PLA	Programmable	Programmable
PAL	Programmable	Fixed

Types of PLDs

Enabling concept

- Shared product terms among outputs

example:

F0 = A + B' C'
F1 = A C' + A B
F2 = B' C' + A B
F3 = B' C + A

personality matrix

product term	inputs A	B	C	outputs F0	F1	F2	F3
AB	1	1	–	0	1	1	0
B'C	–	0	1	0	0	0	1
AC'	1	–	0	0	1	0	0
B'C'	–	0	0	1	0	1	0
A	1	–	–	1	0	0	1

reuse of terms

input side:

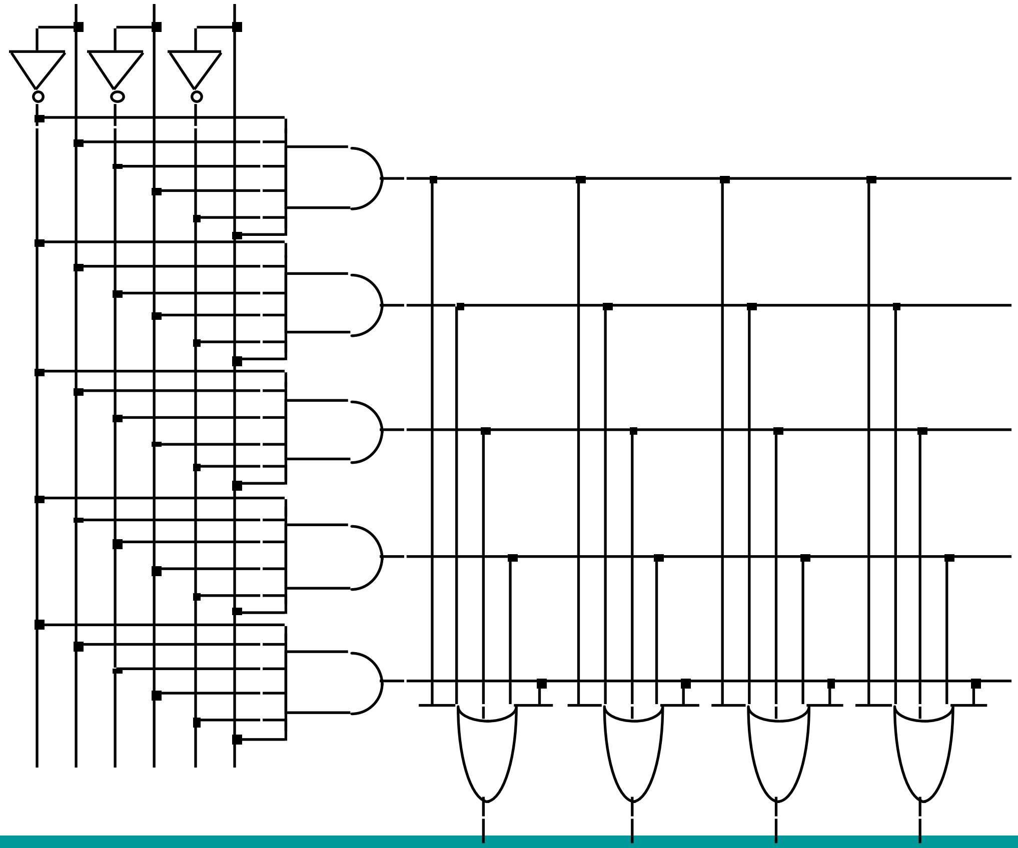
1 = uncomplemented in term
0 = complemented in term
– = does not participate

output side:

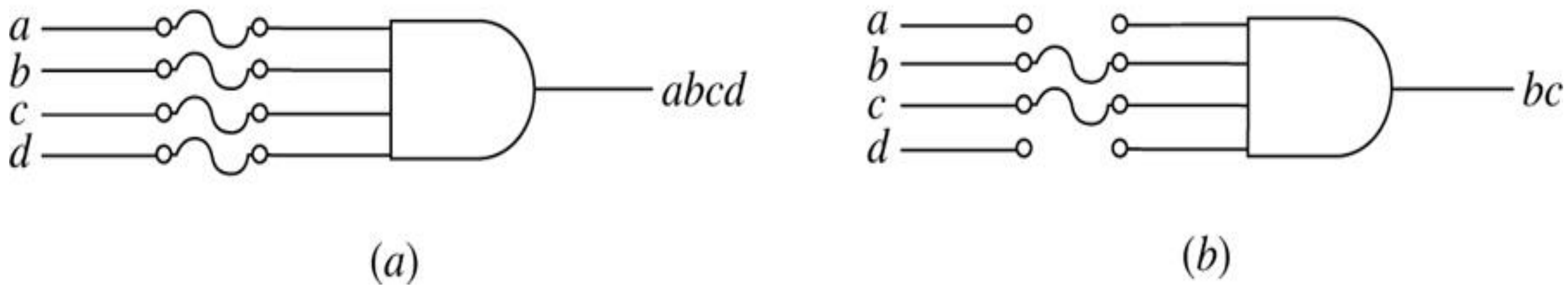
1 = term connected to output
0 = no connection to output

PLA before programming

- All possible connections are available before "programming"
  - in reality, all AND and OR gates are NANDs



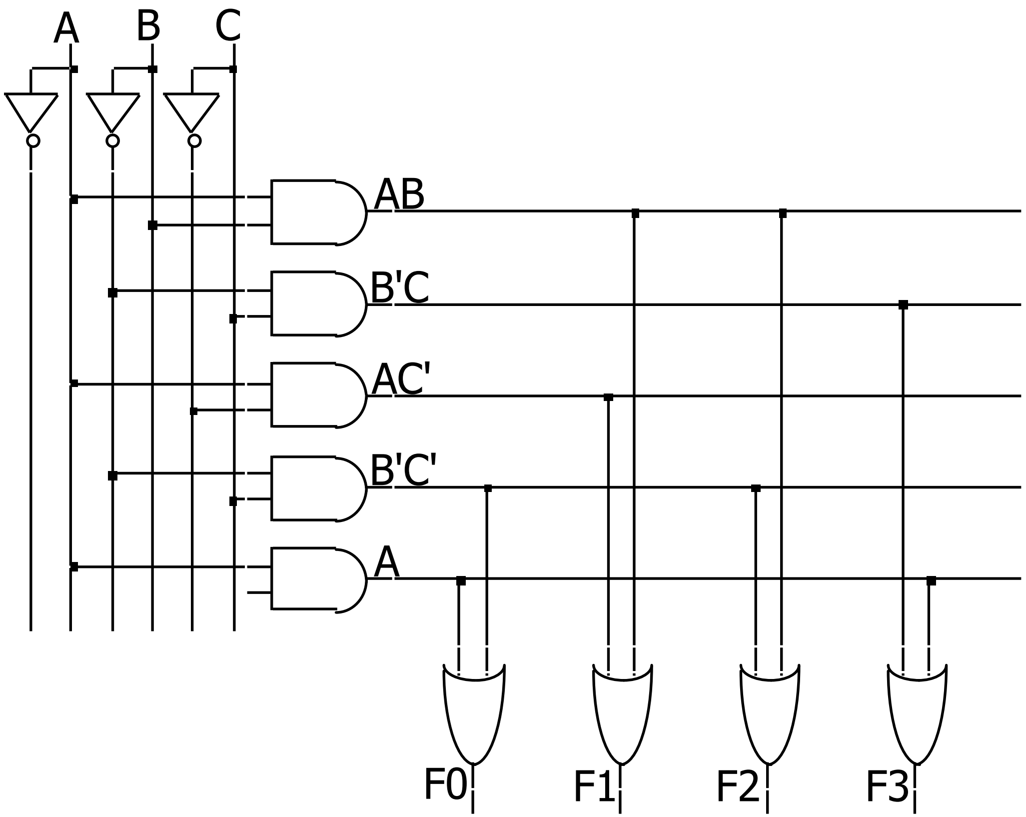
Programming by blowing fuses

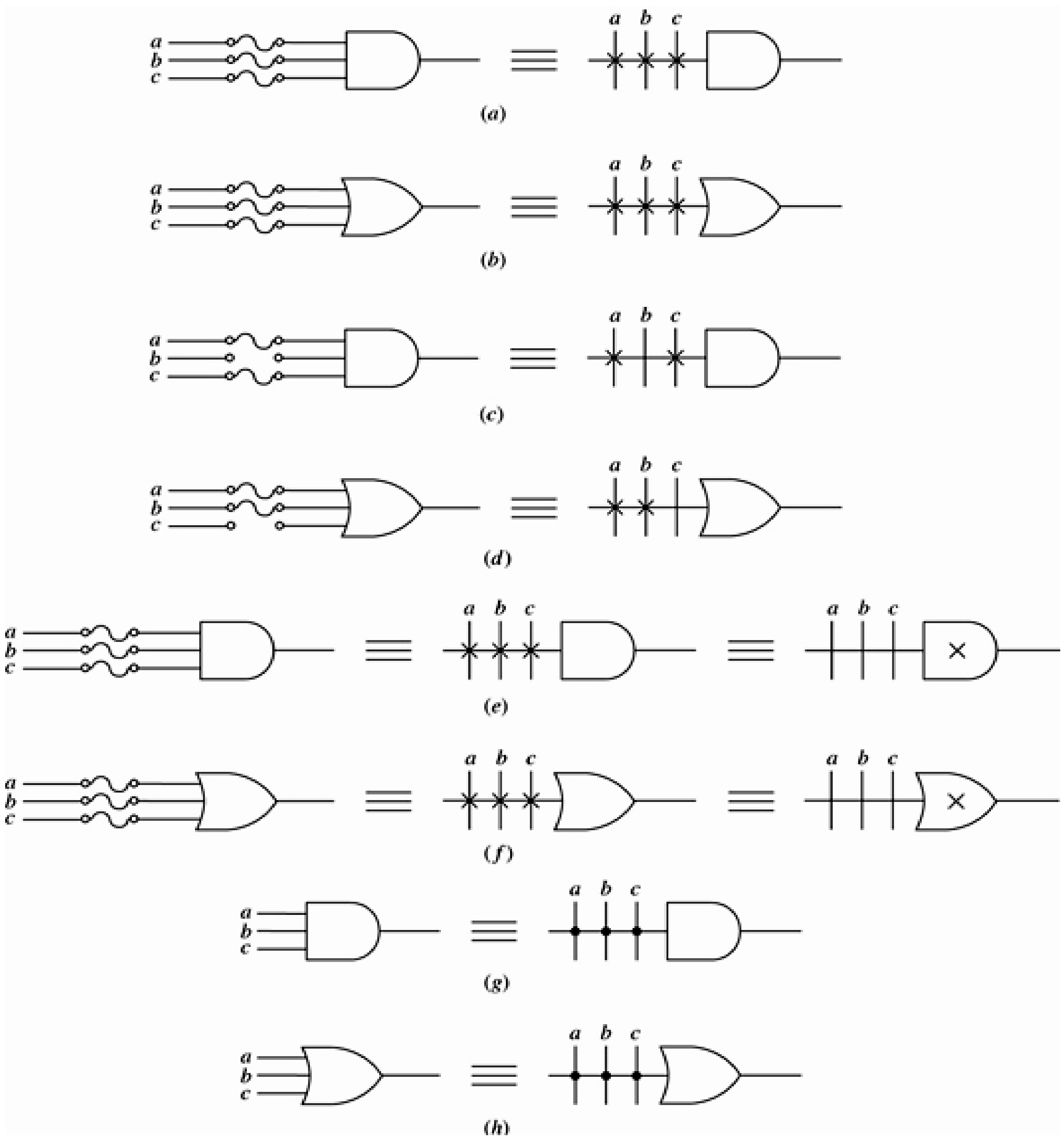


(*a*) Before programming. (*b*) After programming.

After programming

- Unwanted connections are "blown"
  - fuse (normally connected, break unwanted ones)
  - anti-fuse (normally disconnected, make wanted connections)





(*a*)Unprogrammed and-gate.

(*b*)Unprogrammed or-gate.

(*c*)Programmed and-gate realizing the term *ac*

(*d*)Programmed or-gate realizing the term *a* + *b*.

(*e*)Special notation for an and-gate having all its input fuses intact.

(*f*) Special notation for an or-gate having all its input fuses intact

(*g*)And-gate with non-fusible inputs.

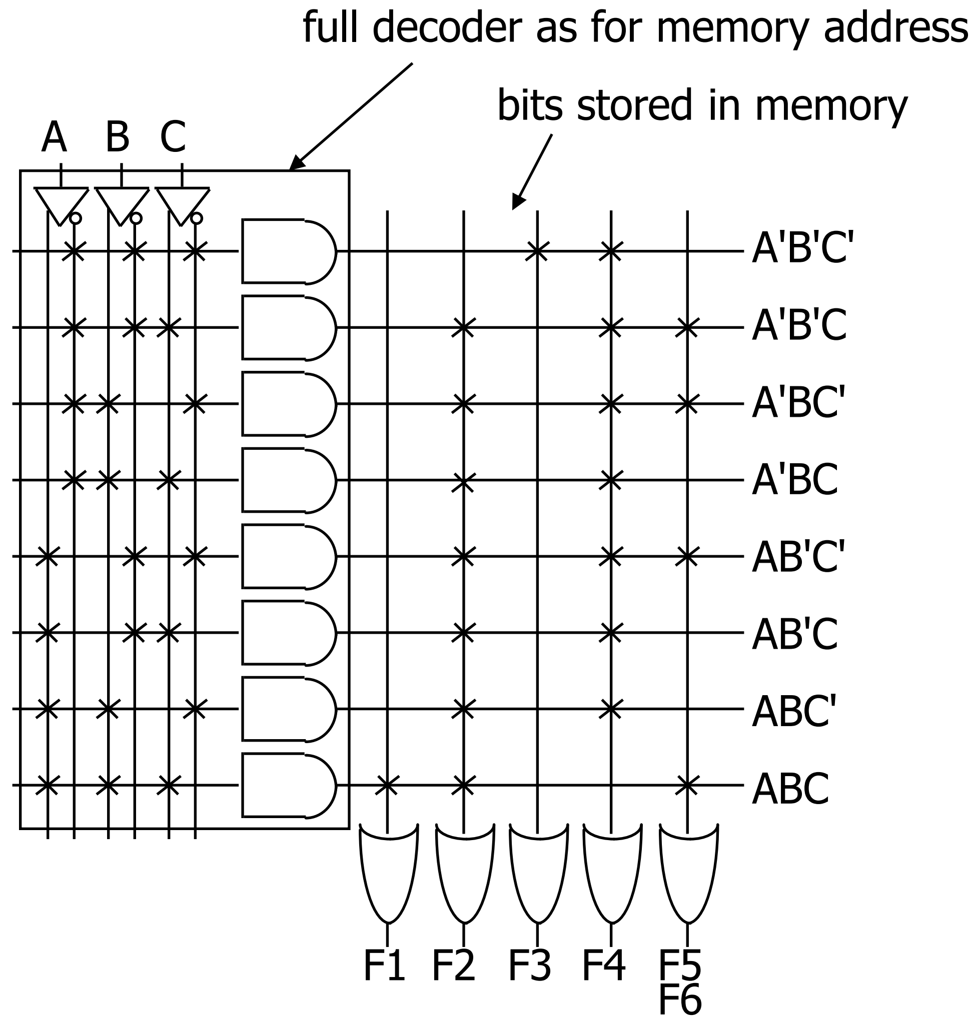
(*h*)Or-gate with non-fusible inputs.

PLA notation.

Programmable logic array example

- Multiple functions of A, B, C
  - F1 = A B C
  - F2 = A + B + C
  - F3 = A' B' C'
  - F4 = A' + B' + C'
  - F5 = A xor B xor C
  - F6 = A xnor B xnor C

A	B	C	F1	F2	F3	F4	F5	F6
0	0	0	0	0	1	1	0	0
0	0	1	0	1	0	1	1	1
0	1	0	0	1	0	1	1	1
0	1	1	0	1	0	1	0	0
1	0	0	0	1	0	1	1	1
1	0	1	0	1	0	1	0	0
1	1	0	0	1	0	1	0	0
1	1	1	1	1	0	0	1	1



PALs and PLAs

PAL

Programmable array logic

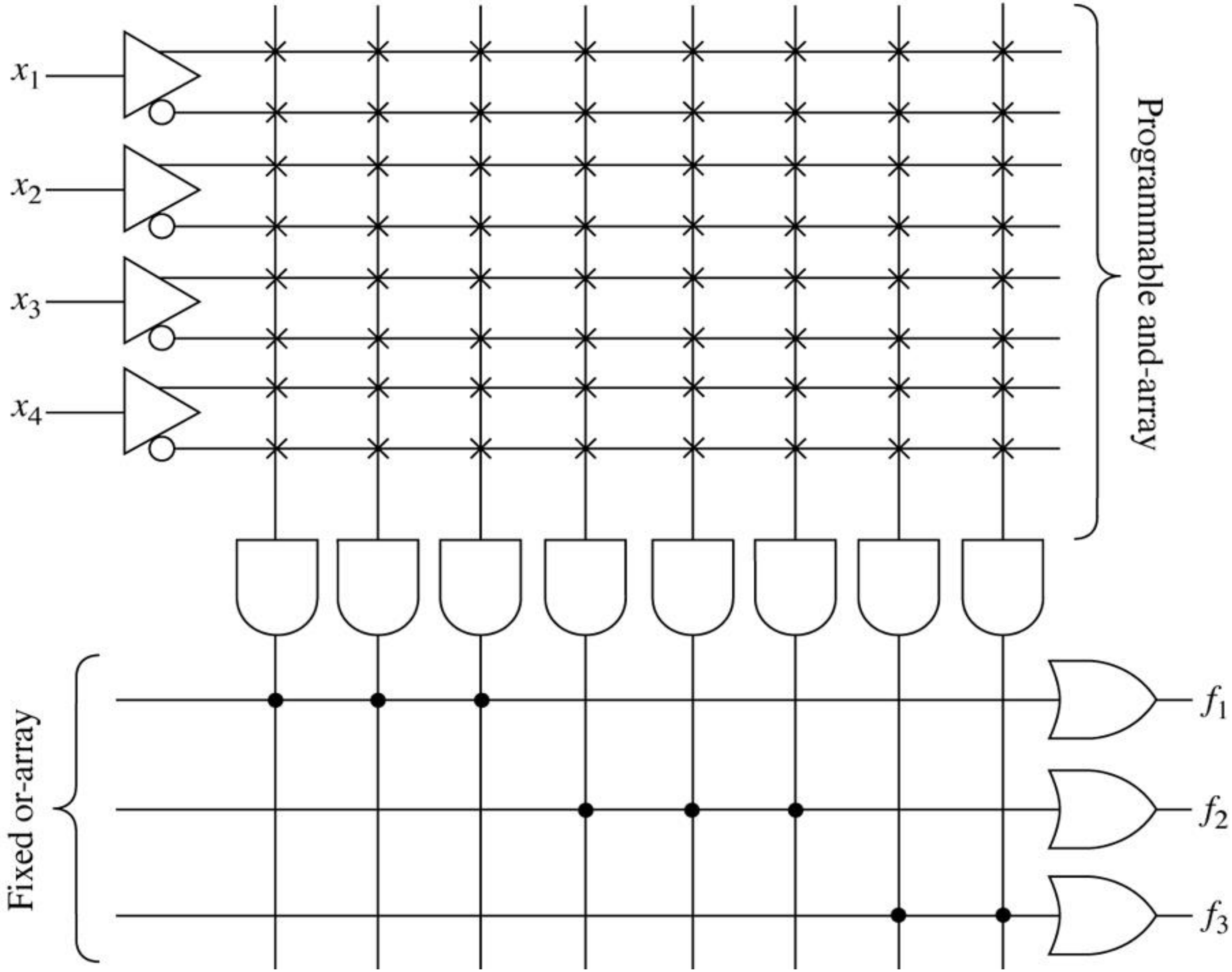
constrained topology of the OR array

PLA

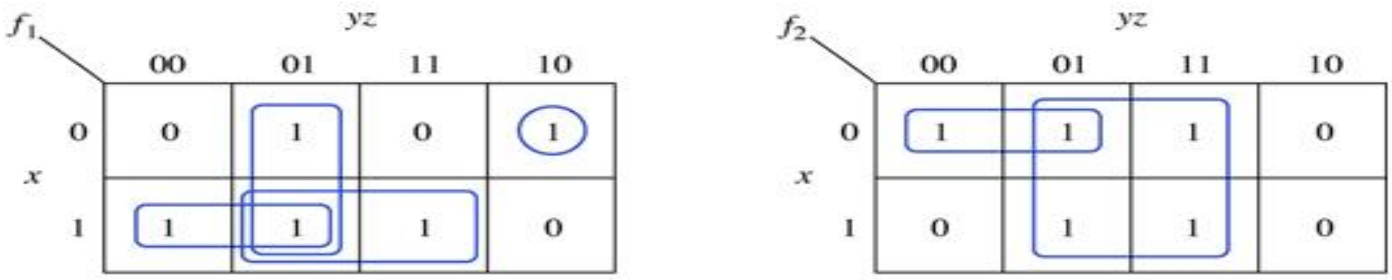
Programmable logic array

unconstrained fully-general AND and OR arrays

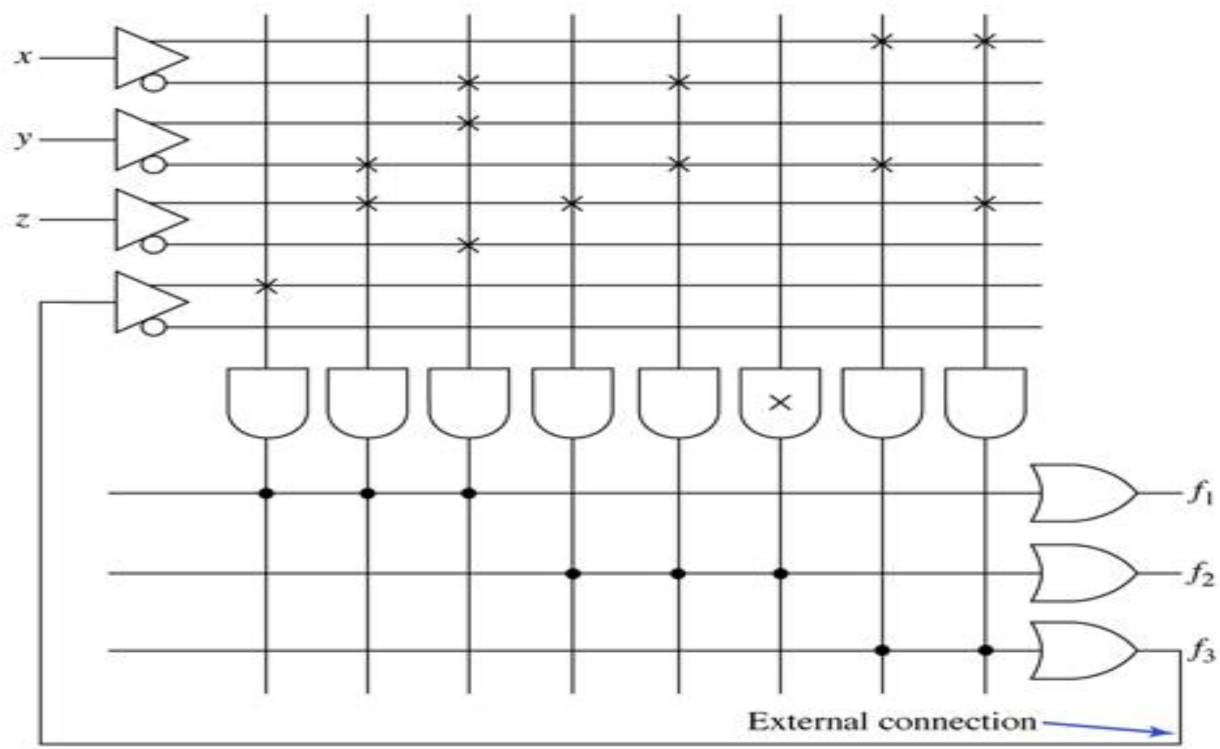
A simple four-input, three-output PAL device.



An example of using a PAL device to realize two Boolean functions. (*a*) Karnaugh maps. (*b*) Realization.



(*a*)



(*b*)

PALs and PLAs: design example

- BCD to Gray code converter

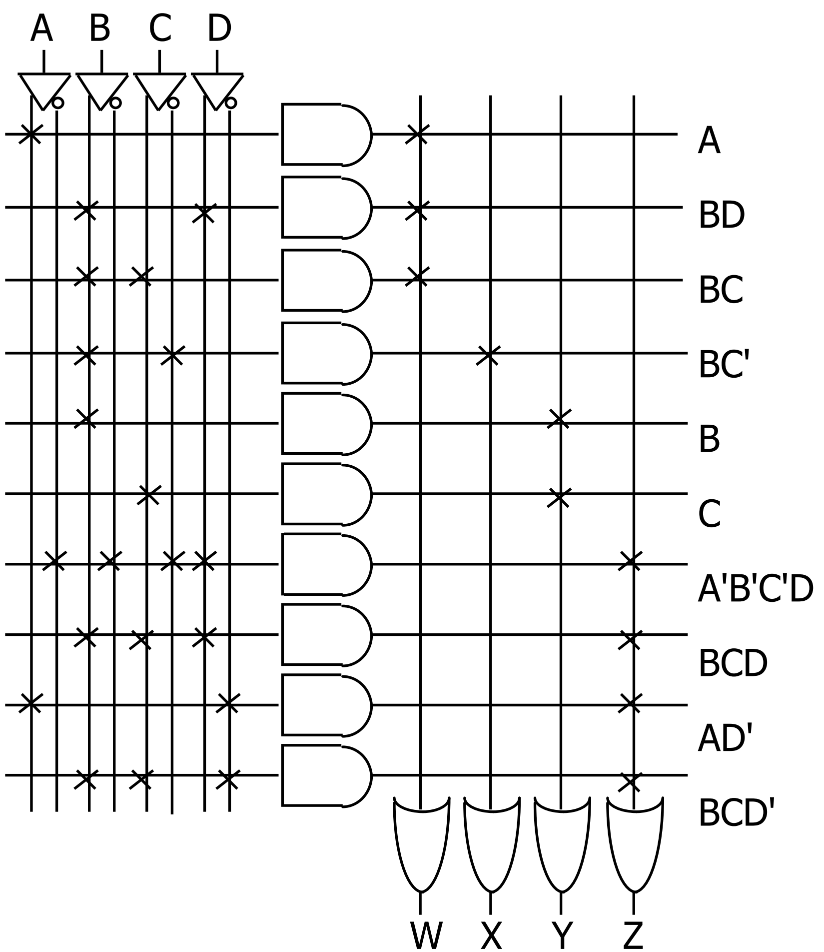
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	1	1	1	0
0	1	1	0	1	0	1	0
0	1	1	1	1	0	1	1
1	0	0	0	1	0	0	1
1	0	0	1	1	0	0	0
1	0	1	–	–	–	–	–
1	1	–	–	–	–	–	–

minimized functions:

W = A + BD + BC
X = BC'
Y = B + C
Z = A'B'C'D + BCD + AD' + B'CD'

PALs and PLAs: design example (cont'd)

- Code converter: programmed PLA



minimized functions:

W = A + BD + BC
X = B C'
Y = B + C
Z = A'B'C'D + BCD + AD' + B'CD'

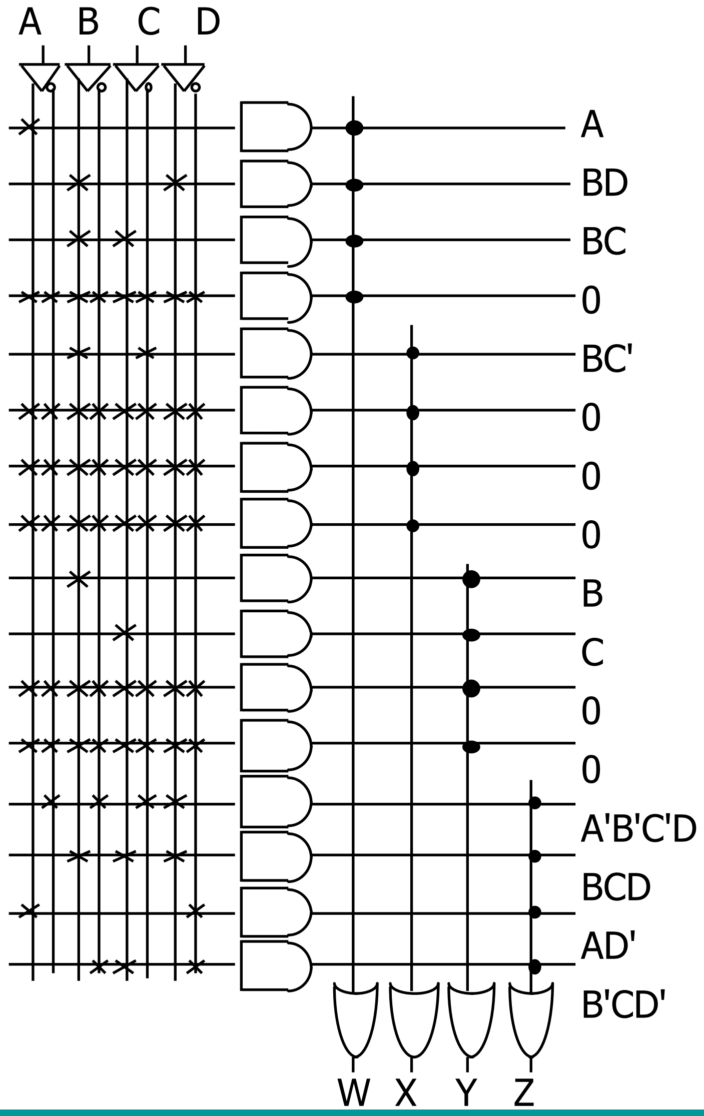
not a particularly good candidate for PAL/PLA implementation since no terms are shared among outputs

however, much more compact and regular implementation when compared with discrete AND and OR gates

PALs and PLAs: design example (cont'd)

- Code converter: programmed PAL

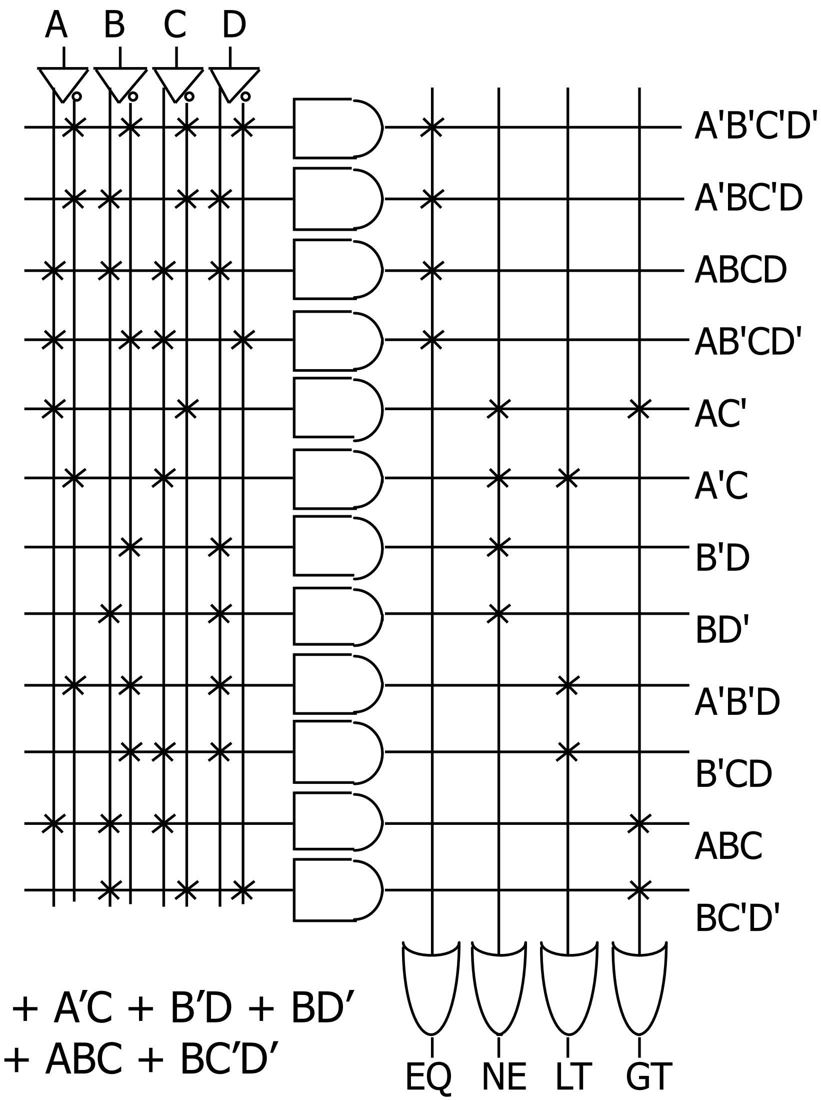
4 product terms
per each OR gate



PALs and PLAs: another design example

- Magnitude comparator

A	B	C	D	EQ	NE	LT	GT
0	0	0	0	1	0	0	0
0	0	0	1	0	1	1	0
0	0	1	0	0	1	1	0
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	1	0	0	0
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	0
1	0	0	0	0	1	0	1
1	0	0	1	0	1	0	1
1	0	1	0	1	0	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	1	0	1
1	1	0	1	0	1	0	1
1	1	1	0	0	1	0	1
1	1	1	1	1	0	0	0



minimized functions:

EQ = A′B′C′D′ + A′BC′D + ABCD + AB′CD′

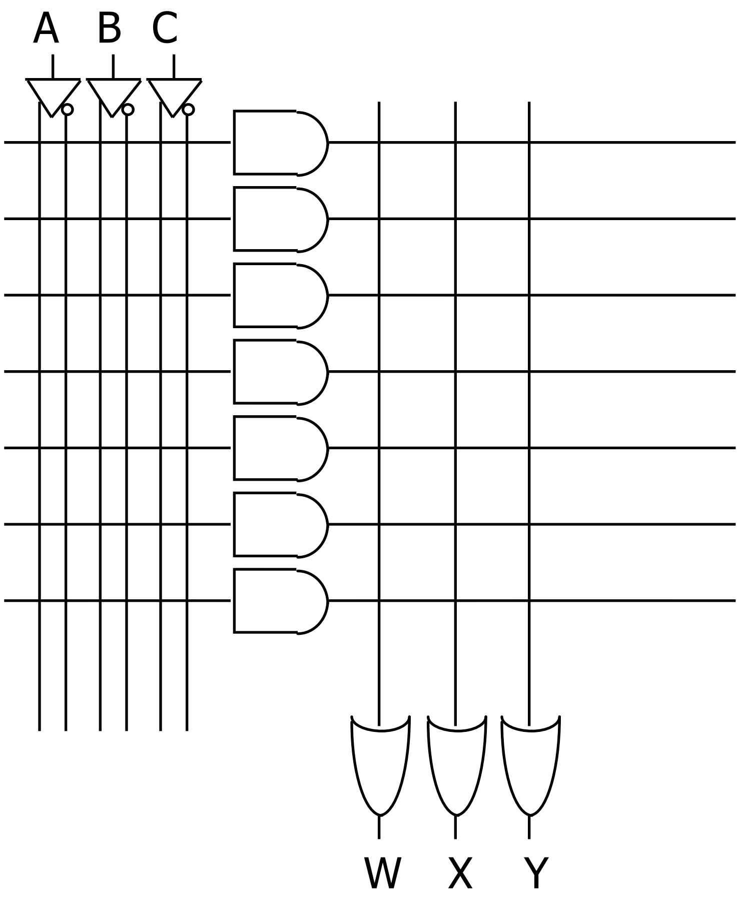
NE = AC′ + A′C + B′D + BD′

LT = A′C + A′B′D + B′CD

GT = AC′ + ABC + BC′D′

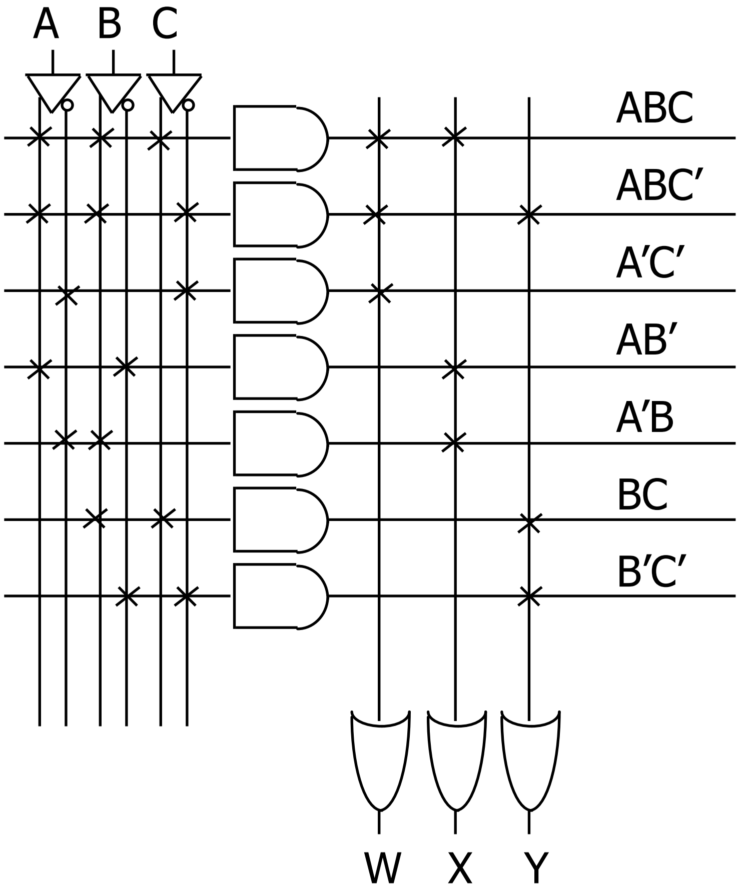
Activity

- Map the following functions to the PLA below:
  - W = AB + A'C' + BC'
  - X = ABC + AB' + A'B
  - Y = ABC' + BC + B'C'



Activity (cont’d)

- 9 terms won’t fit in a 7 term PLA
  - can apply concensus theorem to W to simplify to: W = AB + A’C’
- 8 terms wont’ fit in a 7 term PLA
  - observe that AB = ABC + ABC’
  - can rewrite W to reuse terms: W = ABC + ABC’ + A’C’
- Now it fits
  - W = ABC + ABC’ + A’C’
  - X = ABC + AB’ + A’B
  - Y = ABC’ + BC + B’C’
- This is called technology mapping
  - manipulating logic functions so that they can use available resources



Limitations of PLAs and PALs

These chips are limited to fairly modest size, typically supporting a combined number of inputs plus outputs of not more than 32.

Introduction to FPGA & CPLD

FPGA and CPLD

1. FPGA - Field-Programmable Gate Array.
2. CPLD - Complex Programmable Logic Device
3. FPGA and CPLD is an advance PLD.
4. Support thousands of gate where as PLD only support hundreds of gates.

Complex Programmable Logic Devices(CPLDs)

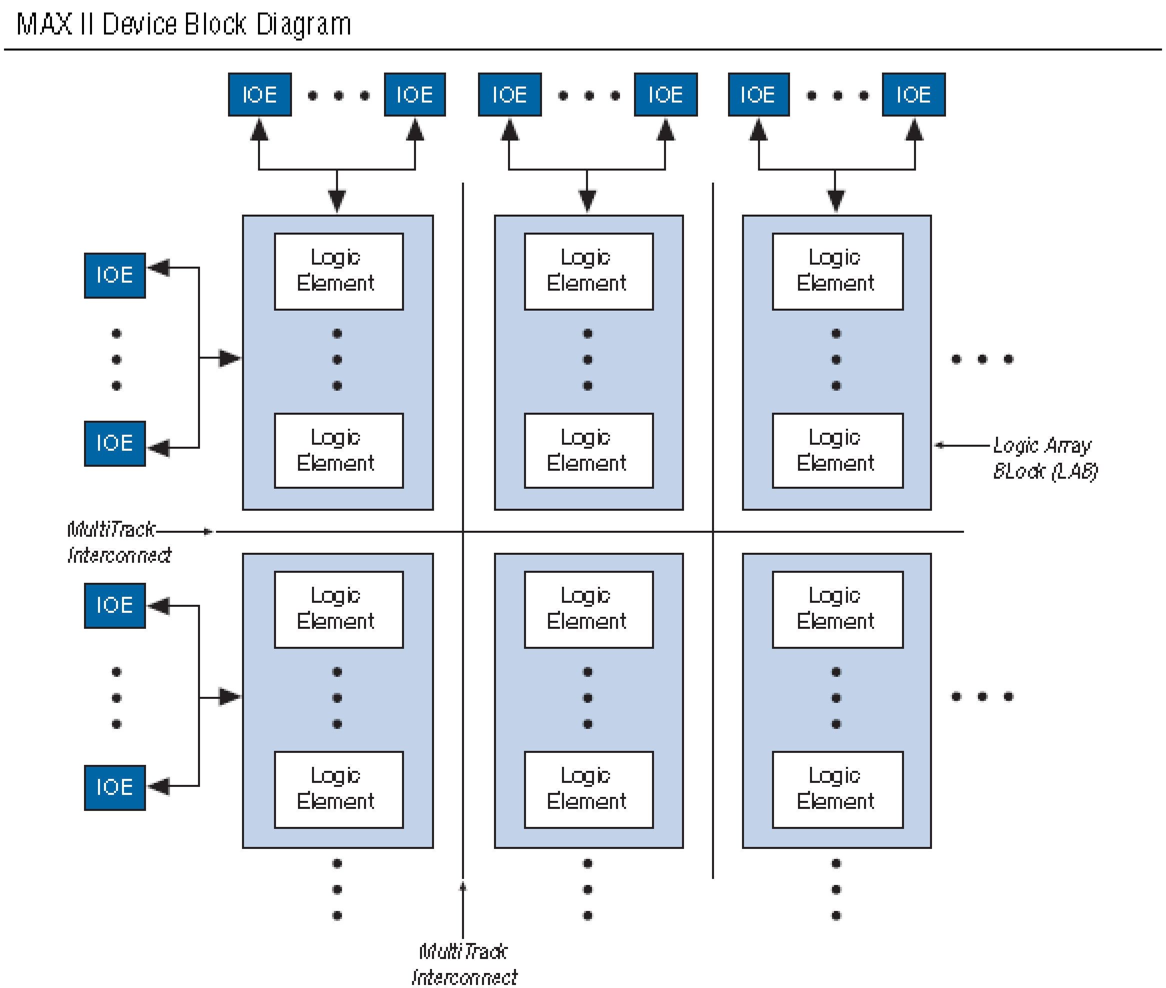
A CPLD comprises multiple PAL-like blocks on a single chip with internal wiring resources to connect the circuit blocks.

It is made to implement complex circuits that cannot be done on a PAL or PLA.

CPLD – Notable supplier

- Altera
  - MAX CPLD series
- Atmel
  - The ATF15xxBE family
- Cypress Semiconductor
  - Ultra37000 family
- Lattice Semiconductor
  - ispMACH 4000ZE CPLD family
- Xilinx
  - CoolRunner™-II CPLDs

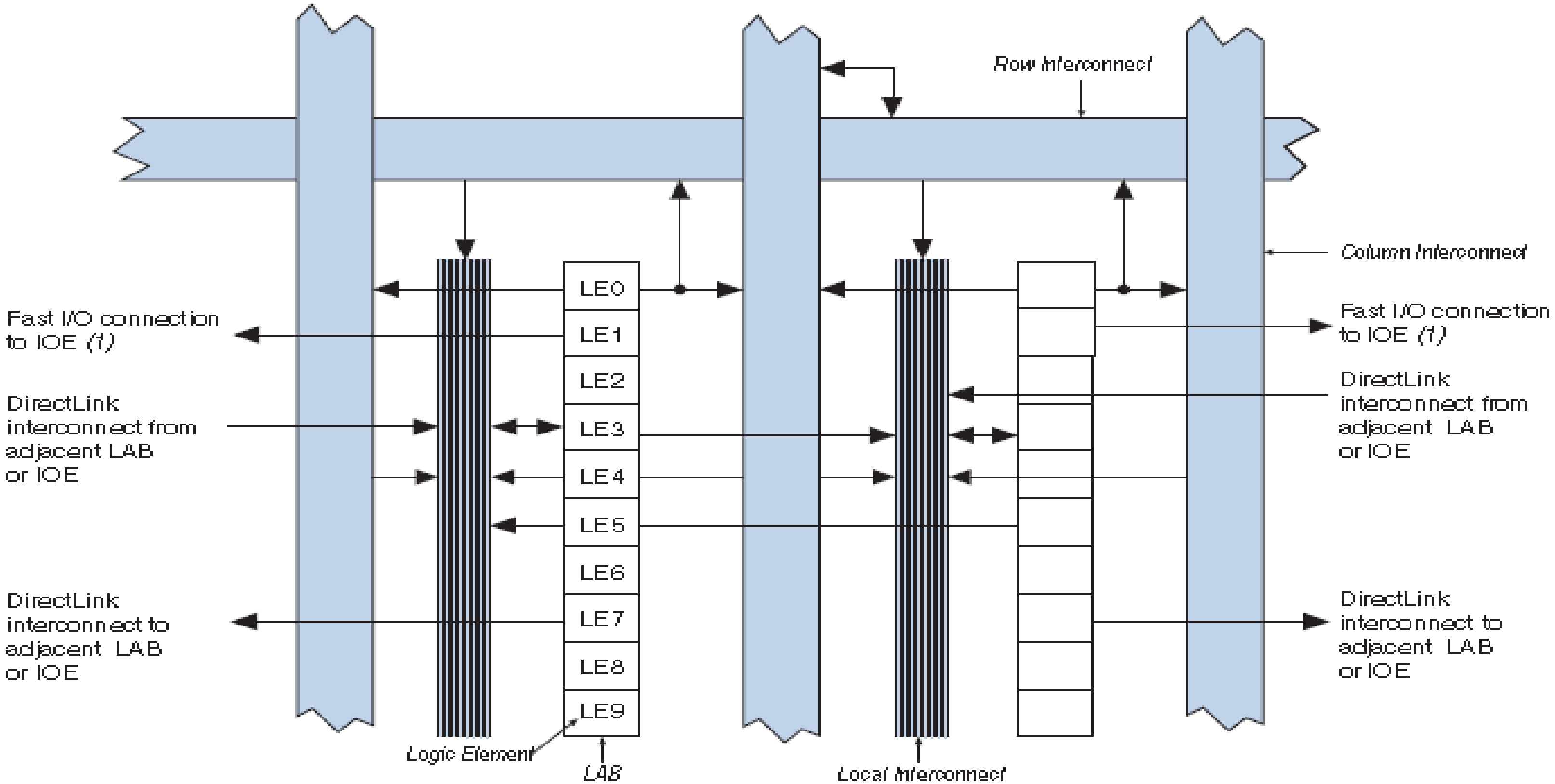
CPLD Architecture



- Row and column interconnects provide signal interconnects between the logic array blocks (LABs).
- 10 logic elements (LEs) in each LAB

CPLD - Logic Array Blocks

- Each LAB consists of 10 LEs, LE carry chains, LAB control signals, a local interconnect, a look-up table (LUT) chain, and register chain connection lines.



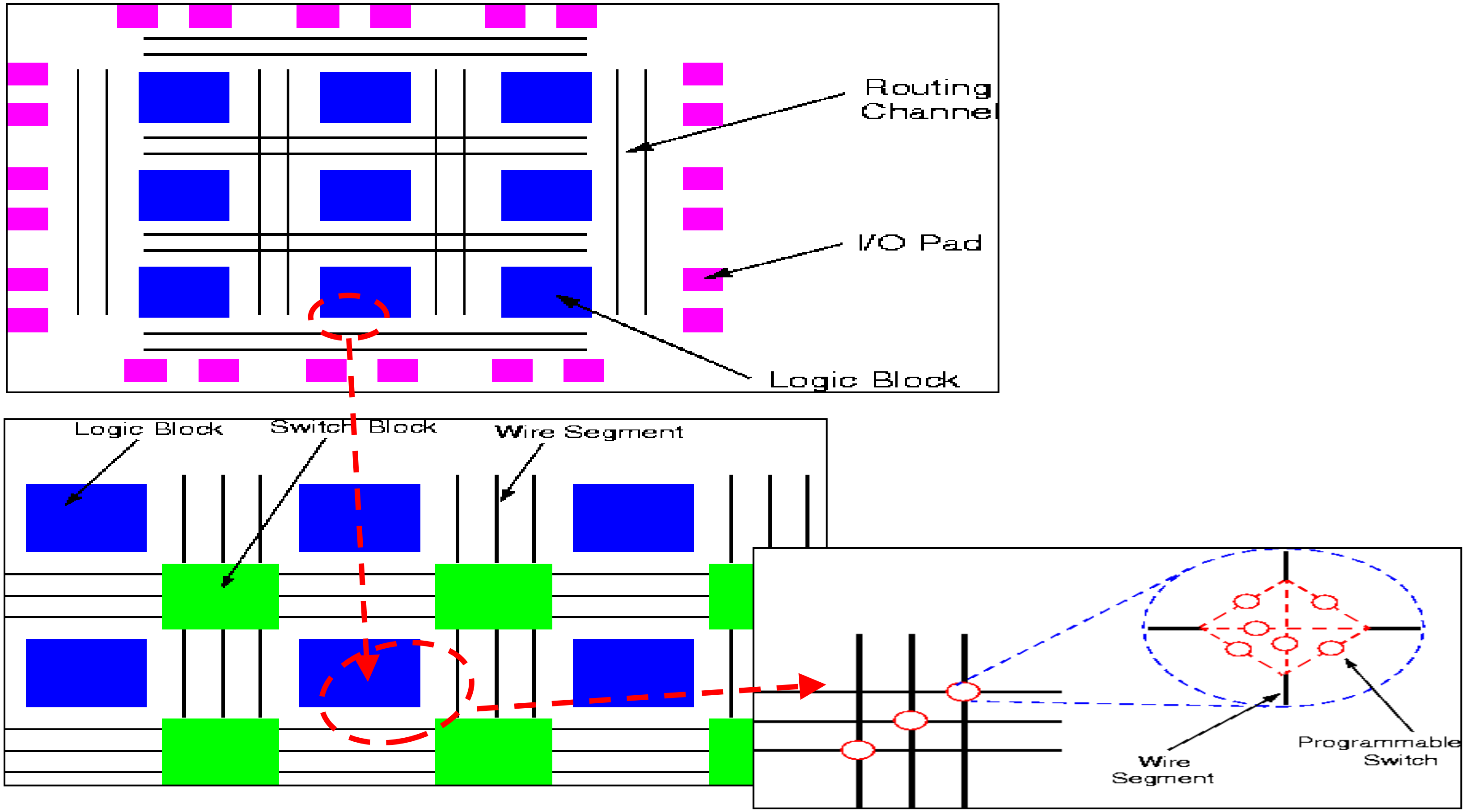
CPLD

1. CPLD featured in common PLD:-
   I. Non-volatile configuration memory – does not need an external configuration PROM.
   II. Routing constraints. Not for large and deeply layered logic.
2. CPLD featured in common FPGA:-
   I. Large number of gates available.
   II. Some provisions for logic more flexible than sum-of-product expressions, can include complicated feedback path.
3. CPLD application:-
   I. Address coding
   II. High performance control logic
   III. Complex finite state machines

What is an FPGA?

- An FPGA is a PLD that supports implementation of large logic circuits.
  It is different from others in that it does not contain AND or OR planes.
- Instead, it contains logic blocks as for implementation
- FPGA architecture consists of an array of logic blocks, I/O pads, and routing channels.

FPGA Architecture

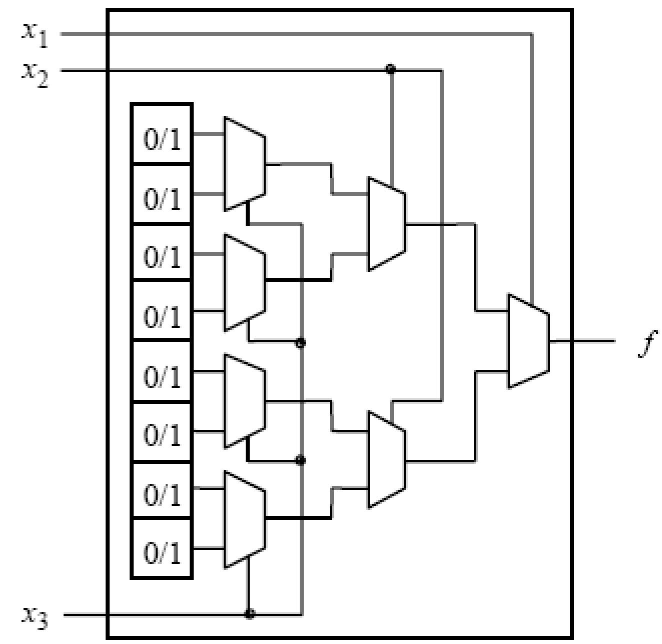


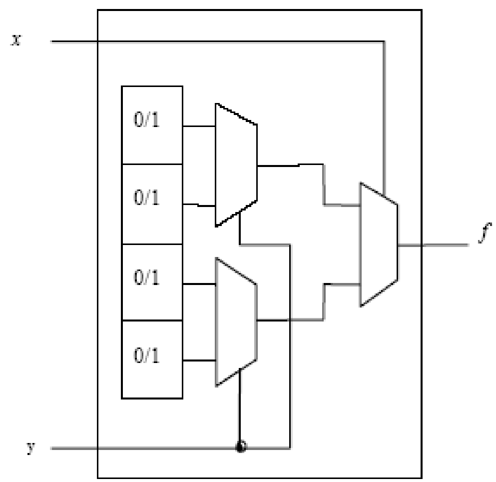
What does a logic cell do?

- Each logic cell combines a few binary inputs (typically between 3 and 10) to one or two outputs according to a Boolean logic function specified in the user program .
- Cell's combinatorial logic may be physically implemented as a small look-up table memory (LUT) or as a set of multiplexers and gates.
- LUT devices tend to be a bit more flexible and provide more inputs per cell than multiplexer cells at the expense of propagation delay.

Typical FPGAs

A three-input LUT





A two-input lookup table

- FPGAs can be used to implement logic circuits of more than a few hundred thousand equivalent gates in size.
- The most commonly used logic block is a *lookup table* (*LUT*) as depicted in these figures.

Field Programmable

- The FPGA's function is defined by a user's program rather than by the manufacturer of the device.
- The program is either 'burned' in permanently or semi-permanently as part of a board assembly process, or is loaded from an external memory each time the device is powered up.
- This user programmability gives the user access to complex integrated designs .

How are FPGA programs created?

- Individually defining the many switch connections and cell logic functions would be a daunting task.
- This task is handled by special software. The software translates a user's schematic diagrams or textual hardware description language code then places and routes the translated design.
- Most of the software packages have hooks to allow the user to influence implementation, placement and routing to obtain better performance and utilization of the device.
- Libraries of more complex function macros (eg. adders) further simplify the design process by providing common circuits that are already optimized for speed or area.

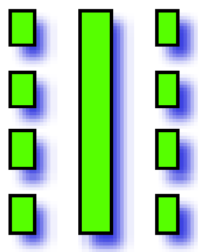
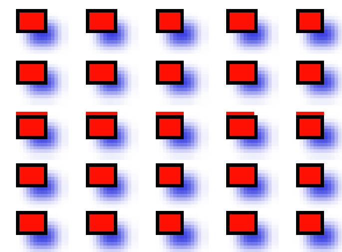
FPGA – Notable Supplier

- Xillinx
  - 7 Series FPGAs
  - Virtex®-6 FPGAs
  - Spartan®-6 FPGAs
  - Virtex-5 FPGAs
  - Extended Spartan-3A FPGAs
  - EasyPath™-6 FPGAs
  - XA Spartan-6 FPGAs
  - XA Spartan-3A FPGAs
  - XA Spartan-3A DSP FPGAs
  - XA Spartan-3E FPGAs
- Altera
  - Stratix® V
  - Arria® II
  - Cyclone® IV
  - Stratix IV
  - Arria
  - Cyclone III
- Lattice Semiconductor
  - LatticeECP3 family
  - LatticeECP2™ and LatticeECP2M™
- Actel
  - IGLOO FPGAs
  - ProASIC3 FPGAs

FPGA

- FPGA applications:-
  i. DSP
  ii. Software-defined radio
  iii. Aerospace
  iv. Defense system
  v. ASIC Prototyping
  vi. Medical Imaging
  vii. Computer vision
  viii. Speech Recognition
  ix. Cryptography
  x. Bioinformatic
  xi. And others.

CPLDs vs. FPGAs

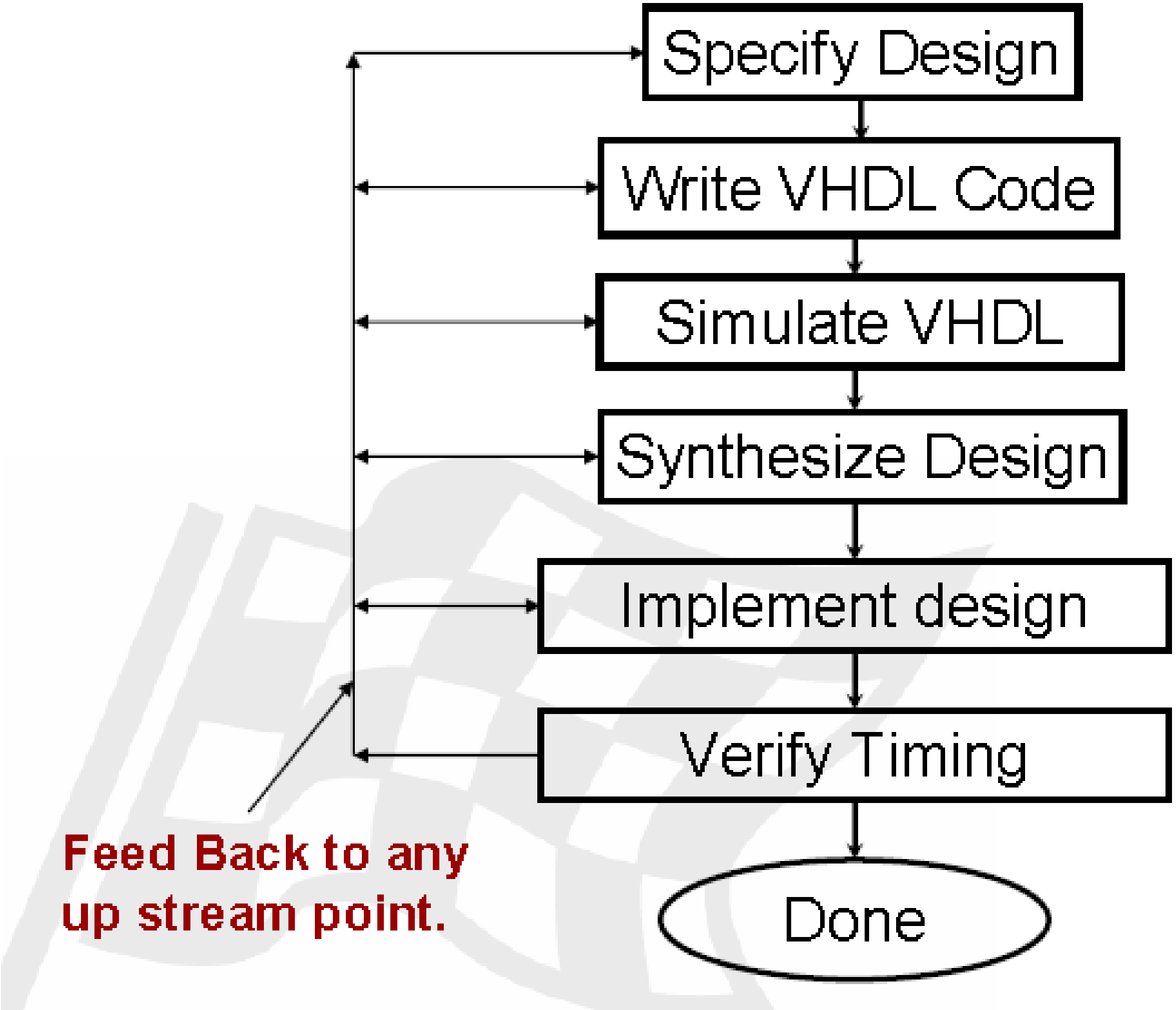
	CPLD	FPGA
	Complex Programmable Logic Device	Field-Programmable Gate Array
		
Architecture	PAL/22V10-like More Combinational	Gate array-like More Registers + RAM
Density	Low-to-medium 0.5-10K logic gates	Medium-to-high 1K to 1M system gates
Performance	Predictable timing Up to 250 MHz today	Application dependent Up to 150 MHz today
Interconnect	"Crossbar Switch"	Incremental

INTRODUCTION TO HARDWARE DESCRIPTION LANGUAGE

Hardware Description Language

- Similar to a typical computer programming language
- But used to describe hardware rather than a program
- IEEE standards :- VHDL (VHIC (Very High Speed Integrated Circuit) Hardware Description Language) & Verilog

VHDL Design Flow



The Entity / Architecture pair

- The basis of all VHDL designs
- Entities can have more then one Architecture
- Architectures can have only one entity
- Entities define the interface (i.e. I/Os) for the design
- Architectures define the function of the design

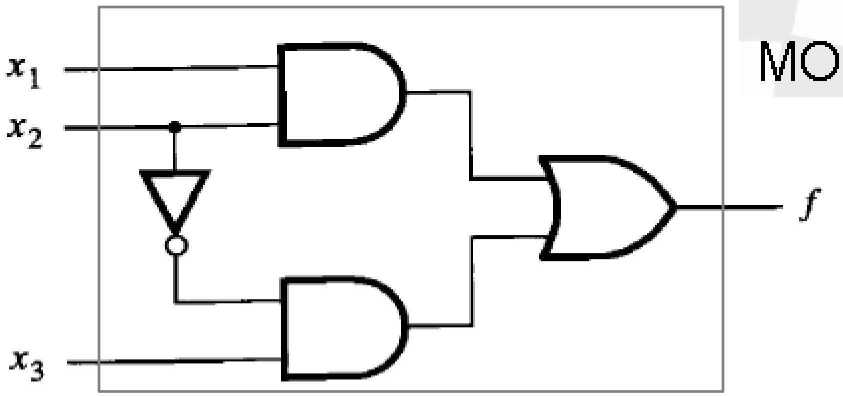
The Entity Details

- Declare the input and output signals

```
entity entity_name is
  generic (generic_list);
  port (port_list);
end entity_name;
```

```
ENTITY example1 IS
    PORT ( x1, x2, x3  : IN   BIT ;
                 f     : OUT  BIT ) ;
END example1 ;
```

example1



(Port_names : MODE type);

MODE types: **in, out, inout** or **buffer**

The Architecture Details

- Declare the functions

```
architecture architecture_name of entity_name is
        declaration section
begin
        concurrent statements
end architecture_name;
```

```
ARCHITECTURE LogicFunc OF example1 IS
BEGIN
        f <= (x1 AND x2) OR (NOT x2 AND x3);
END LogicFunc ;
```

The Architecture Details

- Declaration section
  – Signals, constants and components local to the architecture can be declared here
- Concurrent statements
  – Where the circuit is defined

Complete code

```
ENTITY example1 IS
    PORT ( x1, x2, x3  : IN   BIT ;
           f           : OUT  BIT ) ;
END example1 ;

ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

Logical Operators

- VHDL predefines the logic operators
  - NOT → HIGHER PRECEDENCE
  - AND
  - NAND
  - OR
  - NOR
  - XOR
  - XNOR

There is no implied precedence for these operators. If there are two or more different operators in an equation, the order of precedence is from left to right

- *Note: XNOR supported in standard 1076-1993*

Comments

-- (Double minus sign) is the comment mark

- All text after the -- on the same line is taken as a comment
- Comments only work on a single line
- There is no block comment in VHDL
- The ISE editor does support commenting of selected areas.

Data Types

- DATA types: An ordered set of possible values define a particular type
  - Example: Type **character** is the ASCII character set
- VHDL is a strongly typed language
- All variables must be assigned a type
- Type conversion functions are supplied in add on functions but are not part of the core of VHDL

Predefined Types

- Boolean FALSE, TRUE
- Bit (‘0’,’1’)
- bit_vector(“101010”)
- Integers: range -(2^31-1) to 2^31-1
- Floating real: -1.E38 to 1.0E38
- Time
- Character
- String
- Enumerated (User defined)
- Records, file & access types (Used in Simulation only)

Std_logic & std_ulogic

Not part of 1076

- Part of 1164 library
- Std_logic is a *resolved type*
- Std_logic is a subtype of std_ulogic
- Std_ulogic Values:

```
TYPE std_ulogic IS ('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
);
```

Standard Logic Vectors

- Defined in IEEE 1164
- Ordered set of signals

```
library IEEE;
use IEEE.std_logic_1164.all;
entity busses is
    port (
        In_bus1, In_bus2        : in std_logic_vector (7 downto 0);
        In_bus3                 : in std_logic_vector (0 to 7);
        Out_bus                 : out std_logic_vector (7 downto 0)
          );
end busses;
```

Vector Properties

- Vectors are filled from left to right, always
- Indexes are assigned ascending or descending depending on the key word **to or downto**
- examples

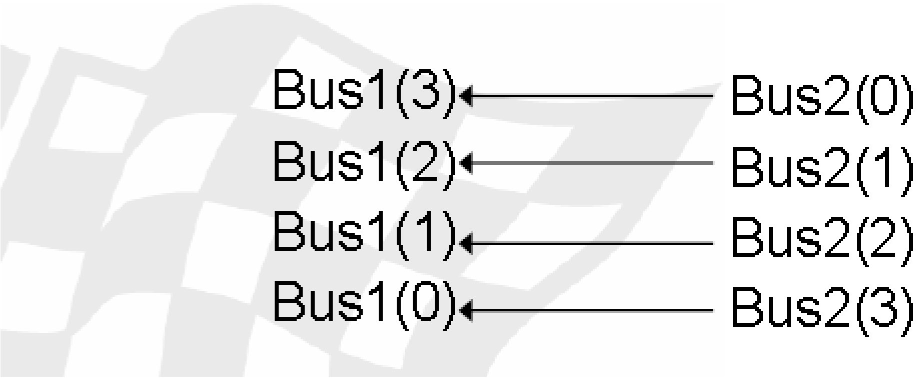
```
In_bus1, In_bus2  : in std_logic_vector (7 downto 0);
In_bus3           : in std_logic_vector (0 to 7);
Out_bus           : out std_logic_vector (7 downto 0)

……
In_bus1 <="10110010";     -- IN_BUS1(7)= 1, IN_BUS1(0) = 0
In_bus2(3) < ='1' ;       -- IN_BUS2 = (U,U,U,U,1,U,U,U,)
In_bus2(6 downto 4) < ="101" ;     -- IN_BUS2 = (U,1,0,1,U,U,U,U,)
In_bus3 <="10110010";     -- IN_BUS3(7)= 0, IN_BUS3(0) = 1
```

Array Ordering

Bus1 : std_logic_vector (3 **downto** 0);
Bus2 : std_logic_vector (0 **to** 3);

Bus1 <= Bus2;



Aggregates

```
signal X_bus, Y_bus, Z_bus  : std_logic_vector (3 downto 0);
signal Byte_bus             : std_logic_vector (7 downto 0);
```

- Aggregates can be used to fill a std_logic_vector in sections

```
Byte_bus <= ( 7 => '1' , 6 downto 4 => '0', others => '1');
-- Byte_bus =10001111
-- Others refers to all the values of the array not yet mentioned
```

- Aggregates can be used to set all members of a std_logic vector to a particular value without knowing the width of the std_logic_vector

```
Z_bus <= (others=>'0');
```

Concatenation

- Concatenation (&) is used to gather pieces of an array to construct a bigger array

```
signal x_bus, y_bus, z_bus      : std_logic_vector (3 downto 0);
signal byte_bus                 : std_logic_vector (7 downto 0);
signal a,b,c,d                  : std_logic;
```

- Building a larger std_logic_vector from small vectors

```
Byte_bus <= x_bus & y_bus;  -- Concatenation operator &
```

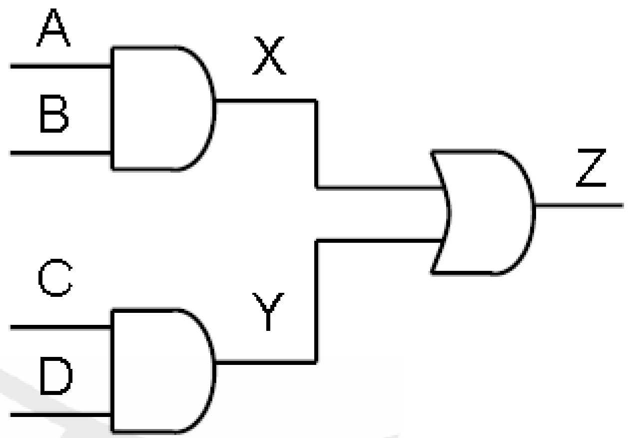
- Building a std_logic_vector from std_logic

```
z_bus <= a&c&b&d;
```

Note: the total width of the right hand side must be equal to the width of the left hand side

Concurrent Statements

- Concurrent statements are **Order independent!!!**




```
Z<=X or  Y;          X<=A and B;
Y<=C and D;    =     Y<=C and D;
X<=A and B;          Z<=X or Y;
```

Relational Operators

- = Equals
- /= Not equal
- < Ordering, less than
- <= Ordering, less than or equal
- > Ordering, greater than
- >= Ordering, greater than or equals

Process and Sequential Statements

- Processes exist inside the Architecture
- Processes have local variables
- Processes contain Sequential Statements
- Processes have a sensitivity list or an optional **wait statement**
- Processes execute **only when a signal in the** sensitivity list changes
- Processes can be used to make **clocked circuits**

The Process Framework

```
Label:-- optional label
process (optional sensitivity list)
-- local process declarations
begin
-- sequential statements
-- optional wait statements
end process;
```

Processes must have a sensitivity list or a **wait** statement, but never both

If Statements

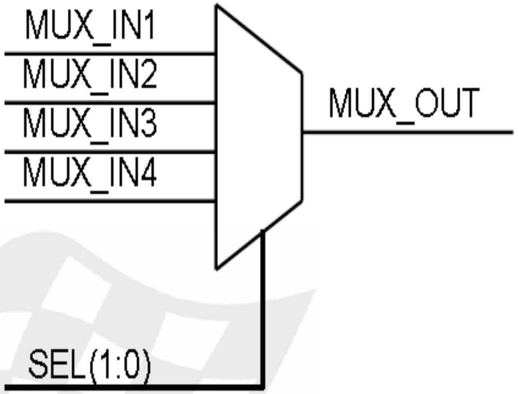
- Can have overlapping conditions
- Imply priority, first true condition is always taken
- Can have incomplete condition lists
- Useful to control signal assignments

Sequential If Statement

- Used inside the **Process**
- Can be used to control variable and signal assignments
- Has optional **elsif structure**

```
if <condition> then
    sequential_statements
elsif <condition> then
    sequential_statements
else
    sequential_statements
end if;
```

Example Multiplexer




```
library IEEE;
use IEEE.std_logic_1164.all;
entity MUX is
   port ( MUX_IN1, MUX_IN2, MUX_IN3, MUX_IN4 : in std_logic;
          SEL                                : in  std_logic_vector (1 downto 0);
          MUX_OUT                            : out std_logic);
end MUX;
architecture IF_MUX_arch of MUX is
begin
  process (SEL, MUX_IN1, MUX_IN2, MUX_IN3, MUX_IN4)
  begin
          if  SEL = "00" then
                    MUX_OUT <= MUX_IN1;
          elsif  SEL = "01" then
                    MUX_OUT <= MUX_IN2;
          elsif  SEL = "10" then
                    MUX_OUT <= MUX_IN3;
          else
                    MUX_OUT <= MUX_IN4;
          end if;
  end process;
end IF_MUX_arch;
```

Avnet SpeedWay Design Workshop™

What Goes Into the Sensitivity List

- If a change on an input signal causes an **IMMEDIATE** change in any signal that is assigned in that process then it should be in the sensitivity list
- If there is **NO IMMEDIATE** change in a signal assigned in the process based on the change of a particular input signal, then that input signal should NOT be in the sensitivity list

When Statement

- The concurrent version of the **IF statement**

```
LABEL1:      -- optional label
  SIG_NAME <= <expression> when <condition> else
              <expression> when <condition> else
              <expression>;
```

```
architecture WHEN_MUX_arch of MUX is
begin
        MUX_OUT <= MUX_IN1 when SEL="00" else
                   MUX_IN2 when SEL="01" else
                   MUX_IN3 when SEL="10" else
                   MUX_IN4;
end WHEN_MUX_arch;
```

The Case Statement

- Used to control signal assignments
- No priority implied
- Control expression must cover all possible signal assignments
- No conditions may overlap

Sequential Case Statement

- Must be inside a process

```
case <expression> is
   when <choices> =>
      <statements>
   when <choices> =>
      <statements>
   when others =>
      <statements>
 end case;
```

```
architecture CASE_MUX_arch of MUX is
begin
    process (MUX_IN1, MUX_IN2, MUX_IN3, MUX_IN4, SEL)
     begin
       case sel is
          when "00" =>
               MUX_OUT <= MUX_IN1;
          when "01" =>
               MUX_OUT <= MUX_IN2;
          when "10" =>
               MUX_OUT <= MUX_IN3;
          when others =>
               MUX_OUT <= MUX_IN4;
          end case;
       end process;
end CASE_MUX_arch;
```

Select; the Concurrent Case Statement

```
LABEL1:     -- optional label
  with <choice_expression> select
  SIG_NAME <= <expression> when <choices>,
              <expression> when <choices>,
              <expression> when others;
```

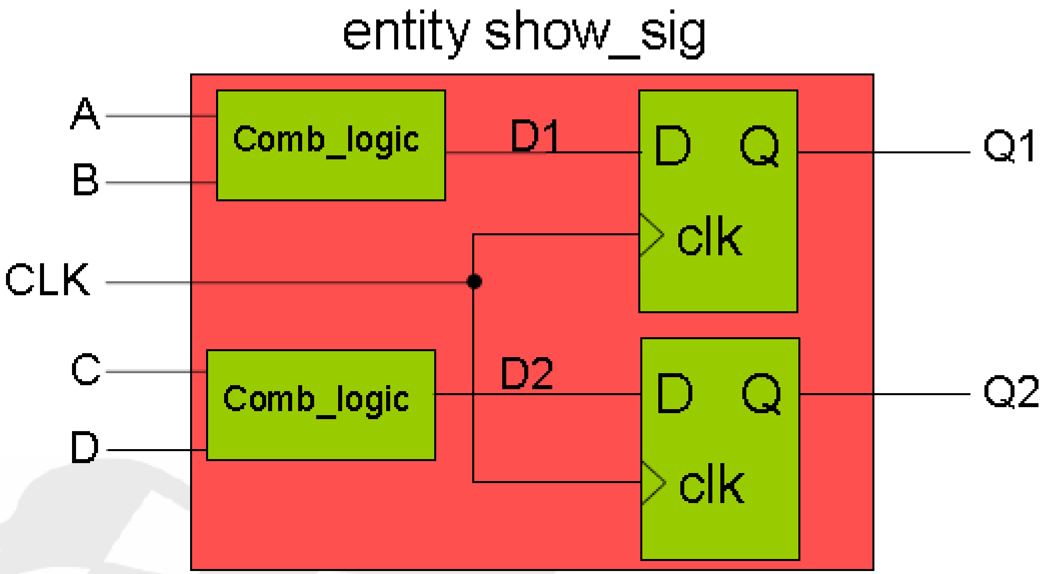
```
architecture SEL_MUX_arch of MUX is
begin
  with SEL select
    MUX_OUT <= MUX_IN1 when "00",
               MUX_IN2 when "01",
               MUX_IN3 when "10",
               MUX_IN4 when others;
end SEL_MUX_arch;
```

Signals

- Signals behave like wires within a VHDL design
- Signals can be local to an Architecture
- Signals have no MODE
- Signals can be declared in the Architecture declarative region
- Signals **must have a type**
- Signals carry information between PROCESS es

```
signal signal_name1,signal_name2 : type;
or
signal signal_name1:type;
signal signal_name2:type;
```

Internal Signals




```
entity show_sig is
    port (A, B, C, D, CLK: in std_logic;
            Q1, Q2            : out std_logic);
end show_sig;
architecture show_sig_arch of show_sig is
      signal D1, D2 : std_ logic;  -- Not visible outside architecture
begin
```

Attributes

- Provide additional information about many VHDL objects
- Can be assigned to most objects including signals, variables, architectures and entities
- Many attributes are predefined by VHDL, however user defined attributes are also allowed
- VHDL pre-defines five kinds of attributes, dependent on the return value type which can be:

– Value

– Function

– Signal

– Type

– Range

Value Attributes

- `right - Returns right most value in array
- `left - Returns left most value in array
- `high - Returns highest index of an array
- `low - Returns lowest index of an array
- `length - Returns the length of an array
- `ascending - Returns Boolean true if array is ascending. i.e. The array is a to array

Value Examples

```
signal demo_array : std_logic_vector ( 7 downto 0 );

signal length_integer : integer := demo_array `length;

signal hi_int,low_int:integer;
signal A_bit, B_bit : std_logic;

demo_array <= "10001000";
A_bit <= demo_array'right;  -- A_bit = 0
B_bit <= demo_array'left;   -- B_bit = 1
hi_int<=demo_array'high;     --hi_int=7
low_int<=demo_array'low;    --low_int=0
-- Note length_integer = 8.  It was pre-assigned.
```

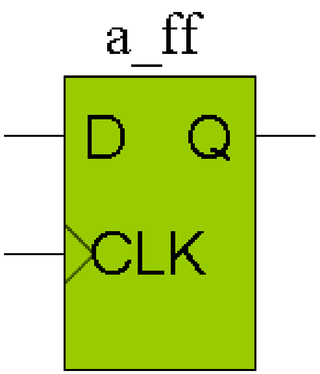
Function Attributes

- `event - Returns true if the signal had an immediate event on it
- `active - Returns true if the signal had a scheduled event on it in the current cycle
- `last_event - Returns time since the last event on a signal
- `last_value - Returns the value of a signal prior to an event
- `last_active - Returns the time since the last scheduled event on a signal

Function Example

- Using the `event attribute to make a clocked circuit

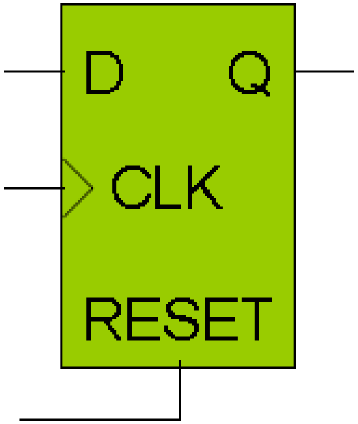
```
library IEEE;
use IEEE.std_logic_1164.all;
entity a_ff is
   port (    D, CLK: in std_logic;
             Q         : out std_logic  );
end a_ff;
architecture a_ff_arch of a_ff is
begin
          process (CLK)
          begin
            if CLK'event and CLK='1' then
            --CLK rising edge
            Q <= D;
            end if;
          end process;
end a_ff_arch;
```



Rising_edge

- rising_edge is a function pre-defined in the *std_logic_1164 package, falling_edge also defined*

```
process (CLK, RESET)
   begin
       if ( RESET = '1' ) then
          Q <= '0';
       elsif ( rising_edge(CLK) )then  --CLK rising edge
            Q <= D;
       end if;
end process;
```



Note: When reset = 1 CLK'event is not evaluated.