- Hello,
  Please distribute this user and pasword information to your registered students.
  User and Password for **ECE 510dsd** video stream at: www.ocate.edu

  User: perkowski
  Password: vhdl
  Thank you.
  Have a good Spring
  D.
- --

- Doug Harksel
- Chief Video & TV Technician
- OCATE
- Television & Media Services
- Office: 503.725.2226
- Wireless: 503.970.6985
- Fax: 503.725.2201
- doug@ocate.edu
- www.ocate.edu

- On 3 Apr 02, at 16:27, XXX wrote:

> I am taking ECE 510 OC7 under Prof. Perkowski. I need a computer
> account to be setup. Thanks

Hi,
If the last four digits of your ID are: 8963, you are already in our database.

Our records indicate:
- your username is ”xxxx"
- you have an active ECE UNIX account
- you have a pending Windows account

Take photo ID to one of our front-desks to have your Windows account validated. Look at http://www.cat.pdx.edu/users/labs.html to determine the place and time most convenient for you. ("XXXXX" indicates when an attendant is on duty to help you.)

If the numbers above are NOT the last four digits of your ID, you'll need to verify that you're registered for the class. (Logging onto PSU Banner and bringing up your class schedule will be acceptable.) Then, either Peter Phelps, John Jendro, or Kim Howard can add you to the database.

I'm happy to help you, but I may not always be immediately available. For future reference: e-mail to support@cat.pdx.edu reaches a team of people.

Kathy

~~~~~~
Kathy McCauley Damtawe (KatMama) damtawek@cat.pdx.edu
User Services Manager, CECS Computing Support
College of Engineering and Computer Science
Portland State University, Portland, Oregon
~~~~~~

- >Professor Perkowski,
>
>I have a question regarding the week of April 15th. I will be unable to
>attend class on Monday 4/15/2002. I really don't want to miss out on the class
>opportunity. Are/can classes be made available on video tape.

Yes, the classes are videotaped and also available as streamed video.

>
> I am interested in using Veribest Design Capture integrated with
>ModelSim for class projects. This would be beneficial for my work
>interest and interesting since I don't have any practical usage with
>either of these tools. Does this seem acceptible to you?

Yes, this is fine with me, but what project you want to work on? Please think about it and write me a proposal.

Friday’s meetings will be perhaps streamed as well.

You are not restricted to the projects that I specified

Projects will be better explained, but you can start reading now

- >However, the projects listed in your class
  >seem very challenging,
  Remember that I will be explaining them in detail in the class. I just wanted to list them now so interested people can start reading on their own.
  The projects are not trivial but based on my 12 years of teaching this class they are doable
  Also, you can propose your own project and create group of students to work with you. We have so many students that in any case I want to have more projects
  > I am not sure that I can understand everything there.
  It will be explained and more slides will be added. Students will make presentations on these topics using PPT in class
  >Are you assigning teams for each project?
  No, you create teams and inform me. But there is no hurry now, the projects will start in about 2 -3 weeks from now.
  > also, what are subject of the two homeworks listed in your web?
  On the web you have examples of previous homeworks. HOmeworks for this year will be announced in the class.
  Sincerely
  Marek

Last question and answer…..

Dear Dr.Perkowski,

On your webpage,the grading of the VHDL Class stipulates 2 HWS and a Project. But when I look at the 'slides from the lectures' on the webpage,its has some five homeworks.
Nelson

You can choose any of the homeworks that are posted or do something similar.

If you choose one of previous homeworks, you have to solve the problem from scratch rather than copy from previous students. Changing symbol names is not enough.

Project must be explained, all your ideas and methodology, Kmaps, schematics, etc.

Every student will have to do two homeworks. In these homeworks he or she will have to prove ability to simulate and synthesize logic circuits using VHDL or Verilog.

Copyrighted Material

- Some of the materials used in this course come from ARPA RASSP Program and are copyright
  - **R**apid Prototyping of **A**pplication **S**pecific **S**ignal **P**rocessors Program
  - http://rassp.scra.org
- Some other of materials are copyright K. J. Hintz
- Some other from J. Wakerly.
- All sources will be acknowledged.

Review

- Please review the following material from Lecture 1:
  - 1. D, T, and JK flip-flops
  - 2. Shift operations using flip-flops and muxes
  - 3. Design of a generalized register with arbitrary set of operations
  - 4. Register transfer statements that involve several generalized registers and simple control.
  - 5. Karnaugh Maps.
  - 6. Sorter versions as examples of combinational, pipelined and sequential circuits.

All this material will be reviewed again on Friday.

Lecture 2

Documentation and Timing Diagrams

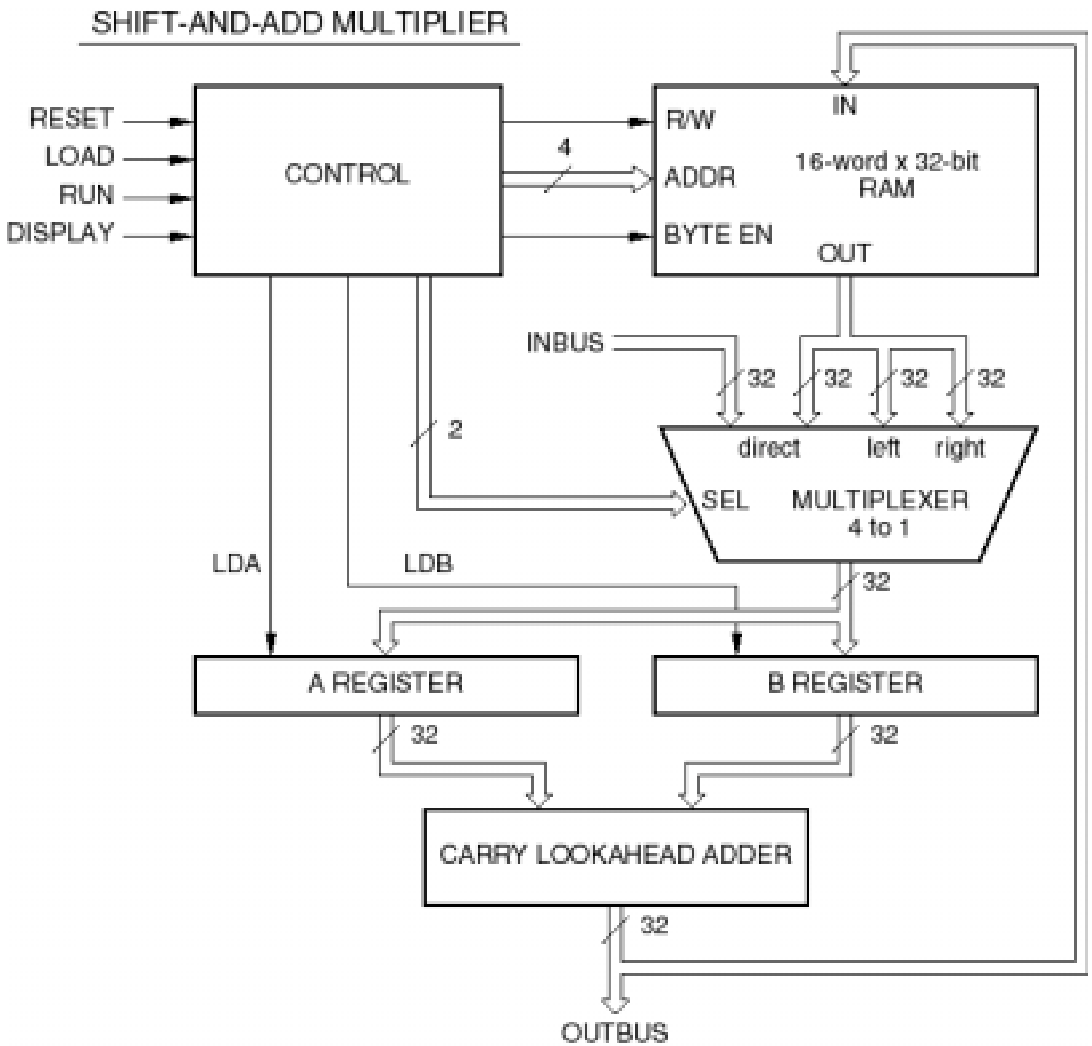
Lecture Goals

- Introduce documentation standards.
- Explain basic logic gates
- Explain basic logic blocks.
- Explain basic technologies.

Documentation Standards

- Block diagrams
  - first step in hierarchical design
- Schematic diagrams
- HDL programs (ABEL, Verilog, VHDL)
- Timing diagrams
- Circuit descriptions

Block Diagram



In homeworks and projects you need to give a complete documentation, not only VHDL or Verilog code.

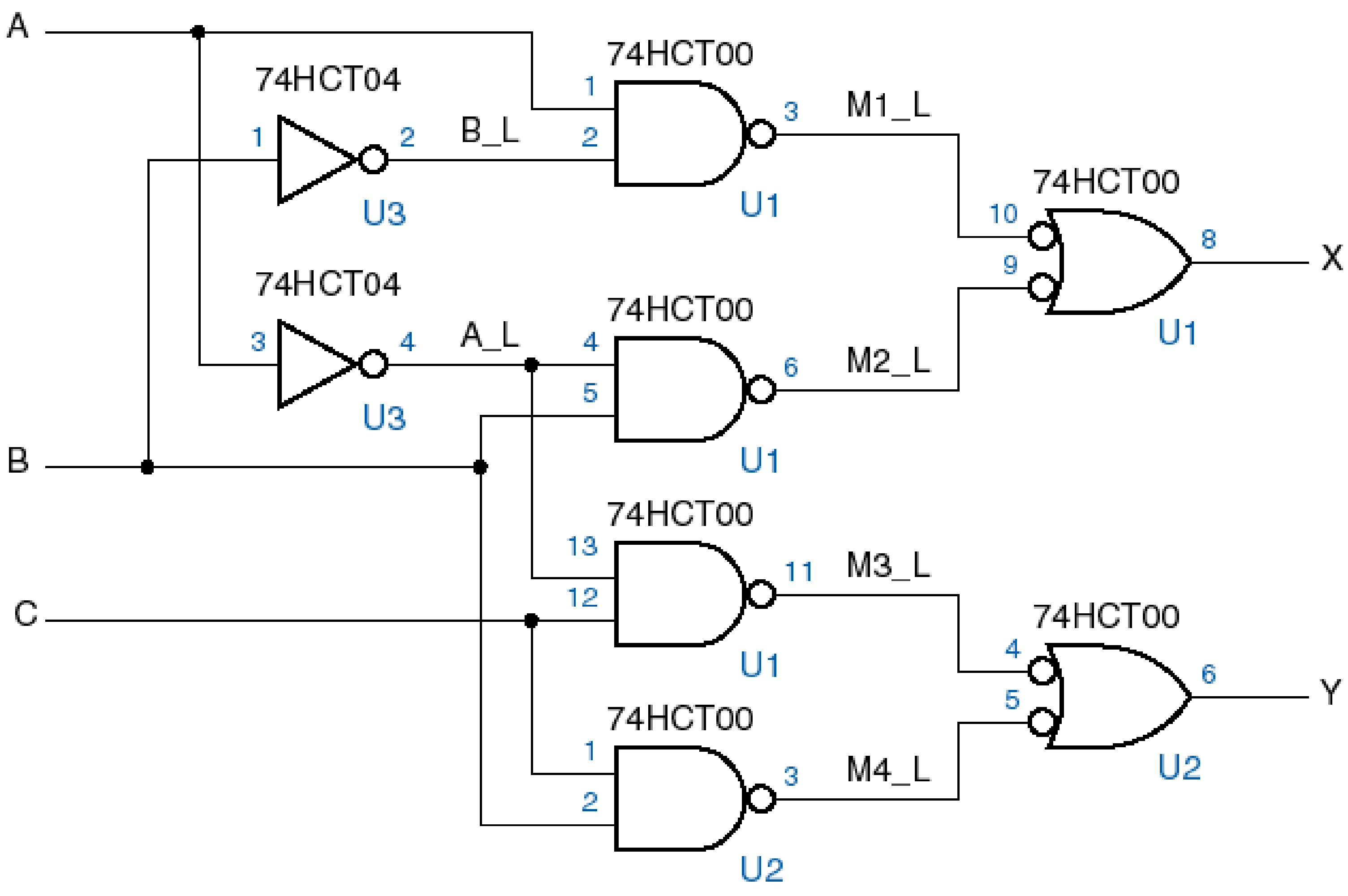
Your ideas must be also clearly explained together with design goals.

Schematic diagrams

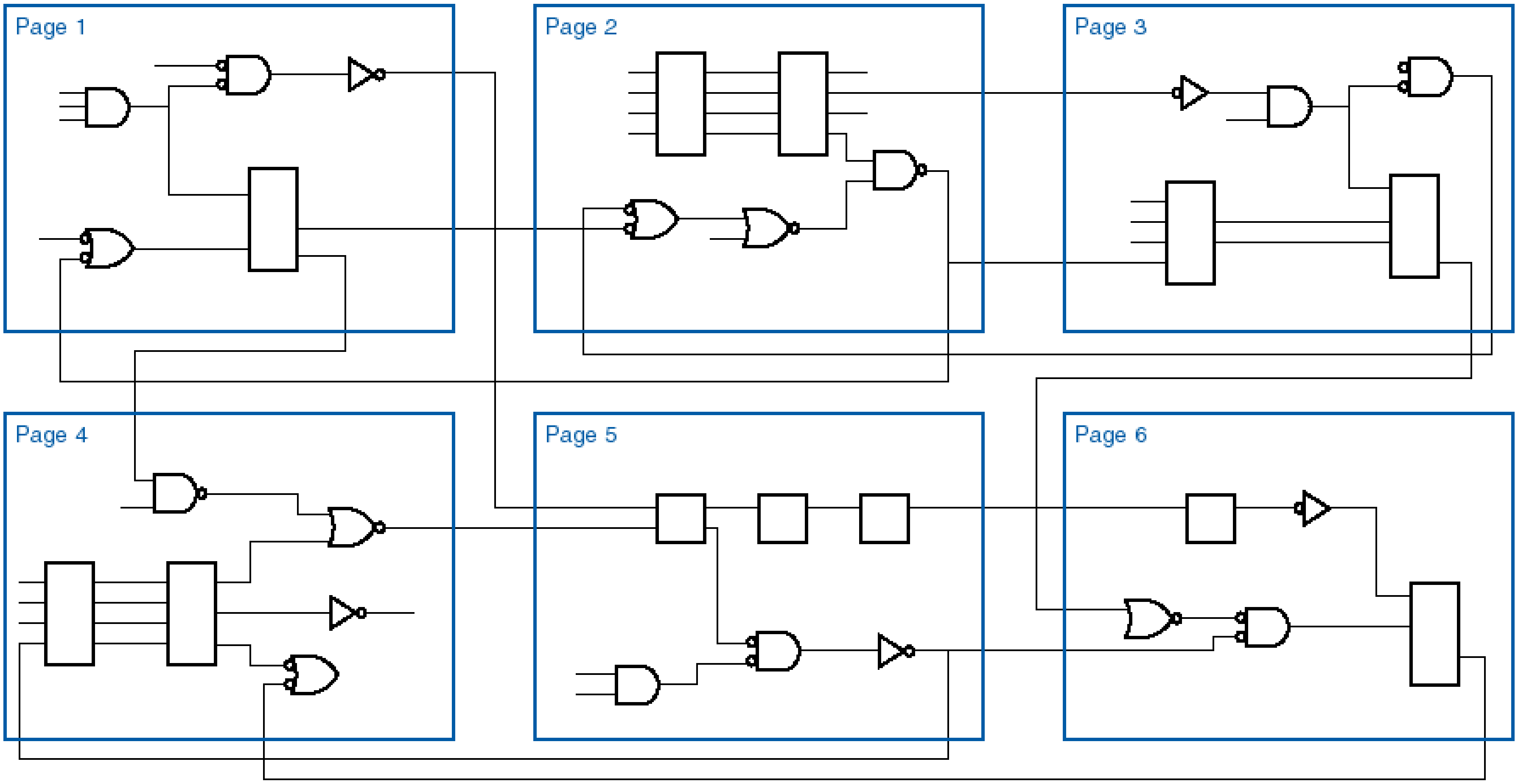
- Details of component inputs, outputs, and interconnections
- Reference designators
- Pin numbers
- Title blocks
- Names for all signals
- Page-to-page connectors

Use names that have some meaning, like *addr4*

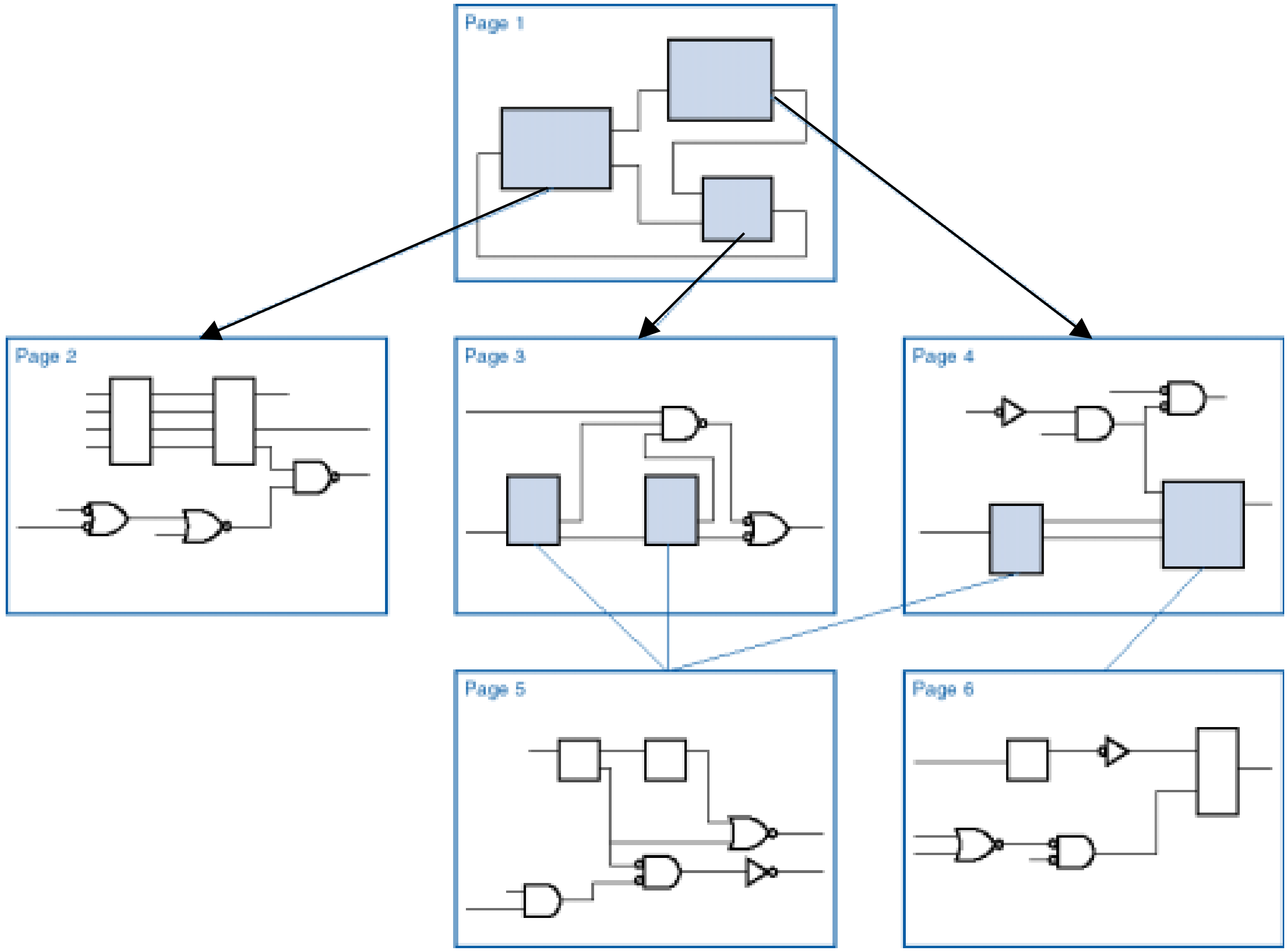
Example schematic



Flat Schematic Structure



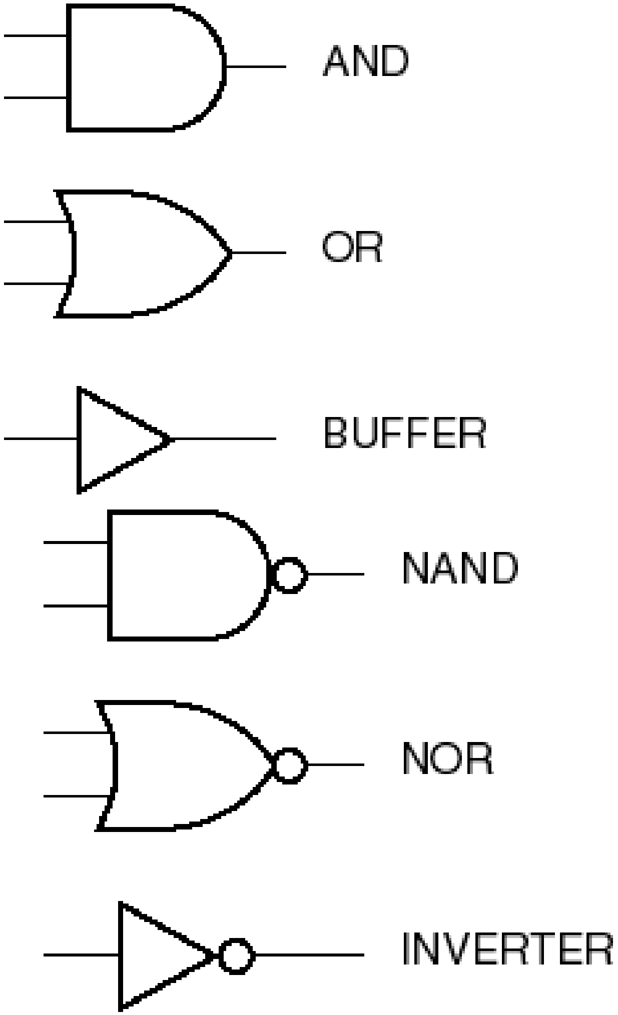
Hierarchical Schematic Structure



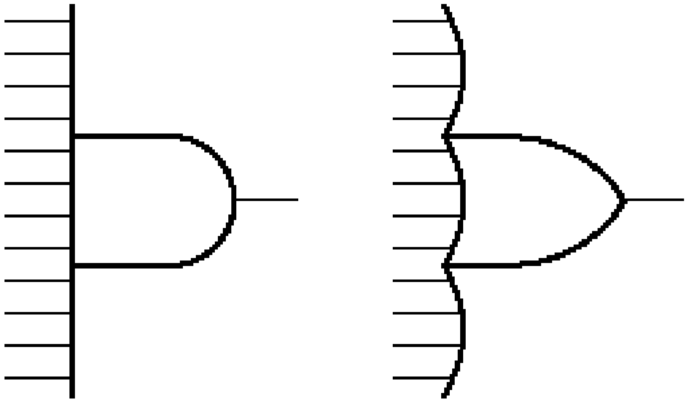
Other Documentation

- Timing diagrams
  - Output from simulator
  - Specialized timing-diagram drawing tools
- Circuit descriptions
  - Text (word processing)
  - Can be as big as a book (e.g., typical Cisco ASIC descriptions)
  - Typically incorporate other elements (block diagrams, timing diagrams, etc.)

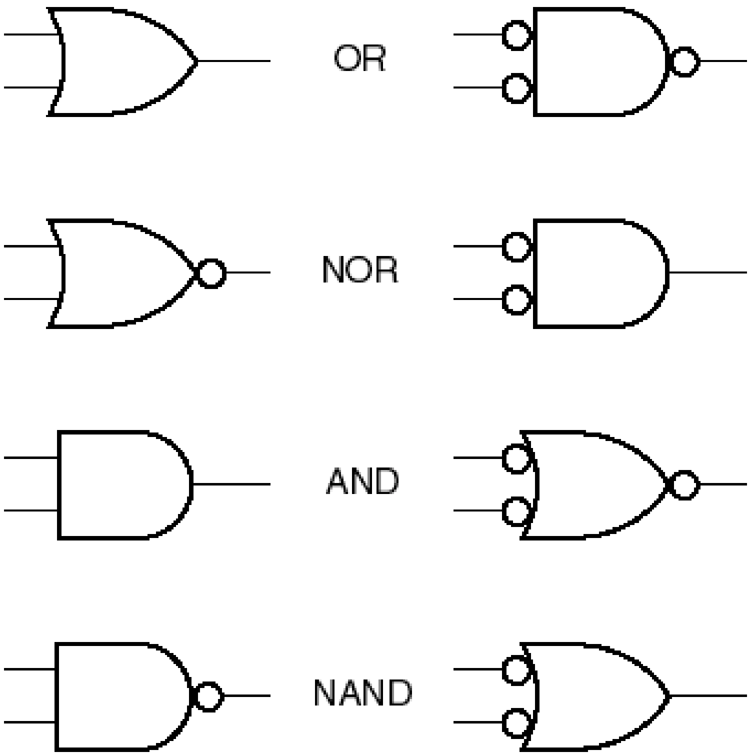
Gate symbols

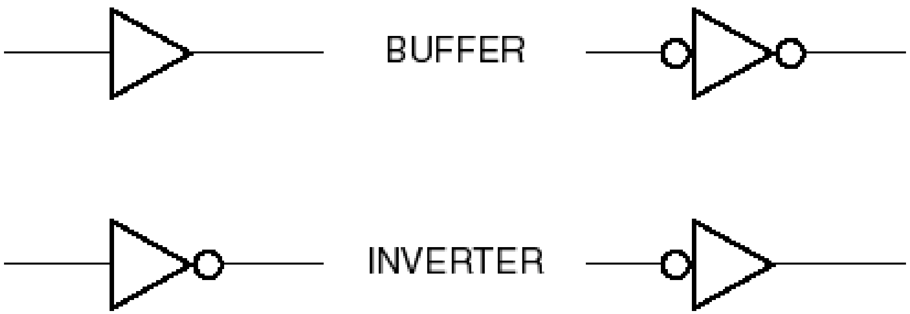


You must be able to write a truth table and a Kmap for every gate that you are using



DeMorgan Equivalent Symbols





Which symbol to use?

Answer depends on
signal names and active levels.

Please review these equivalencies using truth tables and formulas

Signal Names and Active Levels

» Signal names are chosen to be descriptive.

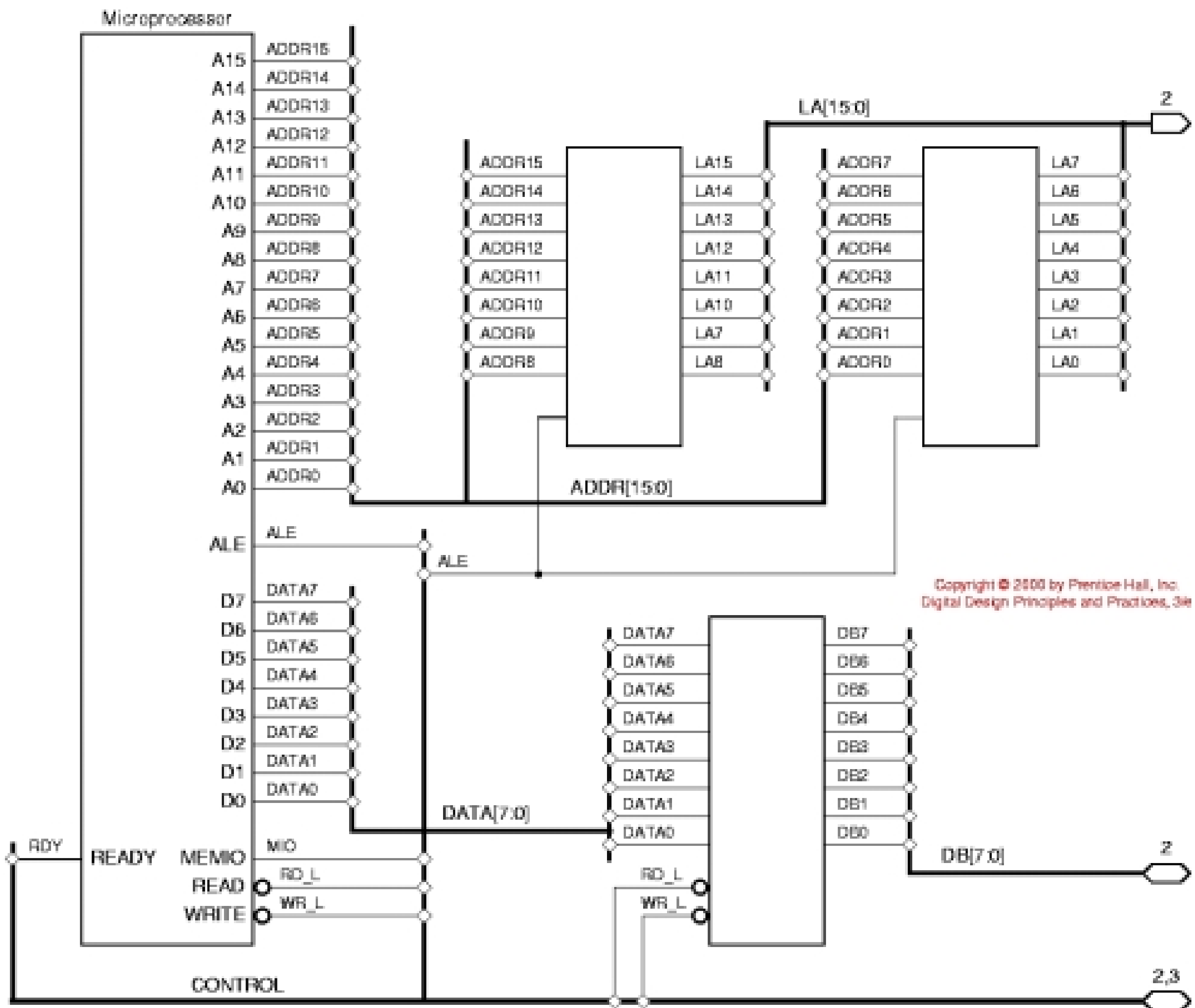
» Active levels -- HIGH or LOW

- named condition or action occurs in either the HIGH or the LOW state, according to the active-level designation in the name.

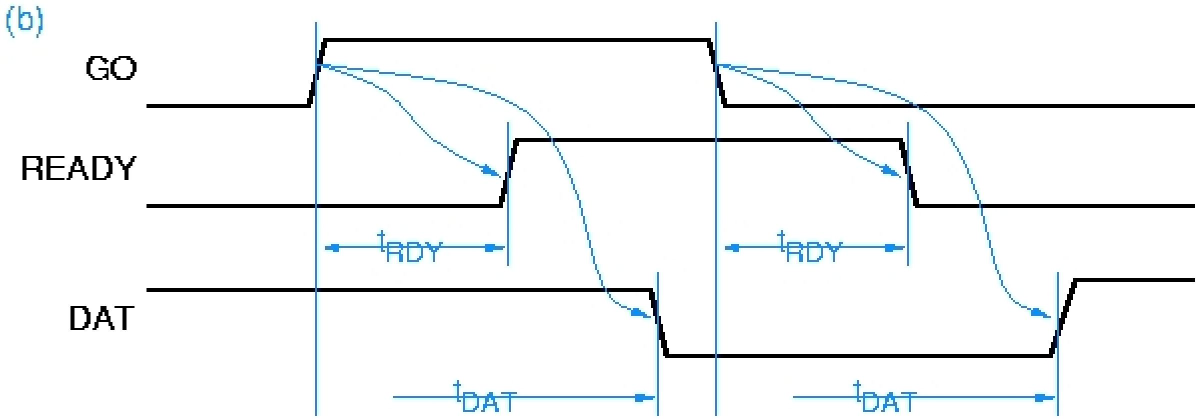
Active low

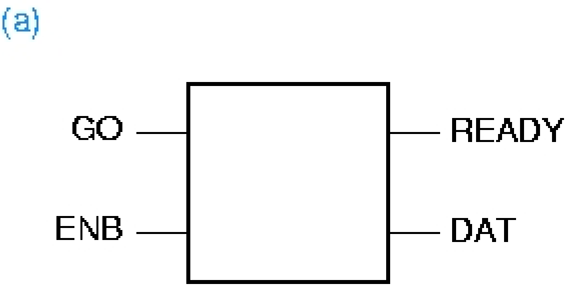
Active Low	*Active High*
READY–	READY+
ERROR.L	ERROR.H
ADDR15(L)	ADDR15(H)
RESET*	RESET
ENABLE~	ENABLE
~GO	GO
/RECEIVE	RECEIVE
TRANSMIT_L	TRANSMIT

Examples of Buses



Timing Diagrams



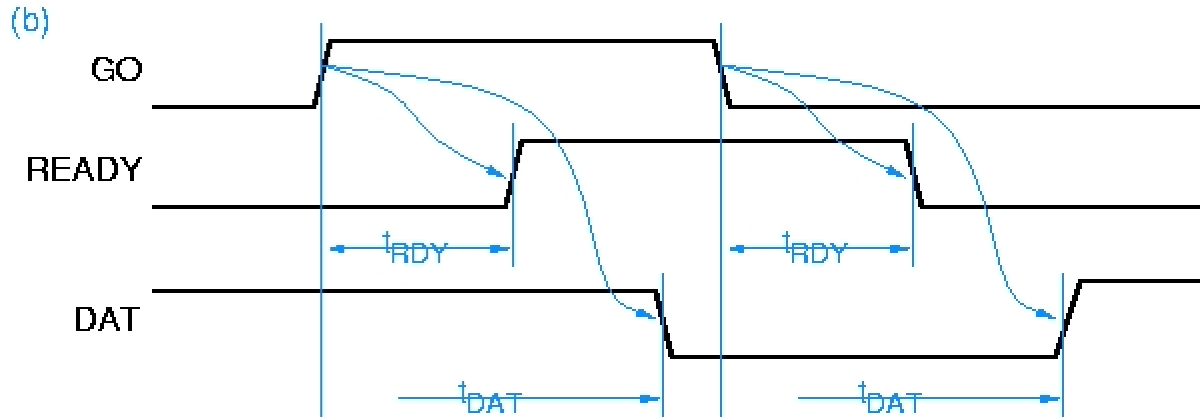


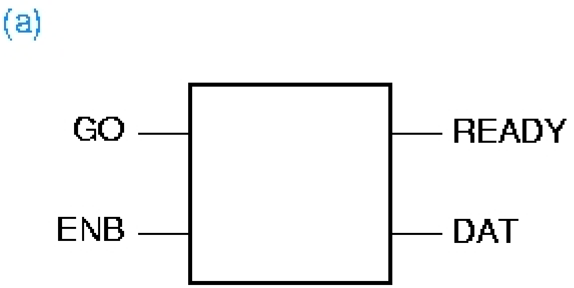
This is taken from Wakerly, page 331

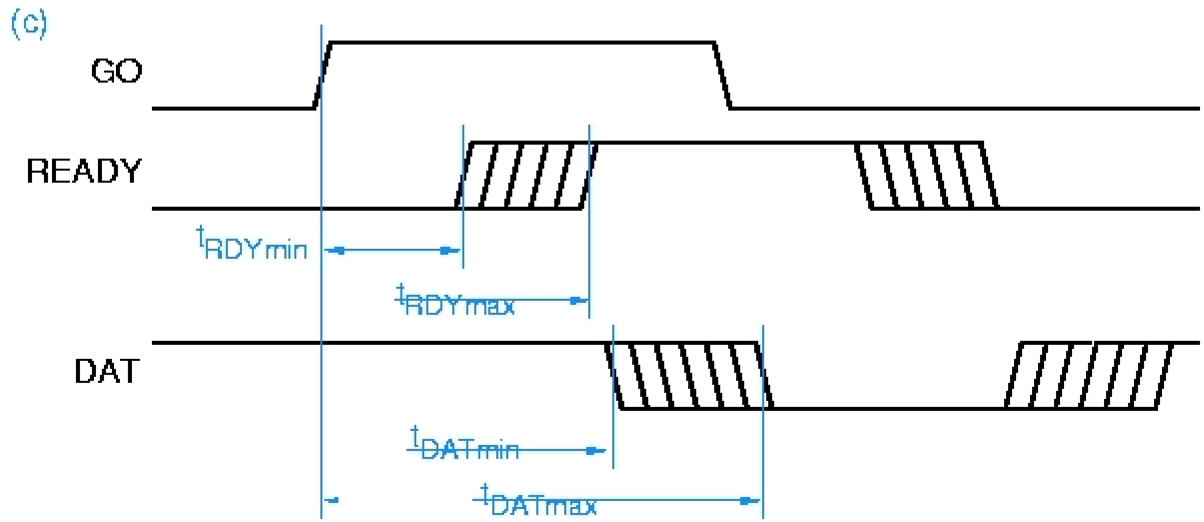
Timing Diagrams

b) causality and propagation delay

c) minimum and maximum delays

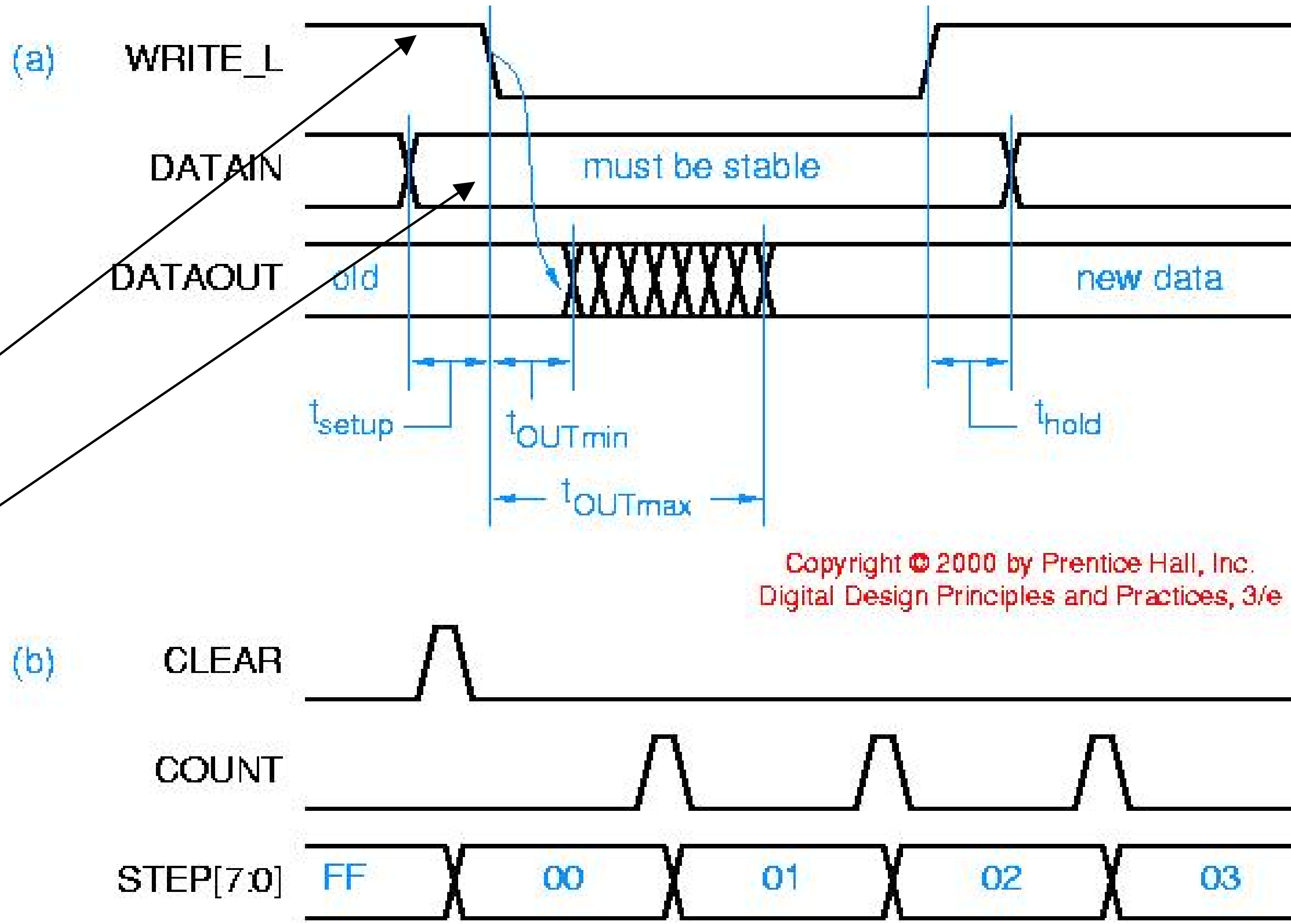






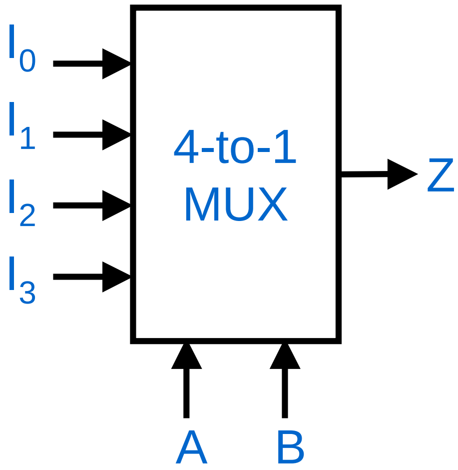
Bus Timing Diagram

Timing diagrams for “data” signals, (a) certain and uncertain transitions, (b) sequence of values on an 8-bit bus

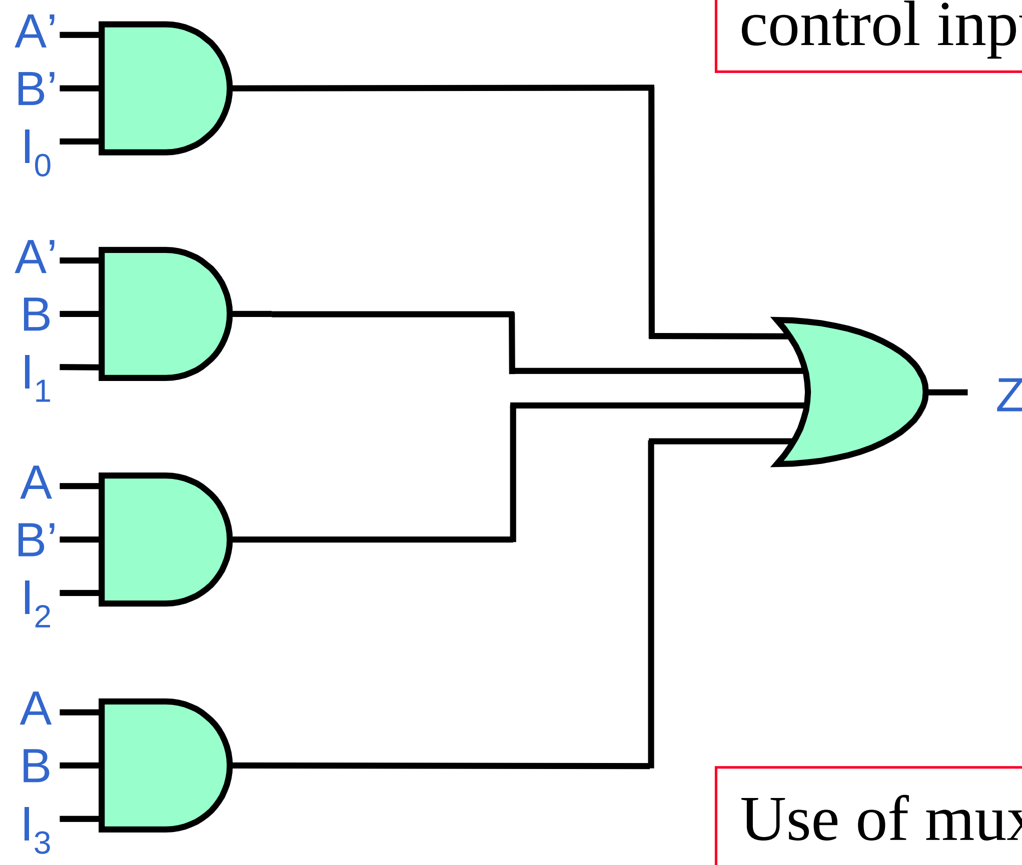


Multiplexers

Data inputs versus control inputs



A	B	Z
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

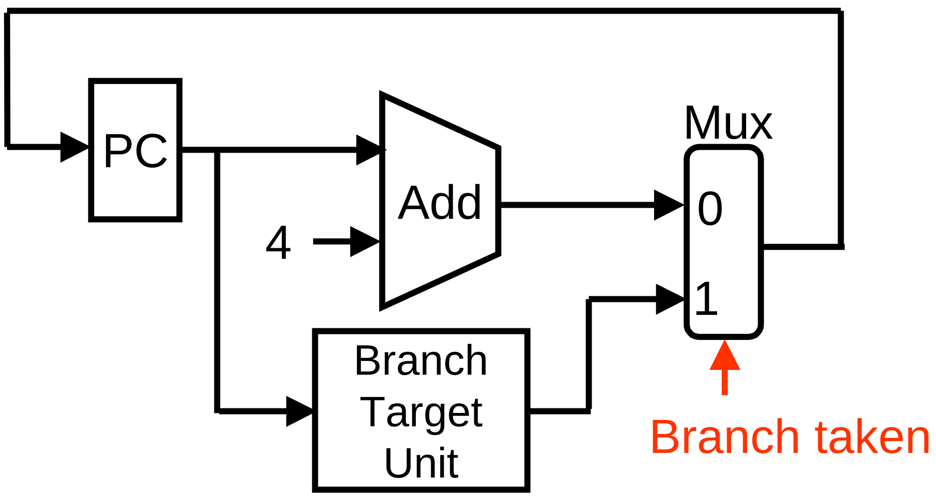


Use of muxes in control and data path

A typical use of a MUX in a processor control path

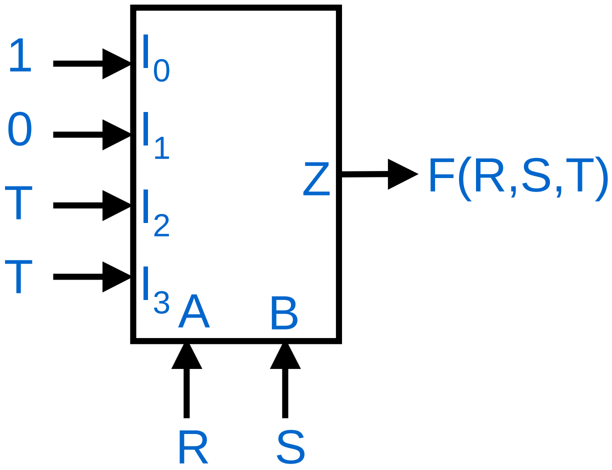
Consider the following sequence of instructions:

0x7F800	add $16, $18, $15	# reg16 ← reg18 + reg15
0x7F804	beq $8, $0, target	# if reg16 == 0 goto target
0x7F808	sub $17, $17 $15	# reg17 ← reg17 - reg15



Recall our example about systematically designing data path for a set of register-transfer operations

A 4-to-1 MUX can implement any 3-variable function



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

Example: Implement the function
F(R, S, T) = R'S' + RT

F(R,S,T) = R'S'•1 + RT•(S+S')

= R'S'•1 + RST + RS'T

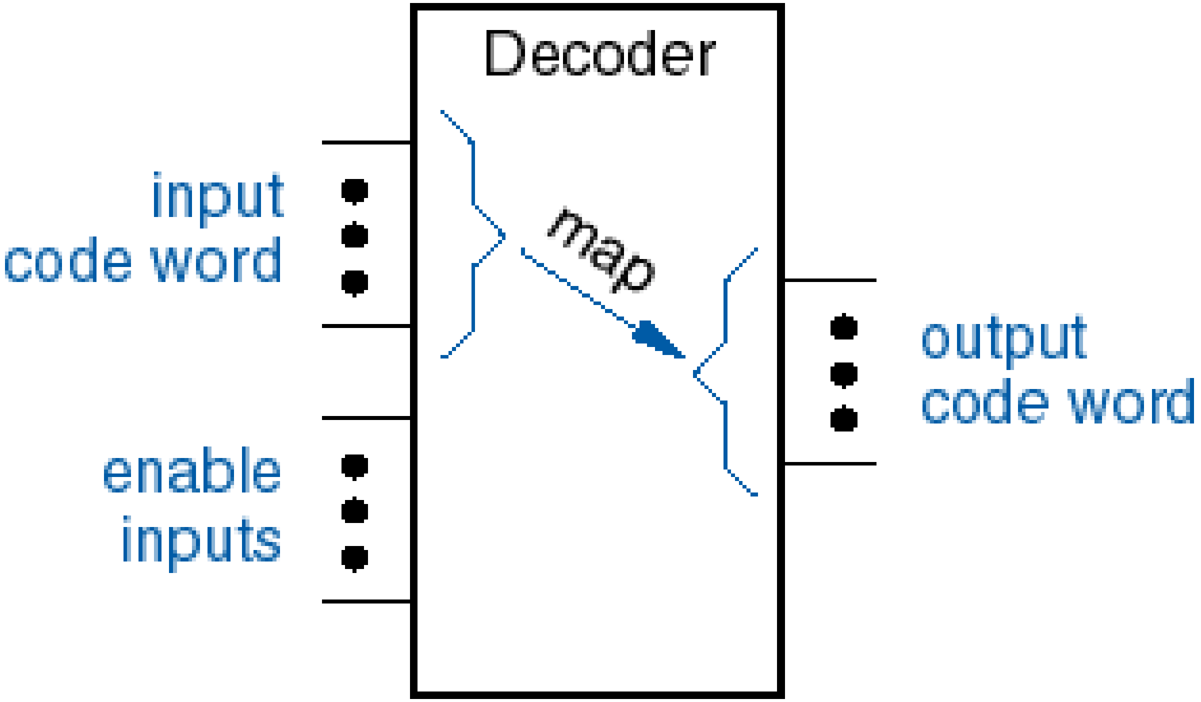
Functions of how many input variables can be implemented by an 8-t0-1 MUX?

Use an 8-t0-1 MUX to implement the function:
F(X,Y,Z,T) = XY' + Z'T

Drawing Kmaps is useful for such problems

Decoders

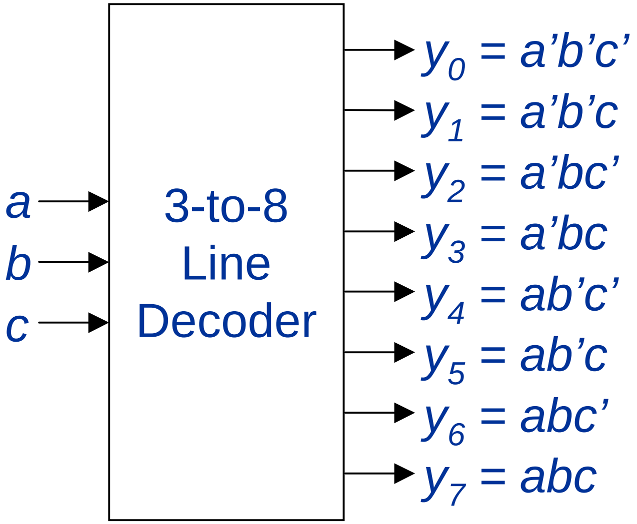
– General decoder structure



– Typically n inputs, 2^n outputs

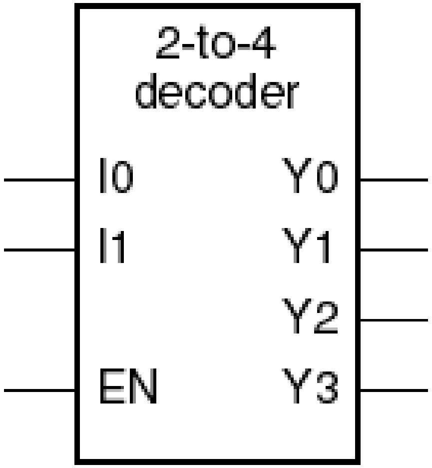
– 2-to-4, 3-to-8, 4-to-16, etc.

Decoders



a	b	c	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Binary 2-to-4 decoder

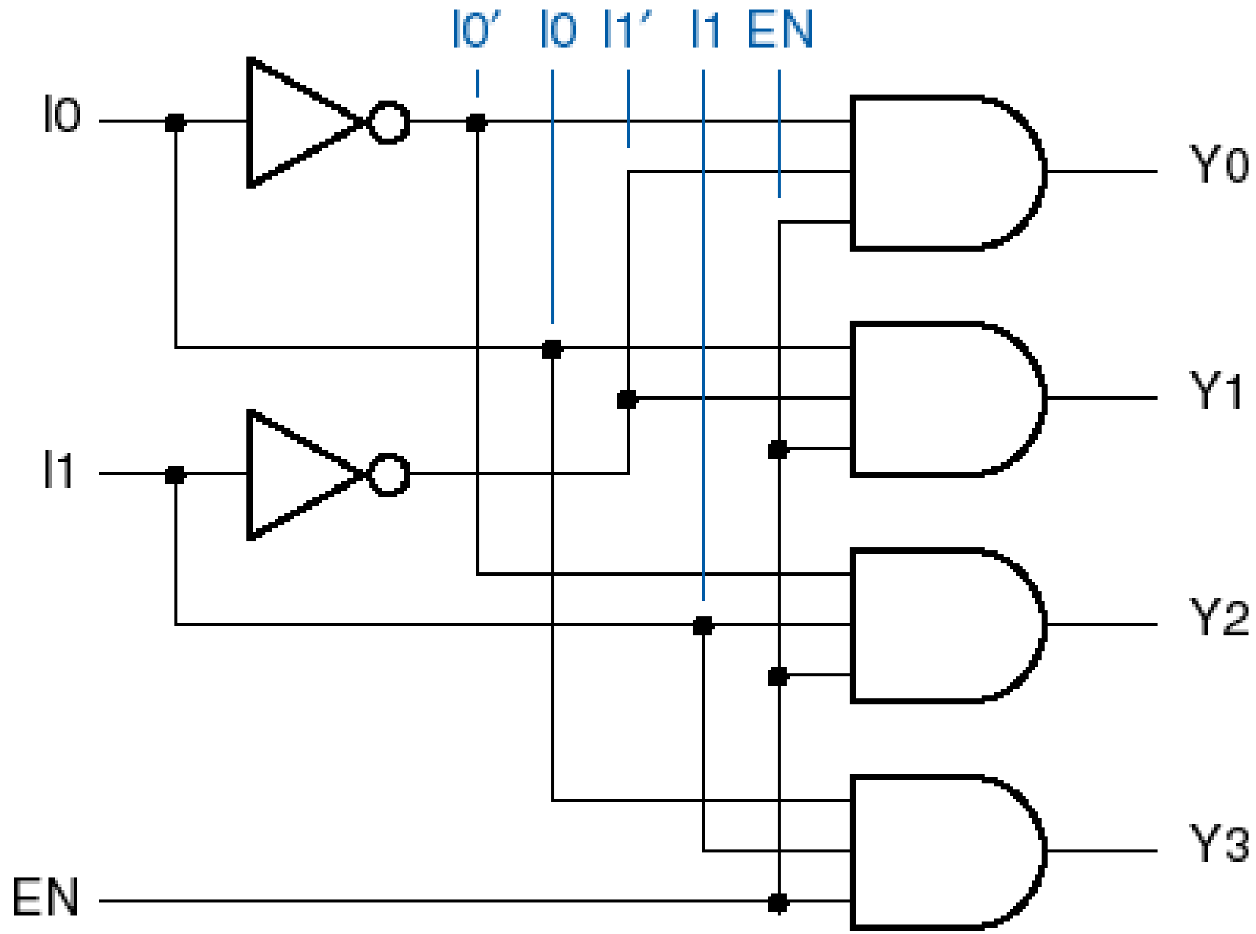


Inputs			*Outputs*			
EN	I1	I0	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

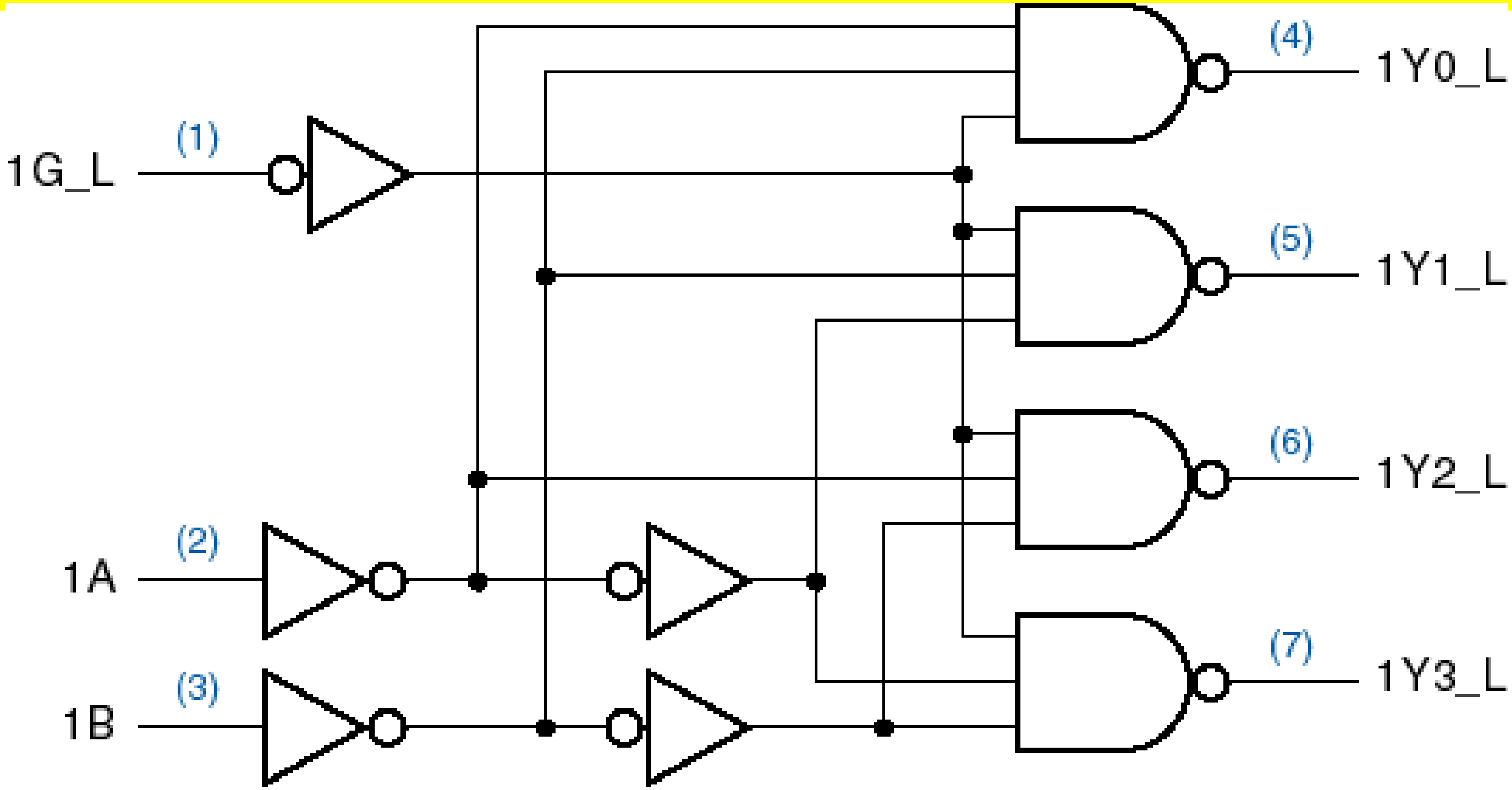
Note “x” (don’t care) notation.

You have to understand various interpretations of don’t care

2-to-4-decoder logic diagram

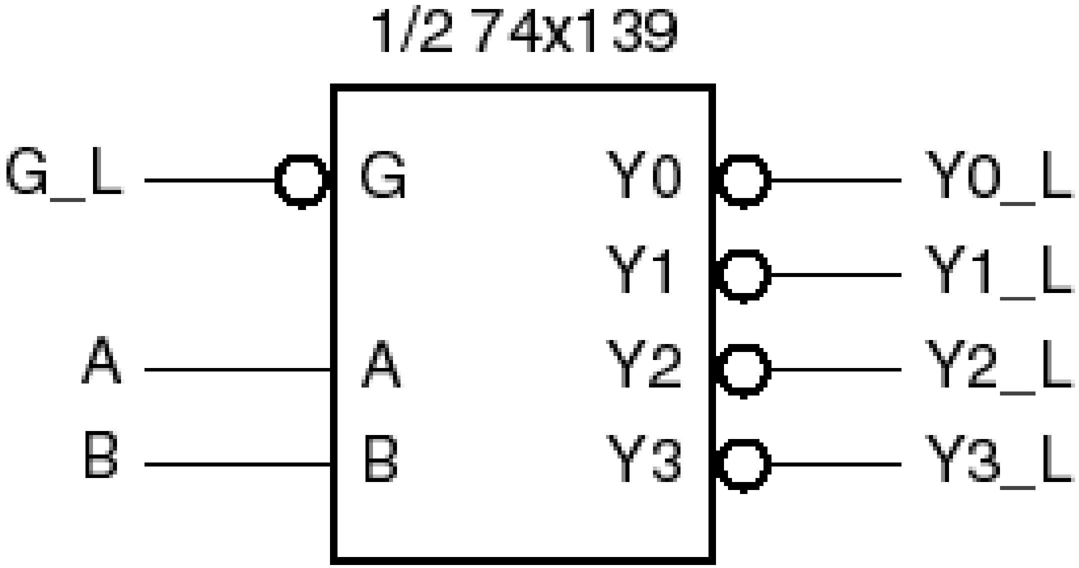


MSI 2-to-4 decoder

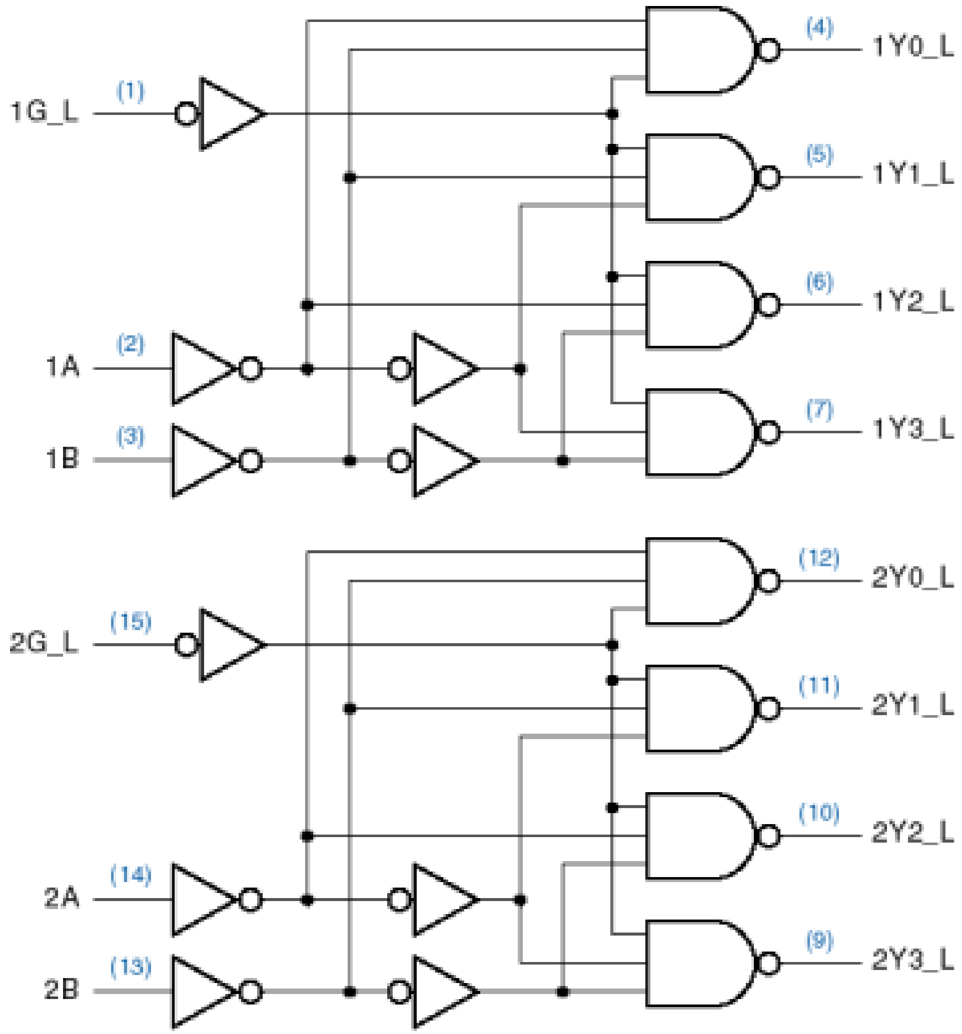


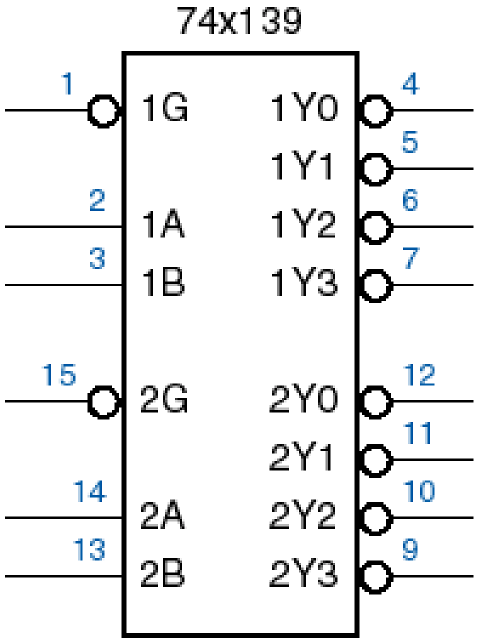
- Input buffering (less load)
- NAND gates (faster)

Decoder Symbol

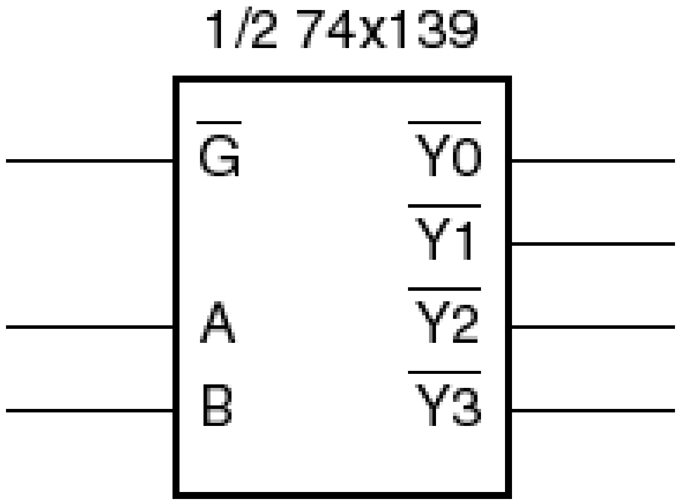


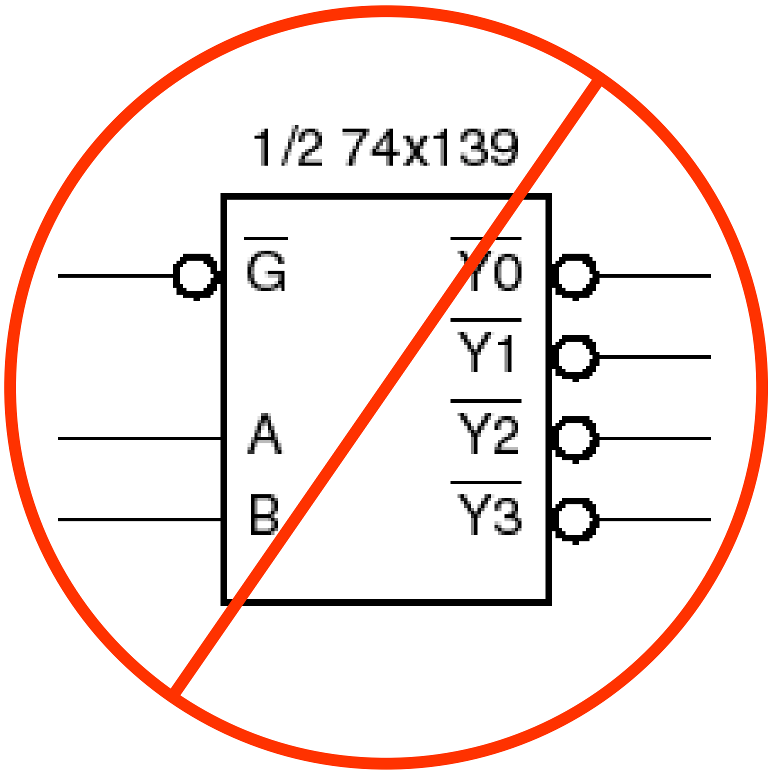
Complete 74x139 Decoder



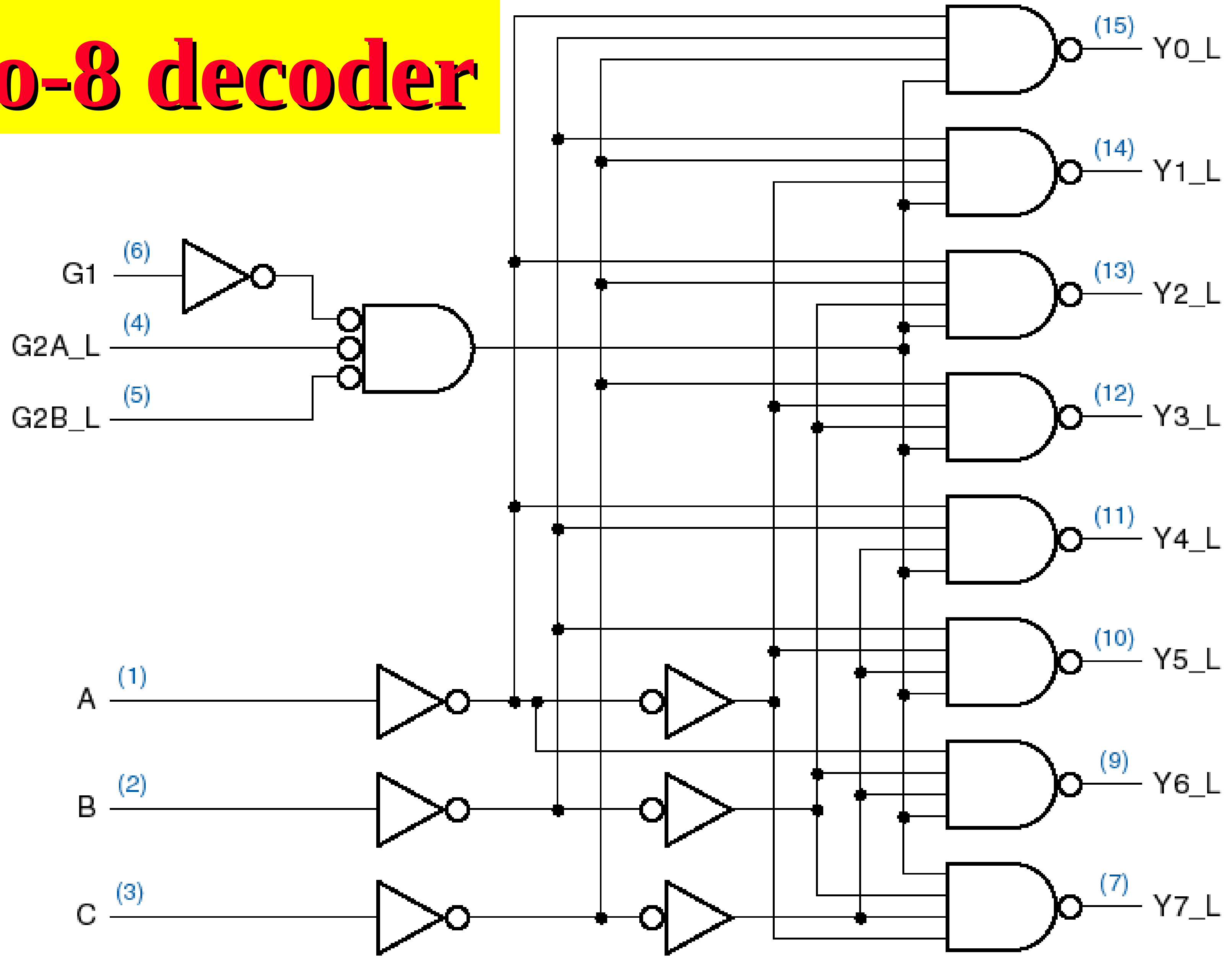


More decoder symbols

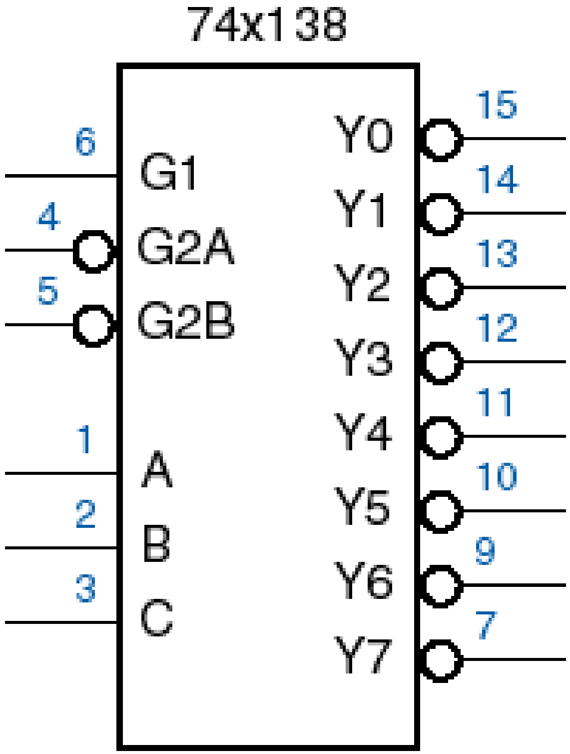




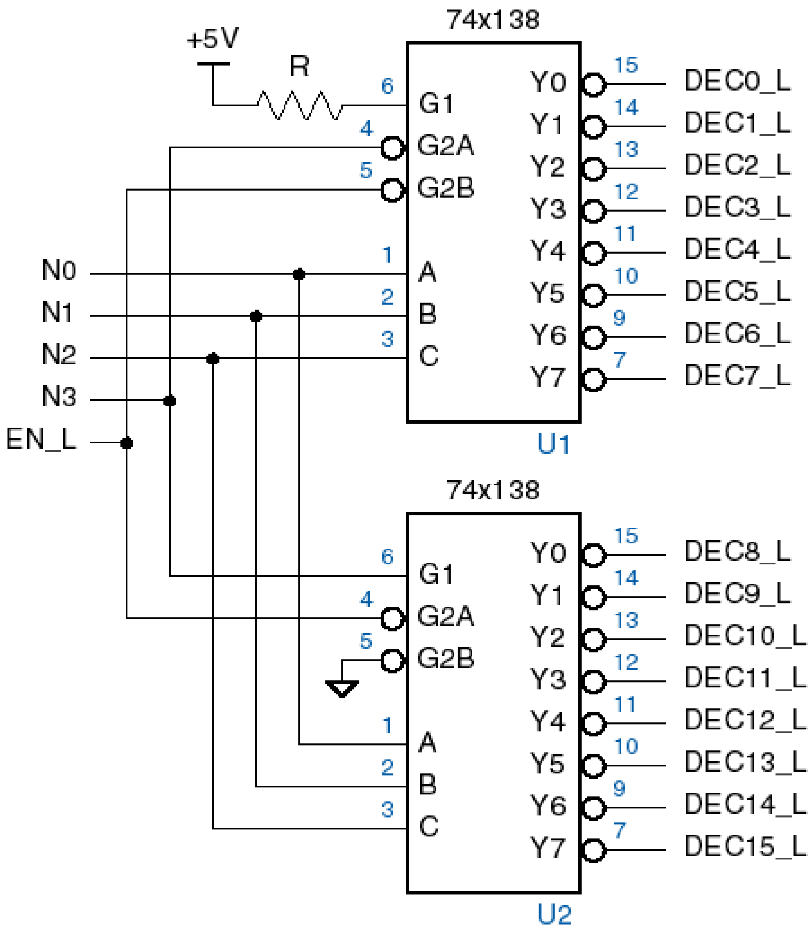
3-to-8 decoder



74x138 3-to-8-decoder symbol

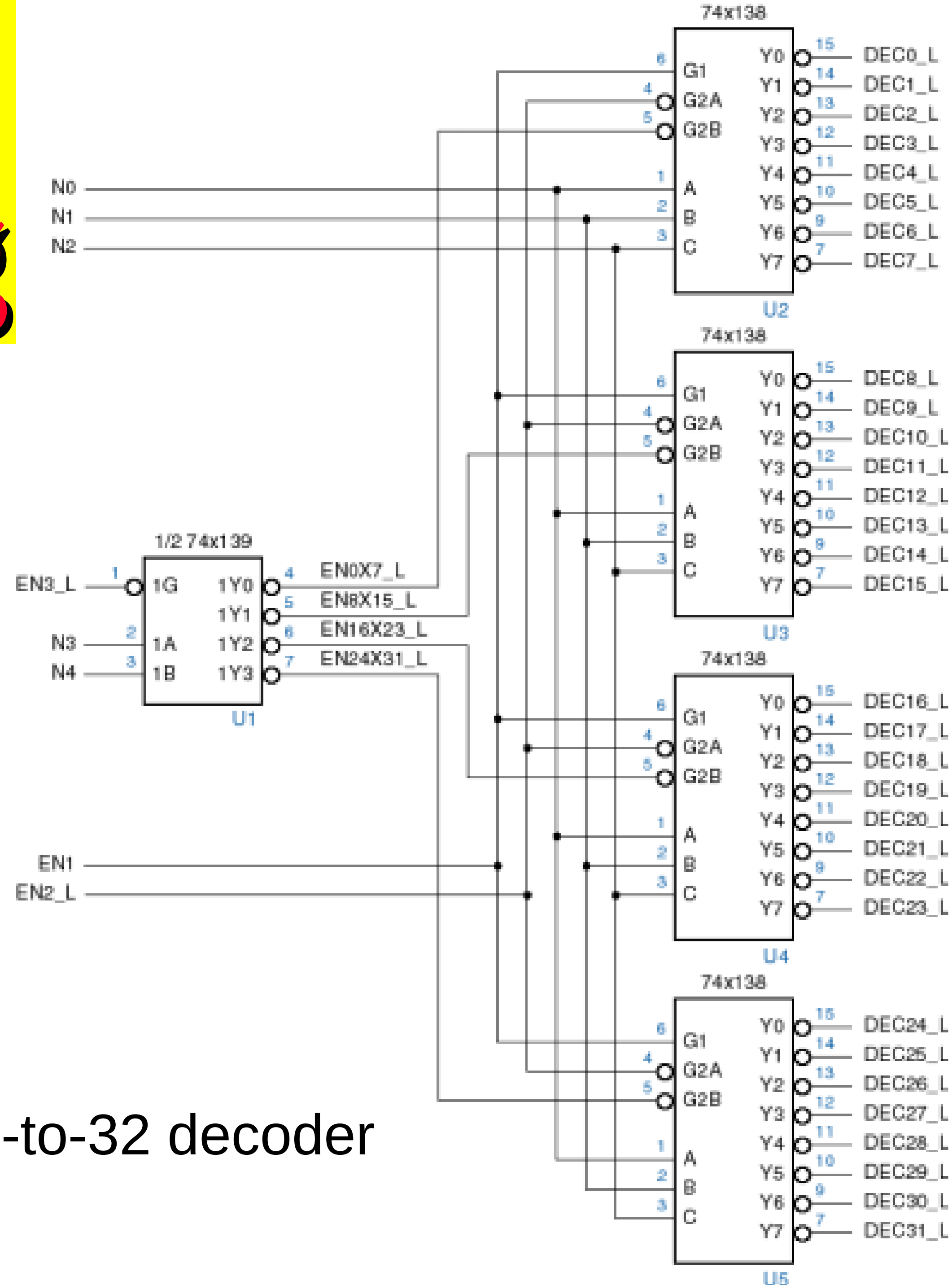


Decoder Cascading



4-to-16 decoder

More Cascading

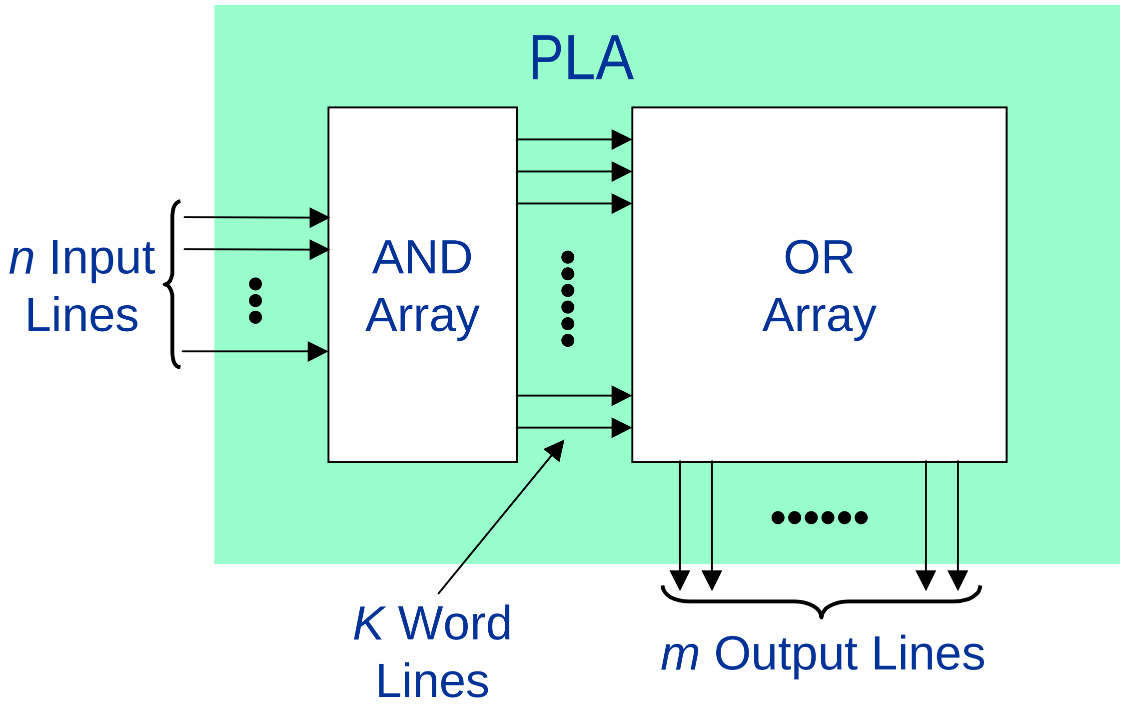


5-to-32 decoder

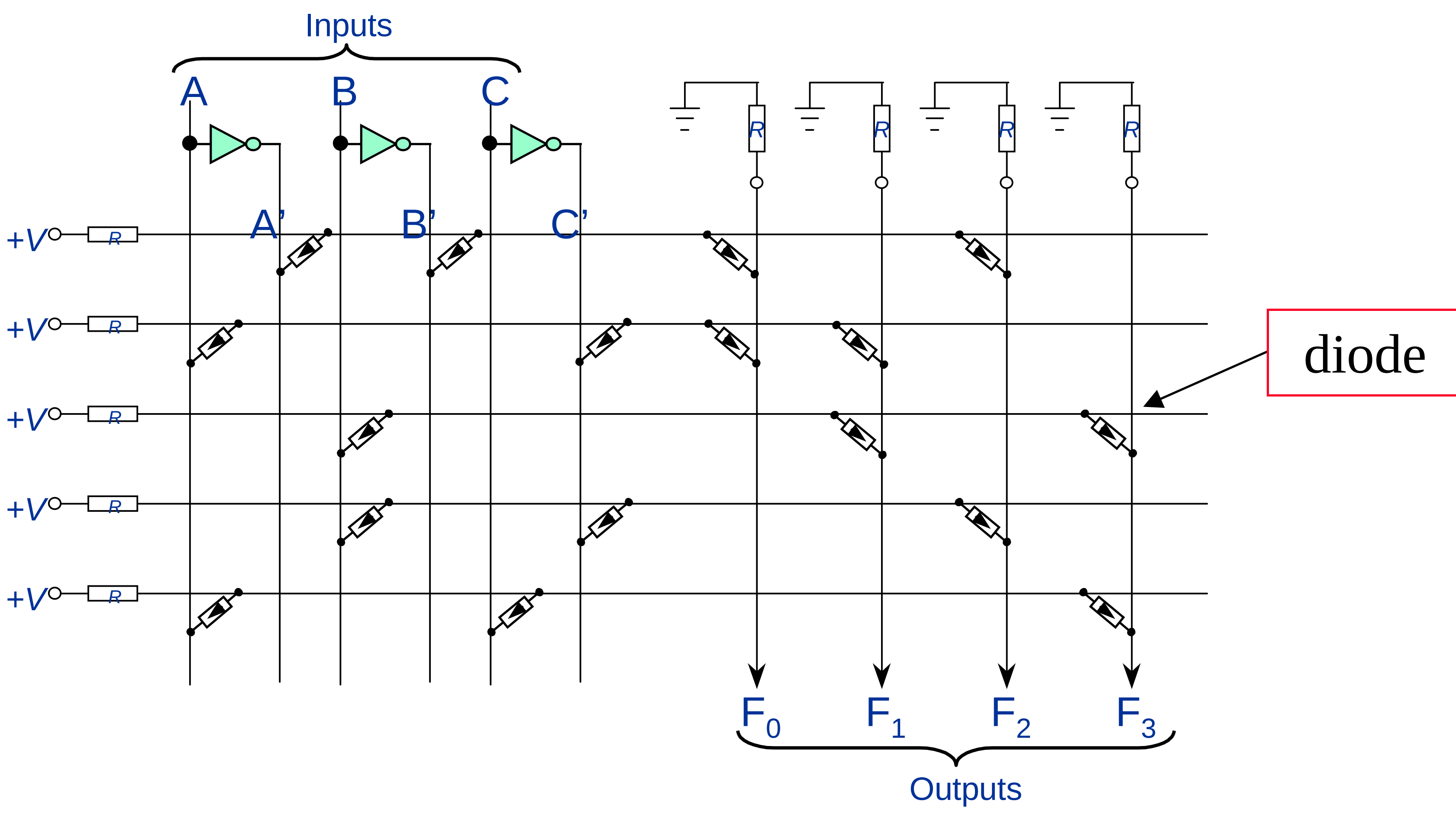
Decoder applications

- Microprocessor memory systems
  - selecting different banks of memory
- Microprocessor input/output systems
  - selecting different devices
- Microprocessor instruction decoding
  - enabling different functional units
- Memory chips
  - enabling different rows of memory depending on address
- Lots of other applications

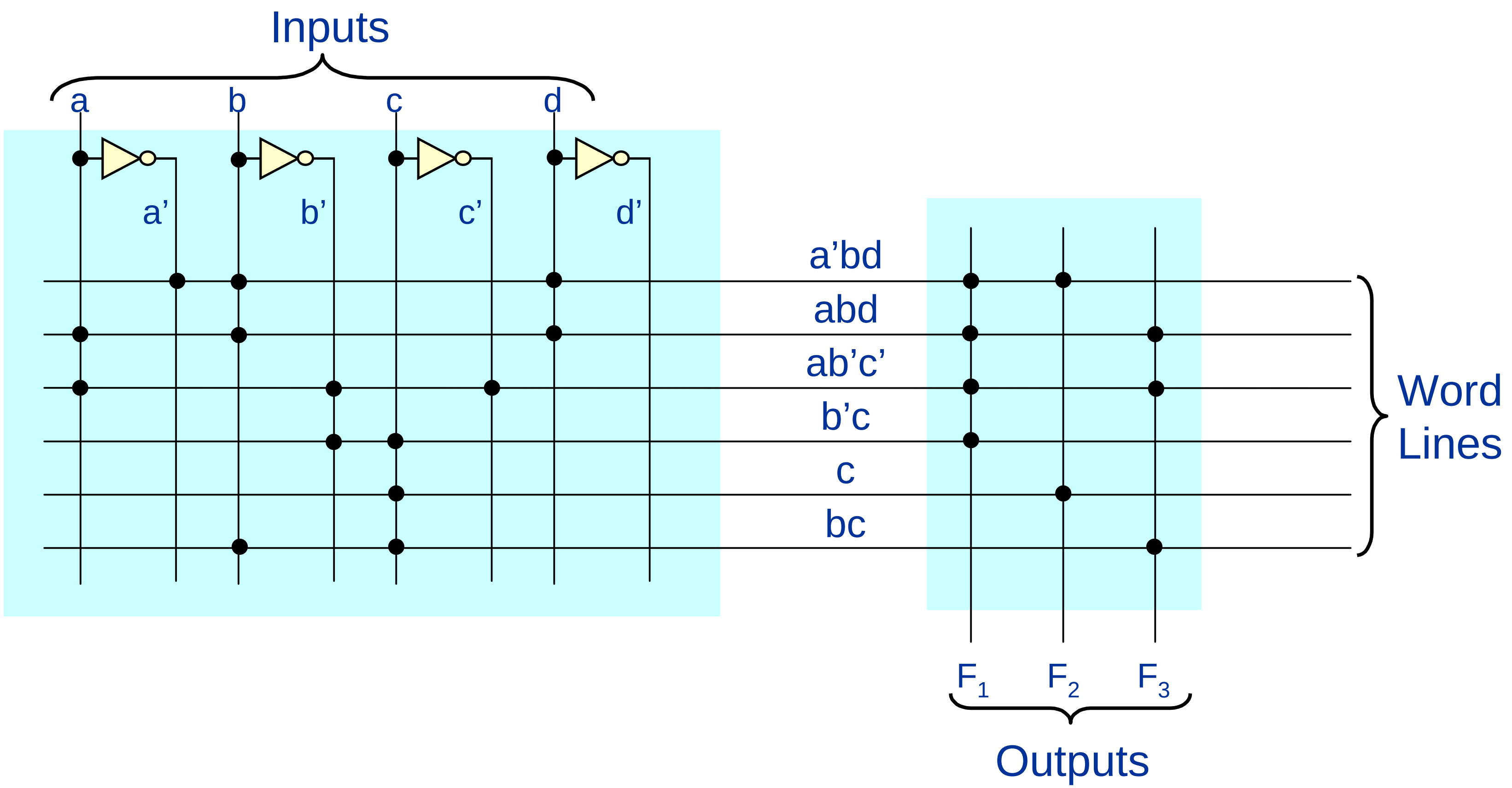
Programmable Logic Array Structure



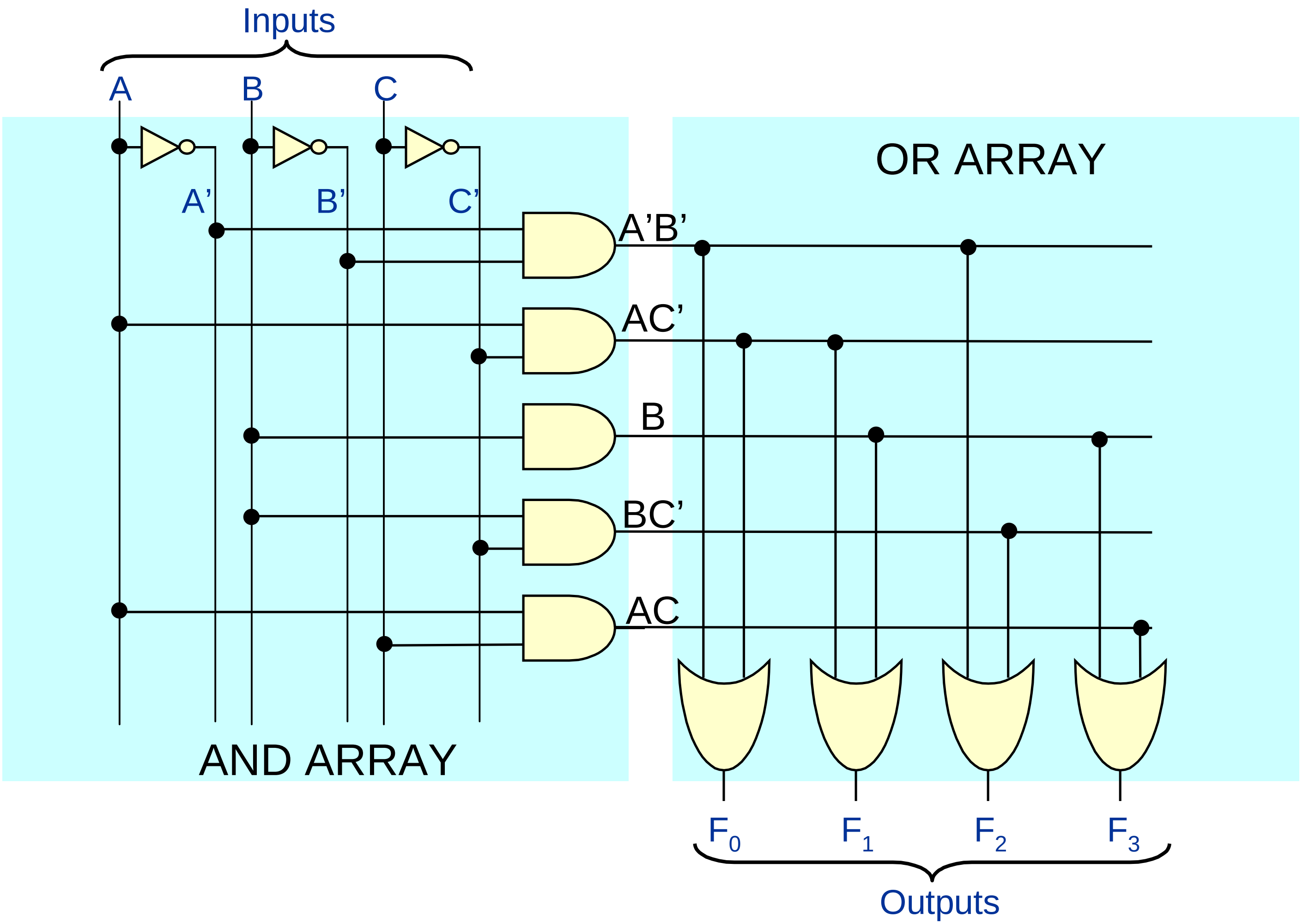
Internal Structure of a PLA



Internal Structure of a PLA



Internal Structure of a PLA



Programmable Logic Arrays (PLAs)

– Idea: Build a large AND-OR array with lots of inputs and product terms, and programmable connections.

  » *n* inputs

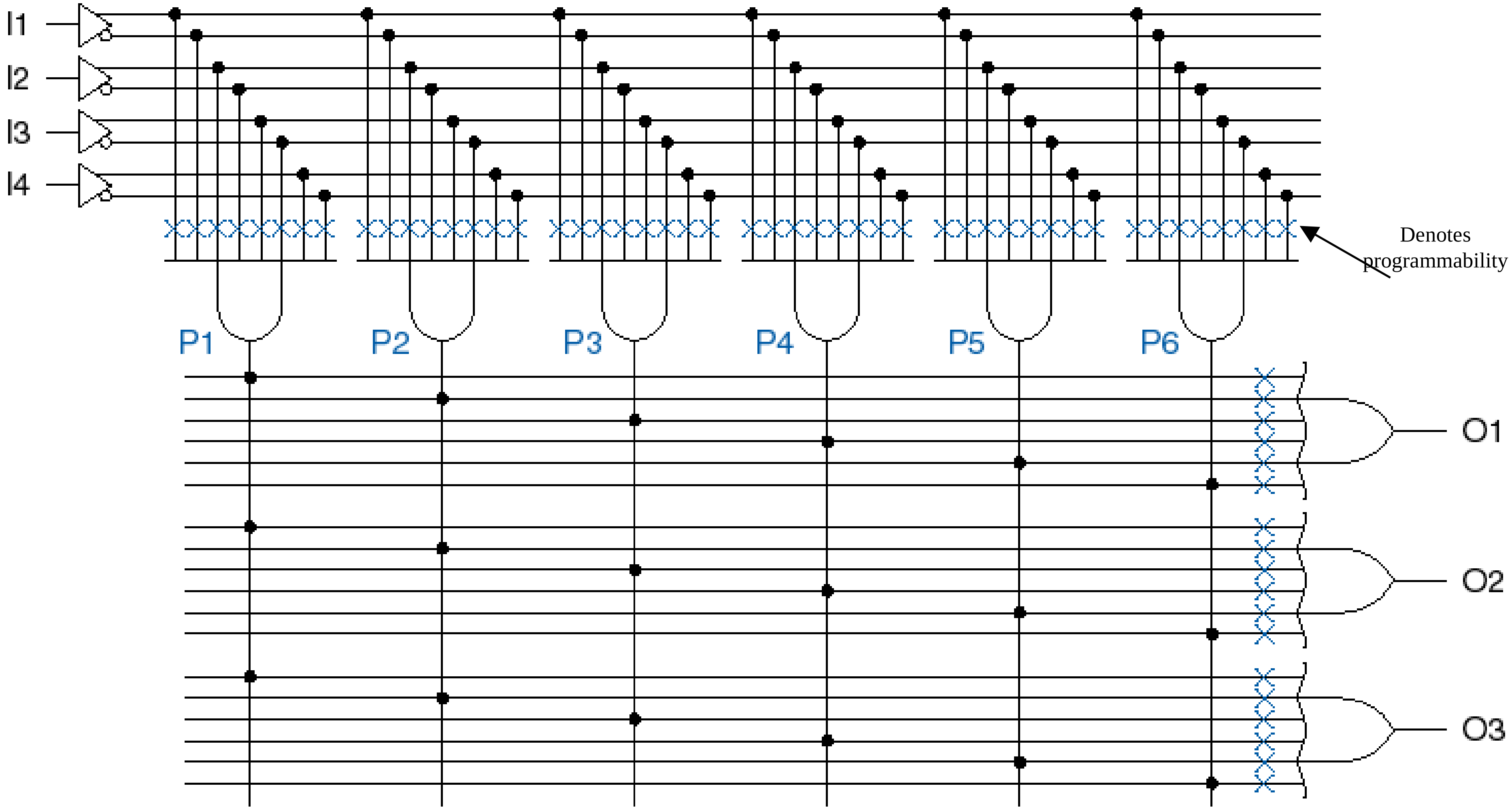
    • AND gates have 2*n* inputs -- true and complement of each variable.

  » *m* outputs, driven by large OR gates

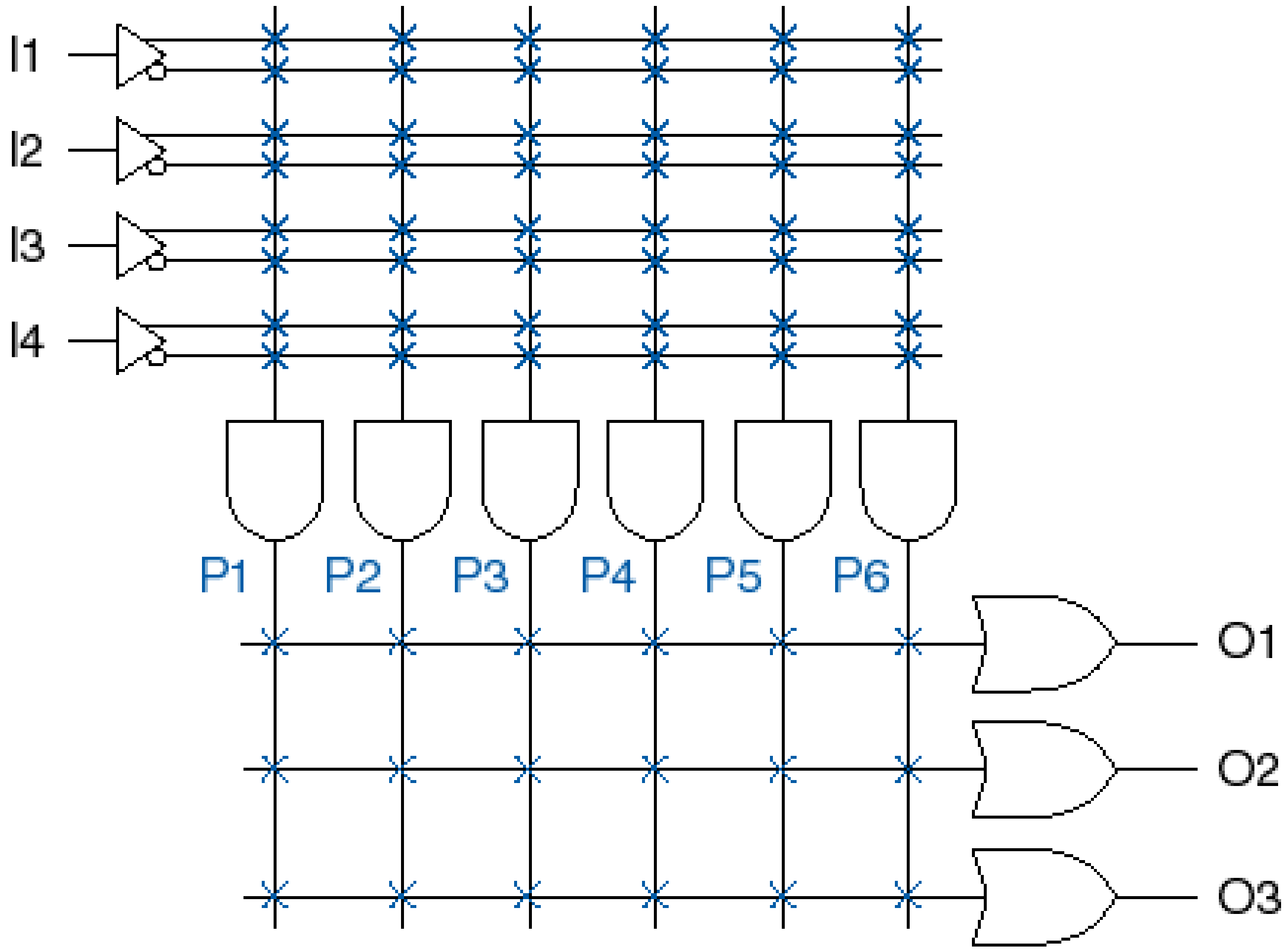
    • Each AND gate is programmably connected to each output's OR gate.

  » *p* AND gates ($p \ll 2^n$)

Example: 4x3 PLA, 6 product terms

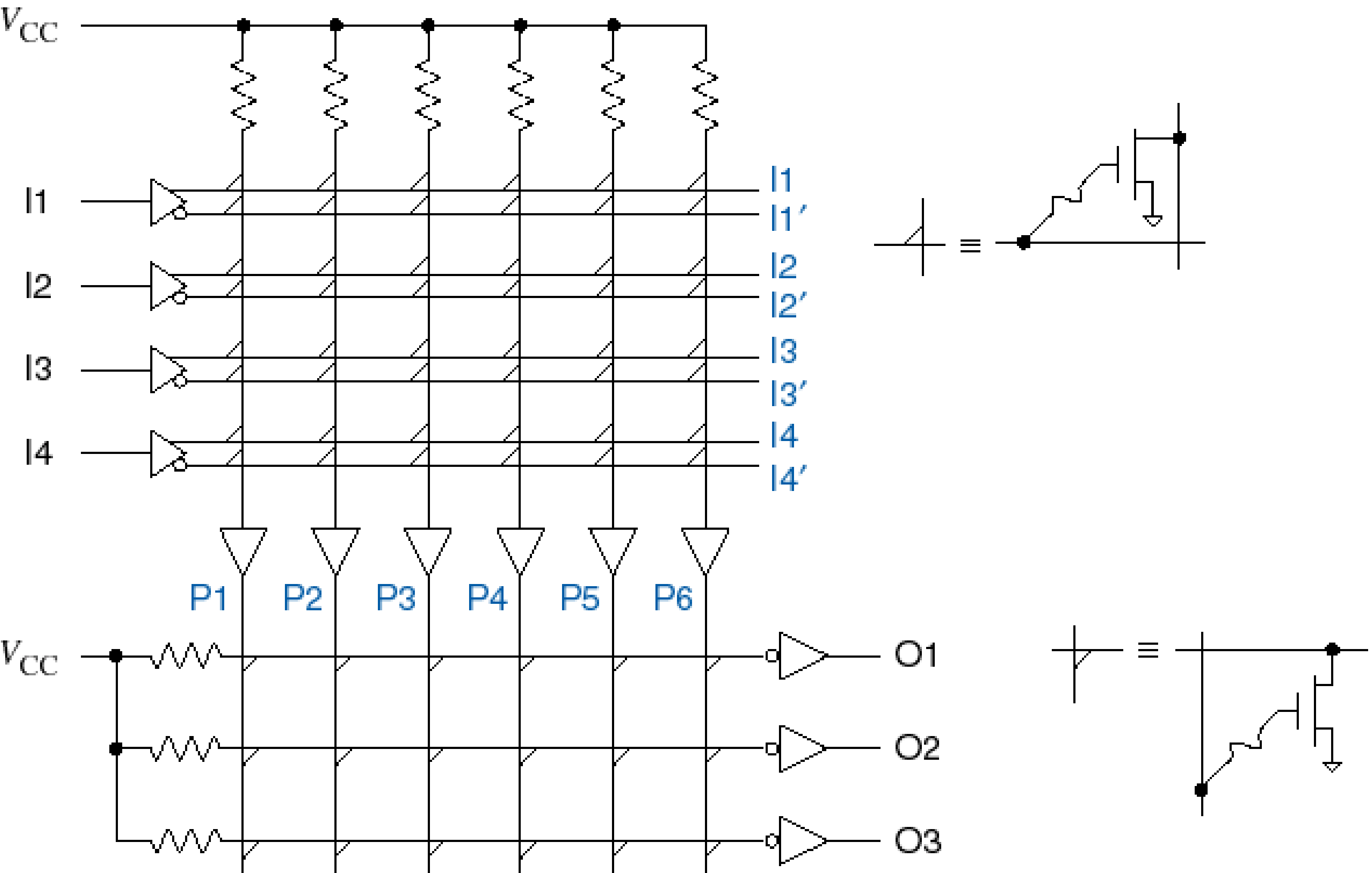


Compact Representation



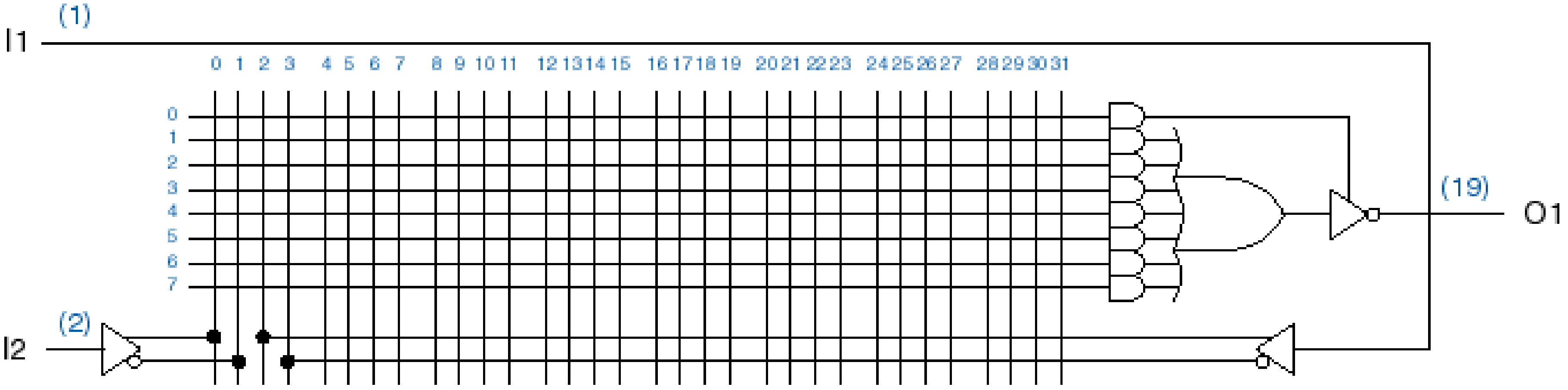
PLA Electrical Design

- See Section 5.3.5 -- wired-AND logic



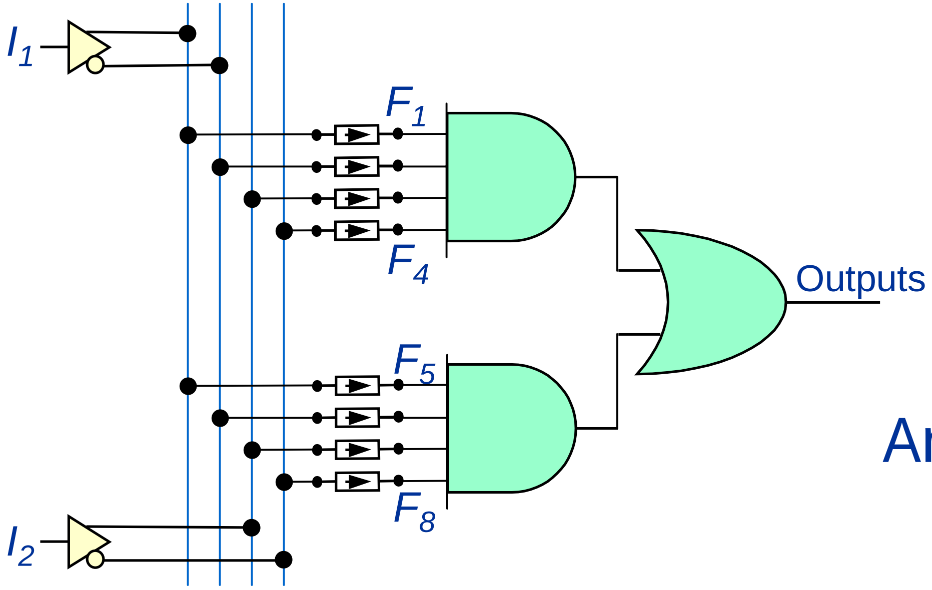
Programmable Array Logic (PALs)

– How beneficial is product sharing?
  » Not enough to justify the extra AND array
– PALs ==> *fixed* OR array
  » Each AND gate is permanently connected to a certain OR gate.
– Example: PAL16L8



Programmable Array Logic (PAL)

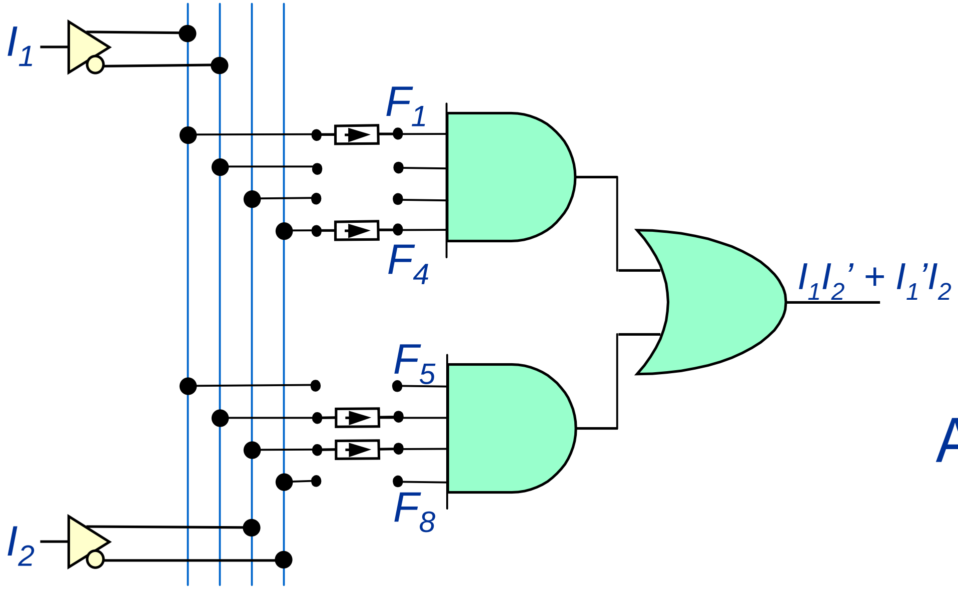
A PAL is a special case of a PLA in which the AND array is programmable but the OR array is fixed.



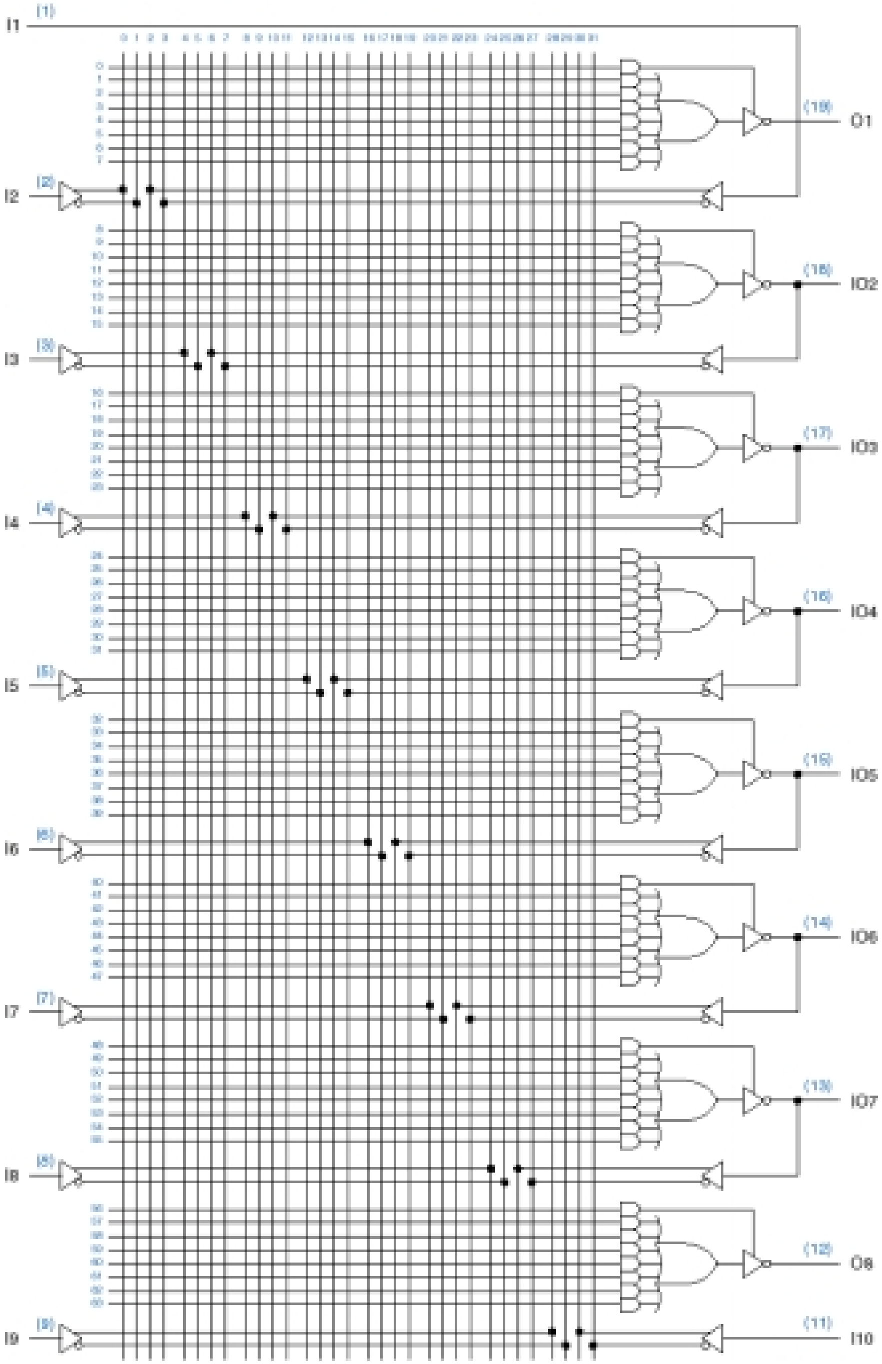
An unprogrammed PAL

Programmable Array Logic (PAL)

A PAL is a special case of a PLA in which the AND array is programmable but the OR array is fixed.



A programmed PAL



- 10 primary inputs
- 8 outputs, with 7 ANDs per output
- 1 AND for 3-state enable
- 6 outputs available as inputs
  - » more inputs, at expense of outputs
  - » two-pass logic, helper terms
- Note inversion on outputs
  - » output is complement of sum-of-products
- newer PALs have selectable inversion

Designing with PALs

- Compare number of inputs and outputs of the problem with available resources in the PAL.
- Write equations for each output using VHDL.
- Compile the VHDL program, determine whether minimized equations fit in the available AND terms.
- If they do not fit, try to modify the equations or to provide "helper" terms.

Some Questions

Is the criterion to minimize a set of functions to implement in a PAL the same that we used for the implementation with individual gates?

What is the problem formulation for the implementation of a set of logic functions in a PAL?

First Steps in VHDL

Lecture Goals

- Introduce VHDL Concept and Motivation for VHDL
- Introduce the VHDL Hierarchy and Alternative Architectures **Model**
- Start Defining VHDL Syntax

Motivation for VHDL

- Digital System **Complexity** Continues to Increase
  - No longer able to breadboard systems
    - » Number of chips
    - » Number of components
    - » Length of interconnects
  - Need to simulate before committing to hardware
    - » Not just logic, but timing

Motivation

- Different Types of Models are Required at Various Development Stages
  - Logic models
  - Performance models
  - Timing models
  - System Models

Motivation

- Non-Proprietary ***Lingua Franca***
  - Need a universal language for various levels of system design
  - Replacement for schematics
  - Unambiguous, formal language
  - Partitions problem
    - » Design
    - » Simulation and Verification
    - » Implementation

Motivation

- Standard for Development of Upgrades
  - Testbenches and results
  - System modifications must still pass original testbench
  - Testbench can (and should) be **written by people other than designers**

VHDL

Very High Speed Integrated Circuit (VHSIC)

Hardware

Description

Language

Need for VHDL

- Leads to Automatic Implementation--**Synthesis**
  - Routing tools
  - Standard cell libraries
  - FPGA
  - CPLD
  - Formal Language description is independent of physical implementation

Need for VHDL

- Need a ***Unified Development Environment***
  - Errors occur at translations from one stage of design to another
  - VHDL language the same at all levels
  - All people involved speak the same HDL
  - Testing and verification
- Performance, Reliability, and Behavioral Modeling Available at All Design Levels

Need for VHDL

- Need to Have Power and Flexibility to Model Digital Systems at Many Different Levels of Description
  - Support "mixed" simulation at different levels of abstraction, representation, and interpretation with an ability for step-wise refinement
  - Can model to high or low levels of detail, but still simulate

VHDL

- International IEEE Standard Specification Language (IEEE 1076-1993) for Describing Digital Hardware
- A Formal Language
  - Specification of designs
  - Simulation of performance
  - Interface to hardware detail design tools

Why VHDL?

- The Complexity and Size of Digital Systems leads to
  - Breadboards and prototypes which are too costly
  - Software and hardware interactions which are difficult to analyze without prototypes or simulations
  - Difficulty in **communicating** accurate design information

VHDL Model Components

- Complete VHDL Component Description Requires
  - **Entity**
    - » Defines a component's interface
  - **Architecture**
    - » Defines a component's function
- Several Alternative Architectures May Be Developed for Use With the Same Entity

Languages Other Than VHDL

- **VHDL:** **V**HSIC (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit) **H**ardware **D**escription **L**anguage
  - Not the only hardware description language

- Most others are proprietary

ABEL

- ABEL
  - Simplified HDL
  - PLD language
  - Dataflow primitives, *e.g.*, registers
  - Can use to Program XILINX FPGA

ALTERA

- ALTERA
  - Created by Altera Corporation
  - Simplified dialect of HDL
    - » AHDL

AHPL

- AHPL: A Hardware Programming Language
  - Dataflow language
  - Implicit clock
  - Does not support asynchronous circuits
  - Fixed data types
  - Non-hierarchical

CDL

- CDL: Computer Design Language
  - Academic language for teaching digital systems
  - Dataflow language
  - Non-hierarchical
  - Contains conditional statements

CONLAN

- CONLAN: CONsensus LANguage
  - Family of languages for describing various levels of abstraction
  - Concurrent
  - Hierarchical

IDL

- IDL: Interactive Design Language
  - Internal IBM language
  - Originally for automatic generation of PLA structures
  - Generalized to cover other circuits
  - Concurrent
  - Hierarchical

ISPS

- ISPS: Instruction Set Processor Specification
  - Behavioral language
  - Used to design software based on specific hardware
  - Statement level timing control, but no gate level control

TEGAS

- TEGAS: TEst Generation And Simulation
  - Structural with behavioral extensions
  - Hierarchical
  - Allows detailed timing specifications

TI-HDL

- TI-HDL: Texas Instruments Hardware Description Language
  - Created at Texas Instruments
  - Hierarchical
  - Models synchronous and asynchronous circuits
  - Non-extendable fixed data types

VERILOG

- Verilog
  - Essentially identical in function to VHDL
  - Simpler and syntactically different
  - Gateway Design Automation Co., 1983
  - Early ***de facto*** standard for ASIC programming
  - Open Verilog International standard
  - Programming language interface to allow connection to non-Verilog code - **PLI**

ZEUS

- ZEUS
  - Created at GTE
  - Hierarchical
  - Functional Descriptions
  - Structural Descriptions
  - Clock timing, but no gate delays
  - No asynchronous circuits

Different Representation Models

- Some, Not Mutually Exclusive, Models
  - Functional
  - Behavioral
  - Dataflow
  - Structural
  - Physical

Functional Model

- Describes the logical Function of Hardware
- Independent of Any Specific Implementation or Timing Information
  – Can exist at multiple levels of abstraction, depending on the granularity and the data types that are used in the behavioral description

Behavioral Model

- Describes the **Function and Timing** of Hardware Independent of Any Specific Implementation
  - Can exist at multiple levels of abstraction, depending on the granularity of the timing that are used in the functional description

Functional & Behavioral Descriptions

- Functional & Behavioral Models May Bear Little Resemblance to System Implementation
  - Structure not necessarily implied

Dataflow Model

- Describes How Data Moves Through the System and the Various Processing Steps
  - Register Transfer Level (RTL)
  - No registers are native to VHDL
  - Hides details of underlying combinational circuitry and functional implementation

Structural Model

- Represents a System in Terms of the **Interconnections of a Set of Components**
  - Components are interconnected in a hierarchical manner
  - Components themselves are described structurally, behaviorally, or functionally
    - » with interfaces between structural and their behavioral-level implementations

Structural Descriptions

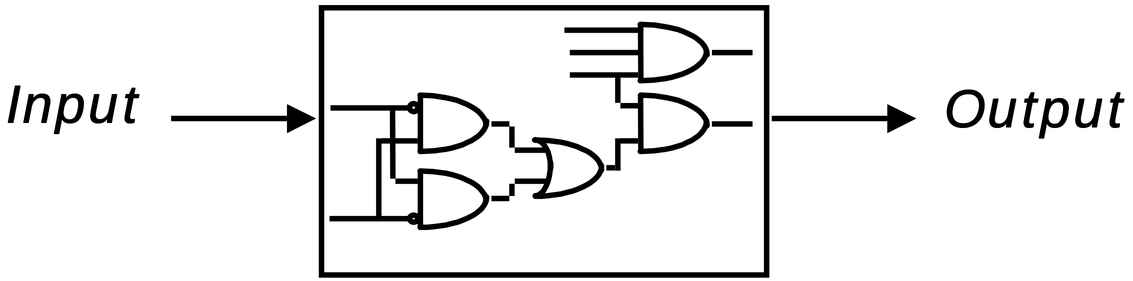
- Pre-Defined VHDL Components Are 'Instantiated' and Connected Together

- Structural Descriptions May Connect Simple Gates or Complex, Abstract Components

Structural Descriptions

- Mechanisms for Supporting Hierarchical Description

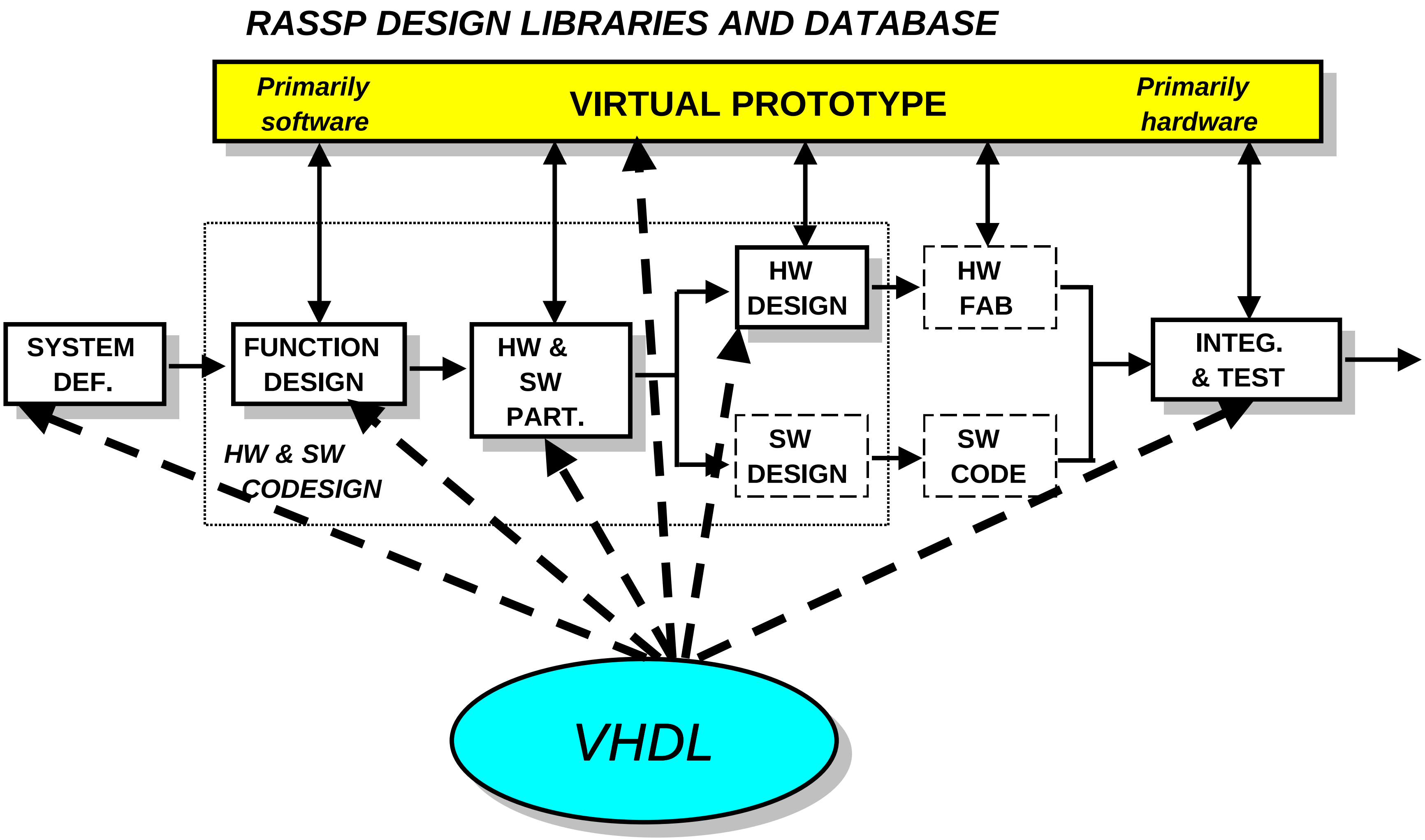
- Mechanisms for Describing Highly Repetitive Structures Easily



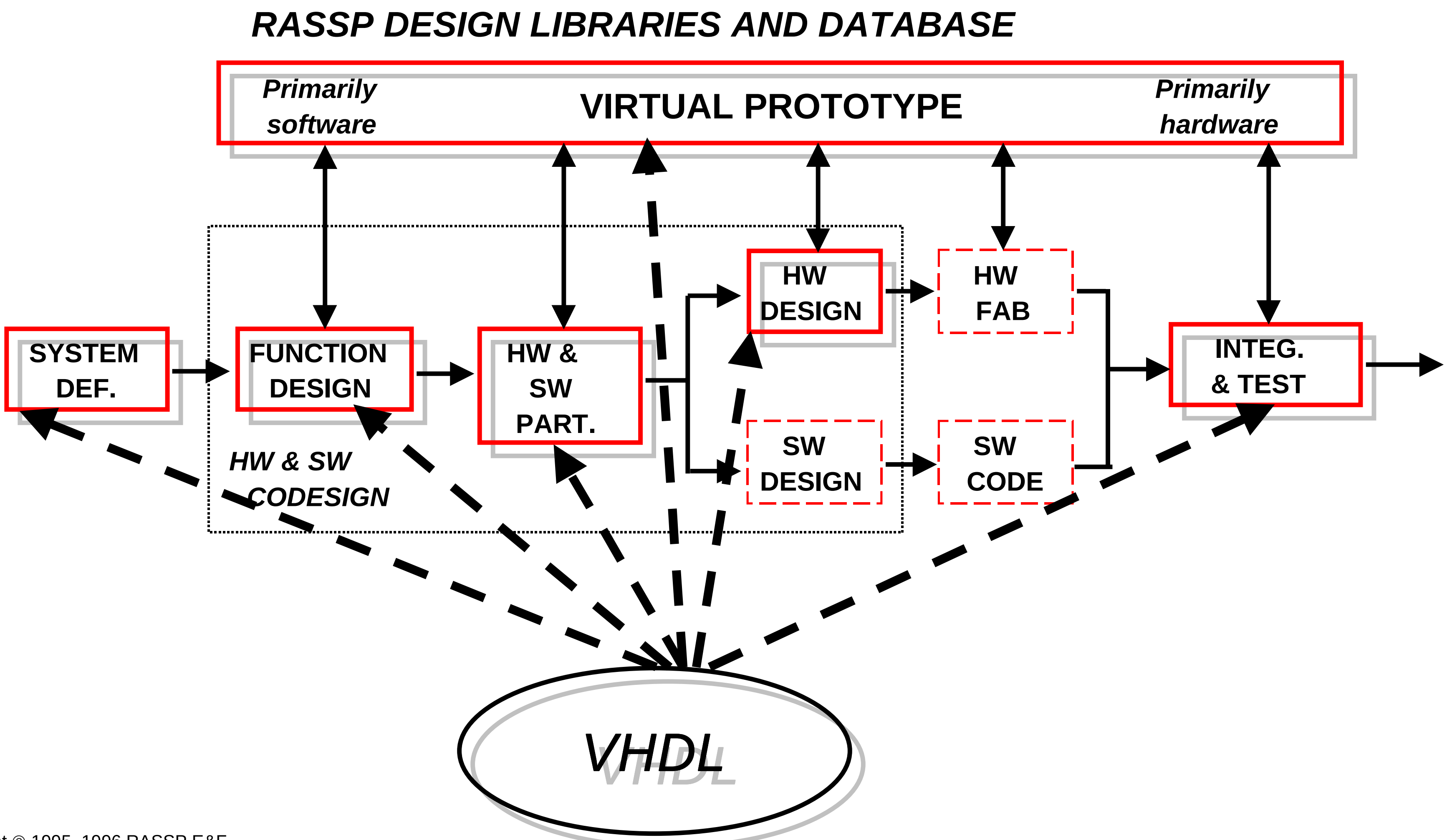
Physical Model

- Specifies the Relationship Between the Component Model and the Physical Packaging of the Component.
  - Contains all the timing and performance details to allow for an accurate simulation of physical reality
  - Back annotation allows precise simulations

RASSP Roadmap



RASSP Roadmap



Outline

- VHDL Background/History
- VHDL Design Example
- VHDL Model Components
  - Entity Declarations
  - Architecture Descriptions
- Basic Syntax and Lexicographical Conventions

Reasons for Using VHDL

- VHDL Is an International IEEE Standard Specification Language (IEEE 1076-1993) for Describing Digital Hardware Used by Industry Worldwide
  - VHDL is an acronym for **V**HSIC (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit) **H**ardware **D**escription **L**anguage

Reasons for Using VHDL

- VHDL enables hardware modeling from the gate to system level
- VHDL provides a mechanism for digital design and reusable design documentation
- VHDL Provides a Common Communications Medium

A Brief History of VHDL

- Very High Speed Integrated Circuit (VHSIC) Program
  - Launched in 1980
  - Object was to achieve significant gains in VLSI technology by shortening the time from concept to implementation (18 months to 6 months)
  - Need for common descriptive language

A Brief History of VHDL

- Woods Hole Workshop
  - Held in June 1981 in Massachusetts
  - Discussion of VHSIC goals
  - Comprised of members of industry, government, and academia

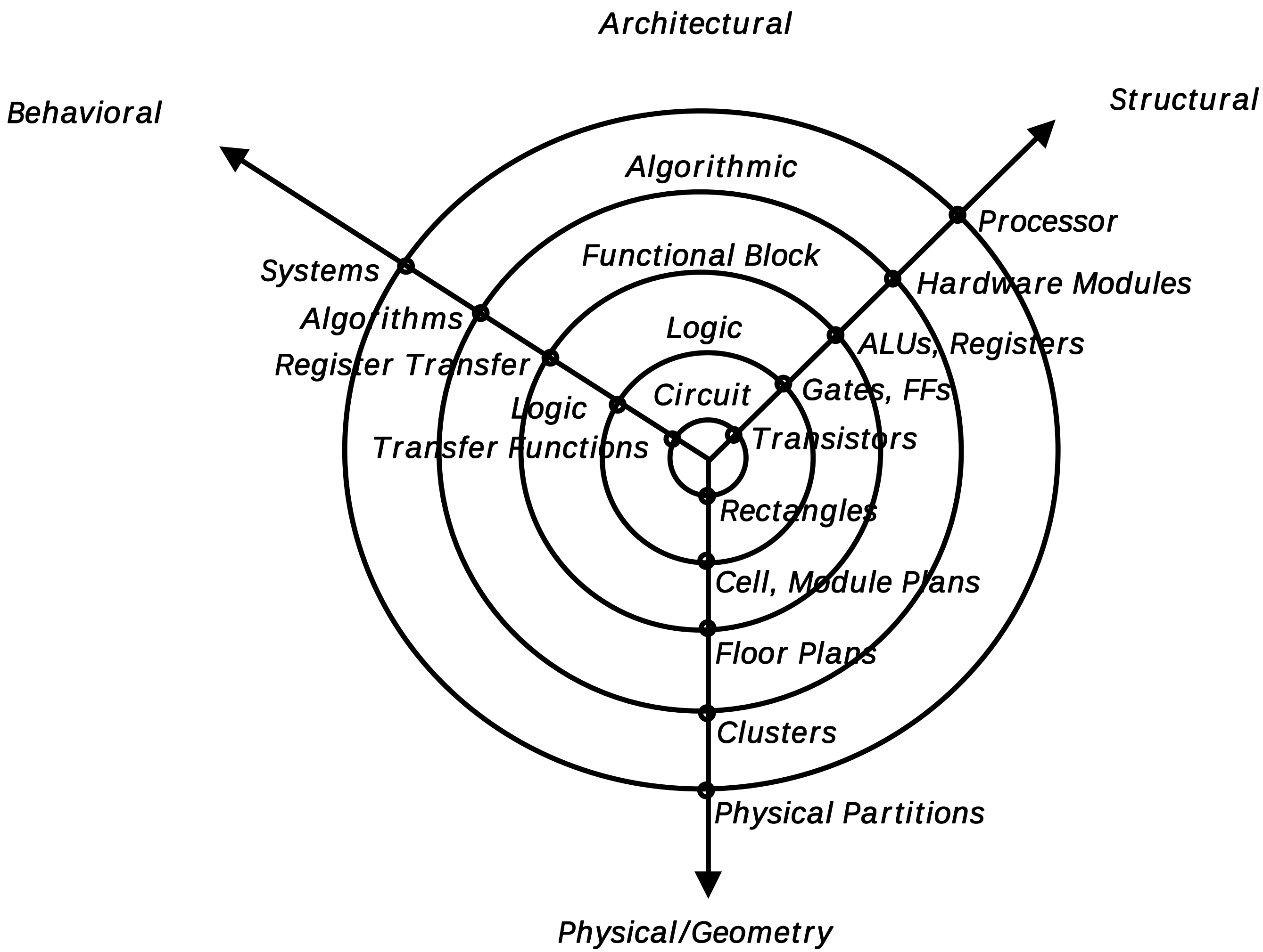
A Brief History of VHDL

- July 1983: contract awarded to develop VHDL
  - Intermetrics
  - IBM
  - Texas Instruments
- August 1985: VHDL Version 7.2 released

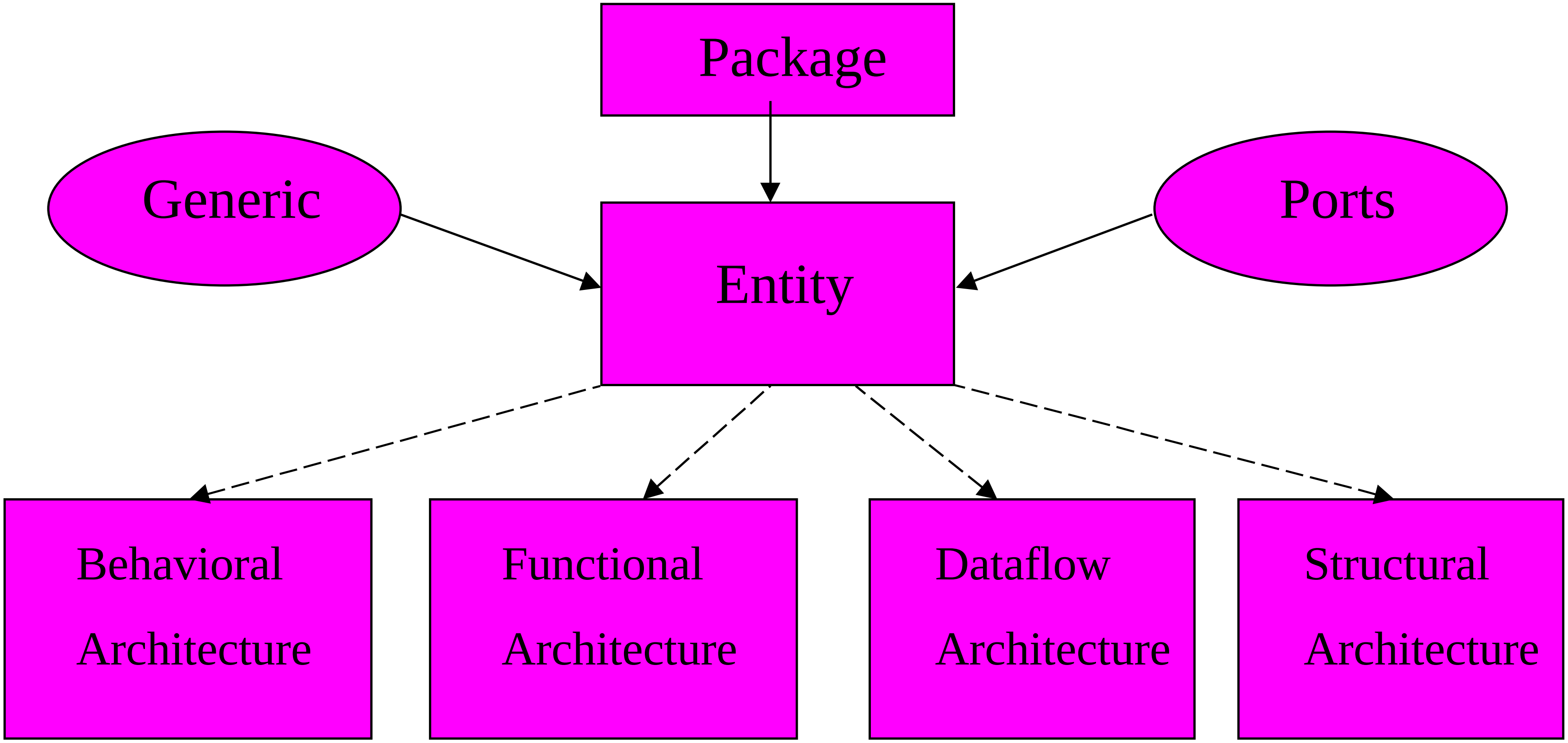
A Brief History of VHDL

- December 1987: VHDL became IEEE Standard 1076-1987 and in 1988 an ANSI standard
- September 1993: VHDL was restandardized to clarify and enhance the language
- VHDL has been accepted as a Draft International Standard by the IEC

Gajski and Kuhn's Y Chart



VHDL Model



Package
Generic
Entity
Ports
Behavioral Architecture
Functional Architecture
Dataflow Architecture
Structural Architecture

VHDL Design Example

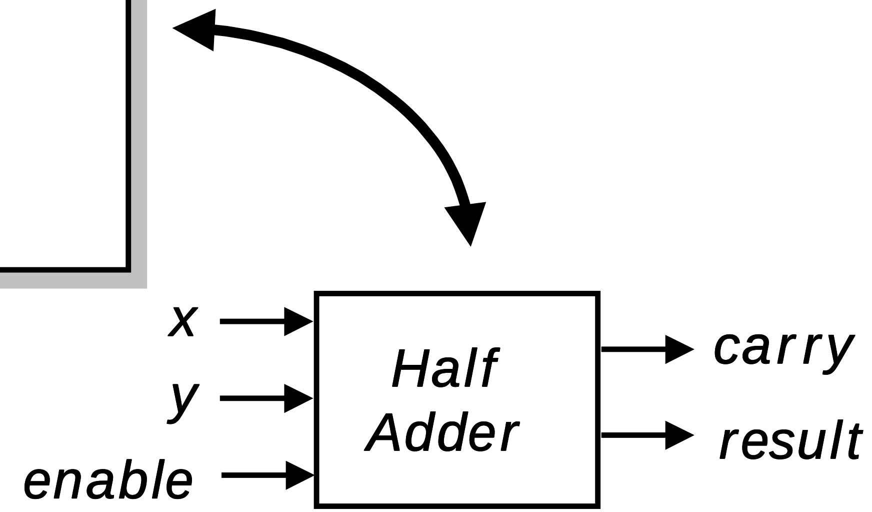
- Problem: Design a single bit half adder with carry and enable
- Specifications
  - Inputs and outputs are each one bit
  - When enable is high, result gets *x* plus *y*
  - When enable is high, carry gets any carry of *x* plus *y*
  - Outputs are zero when enable input is low

VHDL Design Example

Entity Declaration

- As a first step, the entity declaration describes the interface of the component
  - input and output *ports* are declared

```
ENTITY half_adder IS

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

END half_adder;
```



VHDL Design Example

Functional Specification

- A high level description can be used to describe the function of the adder

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

- The model can then be simulated to verify correct functionality of the component

VHDL Design Example

Behavioral Specification

- A high level description can be used to describe the function of the adder

```
ARCHITECTURE half_adder_b OF half_adder IS
  BEGIN
   PROCESS (x, y, enable)
   BEGIN
    IF enable = ‘1’ THEN
     result <= x XOR y after 10ns;
     carry  <= x AND y  after 12 ns;
    ELSE
     carry  <= ‘0’  after 10ns;
     result <= ‘0’ after 12ns;
    END IF;
          END PROCESS;
     END half_adder_b;
```

timing

- The model can then be simulated to verify correct timing of the entity

VHDL Design Example

Data Flow Specification

- A Third Method Is to Use Logic Equations to Develop a Data Flow Description

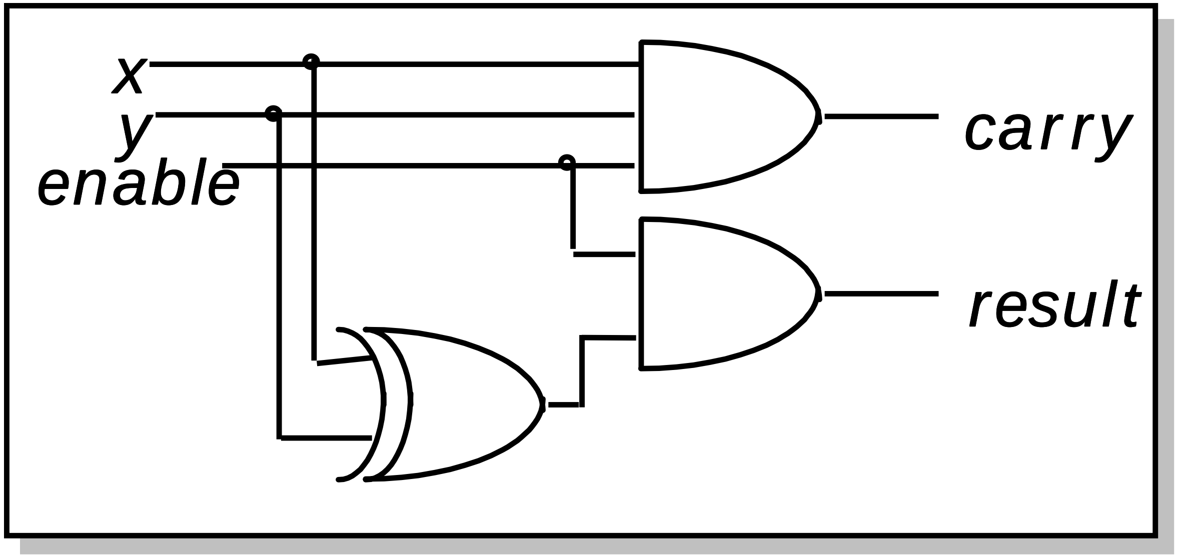
```
ARCHITECTURE half_adder_c OF half_adder IS
     BEGIN
       carry <= enable AND (x AND y);
       result <= enable AND (x XOR y);
     END half_adder_c;
```

- **Again, the model can be simulated at this level to confirm the logic equations**

VHDL Design Example

Structural Specification

- As a Fourth Method, a Structural Description Can Be Created From Previously Described Components
- These gates can be taken from a library of parts



VHDL Design Example

Structural Specification (Cont.)

```
ARCHITECTURE half_adder_d OF half_adder IS

    COMPONENT and2
      PORT (in0, in1 : IN BIT;
            out0 : OUT BIT);
    END COMPONENT;

    COMPONENT and3
      PORT (in0, in1, in2 : IN BIT;
            out0 : OUT BIT);
    END COMPONENT;

    COMPONENT xor2
      PORT (in0, in1 : IN BIT;
            out0 : OUT BIT);
    END COMPONENT;

    FOR ALL : and2 USE ENTITY gate_lib.and2_Nty(and2_a);
    FOR ALL : and3 USE ENTITY gate_lib.and3_Nty(and3_a);
    FOR ALL : xor2 USE ENTITY gate_lib.xor2_Nty(xor2_a);

-- description is continued on next slide
```

VHDL Design Example

Structural Specification (Cont.)

```
-- continuing half_adder_d description

    SIGNAL xor_res : BIT; -- internal signal
    -- Note that other signals are already declared in entity

    BEGIN

      A0 : and2 PORT MAP (enable, xor_res, result);
      A1 : and3 PORT MAP (x, y, enable, carry);
      X0 : xor2 PORT MAP (x, y, xor_res);

    END half_adder_d;
```

VHDL Model Components

- A Complete VHDL Component Description Requires a VHDL *Entity* and a VHDL *Architecture*
  - The *entity* defines a component's interface
  - The *architecture* defines a component's function
- **Several Alternative Architectures** May Be Developed for Use With the Same Entity

VHDL Model Components

- Three Areas of Description for a VHDL Component:
  - Structural descriptions
  - Functional descriptions
  - Timing and delay descriptions (Behavioral)

Process

- Fundamental Unit for Component Behavior Description Is the *Process*
  - Processes may be explicitly or implicitly defined
  - They are packaged in architectures

VHDL Model Components

- Primary Communication Mechanism Is the ***Signal*** (**distinct from a *variable***)
  - Process executions result in new values being assigned to signals which are then accessible to other processes
  - Similarly, a signal may be accessed by a process in another architecture by connecting the signal to ports in the the entities associated with the two architectures

```
Output <= My_id + 10;
```

Note symbol used for signals

VHDL Entity

- The Primary Purpose of an Entity Is to Declare the Input and Output Signals Which Communicate With It.
  - Interface signals are listed in the **PORT** clause which has 3 parts
    - » Name
    - » Mode
    - » Data type

VHDL Entity Example

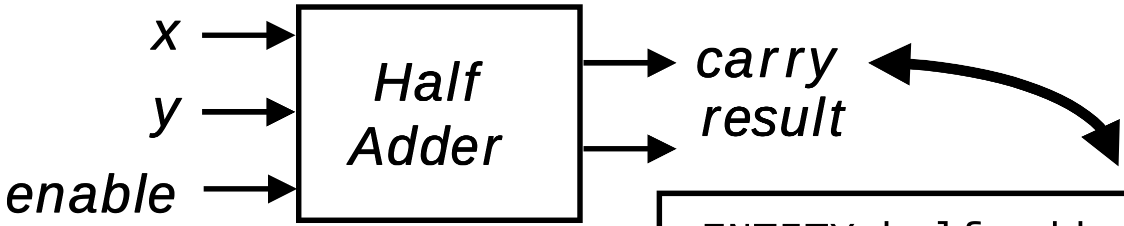
```
entity OR3 is

  port   ( A, B, C  : in    bit;
               D        : out  bit );

end OR3;
```

Entity Declarations

- The Primary Purpose of the Entity Is to Declare the Signals in the Component's Interface
  - The interface signals are listed in the PORT clause
    - In this respect, the *entity* is akin to the schematic *symbol* for the component

Entity versus Schematic Symbol

Entity Example




```
ENTITY half_adder IS

    GENERIC(prop_delay : TIME := 10 ns);

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

END half_adder;
```

Entity Declarations

Port Clause

- PORT clause declares the interface signals of the object to the outside world

- Three parts of the PORT clause
  - Name
  - Mode
  - Data type

```
PORT (signal_name : mode data_type);
```

  - Note port signals (i.e. ‘ports’) of the same mode and type or subtype may be declared on the same line

name mode Data type

```
PORT ( input : IN BIT_VECTOR(3 DOWNTO 0);
       ready, output : OUT BIT );
```

Entity Declarations

Port Clause (Cont.)

- The Port Mode of the Interface Describes the Direction in Which Data Travels With Respect to the *Component*
- **Five Port Modes**

1. **In**: data comes in this port and can only be read

2. **Out**: data travels out this port

Entity Declarations

Port Clause (Cont.)

3. **Buffer**: bidirectional data, but *only one signal driver may be enabled* at any one time

4. **Inout**: bidirectional data with any number of active drivers allowed but requires a Bus Resolution Function

5. **Linkage**: direction of data is unknown

Entity Declarations

Generic Clause

- Generics May Be Used for Readability, Maintenance and Configuration
- Generic Clause Syntax :

```
GENERIC (generic_name : type [:= default_value]);
```

  - If optional `default_value` missing in generic clause declaration, it must be present when component is to be used (*i.e.* instantiated)

Behavioral Descriptions

- VHDL Provides Two Styles of Describing Component Behavior
  - **Data Flow**: concurrent signal assignment statements
  - **Behavioral**: *processes* used to describe complex behavior by means of high-level language constructs
    - » variables, loops, if-then-else statements, *etc*.

Generic Clause

- Generic Clause Example :

```
GENERIC (My_ID : INTEGER := 37);
```

  - The generic *My_ID*, with a default value of 37, can be referenced by any architecture of the entity with this generic clause
  - The default can be overridden at component instantiation

Architecture Bodies

- Describes the Operation of the Component, Not Just Its Interface
- More Than One Architecture Can (and Usually Is) Associated With Each Entity

Architecture Bodies

- Consist of Two Parts:
  1. Declarative part -- includes necessary declarations, *e.g.* :
     - » type declarations
     - » signal declarations
     - » component declarations
     - » subprogram declarations

Architecture Bodies

2. Statement part -- includes statements that describe organization and/or functional operation of component, *e.g.* :
   - » concurrent signal assignment statements
   - » process statements
   - » component instantiation statements

Architecture Body, e.g.

```
ARCHITECTURE half_adder_d OF half_adder
IS
-- architecture declarative part
 SIGNAL xor_res : BIT ;
-- architecture statement part
BEGIN
 carry   <= enable AND (x AND y) ;
 result  <= enable AND xor_res ;
 xor_res <= x XOR y ;
END half_adder_d ;
```

Lexical Elements of VHDL

- Comments
  - two dashes to end of line is a comment, *e.g.*,

```
--this is a comment
```

Lexical Elements of VHDL

- Basic Identifiers
  - Can Only Use
    - » alphabetic letters (A-Z, a-z), or
    - » Decimal digits (0-9), or
    - » Underline character (_)
  - Must Start With Alphabetic Letter (`MyVal`)

Lexical Elements of VHDL

- Basic Identifiers
  - **Not case sensitive**

    (`LastValue = = lAsTvALue`)
  - May NOT end with underline (`MyVal_`)
  - May NOT contain sequential underlines (`My__Val`)

Lexical Elements of VHDL

- Extended Identifiers
  - Any character(s) enclosed by \ \
  - Case IS significant
  - Extended identifiers are distinct from basic identifiers
  - If " \ " is needed in extended identifier, use
    " \\ "

Lexical Elements of VHDL

- Reserved Words
  - Do not use as identifiers
- Special Symbols
  - Single characters

```
& ' ( ) * + , - . / : ; < = > |
```

  - Double characters (no intervening space)

```
=>   **   :=   /=   >=   <=   <>
```

Lexical Elements of VHDL

- Numbers
  - Underlines are NOT significant

    (10#8_192)

  - Exponential notation allowed

    (46e5 , 98.6E+12)

  - Integer Literals (12)
    - » Only positive numbers; negative numbers are preceded by unary negation operator
    - » No radix point

Lexical Elements of VHDL

- Real Literals (`23.1`)
  - » Always include decimal point
  - » Radix point must be preceded and followed by at least one digit.
- Radix (radix # number expressed in radix)
  - » Any radix from binary (`2`) to hexadecimal (`16`)
  - » Numbers in radices > 10 use letters `a-f` for 10-15.

Lexical Elements of VHDL

- String
  - A sequence of any printable characters enclosed in **double quotes**

```
("a string" )
```

  - Quote uses double quote

```
(" he said ""no!"" ")
```

  - Strings longer than one line use the concatenation operator (&) at beginning of continuation line.

Lexical Elements of VHDL

- Characters
  - Any printable character including space enclosed in single quotes (`‘x‘`)
- Bit Strings
  - B for binary (`b”0100_1001”`)
  - O for Octal (`o”76443”`)
  - X for hexadecimal (`x”FFFE_F138”`)

VHDL Syntax

- Extended Backus-Naur Form (EBNF)
  - Language divided into syntactic categories
  - Each category has a rule describing how to build a rule of that category
  - Syntactic category <= pattern
  - “<=“ is read as “...is defined to be...”

VHDL Syntax

– ***e.g.,***

variable_assignment `<=` *`target :=`*
`expression;`

– A clause of the category *variable_assignment* is defined to be a clause from the category *target* followed by the symbol " `:=` " followed by a clause from the *expression* category followed by a terminating " `;` "

VHDL Syntax

– syntax between outline brackets [ ] is optional

– syntax between outline braces { } can be repeated none or more times, *a.k.a.* “Kleene Star”

VHDL Syntax

– A preceding lexical element can be repeated an arbitrary number of times if ellipses are present, *e.g.*,

case-statement <=

```
case expression is
 case_statement_alternative
 { . . . }
end case ;
```

repeated

VHDL Syntax

– If a delimiter is needed, it is included with the ellipses as

```
identifier_list <=
        identifier { , . . . }
```

VHDL Syntax

- "OR" operator, " | ", in a list of alternatives, *e.g.*,

mode <= **in** | **out** | **inout**

- When grouping is ambiguous, parenthesis are used, *e.g.*,

term <=

factor { (* | / | mod | rem) factor }

VHDL Syntax

- *e.g.* an identifier may be defined in EBNF as

identifier **<=**

```
letter { [ underline ] letter_or_digit }
```

You can start working on Homework One

- **For those who look for easy projects:**
  - 1. Big Decoder and timing optimization.
  - 2. Generalized register with any set of operations, your choice but not only trivial.
  - 3. Robot control state machine
  - 4. Counter of large capacity without spikes
  - 5. Your choice, must be approved by me.
- **For those who look for medium projects:**
  - 1. Sorter but different from those on my www page
  - 2. Any circuit that has a state machine control unit and a register-transfer data path, for instance, GCD, Fibonacci, etc.
- **For those who look for challenging projects:**
  - 1. Any component of CCM or DSP processor.
  - 2. ALU using reversible logic
  - 3. Counters using reversible logic
  - 4. Controlling state machines in reversible logic.
  - 5. Any other component of your future final project, must be approved by me.

Homework Tools

- Mentor Graphics QuickVHDL
  - Covered in ECE 271
  - Look to my WWW page and link to ECE 271.
- Other Mentor tools on Unix
- IEEE VHDL Tutorial and VHDL Language Standard On-line
- send email to **damtawek@ece.pdx.edu** if you still have no account.

Optional Homework Tool

- Cypress Semiconductor (Warp release 6.x)
  - PC-based, Windows 3.1 with win32s extension
  - ~$99 with textbook
  - Oriented towards Cypress PLD & FPGA devices
  - Partial VHDL simulator
  - It is good to have Skahill’s book
- Any other tool that you have and wish to use.

Additional Reading

- Sections 5.1, 5.2, 5.3, 5.4, 5.5 (Wakerly Textbook)
  - Note, this book has Xilinx tools in it.
  - You can do most of your project at home if you have a PC and this book.
- First 4 chapters from Wakerly as a review.
- First three chapters from Mano/Kime.

This is not mandatory

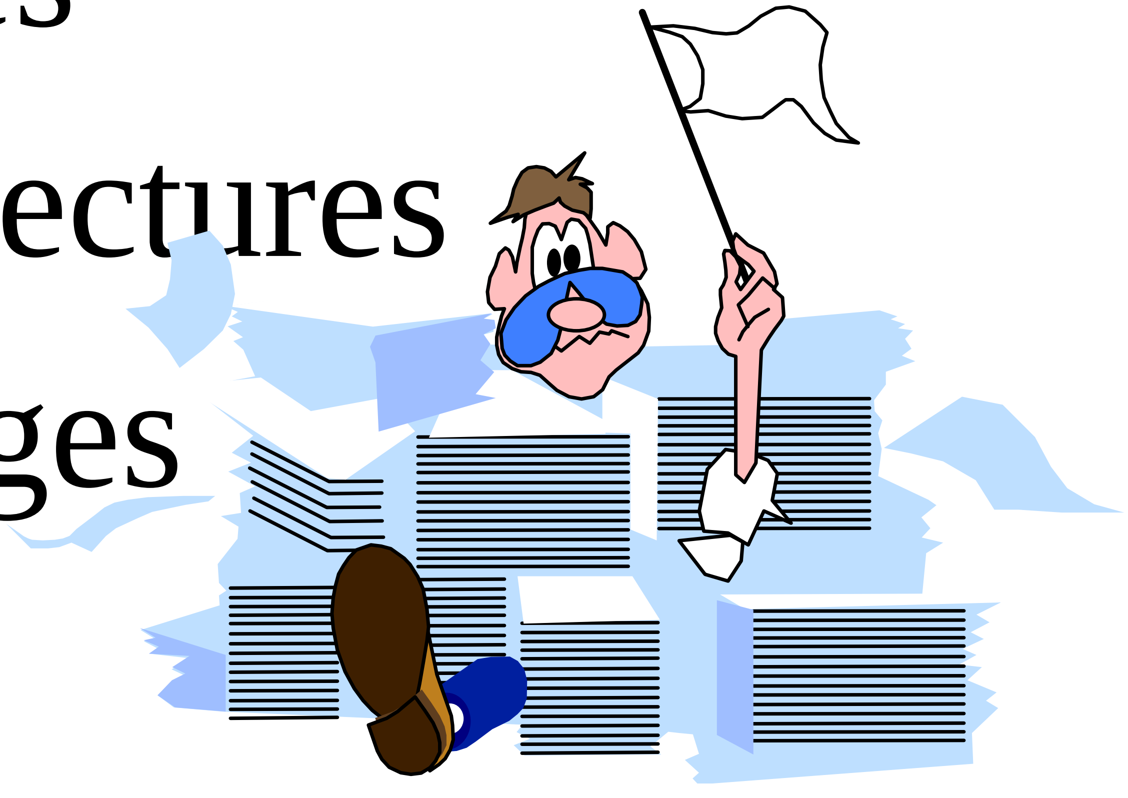
John F. Wakerly, Digital Design. Principles and Practices, Third Edition, Prentice Hall

Includes the XILINX Student Edition Foundation Series Software

Morris Mano and Charles Kime, Logic and Computer Design Fundamentals, 2nd edition. Includes the same software as Wakerly

Both these books were highly recommended by my students and professors from other universities

- Entities
- Architectures
- Packages



- VHDL-II Structural Modeling

Variables

- Exist Only Within an Architecture
  - Values of variables cannot be passed to other entities except through signals
- Variables Change Value When They Are Evaluated.
  - Signals change at a “later” time

Signals

- Entities are Interconnected by Signals
  - Process executions result in new values being assigned to signals which are then accessible to other processes
  - A signal <u>may be accessed</u> *by a process in another architecture* by connecting the signal to ports in the the entities associated with the two architectures

Signals

- Signals Can Be Declared **Internal to an Architecture** to Connect Internal Entities
- Variables Are Not Appropriate Since They Do Not Have the Temporal Characteristics of Hardware
- Signals Declared Within an Entity <u>Are Not Available to Other Entities Unless Specified in the Port Clause</u> of the Entity Declaration.

Entity Syntax

```
entity identifier is

  [ port ( port_interface_list ); ]

  { entity_declarative_item }

end [ entity ] [ identifier ] ;
```

Entity Syntax

```
port_interface_list <=
  ( identifier { , . . . } :
   [ mode ] subtype_indication
    [ := expression ] )
   { ; . . . }

   mode <=    in | out |  inout
```

Entity Example

```
entity NiCadCharger is

 port (
  Voltage, Current : in  real := 0.0 ;
  AC               : in  bit  := '1' ;
  Charged, Recharge: out bit        );

end entity NiCadCharger ;
```

mode

Architecture Syntax

```
architecture identifier of
 entity_name is

   { block_declarative_item }

begin
   { concurrent_statement }
end [architecture][ identifier ];
```

Structural Model

- A Representation of a System in Terms of the Interconnections of a Set of Defined Components.
  - Components can be described either <u>structurally or behaviorally</u>
  - Smallest components are behavioral entities
  - Components usually stored in <u>libraries</u>

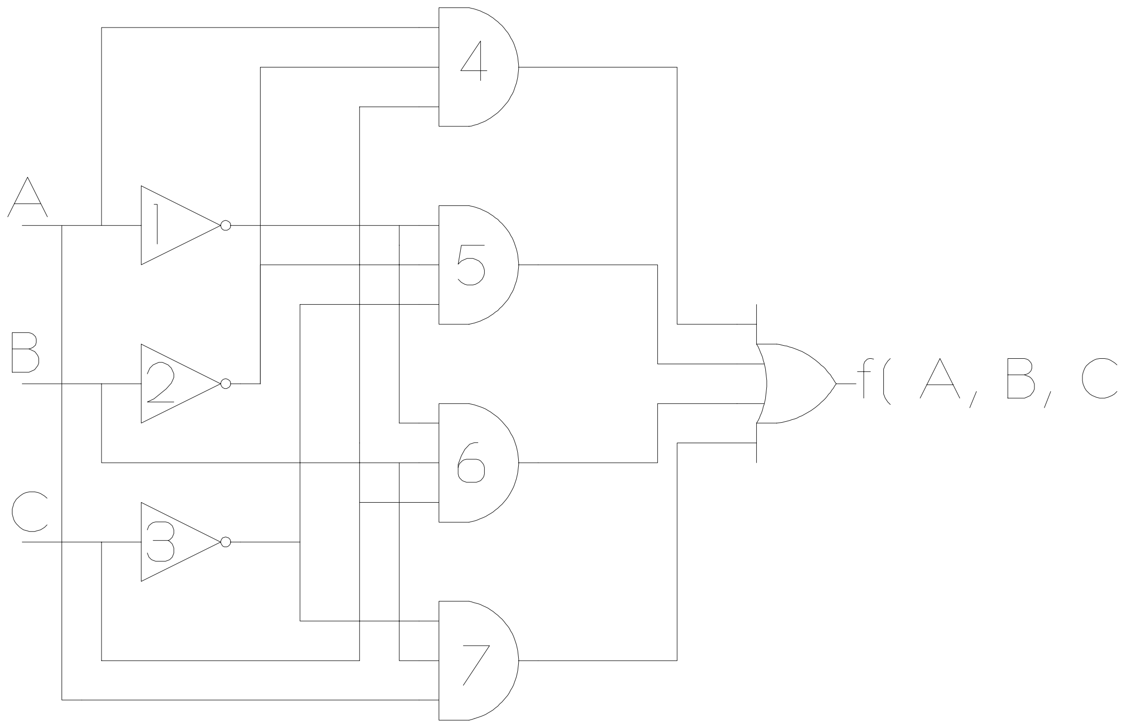
Structural Models

- Components Can Be Instantiated As Concurrent Statements in Architectures
  - If architecture not specified in statement
    - » Must be specified later, or
    - » Most recently analyzed architecture used
  - Ports can be specified two ways
    - » Positional association
    - » Named association

Structural Model
Internal Signals

- Entity Ports Which are Declared within an Architecture Body Are Local Signals
  - These signals are not available outside the architecture unless connected to one of the architecture's ports

Odd Parity Generator Example



Parity Entity

```
entity Odd_Parity is

 port(
  IN_1, IN_2, IN_3 : in bit ;
  Out_1            : out bit );
 end entity Odd_Parity ;
```

Odd Parity Behavior Architecture

```
architecture Odd_Parity_B of
  Odd_Parity is
```

$$f_{odd}(A,B,C) = A\bar{B}C + \bar{A}\bar{B}\bar{C} + \bar{A}BC + AB\bar{C}$$

```
begin
 Out_1 <= ( IN_1 and not IN_2 and IN_3 )
  OR ( not IN_1 and not IN_2 and not IN_3 )
  OR ( not IN_1 and IN_2 and IN_3 )
  OR ( IN_1 and IN_2 and not IN_3 )
end architecture Odd_Parity_B ;
```

INV Entity/Architecture

```
entity INV is
 port(
  In_1       : in  bit ;
  In_1_Bar : out bit );
 end entity INV ;
architecture INV_B of INV is
 begin
  In_1_Bar <=  not IN_1 ;
 end architecture INV_B ;
```

AND_3 Entity/Architecture

```
entity AND_3 is
 port(
  IN_1, IN_2, IN_3 : in bit ;
  Out_1            : out bit );
 end entity AND_3 ;
architecture AND_3_B of AND_3 is
 begin
  Out_1 <=  IN_1 and IN_2 and IN_3 ;
 end architecture AND_3_B ;
```

OR_4 Entity/Architecture

```
entity OR_4 is
 port(
  IN_1, IN_2, IN_3, IN_4 : in bit ;
  Out_1                  : out bit );
 end entity OR_4 ;
architecture OR_4_B of OR_4 is
 begin
  Out_1 <= IN_1 or IN_2 or IN_3 or IN_4 ;
 end architecture OR_4_B ;
```

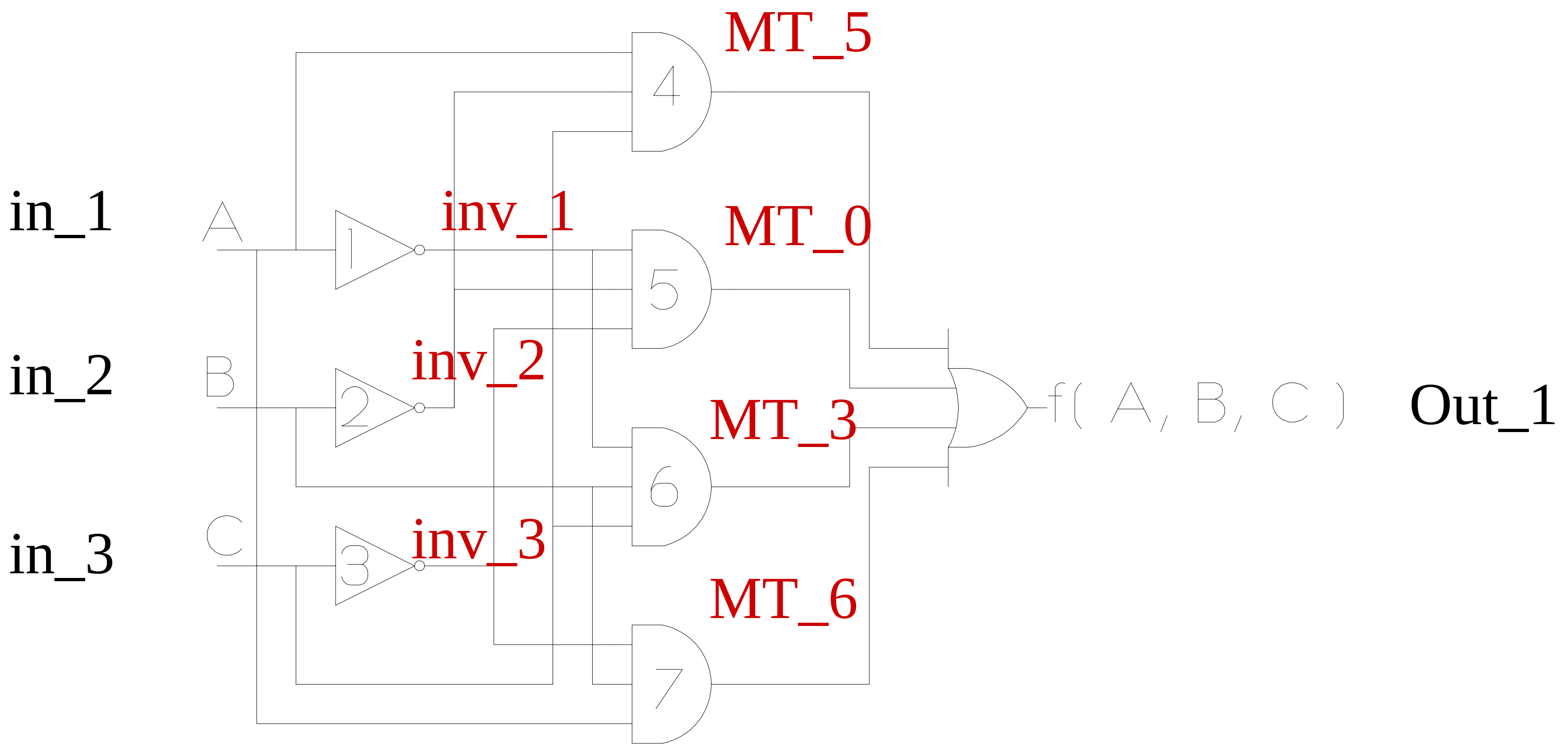
Odd Parity Structural Architecture

```
architecture Odd_Parity_S of
  Odd_Parity is
--block_declarative_items
--components
 component INV is
  port(
   In_1     : in  bit ;
   In_1_Bar : out bit );
 end component INV ;
```

Odd Parity Structural Architecture

```
component AND_3 is
  port( IN_1, IN_2, IN_3 : in bit ;
        Out_1            : out bit );
  end component AND_3 ;

 component OR_4 is
  port( IN_1, IN_2, IN_3, IN_4 : in bit ;
        Out_1                  : out bit );
  end component OR_4 ;
```

Structural Mapping



Odd Parity Structural Architecture

```
--block_declarative_items
--internal signals
 signal MT_0, MT_3, MT_5, MT_6 : bit ;
 signal INV_1, INV_2, INV_3     : bit ;
 begin --parity structural architecture
--connect gates
  G1: INV port map ( In_1, INV_1 );
  G2: INV port map ( In_2, INV_2 );
  G3: INV port map ( In_3, INV_3 );
```

Odd Parity Structural Architecture

```
G4: AND_3 port map
      ( IN_1, INV_2, IN_3, MT_5 );
G5: AND_3 port map
      ( INV_1, INV_2, INV_3, MT_0 );
G6: AND_3 port map
      ( INV_1, IN_2, IN_3, MT_3 );
G7: AND_3 port map
      ( IN_1, IN_2, INV_3, MT_6 );
```

Odd Parity Structural Architecture

```
G8: OR_4 port map
      ( MT_0, MT_3, MT_5, MT_6, Out_1 );
end architecture Odd_Parity_S ;
```

Packages

- Method for Grouping Related Declarations Which Serve a Common Purpose
  - Set of subprograms to operate on particular data type
  - Set of declarations for particular model
  - Allows declaration of "global" signals, *e.g.*, clocks.

Packages

- Design Unit Similar to Entity Declarations and Architecture Bodies
  - Can be put **in library** and made accessible to other units
  - Access to items declared in the package is through using its *Selected Name*
    - » library name . package name . item name
  - Aliases can be used to allow shorter names for accessing declared items

Packages

- Two Components to Packages
  - Package declaration
  - Package body
    - Not necessary if package declaration does not declare subprograms

Package Declaration

- Declares
  - Subprograms using header, implementation is hidden
  - Constants, do not need to be initialized in declaration
  - Types, must be completely specified
    - Can have variable size arrays
  - Signals must be completely specified

Package Declaration Syntax

```
package identifier is

  { package_declarative_item }

  end [ package ] [ identifier ] ;
```

Package Declaration Example

```
package dp32_types is

 constant unit_delay : Time := 1 ns;

 type bool_to_bit_table is array  (boolean) of
  bit;

 end dp32_types ;
```

Package Body

- Declared Subprograms Must Include the Full Declaration As Used in Package Declaration
  - Numeric literals can be written differently if same value
  - Simple name may be replaced by a selected name provided it refers to same item

Package Body

- May Contain Additional Declarations Which Are Local to the Package Body
  - Cannot declare signals in body

Package Body

```
package body identifier is

  { package_ body_declarative_item }

  end [ package body ] [ identifier ] ;
```

Sources

- Prof. K. J. Hintz, Department of Electrical and Computer Engineering, George Mason University
- Prof. John Wakerly, CISCO Systems and Stanford University.
- Dr. Jose Nelson Amaral, University of Alberta
- More information on ECE 271 class of Marek Perkowski.