Combinational Circuits

Design of Digital Circuits 2017
Srdjan Capkun
Onur Mutlu
(Guest starring: Frank K. Gürkaynak and Aanjhan Ranganathan)

http://www.syssec.ethz.ch/education/Digitaltechnik_17
What we will learn

- Tools to build digital circuits
  - Transistors
  - Gates
  - ... and why they matter

- Boolean Algebra

- Combinational Circuits

- Verilog
## Abstraction

<table>
<thead>
<tr>
<th>Abstraction Levels</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Software</strong></td>
<td>Programs</td>
</tr>
<tr>
<td><strong>Operating Systems</strong></td>
<td>Device drivers</td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>Instructions, Registers</td>
</tr>
<tr>
<td><strong>Micro architecture</strong></td>
<td>Datapath, Controllers</td>
</tr>
<tr>
<td><strong>Logic</strong></td>
<td>Adders, Memories</td>
</tr>
<tr>
<td><strong>Digital Circuits</strong></td>
<td>AND gates, NOT gates</td>
</tr>
<tr>
<td><strong>Analog Circuits</strong></td>
<td>Amplifiers</td>
</tr>
<tr>
<td><strong>Devices</strong></td>
<td>Transistors, Diodes</td>
</tr>
<tr>
<td><strong>Physics</strong></td>
<td>Electrons</td>
</tr>
</tbody>
</table>
Crash course in EE

- All the good parts of EE in short
  - Just the best parts, only highlights

- We will have exercises where we need to put this into use
  - We will use Verilog to program FPGAs...
  - ... and we will learn what these mean in a few hours

- What do we need to know to build computers
  - Understanding how they work will help you

- If you want to know more
  - Plenty of lectures/theses at EE (Prof. Benini, Gürkaynak)
(micro)-Processors
FPGAs
Custom ASICs
## But they all look the same....

<table>
<thead>
<tr>
<th></th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Microprocessor" /></td>
<td><img src="image2" alt="FPGA" /></td>
<td><img src="image3" alt="ASIC" /></td>
<td></td>
</tr>
<tr>
<td><strong>In short:</strong></td>
<td>Common building block of Computers</td>
<td>Reprogrammable Hardware, flexible</td>
<td>You customize everything</td>
</tr>
<tr>
<td>Development Time</td>
<td>minutes</td>
<td>day</td>
<td>months</td>
</tr>
<tr>
<td>Performance</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Good for</td>
<td>Simple to use Ubiquitous</td>
<td>Reprogramming, Small volume</td>
<td>Mass production, Max performance</td>
</tr>
<tr>
<td>Programming</td>
<td>Compiler</td>
<td>Bitfile</td>
<td>Design Masks</td>
</tr>
<tr>
<td>Languages</td>
<td>C/C++, Java, ...</td>
<td>Verilog/VHDL</td>
<td>Verilog/VHDL</td>
</tr>
<tr>
<td>Main Companies</td>
<td>Intel, ARM</td>
<td>Xilinx, Altera, ...</td>
<td>TSMC, UMC, ST, Globalfoundries</td>
</tr>
</tbody>
</table>
But they all look the same....

<table>
<thead>
<tr>
<th></th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICS</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
<td></td>
</tr>
<tr>
<td><strong>In short:</strong></td>
<td>Common building block of Computers</td>
<td>Reprogrammable Hardware, flexible</td>
<td>You customize everything</td>
</tr>
<tr>
<td><strong>Development Time</strong></td>
<td>minutes</td>
<td>days</td>
<td>months</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>—</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td><strong>Good for</strong></td>
<td>Simple to use, small volumes</td>
<td>Prototyping, Small volumes</td>
<td>Mass production, Max performance</td>
</tr>
<tr>
<td><strong>Programming</strong></td>
<td>Compiler</td>
<td>Bitfile</td>
<td>Design Masks</td>
</tr>
<tr>
<td><strong>Languages</strong></td>
<td>C/C++, Java/...</td>
<td>Verilog/VHDL</td>
<td>Verilog/VHDL</td>
</tr>
<tr>
<td><strong>Main Companies</strong></td>
<td>Intel, ARM</td>
<td>Xilinx, Altera, ...</td>
<td>TSMC, UMC, ST, Globalfoundries</td>
</tr>
</tbody>
</table>
But they all look the same....

<table>
<thead>
<tr>
<th>In short:</th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common building</td>
<td>Reprogrammable</td>
<td>You customize</td>
<td></td>
</tr>
<tr>
<td>block of</td>
<td>Hardware, flexible</td>
<td>everything</td>
<td></td>
</tr>
<tr>
<td>Computers</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Development Time</th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td>minutes</td>
<td></td>
<td>days</td>
<td>months</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance</th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td></td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Good for</th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple to use</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low power</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prototyping</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Small volume</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mass production</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High performance</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming</th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitfile</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design Masks</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Languages</th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++, Java, ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verilog, VHDL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSMC, UMC, ST,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Globalfoundries</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Companies</th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel, ARM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xilinx, Altera,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSMC, UMC, ST,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Globalfoundries</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
But they all look the same....

<table>
<thead>
<tr>
<th></th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image" alt="Microprocessor" /></td>
<td><img src="image" alt="FPGA" /></td>
<td><img src="image" alt="ASIC" /></td>
</tr>
<tr>
<td><strong>In short:</strong></td>
<td>Common building block of Computers</td>
<td>Reprogrammable Hardware, flexible</td>
<td>You customize everything</td>
</tr>
<tr>
<td><strong>Development Time</strong></td>
<td>minutes</td>
<td>days</td>
<td>months</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>o</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td><strong>Good for</strong></td>
<td>Simple to use Ubiquitous</td>
<td>Prototyping Small volume</td>
<td>Mass production, Max performance</td>
</tr>
<tr>
<td><strong>Programming</strong></td>
<td>Compiler</td>
<td>Bitfile</td>
<td>Design Masks</td>
</tr>
<tr>
<td><strong>Languages</strong></td>
<td>C/C++/Java/...</td>
<td>Verilog/VHDL</td>
<td>Verilog/VHDL</td>
</tr>
<tr>
<td><strong>Main Companies</strong></td>
<td>Intel, ARM</td>
<td>Xilinx, Altera, ...</td>
<td>TSMC, UMC, ST, Globalfoundries</td>
</tr>
</tbody>
</table>

- **Microprocessors** are common building blocks of computers.
- **FPGAs** are reprogrammable, hardware flexible that you customize everything.
- **ASICs** offer development time in minutes, performance +++, good for simple to use ubiqitous, prototyping small volume, mass production, max performance.

- **Main Companies** for Microprocessors: Intel, ARM.
- **Main Companies** for FPGAs: Xilinx, Altera.
- **Main Companies** for ASICs: TSMC, UMC, ST, Globalfoundries.
But they all look the same....

<table>
<thead>
<tr>
<th></th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
<td></td>
</tr>
<tr>
<td><strong>In short:</strong></td>
<td>Common building block of Computers</td>
<td>Reprogrammable Hardware, flexible</td>
<td>You customize everything</td>
</tr>
<tr>
<td><strong>Development Time</strong></td>
<td>minutes</td>
<td>days</td>
<td>months</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>o</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td><strong>Good for</strong></td>
<td>Simple to use Ubiquitous</td>
<td>Prototyping Small volume</td>
<td>Mass production, Max performance</td>
</tr>
<tr>
<td><strong>Programming</strong></td>
<td>Compiler</td>
<td>Bitfile</td>
<td>Design Masks</td>
</tr>
<tr>
<td><strong>Languages</strong></td>
<td>C/C++/Java/...</td>
<td>Verilog/VHDL</td>
<td>Verilog/VHDL</td>
</tr>
<tr>
<td><strong>Main Companies</strong></td>
<td>Intel, ARM</td>
<td>Xilinx, Altera, Lattice</td>
<td>TSMC, UMC, ST, Globalfoundries</td>
</tr>
</tbody>
</table>
But they all look the same....

<table>
<thead>
<tr>
<th>Want to learn how these work</th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>By programming these</th>
</tr>
</thead>
<tbody>
<tr>
<td>In short:</td>
<td>Common building block of Computers</td>
<td>Reprogrammable Hardware, flexible</td>
<td></td>
</tr>
<tr>
<td>Development Time</td>
<td>minutes</td>
<td>days</td>
<td>months</td>
</tr>
<tr>
<td>Performance</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Good for</td>
<td>Simple to use Ubiquitous</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming</td>
<td>Compiler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Languages</td>
<td>C/C++/Java/...</td>
<td>Verilog/VHDL</td>
<td>Verilog/VHDL</td>
</tr>
<tr>
<td>Main Companies</td>
<td>Intel, ARM</td>
<td>Xilinx, Altera, Lattice</td>
<td>TSMC, UMC, ST, Globalfoundries</td>
</tr>
</tbody>
</table>

Using this language

- Verilog/VHDL
- Design Masks
Building Blocks For Microchips

- **Conductors**
  - Metals: Aluminum, Copper

- **Insulators**
  - Glass (SiO$_2$), Air

- **Semiconductors**
  - Silicon (Si), Germanium (Ge)
Semiconductor
N-type Doping

N type region negatively charged (extra electrons)

one extra electron
P-type Doping

P type region positively charged (missing electrons)

one missing electron
What Is So Great About Semiconductors?

- You can “engineer” its properties
  - Make it **P type** by injecting type-III elements (B, Ga, In)
  - Make it **N type** by injecting elements from type-V (P,As)

- Starting with a pure semiconductor, you can combine P and N regions next to each other

- Allows you to make interesting electrical devices
  - Diodes
  - Transistors
  - Thrystors
The transistor

- By combining
  - Conductors (Metals)
  - Insulators (Oxide)
  - Semiconductors

- We get a Transistor (MOS)
  - Basically it is a switch
  - Apply (positive) voltage to GATE
  - Current flows from DRAIN to SOURCE

- Why is this cool?
  - We can combine many of these to realize simple logic gates
nMOS + pMOS = CMOS

Gate voltage at logic-0 level
nMOS + pMOS = CMOS

When One Type MOS works, the other is the load
The general form used to construct any inverting logic gate, such as: NOT, NAND, or NOR.

- The networks may consist of transistors in series or in parallel.
- When transistors are in parallel, the network is **ON** if either transistor is **ON**.
- When transistors are in series, the network is **ON** only if all transistors are **ON**.
CMOS Gates: NOT Gate

\[ Y = \overline{A} \]

<table>
<thead>
<tr>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\begin{array}{|c|c|c|c|c|}
\hline
A & P1 & N1 & Y \\
\hline
0 & & & & \\
1 & & & & \\
\hline
\end{array}
\]
CMOS Gates: NOT Gate

\[ Y = \overline{A} \]

<table>
<thead>
<tr>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>P1</th>
<th>N1</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ON</td>
<td>OFF</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>OFF</td>
<td>ON</td>
<td>0</td>
</tr>
</tbody>
</table>
CMOS Gates: NAND Gate

NAND

\[ Y = \overline{AB} \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A | B | P1 | P2 | N1 | N2 | Y
---|---|----|----|----|----|---
0 | 0 |    |    |    |    | 0 |
0 | 1 |    |    |    |    | 1 |
1 | 0 |    |    |    |    | 1 |
1 | 1 |    |    |    |    | 0 |
CMOS Gates: NAND Gate

NAND

\[ Y = \overline{AB} \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

P1 and P2 are in parallel; only one must be ON to pull the output up to VDD.

N1 and N2 are connected in series; both must be ON to pull the output to GND.
Common Logic Gates

**Buffer**

<table>
<thead>
<tr>
<th>A</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**AND**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**OR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**XOR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Inverter**

<table>
<thead>
<tr>
<th>A</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**NAND**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**NOR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**XNOR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Moore’s Law

Number of Transistors of Intel Processors

Year of Introduction

Number of Transistors

Source: www.intel.com/go/museum
Moore’s Law

- Coined by Gordon Moore:
  - In 1965
  - Co-founder of Intel

Number of transistors that can be manufactured doubles roughly every 18 months.

- In other words: it increases 100 fold over 10 years.
How Do We Keep Moore’s Law

- Manufacturing smaller structures
  - Some structures are already a few atoms in size

- Developing materials with better properties
  - Copper instead of Aluminum (better conductor)
  - Hafnium Oxide, air for Insulators
  - Making sure all materials are compatible is the challenge

- Optimizing the manufacturing steps
  - How to use 193nm ultraviolet light to pattern 20nm structures

- New technologies
  - FinFET, Gate All Around transistor, Single Electron Transistor...
We can now build logic circuits

A logic circuit is composed of:

- Inputs
- Outputs

*Functional specification* (describes relationship between inputs and outputs)

*Timing specification* (describes the delay between inputs changing and outputs responding)
Circuits

- **Circuit elements**
  - E1, E2, E3
  - Each itself a circuit

- **Nodes (wires)**
  - Inputs: A, B, C
  - Outputs: Y, Z
  - Internal: n1
  - To count the nodes look at
    - outputs of every circuit element
    - inputs to the entire circuit
Types of Logic Circuits

- **Combinational Logic**
  - Memoryless
  - Outputs determined by current values of inputs
  - In some books called Combinatorial Logic

- **Sequential Logic**
  - Has memory
  - Outputs determined by previous and current values of inputs
Rules of Combinational Composition

- Every circuit element is itself combinational

- Every node of the circuit is either
  - designated as an input to the circuit or
  - connects to exactly one output terminal of a circuit element

- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once

- Example: (If E1-3 combinational)
Boolean Equations

- Functional specification of outputs in terms of inputs

- Example (full adder – more later):

\[
S = F(A, B, C_{in})
\]

\[
C_{out} = G(A, B, C_{in})
\]

\[
S = A \oplus B \oplus C_{in}
\]

\[
C_{out} = AB + AC_{in} + BC_{in}
\]
Boolean Algebra

- Set of axioms and theorems to simplify Boolean equations

- Like regular algebra, but in some cases simpler because variables can have only two values (1 or 0)

- Axioms and theorems obey the principles of duality:
  - stay correct if
    ANDs and ORs interchanged and
    0’s and 1’s interchanged
  - Examples:

\[
\overline{0} = 1 \\
B \cdot \overline{B} = 0
\]
Boolean Algebra

- Set of axioms and theorems to simplify Boolean equations
- Like regular algebra, but in some cases simpler because variables can have only two values (1 or 0)
- Axioms and theorems obey the principles of duality:
  - stay correct if ANDs and ORs interchanged and 0’s and 1’s interchanged
  - Examples:
    - \( \overline{0} = 1 \)
    - \( B \cdot \overline{B} = 0 \)
    - \( \overline{1} = 0 \)
    - \( B + \overline{B} = 1 \)
## Boolean Axioms

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Dual</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>$B = 0$ if $B \neq 1$</td>
<td>A1'</td>
</tr>
<tr>
<td>A2</td>
<td>(\overline{0} = 1)</td>
<td>A2'</td>
</tr>
<tr>
<td>A3</td>
<td>$0 \cdot 0 = 0$</td>
<td>A3'</td>
</tr>
<tr>
<td>A4</td>
<td>$1 \cdot 1 = 1$</td>
<td>A4'</td>
</tr>
<tr>
<td>A5</td>
<td>$0 \cdot 1 = 1 \cdot 0 = 0$</td>
<td>A5'</td>
</tr>
</tbody>
</table>

**Duality:** If the symbols 0 and 1 and the operators \(\cdot\) (AND) and \(+\) (OR) are interchanged, the statement will still be correct.
### Boolean Theorems: Summary

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Dual</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T1'</td>
<td>Identity</td>
</tr>
<tr>
<td>$B \cdot 1 = B$</td>
<td>$B + 0 = B$</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>T2'</td>
<td>Null Element</td>
</tr>
<tr>
<td>$B \cdot 0 = 0$</td>
<td>$B + 1 = 1$</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>T3'</td>
<td>Idempotency</td>
</tr>
<tr>
<td>$B \cdot B = B$</td>
<td>$B + B = B$</td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>$\overline{\overline{B}} = B$</td>
<td>Involution</td>
</tr>
<tr>
<td>T5</td>
<td>T5'</td>
<td>Complements</td>
</tr>
<tr>
<td>$B \cdot \overline{B} = 0$</td>
<td>$B + \overline{B} = 1$</td>
<td></td>
</tr>
</tbody>
</table>
## Boolean Theorems of Several Variables

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Equation</th>
<th>Dual</th>
<th>Equation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>T6</td>
<td>$B \cdot C = C \cdot B$</td>
<td>T6’</td>
<td>$B + C = C + B$</td>
<td>Commutativity</td>
</tr>
<tr>
<td>T7</td>
<td>$(B \cdot C) \cdot D = B \cdot (C \cdot D)$</td>
<td>T7’</td>
<td>$(B + C) + D = B + (C + D)$</td>
<td>Associativity</td>
</tr>
<tr>
<td>T8</td>
<td>$(B \cdot C) + (B \cdot D) = B \cdot (C + D)$</td>
<td>T8’</td>
<td>$(B + C) \cdot (B + D) = B + (C \cdot D)$</td>
<td>Distributivity</td>
</tr>
<tr>
<td>T9</td>
<td>$B \cdot (B + C) = B$</td>
<td>T9’</td>
<td>$B + (B \cdot C) = B$</td>
<td>Covering</td>
</tr>
<tr>
<td>T10</td>
<td>$(B \cdot C) + (B \cdot \overline{C}) = B$</td>
<td>T10’</td>
<td>$(B + C) \cdot (B + \overline{C}) = B$</td>
<td>Combining</td>
</tr>
<tr>
<td>T11</td>
<td>$(B \cdot C) + (\overline{B} \cdot D) + (C \cdot D)$</td>
<td>T11’</td>
<td>$(B + C) \cdot (\overline{B} + D) \cdot (C + D)$</td>
<td>Consensus</td>
</tr>
<tr>
<td></td>
<td>$= B \cdot C + \overline{B} \cdot D$</td>
<td></td>
<td>$= (B + C) \cdot (\overline{B} + D)$</td>
<td></td>
</tr>
<tr>
<td>T12</td>
<td>$B_0 \cdot B_1 \cdot B_2 \ldots$</td>
<td>T12’</td>
<td>$B_0 + B_1 + B_2 \ldots$</td>
<td>De Morgan’s Theorem</td>
</tr>
<tr>
<td></td>
<td>$= (B_0 + B_1 + B_2 \ldots)$</td>
<td></td>
<td>$= (B_0 \cdot B_1 \cdot B_2)$</td>
<td></td>
</tr>
</tbody>
</table>
Simplifying Boolean Expressions: Example 2

\[ Y = A(AB + ABC) \]
Simplifying Boolean Expressions: Example 2

\[ Y = A(AB + ABC) \]

\[ = A(AB(1 + C)) \quad T8 \]

\[ = A(AB(1)) \quad T2' \]

\[ = A(AB) \quad T1 \]

\[ = (AA)B \quad T7 \]

\[ = AB \quad T3 \]
### Boolean Equations Example

- You are going to the cafeteria for lunch
  - You won’t eat lunch ($\overline{E}$)
  - If it is not open ($\overline{O}$) or
  - If they only serve cabbage ($C$)

- Write a truth table for determining if you will eat lunch ($E$)

<table>
<thead>
<tr>
<th>$O$</th>
<th>$C$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Some Definitions

- **Complement:** variable with a bar over it
  \( \overline{A}, \overline{B}, \overline{C} \)

- **Literal:** variable or its complement
  \( A, \overline{A}, B, \overline{B}, C, \overline{C} \)

- **Implicant:** product (AND) of literals
  \( (A \cdot B \cdot \overline{C}), (\overline{A} \cdot C), (B \cdot \overline{C}) \)

- **Minterm:** product (AND) that includes all input variables
  \( (A \cdot B \cdot \overline{C}), (\overline{A} \cdot \overline{B} \cdot C), (\overline{A} \cdot B \cdot \overline{C}) \)

- **Maxterm:** sum (OR) that includes all input variables
  \( (A + \overline{B} + \overline{C}), (\overline{A} + B + \overline{C}), (A + B + \overline{C}) \)
### Sum-of-Products (SOP) Form

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- All Boolean equations can be written in SOP form
  - Each row in a truth table has a minterm
  - A minterm is a product (AND) of literals
  - Each minterm is TRUE for that row (and only that row)
  - *blackboard*

- Formed by **ORing** the **minterms** for which the output is **TRUE**

$Y = F(A, B)$ = ?
### Sum-of-Products (SOP) Form

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>minterm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( \overline{A} \overline{B} )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( \overline{A} B )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( A \overline{B} )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( A B )</td>
</tr>
</tbody>
</table>

\[ Y = F(A, B) = (\overline{A} \cdot B) \]

- All Boolean equations can be written in SOP form
  - Each row in a truth table has a minterm
  - A minterm is a product (AND) of literals
  - Each minterm is TRUE for that row (and only that row)
  - blackboard

- Formed by ORing the minterms for which the output is TRUE
From Logic to Gates

- SOP (sum-of-products) leads to two-level logic

- Example: \[ Y = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot C) \]
Combinational Building Blocks

- Combinational logic is often grouped into larger building blocks to build more complex systems
- Hide the unnecessary gate-level details to emphasize the function of the building block
- We have already studied some building blocks
  - full adders
  - priority circuits
- We now look at:
  - multiplexers
  - decoders
Multiplexer (Mux)

- Selects between one of $N$ inputs to connect to the output.
- Needs $\log_2 N$-bit control input

2:1 Mux Example:

<table>
<thead>
<tr>
<th>$S$</th>
<th>$D_1$</th>
<th>$D_0$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiplexer (Mux)

- Selects between one of \( N \) inputs to connect to the output.
- Needs \( \log_2 N \)-bit control input
- 2:1 Mux Example:
Hardware Description Languages

- In the beginning HDLs were developed as a ‘standard way’ of drawing circuit schematics.

- Modeled the interface of circuits, described how they were connected

- Allowed connections between these modules

- Supported some common logic functions
  - AND OR NOT XOR
  - Multiplexers
Convenient Way of Drawing Schematics
Convenient Way of Drawing Schematics

- It is standard
  - Everybody will interpret the schematic the same way

- It is not proprietary
  - HDLs are not tool specific

- It is machine readable
  - It is easier for computers to understand the circuit

- Only later on additional benefits were discovered
  - Simulation and Synthesis
Two Hardware Description Languages

■ Verilog
  ▪ developed in 1984 by Gateway Design Automation
  ▪ became an IEEE standard (1364) in 1995
  ▪ More popular in US

■ VHDL (VHSIC Hardware Description Language)
  ▪ Developed in 1981 by the Department of Defense
  ▪ Became an IEEE standard (1076) in 1987
  ▪ More popular in Europe

■ In this course we will use Verilog
Defining a module

- A module is the main building block in Verilog

- We first need to declare:
  - Name of the module
  - Types of its connections (input, output)
  - Names of its connections
Defining a module

module example (a, b, c, y);
  input a;
  input b;
  input c;
  output y;

// here comes the circuit description
endmodule
A question of style

The following two codes are identical

```vhdl
module test ( a, b, y );
  input a;
  input b;
  output y;
endmodule
```

```vhdl
module test ( input a,
  input b,
  output y );
endmodule
```
What if we have busses?

- You can also define multi-bit busses.
  - [ range_start : range_end ]

- Example:

```vhdl
input [31:0] a;  // a[31], a[30] .. a[0]
output [15:8] b1;  // b1[15], b1[14] .. b1[8]
output [7:0] b2;  // b2[7], b2[6] .. b1[0]
input  clk;  // single signal
```
Basic Syntax

- Verilog is case sensitive:
  - SomeName and somename are not the same!

- Names cannot start with numbers:
  - 2good is not a valid name

- Whitespace is ignored

```
// Single line comments start with a //

/* Multiline comments
   are defined like this */
```
Good Practices

- Develop/use a consistent naming style

- Use MSB to LSB ordering for busses (little-endian)
  - Try using “a[31:0]” and not “a[0:31]”

- Define one module per file
  - Makes managing your design hierarchy easier

- Use a file name that equals module name
  - i.e. module TryThis is defined in a file called TryThis.v
There are Two Main Styles of HDL

- **Structural**
  - Describe how modules are interconnected
  - Each module contains other modules (instances)
  - ... and interconnections between these modules
  - Describes a hierarchy

- **Behavioral**
  - The module body contains functional description of the circuit
  - Contains logical and mathematical operators

- **Practical circuits would use a combination of both**
Structural HDL: Instantiating a Module
Structural HDL Example

Module Definitions

module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;
endmodule

module small (A, B, Y);
    input A;
    input B;
    output Y;
    // description of small
endmodule
Module Definitions

```verilog
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;
endmodule

module small (A, B, Y);
    input A;
    input B;
    output Y;

    // description of small
endmodule
```
Structural HDL Example

Wire definitions

```verilog
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;
endmodule

module small (A, B, Y);
    input A;
    input B;
    output Y;

// description of small
endmodule
```
Structural HDL Example

**Instantiate first module**

```verilog
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;
    // instantiate small once
    small i_first ( .A(A),
                    .B(SEL),
                    .Y(n1) );
endmodule
```

```verilog
module small (A, B, Y);
    input A;
    input B;
    output Y;
    // description of small
endmodule
```

![Diagram showing the instantiation of the small module](image-url)
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;

  // instantiate small once
  small i_first ( .A(A),
                  .B(SEL),
                  .Y(n1)   );

  // instantiate small second time
  small i2 ( .A(n1),
             .B(C),
             .Y(Y) );

endmodule

module small (A, B, Y);
  input A;
  input B;
  output Y;

  // description of small
endmodule
**Short Instantiation**

```verilog
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;

    // alternative
    small i_first ( A, SEL, n1 );

    /* Shorter instantiation,
       pin order very important */

    // any pin order, safer choice
    small i2 ( .B(C),
               .Y(Y),
               .A(n1) );

endmodule
```

```verilog
module small (A, B, Y);
    input A;
    input B;
    output Y;

    // description of small

endmodule
```
What Happens with HDL code?

- **Automatic Synthesis**
  - Modern tools are able to map a behavioral HDL code into gate-level schematics
  - They can perform many optimizations
  - ... however they can not guarantee that a solution is optimal
  - Most common way of Digital Design these days

- **Simulation**
  - Allows the behavior of the circuit to be verified without actually manufacturing the circuit
  - Simulators can work on behavioral or gate-level schematics
Behavioral HDL: Defining Functionality

module example (a, b, c, y);
    input a;
    input b;
    input c;
    output y;

// here comes the circuit description
assign y = ~a & ~b & ~c |
    a & ~b & ~c |
    a & ~b & c;

endmodule
Behavioral HDL: Synthesis Results

\[ b \rightarrow \text{un5}_y \rightarrow y \]
\[ c \rightarrow \text{un8}_y \rightarrow y \]
\[ a \rightarrow \text{un5}_y \rightarrow y \]
Behavioral HDL: Simulating the Circuit
Bitwise Operators

module gates(input [3:0] a, b,  
            output [3:0] y1, y2, y3, y4, y5);

    /* Five different two-input logic
     gates acting on 4 bit busses */

    assign y1 = a & b;    // AND
    assign y2 = a | b;    // OR
    assign y3 = a ^ b;    // XOR
    assign y4 = ~(a & b); // NAND
    assign y5 = ~(a | b); // NOR

eendmodule
Bitwise Operators: Synthesis Results
module and8(input [7:0] a, 
            output y);

    assign y = &a;

    // &a is much easier to write than 

endmodule
Reduction Operators: assign y = &a;
Conditional Assignment

module mux2(input [3:0] d0, d1, input s, output [3:0] y);

assign y = s ? d1 : d0;
// if (s) then y=d1 else y=d0;
endmodule

? : is also called a ternary operator as it operates on three inputs:

- s
- d1
- d0.
Conditional Assignment: \( y = s \ ? \ d1: \ d0; \)
module mux4(input [3:0] d0, d1, d2, d3
  input [1:0] s,
  output [3:0] y);

  assign y = s[1] ? ( s[0] ? d3 : d2)
         : ( s[0] ? d1 : d0);

  // if (s1) then
  //    if (s0) then y=d3 else y=d2
  // else
  //    if (s0) then y=d1 else y=d0

endmodule
module mux4(input [3:0] d0, d1, d2, d3
    input [1:0] s,
    output [3:0] y);

    assign y = (s == 2’b11) ? d3 :
        (s == 2’b10) ? d2 :
            (s == 2’b01) ? d1 :
                d0;

    // if      (s = “11” ) then y= d3
    // else if (s = “10” ) then y= d2
    // else if (s = “01” ) then y= d1
    // else                     y= d0

endmodule
How to Express numbers?

**N’ Bxx**

8’ b0000_0001

- **(N) Number of bits**
  - Expresses how many bits will be used to store the value

- **(B) Base**
  - Can be b (binary), h (hexadecimal), d (decimal), o (octal)

- **(xx) Number**
  - The value expressed in base, apart from numbers it can also have X and Z as values.
  - Underscore _ can be used to improve readability
# Number Representation in Verilog

<table>
<thead>
<tr>
<th>Verilog</th>
<th>Stored Number</th>
<th>Verilog</th>
<th>Stored Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>4’ b1001</td>
<td>1001</td>
<td>4’ d5</td>
<td>0101</td>
</tr>
<tr>
<td>8’ b1001</td>
<td>0000 1001</td>
<td>12’ hFA3</td>
<td>1111 1010 0011</td>
</tr>
<tr>
<td>8’ b0000_1001</td>
<td>0000 1001</td>
<td>8’ o12</td>
<td>00 001 010</td>
</tr>
<tr>
<td>8’ bxX0X1zZ1</td>
<td>XX0X 1ZZ1</td>
<td>4’ h7</td>
<td>0111</td>
</tr>
<tr>
<td>‘b01</td>
<td>0000 .. 0001</td>
<td>12’ h0</td>
<td>0000 0000 0000</td>
</tr>
</tbody>
</table>
What have seen so far:

- Describing structural hierarchy with Verilog
  - Instantiate modules in an other module

- Writing simple logic equations
  - We can write AND, OR, XOR etc

- Multiplexer functionality
  - If ... then ... else

- We can describe constants

- But there is more:
## Precedence of operations in Verilog

<table>
<thead>
<tr>
<th>Highest</th>
<th>Lowest</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>NOT</td>
</tr>
<tr>
<td>*, /, %</td>
<td>mult, div, mod</td>
</tr>
<tr>
<td>+, -</td>
<td>add, sub</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td>shift</td>
</tr>
<tr>
<td>&lt;&lt;&lt;, &gt;&gt;&gt;</td>
<td>arithmetic shift</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>comparison</td>
</tr>
<tr>
<td>==, !=</td>
<td>equal, not equal</td>
</tr>
<tr>
<td>&amp;, ~&amp;</td>
<td>AND, NAND</td>
</tr>
<tr>
<td>^, ~^</td>
<td>XOR, XNOR</td>
</tr>
<tr>
<td></td>
<td>, ~</td>
</tr>
<tr>
<td>?:</td>
<td>ternary operator</td>
</tr>
</tbody>
</table>
Example: Comparing two numbers

An XNOR gate

```
module MyXnor (input a, b, output z);
    assign z = ~(a ^ b); //not XOR
endmodule
```

An AND gate

```
module MyAnd (input a, b, output z);
    assign z = a & b;   // AND
endmodule
```
Example: Comparing Two Numbers

```verilog
module compare (input a0, a1, a2, a3, b0, b1, b2, b3, output eq);
    wire c0, c1, c2, c3, c01, c23;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
    MyAnd haha (.A(c0), .B(c1), .Z(c01) ); // AND
    MyAnd hoho (.A(c2), .B(c3), .Z(c23) ); // AND
    MyAnd bubu (.A(c01), .B(c23), .Z(eq) ); // AND

endmodule
```
Example: Comparing Two Numbers

```verilog
module compare (input a0, a1, a2, a3, b0, b1, b2, b3, 
                output eq);
    wire c0, c1, c2, c3, c01, c23;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR

    assign c01 = c0 & c1;
    assign c23 = c2 & c3;
    assign eq = c01 & c23;

endmodule
```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3, output eq);
  wire c0, c1, c2, c3;

MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR

assign eq  = c0 & c1 & c2 & c3;

endmodule
Example: Comparing Two Numbers

```verilog
class compare (input a0, a1, a2, a3, b0, b1, b2, b3, output eq);
   wire [3:0] c; // bus definition

MyXnor i0 (.A(a0), .B(b0), .Z(c[0])); // XNOR
MyXnor i1 (.A(a1), .B(b1), .Z(c[1])); // XNOR
MyXnor i2 (.A(a2), .B(b2), .Z(c[2])); // XNOR
MyXnor i3 (.A(a3), .B(b3), .Z(c[3])); // XNOR

assign eq = &c; // short format

endmodule
```
Example: Comparing Two Numbers

```verilog
module compare (input [3:0] a, input [3:0] b, 
                   output eq);
    wire [3:0] c; // bus definition

    MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0])); // XNOR
    MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1])); // XNOR
    MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2])); // XNOR
    MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3])); // XNOR

    assign eq = &c; // short format

endmodule
```
Example: Comparing Two Numbers

```verilog
module compare (input [3:0] a, input [3:0] b, output eq);
    wire [3:0] c; // bus definition

assign c = ~(a ^ b); // XNOR

assign eq = &c; // short format

endmodule
```
Example: Comparing Two Numbers

```verilog
module compare (input [3:0] a, input [3:0] b,
    output eq);

assign eq = (a == b) ? 1 : 0;  // really short

endmodule
```
What is the BEST way of writing Verilog

- Quite simply IT DOES NOT EXIST!

- Code should be easy to understand
  - Sometimes longer code is easier to comprehend

- Hierarchy is very useful
  - In the previous example it did not look like that, but for larger designs it is indispensable

- Try to stay closer to hardware
  - After all the goal is to design hardware
Parameterized Modules

module mux2
    #(parameter width = 8) // name and default value
    (input [width-1:0] d0, d1,
        input s,
        output [width-1:0] y);

    assign y = s ? d1 : d0;
endmodule

- We can pass parameters to a module
Parameterized Modules: Instantiating

```verilog
class module mux2
  #(parameter width = 8) // name and default value
  (input [width-1:0] d0, d1,
   input s,
   output [width-1:0] y);

  assign y = s ? d1 : d0;
endmodule

// If parameter is not given, default is assumed (here 8)
mux2 i_mux (d0, d1, s, out);

// The same module with 12-bit bus width:
mux2 #(12) i_mux_b (d0, d1, s, out);

// More verbose version:
mux2 #(.width(12)) i_mux_b (.d0(d0), .d1(d1),
  .s(s), .out(out));
```
Manipulating Bits

// You can assign partial busses
wire [15:0] longbus;
wire [7:0] shortbus;
assign shortbus = longbus[12:5];

// Concatenating is by {}
assign y = {a[2],a[1],a[0],a[0]};

// Possible to define multiple copies
assign x = {a[0], a[0], a[0], a[0]};
assign y = { 4{a[0]} }
Z floating output

```verilog
module tristate(input [3:0] a,
                input en,
                output [3:0] y);

    assign y = en ? a : 4'bz;

endmodule
```
### Truth Table for AND with Z and X

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
It is possible to define timing relations in Verilog

- These are **ONLY** for Simulation
- They **CAN NOT** be synthesized
- They are used for modeling delays in simulation

```verilog
'timescale 1ns/1ps
module simple (input a, output z1, z2);

assign #5 z1 = ~a; // inverted output after 5ns
assign #9 z2 = a; // output after 9ns
endmodule
```
Next Steps

- We have seen an overview of Verilog
- Discussed behavioral and structural modeling
- Showed combinational logic constructs

Still to come: (later)

- Sequential circuit description in Verilog
- Developing testbenches for simulation