

Last Lecture

- The basic component of a digital circuit is the MOS transistor
- Transistor have intrinsic resistance and capacitance, so voltage values in the circuit take some time to change (“delay”)
- There exist two kinds: nMOS and pMOS, with complementary behavior and advantages/disadvantages
- A logic gate implementing a certain boolean function can be built with a circuit composed of:
 - A Pull-Down network of nMOS
 - A Pull-Up network of pMOS
- There exist automatic rules to determine the topology of the Pull-Down and the Pull-Up network for a gate
- Multiple gates can be connected together to form more complicated components
- In CSE140 we saw logic gates (like NOT and NAND)
- Today here we will look at slightly more high level components (because it’s what you have to implement in homework 2)



CSE140L: Components and Design Techniques for Digital Systems Lab

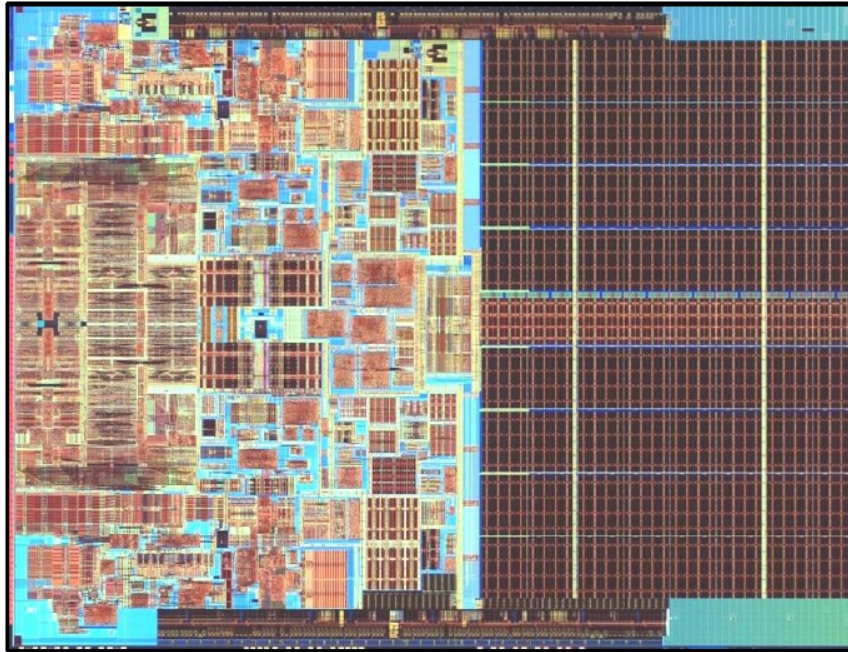
Some basic digital components

System on a Chip (SoC)

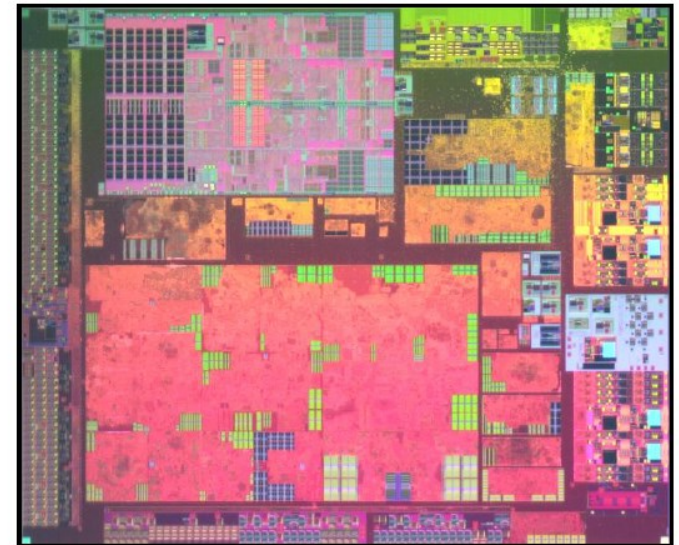
- Combines many complex and diverse functional blocks on a single silicon die
- Combined features include:
 - Graphics
 - CPU
 - Memory
 - Peripheral interface logic
 - External memory control
 - Signal Processing
 - Communication signal demodulation
- **Benefits**
 - Lower cost
 - Industry tools support rapid integration of IP
 - High speed communication between functional blocks

Higher Integration Levels

65 nm CPU
(2005)



22 nm SoC
(2013)



SoC products integrate a wider range of circuit and device types

* Source Intel

FPGAs

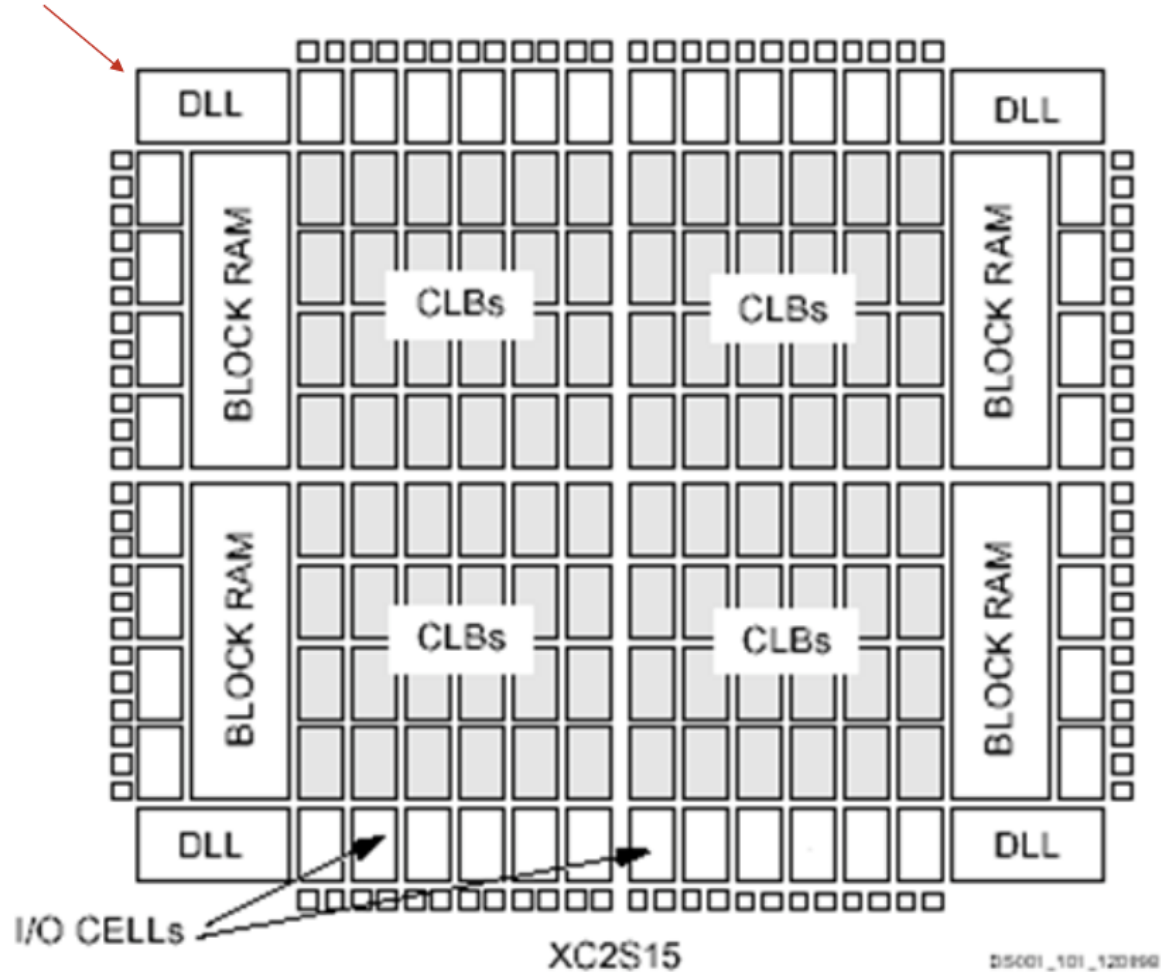


- Field Programmable Gate Array
- Reconfigurable
 - Can implement programmed logic
 - Logic can be described via an HDL (like Verilog) or schematics (not preferred)
- Composed of flexible logic – e.g. “Universal Logic Models”
- Implements the logic based on timing constraints
- Programmable interconnect



FPGAs – Programmable Logic Cells

Digital Delay Locked Loops



* Source Xilinx

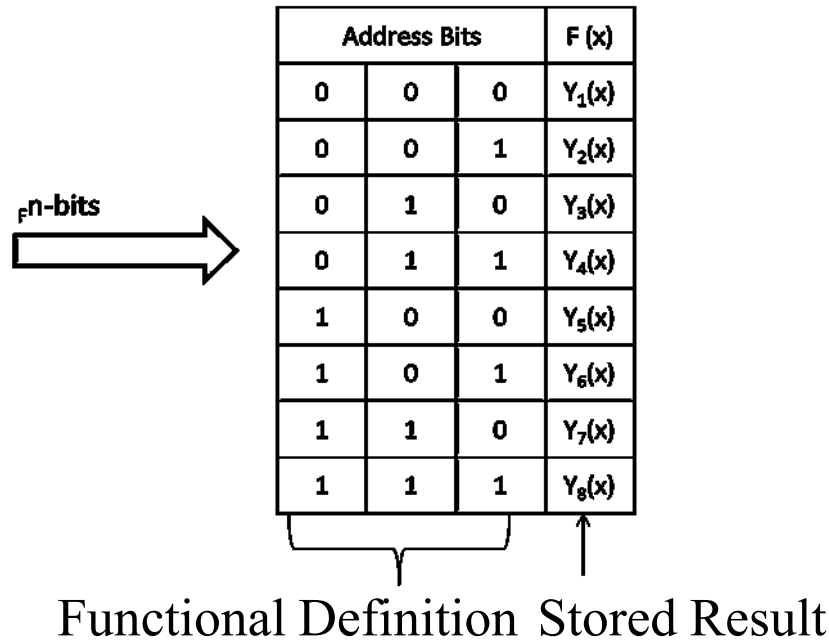
FPGAs - Benefits



- Rapid test and development of IP
 - Goal of high performance and density compared with other prototyping options
 - Time to Market for non cost sensitive solutions
 - Reusability
 - Easy integration of soft IP
 - Great educational tool
 - Over the generations the flexible logic primitives have been enhanced to perform well across critical applications
 - Signal and Image Processing
 - Video
 - Security
 - Communications
- 

FPGAs – Interesting Fact

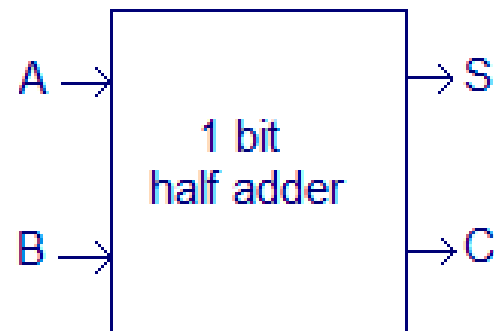
- Universal Logic Modules are designed by exploiting key features of boolean logic
- Combinational building blocks like decoders, multiplexers in addition to memory provide a vast array of flexibility



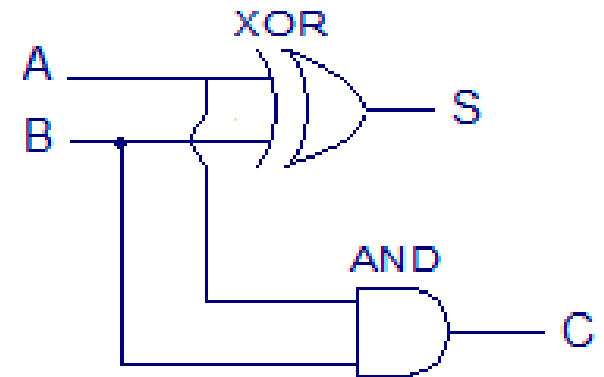
Half Adder

Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table



Schematic



Realization

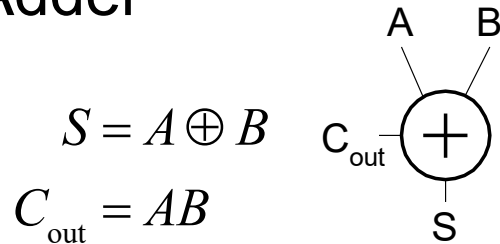
A,B : Inputs
S: Sum
C: Carry

Half Adder: Checkoff Example

- 1) Submit the zipped project folder containing ALL of your source code on TED
 - Tutor checks that the files are there, so the checkoff can start
- 2) Explain the functionality of module
 - TA report example: “knows the behavior of the half adder”
- 3) Review of Verilog code
 - TA report example: “Verilog code has missing port declaration”
- 4) Compilation of code
 - TA report example: “The project compiles correctly”
- 5) Running testbenches to verify functionality
 - TA report example: “The testbench demonstrates the functionality of the module by producing the correct output with all possible combinations of inputs”

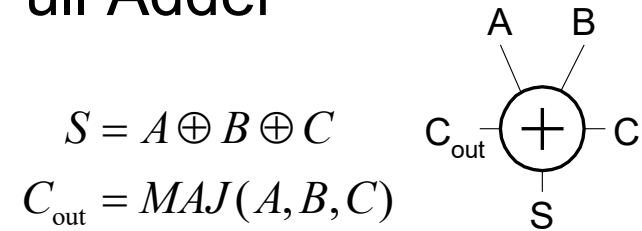
Single-Bit Addition

Half Adder



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full Adder



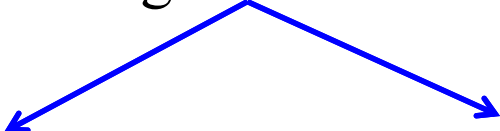
A	B	C	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Binary Addition Equations

$$\begin{array}{r} a_3a_2a_1a_0 \\ + b_3b_2b_1b_0 \\ \hline c_4 \ s_3s_2s_1s_0 \end{array}$$

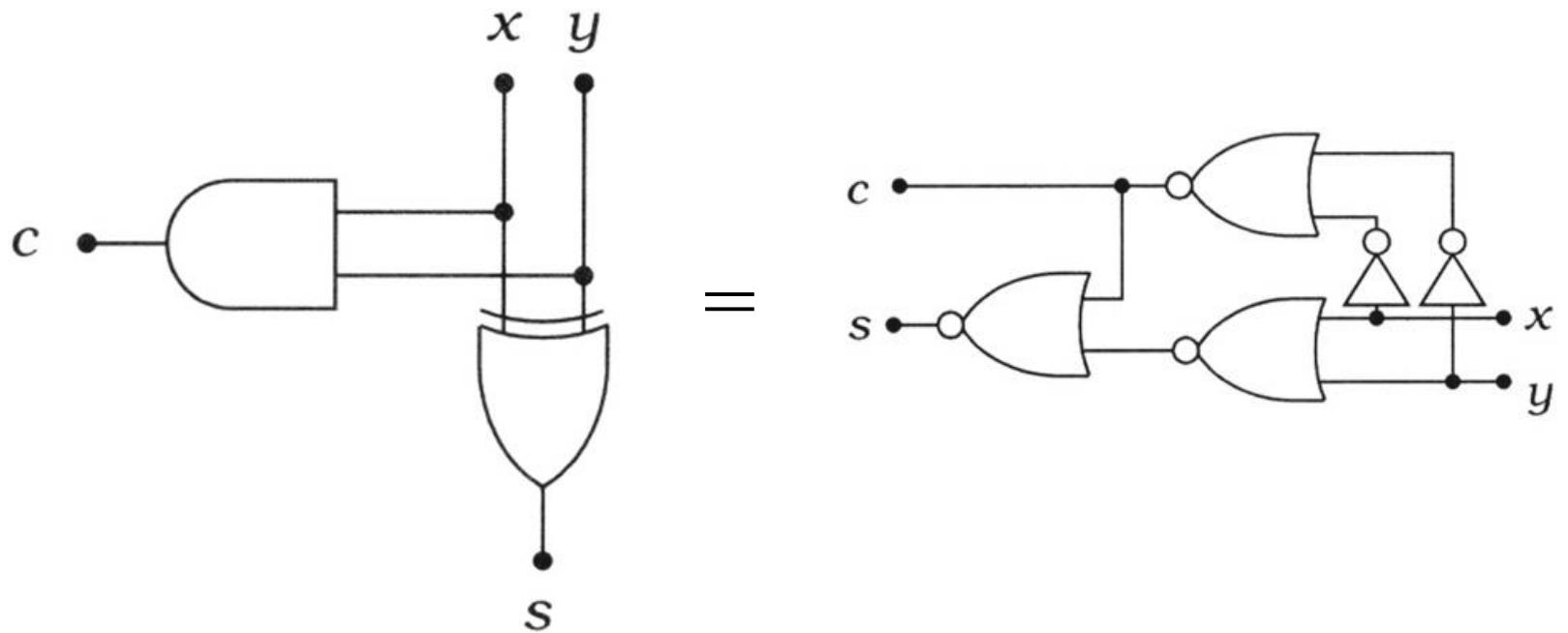
Additional of two 4 bit binary numbers

Single bit addition


$$\begin{array}{r} c_i \\ a_i \\ + b_i \\ \hline c_{i+1} \ s_i \end{array}$$

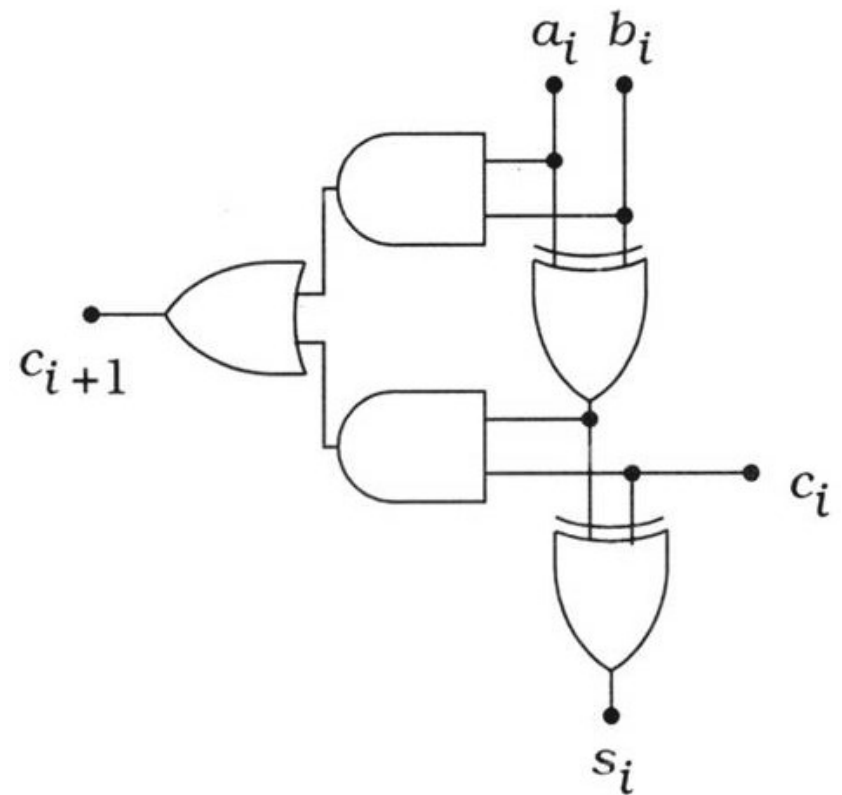
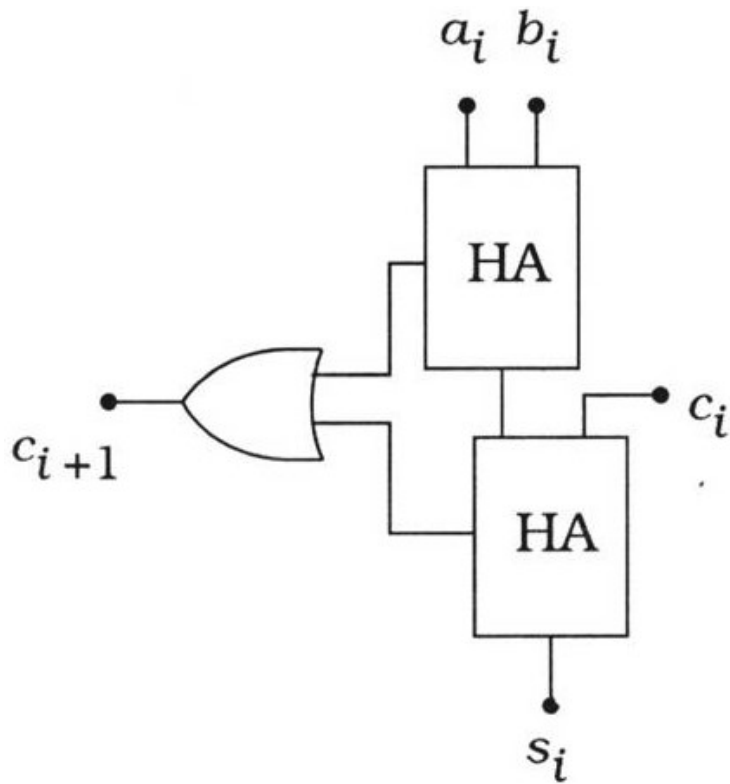
$$\begin{aligned} s_i &= a_i \oplus b_i \oplus c_i \\ c_{i+1} &= a_i \cdot b_i + c_i \cdot (a_i \oplus b_i) \\ \text{or} \\ c_{i+1} &= a_i b_i + c_i \cdot (a_i + b_i) \end{aligned}$$

Half Adder Logic

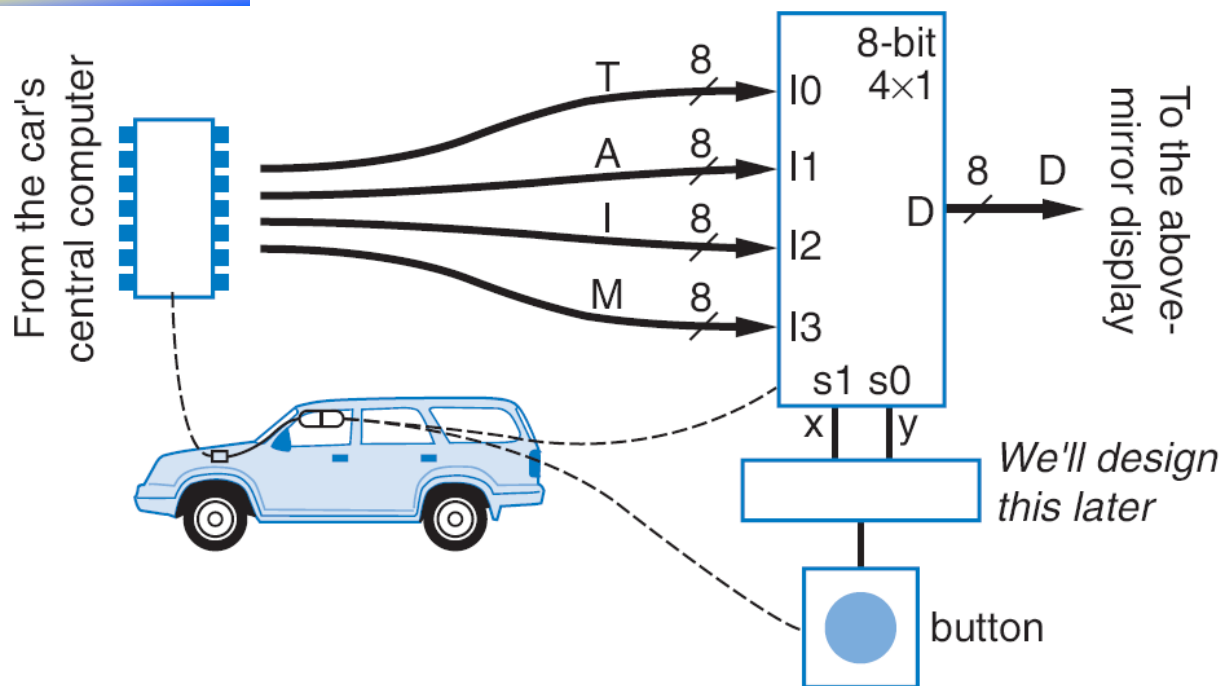


Full Adder Circuit

You can use 2 half adders and a OR gate to implement the Full Adder



Multiplexer (Example)



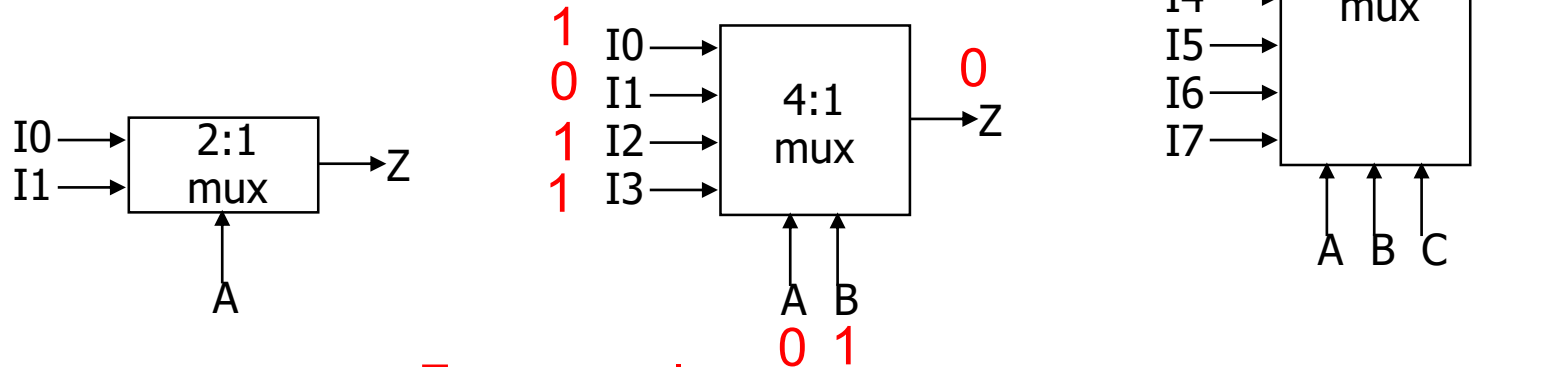
- Four possible display items
 - Temperature (T), Average miles-per-gallon (A), Instantaneous mpg (I), and Miles remaining (M) -- each is 8-bits wide
 - Choose which to display using two inputs x and y
 - Use 8-bit 4x1 mux

Multiplexers/selectors

- 2:1 mux: $Z = A'I_0 + AI_1$
- 4:1 mux: $Z = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$
- 8:1 mux: $Z = A'B'C'I_0 + A'B'CI_1 + A'BC'I_2 + A'BCI_3 + AB'C'I_4 + AB'CI_5 + ABC'I_6 + ABCI_7$

- In general: $\sum_{k=0}^{2^n-1} m_k I_k$

– shorthand form for a $2^n:1$ Mux

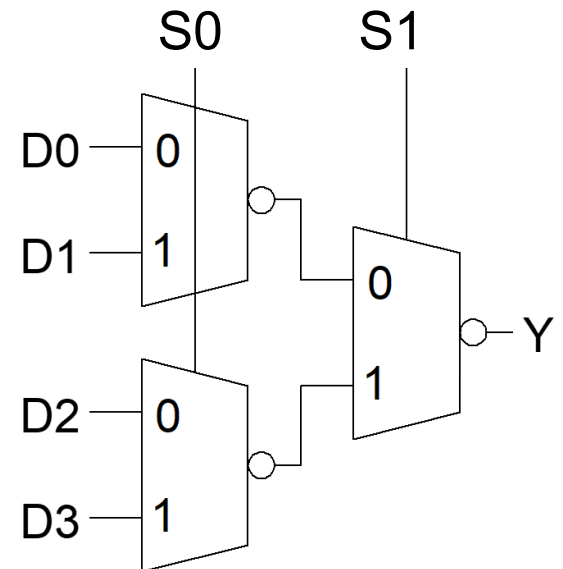
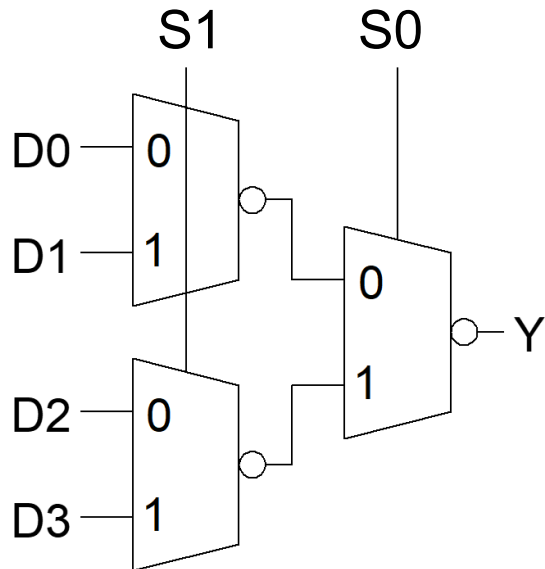


For example

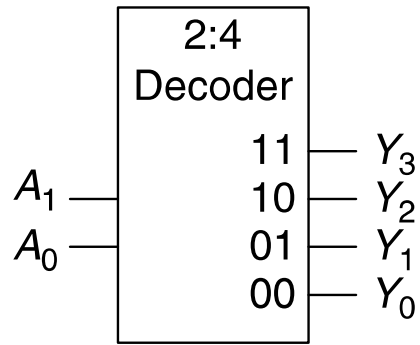
4:1 Multiplexer

- Which figure shows the correct 4:1 multiplexer implementation?

- A. Left
- B. Right
- C. Both
- D. Neighter



2 to 4 Decoder



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

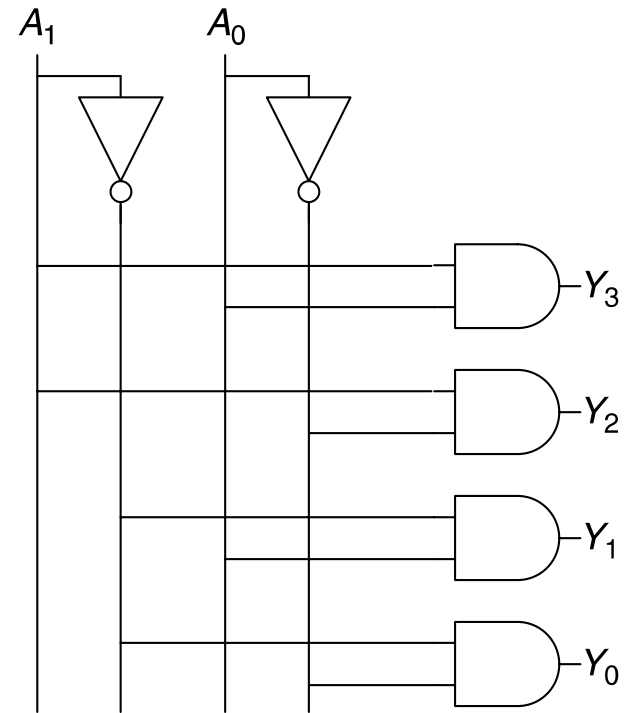
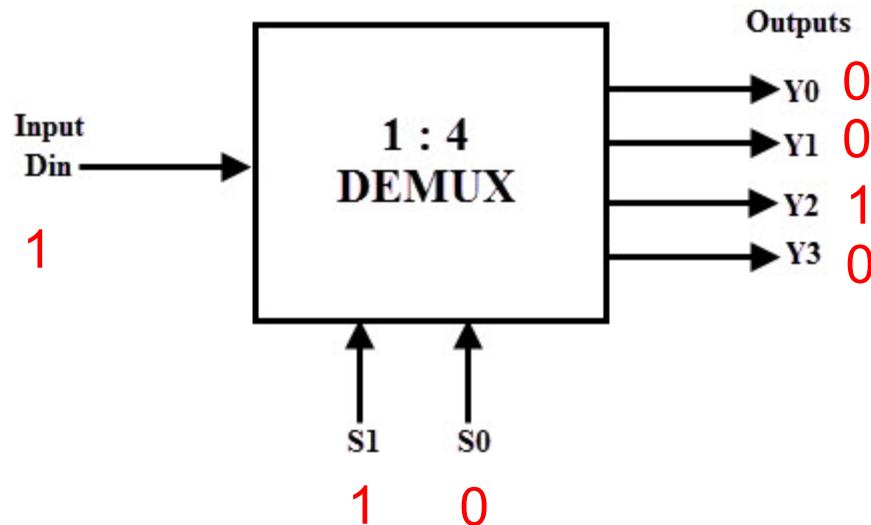


Figure 2.64 2:4 decoder implementation

Demultiplexers/decoders

- Decoders/demultiplexers: general concept
 - single data input, n control inputs, 2^n outputs
 - control inputs (called “selects” (S)) represent binary index of output to which the input is connected
 - data input usually called “enable”



For example



CSE140L: Components and Design Techniques for Digital Systems Lab

Verilog HDL

Slides from Tajana Simunic Rosing

Hardware description languages



- Used to describe & model the operation of digital circuits.
- Specify simulation procedure for the circuit and check its response.
 - Simulation requires a logic simulator.
- Synthesis: transformation of the HDL description into a physical implementation (transistors, gates)
 - When a human does this, it is called logic design.
 - When a machine does this, it is called synthesis.



HDLs

- Abel (circa 1983) - developed by Data-I/O
 - targeted to programmable logic devices
 - not good for much more than state machines
- ISP (circa 1977) - research project at CMU
 - simulation, but no synthesis
- Verilog (circa 1985) - developed by Gateway (absorbed by Cadence)
 - similar to C
 - delays are the only interaction with the simulator
 - fairly efficient and easy to write
 - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
 - VHSIC Hardware Description Language (VHSIC is Very High Speed Integrated Circuit).
 - simulation semantics visible; very general but verbose
 - IEEE standard

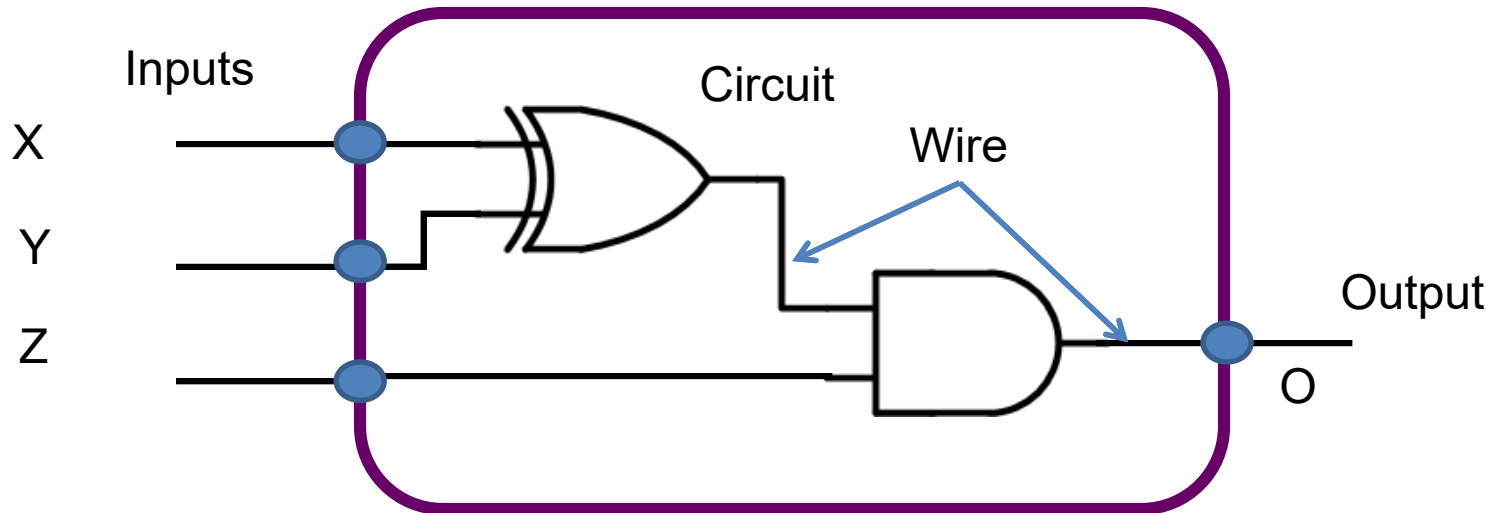
* In US, the commercial industries tend to use more Verilog, while the aerospace and defense industries more heavily favor VHDL

Verilog Usage

- Verilog may be used to model circuits and behaviors at various levels of abstraction:
 - Algorithmic.
 - Behavioral.
 - Logic
 - Gate
 - Transistor
- An FPGA is a “programmable” hardware where you can “download” your synthesized HDL implementation
- Projects in HDL are tested on FPGAs prior to fabrication
- Transistor and gate level modeling are not appropriate for design with FPGA devices.

Verilog

Example Verilog Module



- In this class we will build and simulate circuits.
- We will build hierarchical systems
- We will test for
 - Functional correctness
 - Performance
 - Efficiency

```
module sample (X,Y,Z,O);  
  
input X,Y,Z;  
output O;  
  
// Describe the circuit using logic symbols  
assign O = (X^Y)&Z;  
  
endmodule
```

Verilog

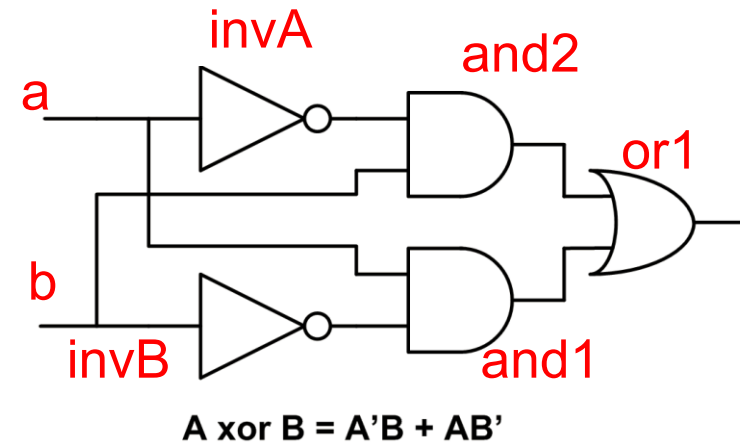


- Supports **structural** and **behavioral** descriptions
- Structural
 - explicit structure of the circuit
 - e.g., each logic gate instantiated and connected to others
- Behavioral
 - program describes input/output behavior of circuit
 - many structural implementations could have same behavior
 - e.g., different implementation of one Boolean function



Structural model

```
module xor_gate (out, a, b);  
    input      a, b;  
    output     out;  
    wire       abar, bbar, t1, t2;  
  
    inverter invA (abar, a);  
    inverter invB (bbar, b);  
    and_gate and1 (t1, a, bbar);  
    and_gate and2 (t2, b, abar);  
    or_gate  or1 (out, t1, t2);  
  
endmodule
```



Behavioral model

```
module xor_gate (out_or, out_and, a, b);  
    input        a, b;  
    output       out_or, out_and;  
    reg         out_or, out_and;  
  
    always @ (a or b) begin  
        out_or = a ^ b;  
    end  
  
    assign out_and = a & b;  
  
endmodule
```

Question

- Is the following model structural?
- A. Yes
- B. No
- C. I don't know!

```
module and_or (in1, in2, in3, in4, out );  
    input in1, in2, in3, in4;  
    output out;  
    wire tmp  
    and #10 u1 (tmp, in1, in2 ) , u2 (undec, in3, in4 );  
    or #20 (out, tmp, undec );  
endmodule
```

Data Values



- For our logic design purposes, we'll consider Verilog to have four different bit values:
 - 0, logic zero.
 - 1, logic one.
 - z, high impedance.
 - x, unknown.



Data Types and Values

- There are two main data types in Verilog.
 - Wires.
 - Regs.
- These data types may be single bit or multi-bit.
 - The general syntax is: {bit width}'{base}{value}
 - 4'd14 // 4-bit value, specified in decimal
 - 4'he // 4-bit value, specified in hex
 - 4'b1110 // 4-bit value, specified in binary
 - 4'b10xz // 4-bit value, with x and z, in binary

Data Types



- Wires:
 - “continuously assigned” values and do not store information.
 - may have multiple drivers assigning values.
 - When multiple drivers exist, the simulator resolves them into a single value for the wire.
 - Every time a driver changes its output value, the resulting wire value is re-evaluated.
- This behaves much like an electrical wire...



Data Types

- Regs

- “procedurally assigned” values that store information until the next value assignment is made.
- can be used to model combinational or sequential logic.
- The name “reg” does **not** mean it is a register!
- A reg may be assigned by multiple processes.
- Other reg varieties include integer, real, and time.

Operators cont.

- Arithmetic operators (signed and can generate carry out):
 - * Multiplication.
 - / Division.
 - % Modulus.
 - + Addition.
 - Subtraction.
- Logical shift operators:
 - << Shift left.
 - >> Shift right.
- Relational operators:
 - > Greater than.
 - < Less than.
 - >= Greater than or equal.
 - <= Less than or equal.

Operators cont.

- Comparison operators:
 - == Equality operator (compares to z, x are invalid).
 - != Not equal.
- Binary bit-wise operators:
 - & Bit-wise logical AND.
 - | Bit-wise logical OR.
 - ^ Bit-wise logical XOR.
 - ~^ Bit-wise logical XNOR.
- Binary logical operators:
 - && Binary logical AND.
 - || Binary logical OR.
- Operator for conditional selection:
 - sel ? value_if_sel_is_one : value_if_sel_is_zero
 - oe ? driven_value : 1'bz

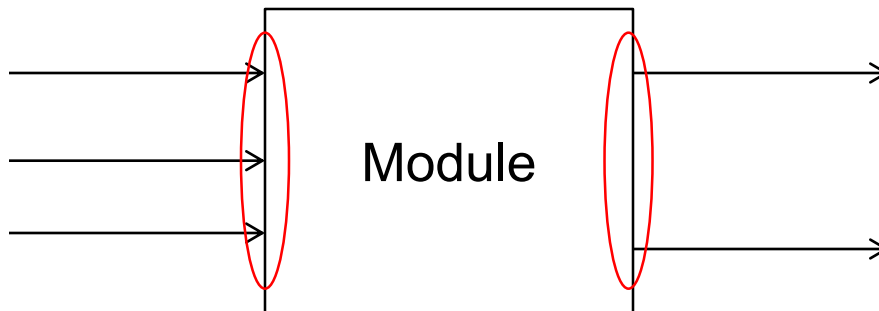
Modules and Ports

- Consider a top level module declaration:

```
module testbench;  
    // Top level modules do not have ports.  
endmodule
```

- Consider a module declaration with ports:

```
module two_input_xor (in1, in2, out);  
    input in1, in2;  
    output out;  
    // We'll add more later...  
endmodule
```



Ports: define how
the module
interacts with the
external world

Modules and Ports

- Ports may be of type {input, output, inout} and can also be multiple bits wide.

```
module some_random_design (fred, bob, joe, sam, tom, ky);  
    input          fred; // 1-bit input port  
    input  [7:0]  bob;   // 8-bit input port  
    output         joe;  // 1-bit output port  
    output [1:0]  sam;   // 2-bit output port  
    inout         tom;   // 1-bit bidirectional port  
    inout  [3:0]  ky;    // 4-bit bidirectional port  
  
    // Some design description would be here...  
endmodule
```

Port and Data Types

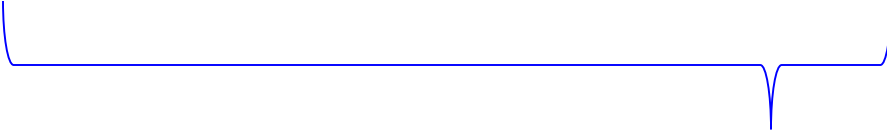
- Input port:
 - driven from *outside* the module by a wire or a reg,
 - *inside* the module it can only drive a wire
- Output port
 - driven from *inside* the module by a wire or a reg,
 - *outside* the module it can only drive a wire.
- Inout port
 - May be driven by a wire, and drive a wire.

Instantiation

```
module testbench;
wire    sig3;           // wire driven by submodule
reg     sig1, sig2;     // regs assigned by testbench

two_input_xor my_xor (.in1(sig1), .in2(sig2), .out(sig3));

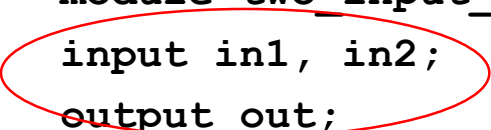
endmodule
```



Explicit module
connection:

.<port>(signal)

```
module two_input_xor (in1, in2, out);
input in1, in2;
output out;
// We'll add more later...
endmodule
```



By default, ports are wires inside the
module (so if you want regs instead,
you must explicitly state it)

Operators

- Used in both procedural and continuous assignments.
- Listed in the order of evaluation precedence:
 - **{ } is used for concatenation.**

Say you have two 1-bit data objects, sam and bob.
{sam, bob} is a 2-bit value from concatenation
 - **{ { } } is used for replication.**

Say you have a 1-bit data object, ted.
{4{ted}} is a 4-bit value, ted replicated four times.
 - **Unary operators:**
 - ! Performs logical negation (test for non-zero).
 - ~ Performs bit-wise negation (complements).
 - & Performs unary reduction of logical AND.
 - | Performs unary reduction of logical OR.
 - ^ Performs unary reduction of logical XOR.

Continuous Assignment

- Continuous assignments are made with the assign statement:
 - **assign LHS = RHS;**
 - The left hand side, LHS, must be a wire.
 - The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.
- The value of the RHS is **continuously driven** onto the wire of the LHS.
- Values x and z are allowed and processed.
- All assign statements operate concurrently.

Continuous Assignment

```
module two_input_xor (in1, in2, out);  
    input in1, in2;                // use these as a wire  
    output out;                    // use this as a wire
```

```
    assign out = in1 ^ in2;
```

```
endmodule
```

```
module two_input_xor (in1, in2, out);  
    input in1, in2;  
    output out;  
    wire product1, product2;
```

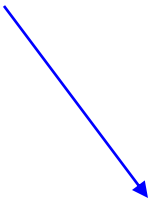
```
    assign product1 = in1 & !in2;    // could have done all in  
    assign product2 = !in1 & in2;    // assignment of out with  
    assign out = product1 | product2; // bigger expression
```

```
endmodule
```

Continuous Assignment

```
module two_input_xor (in1, in2, out);  
    input in1, in2;  
    output out;  
        assign out = (in1 != in2);  
endmodule
```

```
module two_input_xor (in1, in2, out);  
    input in1, in2;  
    output out;  
        assign out = in1 ? (!in2) : (in2);  
endmodule
```



What does this line represent?

Procedural Assignment

- Models **combinational** and **sequential** logic
- Operates concurrently and is preceded by event control.
- In block statements start with “begin” and end with “end”.
 - Single assignments can omit begin and end.
- A sensitivity list specifies events which cause the execution to begin:
 - always @(a or b) // any changes in a or b
 - always @(posedge a) // a transitions from 0 to 1
 - always @(negedge a) // a transitions from 1 to 0
 - always @(a or b or negedge c or posedge d)

Procedural Assignment

```
initial
begin
    // These procedural assignments are executed
    // one time at the beginning of the simulation.
end

always @(sensitivity list)
begin
    // These procedural assignments are executed
    // whenever the events in the sensitivity list
    // occur.
end
```

Procedural Assignment

- Inside a block, two types of assignments exist:
 - `LHS = RHS;` // blocking assignment
 - `LHS <=RHS;` // non-blocking assignment
 - Do not mix them in a given block.
- Assignment rules:
 - The left hand side, LHS, must be a reg.
 - The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.

Procedural Assignment

- Do I use blocking or non-blocking assignments?
 - **Blocking assignments** are especially useful when describing combinational logic.
 - Lines execute in order.
 - You can “build up” complex logic expressions.
 - Blocking assignments make your description evaluate in the order you described it.
 - **Non-blocking assignments** are useful when describing sequential logic.
 - All lines execute at the same time.
 - At a clock or reset event, all memory elements change state at the same time, best modeled by non-blocking assignments.
- For conditional assignments use if-else and various types of case statements.
- You also can make use of additional timing control:
 - wait, #delay, repeat, while, etc...

Procedural Assignment

Example: Blocking Vs Nonblocking Assignments

```
1 module block_nonblock();
2 reg a, b, c, d, e, f;
3
4 // Blocking assignments
5 initial begin
6     a = #10 1'b1; // The simulator assigns 1 to a at time 10
7     b = #20 1'b0; // The simulator assigns 0 to b at time 30
8     c = #40 1'b1; // The simulator assigns 1 to c at time 70
9 end
10
11 // Nonblocking assignments
12 initial begin
13     d <= #10 1'b1; // The simulator assigns 1 to d at time 10
14     e <= #20 1'b0; // The simulator assigns 0 to e at time 20
15     f <= #40 1'b1; // The simulator assigns 1 to f at time 40
16 end
17
18 endmodule
```