Logic Synthesis

- Primitive logic gates, universal gates
- Truth tables and sum-of-products
- Logic simplification
- Karnaugh Maps, Quine-McCluskey
- General implementation techniques: muxes and look-up tables (LUTs)

Reminder: Lab #1 due this Thursday!

Late Policies

- Lab 1 check-offs – sign-up on checkoff queue in lab – FIFO during staffed lab hours.
- Please don't assume that *you* can wait until the last minute!
- No check-offs on Friday or Saturday
- Lab grade = Checkoff + Verilog grade (~~equal weighting~~)
- On-time check-off:
  - 20%/day late penalty (no penalty for Friday or Saturday)
  - Max penalty 80% reduction.
- All labs must be checked off before you can start your final project. We've learned that if you're struggling with the labs, the final project won't go very well.

- Lpset – must be submitted on time.

Schematics & Wiring

- IC power supply connections generally not drawn. All integrated circuits need power!
- Use standard color coded wires to avoid confusion.
  - red: positive
  - black: ground or common reference point
  - Other colors: signals
- Circuit flow, signal flow left to right
- Higher voltage on top, ground negative voltage on bottom
- Neat wiring helps in debugging!

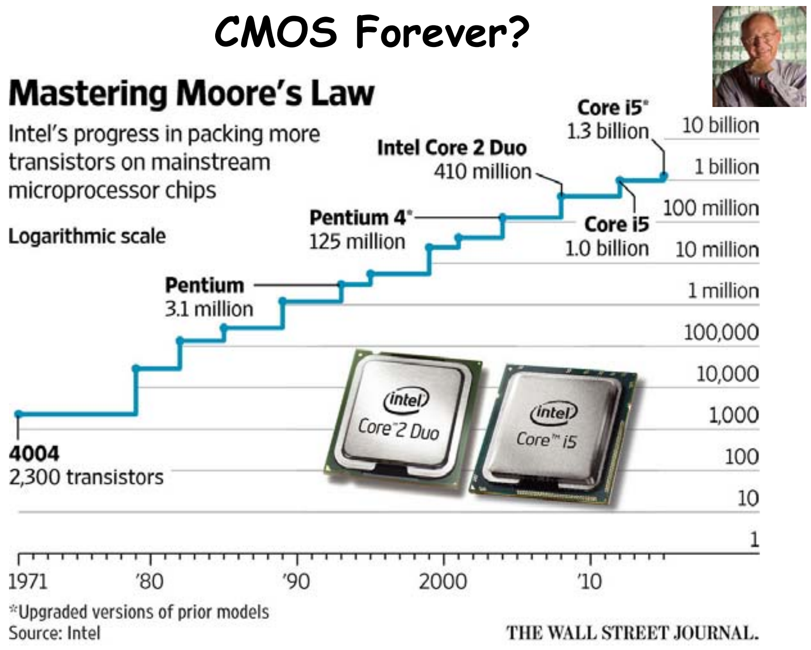
Wire Gauge

- Wire gauge: diameter is inversely proportional to the wire gauge number. Diameter increases as the wire gauge decreases. 2, 1, 0, 00, 000(3/0) up to 7/0.

- Resistance
  - 22 gauge .0254 in 16 ohm/1000 feet
  - 12 gauge .08 in 1.5 ohm/1000 feet
  - High voltage AC used to reduce loss

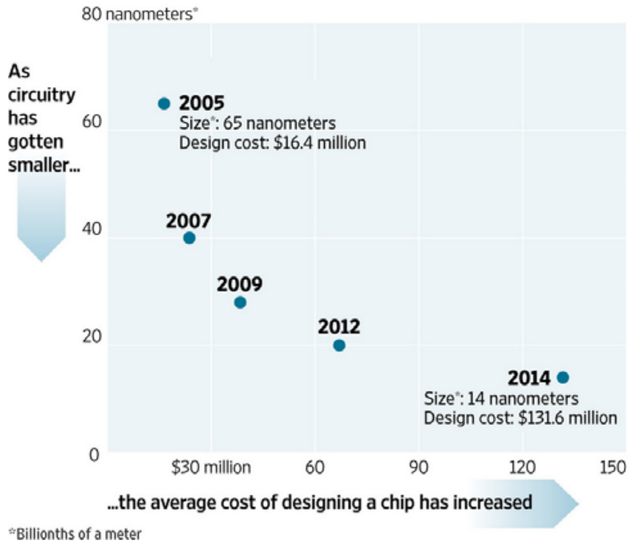
- 1 cm cube of copper has a resistance of 1.68 micro ohm (resistance of copper wire scales linearly : length/area)

CMOS Forever?



Diminishing Returns *

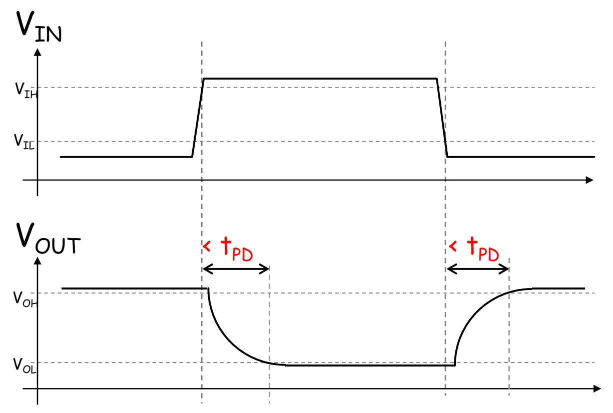
Creating smaller circuitry has placed more transistors on chips but triggered higher costs.



* Intel

Timing Specifications

Propagation delay (t_{PD}): An *upper bound* on the delay from valid inputs to valid outputs (aka "$t_{PD,MAX}$")

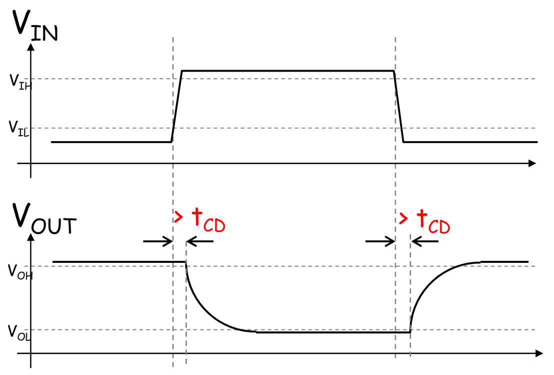


Design goal:
minimize propagation delay

Contamination Delay

an optional, additional timing spec

Contamination delay(t_{CD}): A *lower bound* on the delay from invalid inputs to invalid outputs (aka "$t_{PD,MIN}$")

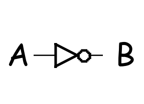


Do we really need t_{CD}?

Usually not… it'll be important when we design circuits with registers (coming soon!)

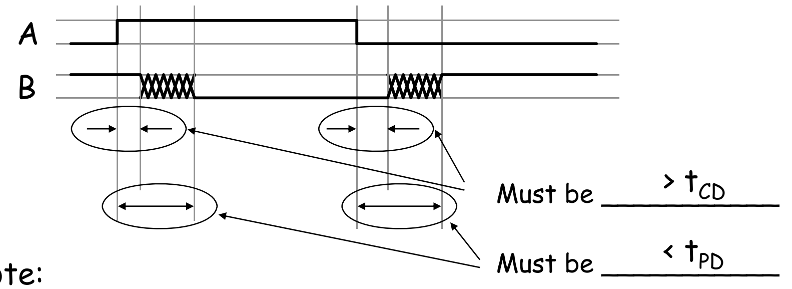
If t_{CD} is not specified, safe to assume it's 0.

The Combinational Contract



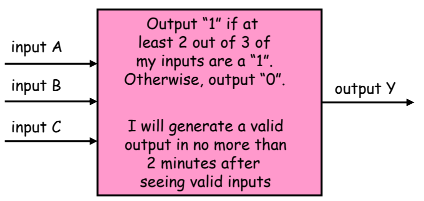
A	B
0	1
1	0

t_{PD} propagation delay
t_{CD} contamination delay



Note:
1. *No Promises* during ⊠⊠⊠
2. Default (conservative) spec: $t_{CD} = 0$

Functional Specifications


input A
input B
input C
Output "1" if at least 2 out of 3 of my inputs are a "1". Otherwise, output "0".
I will generate a valid output in no more than 2 minutes after seeing valid inputs
output Y

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

3 binary inputs
so 2^3 = 8 rows in our truth table

An concise, unambiguous technique for giving the functional specification of a combinational device is to use a *truth table* to specify the output value for each possible combination of input values (N binary inputs -> 2^N possible combinations of input values).

Here's a Design Approach

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

1. Write out our functional spec as a truth table
2. Write down a Boolean expression with terms covering each '1' in the output:

$$Y = \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot C + A\cdot B\cdot\overline{C} + A\cdot B\cdot C$$

This approach creates equations of a particular form called

SUM-OF-PRODUCTS

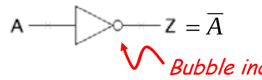
Sum (+): ORs

Products (•): ANDs

-it's systematic!
-it works!
-it's easy!
-are we done yet???

S-O-P Building Blocks

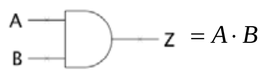
INVERTER:



Bubble indicates inversion

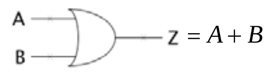
A	Z
0	1
1	0

AND:



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

OR:



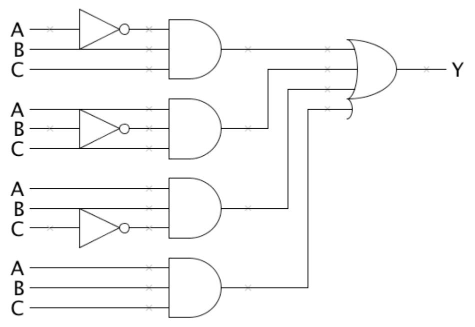
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

Straightforward Synthesis

$$Y = \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot C + A\cdot B\cdot\overline{C} + A\cdot B\cdot C$$

We can use SUM-OF-PRODUCTS to implement *any* logic function.

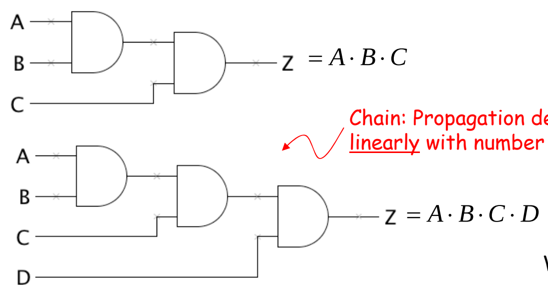
Only need 3 gate types: INVERTER, AND, OR



Propagation delay:

- 3 levels of logic
- No more than 3 gate delays assuming gates with an arbitrary number of inputs. But, in general, we'll only be able to use gates with a bounded number of inputs (bound is ~4 for most logic families).

ANDs and ORs with > 2 inputs

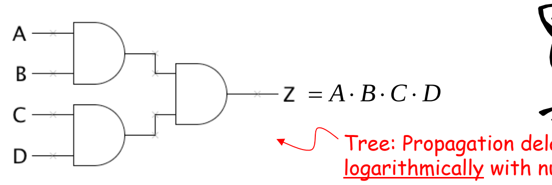


$Z = A\cdot B\cdot C$

Chain: Propagation delay increases linearly with number of inputs

$Z = A\cdot B\cdot C\cdot D$

Which one should I use?

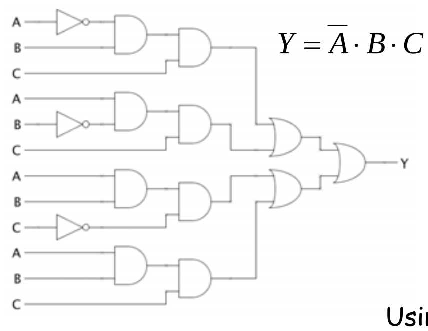


$Z = A\cdot B\cdot C\cdot D$

Tree: Propagation delay increases logarithmically with number of inputs

SOP w/ 2-input gates

Previous example restricted to 2-input gates:



$$Y = \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot C + A\cdot B\cdot\overline{C} + A\cdot B\cdot C$$

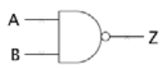
	INV	AND2	OR2
t_{PD}	8ps	15ps	18ps
t_{CD}	1ps	3ps	3ps

Using the timing specs given to the left, what are t_{PD} and t_{CD} for this combinational circuit?

Hint: to find overall t_{PD} we need to find max t_{PD} considering all paths from inputs to outputs.

More Building Blocks

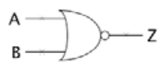
NAND (not AND)



$Z = \overline{A\cdot B}$

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

NOR (not OR)

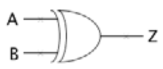


$Z = \overline{A+B}$

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

CMOS gates are naturally inverting so we want to use NANDs and NORs in CMOS designs…

XOR (exclusive OR)



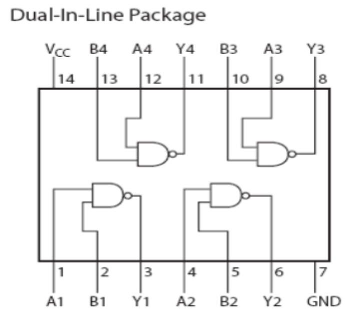
$Z = A \oplus B$

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

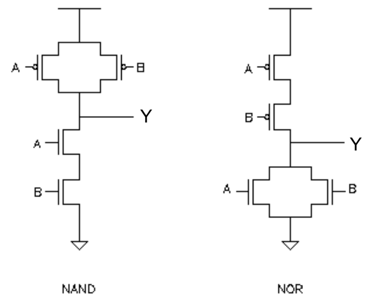
XOR is very useful when implementing parity and arithmetic logic. Also used as a "programmable inverter": if A=0, Z=B; if A=1, Z=~B

Wide fan-in XORs can be created with chains or trees of 2-input XORs.

NAND – NOR Internals

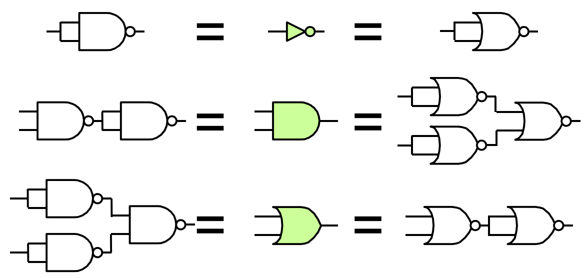


This device contains four independent gates each of which performs the logic NAND function.



Universal Building Blocks

NANDs and NORs are <u>universal</u>:



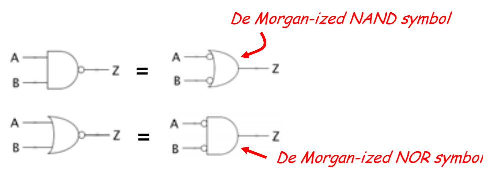
Any logic function can be implemented using only NANDs (or, equivalently, NORs). Note that chaining/treeing technique doesn't work directly for creating wide fan-in NAND or NOR gates. But wide fan-in gates can be created with trees involving both NANDs, NORs and inverters.

SOP with NAND/NOR

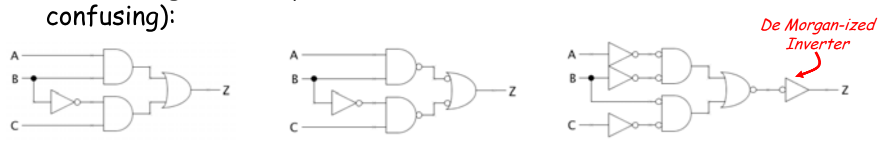
When designing with NANDs and NORs one often makes use of De Morgan's laws:

NAND form: $\overline{A \cdot B} = \overline{A} + \overline{B}$

NOR form: $\overline{A + B} = \overline{A} \cdot \overline{B}$



So the following "SOP" circuits are all equivalent (note the use of De Morgan-ized symbols to make the inversions less confusing):



AND/OR form

NAND/NAND form

This will be handy in Lab 1 since you'll be able to use just 7400's to implement your circuit!

NOR/NOR form

All these "extra" inverters may seem less than ideal but often the buffering they provide will reduce the capacitive load on the inputs and increase the output drive.

Logic Simplification

- Can we implement the same function with fewer gates? Before trying we'll add a few more tricks in our bag.
- BOOLEAN ALGEBRA:

OR rules:	$a+1=1 \quad a+0=a \quad a+a=a$
AND rules:	$a \cdot 1=a \quad a \cdot 0=0 \quad a \cdot a=a$
Commutative:	$a+b=b+a \quad a \cdot b=b \cdot a$
Associative:	$(a+b)+c=a+(b+c) \quad (a \cdot b) \cdot c=a \cdot (b \cdot c)$
Distributive:	$a \cdot (b+c)=a \cdot b+a \cdot c \quad a+b \cdot c=(a+b) \cdot (a+c)$
Complements:	$a+\overline{a}=1 \quad a \cdot \overline{a}=0$
Absorption:	$a+a \cdot b=a \quad a+\overline{a} \cdot b=a+b \quad a \cdot (a+b)=a \quad a \cdot (\overline{a}+b)=a \cdot b$
De Morgan's Law:	$\overline{a \cdot b}=\overline{a}+\overline{b} \quad \overline{a+b}=\overline{a} \cdot \overline{b}$
Reduction:	$a \cdot b+\overline{a} \cdot b=b \quad (a+b) \cdot (\overline{a}+b)=b$

Key to simplification: equations that match the pattern of the LHS (where "b" might be any expression) tell us that when "b" is true, the value of "a" doesn't matter. So "a" can be eliminated from the equation, getting rid of two 2-input ANDs and one 2-input OR.

Boolean Minimization:

An Algebraic Approach

Lets simplify the equation from slide #3:

$$Y = \overline{A}\cdot B\cdot C + A\cdot \overline{B}\cdot C + A\cdot B\cdot \overline{C} + A\cdot B\cdot C$$

Using the identity

$$\alpha A + \alpha \overline{A} = \alpha$$

For any expression α and variable A:

$$Y = \overline{A}\cdot B\cdot C + A\cdot \overline{B}\cdot C + A\cdot B\cdot \overline{C} + A\cdot B\cdot C$$

$$Y = B\cdot C + A\cdot C + A\cdot B$$

The tricky part: some terms participate in more than one reduction so can't do the algebraic steps one at a time!

Karnaugh Maps: A Geometric Approach

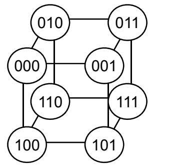
K-Map: a truth table arranged so that terms which differ by exactly one variable are adjacent to one another so we can see potential reductions easily.

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Here's the layout of a 3-variable K-map filled in with the values from our truth table:

C \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

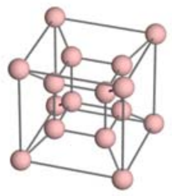
It's cyclic. The left edge is adjacent to the right edge. It's really just a flattened out cube.



On to Hyperspace

Here's a 4-variable K-map:

CD \ AB	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	1	1	0	1
10	1	1	0	1

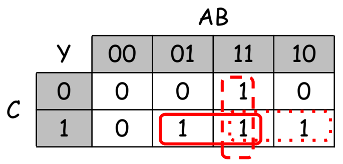


Again it's cyclic. The left edge is adjacent to the right edge, and the top is adjacent to the bottom.

We run out of steam at 4 variables – K-maps are hard to draw and use in three dimensions (5 or 6 variables) and we're not equipped to use higher dimensions (> 6 variables)!

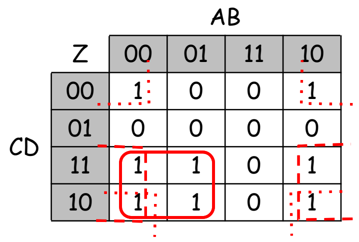
Finding Subcubes

We can identify clusters of "irrelevent" variables by circling adjacent subcubes of 1s. A subcube is just a lower dimensional cube.



C \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Three 2x1 subcubes



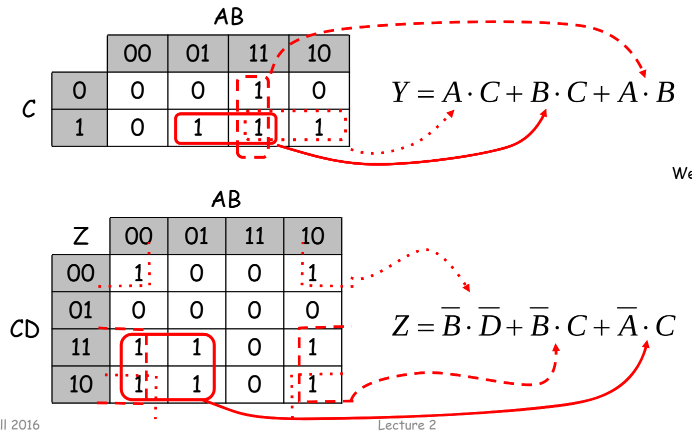
CD \ AB	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	1	1	0	1
10	1	1	0	1

Three 2x2 subcubes

The best strategy is generally a greedy one.
- Circle the largest N-dimensional subcube (2^N adjacent 1's)
  4x4, 4x2, 4x1, 2x2, 2x1, 1x1
- Continue circling the largest remaining subcubes (even if they overlap previous ones)
- Circle smaller and smaller subcubes until no 1s are left.

Write Down Equations

Write down a product term for the portion of each cluster/subcube that is invariant. You only need to include enough terms so that all the 1's are covered. Result: a minimal sum of products expression for the truth table.



$$Y = A \cdot C + B \cdot C + A \cdot B$$

$$Z = \overline{B} \cdot \overline{D} + \overline{B} \cdot C + \overline{A} \cdot C$$

Two-Level Boolean Minimization

Two-level Boolean minimization is used to find a sum-of-products representation for a multiple-output Boolean function that is optimum according to a given cost function. The typical cost functions used are the number of product terms in a two-level realization, the number of literals, or a combination of both. The two steps in two-level Boolean minimization are:

•Generation of the set of prime product-terms for a given function.

•Selection of a minimum set of prime terms to implement the function.

We will briefly describe the Quine-McCluskey method which was the first algorithmic method proposed for two-level minimization and which follows the two steps outlined above. State-of-the-art logic minimization algorithms are all based on the Quine-McCluskey method and also follow the two steps above.

Prime Term Generation

Start by expressing your Boolean function using 0-terms (product terms with no don't care care entries). For compactness the table for example 4-input, 1-output function F(w,x,y,z) shown to the right includes only entries where the output of the function is 1 and we've labeled each entry with it's decimal equivalent.

F = f(W,X,Y,Z)

W X Y Z	*label*
0 0 0 0	0
0 1 0 1	5
0 1 1 1	7
1 0 0 0	8
1 0 0 1	9
1 0 1 0	10
1 0 1 1	11
1 1 1 0	14
1 1 1 1	15

Look for pairs of 0-terms that differ in only one bit position and merge them in a 1-term (i.e., a term that has exactly one '-' entry). Next 1-terms are examined in pairs to see if the can be merged into 2-terms, etc. Mark k-terms that get merged into (k+1) terms so we can discard them later.

1-terms:

0, 8	-000	[A]
5, 7	01-1	[B]
7,15	-111	[C]
8, 9	100-	
8,10	10-0	
9,11	10-1	
10,11	101-	
10,14	1-10	
11,15	1-11	
14,15	111-	

2-terms:

8, 9,10,11	10--	[D]
10,11,14,15	1-1-	[E]

3-terms: none!

Label unmerged terms: these terms are prime!

Example due to Srini Devadas

Prime Term Table

An "X" in the prime term table in row R and column K signifies that the 0-term corresponding to row R is contained by the prime corresponding to column K.

Goal: select the minimum set of primes (columns) such that there is at least one "X" in every row. This is the classical minimum covering problem.

	A	B	C	D	E	
0000	X	.	.	.	.	→ A is essential -000
0101	.	X	.	.	.	→ B is essential 01-1
0111	.	X	X	.	.	
1000	X	.	.	X	.	
1001	.	.	.	X	.	→ D is essential 10--
1010	.	.	.	X	X	
1011	.	.	.	X	X	
1110	.	.	.	.	X	→ E is essential 1-1-
1111	.	.	X	.	X	

Each row with a single X signifies an essential prime term since any prime implementation will have to include that prime term because the corresponding 0-term is not contained in any other prime.

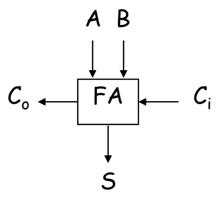
In this example the essential primes "cover" all the 0-terms.

$$F = f(W,X,Y,Z) = \bar{X}\bar{Y}\bar{Z} + \bar{W}XZ + W\bar{X} + WY$$

Logic that defies SOP simplification

C_i	A	B	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full Adder



S

C/AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

C_O

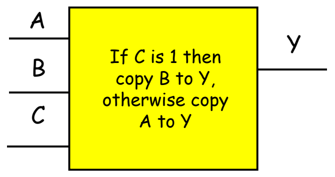
C/AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$S = \overline{A}\cdot B\cdot \overline{C} + A\cdot \overline{B}\cdot \overline{C} + \overline{A}\cdot \overline{B}\cdot C + A\cdot B\cdot C = A \oplus B \oplus C_i$$

$$C_O = A\cdot C + B\cdot C + A\cdot B$$

The sum S doesn't have a simple sum-of-products implementation even though it can be implemented using only two 2-input XOR gates.

Logic Synthesis Using MUXes

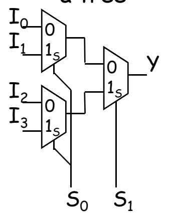


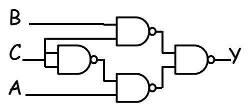
2-input Multiplexer

Truth Table

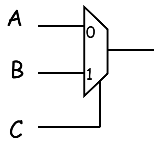
C	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

A 4-input Mux implemented as a tree





schematic



Gate symbol

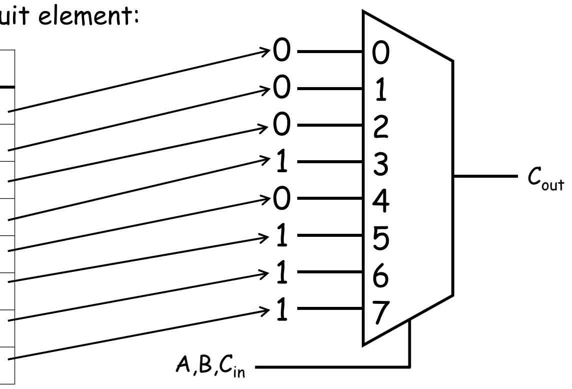
Systematic Implementation of Combinational Logic

Consider implementation of some arbitrary Boolean function, F(A,B)

... using a MULTIPLEXER as the only circuit element:

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Full-Adder Carry Out Logic

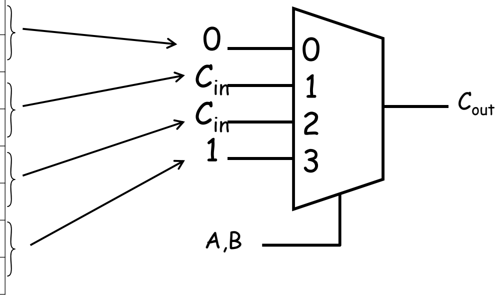


Systematic Implementation of Combinational Logic

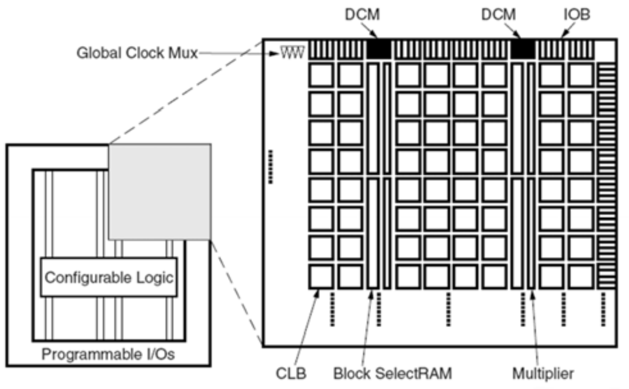
Same function as on previous slide, but this time let's use a 4-input mux

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Full-Adder Carry Out Logic



Xilinx Virtex II FPGA

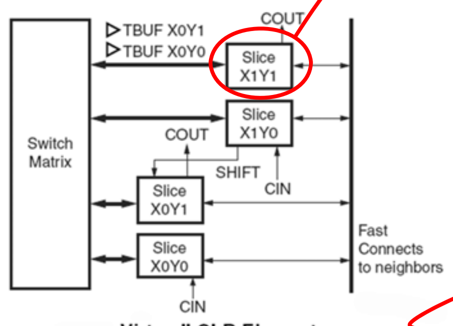


Virtex-II Architecture Overview

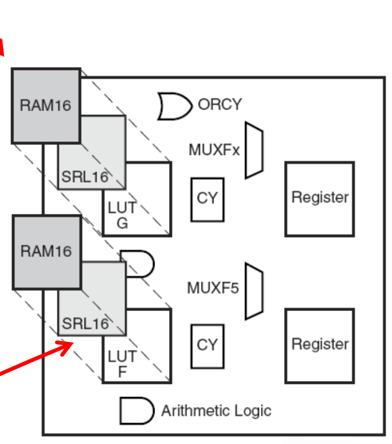
XC2V6000:
- 957 pins, 684 IOBs
- CLB array: 88 cols x 96/col = 8448 CLBs
- 18Kbit BRAMs = 6 cols x 24/col = 144 BRAMs = 2.5Mbits
- 18x18 multipliers = 6 cols x 24/col = 144 multipliers

Figures from Xilinx Virtex II datasheet

Virtex II CLB



Virtex-II CLB Element

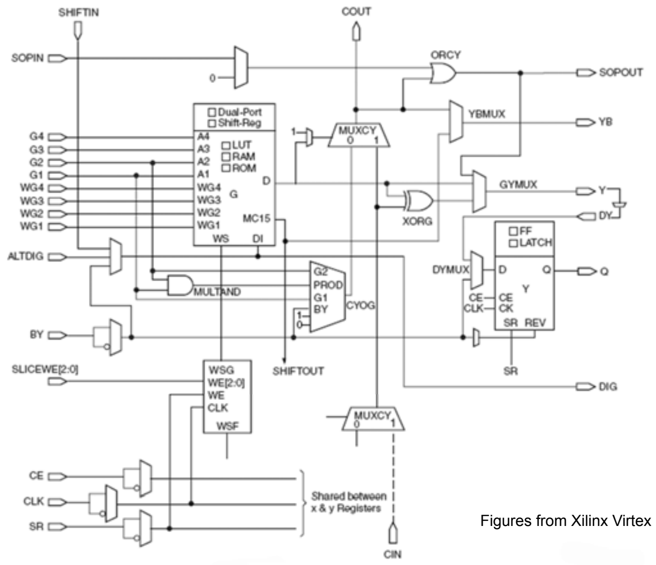


Virtex-II Slice Configuration

16 bits of RAM which can be configured as a 16x1 single- or dual-port RAM, a 16-bit shift register, or a 16-location lookup table

Figures from Xilinx Virtex II datasheet

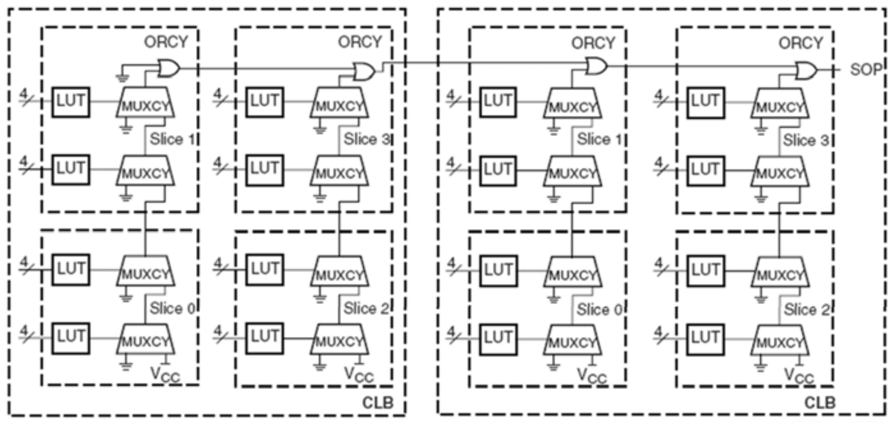
Virtex II Slice Schematic



Virtex-II Slice (Top Half)

Figures from Xilinx Virtex II datasheet

Virtex II Sum-of-products



Horizontal Cascade Chain

Figures from Xilinx Virtex II datasheet

Spartan 6 FPGA

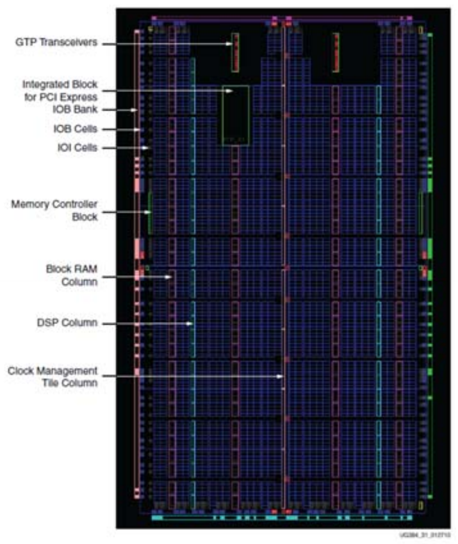
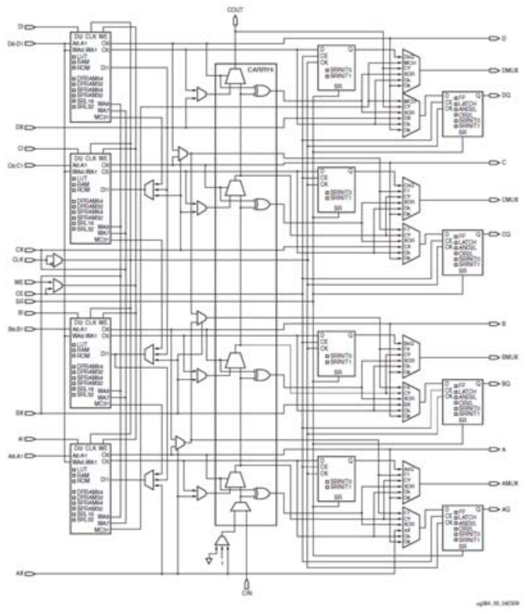


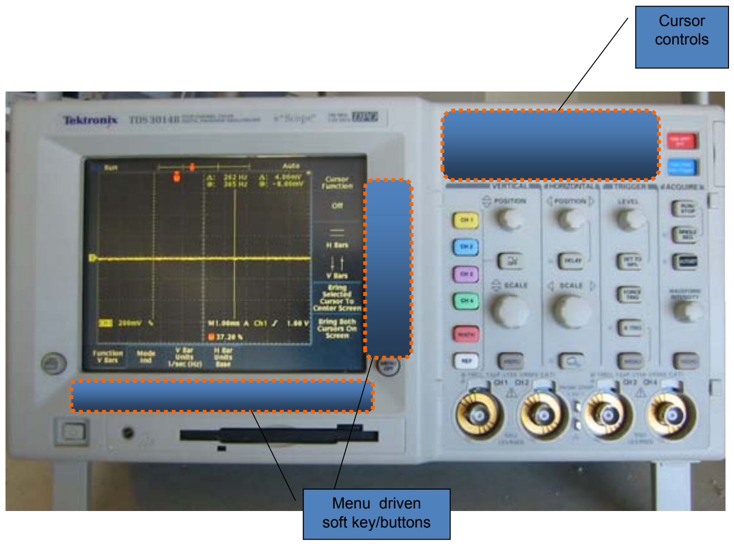
Figure 31: **XC6SLX45T Floorplan View in PlanAhead**

Spartan 6 SliceM Schematic



Figures from Xilinx Spartan 6 CLB datasheet

Oscilloscope



Oscilloscope Controls

- Auto Set, soft menu keys
- Trigger
  - channel,
  - slope,
  - Level
- Input
  - AC, DC coupling,
  - 10x probe,
  - 1khz calibration source,
  - probe calibration,
  - bandwidth filter
- Signal measurement
  - time,
  - frequency,
  - voltage
  - cursors
  - single sweep
- Image capture