Logic Synthesis

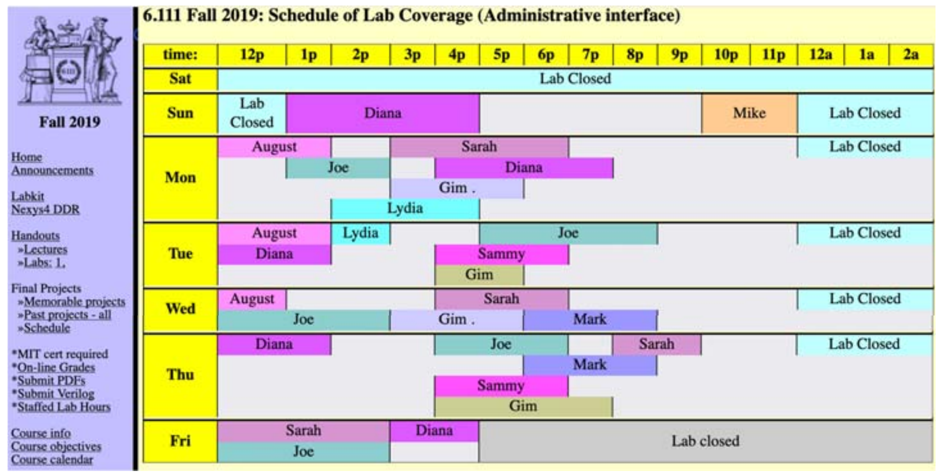
- Primitive logic gates, universal gates
- Truth tables and sum-of-products
- Logic simplification, Karnaugh Maps
- General implementation techniques:
  muxes and look-up tables (LUTs)
- Nexy4
- Verilog basics

Handouts
- lecture slides,
- LPset #2

Reminder: Lab #1 due this Thu/Fri

Lab Hours

Lab hours: eds.mit.edu/labs
Sun 1-11:45p, M-R 9-11:45p, F 9-5p



Fall 2019

Home
Announcements

Labkit
Nexys4 DDR

Handouts
»Lectures
»Labs: 1.

Final Projects
»Memorable projects
»Past projects - all
»Schedule

*MIT cert required
*On-line Grades
*Submit PDFs
*Submit Verilog
*Staffed Lab Hours

Course info
Course objectives
Course calendar

6.111 Fall 2019: Schedule of Lab Coverage (Administrative interface)

time:	12p	1p	2p	3p	4p	5p	6p	7p	8p	9p	10p	11p	12a	1a	2a
Sat	Lab Closed														
Sun	Lab Closed	Diana									Mike		Lab Closed		
Mon	August			Sarah									Lab Closed		
		Joe			Diana										
				Gim .											
			Lydia												
Tue	August		Lydia			Joe							Lab Closed		
	Diana				Sammy										
					Gim										
Wed	August				Sarah								Lab Closed		
	Joe			Gim .			Mark								
Thu	Diana				Joe				Sarah				Lab Closed		
							Mark								
					Sammy										
					Gim										
Fri	Sarah			Diana		Lab closed									
	Joe														

Late Policies

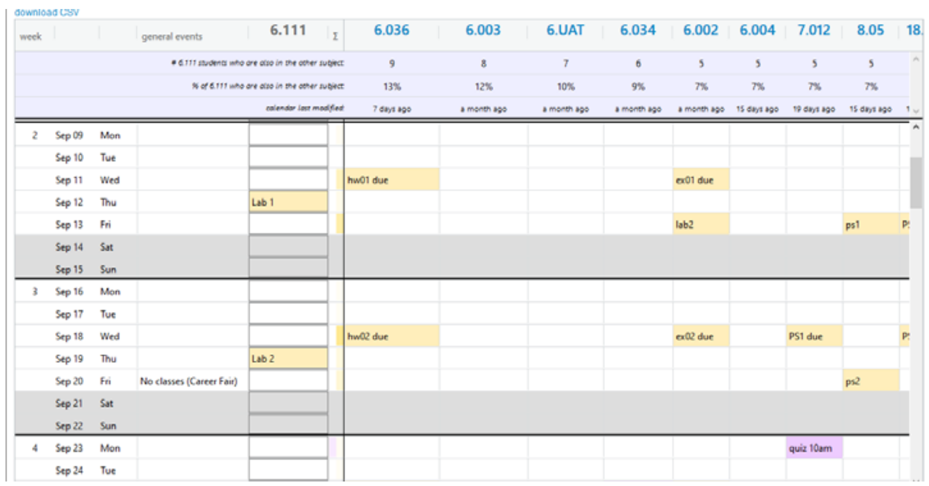
- Lab 1 check-offs (early) – sign-up on checkoff queue in lab – FIFO during staffed lab hours. Note bench number…
- Please don't assume that you can wait until the last minute!
- No check-offs Saturday
- Checkoff: Lab 1: Thu 5p, Fri 1pm.
- Lab grade = Checkoff + Verilog grade (when needed)
- Late labs:
  - 1 point/day late penalty (no penalty for Saturday)
  - Late completed labs will receive 1 point.
  - 5 slack days available. This covers illness, interviews, overload, etc.
- A missing lab will result in a failing grade. We've learned that if you're struggling with the labs, the final project won't go very well.
- Lpset – no late submissions. Solutions at times presented in lecture.

Checkoff Process

- May checkoff at any time prior to checkoff date.
- On checkoff date, checkoff will staff's be main priority
- Two checkoff dates: last name A-M (Thu), N-Z (Fri)
- Thu checkoff starts at 5pm, Fri 1pm
- Schedule time on google doc

Check for conflicting dates

https://unical.csail.mit.edu/fa19



download CSV

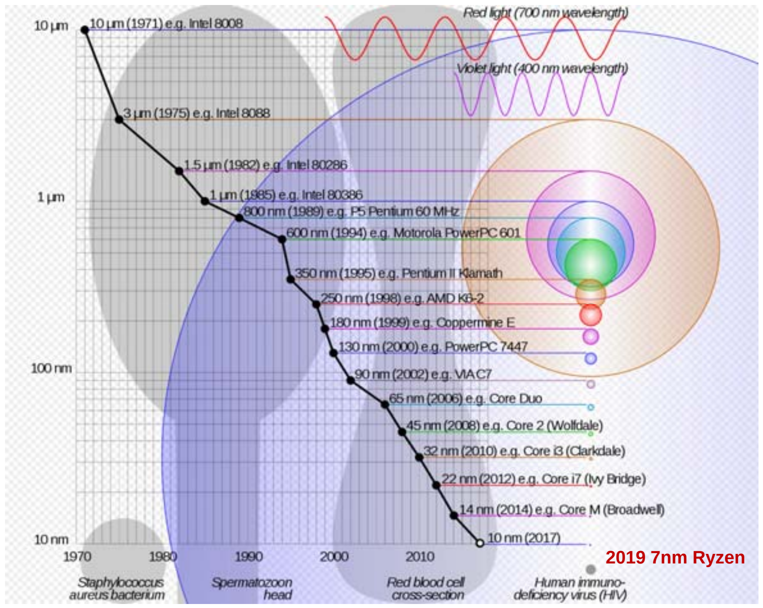
week			general events	6.111	6.036	6.003	6.UAT	6.034	6.002	6.004	7.012	8.05	18.
			# 6.111 students who are also in the other subject		9	8	7	6	5	5	5	5	
			% of 6.111 who are also in the other subject		13%	12%	10%	9%	7%	7%	7%	7%	
			calendar last modified		7 days ago	a month ago	a month ago	a month ago	a month ago	15 days ago	19 days ago	15 days ago	1
2	Sep 09	Mon											
	Sep 10	Tue											
	Sep 11	Wed			hw01 due				ex01 due				
	Sep 12	Thu		Lab 1									
	Sep 13	Fri							lab2			ps1	P
	Sep 14	Sat											
	Sep 15	Sun											
3	Sep 16	Mon											
	Sep 17	Tue											
	Sep 18	Wed			hw02 due				ex02 due		PS1 due		P
	Sep 19	Thu		Lab 2									
	Sep 20	Fri	No classes (Career Fair)									ps2	
	Sep 21	Sat											
	Sep 22	Sun											
4	Sep 23	Mon									quiz 10am		
	Sep 24	Tue											

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

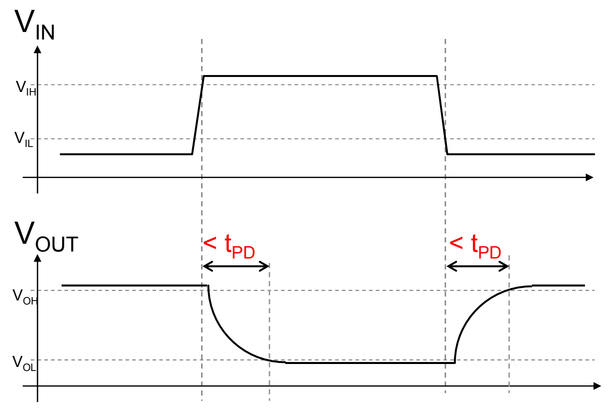
- Wire gauge: diameter is inversely proportional to the wire gauge number. Diameter increases as the wire gauge decreases. 2, 1, 0, 00, 000(3/0) up to 7/0.
- Resistance
  - 22 gauge .0254 in 16 ohm/1000 feet
  - 12 gauge .08 in 1.5 ohm/1000 feet
  - High voltage AC used to reduce loss
- 1 cm cube of copper has a resistance of 1.68 micro ohm (resistance of copper wire scales linearly : length/area)



By Cmglee CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=16991155

Timing Specifications

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

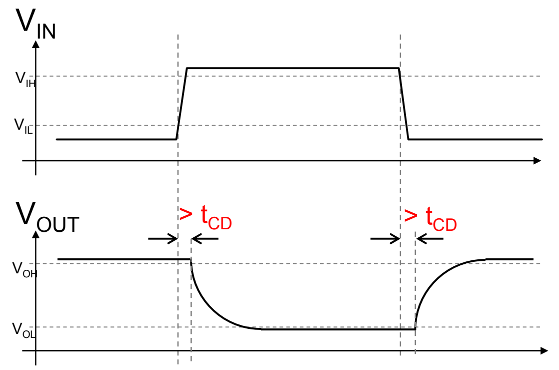


Design goal:
minimize propagation delay

Contamination Delay

an optional, additional timing spec

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

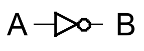


Do we really need t_{CD}?

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

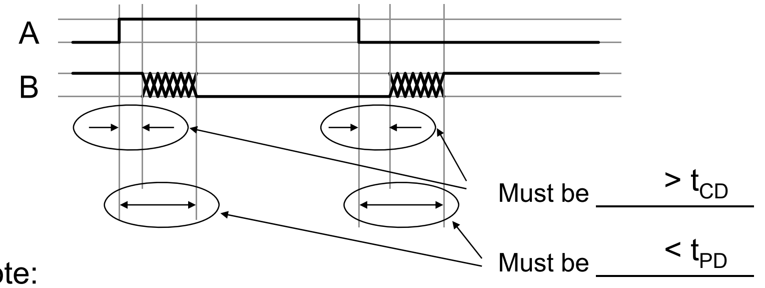
If t_{CD} is not specified, safe to assume it's 0.

The Combinational Contract Design and Parts Quality



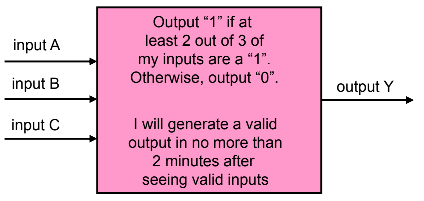
A	B
0	1
1	0

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

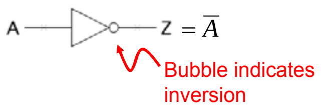
SUM-OF-PRODUCTS

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

Sum (+): ORs	Verilog: ||
Products (•): ANDs	Verilog: &&

S-O-P Building Blocks

INVERTER:

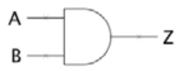


$Z = \overline{A}$

assign Z = !A

A	Z
0	1
1	0

AND:

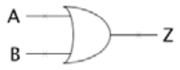


$Z = A\cdot B$

assign Z = A && B

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

OR:



$Z = A + B$

assign Z = A || B

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

Basic Boolean operators

- Bitwise operators perform bit-oriented operations on vectors
  - ~(4'b0101) = {~0,~1,~0,~1} = 4'b1010
  - 4'b0101 & 4'b0011 = {0&0, 1&0, 0&1, 1&1} = 4'b0001
- Logical operators return one-bit (true/false) results
  - !(4'b0101) = 1'b0

Bitwise

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b a ^~ b	XNOR

Logical

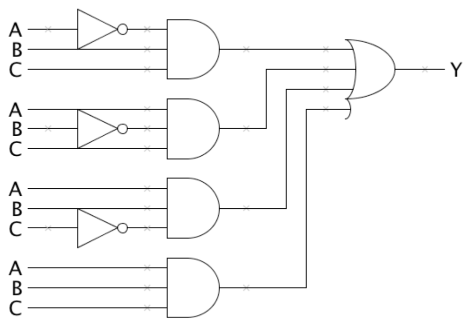
!a	NOT
a && b	AND
a || b	OR
a == b a != b	[in]equality returns x when x or z in bits. Else returns 0 or 1
a === b a !== b	case [in]equality returns 0 or 1 based on bit by bit comparison

Note distinction between ~a and !a when operating on multi-bit values

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.

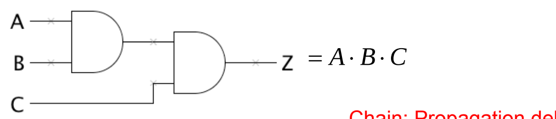


Only need 3 gate types: INVERTER, AND, OR

Propagation delay:

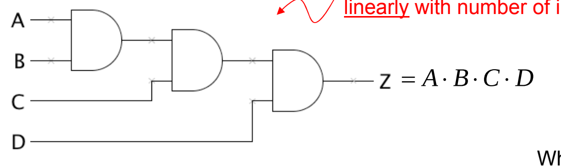
- 3 levels of logic
- No more than 3 gate delays assuming gates with an arbitrary number of inputs. But, in general, we'll only be able to use gates with a bounded number of inputs (bound is ~4 for most logic families).

ANDs and ORs with > 2 inputs



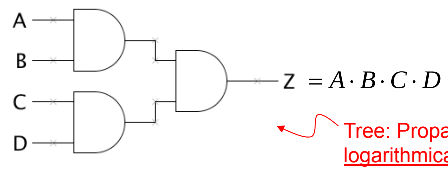
$= A \cdot B \cdot C$

Chain: Propagation delay increases linearly with number of inputs



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

Which one should I use?

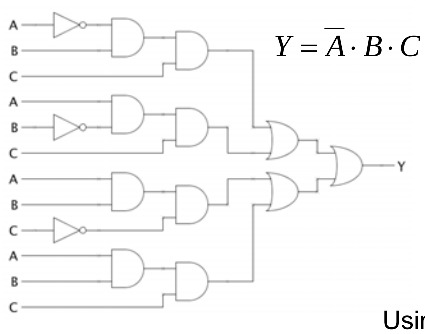


$= A \cdot B \cdot C \cdot D$

Tree: Propagation delay increases logarithmically with number of inputs

SOP w/ 2-input gates

Previous example restricted to 2-input gates:



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

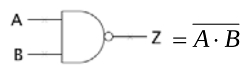
	INV	AND2	OR2
t_{PD}	8ps	15ps	18ps
t_{CD}	1ps	3ps	3ps

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)

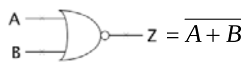


$= \overline{A \cdot B}$

assign Z = !(A&&B)

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

NOR (not OR)



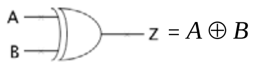
$= \overline{A + B}$

assign Z = !(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)



$= A \oplus B$

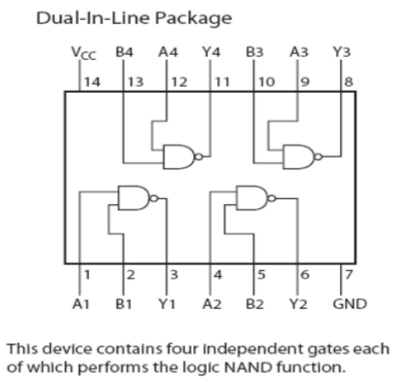
assign Z = A^B

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

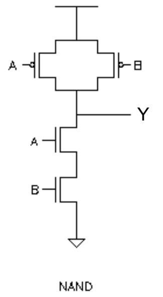
XOR is very useful when implementing parity and arithmetic logic. Also used as a "programmable inverter": if A=0, Z=B; if A=1, Z=~B

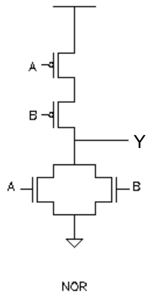
Wide fan-in XORs can be created with chains or trees of 2-input XORs.

NAND – NOR Internals



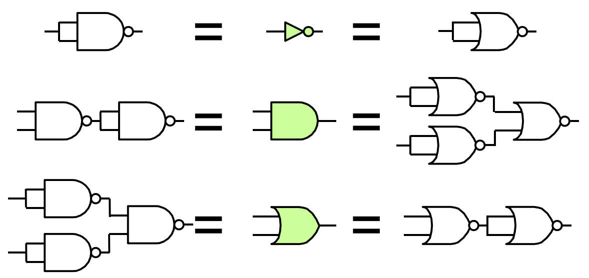
This device contains four independent gates each of which performs the logic NAND function.





Universal Building Blocks

NANDs and NORs are universal:



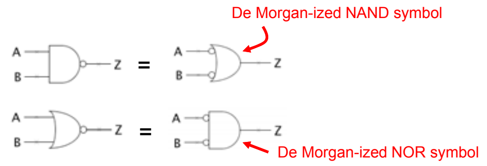
Any logic function can be implemented using only NANDs (or, equivalently, NORs). Note that chaining/treeing technique doesn't work directly for creating wide fan-in NAND or NOR gates. But wide fan-in gates can be created with trees involving both NANDs, NORs and inverters.

SOP with NAND/NOR

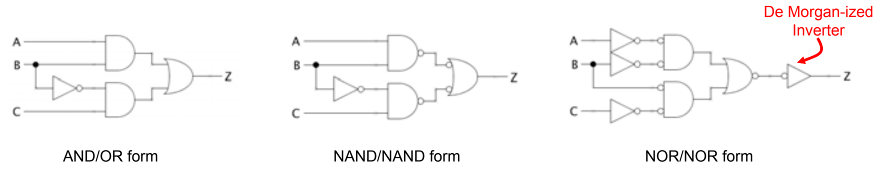
When designing with NANDs and NORs one often makes use of De Morgan's laws:

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):



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

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!

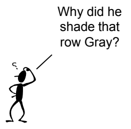
Karnaugh Maps: A Geometric Approach

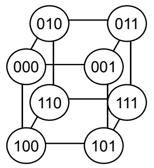
K-Map: a truth table arranged so that terms which differ by exactly one variable are adjacent to one another so we can see potential reductions easily.

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:

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



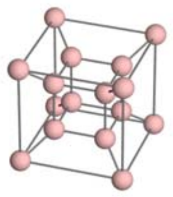


It's cyclic. The left edge is adjacent to the right edge. It's really just a flattened out cube.

On to Hyperspace

Here's a 4-variable K-map:

		AB			
	Z	00	01	11	10
CD	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.

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

Three 2x1 subcubes

		AB			
	Z	00	01	11	10
CD	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.

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

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

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

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

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-
```

Example due to Srini Devadas

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!

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.

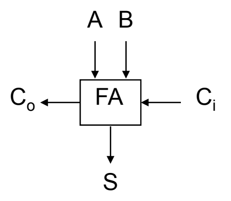
In this example the essential primes "cover" all the 0-terms.

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

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

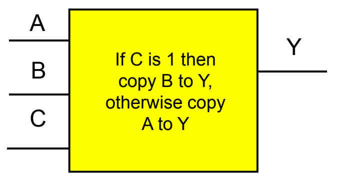
C/AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

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

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

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

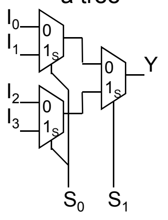


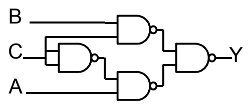
2-input Multiplexer

Truth Table

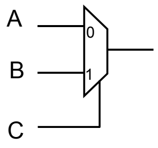
C	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

A 4-input Mux implemented as a tree





schematic



Gate symbol

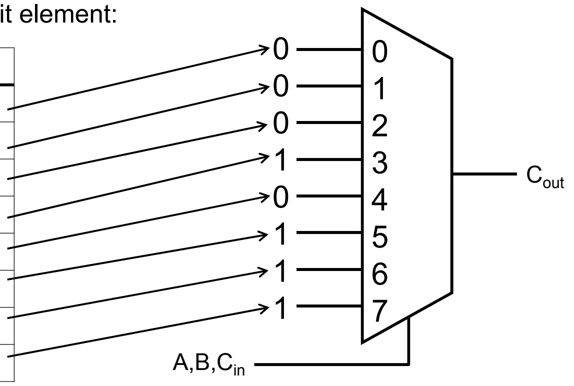
Systematic Implementation of Combinational Logic

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

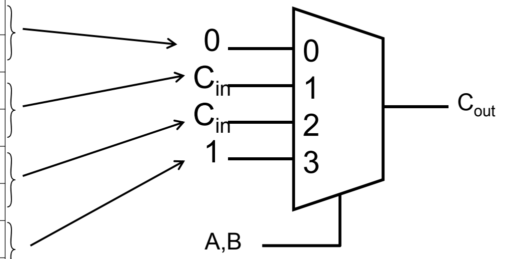


Systematic Implementation of Combinational Logic

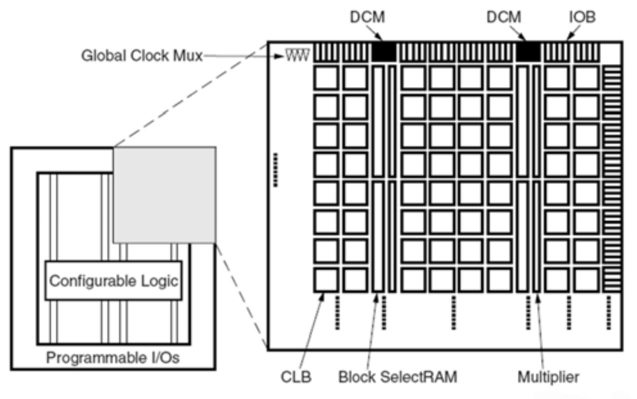
Same function as on previous slide, but this time let's use a 4-input mux

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

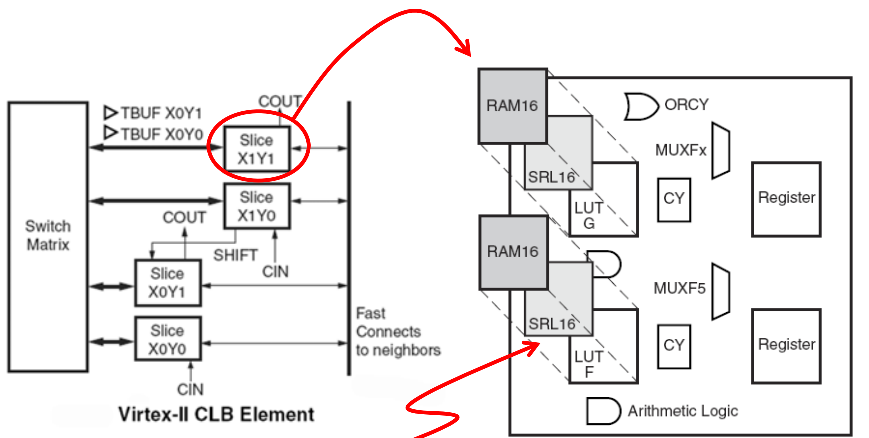


Virtex-II Architecture Overview

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



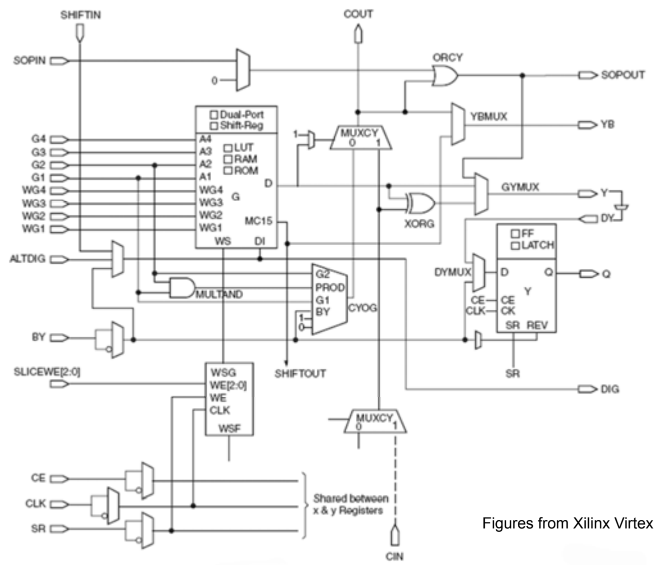
Virtex-II CLB Element

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

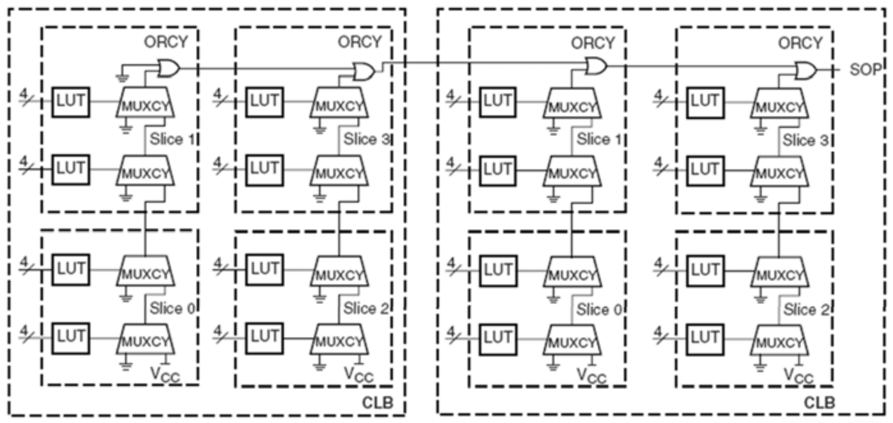
Virtex II Slice Schematic



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

The Need for HDLs

A specification is an engineering contract that lists all the goals for a project:

• goals include area, power, throughput, latency, functionality, test coverage, costs (NREs and piece costs), … Helps you figure out when you're done and how to make engineering tradeoffs. Later on, goals help remind everyone (especially management) what was agreed to at the outset!

• top-down design: partition the project into modules with well-defined interfaces so that each module can be worked on by a separate team. Gives the SW types a head start too! (Hardware/software codesign is currently all the rage…)

- • Example – a well defined Instruction Set Architecture (ISA) can last for generations …

The Need for HDLs (cont'd.)

A behavioral model serves as an executable functional specification that documents the exact behavior of all the individual modules and their interfaces. Since one can run tests, this model can be refined and finally verified through simulation.

We need a way to talk about what hardware should do without actually designing the hardware itself, i.e., we need to separate behavior from implementation. We need a

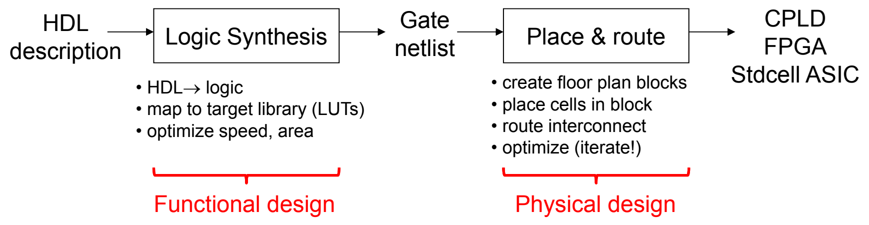
Hardware Description Language

If we were then able to synthesize an implementation directly from the behavioral model, we'd be in good shape!

Using an HDL description

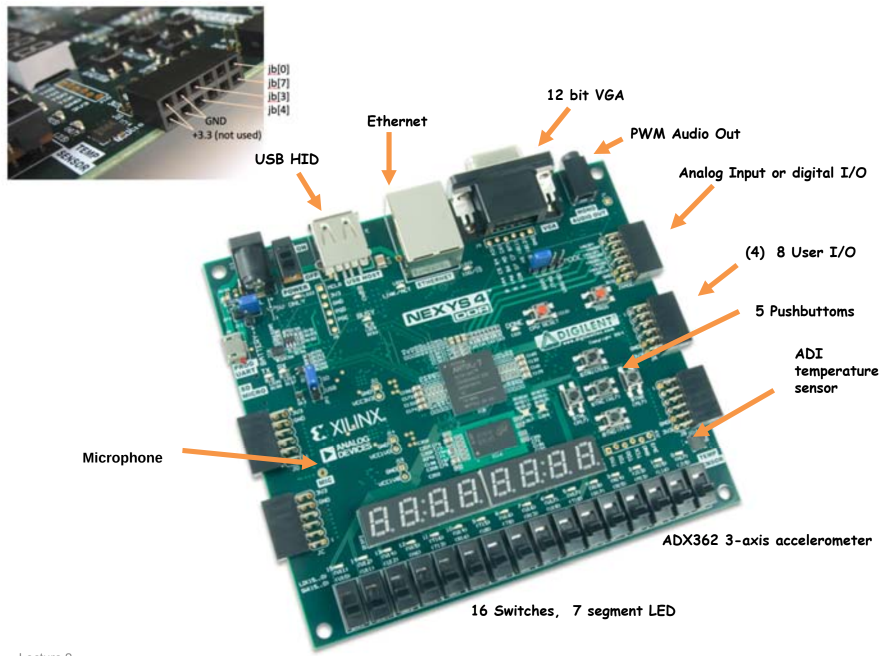
So, we have an executable functional specification that
- documents exact behavior of all the modules and their interfaces
- can be tested & refined until it does what we want

An HDL description is the first step in a mostly automated process to build an implementation directly from the behavioral model

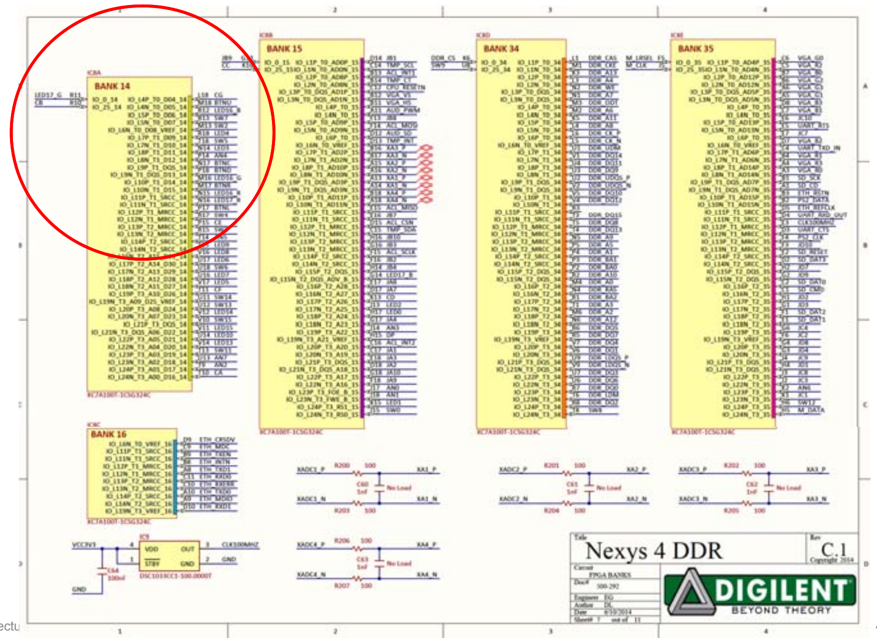


A Tale of Two HDLs

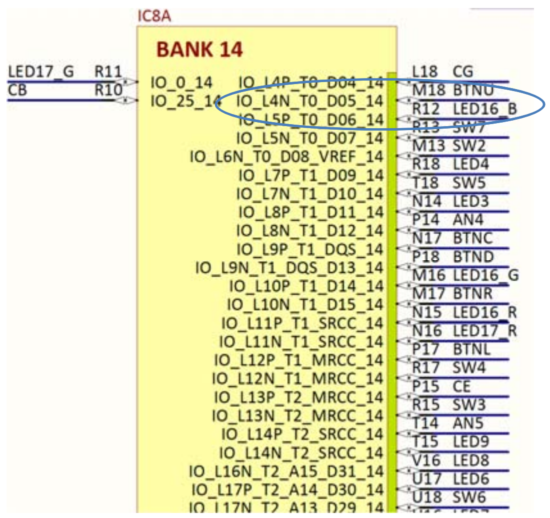
VHDL	Verilog
ADA-like verbose syntax, lots of redundancy (which can be good!)	C-like concise syntax
Extensible types and simulation engine. Logic representations are not built in and have evolved with time (IEEE-1164).	Built-in types and logic representations. Oddly, this led to slightly incompatible simulators from different vendors.
Design is composed of entities each of which can have multiple architectures. A configuration chooses what architecture is used for a given instance of an entity.	Design is composed of modules.
Behavioral, dataflow and structural modeling. Synthesizable subset...	Behavioral, dataflow and structural modeling. Synthesizable subset...
Harder to learn and use, not technology-specific, DoD mandate	Easy to learn and use, fast simulation, good for hardware design



Nexys4 Schematic



XDC to Hardware Mapping



```
set_property -dict { PACKAGE_PIN R12   IOSTANDARD LVCMOS33 }
       [get_ports { led16_b }]; #IO_L5P_T0_D06_14 Sch=led16_b
```

Constraint File

- Text file (.XDC) containing the mapping from a device independent HDL circuit net to the physical I/O pin. This allows Verilog (HDL) to be device independent.

```
set_property -dict { PACKAGE_PIN R12   IOSTANDARD LVCMOS33 }
       [get_ports { led16_b }]; #IO_L5P_T0_D06_14 Sch=led16_b
```

  - led16_b is physically tied to IC package R pin 12
  - Voltage spec based on low voltage CMOS 3.3
  - Schematic name is led16_b #IO_L5P_T0_D06_14

- All signals defined in XDC but commented out.

SystemVerilog *logic* values

Since we're describing hardware, we'll need to represent the values that can appear on wires. SystemVerilog uses a 4-valued logic:

When using a tri-state bus, we'll need to represent the values that can appear on bus and need to use Verilog with a 4-valued logic:

Value	Meaning
0	Logic zero, "low"
1	Logic one, "high"
Z or ?	High impedance (tri-state buses)
X	Unknown value (simulation)

"X" is used by simulators when a wire hasn't been initialized to a known value or when the predicted value is an illegitimate logic value (e.g., due to contention on a tri-state bus).

Numeric Constants

Constant values can be specified with a specific width and radix:

```
123          // default: decimal radix, unspecified width
'd123        // 'd = decimal radix
'h7B         // 'h = hex radix
'o173        // 'o = octal radix
'b111_1011   // 'b = binary radix, "_" are ignored
'hxx         // can include X, Z or ? in non-decimal constants
16'd5        // 16-bit constant 'b0000_0000_0000_0101
11'h1X?      // 11-bit constant 'b001_XXXX_ZZZZ
```

By default constants are unsigned and will be extended with 0's on left if need be (if high-order bit is X or Z, the extended bits will be X or Z too). You can specify a signed constant as follows:

```
8'shFF      // 8-bit twos-complement representation of -1
```

To be absolutely clear in your intent **it's usually best to explicitly specify the width and radix.**

Logic (SystemVerilog) Wires (Verilog)

We have to provide declarations* for all our named wires (aka "nets"). We can create buses – indexed collections of wires – by specifying the allowable range of indices in the declaration:

```
logic a,b,z;             // three 1-bit wires
logic [31:0] memdata;    // a 32-bit bus
logic [7:0] b1,b2,b3,b4; // four 8-bit buses
logic [W-1:0] input;     // parameterized bus
```

Note that [0:7] and [7:0] are both legitimate but it pays to develop a convention and stick with it. Common usage is [MSB:LSB] where MSB > LSB; usually LSB is 0. Note that we can use an expression in our index declaration but the expression's value must be able to be determined at compile time. We can also build unnamed buses via concatenation:

```
{b1,b2,b3,b4}  // 32-bit bus, b1 is [31:24], b2 is [23:16], …
{4{b1[3:0]},16'h0000}  // 32-bit bus, 4 copies of b1[3:0], 16 0's
```

* Actually by default undeclared identifiers refer to a 1-bit wire, but this means typos get you into trouble. Specify "**`default_nettype none**" at the top of your source files to avoid this bogus behavior.

Verilog Syntax

- Bit selected allowed on a wire but not sum

```
logic [2:0] sum;
 sum = sw[1:0] + sw[3:2];
 assign led_r = sum[1];
```

~~assign led_r = (sw[1:0] + sw[3:2])[2];~~

- Assign not allowed in always block

Gesture Controlled Drone Fall 2014

- Track hands with a camera and determine x,y coordinates
- Based on movement of the coordinates, recognize gestures.
- Generate real time digital signals and convert to analog format for transmission to drone – controlling pitch, roll, hover
- Innovation: using hand motion and recognition of gestures to control flight