EECS 150 - Components and Design Techniques for Digital Systems

Lec 8 – Timing Intro, KMAP, Synthesis

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

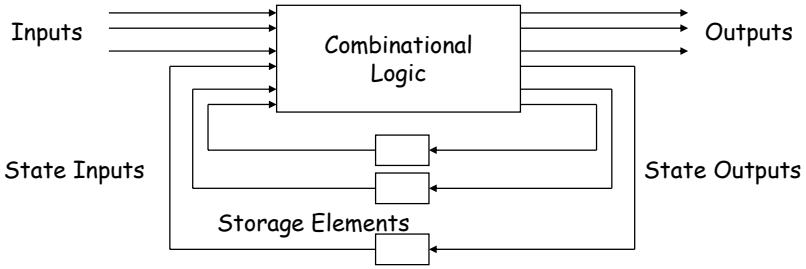
http://www.eecs.berkeley.edu/~culler
http://inst.eecs.berkeley.edu/~cs150

Outline

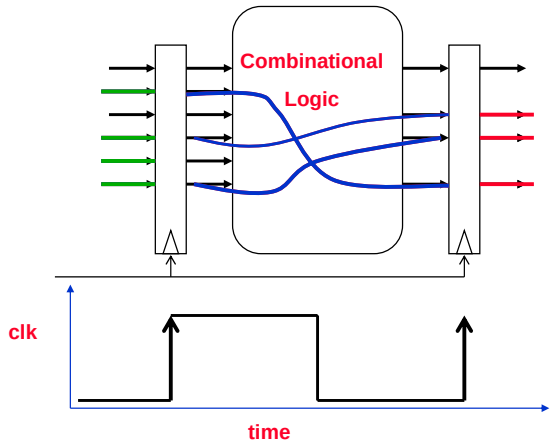
- Timing Methodology for Synchronous Circuits
- Boolean Logic minimization (Kmaps)
- Synthesis – what else the tools do [to the extent time permits]

Review: Fundamental Design Principle

- Divide circuit into combinational logic and state
- Localize feedback loops and make it easy to break cycles
- Implementation of storage elements leads to various forms of sequential logic



Recall: What makes Digital Systems tick?

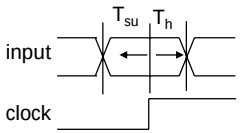


Timing Methodology

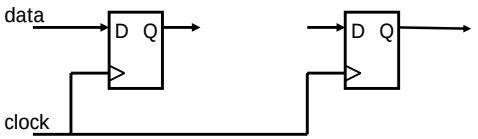
- Rules for interconnecting components and clocks
  - Guarantee proper operation of system when strictly followed
- Approach depends on building blocks used for storage elements
  - Focus on systems with edge-triggered flip-flops
    - » Found in programmable logic devices
  - Many custom integrated circuits focus on level-sensitive latches
- Basic rules for correct timing:
  - (1) Correct inputs, with respect to time, are provided to the flip-flops
    - » Everything is stable when the clock ticks
  - (2) No flip-flop changes state more than once per clocking event

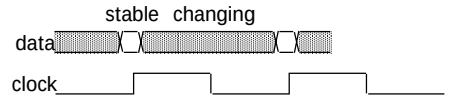
Timing Methodologies (cont'd)

- Definition of terms
  - clock: periodic event, causes state of storage element to change; can be rising or falling edge, or high or low level
  - setup time: minimum time before the clocking event by which the input must be stable (Tsu)
  - hold time: minimum time after the clocking event until which the input must remain stable (Th)

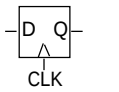


there is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized

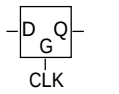




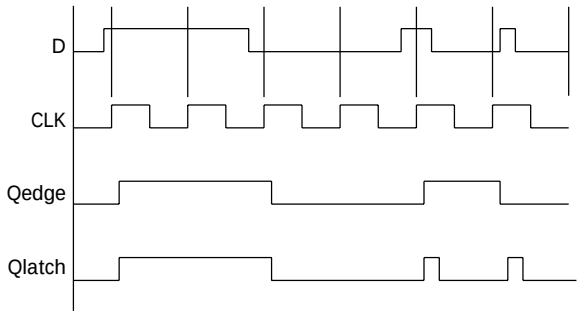
Comparison of Latches and Flip-Flops



positive edge-triggered flip-flop



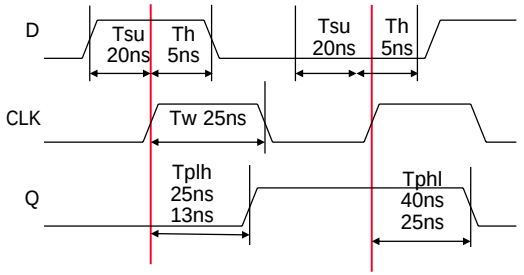
transparent (level-sensitive) latch



behavior is the same unless input changes while the clock is high

Typical Timing Specifications

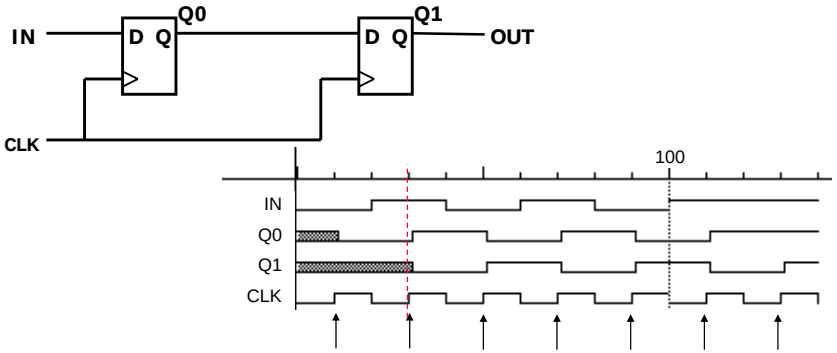
- Positive edge-triggered D flip-flop
  - Setup and hold times
  - Minimum clock width
  - Propagation delays (low to high, high to low, max and typical)



all measurements are made from the clocking event, i.e. the rising edge of the clock

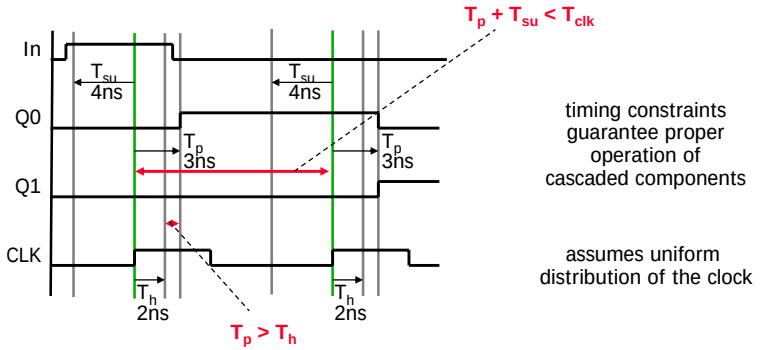
Cascading Edge-triggered Flip-Flops

- **Shift register**
  - New value goes into first stage
  - While previous value of first stage goes into second stage
  - Consider setup/hold/propagation delays (*prop must be > hold*)



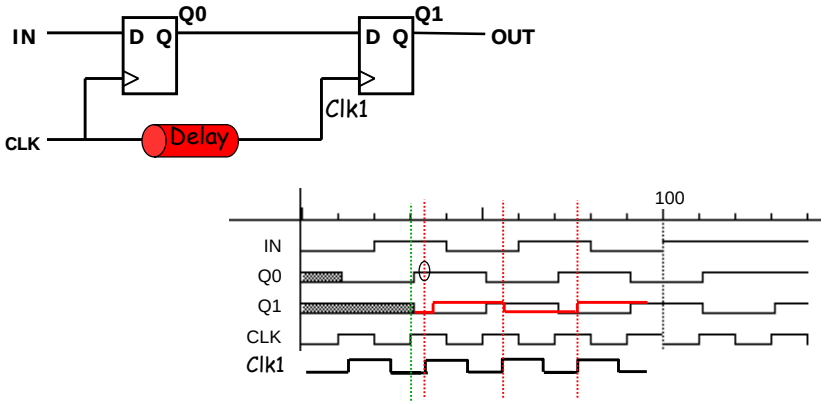
Cascading Edge-triggered Flip-Flops

- **Why this works**
  - Propagation delays exceed hold times
  - Clock width constraint exceeds setup time
  - This guarantees following stage will latch current value before it changes to new value



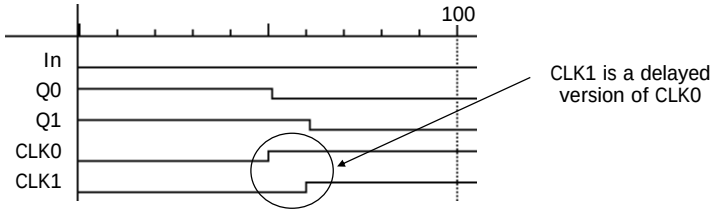
Cascading Edge-triggered Flip-Flops

- **Shift register**
  - New value goes into first stage
  - While previous value of first stage goes into second stage
  - Consider setup/hold/propagation delays (prop must be > hold)



Clock Skew

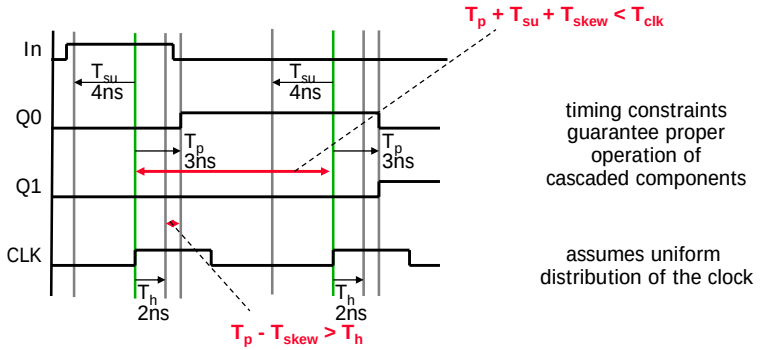
- **The problem**
  - Correct behavior assumes next state of all storage elements determined by all storage elements at the same time
  - Difficult in high-performance systems because time for clock to arrive at flip-flop is comparable to delays through logic (and will soon become greater than logic delay)
  - Effect of skew on cascaded flip-flops:



original state: IN = 0, Q0 = 1, Q1 = 1
due to skew, next state becomes: Q0 = 0, Q1 = 0, and not Q0 = 0, Q1 = 1

Cascading Edge-triggered Flip-Flops

- **Why this works (redux)**
  - **Propagation delays exceed hold times**
  - **Clock width constraint exceeds setup time**
  - **This guarantees following stage will latch current value before it changes to new value**



timing constraints guarantee proper operation of cascaded components

assumes uniform distribution of the clock

Comparison of Latches and Flip-Flops

Type	When inputs are sampled	When output is valid
unclocked latch	always	propagation delay from input change
level-sensitive latch	clock high (Tsu/Th around falling edge of clock)	propagation delay from input change or clock edge (whichever is later)
master-slave flip-flop	clock high (Tsu/Th around falling edge of clock)	propagation delay from falling edge of clock
negative edge-triggered flip-flop	clock hi-to-lo transition (Tsu/Th around falling edge of clock)	propagation delay from falling edge of clock

Summary of Latches and Flip-Flops

- **Development of D-FF**
  - **Level-sensitive used in custom integrated circuits**
    - » **can be made with 4 switches**
  - **Edge-triggered used in programmable logic devices**
  - **Good choice for data storage register**
- **Historically J-K FF was popular but now never used**
  - **Similar to R-S but with 1-1 being used to toggle output (complement state)**
  - **Good in days of TTL/SSI (more complex input function: D = JQ' + K'Q**
  - **Not a good choice for PLAs as it requires two inputs**
  - **Can always be implemented using D-FF**
- **Preset and clear inputs are highly desirable on flip-flops**
  - **Used at start-up or to reset system to a known state**

Logic Minimization

- **One piece of synthesis**

Quick Review: Canonical Forms

- **Standard form for a Boolean expression - *unique* algebraic expression directly from a true table (TT) description.**
- **Two Types:**
  - ***Sum of Products (SOP)***
  - ***Product of Sums (POS)***
- **Sum of Products (disjunctive normal form, minterm expansion).**

Example:

minterms	a b c	f	f'
a'b'c'	0 0 0	0	1
a'b'c	0 0 1	0	1
a'bc'	0 1 0	0	1
a'bc	0 1 1	1	0
ab'c'	1 0 0	1	0
ab'c	1 0 1	1	0
abc'	1 1 0	1	0
abc	1 1 1	1	0

One product (and) term for each 1 in f:

f = a'bc + ab'c' + ab'c +abc' +abc

f' = a'b'c' + a'b'c + a'bc'

Quick Review: Sum of Products (cont.)

Canonical Forms are usually not minimal:

Our Example:

f = a'bc + ab'c' + ab'c + abc' +abc **(xy' + xy = x)**

= a'bc + ab' + ab

= a'bc + a **(x'y + x = y + x)**

= a + bc

f' = a'b'c' + a'b'c + a'bc'

= a'b' + a'bc'

= a' (b' + bc')

= a' (b' + c')

= a'b' + a'c'

Quick Review: Canonical Forms

- **Product of Sums (conjunctive normal form, maxterm expansion).**

Example:

maxterms	a b c	f	f'
a+b+c	0 0 0	0	1
a+b+c'	0 0 1	0	1
a+b'+c	0 1 0	0	1
a+b'+c'	0 1 1	1	0
a'+b+c	1 0 0	1	0
a'+b+c'	1 0 1	1	0
a'+b'+c	1 1 0	1	0
a'+b'+c'	1 1 1	1	0

One sum (**or**) term for each 0 in f:

f = (a+b+c)(a+b+c')(a+b'+c)

f' = (a+b'+c')(a'+b+c)(a'+b+c')

(a'+b'+c)(a+b+c')

Mapping from SOP to POS (or POS to SOP): ***Derive truth table then proceed.***

Incompletely specified functions

- **Example: binary coded decimal increment by 1**
  - **BCD digits encode decimal digits 0 – 9 in bit patterns 0000 – 1001**

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

off-set of W

on-set of W

don't care (DC) set of W

these inputs patterns should never be encountered in practice - "don't care" about associated output values, can be exploited in minimization

Implementing the TT

- Circuit must “cover the 1s” and “none of the 0s”.
- Don’t care can go either way

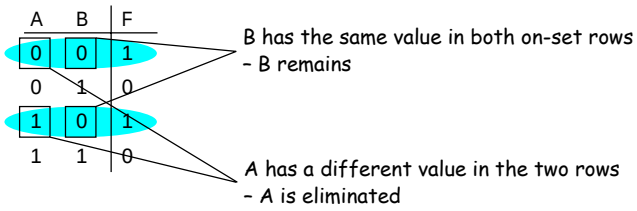
A	B	f
0	0	0
0	1	1
1	0	1
1	1	1

A	B	f
0	0	0
0	1	1
1	1	1
1	0	1

The Uniting Theorem

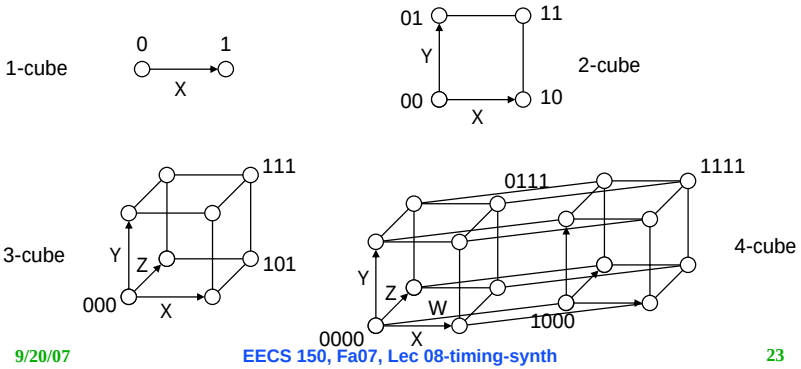
- Key tool to simplification: A (B' + B) = A
- Essence of simplification of two-level logic
  - Find two element subsets of the ON-set where only one variable changes its value – this single varying variable can be eliminated and a single product term used to represent both elements

F = A'B'+AB' = (A'+A)B' = B'



Boolean cubes

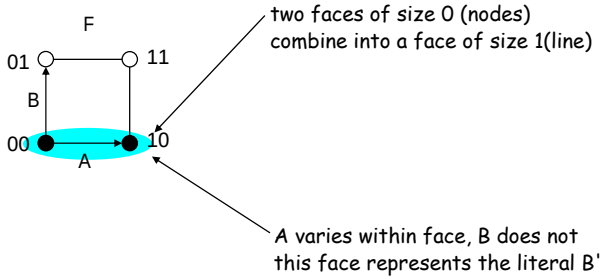
- Visual technique for identifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube“
- Neighbors “address” differs by one bit flip



Mapping truth tables onto Boolean cubes

- Uniting theorem combines two "faces" of a cube into a larger "face"
- Example:

A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

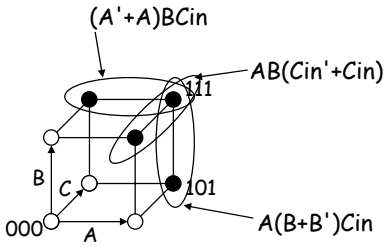


ON-set = solid nodes
OFF-set = empty nodes
DC-set = ×'d nodes

Three variable example

- **Binary full-adder carry-out logic**

A	B	Cin	Cout
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

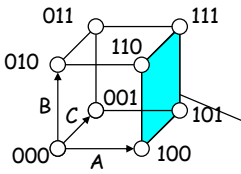


the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that "111" is covered three times

Cout = BCin+AB+ACin

Higher dimensional cubes

- **Sub-cubes of higher dimension than 2**



F(A,B,C) = Σm(4,5,6,7)

on-set forms a square
i.e., a cube of dimension 2

represents an expression in one variable
i.e., 3 dimensions – 2 dimensions

A is asserted (true) and unchanged
B and C vary

This subcube represents the literal A

m-dimensional cubes in a n-dimensional Boolean space

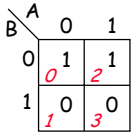
- **In a 3-cube (three variables):**
  - **0-cube, i.e., a single node, yields a term in 3 literals**
  - **1-cube, i.e., a line of two nodes, yields a term in 2 literals**
  - **2-cube, i.e., a plane of four nodes, yields a term in 1 literal**
  - **3-cube, i.e., a cube of eight nodes, yields a constant term "1"**
- **In general,**
  - **m-subcube within an n-cube (m < n) yields a term with n – m literals**

Announcements

- **Typo corrected on HW3, prob. 1**
- **P3, yes there is an input to each controller described in the text that is not shown in the picture.**
- **HW4 out tonight – it is a mid term review**
- **Review session Tues**
- **Mid term next Thurs in 125 Cory**
  - **Everything you want to know at hkn/student/online/cs/150 …**

Karnaugh maps

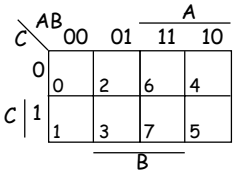
- Flat map of Boolean cube
  - Wrap–around at edges
  - Hard to draw and visualize for more than 4 dimensions
  - Virtually impossible for more than 6 dimensions
- Alternative to truth-tables to help visualize adjacencies
  - Guide to applying the uniting theorem
  - On-set elements with only one variable changing value are adjacent unlike the situation in a linear truth-table

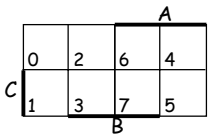


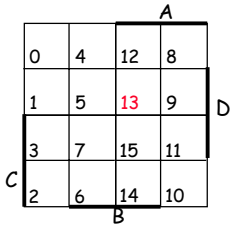
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

Karnaugh maps (cont'd)

- Numbering scheme based on Gray–code
  - e.g., 00, 01, 11, 10
  - 2^n values of n bits where each differs from next by one bit flip
    - » Hamiltonian circuit through n-cube
  - Only a single bit changes in code for adjacent map cells



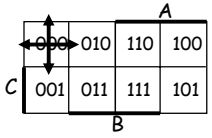


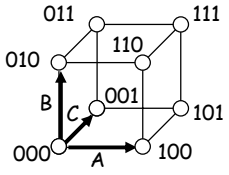


13 = 1101= ABC'D

Adjacencies in Karnaugh maps

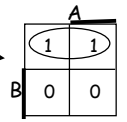
- Wrap from first to last column
- Wrap top row to bottom row



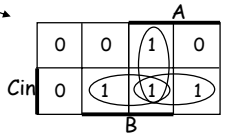


Karnaugh map examples

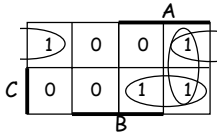
- F =
- Cout =
- f(A,B,C) = Σm(0,4,6,7)



B'



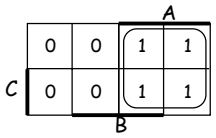
AB + ACin+ BCin



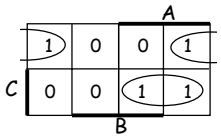
AC+ B'C' + ~~AB'~~

obtain the complement of the function by covering 0s with subcubes

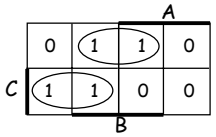
More Karnaugh map examples



G(A,B,C) = A



F(A,B,C) = Σm(0,4,5,7)= AC + B'C'

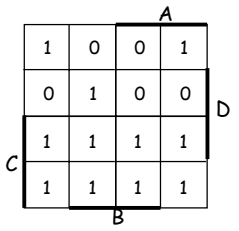


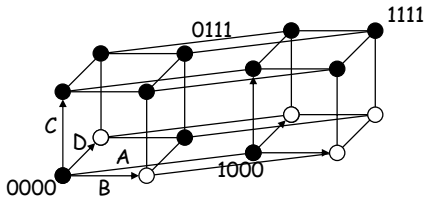
F' simply replace 1's with 0's and vice versa

F'(A,B,C) = Σ m(1,2,3,6) = BC' + A'C

K-map: 4-variable interactive quiz

- **F(A,B,C,D) = Σm(0,2,3,5,6,7,8,10,11,14,15)**
  F =



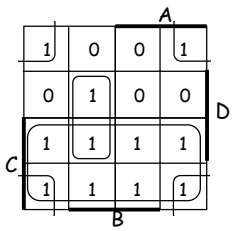


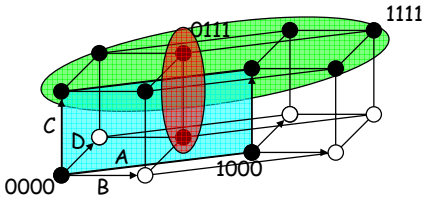
find the smallest number of the largest possible subcubes to cover the ON-set
(fewer terms with fewer inputs per term)

Karnaugh map: 4-variable example

- **F(A,B,C,D) = Σm(0,2,3,5,6,7,8,10,11,14,15)**
  F =

  C + A'BD + B'D'



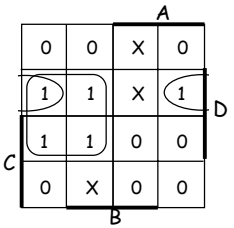


find the smallest number of the largest possible subcubes to cover the ON-set
(fewer terms with fewer inputs per term)

Karnaugh maps: don't cares

- **f(A,B,C,D) = Σ m(1,3,5,7,9) + d(6,12,13)**
  - **without don't cares**
    - » **f =**

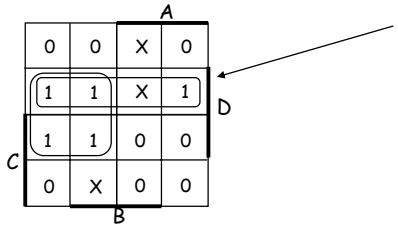
      A'D + B'C'D



Karnaugh maps: don't cares (cont'd)

- **f(A,B,C,D) = Σ m(1,3,5,7,9) + d(6,12,13)**
  - **f = A'D + B'C'D** **without don't cares**
  - **f =** **with don't cares**

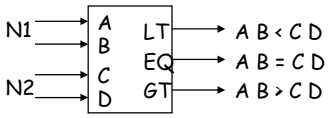
A'D + C'D



by using don't care as a "1"
a 2-cube can be formed
rather than a 1-cube to cover
this node

don't cares can be treated as
1s or 0s
depending on which is more
advantageous

Design example: two-bit comparator

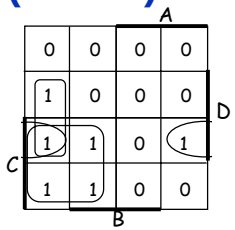


block diagram
and
truth table

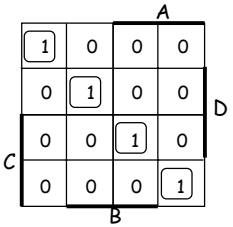
A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

we'll need a 4-variable Karnaugh map
for each of the 3 output functions

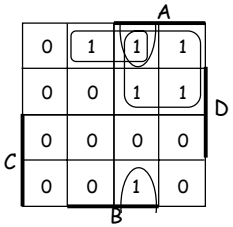
Design example: two-bit comparator (cont'd)



K-map for LT



K-map for EQ



K-map for GT

LT = A' B' D + A' C + B' C D

EQ = A'B'C'D' + A'BC'D + ABCD + AB'CD' = (A xnor C) • (B xnor D)

GT = B C' D' + A C' + A B D'

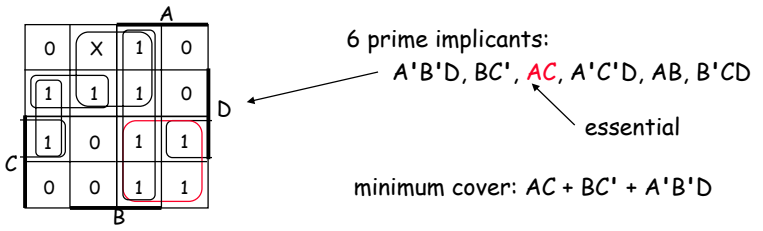
Canonical PofS vs minimal?

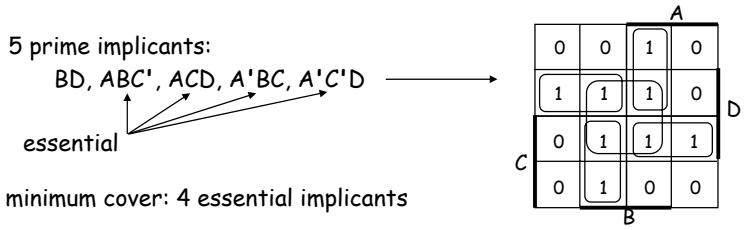
LT and GT are similar (flip A/C and B/D)

Definition of terms for two-level simplification

- **Implicant**
  - **Single element of ON-set or DC-set or any group of these elements that can be combined to form a subcube**
- **Prime implicant**
  - **Implicant that can't be combined with another to form a larger subcube**
- **Essential prime implicant**
  - **Prime implicant is essential if it alone covers an element of ON-set**
  - **Will participate in ALL possible covers of the ON-set**
  - **DC-set used to form prime implicants but not to make implicant essential**
- **Objective:**
  - **Grow implicant into prime implicants (minimize literals per term)**
  - **Cover the ON-set with as few prime implicants as possible (minimize number of product terms)**

Examples to illustrate terms

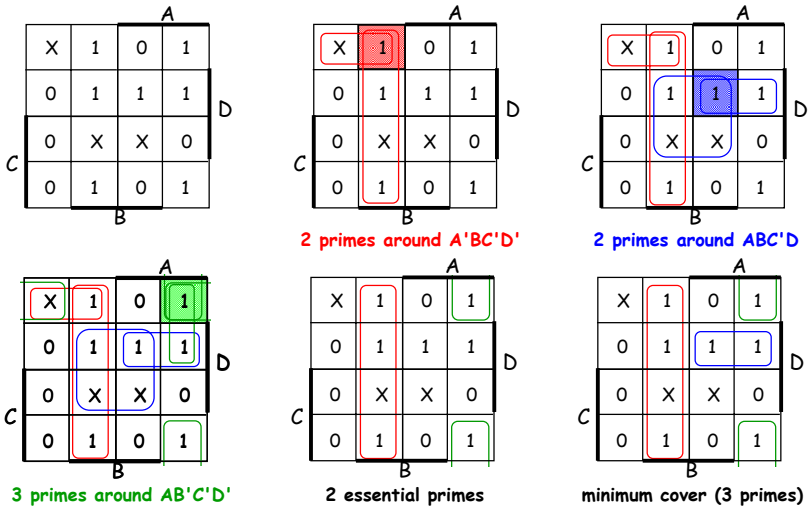




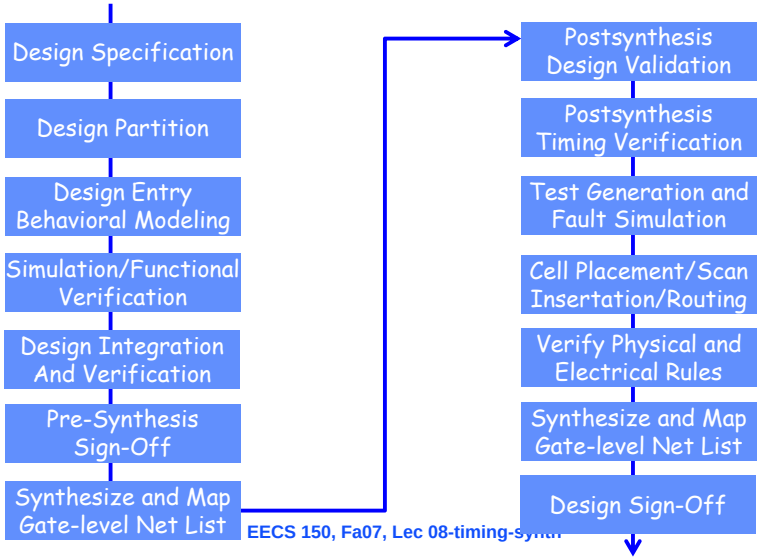
Algorithm for two-level simplification

- **Algorithm: minimum sum-of-products expression from a Karnaugh map**
  - **Step 1: choose an element of the ON-set**
  - **Step 2: find "maximal" groupings of 1s and Xs adjacent to that element**
    - » **consider top/bottom row, left/right column, and corner adjacencies**
    - » **this forms prime implicants (number of elements always a power of 2)**
  - **Repeat Steps 1 and 2 to find all prime implicants**
  - **Step 3: revisit the 1s in the K-map**
    - » **if covered by single prime implicant, it is essential, and participates in final cover**
    - » **1s covered by essential prime implicant do not need to be revisited**
  - **Step 4: if there remain 1s not covered by essential prime implicants**
    - » **select the smallest number of prime implicants that cover the remaining 1s**

Algorithm for two-level simplification (example)



Recall: Design Methodology



Design Specification

- Written statement of functionality, timing, area, power, testability, fault coverage, etc.
- Functional specification methods:
  - State Transition Graphs
  - Timing Charts
  - Algorithm State Machines (like flowcharts)
  - HDLs (Verilog and VHDL)

Design Partition

- Partition to form an Architecture
  - Interacting functional units
    - » Control vs. datapath separation
    - » Interconnection structures within datapath
    - » Structural design descriptions
  - Components described by their behaviors
    - » Register-transfer descriptions
  - Top-down design method exploiting hierarchy and reuse of design effort

Design Entry

- Primary modern method: hardware description language
  - Higher productivity than schematic entry
  - Inherently easy to document
  - Easier to debug and correct
  - Easy to change/extend and hence experiment with alternative architectures
- Synthesis tools map description into generic technology description
  - E.g., logic equations or gates that will subsequently be mapped into detailed target technology
  - Allows this stage to be technology independent (e.g., FPGA LUTs or ASIC standard cell libraries)
- Behavioral descriptions are how it is done in industry today

Simulation and Functional Verification

- Simulation vs. Formal Methods
- Test Plan Development
  - What functions are to be tested and how
  - Testbench Development
    - » Testing of independent modules
    - » Testing of composed modules
  - Test Execution and Model Verification
    - » Errors in design
    - » Errors in description syntax
    - » Ensure that the design can be synthesized
  - The model must be VERIFIED before the design methodology can proceed

Design Integration and Verification

- Integrate and test the individual components that have been independently verified
- Appropriate testbench development and integration
- Extremely important step and one that is often the source of the biggest problems
  - Individual modules thoroughly tested
  - Integration not as carefully tested
  - Bugs lurking in the interface behavior among modules!

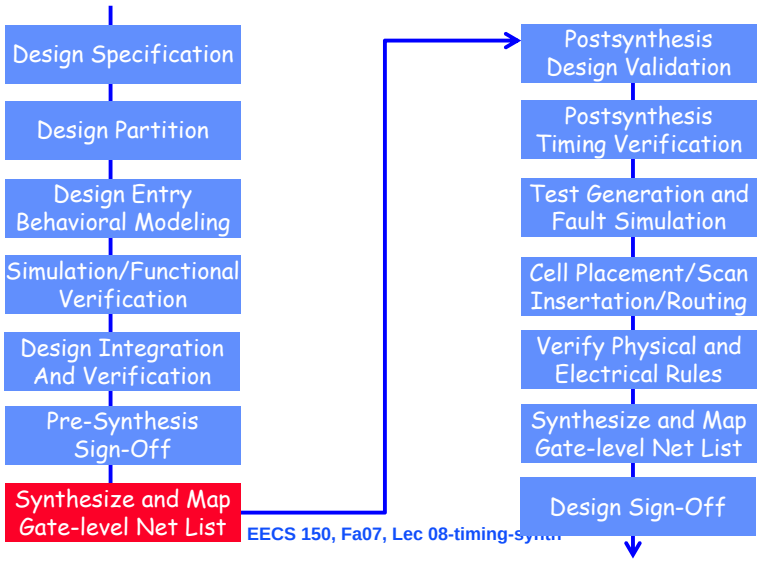
Presynthesis Sign-off

- Demonstrate full functionality of the design
- Make sure that the behavior specification meets the design specification
  - Does the demonstrated input/output behavior of the HDL description represent that which is expected from the original design specification
- Sign-off only when all functional errors have been eliminated

Gate-Level Synthesis and Technology Mapping

- Once all syntax and functional errors have been eliminated, synthesize the design from the behavior description
  - Optimized Boolean description
  - Map onto target technology
- Optimizations include
  - Minimize logic
  - Reduce area
  - Reduce power
  - Balance speed vs. other resources consumed
- Produces netlist of standard cells or database to configure target FPGA

Design Methodology in Detail

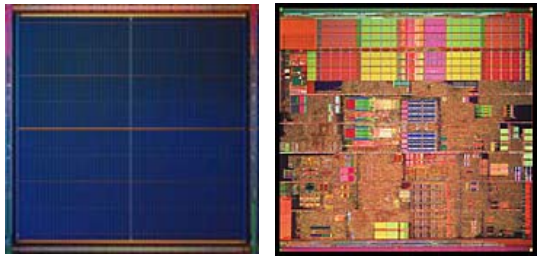


Logic Synthesis

- **Verilog and VHDL started out as simulation languages, but quickly people wrote programs to automatically convert Verilog code into low-level circuit descriptions (netlists).**

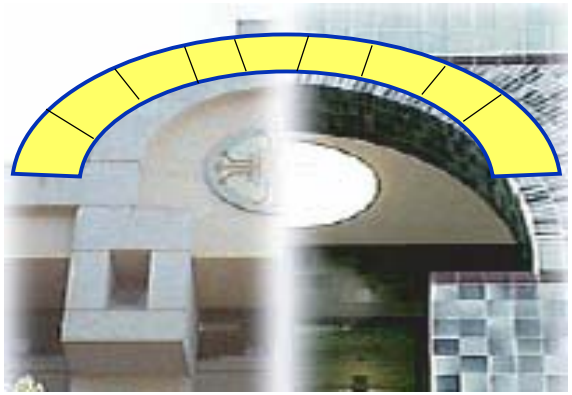
- **Synthesis converts Verilog (or other HDL) descriptions to implementation technology specific primitives:**
  - **For FPGAs: LUTs, flip-flops, and RAM blocks**
  - **For ASICs: standard cell gate and flip-flop libraries, and memory blocks.**

Die Photos: Vertex vs. Pentium IV



- **FGPA Vertex chip looks remarkably structured**
  - **Very dense, very regular structure**
  - **Lots of volume, low NRE, high silicon overhead**
- **Full Custom Pentium chip somewhat more random in structure**
  - **Large on-chip memories (caches) are visible**
- **Logic Synthesis essential for both**

Logic Synthesis – where EE and CS meet



Why Logic Synthesis?

1. **Automatically manages many details of the design process:**
   - ⇒ **Fewer bugs**
   - ⇒ **Improved productivity**
2. **Abstracts the design data (HDL description) from any particular implementation technology.**
   - **Designs can be re-synthesized targeting different chip technologies. Ex: first implement in FPGA then later in ASIC.**
3. **In some cases, leads to a more optimal design than could be achieved by manual means (ex: logic optimization)**

Why Not Logic Synthesis?

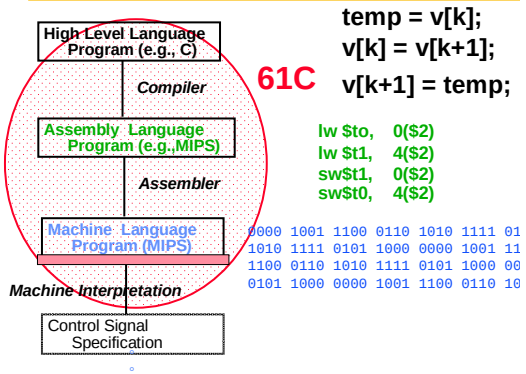
1. May lead to non-optimal designs in some cases.

How does it work?

- **A variety of general and ad-hoc (special case) methods:**
  - ***Instantiation:*** **maintains a library of primitive modules (AND, OR, etc.) and user defined modules.**
  - ***"macro expansion" / substitution:*** **a large set of language operators (+, -, Boolean operators, etc.) and constructs (if-else, case) expand into special circuits.**
  - ***Inference:*** **special patterns are detected in the language description and treated specially (ex: inferring memory blocks from variable declaration and read/write statements, FSM detection and generation from "always @ (posedge clk)" blocks).**
  - ***Logic optimization:*** **Boolean operations are grouped and optimized with logic minimization techniques.**
  - ***Structural reorganization:*** **advanced techniques including sharing of operators, and retiming of circuits (moving FFs), and others?**

Synthesis vs Compilation

Levels of Representation

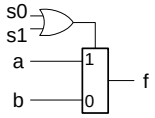


- **Compiler**
  - **recognizes all possible constructs in a formally defined program language**
  - **translates them to a machine language representation of execution process**
- **Synthesis**
  - **Recognizes a target dependent subset of a hardware description language**
  - **Maps to collection of concrete hardware resources**
  - **Iterative tool in the design flow**

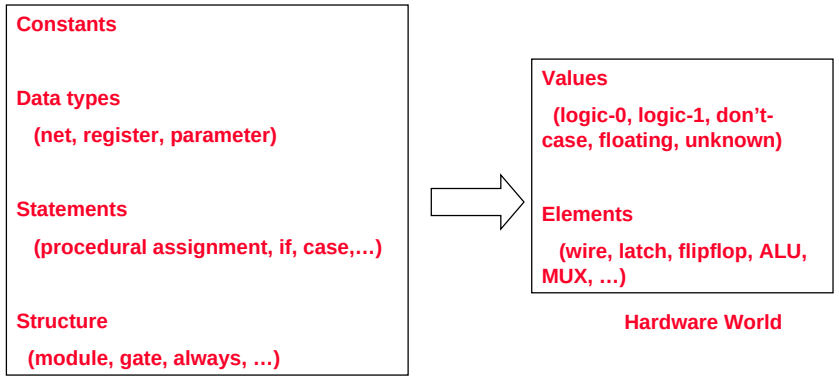
Simple Example

```
module foo (a,b,s0,s1,f);
input [3:0] a;
input [3:0] b;
input s0,s1;
output [3:0] f;
reg f;
always @ (a or b or s0 or s1)
  if ('s0 && s1 || s0) f=a; else f=b;
endmodule
```

- **Should expand if-else into 4-bit wide multiplexor and optimize the control logic:**



Mapping



Module Template

Synthesis tools expects to find modules in this format.

```
module <top_module_name>(<port list>);
/* Port declarations. followed by wire, reg, integer, task and function declarations */
/* Describe hardware with one or more continuous assignments, always blocks,  module
    instantiations and gate instantiations */
// Continuous assignment
wire <result_signal_name>;
assign <result_signal_name> = <expression>;
// always block
always @(<event expression>)
begin
// Procedural assignments
// if statements
// case, casex, and casez statements
// while, repeat and for loops
// user task and user function calls
end
// Module instantiation
<module_name> <instance_name> (<port list>);
// Instantiation of built-in gate primitive
gate_type_keyword (<port list>);
endmodule
```

- The order of these statements is irrelevant, all execute concurrently.
- The statements between the **begin** and **end** in an always block execute sequentially from top to bottom. (However, beware of blocking versus non-blocking assignment)
- Statements within a fork-join statement in an always block execute concurrently.

Supported Verilog Constructs

- **Net types:**
  - wire, tri, supply1, supply0;
  - register types: reg, integer, time (64 bit reg); arrays of reg.
- **Continuous assignments.**
- **Gate primitive and module instantiations.**
- **always blocks, user tasks, user functions.**
- **inputs, outputs, and inouts to a module.**
- **All operators**
  - +, -, *, /, %, <, >, <=, >=, ==, !=, ===, !==, &&, ||, !, ~, &, ~&, |, ~|, ^~, ~^, ^, <<, >>, ?:, { }, {{ }}}
  - Note: / and % are supported for compile-time constants and constant powers of 2.
- **Procedural statements:**
  - if-else-if, case, casex, casez, for, repeat, while, forever, begin, end, fork, join.
- **Procedural assignments:**
  - blocking assignments =,
  - nonblocking assignments <=
  - Note: <= cannot be mixed with = for the same register.
- **Compiler directives: `define, `ifdef, `else, `endif, `include, `undef**
- **Miscellaneous:**
  - Integer ranges and parameter ranges.
  - Local declarations to begin-end block.
  - Variable indexing of bit vectors on the left and right sides of assignments.

Unsupported Language Constructs

Generate error and halt synthesis

- **Net types: trireg, wor, trior, wand, triand, tri0, tri1, and charge strength;**
- **register type: real.**
- **Built-in unidirectional and bidirectional switches, and pull-up, pull-down.**
- **Procedural statements: assign (different from the "continuous assignment"), deassign, wait.**
- **Named events and event triggers.**
- **UDPs (user defined primitives) and specify blocks.**
- **force, release, and hierarchical net names (for simulation only).**

Simply ignored

- **delay, delay control, and drive strength.**
- **scalared, vectored.**
- **initial block.**
- **Compiler directives (except for `define, `ifdef, `else, `endif, `include, and `undef, which are supported).**
- **Calls to system tasks and system functions (they are only for simulation).**

Net Data Type

- **Variable of NET type maps into a wire**
- **wire ➔ wire**
- **supply0 ➔ wire connected to logic-0**
- **supply1 ➔ wire connected to logic-1**
- **tri ➔like a wire**
- **wor**
- **wand**

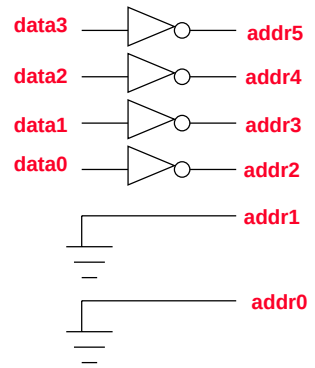
Register Data Type

- Reg declaration specifies size in bits
- Integer type – max size is 32 bits, synthesis may determine size by analysis
  - Wire [1:5] Brq, Rbu
  - Integer Arb
  - …
  - Arb = Brq + Rbu "Arb is 6 bits"
- Variable of reg type maps into wire, latch or flip-flop *depending on context*

Operators

- Logical operators map ino primitive logic gates
- Arithmetic operators map into adders, subtractors, …
  - Unsigned 2s complement
  - Model carry: target is one-bit wider that source
  - Watch out for *, %, and /
- Relational operators generate comparators
- Shifts by constant amount are just wire connections
  - No logic invoved
- Variable shift amounts a whole different story --- shifter
- Conditional expression generates logic or MUX

addr = ~data << 2



Procedural Assignments

- Verilog has two types of assignments within always blocks:
- Blocking **procedural assignment "="**
  - The RHS is executed and the assignment is completed before the next statement is executed. Example:

    Assume A holds the value 1 … A=2; B=A; A is left with 2, B with 2.
- Non-blocking **procedural assignment "<="**
  - The RHS is executed and assignment takes place at the end of the current time step (not clock cycle). Example:

    Assume A holds the value 1 … A<=2; B<=A; A is left with 2, B with 1.
- The notion of the "current time step" is tricky in synthesis, so to guarantee that your simulation matches the behavior of the synthesized circuit, follow these rules:

i. Use blocking assignments to model combinational logic within an always block.
ii. Use non-blocking assignments to implement sequential logic.
iii. Do not mix blocking and non-blocking assignments in the same always block.
iv. Do not make assignments to the same variable from more than one always block.

Combinational Logic

CL can be generated using:

1. **primitive gate instantiation:**
   AND, OR, **etc.**
2. **continuous assignment (assign keyword), example:**

```
Module adder_8 (cout, sum, a, b, cin);
output cout;
output [7:0] sum;
input cin;
input [7:0] a, b;
assign {cout, sum} = a + b + cin;
endmodule
```

3. **Always block:**

```
always @ (event_expression)
begin
  // procedural assignment statements, if statements,
  // case statements, while, repeat, and for loops.
  // Task and function calls
end
```

Combinational logic always blocks

- **Make sure all signals assigned in a combinational** always **block are explicitly assigned values *every* time that the always block executes. Otherwise latches will be generated to hold the last value for the signals not assigned values.**

- Example:
  - Sel case value 2'd2 omitted.
  - Out is not updated when select line has 2'd2.
  - Latch is added by tool to hold the last value of out under this condition.

```
module mux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;
always @(sel or a or b or c or d)
begin
  case (sel)
    2'd0: out = a;
    2'd1: out = b;
    2'd3: out = d;
  endcase
end
endmodule
```


Fixes to the avoid creating latch

```
module mux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;
always @(sel or a or b or c or d)
begin
  case (sel)
    2'd0: out = a;
    2'd1: out = b;
    2'd2:   out = c;
    2'd3: out = d;
  endcase
end
endmodule
```

- **add the missing select line**
- **Or, in general, use the "default" case:**

```
default:  out = foo;
```


Example (cont)

```
module funnymux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;
always @(sel or a or b or c or d)
begin
  case (sel)
    2'd0: out = a;
    2'd1: out = b;
    2'd3: out = d;
  default:  out = 1'bx;
  endcase
end
endmodule
```

- **If you don't care about the assignment in a case (for instance you know that it will never come up) then assign the value "x" to the variable.**
- **The x is treated as a "don't care" for synthesis and will simplify the logic.**

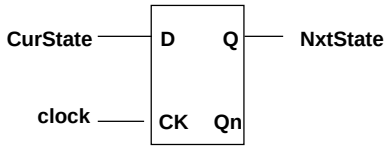
  (The synthesis directive "full_case" will accomplish the same, but can lead to differences between simulation and synthesis.)

Latch rule

- **If a variable is not assigned in all possible executions of an always statement then a latch is inferred**
  - **E.g., when not assigned in all branches of an `if` or `case`**
  - **Even a variable declared locally within an always is inferred as a latch if incompletely assigned in a conditional statement**

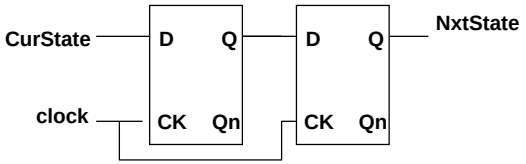
Assign before use ordering

```
module onelatch (clock, curState, nxtState);
 input clock;
 input curState;
 output nxtState;
 reg nxtState

 always @ (Clock or CurrentState)
 begin: L1
  integer temp
  if (clock) begin
   temp = CurState;
   NxtState = temp;
  end
 end
end module
```



Use before Assign ordering

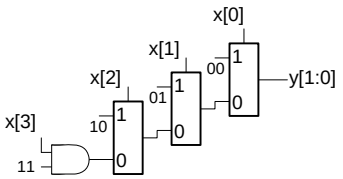
```
module twolatch (clock, curState, nxtState);
 input clock;
 input curState;
 output nxtState;
 reg nxtState

 always @ (Clock or CurrentState)
 begin: L1
  integer temp
  if (clock) begin
      NxtState = temp;
      temp = CurState;
  end
 end
end module
```



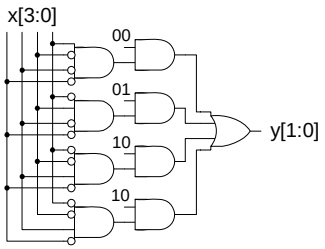
Combinational Logic (cont.)

- Be careful with nested IF-ELSE. They can lead to "priority logic"
  - Example: 4-to-2 encoder

```
always @(x)
begin : encode
if (x[0] == 1'b1) y = 2'b00;
else if (x[1] == 1'b1) y = 2'b01;
else if (x[2] == 1'b1) y = 2'b10;
else if (x[3] == 1'b1) y = 2'b11;
else y = 2'bxx;
end
```




```
always @(x)
begin : encode
case (x)
4'b0001: y = 2'b00;
4'b0010: y = 2'b01;
4'b0100: y = 2'b10;
4'b1000: y = 2'b11;
default: y = 2'bxx;
endcase
end
```



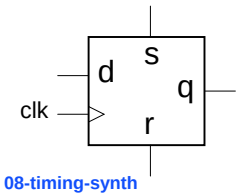
Sequential Logic

- Example: D flip-flop with synchronous set/reset:

```
module dff(q, d, clk, set, rst);
input d, clk, set, rst;
output q;
reg q;
always @(posedge clk)
if (reset)
   q <= 0;
else if (set) begin
   q <= 1;
else begin
   q <= d;
end
endmodule
```

- "@ (posedge clk)" key to flip-flop generation.
- Note in this case, priority logic is appropriate.
- For Xilinx Virtex FPGAs, the tool infers a native flip-flop (no extra logic is needed for the set/reset.

We prefer *synchronous* set/reset, but how would you specify *asynchronous* preset/clear?



Procedural Assignment

- **Target of proc. Assignment is synthesized into a wire, a flip-flop or a latch, depending on the context under which the assignment appears.**
- **A target cannot be assigned using a blocking assignment and a non-blocking assignment.**

Finite State Machines

```
module FSM1(clk,rst, enable, data_in, data_out);
input clk, rst, enable;
input [2:0] data_in;
output data_out;

/* Defined state encoding;
this style preferred over 'defines*/
parameter default=2'bxx;
parameter idle=2'b00;
parameter read=2'b01;
parameter write=2'b10;
reg data_out;
reg [1:0] state, next_state;

/* always block for sequential logic*/
always @(posedge clk)
       if (!rst) state <= idle;
       else state <= next_state;
```

- Style guidelines (some of these are to get the right result, and some just for readability)
  - Must have reset.
  - Use separate always blocks for sequential and combination logic parts.
  - Represent states with defined labels or enumerated types.

FSMs (cont.)

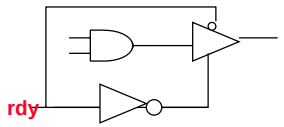
```
/* always block for CL */
always @(state or enable or data_in)
begin
case (state)
/* For each state def output and next*/
  idle : begin
           data_out = 1'b0;
           if (enable && data_in)
             next_state = read;
           else next_state = idle;
         end
  read : begin … end
  write : begin … end

  default : begin
             next_state = default;
             data_out = 1'bx;
           end
endcase
end
endmodule
```

- Use a CASE statement in an always to implement next state and output logic.
- Always use a default case and asset the state variable and output to 'bx:
  - avoids implied latches,
  - allows the use of don't cares leading to simplified logic.
- The "FSM compiler" within the synthesis tool can re-encode your states. This process is controlled by using a synthesis attribute (passed in a comment).
  - See the Synplify guide for details.

Values x and z

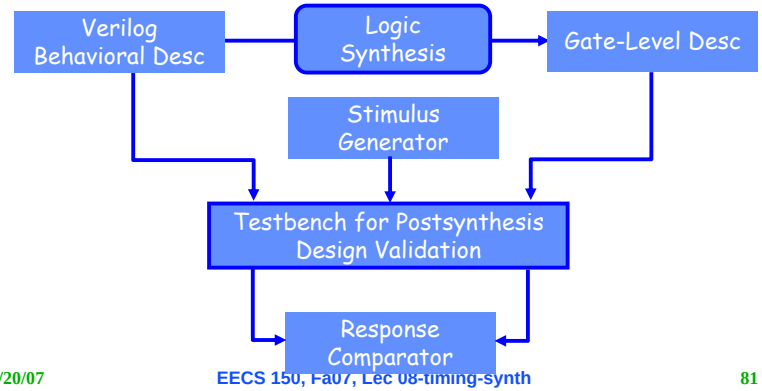
- **Assigning the value x to a variable tells synthesis to treat as dont-care**
- **Assigning z generates tristate gate**
  - **Z can be assigned to any variable in an assignment, but for synthesis tis must occur under the control of a conditional statement**

```
module threestate(rdy, inA, inB, sel)
  input rdy, inA, inB;
  output sel;
  reg sel;
always @(rdy or inA or inB)
  if (rdy) sel = 1'bz
  else sel = inA & inB
endmodule
```



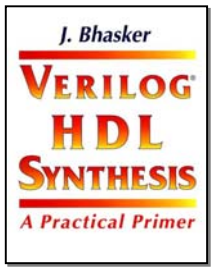
Postsynthesis Design Validation

- Does gate-level synthesized logic implement the same input-output function as the HDL behavioral description?



More Help

- Online documentation for Synplify Synthesis Tool:
  - Under "refs/links" and linked to today's lecture on calendar
  - Online examples from Synplicity.
- Bhasker (same author as Verilog reference book)
- Trial and error with the synthesis tool.
  - Synplify will display the output of synthesis in schematic form for your inspection. Try different input and see what it produces.



Bottom line

- Have the hardware design clear in your mind when you write the verilog.
- Write the verilog to describe that HW
  - it is a Hardware ***Description*** Language not a Hardware Imagination Language.
- If you are very clear, the synthesis tools are likely to figure it out.

Summary

- Timing methodology defines a set of constraints that make life simpler – as long as they are observed
- Boolean Algebra provides framework for logic simplification
- Uniting to reduce minterms
- Karnaugh maps provide visual notion of simplifications
- Algorithm for producing reduced form.
- Synthesis is part algorithms and part pattern matching
  - Work in partnership with your tool
  - Learn the idioms that it does well. Create building blocks out of them. Use those.
  - Think Hardware write code that describes it.