CSC 322: Computer Organization Lab

Lecture 3: Logic Design

Dr. Haidar M. Harmanani

CSC 322: Computer Organization Lab

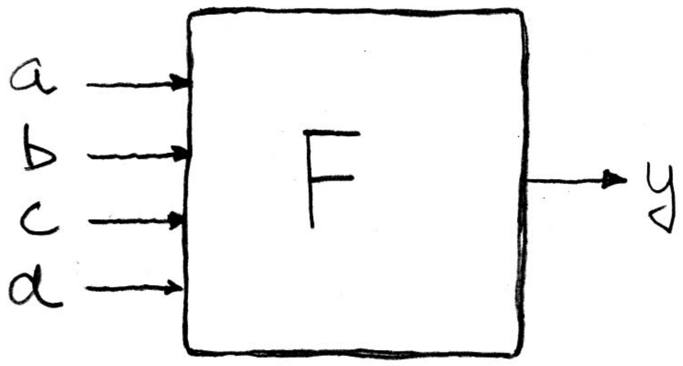
Part I: Combinational Logic

Dr. Haidar M. Harmanani

Logical Design of Digital Systems

- Complex Combinational and Sequential networks (up to thousands of gates)
  - Emphasis on combined datapath + Finite state machine designs for real time applications
- Modern CAD tool usage (schematic entry, simulation, technology mapping, timing analysis, synthesis)
- Logic Synthesis via Verilog
- Modern implementation technologies such as Field Programmable Gate Arrays (FPGAs)

Truth Tables



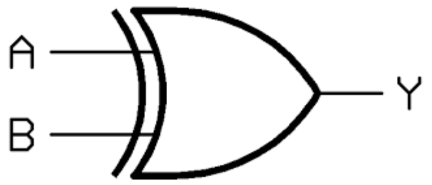
How many Fs (4-input devices)?

0

a	b	c	d	y
0	0	0	0	F(0,0,0,0)
0	0	0	1	F(0,0,0,1)
0	0	1	0	F(0,0,1,0)
0	0	1	1	F(0,0,1,1)
0	1	0	0	F(0,1,0,0)
0	1	0	1	F(0,1,0,1)
0	1	1	0	F(0,1,1,0)
1	1	1	1	F(0,1,1,1)
1	0	0	0	F(1,0,0,0)
1	0	0	1	F(1,0,0,1)
1	0	1	0	F(1,0,1,0)
1	0	1	1	F(1,0,1,1)
1	1	0	0	F(1,1,0,0)
1	1	0	1	F(1,1,0,1)
1	1	1	0	F(1,1,1,0)
1	1	1	1	F(1,1,1,1)

Example #1: 1 iff one (not both) a, b=1

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0



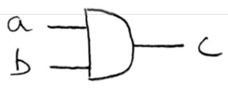
How about a 3-input XOR Gate?

- Easy. Extend the truth table
- How about N-input XOR is the only one which isn't so obvious
  - It's simple: XOR is a 1 iff the # of 1s at its input is odd

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

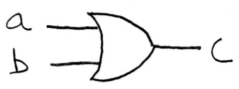
Logic Gates (1/2)

AND



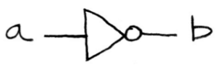
ab	c
00	0
01	0
10	0
11	1

OR



ab	c
00	0
01	1
10	1
11	1

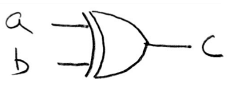
NOT



a	b
0	1
1	0

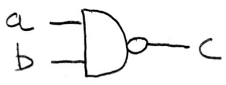
Logic Gates (2/2)

XOR



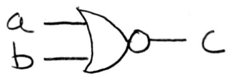
ab	c
00	0
01	1
10	1
11	0

NAND



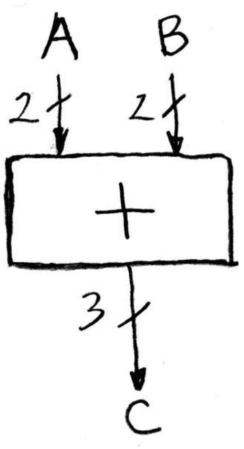
ab	c
00	1
01	1
10	1
11	0

NOR



ab	c
00	1
01	0
10	0
11	0

Example 2: Design a 2-bit Unsigned Adder



How many rows in the truth table?

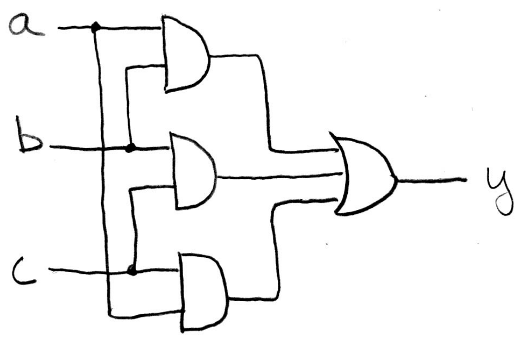
A	B	C
a_1a_0	b_1b_0	$c_2c_1c_0$
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

Example 3: Design of a 32-bit adder

A	B	C
000 ... 0	000 ... 0	000 ... 00
000 ... 0	000 ... 1	000 ... 01
.	.	.
.	.	.
.	.	.
111 ... 1	111 ... 1	111 ... 10

How Many Rows?

Example 5: Majority Function

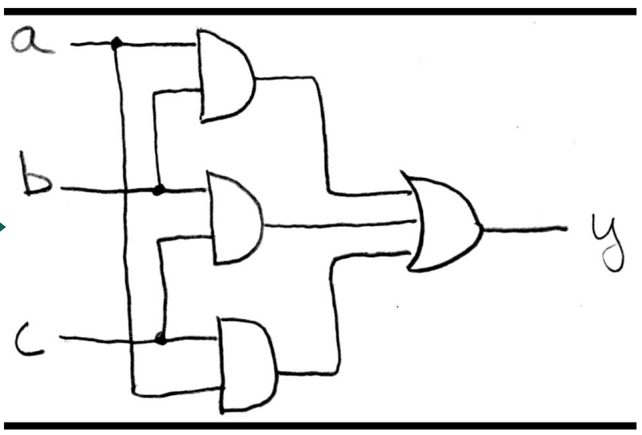


$y = a \cdot b + a \cdot c + b \cdot c$

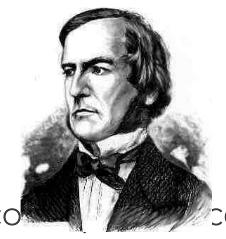
$y = ab + ac + bc$

Example 4: 3-input majority circuit

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



Boolean Algebra



- George Boole, 19th Century mathematician
- Developed an *algebra* involving logic
  - later known as “Boolean Algebra”
- Primitive functions: *AND*, *OR* and *NOT*
- The power f *Boolean Algebra* is there’s a one-to-one co ce between circuits made up of AND, OR and NOT gates and equations.

Boolean algebra

- Boolean algebra
  - B = {0, 1}
  - + is logical OR, • is logical AND
  - ' is logical NOT
- All algebraic axioms hold

An Algebraic Structure

- An algebraic structure consists of
  - a set of elements B
  - binary operations { + , • }
  - and a unary operation { ' }
  - such that the following axioms hold:
    1. the set B contains at least two elements, a, b, such that a ° b
    2. closure: a + b is in B a • b is in B
    3. commutativity: a + b = b + a a • b = b • a
    4. associativity: a + (b + c) = (a + b) + ca • (b • c) = (a • b) • c
    5. identity: a + 0 = a a • 1 = a
    6. distributivity: a + (b • c) = (a + b) • (a + c) a • (b + c) = (a • b) + (a • c)
    7. complementarity: a + a' = 1 a • a' = 0

Axioms and Theorems of Boolean Algebra

- Identity
  1. X + 0 = X 1D. X • 1 = X
- Null
  2. X + 1 = 1 2D. X • 0 = 0
- Idempotency:
  3. X + X = X 3D. X • X = X
- Involution:
  4. (X')' = X
- Complementarity:
  5. X + X' = 1 5D. X • X' = 0
- Commutativity:
  6. X + Y = Y + X 6D. X • Y = Y • X
- Associativity:
  7. (X + Y) + Z = X + (Y + Z) 7D. (X • Y) • Z = X • (Y • Z)

Axioms and Theorems of Boolean Algebra (cont'd)

- Distributivity:
  - 8. X • (Y + Z) = (X • Y) + (X • Z) 8D. X + (Y • Z) = (X + Y) • (X + Z)
- uniting:
  - 9. X • Y + X • Y' = X 9D. (X + Y) • (X + Y') = X
- absorption:
  - 10. X + X • Y = X 10D. X • (X + Y) = X
  - 11. (X + Y') • Y = X • Y 11D. (X • Y') + Y = X + Y
- factoring:
  - 12. (X + Y) • (X' + Z) = X • Z + X' • Y 16D. X • Y + X' • Z = (X + Z) • (X' + Y)
- consensus:
  - 13. (X • Y) + (Y • Z) + (X' • Z) = X • Y + X' • Z 17D. (X + Y) • (Y + Z) • (X' + Z) = (X + Y) • (X' + Z)

Axioms and Theorems of Boolean Algebra (cont'd)

- de Morgan's:
  - 14. (X + Y + ...)' = X' • Y' • ... 12D. (X • Y • ...)' = X' + Y' + ...
- generalized de Morgan's:
  - 15. f'(X1,X2,...,Xn,0,1,+,•) = f(X1',X2',...,Xn',1,0,•,+)
- Establishes relationship between • and +

Axioms and theorems of Boolean algebra (cont')

- Duality
  - a dual of a Boolean expression is derived by replacing
    • by +, + by •, 0 by 1, and 1 by 0, and leaving variables unchanged
  - any theorem that can be proven is thus also proven for its dual!
  - a meta-theorem (a theorem about theorems)
- duality:
  16. $X + Y + \ldots \Leftrightarrow X \bullet Y \bullet \ldots$
- generalized duality:
  17. $f(X1,X2,\ldots,Xn,0,1,+,\bullet) \Leftrightarrow f(X1,X2,\ldots,Xn,1,0,\bullet,+)$
- Different than deMorgan's Law
  - this is a statement about theorems
  - this is not a way to manipulate (re-write) expressions

Logic functions and Boolean algebra

- Any logic function that can be expressed as a truth table can be written as an expression in Boolean algebra using the operators: ', +, and •

X	Y	X • Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X'	X' • Y
0	0	1	0
0	1	1	1
1	0	0	0
1	1	0	0

X	Y	X'	Y'	X • Y	X' • Y'	(X • Y) + (X' • Y')
0	0	1	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	1	0	0	1	0	1

$(X \bullet Y) + (X' \bullet Y') \equiv X = Y$

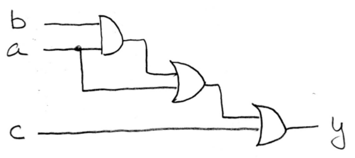
Boolean expression that is true when the variables X and Y have the same value and false, otherwise

X, Y are Boolean algebra variables

Logic functions and Boolean algebra

- Thus, in order to implement an arbitrary logic function, the following procedure can be followed:
  - Derive the truth table
  - Create the product term that has a value of 1 for each valuation for which the output function f has to be 1
  - Take the logical sum of these product terms to realize f

Example 5: Algebraic Simplification



original circuit

$$\downarrow$$

$$y = ((ab) + a) + c$$

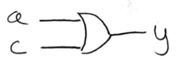
equation derived from original circuit

$$\downarrow$$

$$= ab + a + c$$
$$= a(b+1) + c$$
$$= a(1) + c$$
$$= a + c$$

algebraic simplification

$$\downarrow$$



simplified circuit

Boolean Algebra also great for circuit verification Circ X = Circ Y?
use *Boolean Algebra* to prove!

Example 6: Boolean Algebraic Simplification

$$
\begin{aligned}
y &= ab + a + c \\
&= a(b+1) + c && \textit{distribution, identity} \\
&= a(1) + c && \textit{law of 1's} \\
&= a + c && \textit{identity}
\end{aligned}
$$

Canonical forms (1/2)

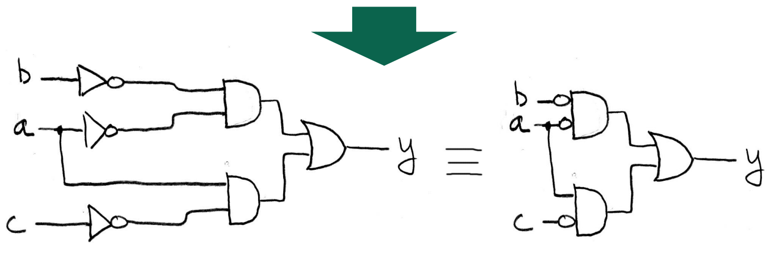
	abc	y
$\overline{a} \cdot \overline{b} \cdot \overline{c}$	000	1
$\overline{a} \cdot \overline{b} \cdot c$	001	1
	010	0
	011	0
$a \cdot \overline{b} \cdot \overline{c}$	100	1
	101	0
$a \cdot b \cdot \overline{c}$	110	1
	111	0

$\Rightarrow$ $y =$

Sum-of-products (ORs of ANDs)

Canonical forms (2/2)

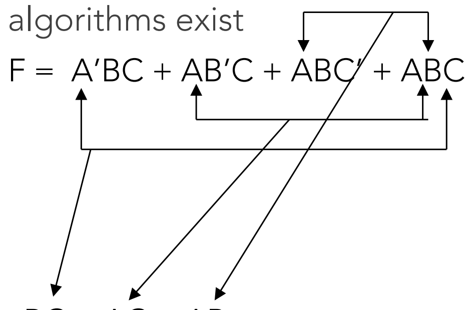
$$\begin{aligned} y &= \overline{a}\overline{b}\overline{c} + \overline{a}\overline{b}c + a\overline{b}\overline{c} + ab\overline{c} & \\ &= \overline{a}\overline{b}(\overline{c} + c) + a\overline{c}(\overline{b} + b) & \textit{distribution} \\ &= \overline{a}\overline{b}(1) + a\overline{c}(1) & \textit{complementarity} \\ &= \overline{a}\overline{b} + a\overline{c} & \textit{identity} \end{aligned}$$



Boolean Minimization

- Reduce a Boolean equation to fewer terms - hopefully, this will result in using less gates to implement the Boolean equation.
- Pencil-Paper: Algebraic techniques, K-maps or
- Automated: Many powerful algorithms exist

A	B	C	F
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



F = A'BC + AB'C + ABC' + ABC

F = BC + AC + AB

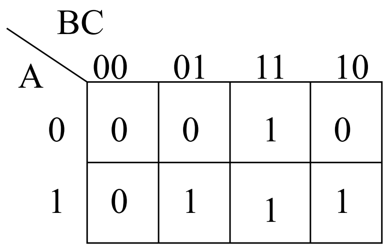
Find Boolean adjacencies to minimize equation; eliminate redundant term

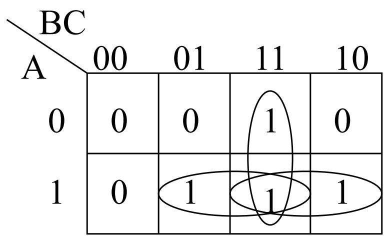
K- maps

Graphical Aid for minimization - used to visualize Boolean adjacencies

A	B	C	F
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

F = BC + AC + AB



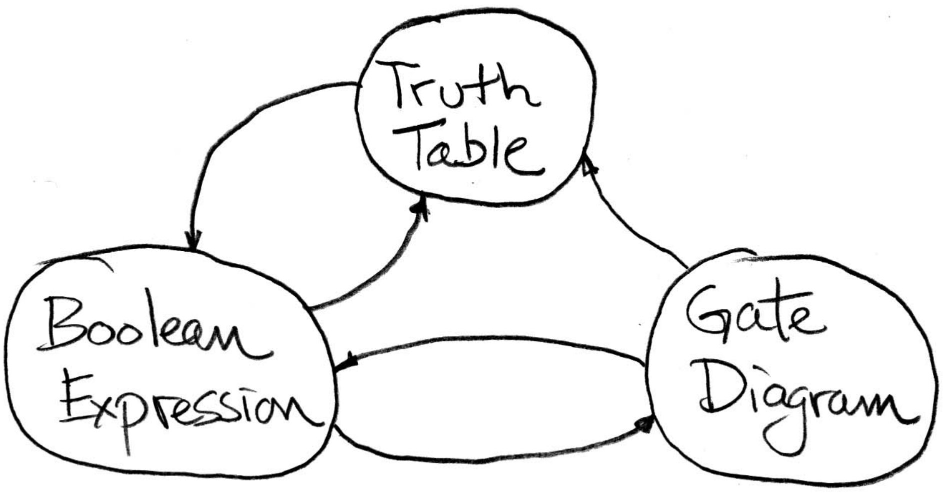


CSC 322: Computer Organization Lab

Part II: Combinatial Blocks

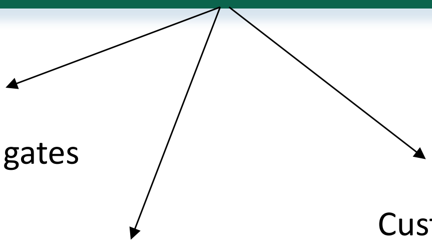
Dr. Haidar M. Harmanani

Recap



Technology Mapping

Technology mapping maps a Boolean equation onto a given technology. The technology can affect what constraints are used when doing minimization for the function.



Discrete gates

Programmable Logic

Custom Integrated Circuits

Logic Synthesis

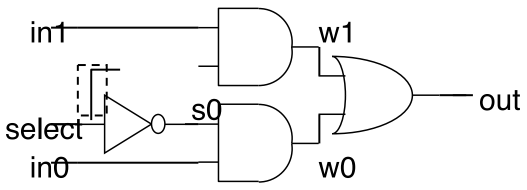
Logic Synthesis is the transformation a digital system described, at the logic level, in a Hardware Description Language (HDL) onto an implementation technology.

Verilog Description

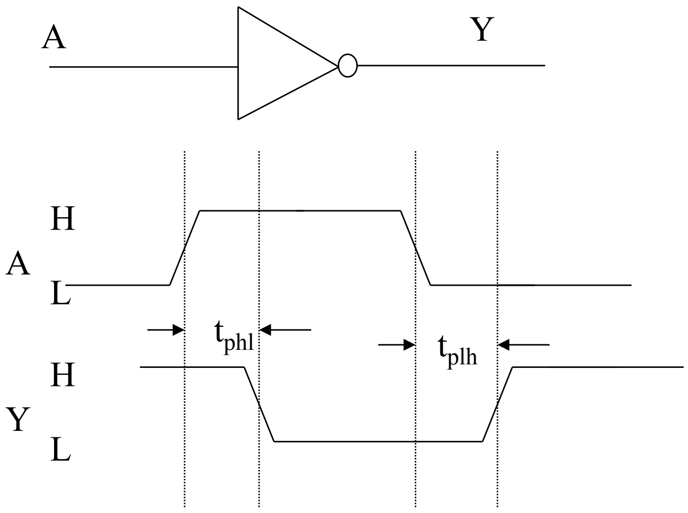
```
//2-input multiplexor in gates
module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;
   wire s0,w0,w1;

   not  (s0, select);
   and  (w0, s0, in0),
        (w1, select, in1);
   or   (out, w0, w1);

endmodule // mux2
```

Synthesis

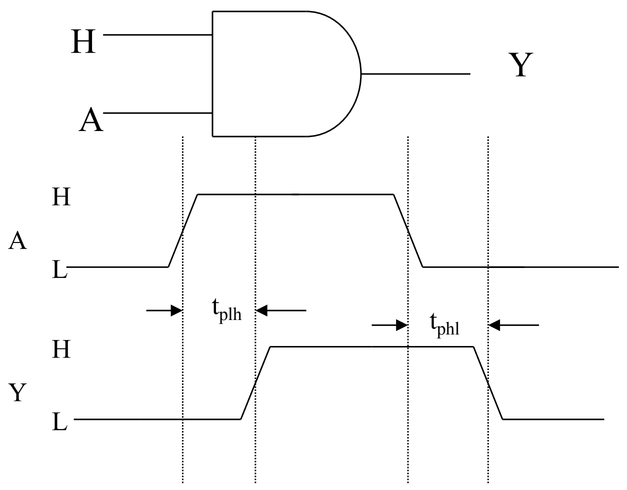
Gates



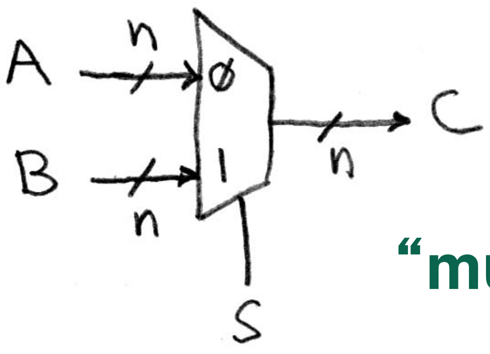
Propagation Delay



Propagation Delay

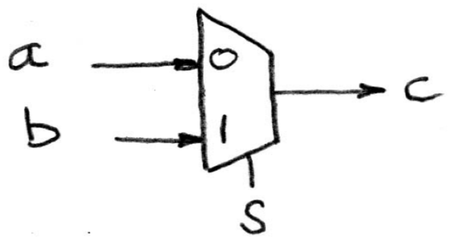


2-1 n-bit Data Multiplexor

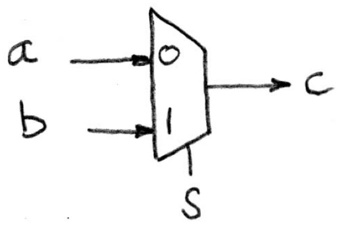


"mux"

How many rows in TT of a 1-bit Mux?



How many rows in TT of a 1-bit Mux?

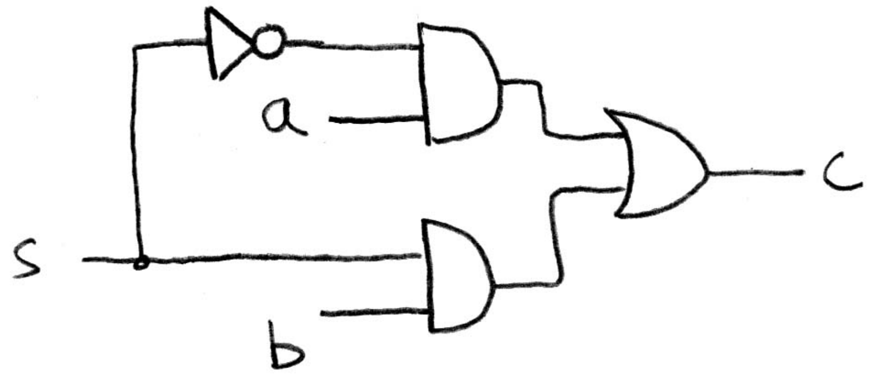


s	ab	c
0	00	0
0	01	0
0	10	1
0	11	1
1	00	0
1	01	1
1	10	0
1	11	1

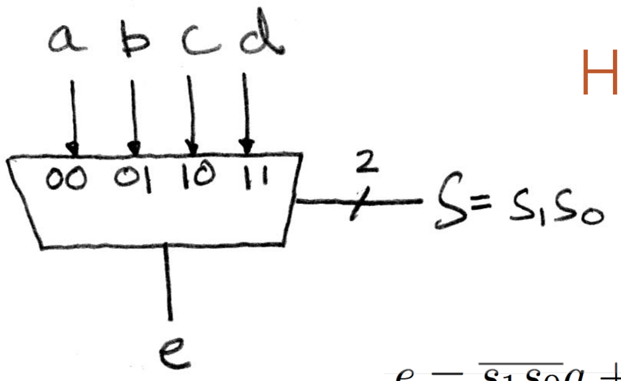
$$\begin{aligned} c &= \overline{s}a\overline{b} + \overline{s}ab + s\overline{a}b + sab \\ &= \overline{s}(a\overline{b} + ab) + s(\overline{a}b + ab) \\ &= \overline{s}(a(\overline{b} + b)) + s((\overline{a} + a)b) \\ &= \overline{s}(a(1) + s((1)b) \\ &= \overline{s}a + sb \end{aligned}$$

How do we build a 1-bit-wide mux?

$$\overline{s}a + sb$$



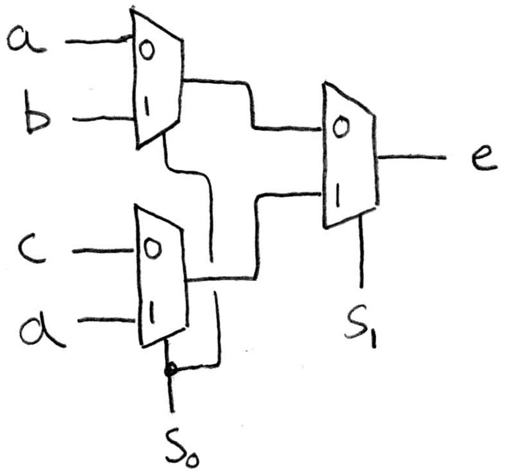
4-to-1 Multiplexor?



How many rows in TT?

$$e = \overline{s_1}\,\overline{s_0}a + \overline{s_1}s_0b + s_1\overline{s_0}c + s_1s_0d$$

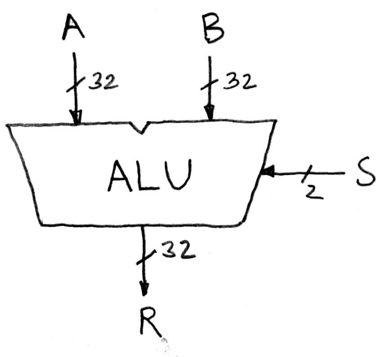
Is there any other way to do it?



Ans: Hierarchically!

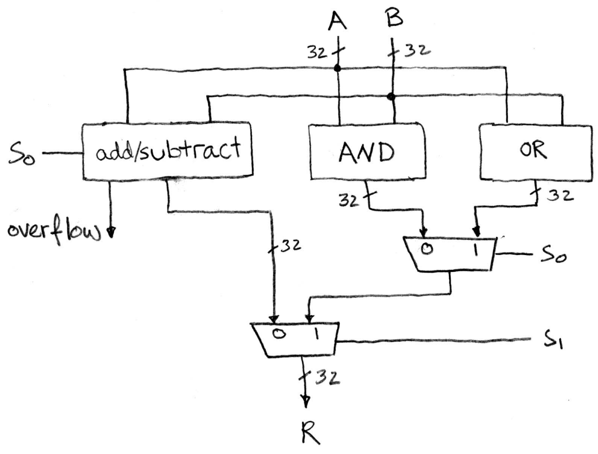
Arithmetic and Logic Unit

- Most processors contain a special logic block called "Arithmetic and Logic Unit" (ALU)
- We'll show you an easy one that does ADD, SUB, bitwise AND, bitwise OR



when S=00, R=A+B
when S=01, R=A-B
when S=10, R=A AND B
when S=11, R=A OR B

A Simple ALU



Adder/Subtracter Design -- how?

- Truth-table, then determine canonical form, then minimize and implement as we've seen before
- Look at breaking the problem down into smaller pieces that we can cascade or hierarchically layer

Adder/Subtracter – One-bit adder LSB…

$$\begin{array}{ccccc|c|} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline & s_3 & s_2 & s_1 & s_0 \end{array}$$

a_0	b_0	s_0	c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

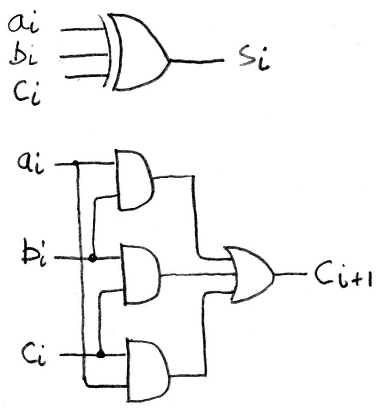
$$s_0 = a_0 \text{ XOR } b_0$$
$$c_1 = a_0 \text{ AND } b_0$$

Adder/Subtracter – One-bit adder (1/2)…

$$\begin{array}{cccc|c|c} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline & s_3 & s_2 & s_1 & s_0 \end{array}$$

a_i	b_i	c_i	s_i	c_{i+1}
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

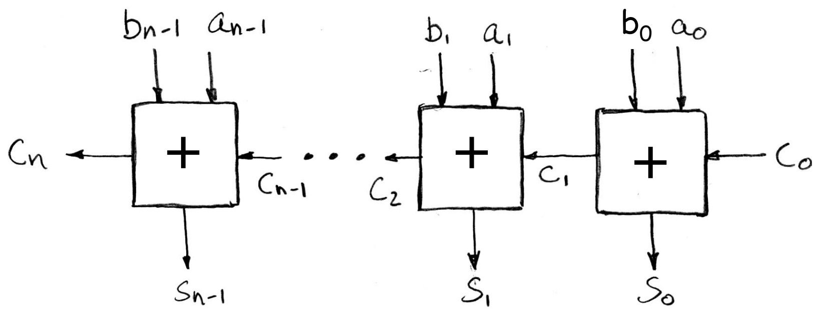
Adder/Subtracter – One-bit adder (2/2)…



a_i	b_i	c_i	s_i	c_{i+1}
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

$$\begin{aligned} s_i &= \text{XOR}(a_i, b_i, c_i) \\ c_{i+1} &= \text{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i \end{aligned}$$

N 1-bit adders ⇒ 1 N-bit adder



What about overflow?
Overflow = c_n?

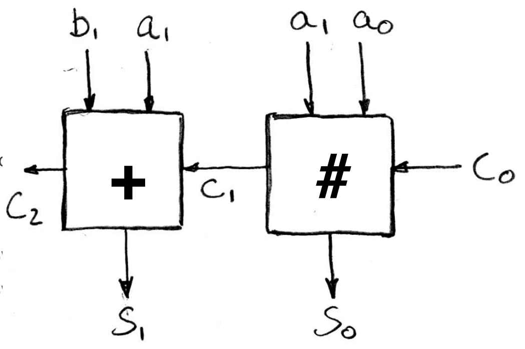
What about overflow?

Consider a 2-bit signed # & overflow:

- `10 = -2 + -2 or -1`
- `11 = -1 + -2 only`
- `00 =  0 NOTHING!`
- `01 =  1 + 1 only`

Highest adder

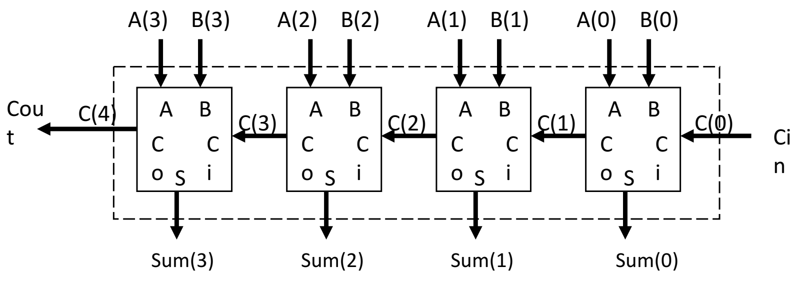
- C_1 = Carry-in = C_{in}, C_2 = Carry-out = C
- No C_{out} or $C_{in} \Rightarrow$ NO overflow!
- C_{in}, and $C_{out} \Rightarrow$ NO overflow!
- C_{in}, but no $C_{out} \Rightarrow$ A,B both > 0, overflo
- C_{out}, but no $C_{in} \Rightarrow$ A,B both < 0, overflo

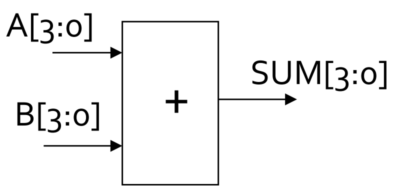


What op?

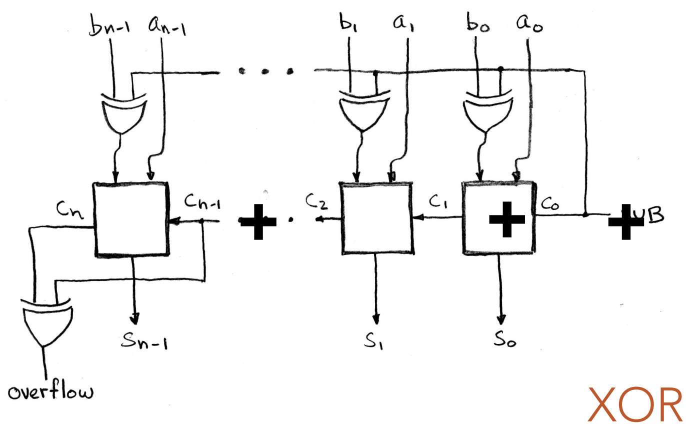
$$\boxed{\text{overflow} = c_n \text{ XOR } c_{n-1}}$$

4 Bit Ripple Carry Adder





Extremely Clever Subtractor



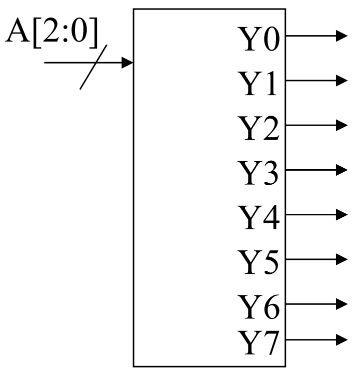
x	y	XOR(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

XOR serves as conditional inverter!

"And In conclusion…"

- Use muxes to select among input
  - S input bits selects 2S inputs
  - Each input can be n-bits wide, indep of S
- Can implement muxes hierarchically
- ALU can be implemented using a mux
  - Coupled with basic block elements
- N-bit adder-subtractor done using N 1-bit adders with XOR gates on input
  - XOR serves as conditional inverter

Decoder



if A= 000 then Y0=1 else Y0=0;
if A= 001 then Y1=1 else Y1=0;
if A= 010 then Y2=1 else Y2=0;
if A= 011 then Y3=1 else Y3=0;
if A= 100 then Y4=1 else Y4=0;
if A= 101 then Y5=1 else Y5=0;
if A= 110 then Y6=1 else Y6=0;
if A= 111 then Y7=1 else Y7=0;

Making a Design Run Fast

- Speed is much more important than saving gates.
  - Speed of a gate directly affects the maximum clock speed of digital system
- Gate speed is TECHNOLOGY dependent
  - 0.35u CMOS process has faster gates than 0.8u CMOS process
- Implementation choice will affect Design speed
  - A Custom integrated circuit will be faster than an FPGA implementation.
- Design approaches will affect clock speed of system
  - Smart designers can make a big difference

CSC 322: Computer Organization Lab

Part III: Sequential Logic

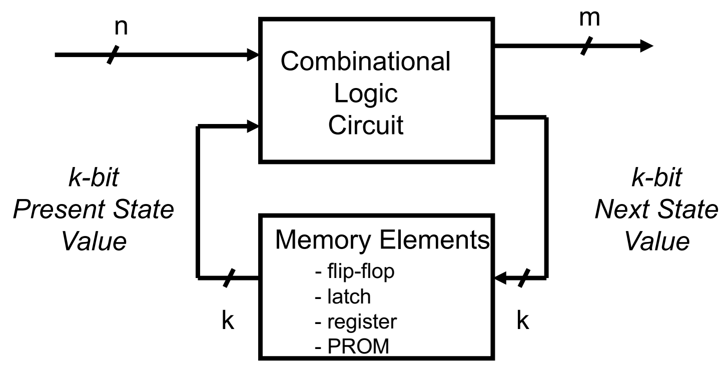
Dr. Haidar M. Harmanani

Sequential Systems Design

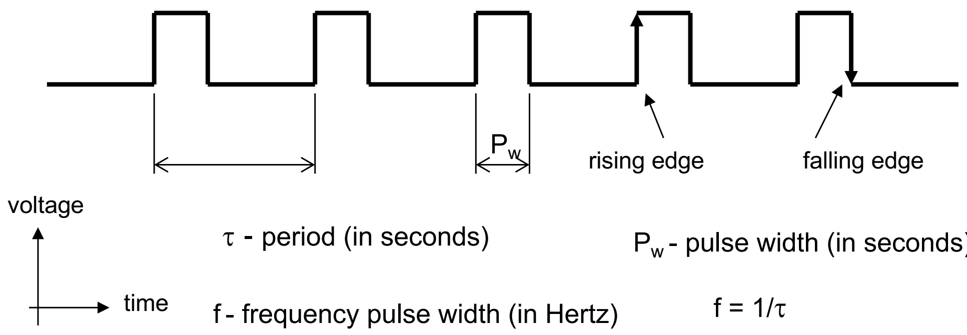
- Combinational Network
  - Output value only depends on input value
- Sequential Network
  - Output Value depends on input value and present state value
  - Sequential network must have some way of retaining state via memory devices.
  - Use a clock signal in a synchronous sequential system to control changes between states

Sequential System Diagram

- m outputs only depend on k PS bits - Moore Machine
- REMEMBER: Moore is Less !!
- m outputs depend on k PS bits AND n inputs - Mealy Machine



Clock Signal Review



τ - period (in seconds)

P_w - pulse width (in seconds)

f - frequency pulse width (in Hertz)

$f = 1/\tau$

duty cycle - ratio of pulse width to period (in %)

duty cycle = P_w/τ

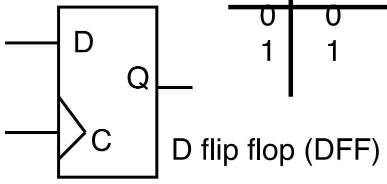
millisecond (ms) 10^{-3}	Kilohertz (KHz) 10^{3}
microsecond (μs) 10^{-6}	Megahertz (MHz) 10^{6}
nanosecond (ns) 10^{-9}	Gigahertz (GHz) 10^{9}

Memory Elements

Memory elements used in sequential systems are flip-flops and latches.

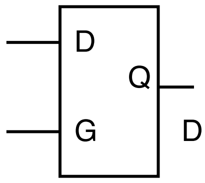
D	Q(t+1)
0	0
1	1

Q(t+1) is Q next state



D flip flop (DFF)

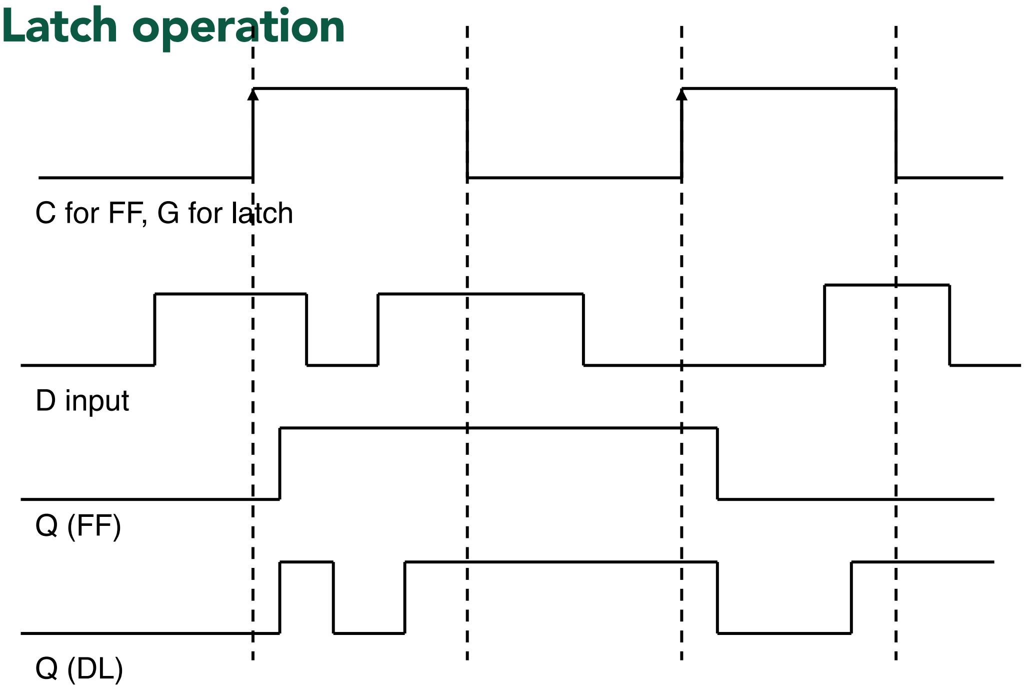
Flip-flops are edge triggered (either rising or falling edge).



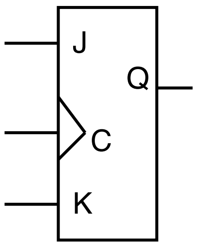
D latch (DL)

Latches are level sensitive. Q follows D when G=1, latches when G goes from 1 to 0.

D FF, D Latch operation

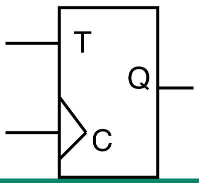


Other State Elements



J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q' (t)

JK useful for single bit flags with separate set(J), reset(K) control.



T	Q(t+1)
0	Q(t)
1	Q' (t)

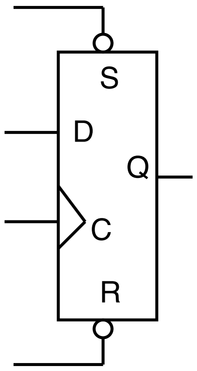
Useful for counter design.

DFFs are most common

- Most FPGA families only have DFFs
- DFF is fastest, simplest (fewest transistors) of FFs
- Other FF types (T, JK) can be built from DFFs
- We will use DFFs almost exclusively in this class
  - Will always used edge-triggered state elements (FFs), not level sensitive elements (latches).

Synchronous vs Asynchronous Inputs

Synchronous input: Output will change after active clock edge
Asychronous input: Output changes independent of clock

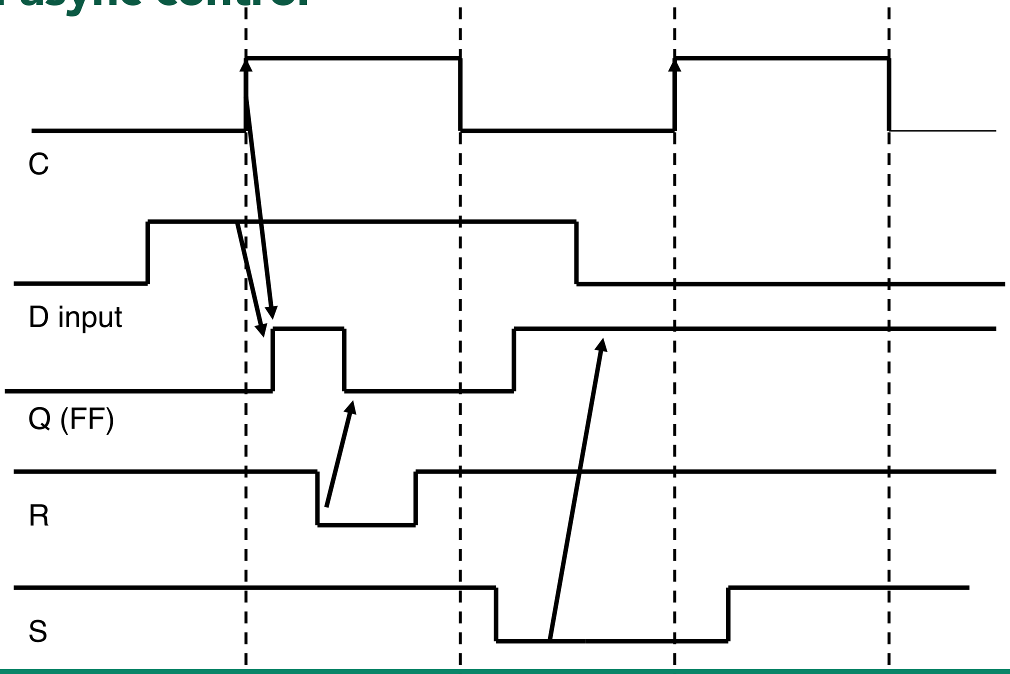


State elements often have async set, reset control.

D input is synchronous with respect to Clk

S, R are asynchronous. Q output affected by S, R independent of C. Async inputs are dominant over Clk.

DFF with async control



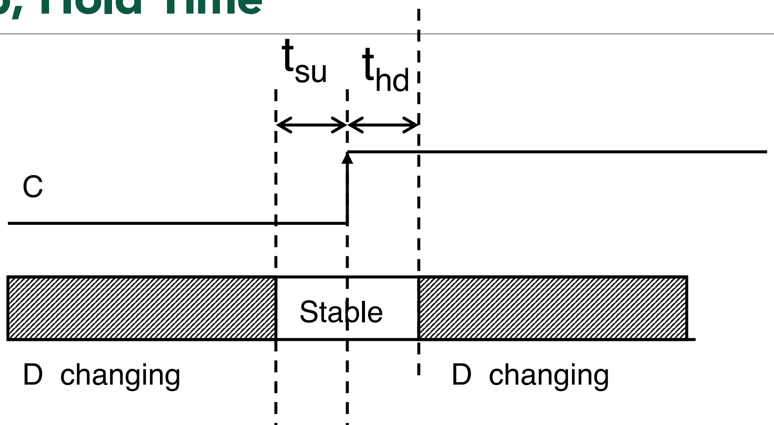
FF Timing

- Propagation Delay
  - C2Q: Q will change some propagation delay after change in C. Value of Q is based on D input for DFF.
  - S2Q, R2Q: Q will change some propagation delay after change on S input, R input
  - Note that there is NO propagation delay D2Q for DFF!
  - D is a Synchronous INPUT, no prop delay value for synchronous inputs

Setup, Hold Times

- Synchronous inputs (e.g. D) have Setup, Hold time specification with respect to the CLOCK input
- Setup Time: the amount of time the synchronous input (D) must be stable before the active edge of clock
- Hold Time: the amount of time the synchronous input (D) must be stable after the active edge of clock.

Setup, Hold Time

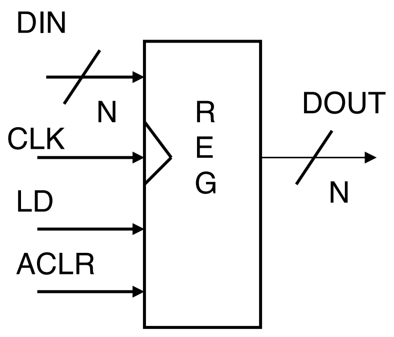


If changes on D input violate either setup or hold time, then correct FF operation is not guaranteed.

Setup/Hold measured around active clock edge.

Registers

The most common sequential building block is the register. A register is N bits wide, and has a load line for loading in a new value into the register.

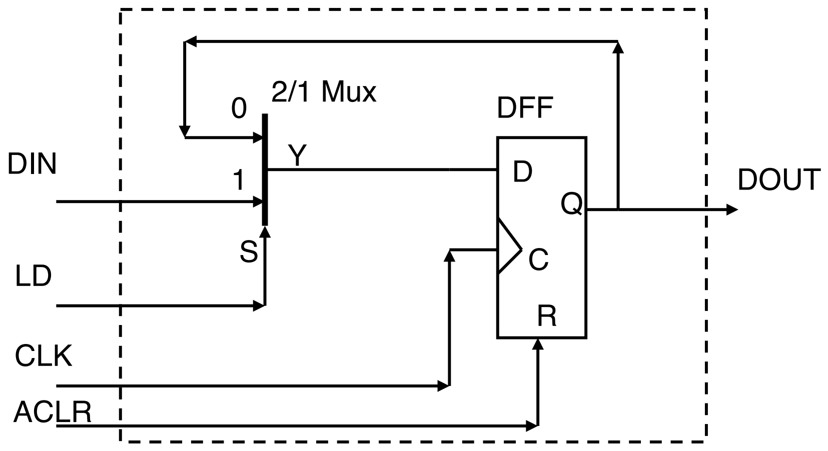


Register contents do not change unless LD = 1 on active edge of clock.

A DFF is NOT a register! DFF contents change every clock edge.

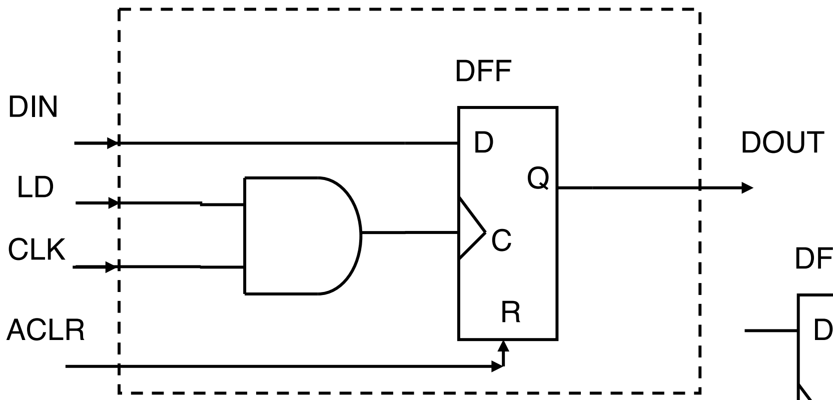
ACLR used to asynchronously clear the register

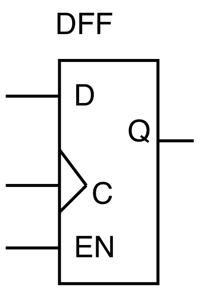
1 Bit Register using DFF, Mux



Note that DFF simply loads old value when LD = 0. DFF is loaded every clock cycle.

1 Bit Register using Gated Clock

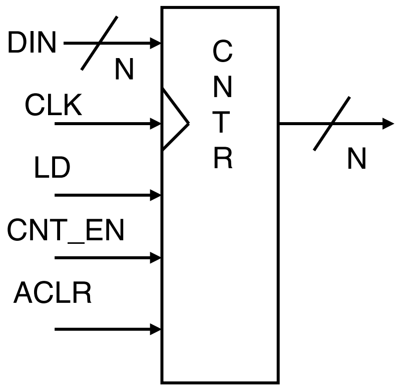




Saves power over previous design since DFF is not clocked every clock cycle. Many FPGAs offer an ‘enabled’ DFF as an integrated unit.

Counter

Very useful sequential building block. Used to generate memory addresses, or keep track of the number of times a datapath operation is performed.

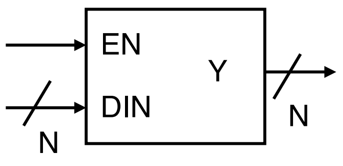


LD asserted loads counter with DIN value.

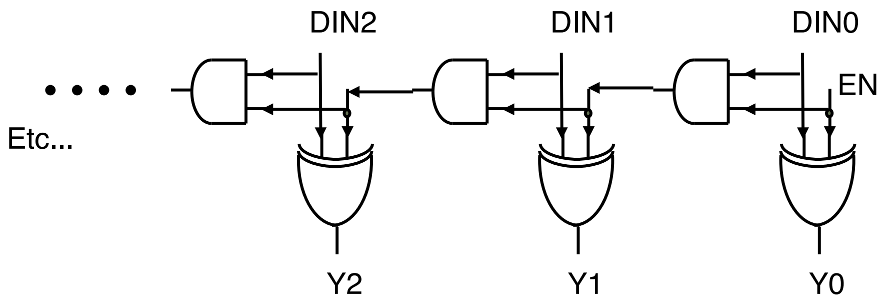
CNT_EN asserted will increment counter on next active clock edge.

ACLR will asynchronously clear the counter.

Incrementer: Combinational Building Block



When EN=1, Y = DIN + 1
When EN=0, Y = DIN



Sequential System Description

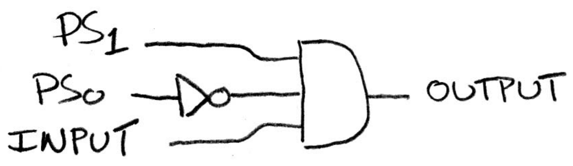
- The Q outputs of the flip-flops form a state vector
- A particular set of outputs is the Present State (PS)
- The state vector that occurs at the next discrete time (clock edge for synchronous designs) is the Next State (NS)
- A sequential circuit described in terms of state is a Finite State Machine (FSM)
  - Not all sequential circuits are described this way; i.e., registers are not described as FSMs yet a register is a sequential circuit.

Describing FSMs

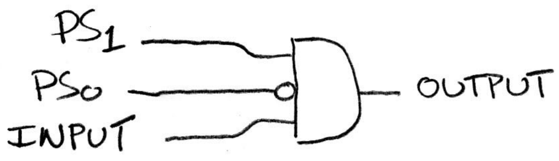
- State Tables
- State Equations
- State Diagrams
- Algorithmic State Machine (ASM) Charts
  - Preferred method in this class
- HDL descriptions

Truth Table State Machine Example

PS	Input	NS	Output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

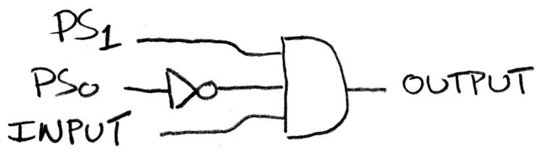


or equivalently…

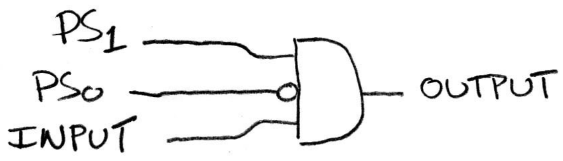


Boolean Algebra (e.g., for FSM)

PS	Input	NS	Output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

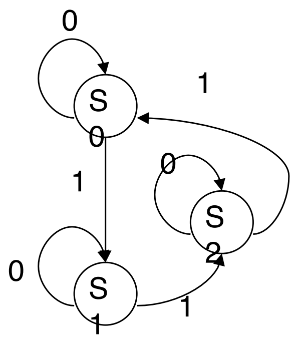


or equivalently…

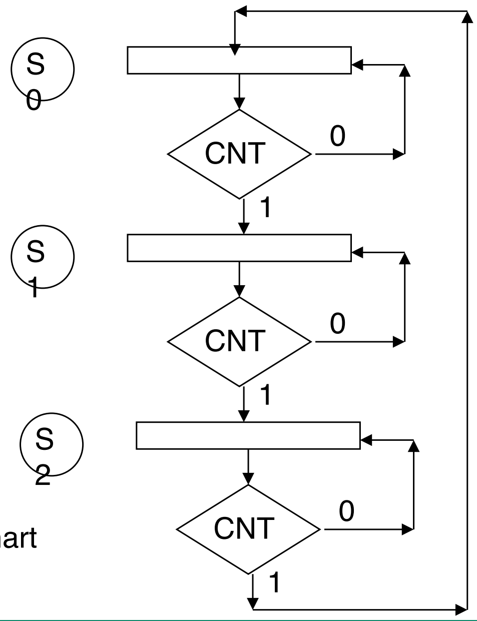


$$y = PS_1 \bullet \overline{PS_0} \bullet \text{INPUT}$$

Example State Machine



State Diagram
(Bubble Diagram)



ASM Chart

State Assignment

- State assignment is the binary coding used to represent the states
- Given N states, need at least $\log_2(N)$ FFs to encode the states
  – (i.e. 3 states, need at least 2 FFs for state information).

  S0 = 00, S1 = 01, S2 = 10 (FSM is now a modulo 3 counter)
- Do not always have to use the fewest possible number of FFs.
- A common encoding is One-Hot encoding - use one FF per state.

  – S0 = 001, S1 = 010, S2 = 100
- State assignment affects speed, gate count of FSM

FSM Implementation

- Use DFFs, State assignment: S0 = 00, S1 = 01, S2 = 10

	PS		NS			
Inc	Q1	Q0	Q1+	Q0	D1	D0
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	1	0	1	0
0	1	1	x	x	x	x
1	0	0	0	1	0	1
1	0	1	1	0	1	0
1	1	0	0	0	0	0
1	1	1	x	x	x	x

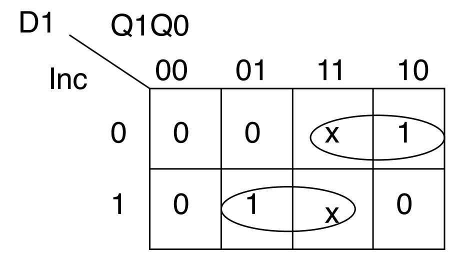
State Table

Equations

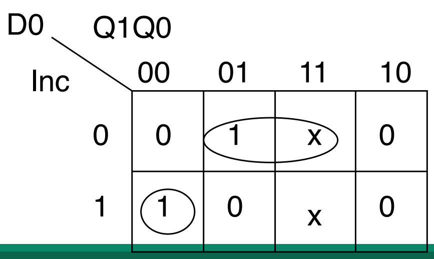
D1 = Inc' Q1Q0' + IncQ1' Q0

D0 = Inc' Q1' Q0 + IncQ1' Q0'

Minimize Equations (if desired)



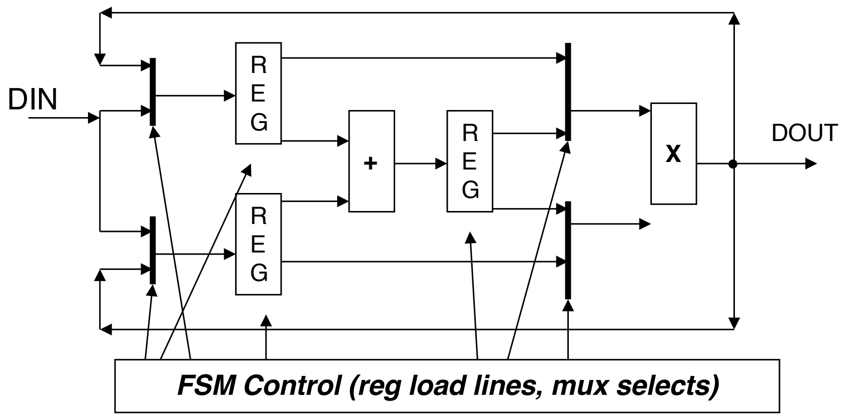
D1 = Inc' Q1 + Inc Q0



D1 = Inc' Q0 + Inc Q1' Q0'

FSM Usage

- Custom counters
- Datapath control



Memories

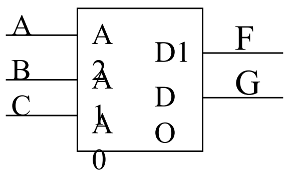
- Memories are K x N devices, K is the # of locations, N is the number bits per location (16 x 2 would be 16 locations, each storing 2 bits)
- K locations require $\log_2(K)$ address lines for selecting a location (i.e. a 16 location memory needs 4 address lines)
- A memory that is K x N, can be used to implement N *Boolean* equations, which use $\log_2(K)$ variables (the N *Boolean* equations must use the same variables).
- One address line is used for each *Boolean* variable, each bit of the output implements a different *Boolean* equation.
- The memory functions as a *Look Up Table (LUT).*

Memory Example

F (A,B,C) = A xor B xor C   G = AB + AC + BC

A B C	F	G
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

8 x 2 Memory



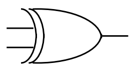
LookUp Table (LUT)

A[2:0] is 3 bit address bus, D[1:0] is 2 bit output bus.
Location 0 has "00",
Location 1 has "10",
Location 2 has "10",
etc....

Recall that Exclusive OR (xor) is

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

$$Y = A \oplus B$$
$$= A \text{ xor } B$$



A gated D latch in Verilog

```
module latch (D, clk, Q);
 input D, clk;
 output reg Q;

 always @(D, clk)
  if (clk)
   Q <= D;

endmodule
```

A D flip-flop.

```
module flipflop (D, Clock, Q);
  input D, Clock;
  output reg Q;

  always @(posedge Clock)
    Q <= D;

endmodule
```

A D flip-flop with asynchronous reset

```
module flipflop_ar (D, Clock, Resetn, Q);
  input D, Clock, Resetn;
  output reg Q;

  always @(posedge Clock, negedge Resetn)
    if (Resetn == 0)
      Q <= 0;
    else
      Q <= D;

endmodule
```

A D flip-flop with synchronous reset.

```
module flipflop_sr (D, Clock, Resetn, Q);
 input D, Clock, Resetn;
 output reg Q;

 always @(posedge Clock)
  if (Resetn == 0)
   Q <= 0;
  else
   Q <= D;

endmodule
```

A four-bit register with asynchronous clear.

```
module reg4 (D, Clock, Resetn, Q);
 input [3:0] D;
 input Clock, Resetn;
 output reg [3:0] Q;

 always @(posedge Clock, negedge Resetn)
 if (Resetn == 0)
  Q <= 4'b0000;
 else
  Q <= D;

endmodule
```

An n-bit register with asynchronous clear and enable.

```
module regne (D, Clock, Resetn, E, Q);
  parameter n = 4;
  input [n-1:0] D;
  input Clock, Resetn, E;
  output reg [n-1:0] Q;

  always @(posedge Clock, negedge Resetn)
    if (Resetn == 0)
      Q <= 0;
    else if (E)
      Q <= D;

endmodule
```

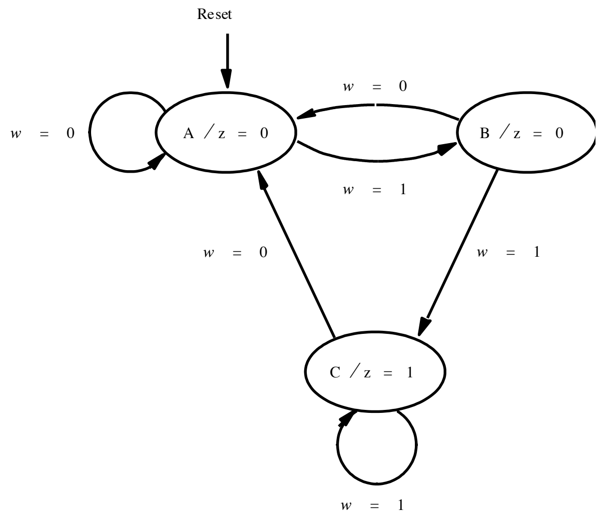
A three-bit shift register

```
module shift3 (w, Clock, Q);
  input w, Clock;
  output reg [1:3] Q;

  always @(posedge Clock)
  begin
    Q[3] <= w;
    Q[2] <= Q[3];
    Q[1] <= Q[2];
  end

endmodule
```

Code for a four-bit counter

```
module count4 (Clock, Resetn, E, Q);
  input Clock, Resetn, E;
  output reg [3:0] Q;

  always @(posedge Clock, negedge Resetn)
    if (Resetn == 0)
        Q <= 0;
    else if (E)
        Q <= Q + 1;

endmodule
```

State diagram of a simple Moore-type FSM



```
module moore (Clock, w, Resetn, z);
 input Clock, w, Resetn;
 output z;
 reg [1:0] y, Y;
 parameter A = 2'b00, B = 2'b01, C = 2'b10;

 always @(w, y)
 begin
  case (y)
   A:  if (w = = 0)  Y = A;
       else  Y = B;
   B:  if (w = = 0)  Y = A;
       else  Y = C;
   C:  if (w = = 0)  Y = A;
       else  Y = C;
   default: Y = 2'bxx;
  endcase
 end

 always @(posedge Clock, negedge Resetn)
 begin
  if (Resetn = = 0)
     y <= A;
  else
     y <= Y;
 end
  assign z = (y = = C);
endmodule
```

Code for a Moore machine.

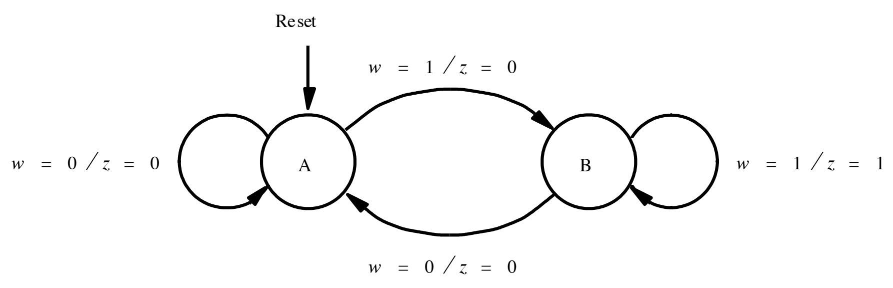
```
module moore (Clock, w, Resetn, z);
 input Clock, w, Resetn;
 output z;
 reg [1:0] y;
 parameter A = 2'b00, B = 2'b01, C = 2'b10;

 always @(posedge Clock, negedge Resetn)
 begin
  if (Resetn = = 0)
   y <= A;
  else
   case (y)
    A:  if (w = = 0)  y <= A;
       else  y <= B;
    B:  if (w = = 0)  y <= A;
       else  y <= C;
    C:  if (w = = 0)  y <= A;
       else  y <= C;
    default:  y <= 2'bxx;
   endcase
 end
  assign z = (y = = C);
endmodule
```

Alternative version of the code for a Moore machine.

```
module mealy (Clock, w, Resetn, z);
 input Clock, w, Resetn ;
 output reg z ;
 reg y, Y;
 parameter A = 1’ b0, B = 1’ b1;

 always @(w, y)
  case (y)
   A:  if (w = = 0)
    begin
     Y = A;
     z = 0;
    end
    else
    begin
     Y = B;
     z = 0;
    end
   B:  if (w == 0)
    begin
     Y = A;
     z = 0;
    end
    else
    begin
     Y = B;
     z = 1;
    end
  endcase

 always @(posedge Clock , negedge Resetn)
  if (Resetn = = 0)
    y <= A;
  else
    y <= Y;
endmodule
```

Code for a Mealy machine.

State diagram of a Mealy-type FSM.



```
module mealy (Clock, w, Resetn, z);
 input Clock, w, Resetn ;
 output reg z ;
 reg y, Y;
 parameter A = 1’ b0, B = 1’ b1;

 always @(w, y)
  case (y)
   A:  if (w = = 0)
    begin
     Y = A;
     z = 0;
    end
    else
    begin
     Y = B;
     z = 0;
    end
   B:  if (w == 0)
    begin
     Y = A;
     z = 0;
    end
    else
    begin
     Y = B;
     z = 1;
    end
  endcase

 always @(posedge Clock , negedge Resetn)
  if (Resetn = = 0)
    y <= A;
  else
    y <= Y;
endmodule
```

Code for a Mealy machine.

Verilog operators and bit lengths.

Category	Examples	Bit Length
Bitwise	~A, +A, –A A & B, A | B, A ~ ^ B, A ^ ~ B	L(*A*) MAX (L(*A*), L(*B*))
Logical	!A, A && B, A || B	1 bit
Reduction	&A, ~&A, |A, ~ |A, ^ ~ A, ~ ^A	1 bit
Relational	A = = B, A != B, A > B, A < B A >= B, A <= B A = = = B, A != = B	1 bit
Arithmetic	A + B, A – B, A * B, A / B A % B	MAX (L(*A*), L(*B*))
Shift	A << B, A >> B	L(*A*)
Concatenate	{A, …,B}	L(*A*) + ⋯ + L(*B*)
Replication	{B{A}}	B ∗L(*A*)
Condition	A ? B : C	MAX (L(*B*), L(*C*))

Verilog Gates

Name	Description	Usage
and	$f = (a \cdot b \cdots)$	**and**(f, a, b,…)
nand	$f = \overline{(a \cdot b \cdots)}$	**nand**(f, a, b, …)
or	$f = (a + b + \cdots)$	**or** (f, a, b, …)
nor	$f = \overline{(a + b + \cdots)}$	**nor**(f, a, b, …)
xor	$f = (a \oplus b \oplus \cdots)$	**xor**(f, a, b, …)
xnor	$f = (a \odot b \odot \cdots)$	**xnor**(f, a, b, …)
not	$f = \overline{a}$	**not**(f, a)
buf	$f = a$	**buf** (f, a)
notif0	$f = (!e\ ?\ \overline{a} : \text{'}bz)$	**notif0**(f, a, e)
notif1	$f = (e\ ?\ \overline{a} : \text{'}bz)$	**notif1**(f, a, e)
bufif0	$f = (!e\ ?\ a : \text{'}bz)$	**bufif0**(f, a, e)
bufif1	$f = (e\ ?\ a : \text{'}bz)$	**bufif1**(f, a, e)