Lecture 15:
FSMs, ROMs, PLAs, EEPROMs

Mark McDermott
Electrical and Computer Engineering
The University of Texas at Austin

Agenda

- Finite State Machines (FSM)
- Read-Only Memories (ROM)
- Programmable Logic Arrays (PLA)
- EEPROMS & Flash Memories

Sequential Logic and FSM

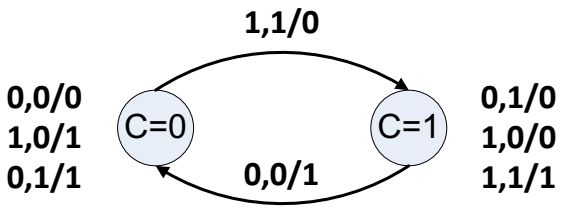
- Computers are made of sequential & combinational logic blocks
- Truth tables are used to design combinational logic, but can't be used to design sequential logic
  - Finite state machines (FSM) are used instead
- FSM describes a sequential logic block in terms of:
  - Set of states
  - State transition function
    - Defined on current state and input
  - Output function
    - Defined on current state and input, or on current state only

Two Types of Finite State Machines

- **Transition Based - Mealy**
  - the output value depends on the state and input state
  - output is associated with transitions
- **State Based - Moore**
  - the output value depends only on the state
  - output is associated with states
- **Which one do you use?**
  - Moore machines are faster
  - Mealy machines are smaller
  - The outputs of Mealy machines can glitch
- **Any Moore machine can be turned into a Mealy machine (and vice versa)**

Serial Adder Implementations

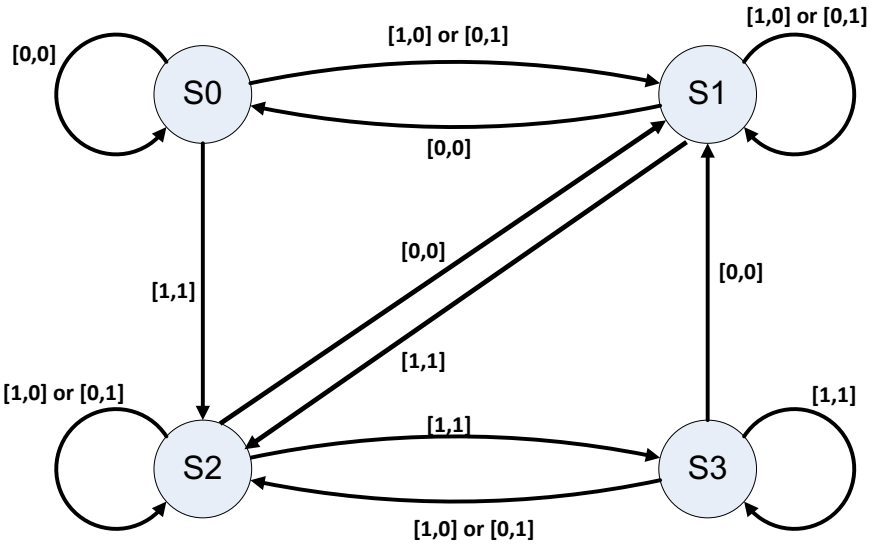
Mealy Machine implementation



Note: X,X/Y => [input, input]/output

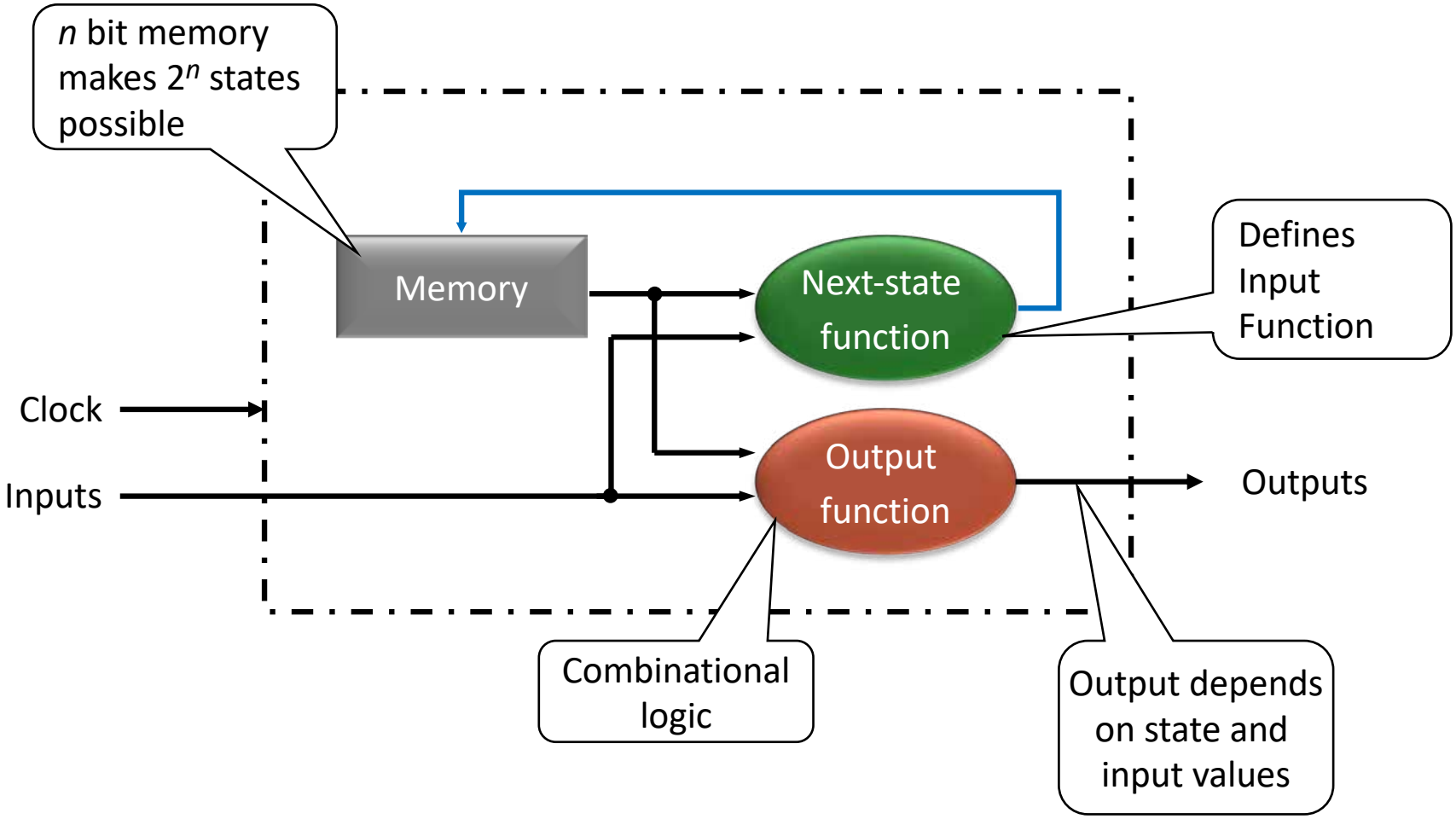
	Inputs			Output
C=0	0	0	C=0	0
C=0	0	1	C=0	1
C=0	1	0	C=0	1
C=0	1	1	C=1	0
C=1	0	0	C=0	1
C=1	0	1	C=1	0
C=1	1	0	C=1	0
C=1	1	1	C=1	1

Moore Machine implementation

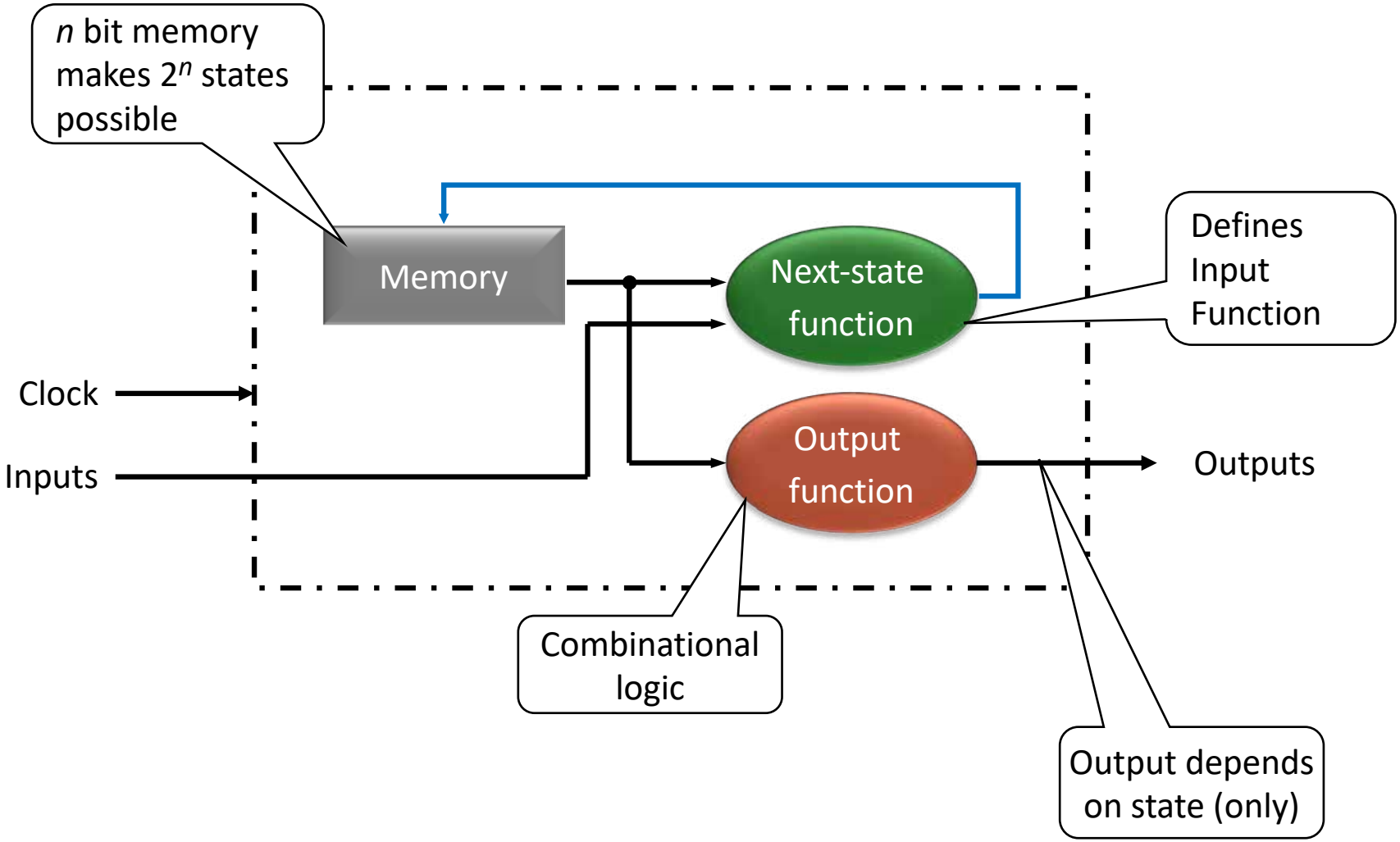


Current State		Inputs		Next State		Carry Out
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	1	0	1
0	1	0	0	0	0	0
0	1	0	1	0	1	0
0	1	1	0	0	1	0
0	1	1	1	1	0	1
1	0	0	0	0	1	0
1	0	0	1	1	0	1
1	0	1	0	1	0	1
1	0	1	1	1	1	1
1	1	0	0	0	1	0
1	1	0	1	1	0	1
1	1	1	0	1	0	1
1	1	1	1	1	1	1

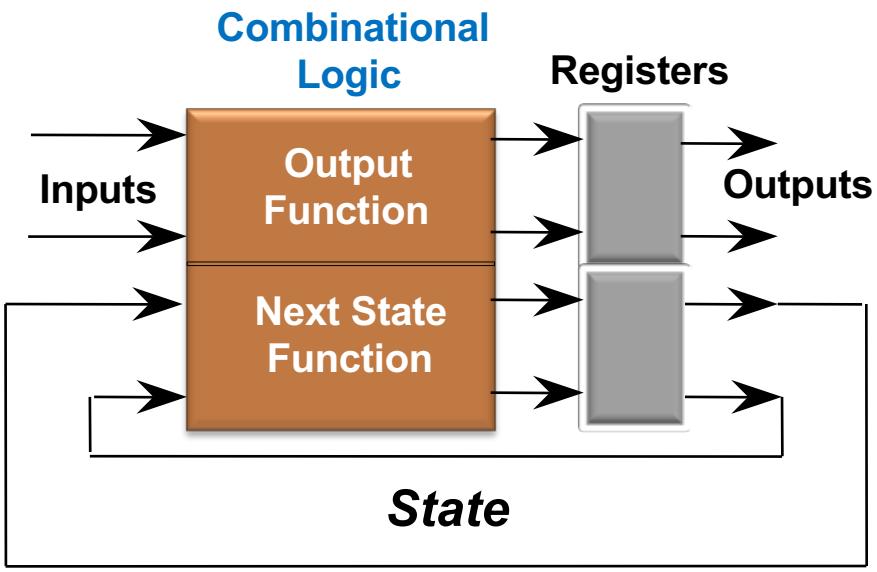
Finite State Machine: Mealy Machine



Finite State Machine: Moore Machine



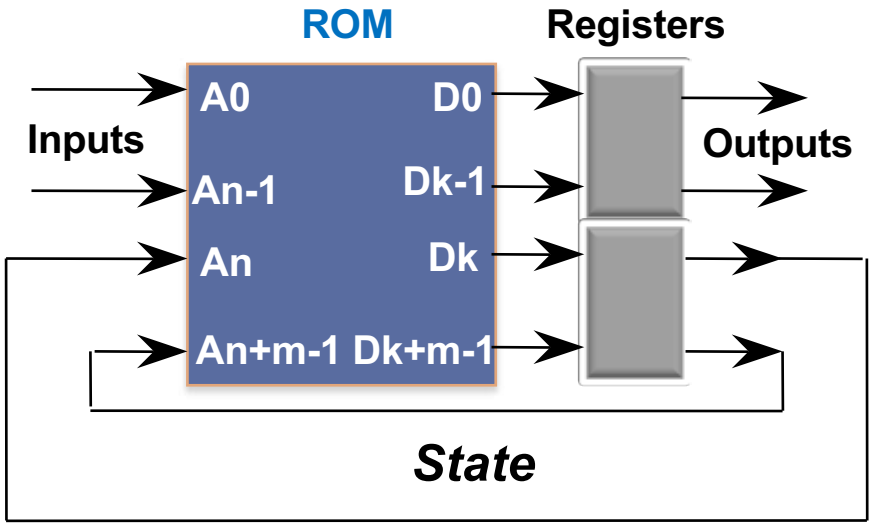
Moore Machine Implementation Strategies



Combinational Logic Realization

Inputs & Current State form inputs to the combinational logic

Output from combinational logic form the Outputs & Next State



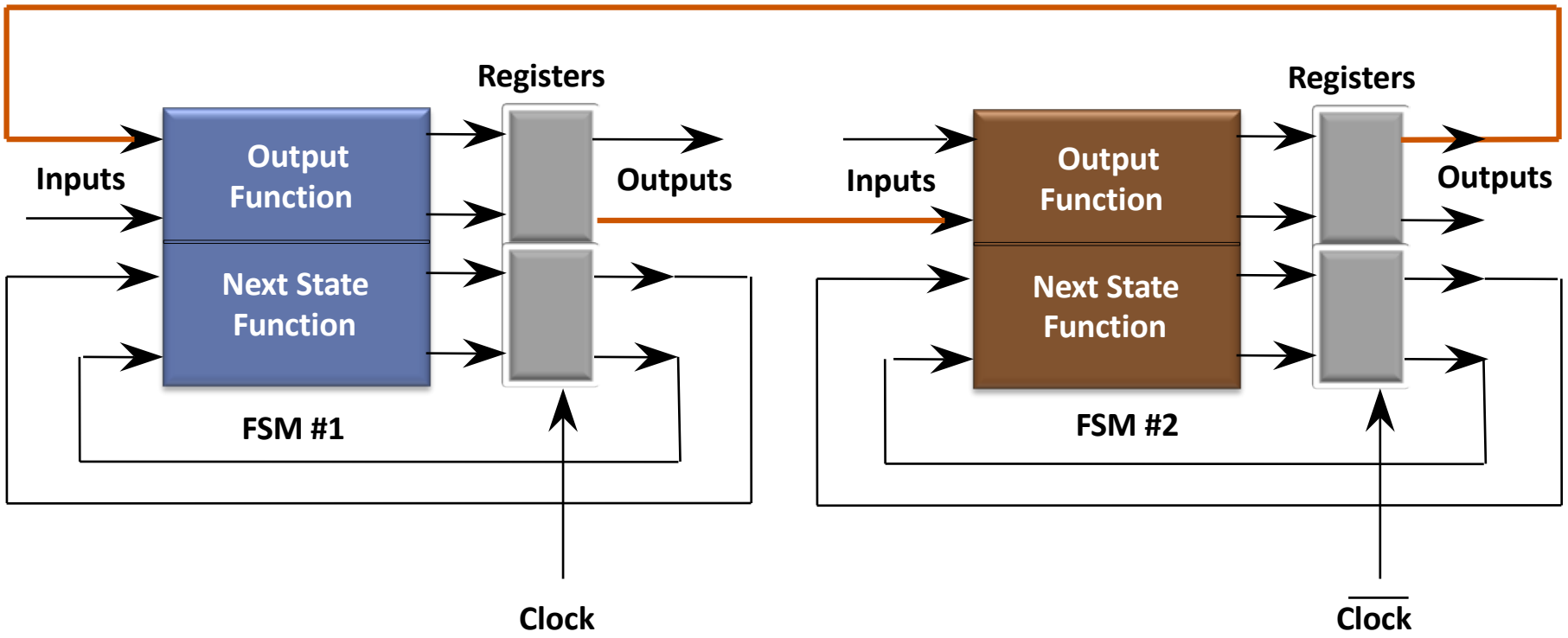
ROM-based Realization

Inputs & Current State form the address to the ROM

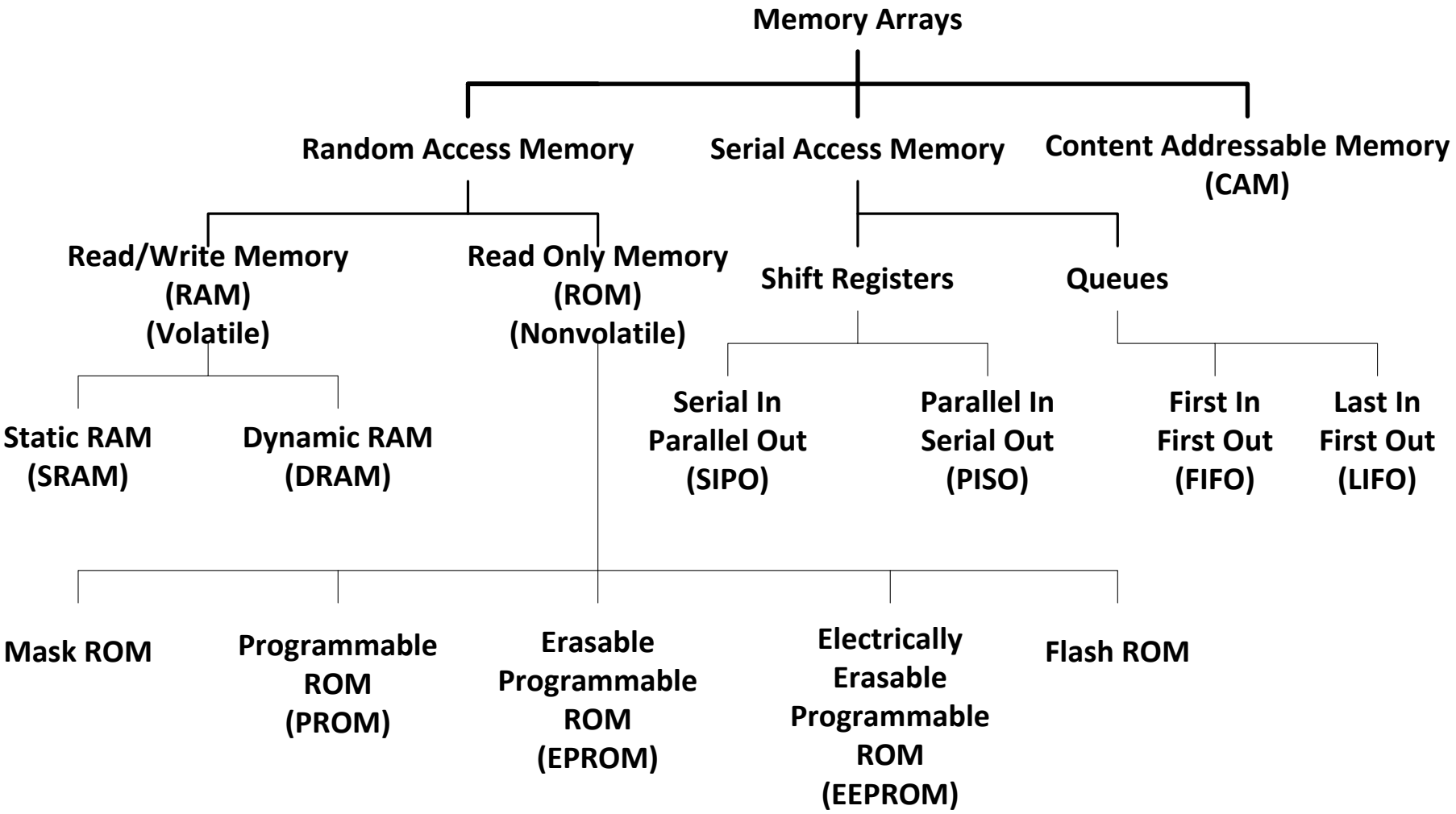
ROM data bits form the Outputs & Next State

Communicating FSMs (CFSM)

- Used in hierarchical FSM applications
- Used to model Petri Nets, Communication Protocols, etc.



Memory Arrays



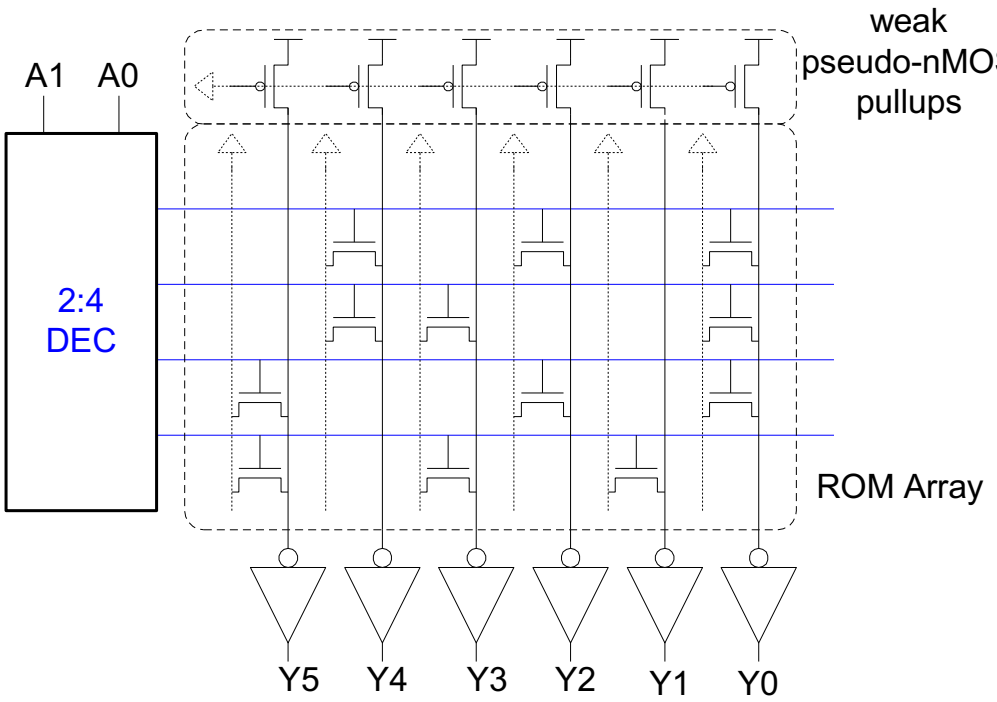
Read-Only Memories

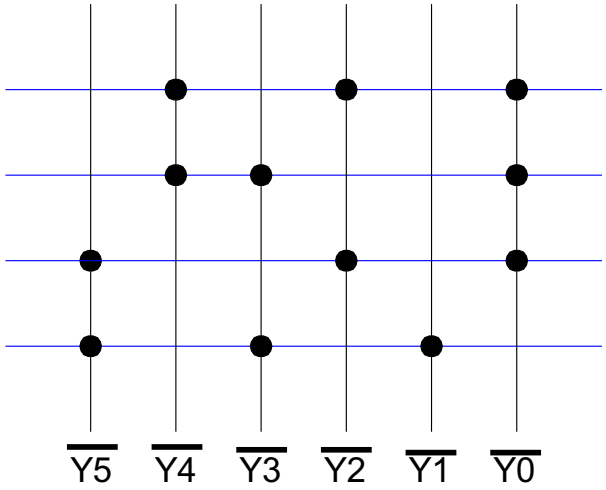
- **Read-Only Memories are nonvolatile**
  - Retain their contents when power is removed
- **Mask-programmed ROMs use one transistor per bit**
  - Presence or absence determines 1 or 0

ROM Example

- **4-word x 6-bit ROM**
  - Represented with dot diagram
  - Dots indicate 1's in ROM

```
Word 0: 010101
Word 1: 011001
Word 2: 100101
Word 3: 101010
```

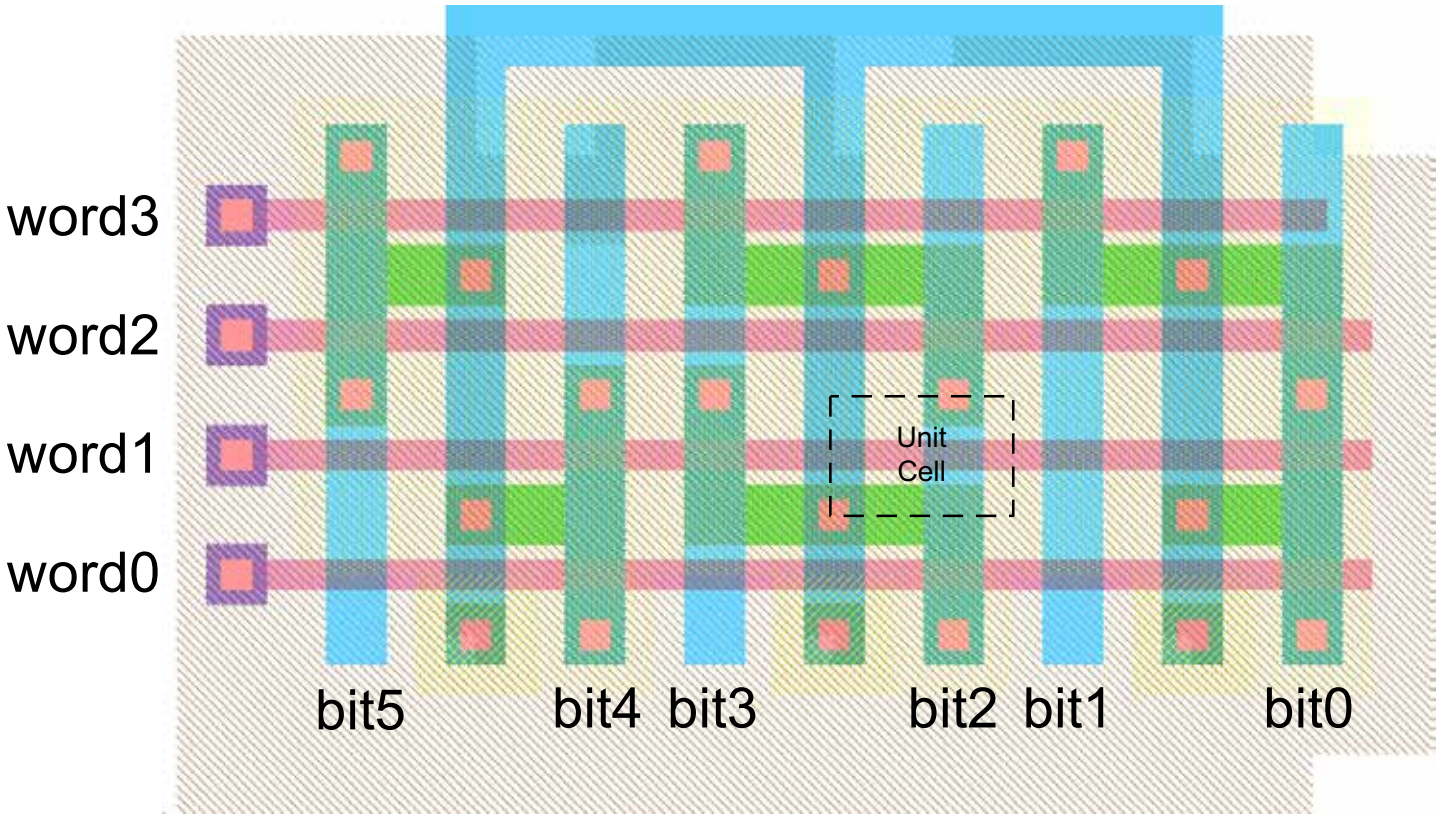




Looks like 6 4-input pseudo-nMOS NORs

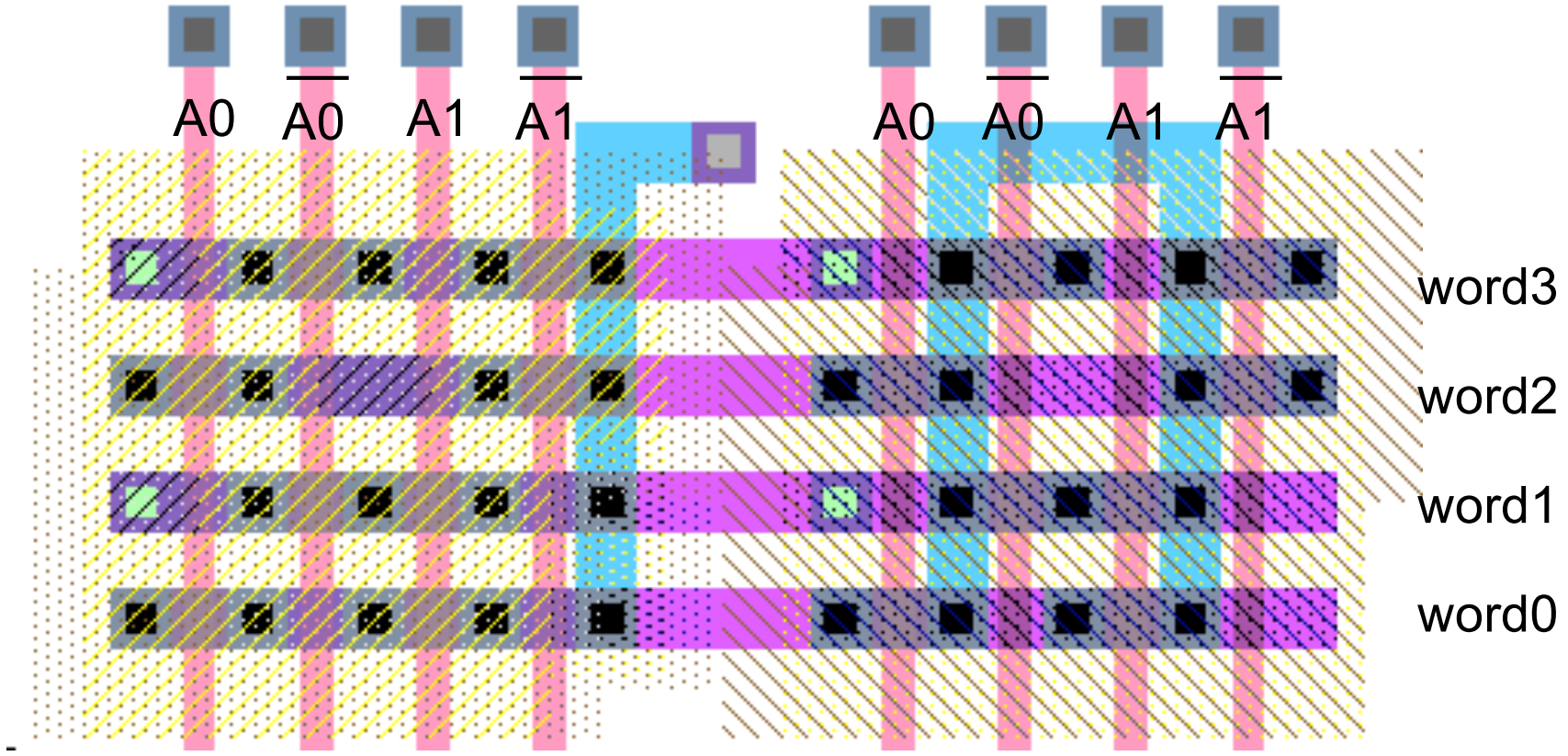
ROM Array Layout

- **Unit cell is 12 x 8 l (about 1/10 size of SRAM)**

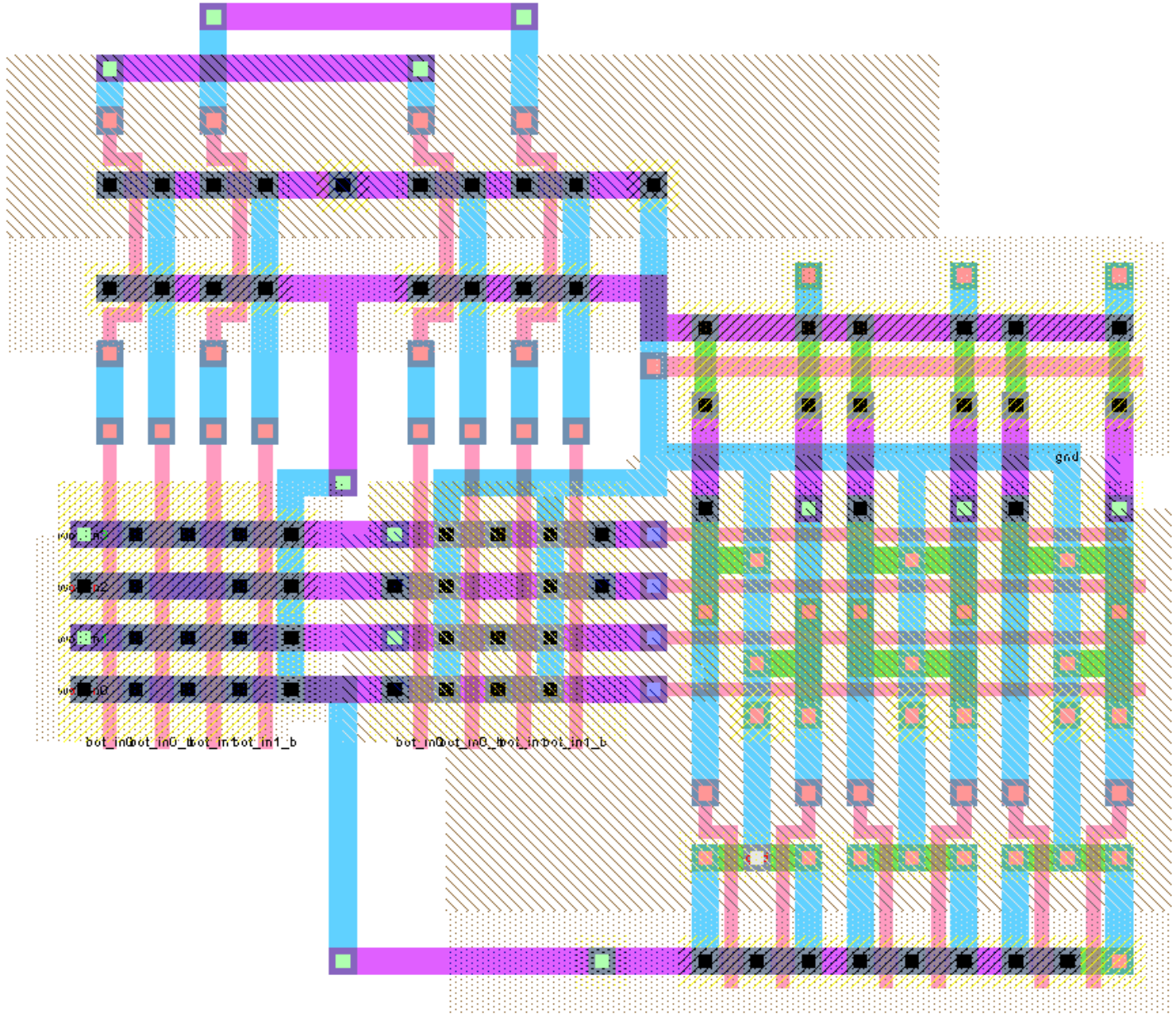


Row Decoders

- **ROM row decoders must pitch-match with ROM**
  - **Only a single track per word!**

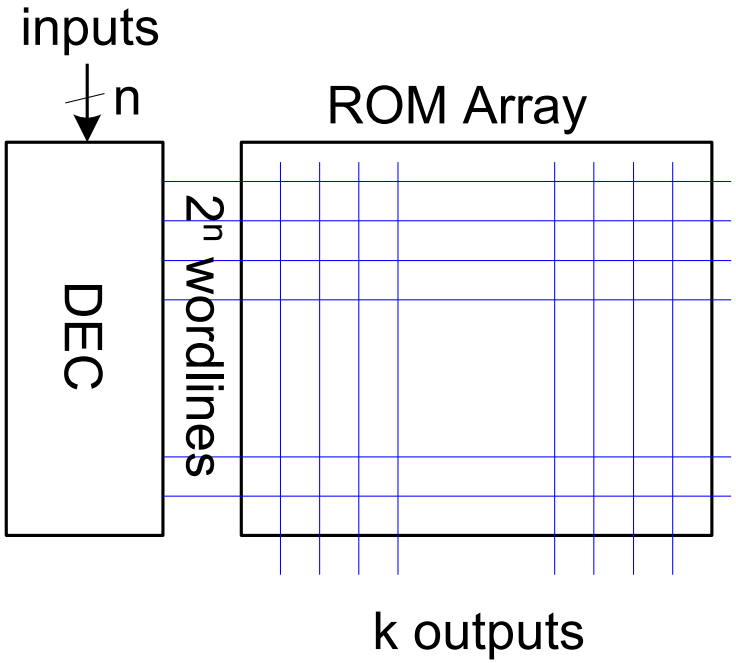


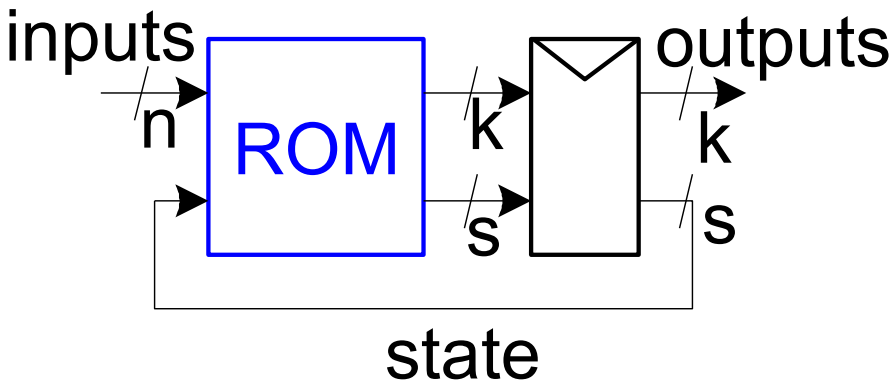
Complete ROM Layout



Building Logic with ROMs

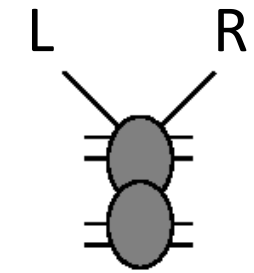
- **ROM as lookup table containing truth table**
  - n inputs, k outputs requires 2^n words x k bits
  - Changing function is easy – reprogram ROM
- **Finite State Machine**
  - n inputs, k outputs, s bits of state
  - Build with 2^{n+s} x (k+s) bit ROM and (k+s) bit reg





Example: RoboAnt

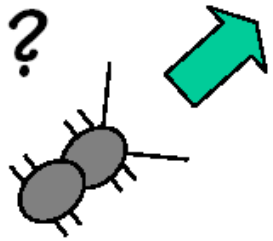
Let's build an Ant



Sensors: Antennae
(L,R) – 1 when in contact
Actuators: Legs
Forward step F
Ten degree turns TL, TR
Goal: make our ant smart enough to get out of a maze
Strategy: keep right antenna on wall

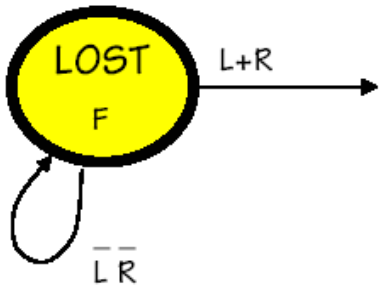
(RoboAnt adapted from MIT 6.004 2002 OpenCourseWare by Ward and Terman)

Lost in space

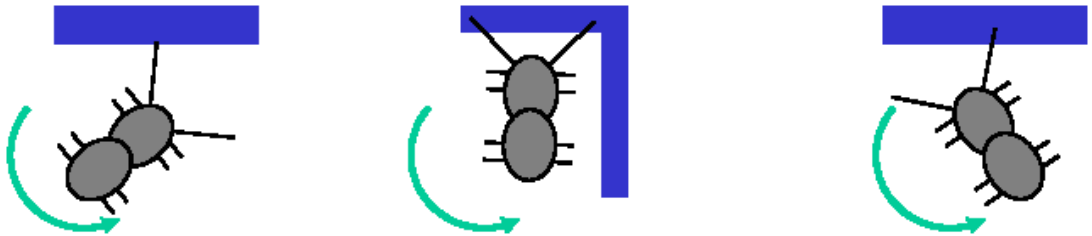


Action: go forward until we hit something

Initial state

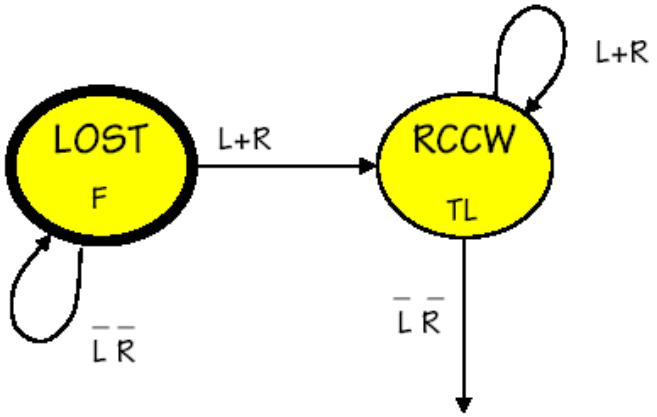


Bonk!!!

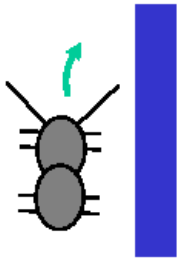


Action: turn left (rotate counterclockwise)

Until we don't touch anymore

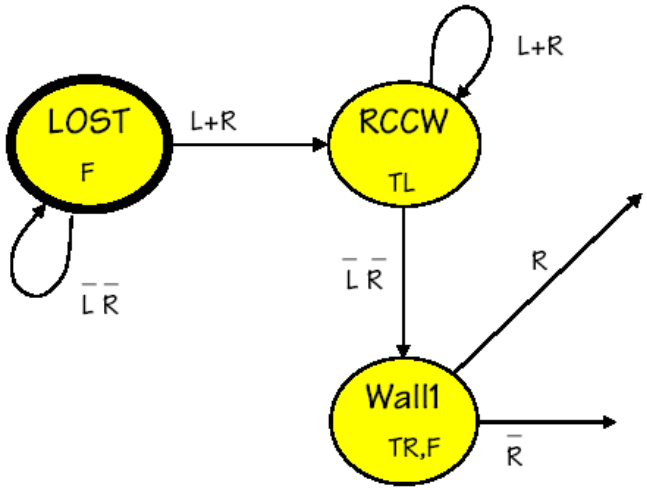


A little to the right

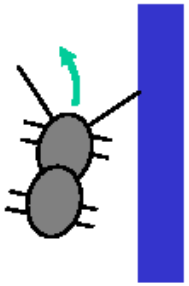


Action: step forward and turn right a little

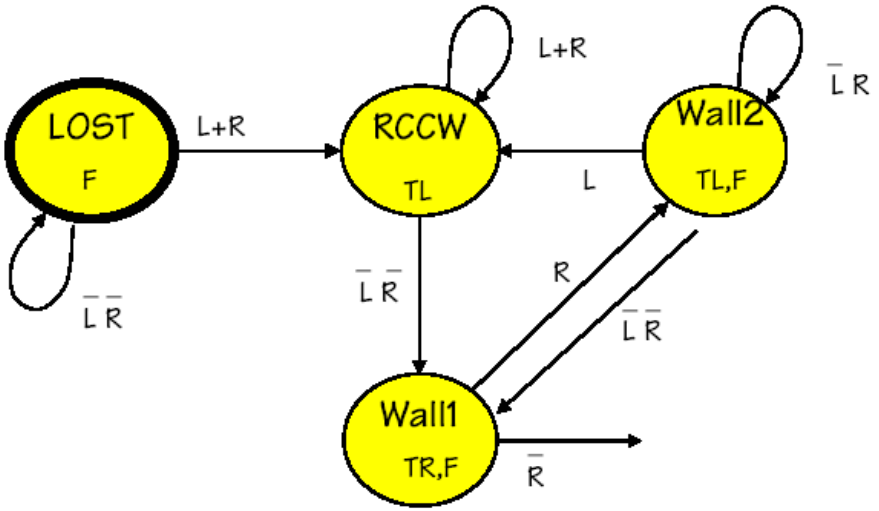
Looking for wall



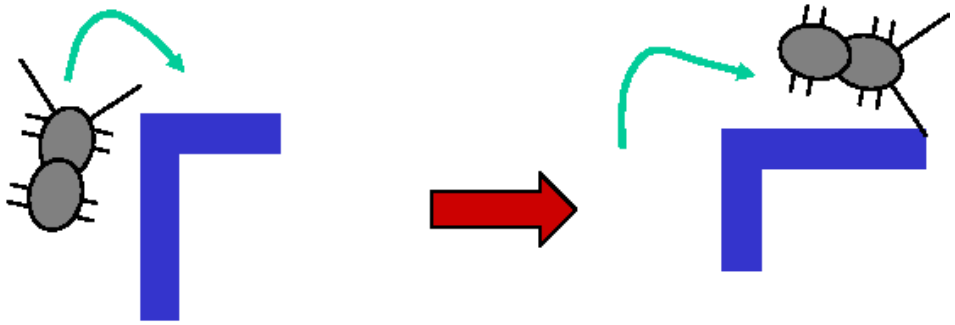
Then a little to the right



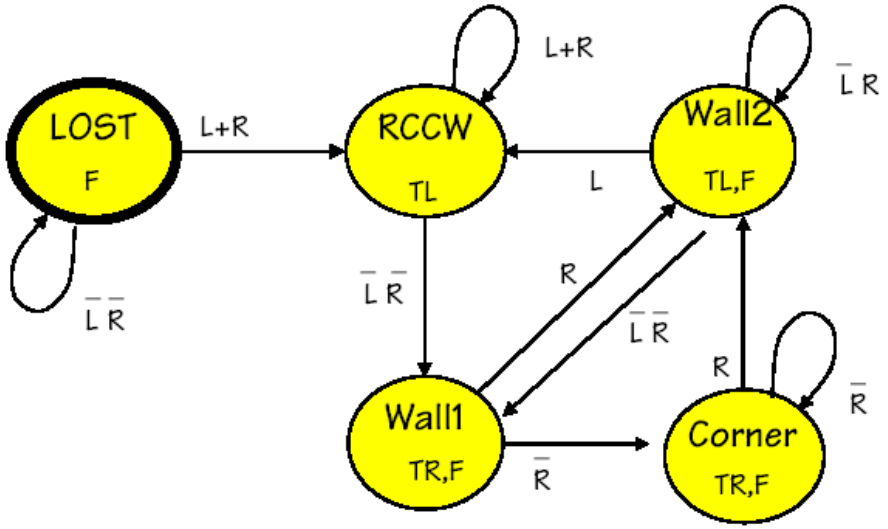
Action: step and turn left a little, until not touching



Whoops – a corner!

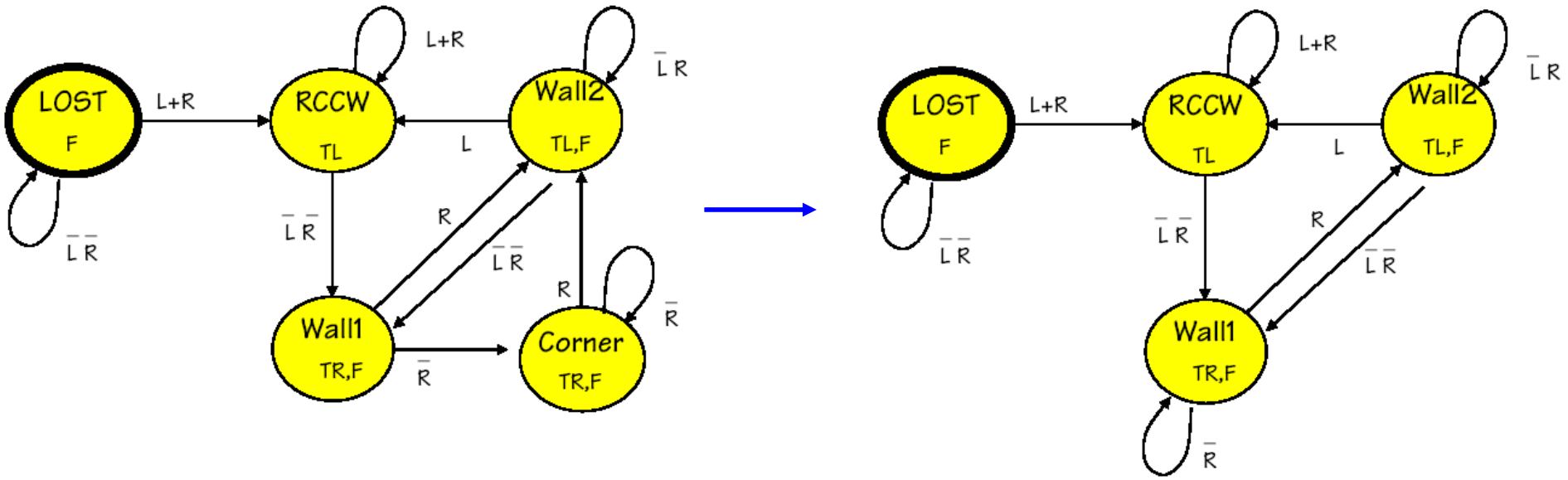


Action: step and turn right until hitting next wall



Simplification

Merge equivalent states where possible



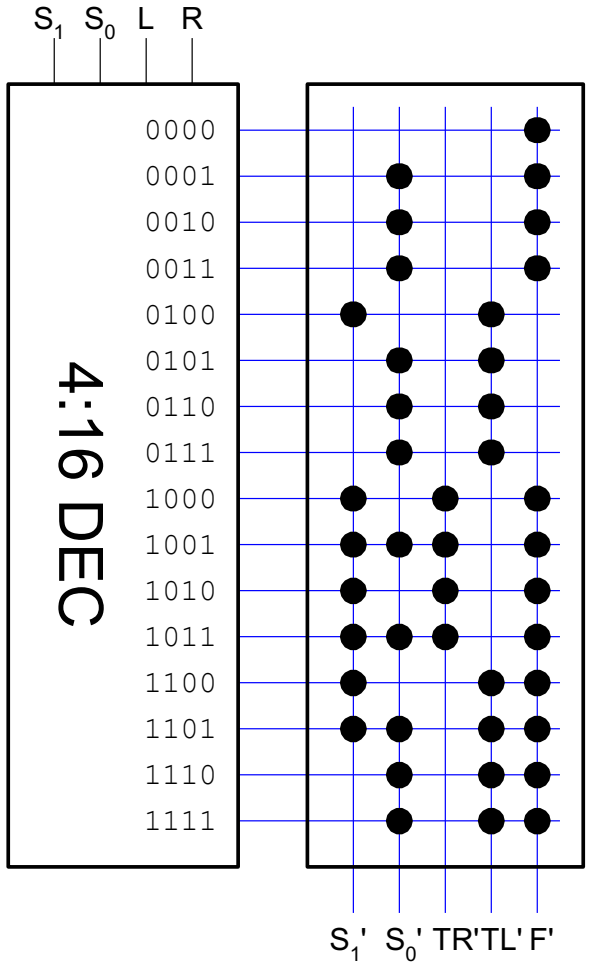
Need to do state assignment next

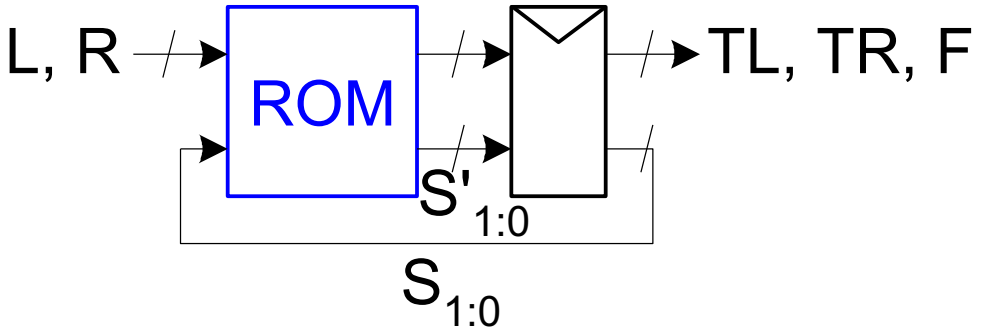
State Transition Table

	Current state	Inputs		Next state	Output values		
	$S_{1:0}$	L	R	$S_{1:0}'$	TR	TL	F
Lost	00	0	0	00	0	0	1
	00	1	X	01	0	0	1
	00	0	1	01	0	0	1
RCCW	01	1	X	01	0	1	0
	01	0	1	01	0	1	0
	01	0	0	10	0	1	0
Wall1	10	X	0	10	1	0	1
	10	X	1	11	1	0	1
Wall2	11	1	X	01	0	1	1
	11	0	0	10	0	1	1
	11	0	1	11	0	1	1

ROM Implementation

- **16-word x 5 bit ROM**





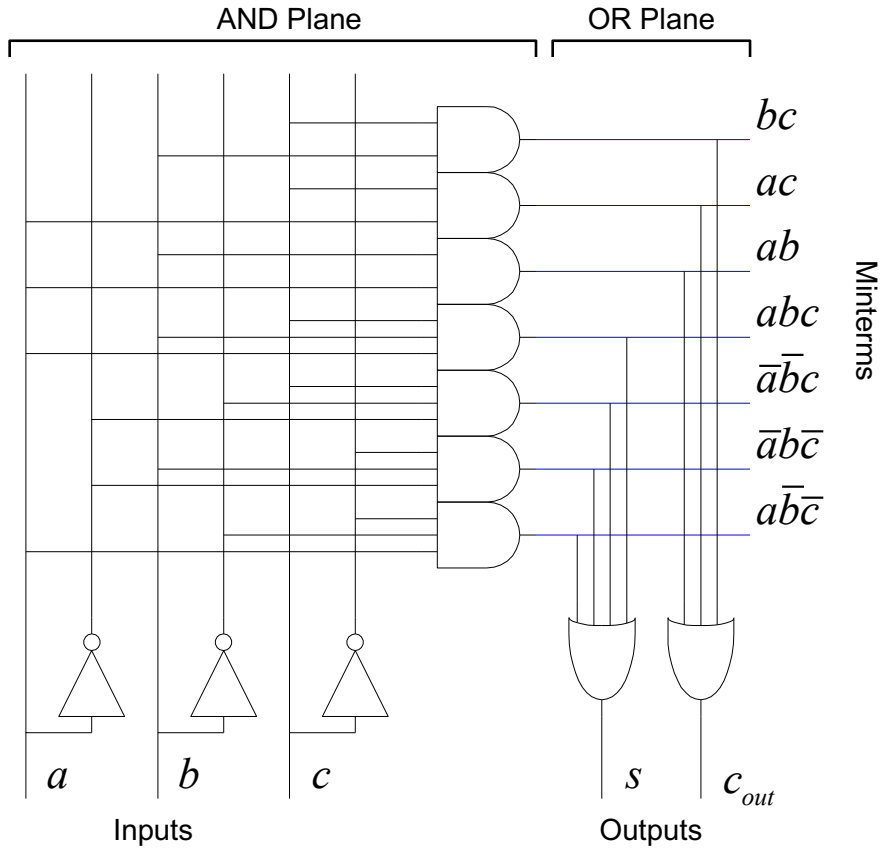
PLAs

- ***A Programmable Logic Array*** **performs any function in sum-of-products form.**
- ***Literals*****: inputs & complements**
- ***Products / Minterms*****: AND of literals**
- ***Outputs*****: OR of Minterms**

- **Example: Full Adder**

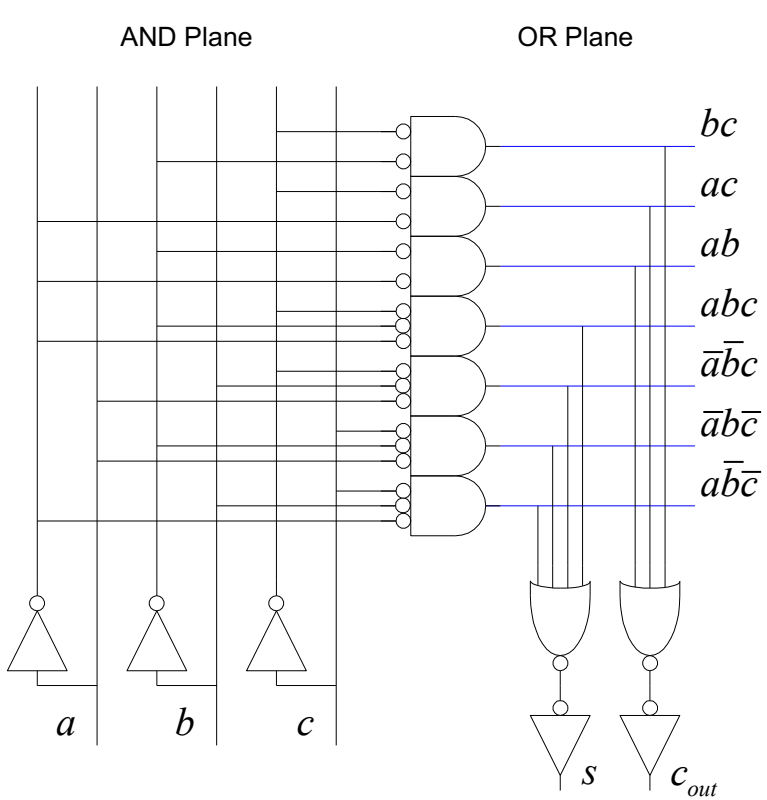
$$s = a\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc$$

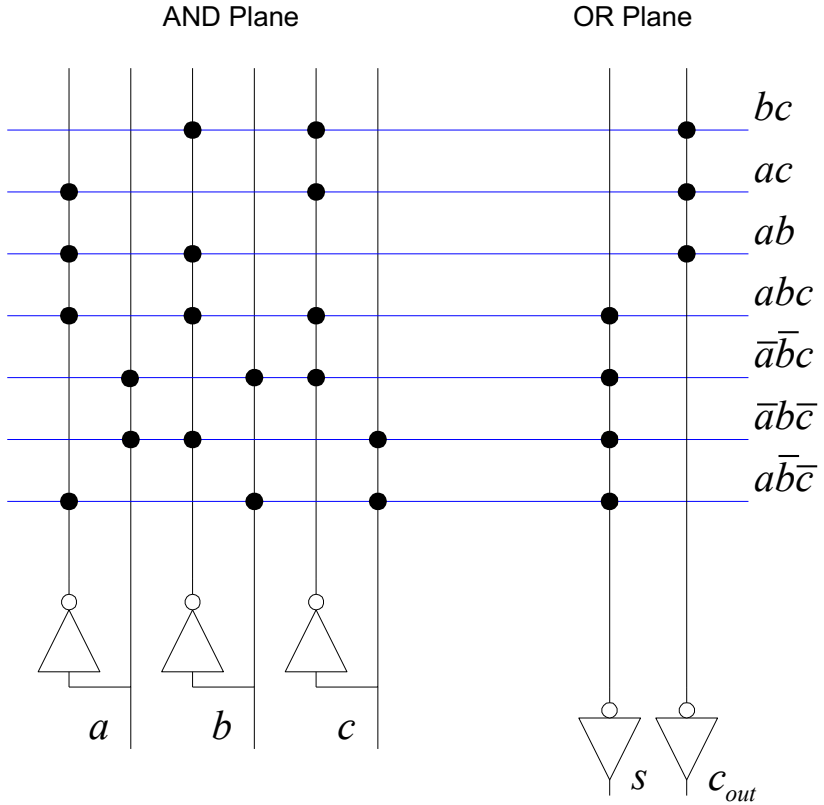
$$c_{\text{out}} = ab + bc + ac$$



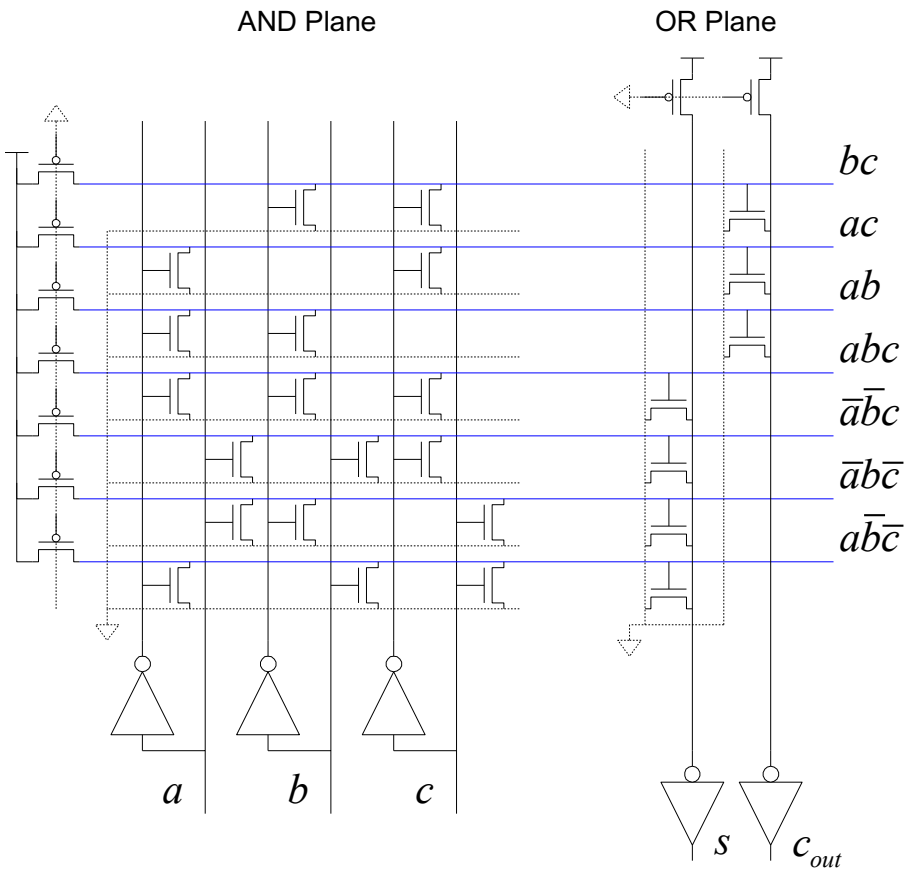
NOR-NOR PLAs

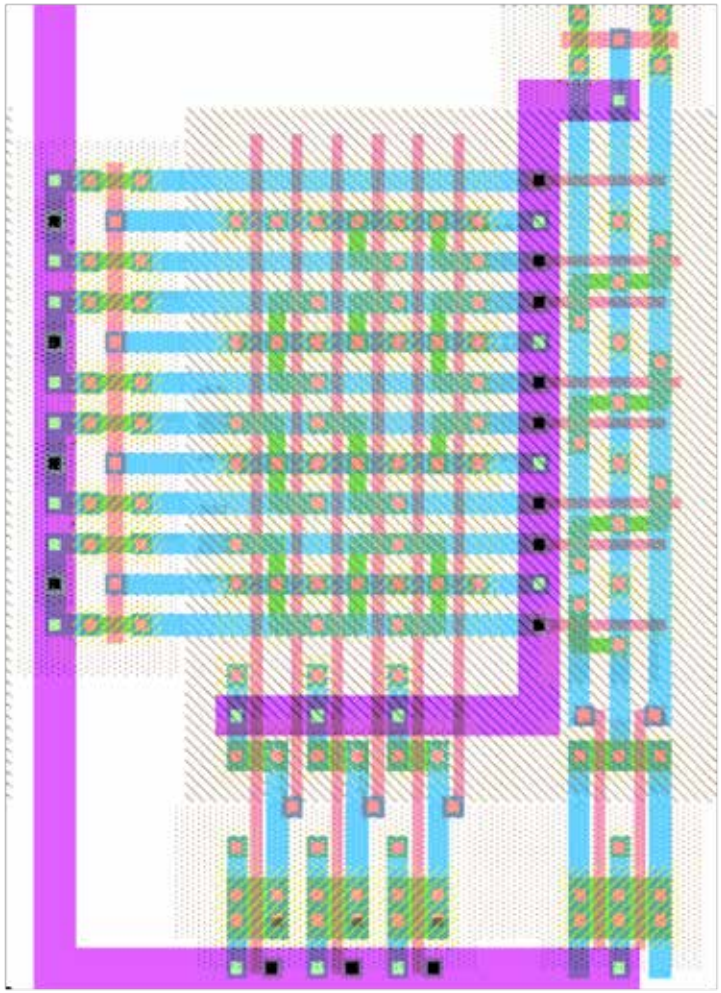
- ANDs and ORs not very efficient in CMOS
- Dynamic or Pseudo-nMOS NORs very efficient
- Use DeMorgan's Law to convert to all NORs





PLA Schematic & Layout





PLAs vs. ROMs

- The NOR plane of the PLA is like the ROM array
- The NAND plane of the PLA is like the ROM decoder
- PLAs are more flexible than ROMs
  - No need to have 2^n rows for n inputs
  - Only generate the minterms that are needed
  - Take advantage of logic simplification

Example: RoboAnt PLA

Convert state transition table to logic

$S_{1:0}$	L	R	$S_{1:0}'$	TR	TL	F
00	0	0	00	0	0	1
00	1	X	01	0	0	1
00	0	1	01	0	0	1
01	1	X	01	0	1	0
01	0	1	01	0	1	0
01	0	0	10	0	1	0
10	X	0	10	1	0	1
10	X	1	11	1	0	1
11	1	X	01	0	1	1
11	0	0	10	0	1	1
11	0	1	11	0	1	1

S1′ S_1S_0

LR \ S_1S_0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	1
10	0	0	0	1

$$S_1' = S_1\overline{S_0} + \overline{L}S_1 + \overline{LR}S_0$$

S0′ S_1S_0

LR \ S_1S_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	1	0

$$S_0' = R + L\overline{S_1} + LS_0$$

$$TR = S_1\overline{S_0}$$

$$TL = S_0$$

$$F = S_1 + \overline{S_0}$$

RoboAnt Dot Diagram

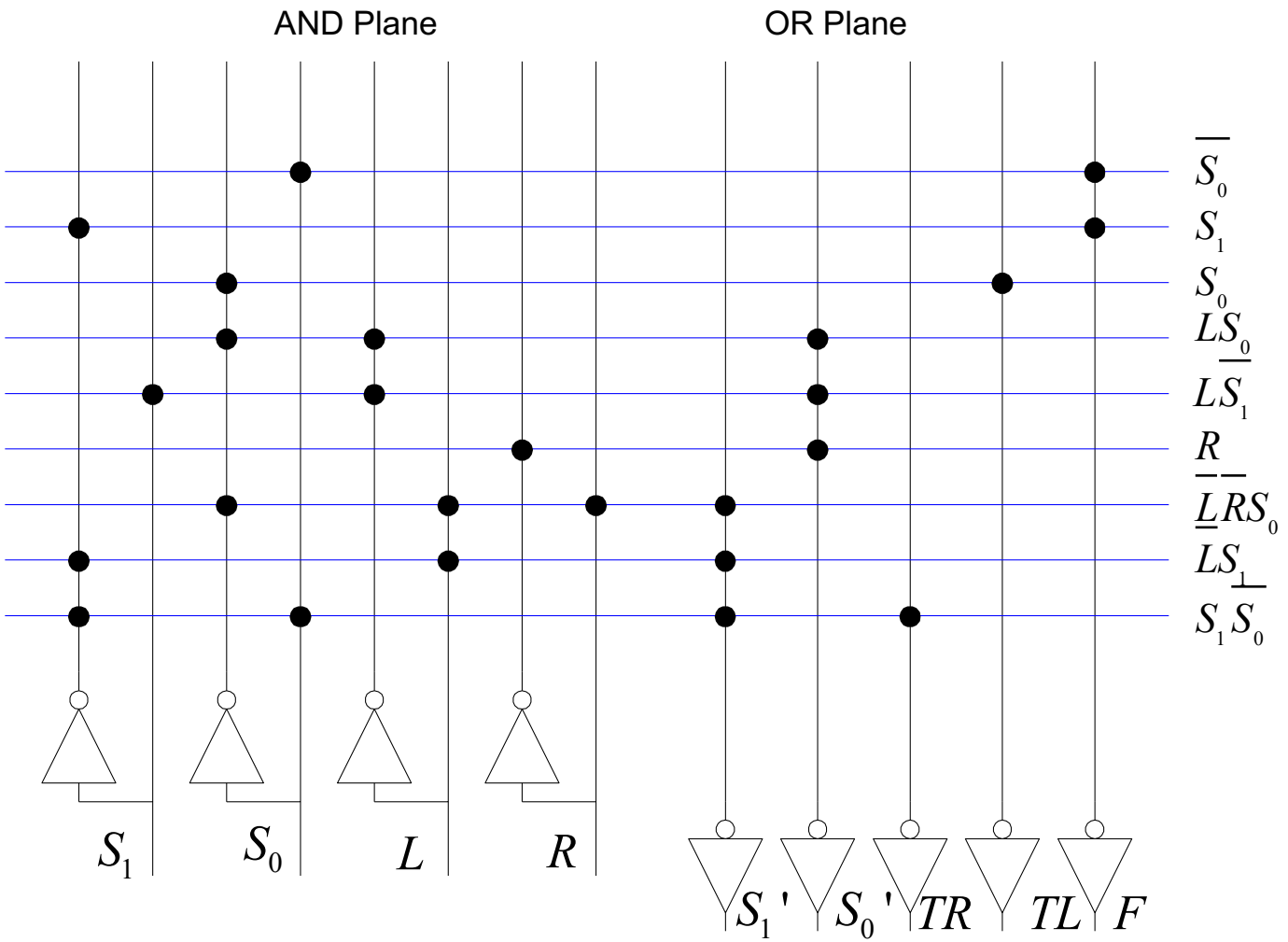
$$S1' = S_1\overline{S_0} + \overline{L}S_1 + \overline{L}\,\overline{R}S_0$$

$$S0' = R + L\overline{S_1} + LS_0$$

$$TR = S_1\overline{S_0}$$

$$TL = S_0$$

$$F = S_1 + \overline{S_0}$$

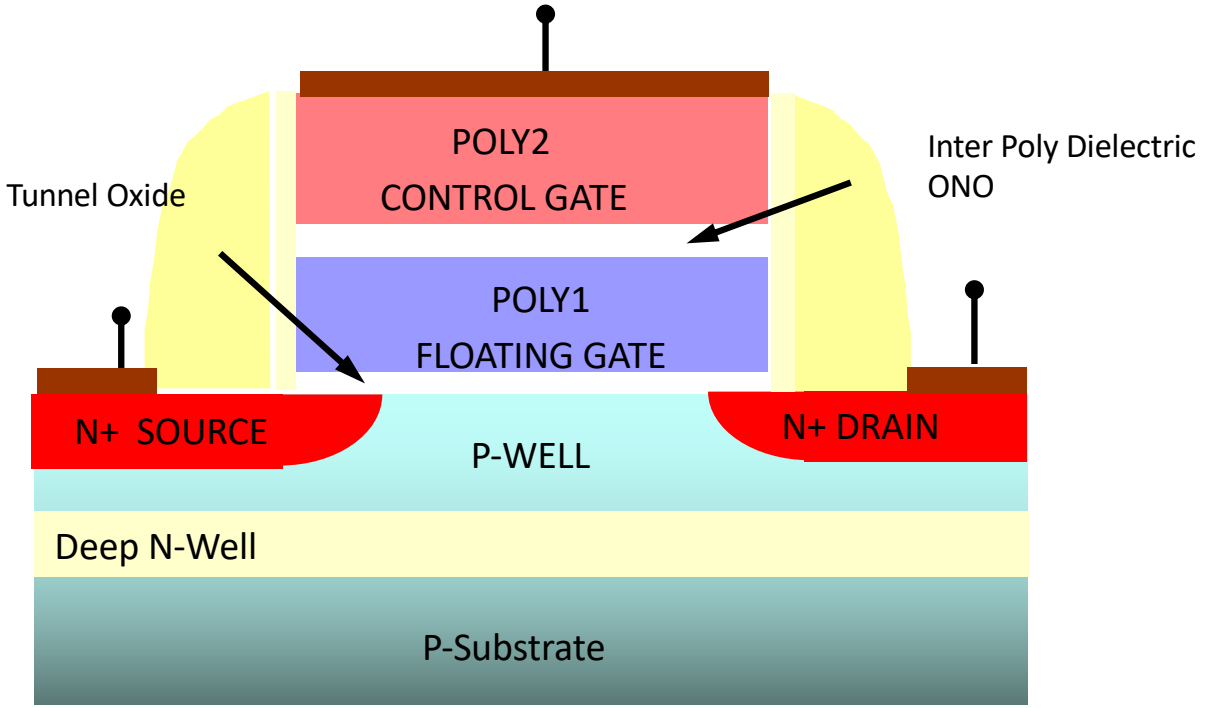


PROMs and EPROMs

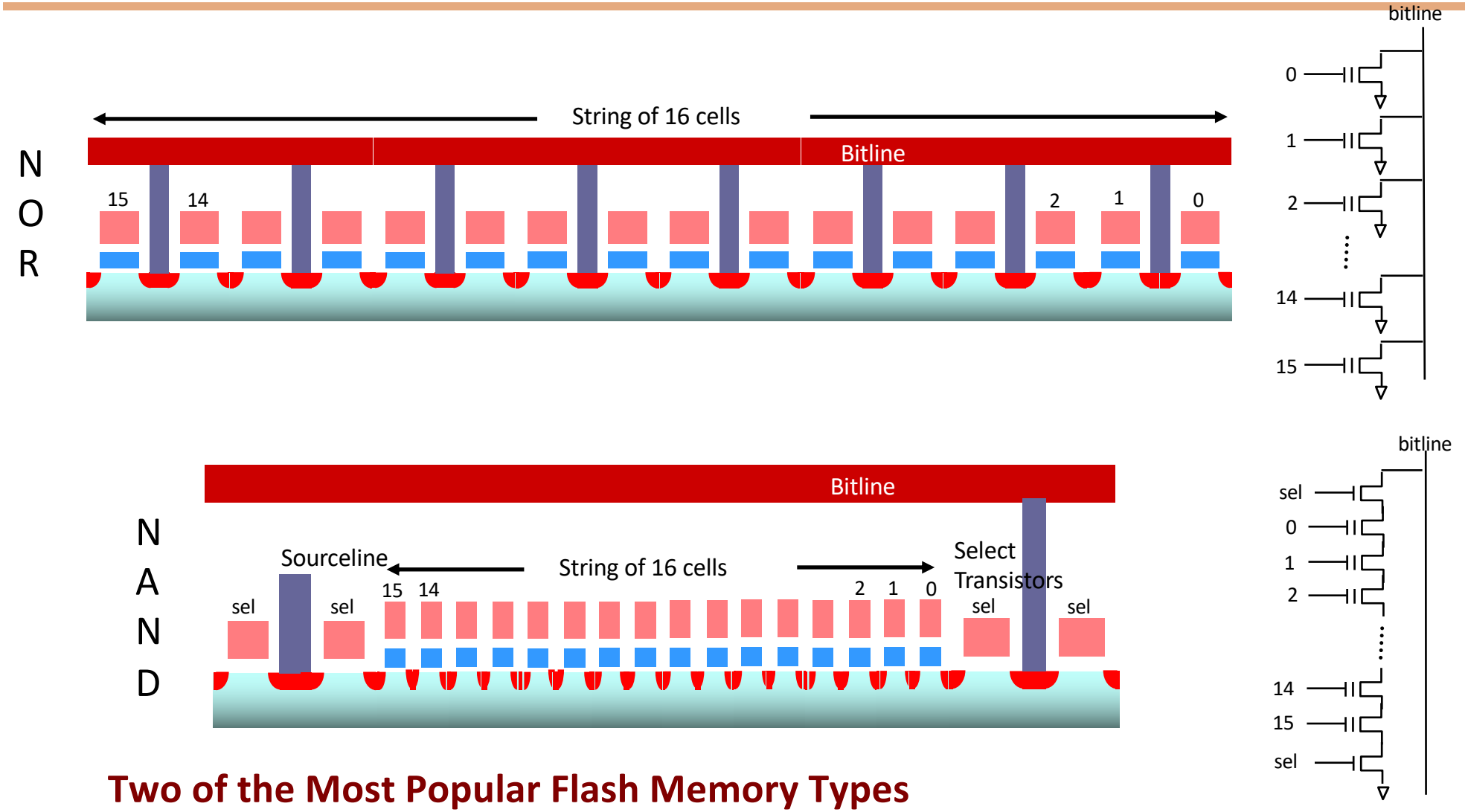
- **Programmable ROMs**
  - Build array with transistors at every site
  - Burn out fuses to disable unwanted transistors
- **Electrically Programmable ROMs**
  - Use floating gate to turn off unwanted transistors
  - EPROM, EEPROM, Flash

Stacked Gate NMOS Transistor

- Poly1 Floating Gate for charge storage
- Poly2 Control Gate for accessing the transistor
- Tunnel-oxide for Gate oxide
- Oxide-Nitride-Oxide (ONO) for the inter Poly Dielectric
- Source/Drain Junctions optimized for Program/Erase/Leakage



NAND vs. NOR Cross-sections

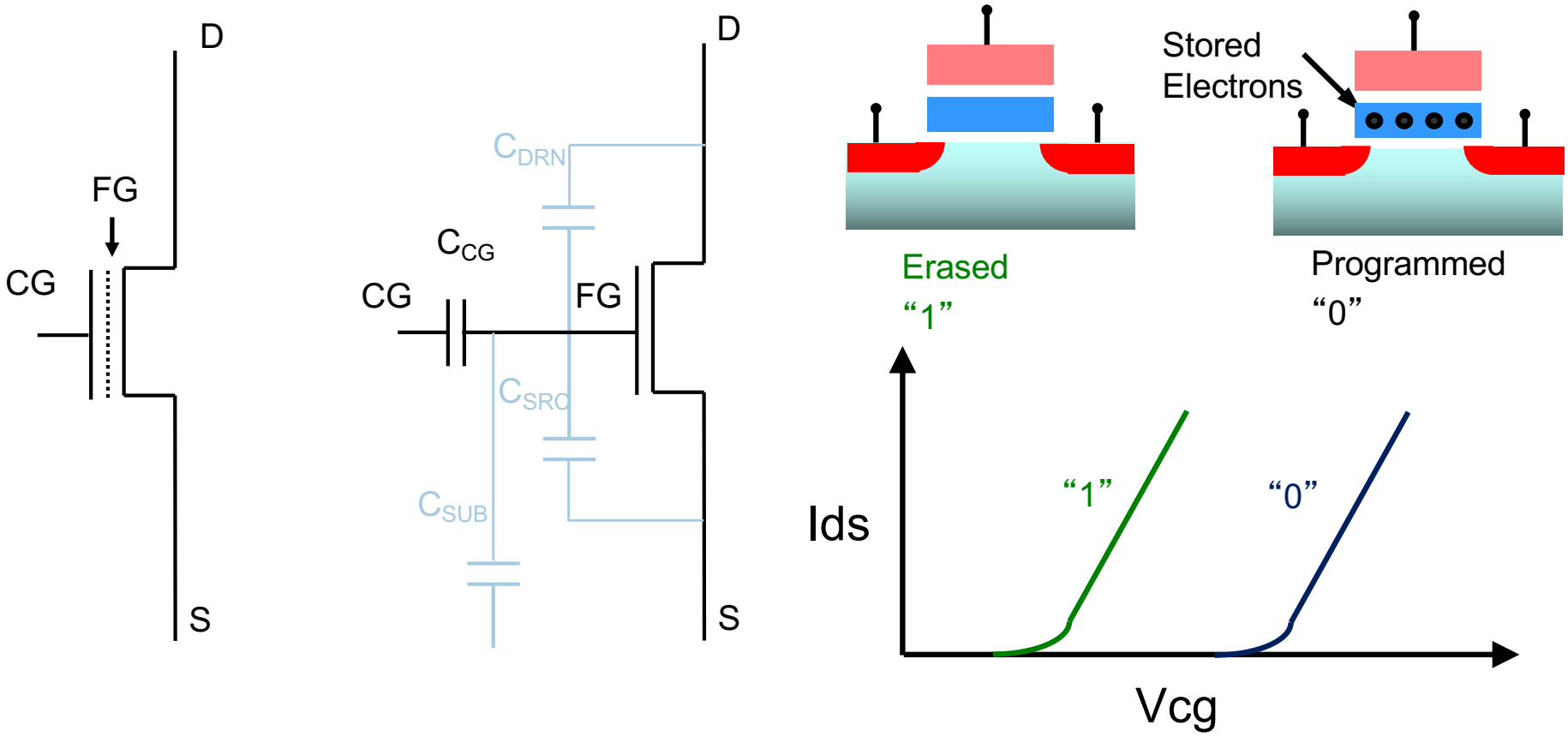


Two of the Most Popular Flash Memory Types

- Both have Dual Gate NMOS with charge storage in Poly1 floating gate
- Lack of contacts in NAND cell makes it inherently smaller in size

Courtesy Intel

Flash Memory Device – Basic Operation



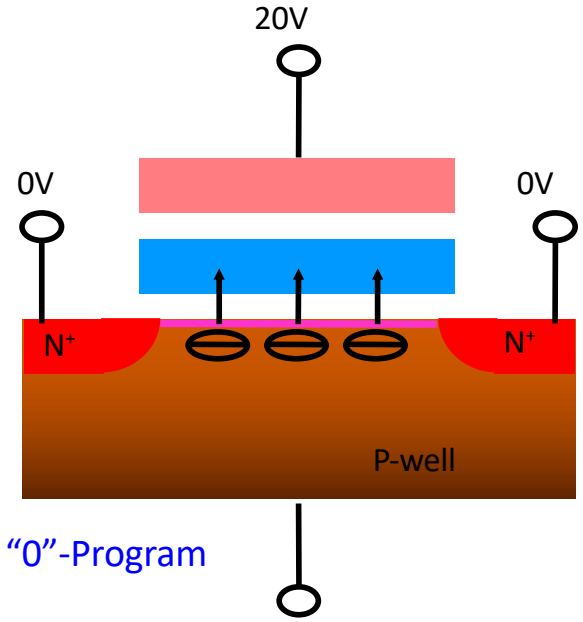
Programming = Electrons Stored on the FG = High Vt

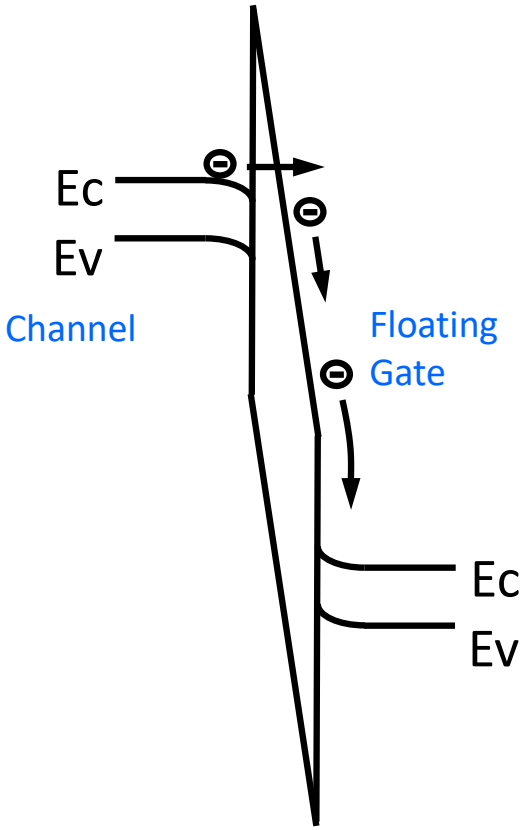
Erasing = Remove electrons from the FG = Low Vt

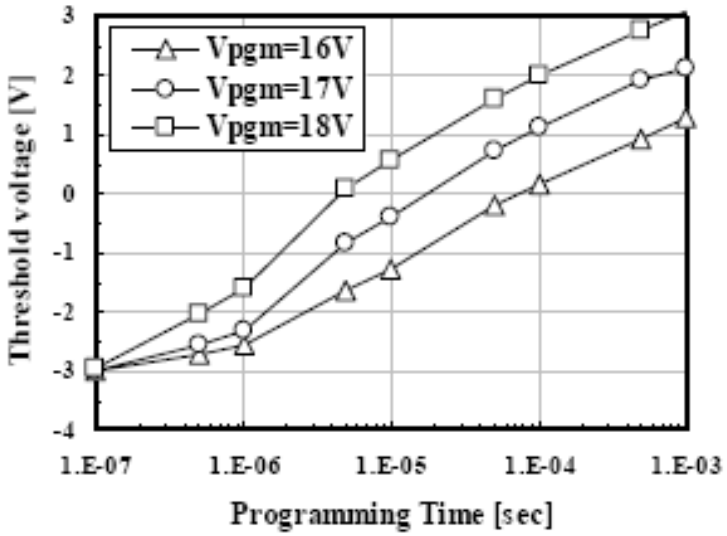
Threshold Voltage shift = $\Delta Q_{FG}/C_{CG}$

Courtesy Intel

Nand Flash Programming – FN Tunneling





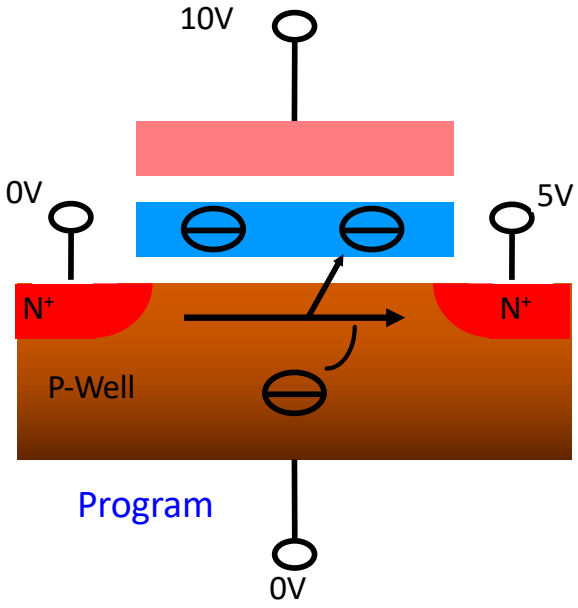


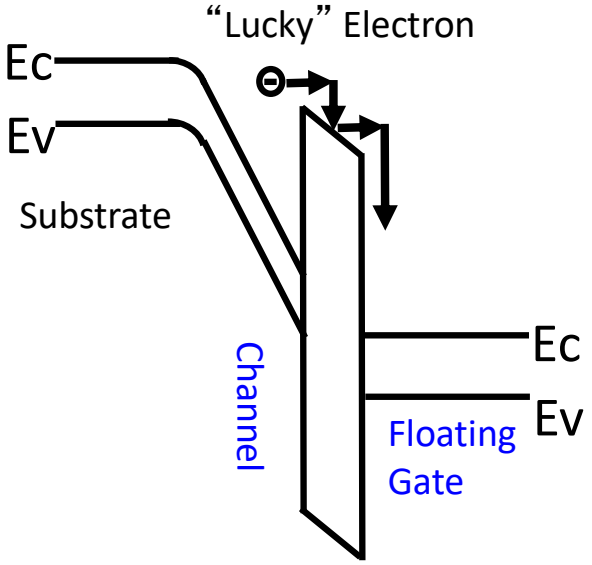
Y. S. Yim, et al. IEDM 2003

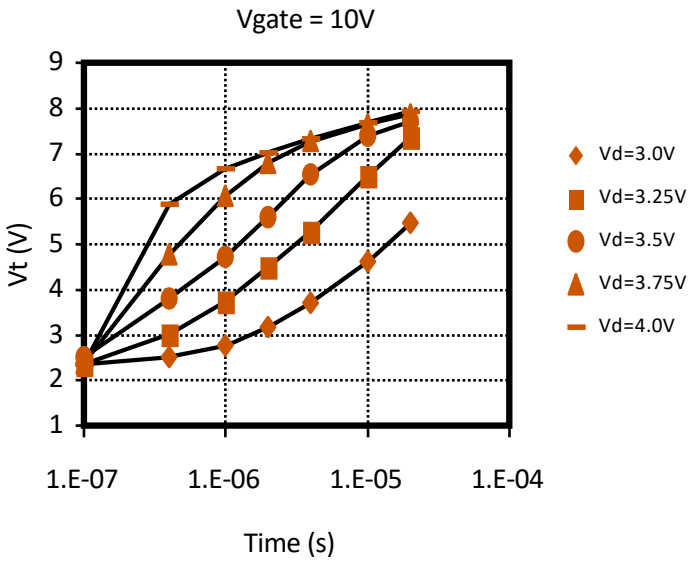
- Tunnel Programming from channel by biasing the Top Gate positive with respect to the ground
- Program Time ~300us
- Program current ~ Displacement plus Tunneling current. Low current allows large parallelism.

Courtesy Intel

NOR Flash Programming – Channel Hot Electron







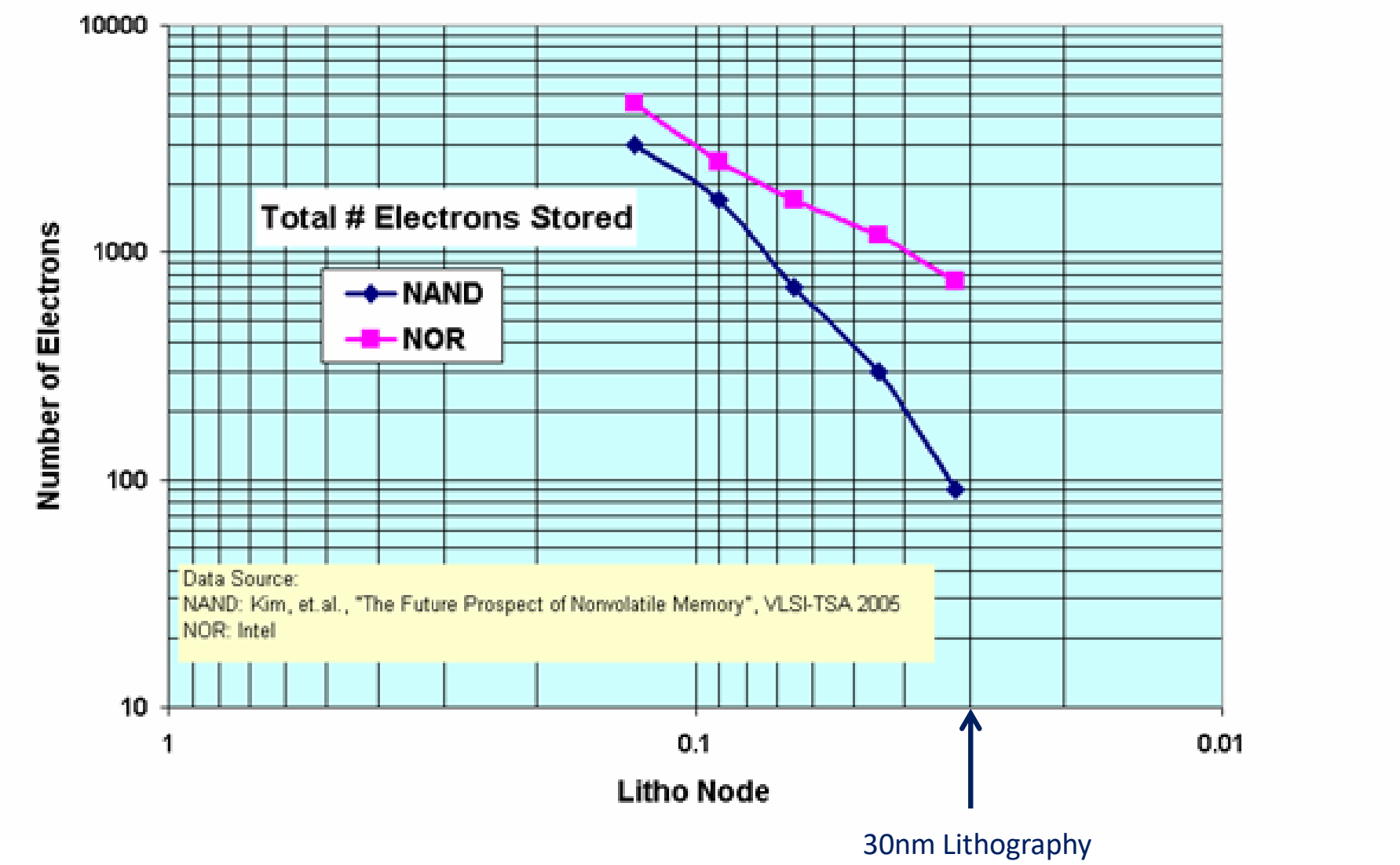
Channel Hot Electron Programming - Gate voltage inverts channel; drain voltage accelerates electrons towards drain; gate voltage pulls them to the floating gate

In Lucky Electron Model, electron crosses channel without collision, gaining > 3.2eV, hits Si atom, bounces over barrier

Program Time ~ 0.5-1ms. Program current ~50mA/cell

Courtesy Intel

Floating Gate Electrons vs. Lithography



Courtesy Intel