EECS151/251A
Spring 2018
Digital Design and Integrated Circuits

Instructors:
John Wawrzynek and Nick Weaver

Lecture 17: State

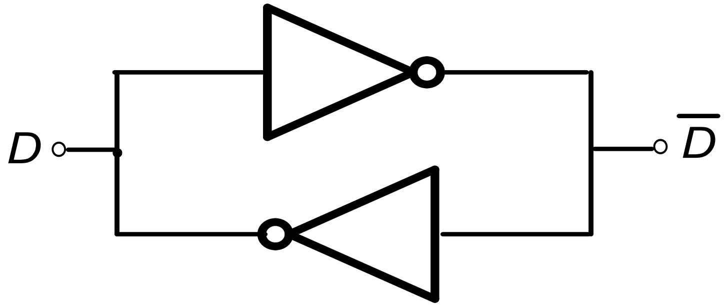
State...

- For digital design, we are always building finite state machines
  - Just the state can be damn large when you include external memory
  - So how do we build state?
- 4 basic state elements:
  - Registers
    - Fast, large, little array overhead
  - SRAM
    - Fairly fast, smaller, significant overhead
  - DRAM
    - Fairly slow, very dense, significant overhead
  - FLASH/EEPROM
    - Glacial, very dense, static, significant overhead

State Requires Storage & Access...

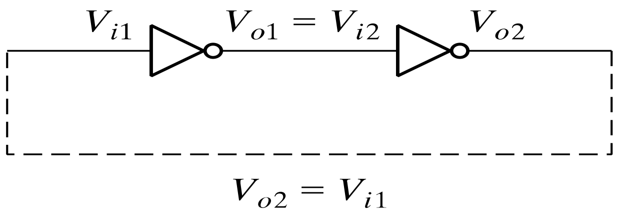
- Either as a feedback loop
  - Latches, Registers, SRAM
  - "Static"
- Or as a stored property
  - Electrons on a capacitor for DRAM
  - Electrons injected into a floating gate for FLASH
- Feedback loop...
  - Can just read the output directly (but there are tricks to speed it up)
- Stored property...
  - ***Need*** measurement circuits

Basic Static Storage Element

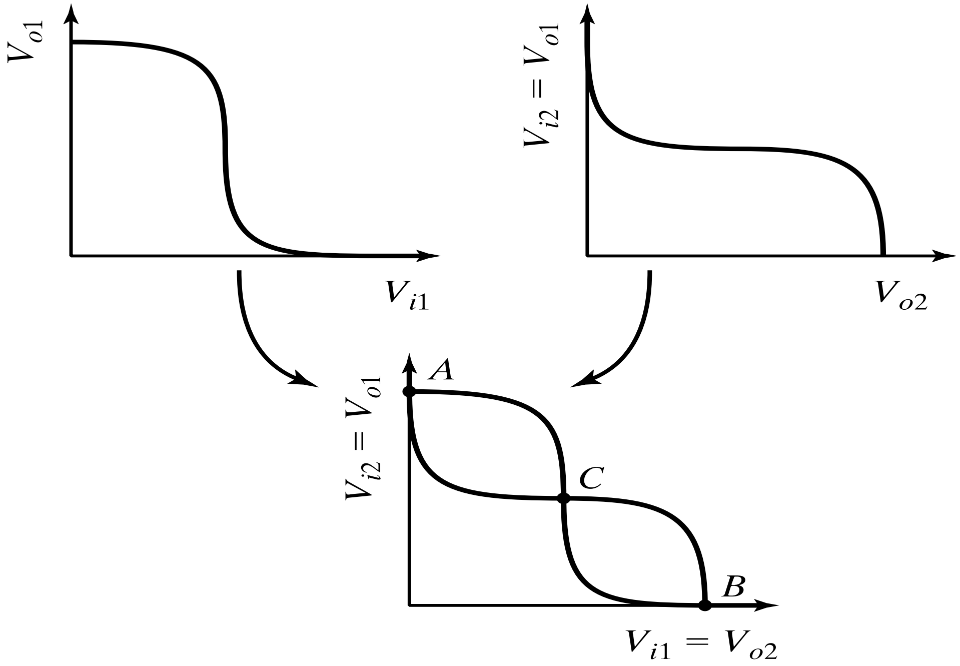


- If D is high, D_b will be driven low
  - Which makes D stay high
- Positive feedback

Positive Feedback: Bi-Stability

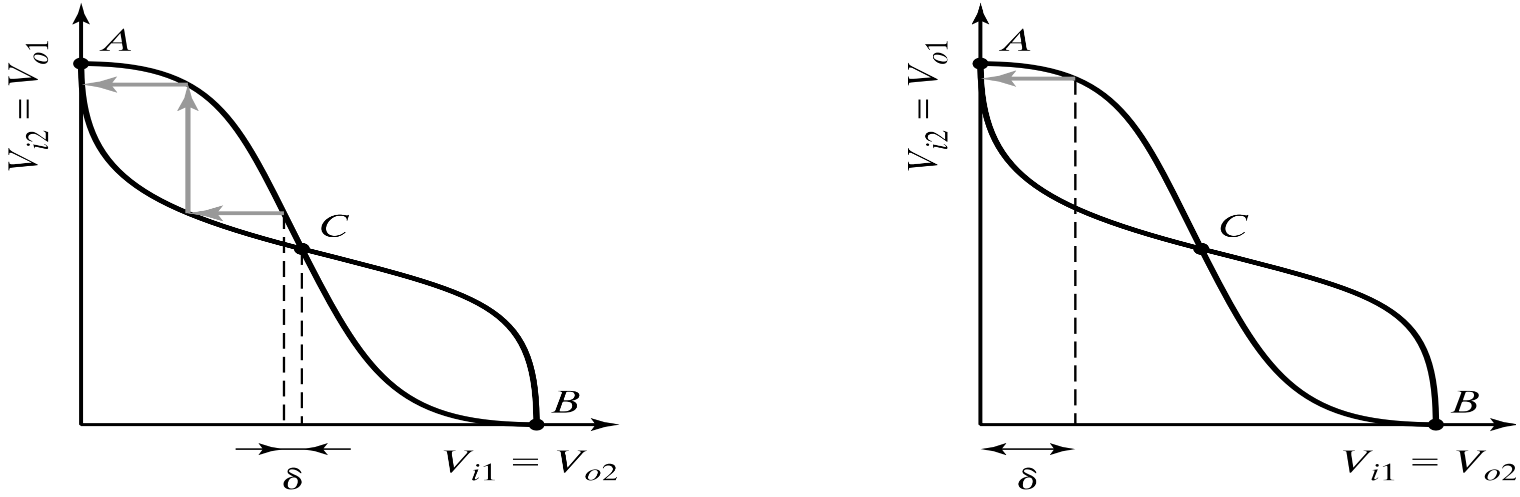


(a)



(b)

Meta-Stability



(a)

(b)

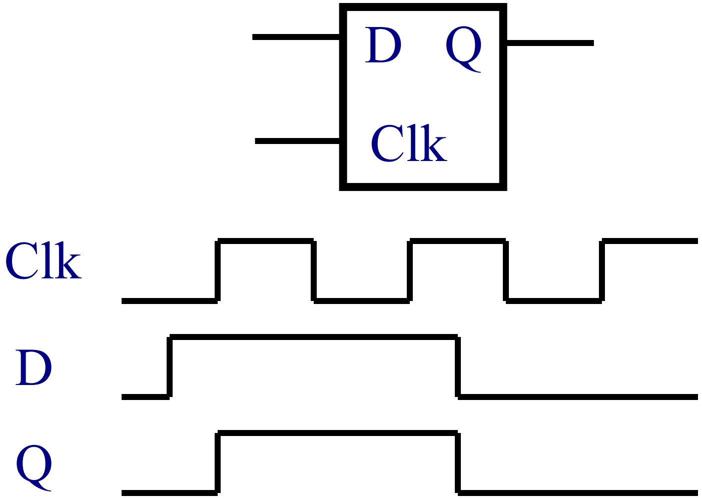
Gain should be larger than 1 in the transition region:

For flip-flops we hate metastability...

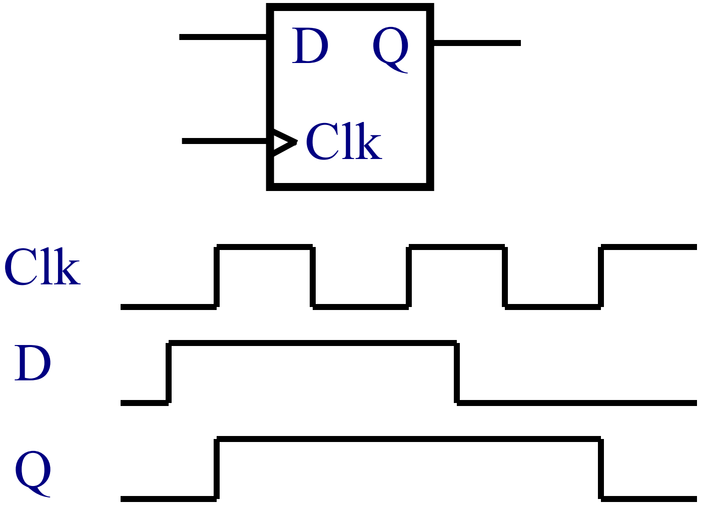
But for DRAM we love it (as we will see soon)!

Reminder: Latch versus Register (Flip-flop)

- Latch: level-sensitive
  clock is low - hold mode
  clock is high - transparent

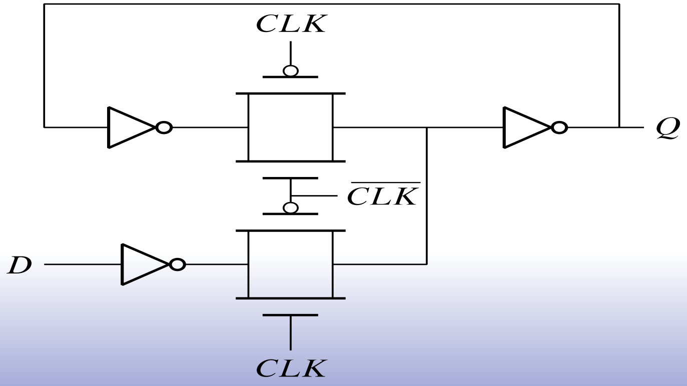


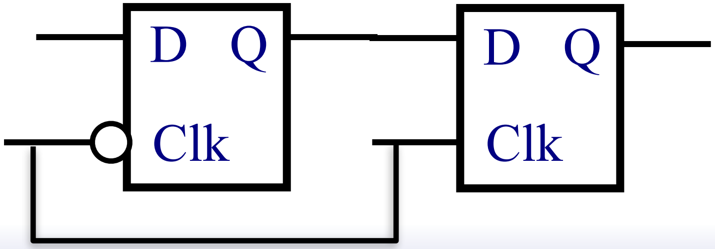
- Register: edge-triggered
  stores data when
  clock rises



Creating a Latch & Flip-Flop...

- Take our basic static storage element...
  - And make it switchable between storage
- When clock is low, have the feedback loop active
- When lock is high, Q gets D
- Can then chain 2 latches to make a Flip/Flop
  - One negative clocked, one positive





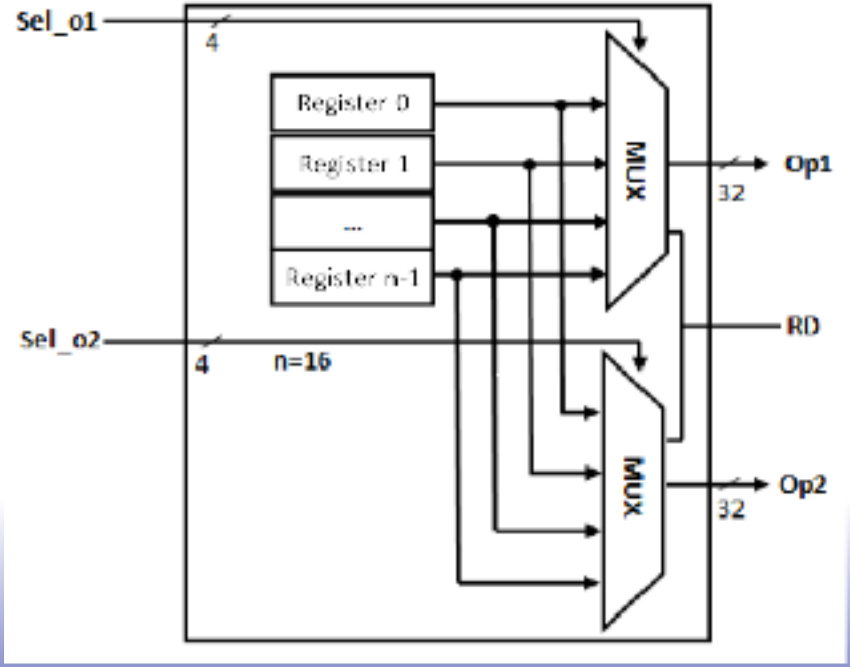
Adding an Enable

- A very common motif
  - `always @posedge clk begin`

    ```
        q := en ? {something} : q
    ```
  - Namely, only update Q if an additional enable signal is high
- Just simply add a higher level feedback loop with a mux.
  - {see board}

Creating a Register File... Reading

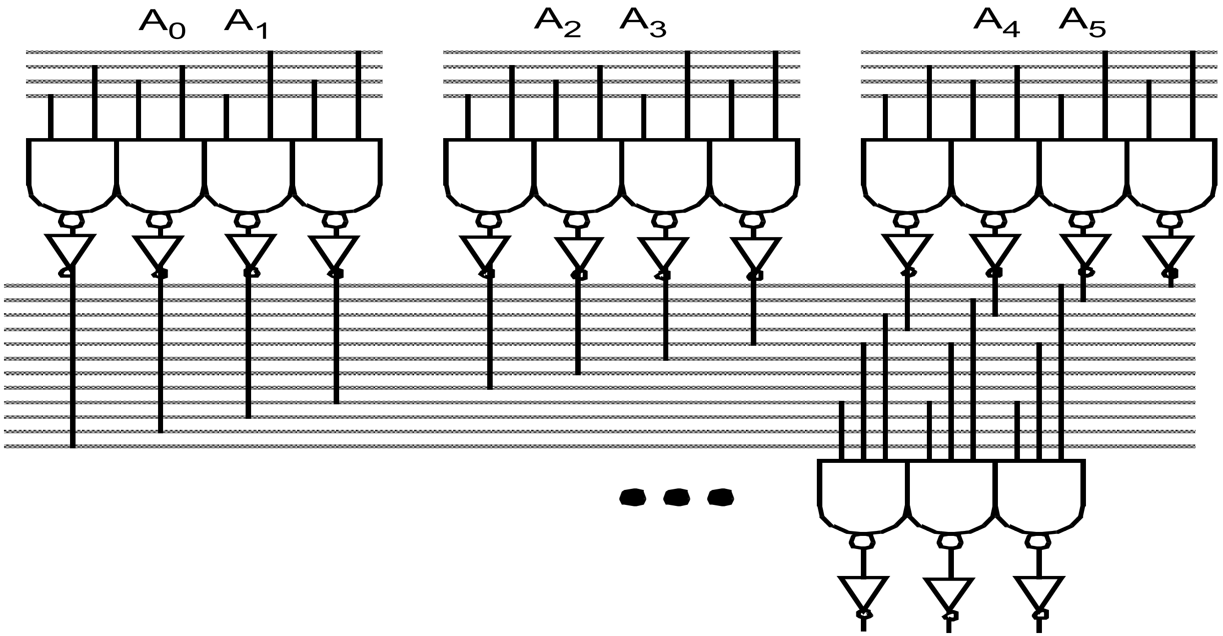
- Easiest thing is to start with a flip-flop...
- For reading, just have a MUX on the output to select the register in question
  - A common motif for all array memories: If you want the nth column/entry, you need a mux to select which one
- If done as flip/flops, this is simple...



Creating A Regfile... Write Enable

- We need to decode/demux the register address to determine which register to write...
  - Decoder is a very common operation:
    Take an ***n*** bit number and assert a different line for each bit
    Basically each output is an AND gate
- Commonly done with a "predecoder"
  - Compute $\overline{A_0}\,\overline{A_1}$, $\overline{A_0}A_1$,$A_0\overline{A_1}$,A_0A_1
  - Compute $\overline{A_2}\,\overline{A_3}$, $\overline{A_2}A_3$,$A_2\overline{A_3}$,A_2A_3
  - Then do the AND of the intermediaries

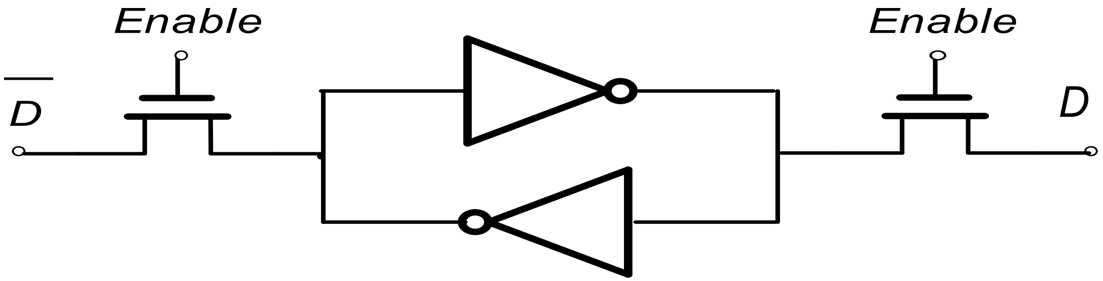
Predecoder and Decoder



But Flip-Flops are big...

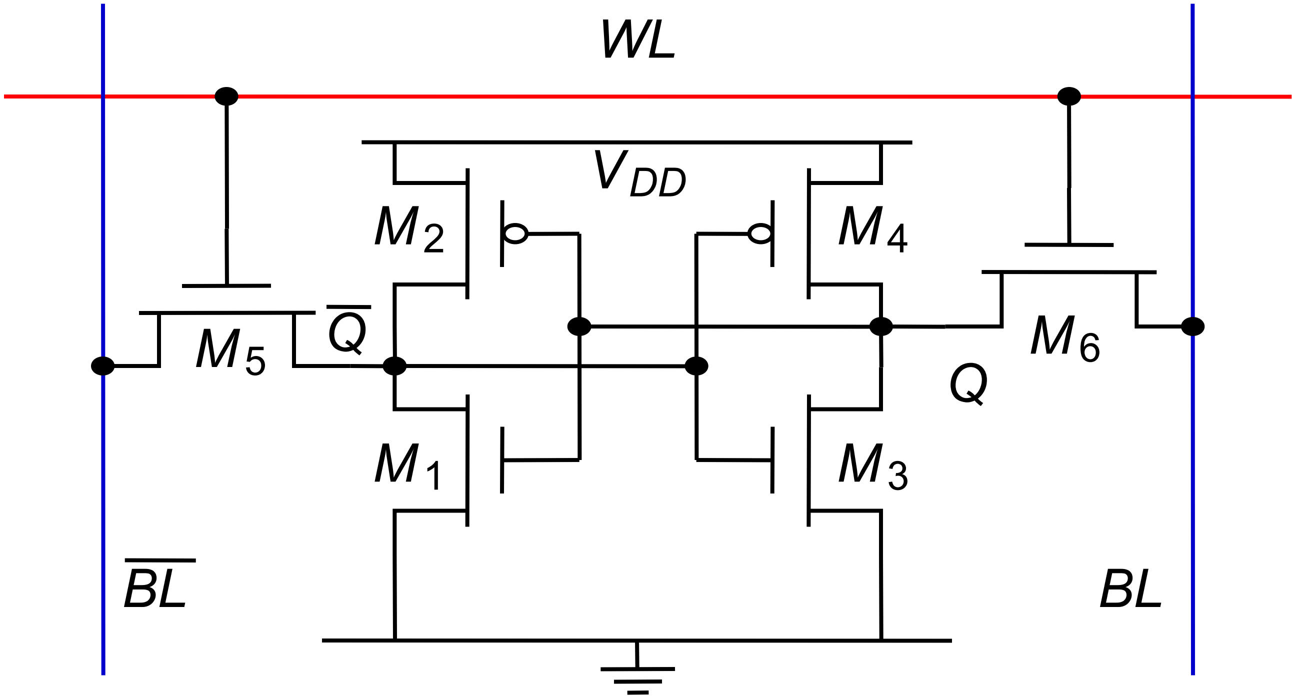
- 12 transistors for both latches...
  - Plus 2 to create a negated clock (can be shared)
- Plus the multiplexor to add an enable...
  - Since we don't want to always write
- Can we use the basic static element...
  - And make something more compact?
- Yes: SRAM

SRAM Memory Cell

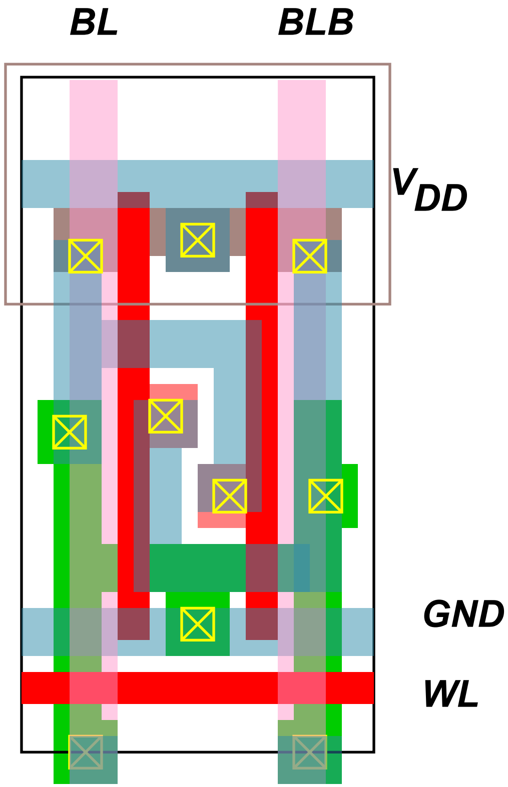


Complementary data values are written (read) from two sides

6-Transistor CMOS SRAM Cell



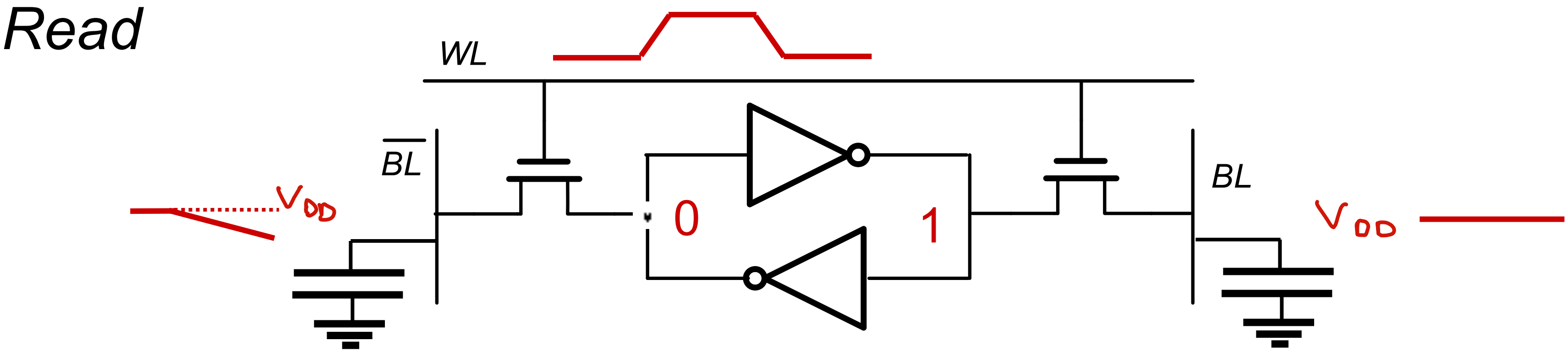
6T-SRAM — Layout



Compact cell

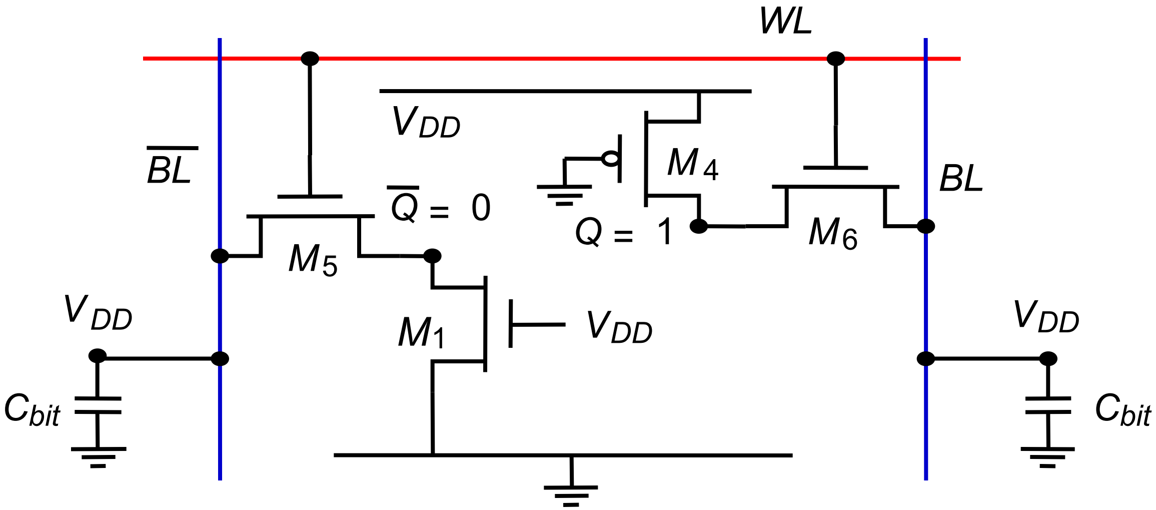
- Bitlines: M2
- Wordline: bootstrapped in M3

SRAM Operation - Read

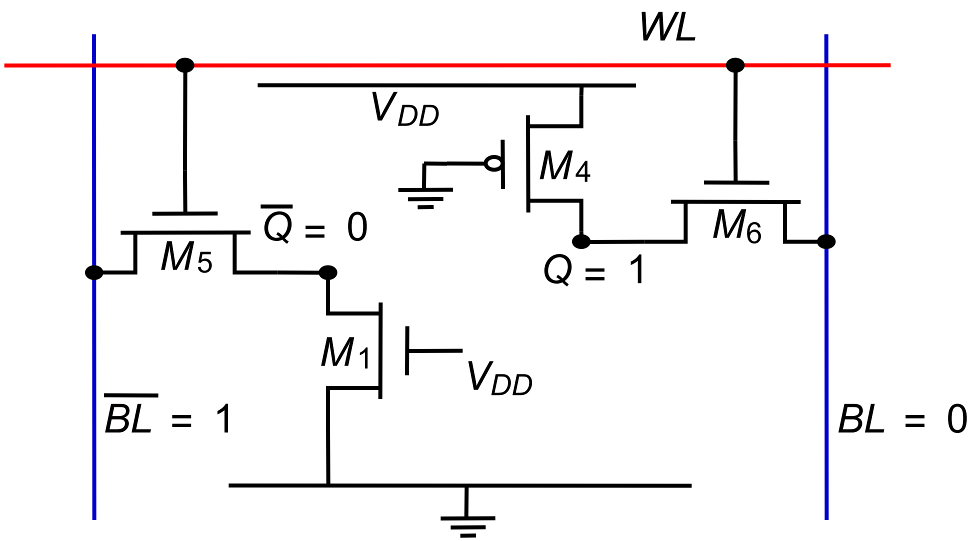


- Initially precharge both the bit lines to VDD
- BL gets pulled down if storing a 1, ~BL if its a 0
  - There is a fair amount of capacitance, must not overwhelm the inverter...

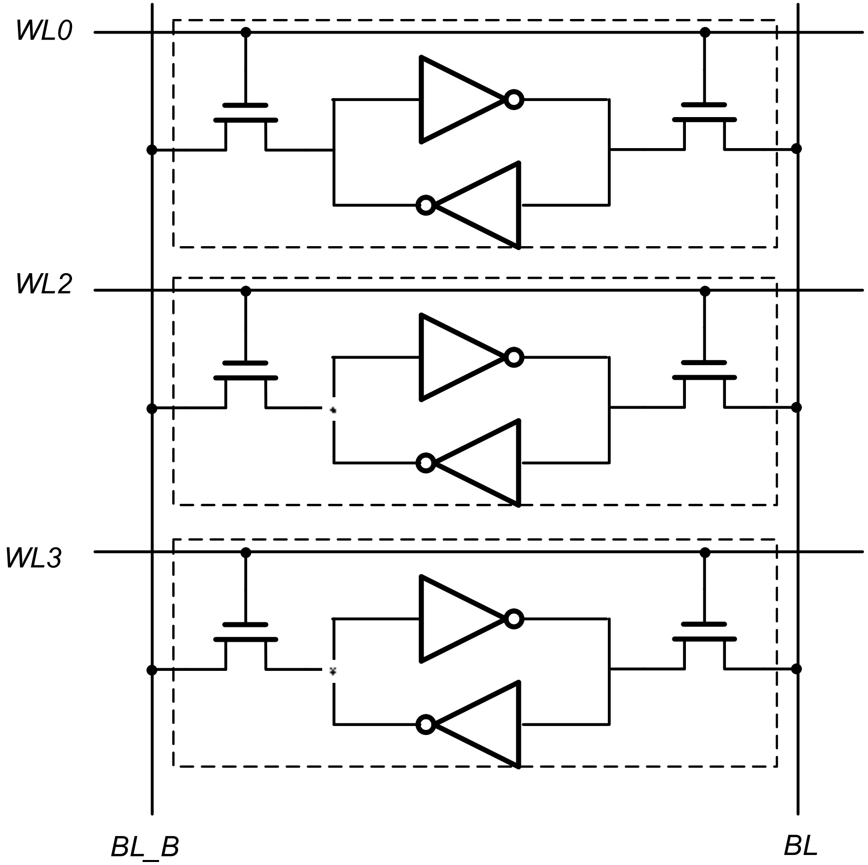
CMOS SRAM Analysis (Read)



CMOS SRAM Analysis (Write is a fight)



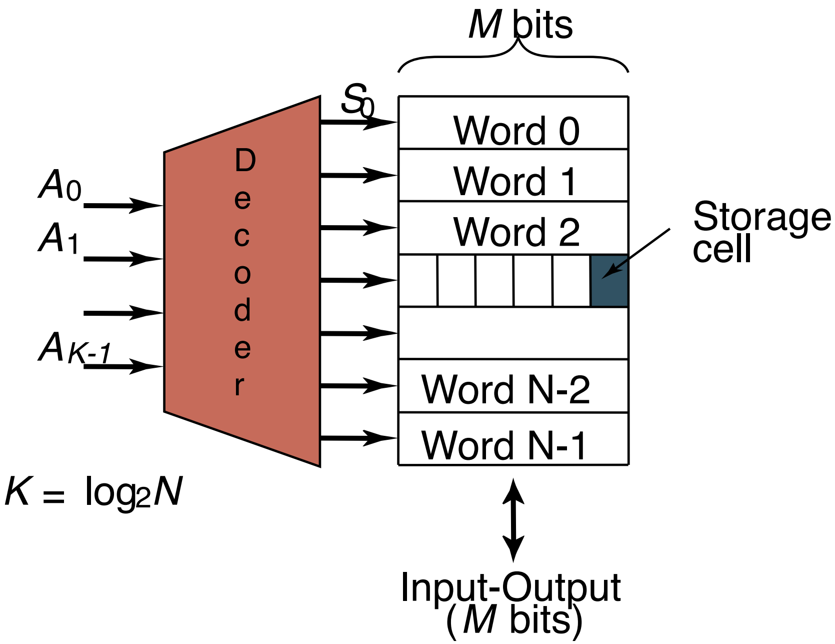
SRAM Column



Periphery

- ❑ Decoders
- ❑ Sense Amplifiers
- ❑ Input/Output Buffers
- ❑ Control / Timing Circuitry

Row Decoder

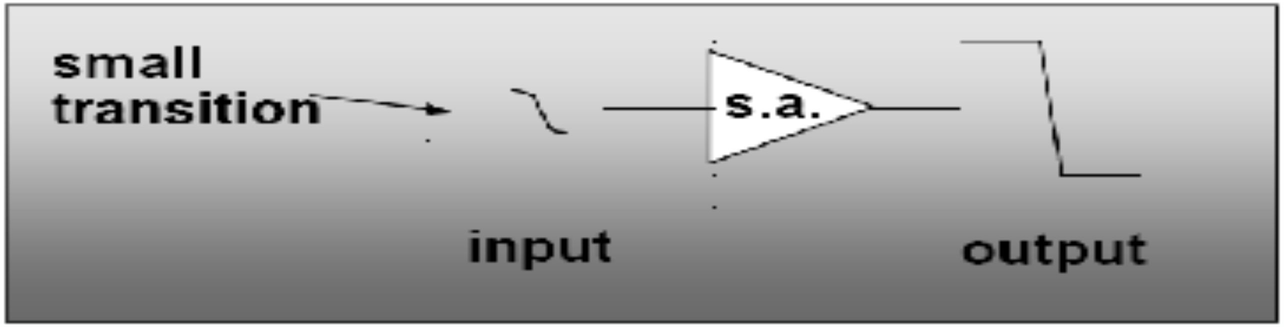


- Expands K address lines into N word lines

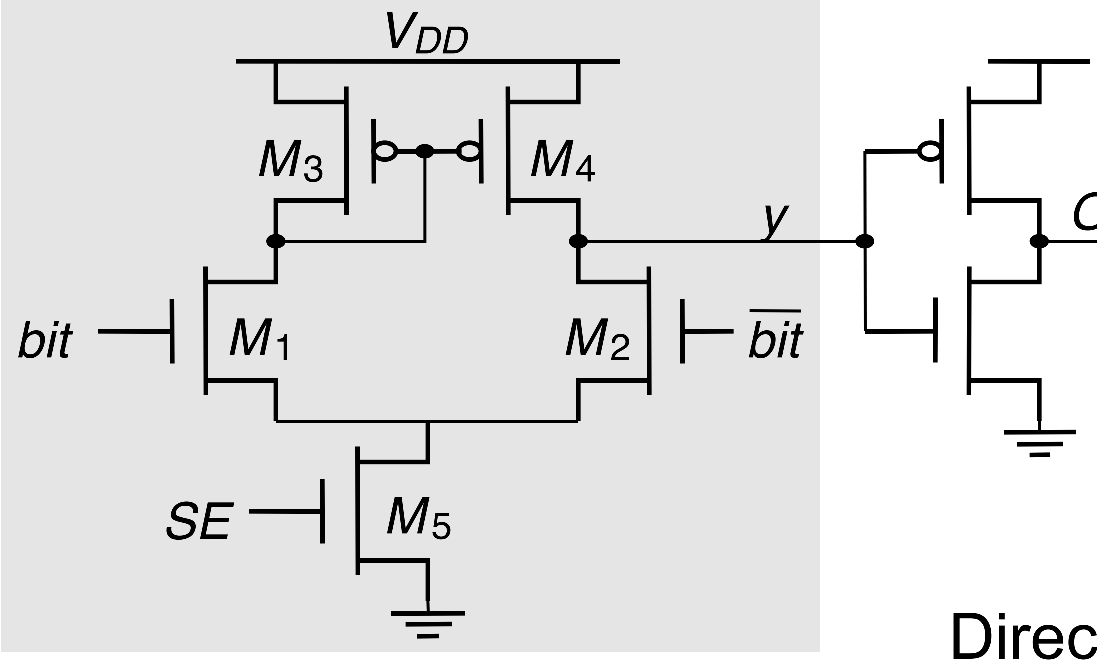
- A perfect example of logic/ wire optimization
- Typically implemented in hierarchical fashion
- Area/Energy Trade-off

Speeding the Read: Sense-Amp

- It takes a fair bit of time to pull down the appropriate bitline...
  - But really, why not just detect a difference
- Idea: Precharge a reference and then look for a small change

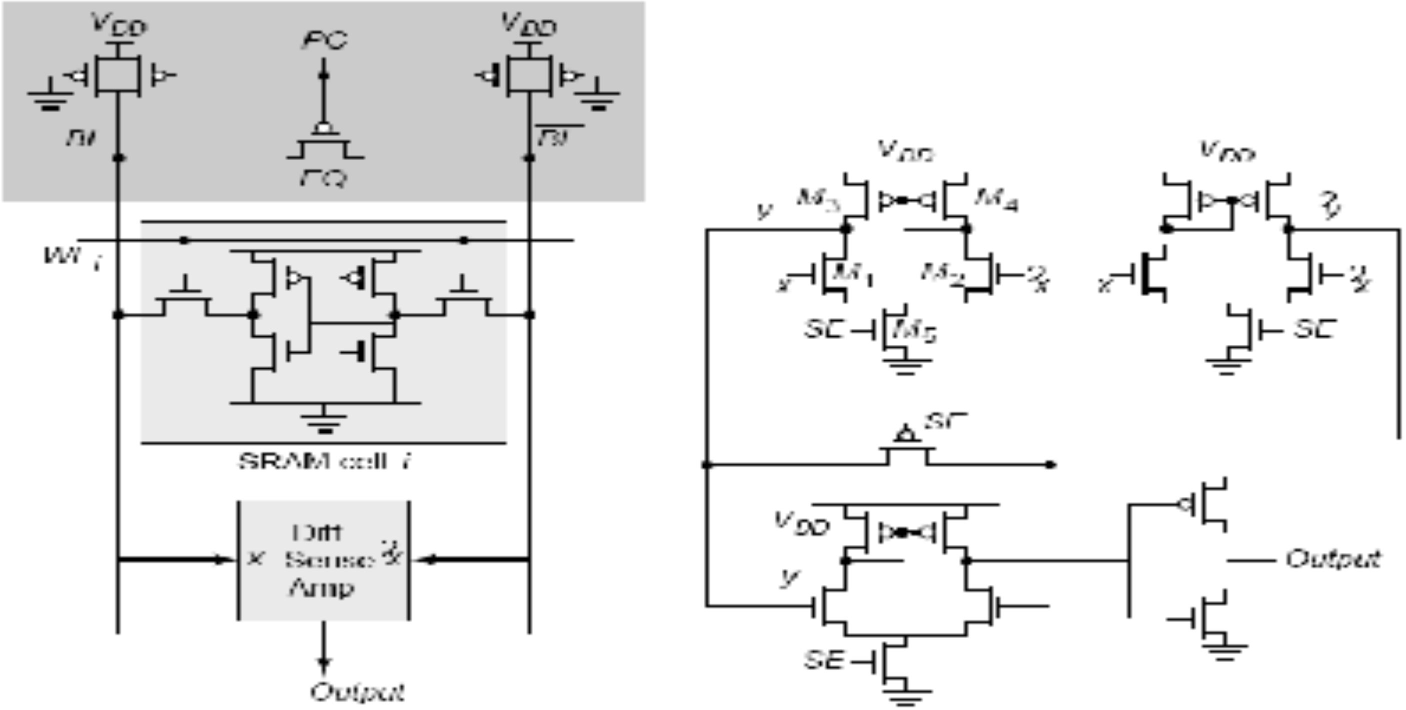


Differential Sense Amplifier



Directly applicable to SRAMs

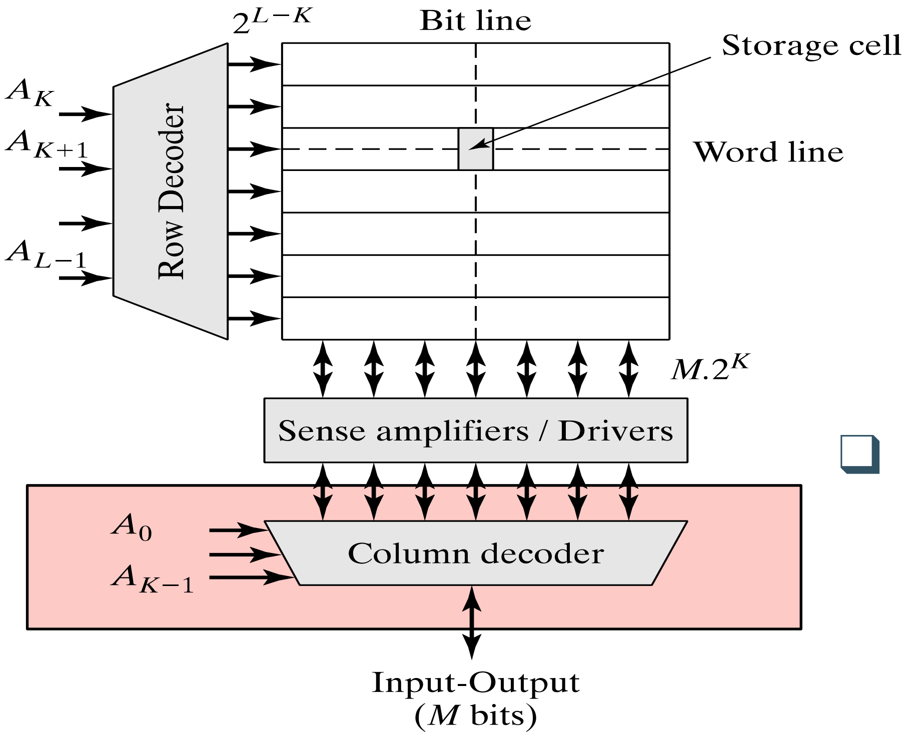
Differential Sensing ― SRAM



(a) SRAM sensing scheme

(b) two stage differential amplifier

Column Decoder



- Basically a multiplexer

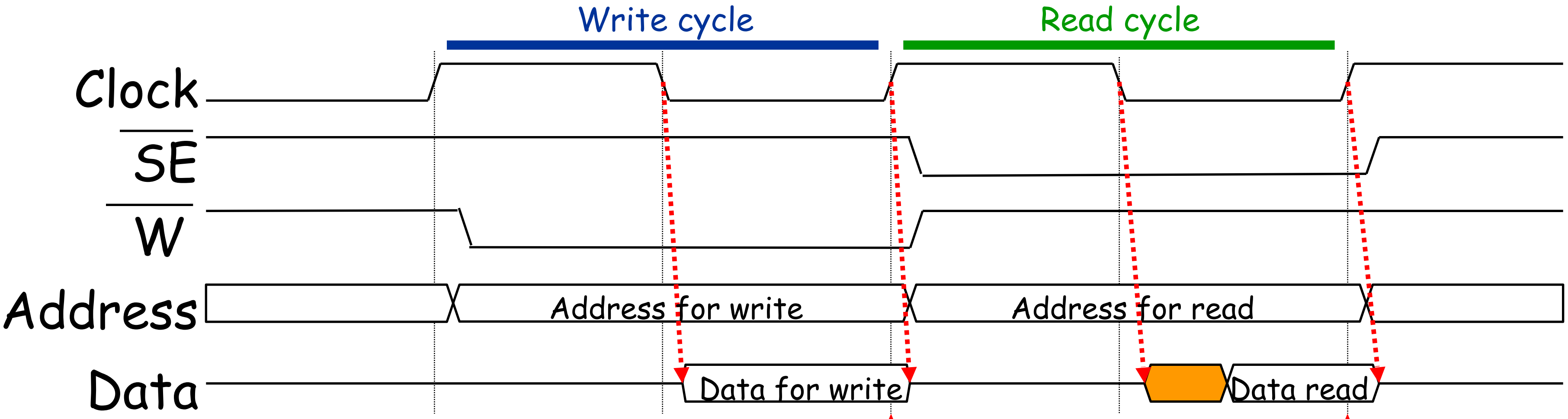
SRAM Read Cycle

- ***Precharge*** all the bitlines to Vdd...
- Then assert the word line from the decoder
  - One or the other bit lines starts to drop towards ground...
  - Sense-amp uses this to amplify the result quickly

SRAM Write Cycle

- Actively drive the bitlines to the correct value
  - Then assert the correct word line
- Result is that it will flip the data to the correct value

Sample Memory Interface Logic

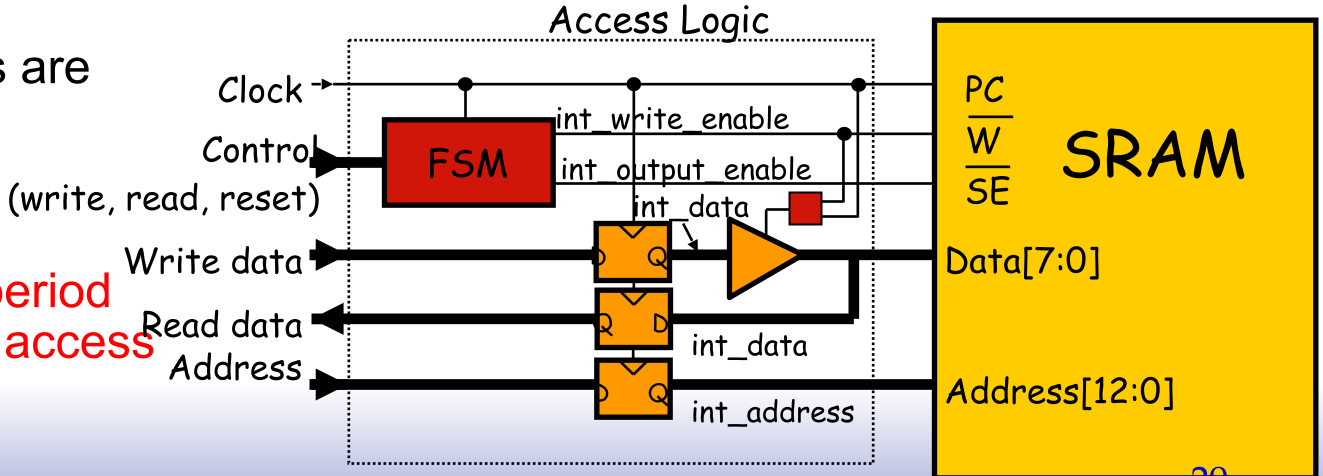


Write occurs here, when E1 goes high

Data can be latched here

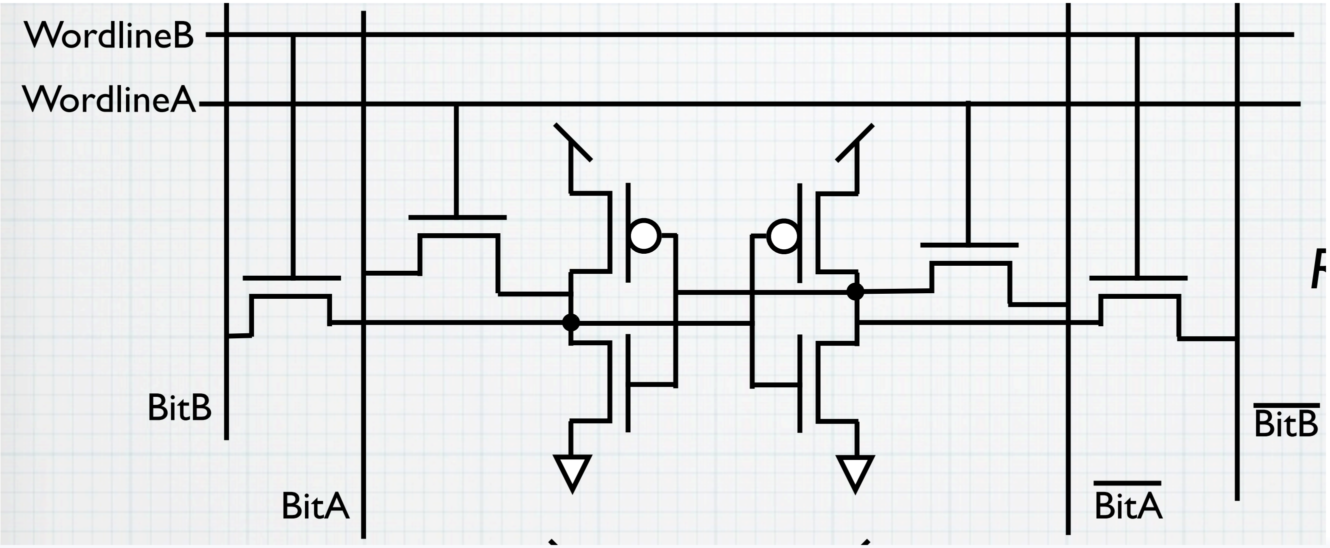
Drive data bus only when clock is low

- Ensures address are stable for writes
- Prevents bus contention
- Minimum clock period is twice memory access time



Adding More Ports

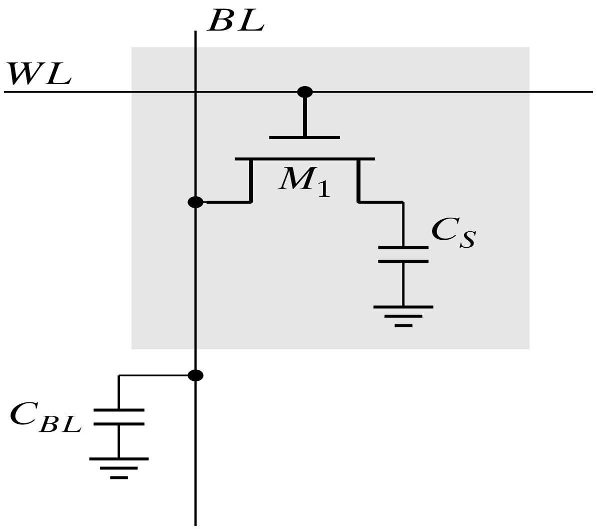
- Just add a separate set of row selection decoders, bitlines, wordlines, sense-amps...
  - But keep the common cell

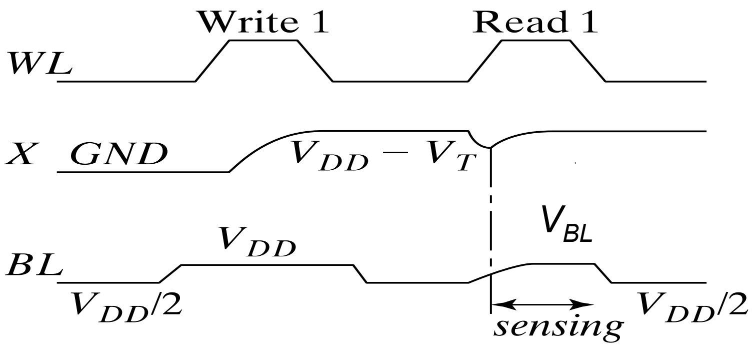


DRAM...

- Damn, that SRAM cell is still big...
  - 6 transistors!!!!
- But we can also build small capacitors...
  - And small capacitors can hold their charge...
- IDEA: Lets store using ***dynamic*** logic...
  - Memory as state of change on a capacitor
- Sense a very small amount of charge...
  - And when we check it, we can then update it

1-Transistor DRAM Cell





$V_{BIT} = 0$ or $(V_{DD} – V_T)$

Write: C_S is charged or discharged by asserting WL and BL.
Read: Charge redistribution takes places between bit line and storage capacitance

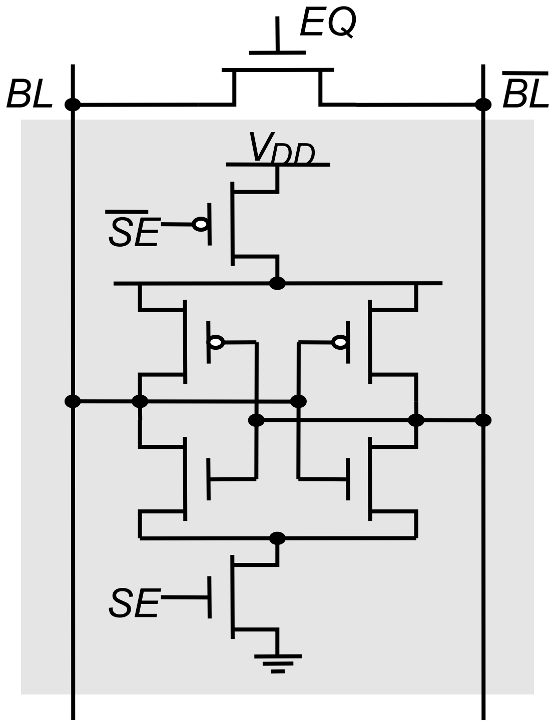
Voltage swing is small; typically around 250 mV.

$$\Delta V_1 = (V_{BIT1} - V_{PRE})\frac{C_S}{C_S + C_{BL}} = \left(\frac{V_{DD}}{2} - V_T\right)\frac{C_S}{C_S + CB_L}$$

$C_S << C_{BL}$

$$\Delta V_0 = (V_{BIT0} - V_{PRE})\frac{C_S}{C_S + C_{BL}} = -\frac{V_{DD}}{2}\frac{C_S}{C_S + CB_L}$$

Latch-Based Sense Amplifier (DRAM)



- Initialized in its meta-stable point with EQ
  - We also precharge the bit lines to the metastable point
- Once adequate voltage gap created, sense amp enabled with SE
- Positive feedback quickly forces output to a stable operating point.
  - Also acts to "write back" the value by driving the bit lines apart

What about that other bitline?

- We want something of the same capacitance as the bitline
  - Idea: Hey, we have a reference: The ***next*** column
- So we only read 1/2 the cells on a row for a given read
  - And use the other set of bit lines for reference

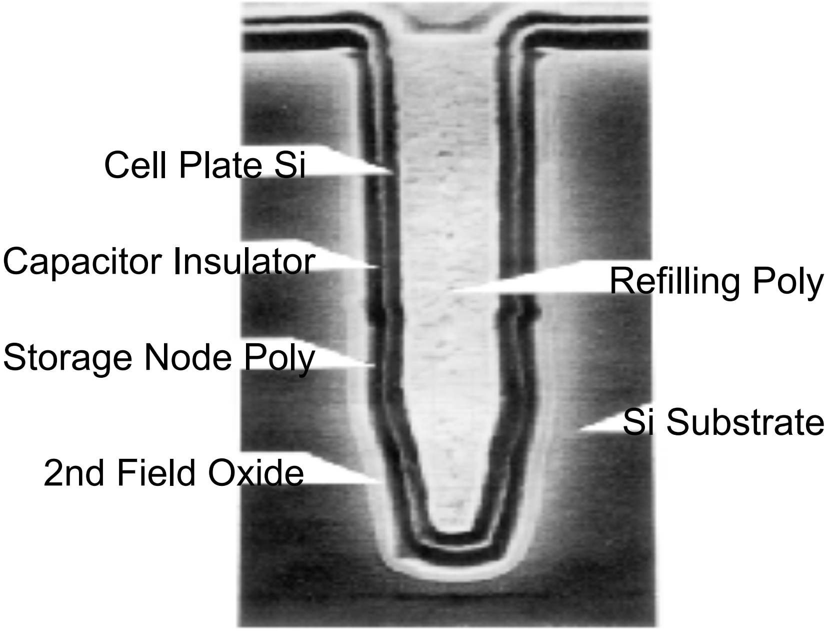
What About "Leakage"

- Those capacitors aren't great...
  - They will slowly lose charge over time...
- Idea: Just read ***every cell*** at a regular interval
  - This is called DRAM refresh
- Highly temperature dependent:
  - Hot DRAM tends to flip

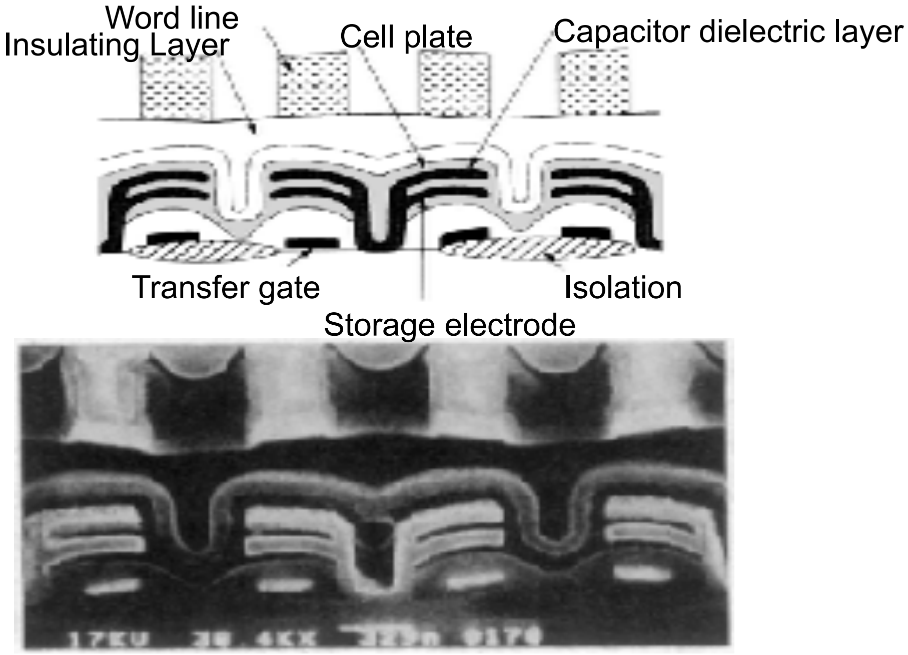
What About Errors?

- Redundancy & sparing for manufacturing faults...
  - Check all the cells. If any fail in a column, replace that column with a spare column
- ECC memory for transient faults
  - EG, radioactive decay, cosmic ray, etc...
  - Cheap systems don't use ECC, but you ***should***

Advanced 1T DRAM Cells



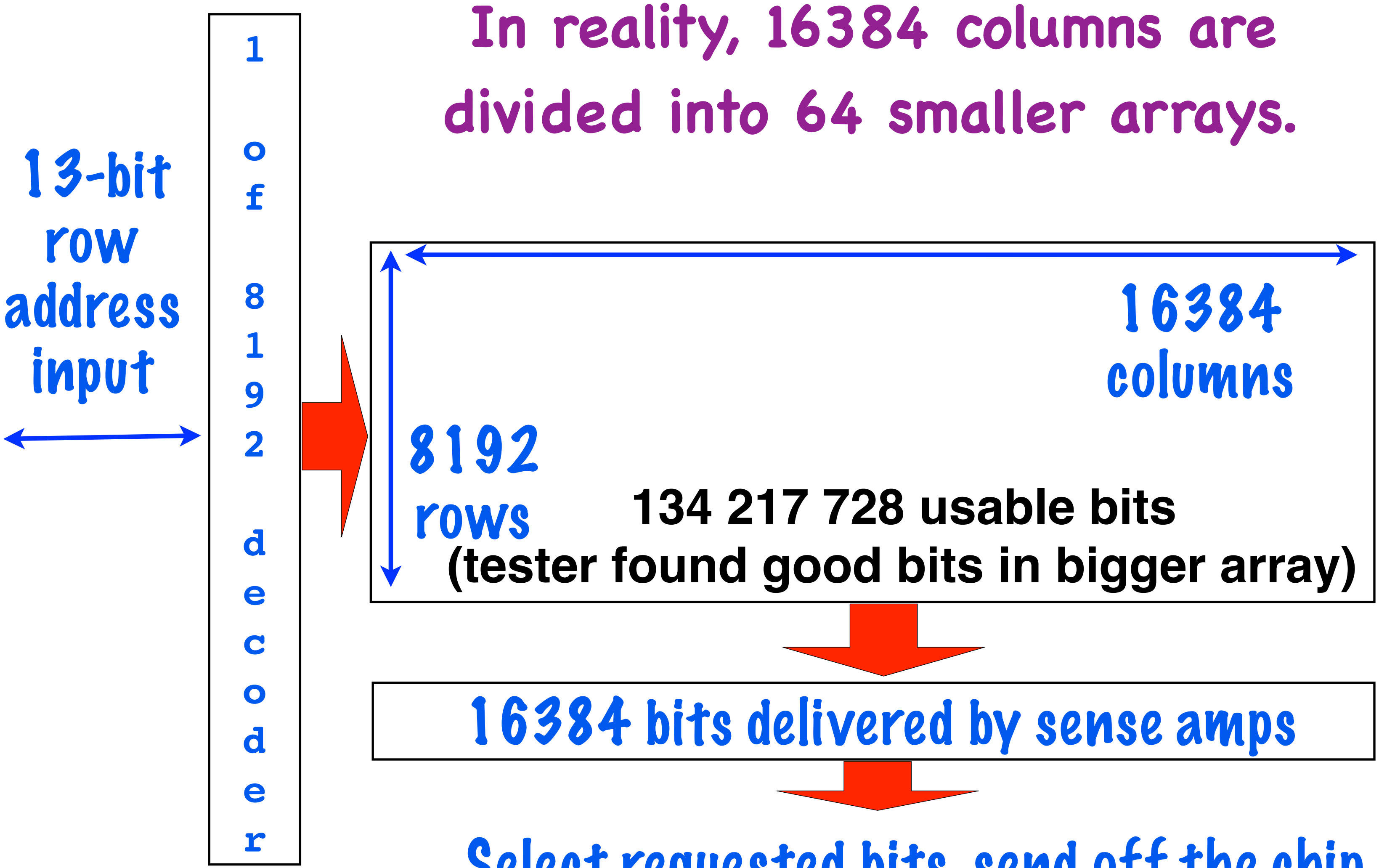
Trench Cell



Stacked-capacitor Cell

A “bank” of 128 Mb (512Mb chip -> 4 banks)

In reality, 16384 columns are divided into 64 smaller arrays.



13-bit row address input

1 of 8192 decoder

16384 columns

8192 rows

134 217 728 usable bits
(tester found good bits in bigger array)

16384 bits delivered by sense amps

Select requested bits, send off the chip

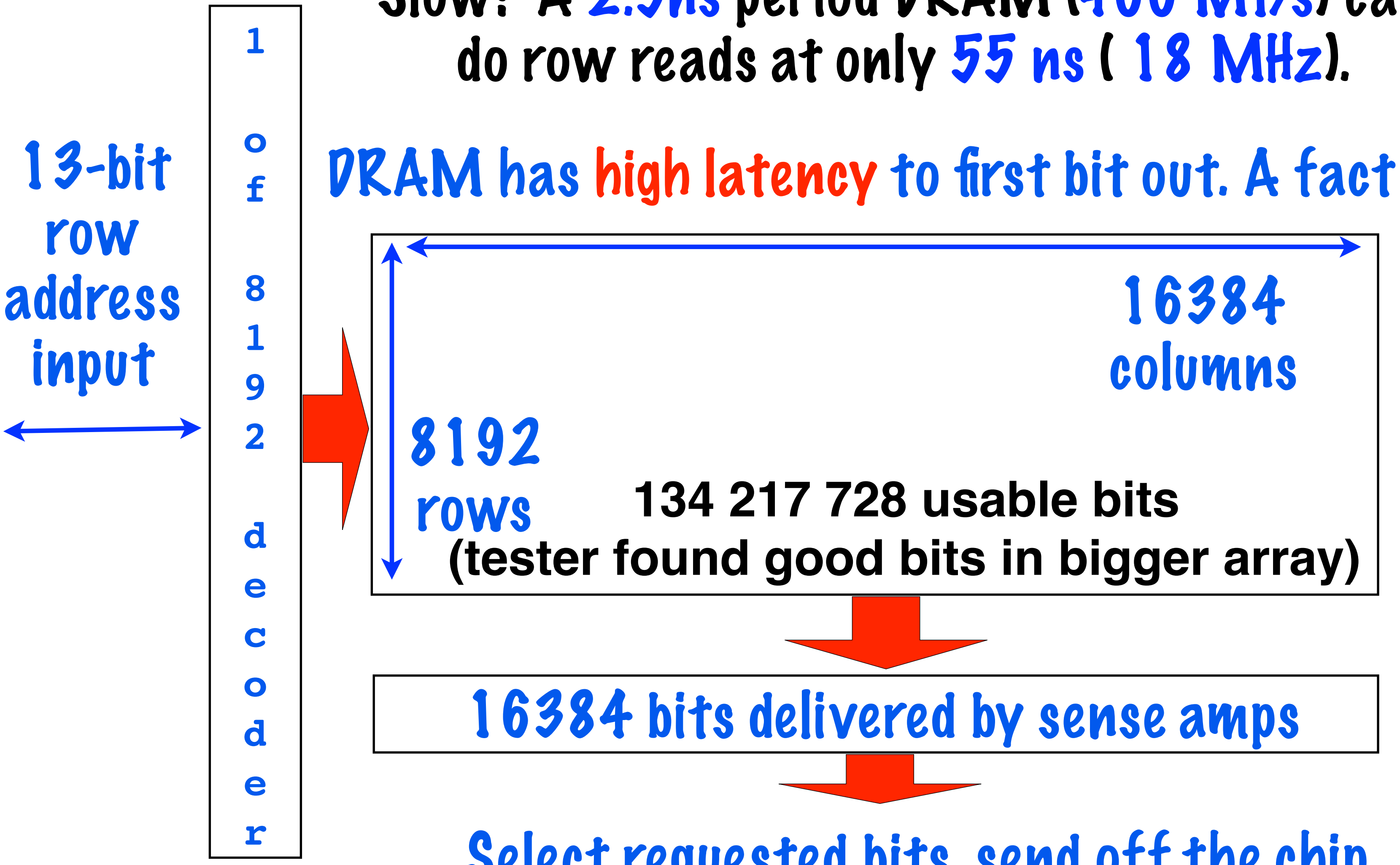
So basic DRAM read operation...

- Precharge all the bit lines in a block to .5V_{dd}
- Enable the word line for the desired row
  - "Row Access"
- Activate the sense-amps
  - Acts to read the bits ***and*** restore the bits
- Once the sense-amps have the row
  - Do the column access to get the sub pieces within the row
- A write will start with a read...
  - Then override the bits you want to set

“Sensing” is row read into sense amps

Slow! A 2.5ns period DRAM (400 MT/s) can do row reads at only 55 ns (18 MHz).

DRAM has high latency to first bit out. A fact of life.



13-bit row address input

1 of 8192 decoder

16384 columns

8192 rows

134 217 728 usable bits
(tester found good bits in bigger array)

16384 bits delivered by sense amps

Select requested bits, send off the chip

Latency is not the same as bandwidth!

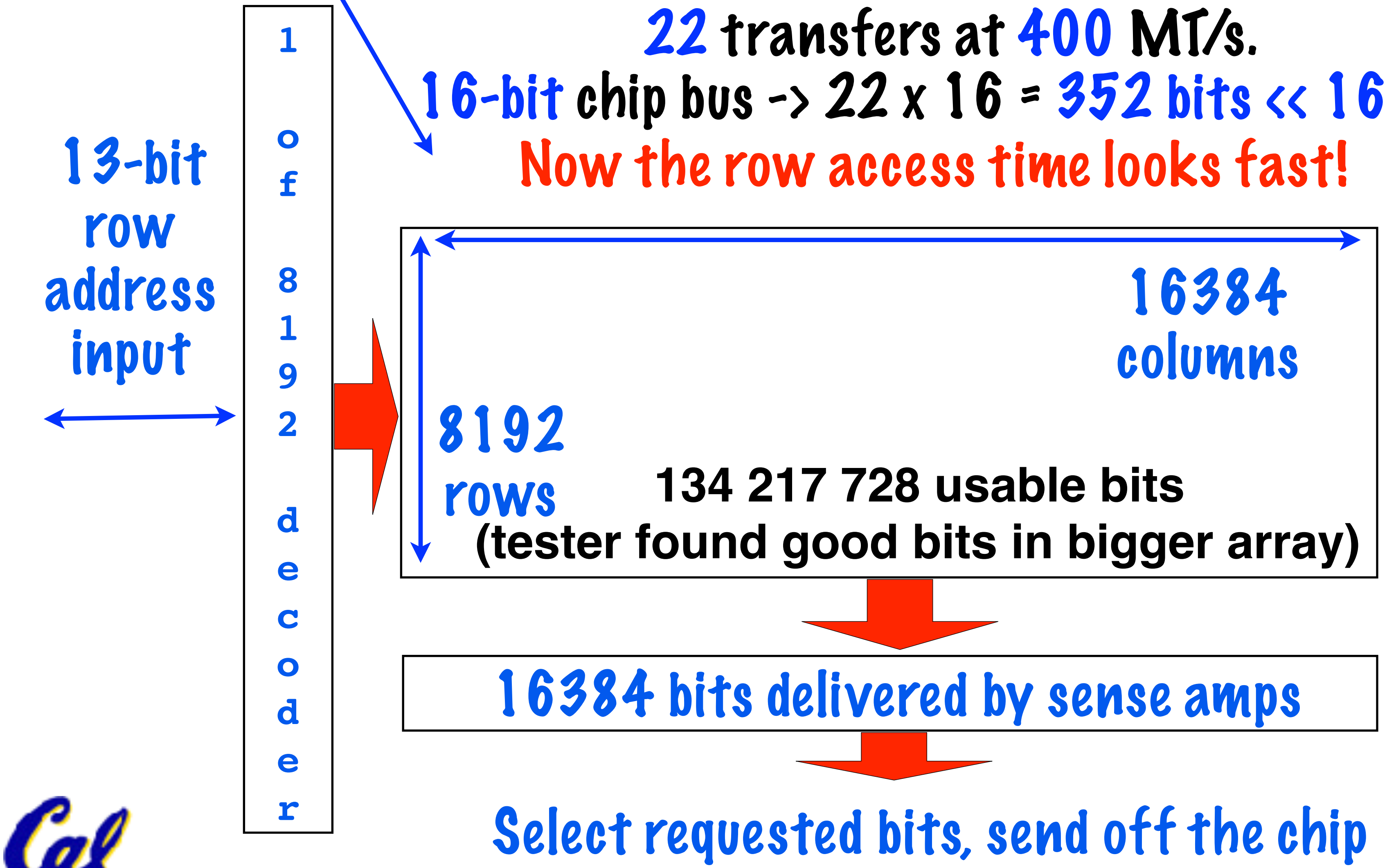
Thus, push to faster DRAM interfaces

What if we want all of the 16384 bits?

In row access time (55 ns) we can do 22 transfers at 400 MT/s.

16-bit chip bus -> 22 x 16 = 352 bits << 16384

Now the row access time looks fast!



13-bit row address input

1 of 8192 decoder

16384 columns

8192 rows

134 217 728 usable bits (tester found good bits in bigger array)

16384 bits delivered by sense amps

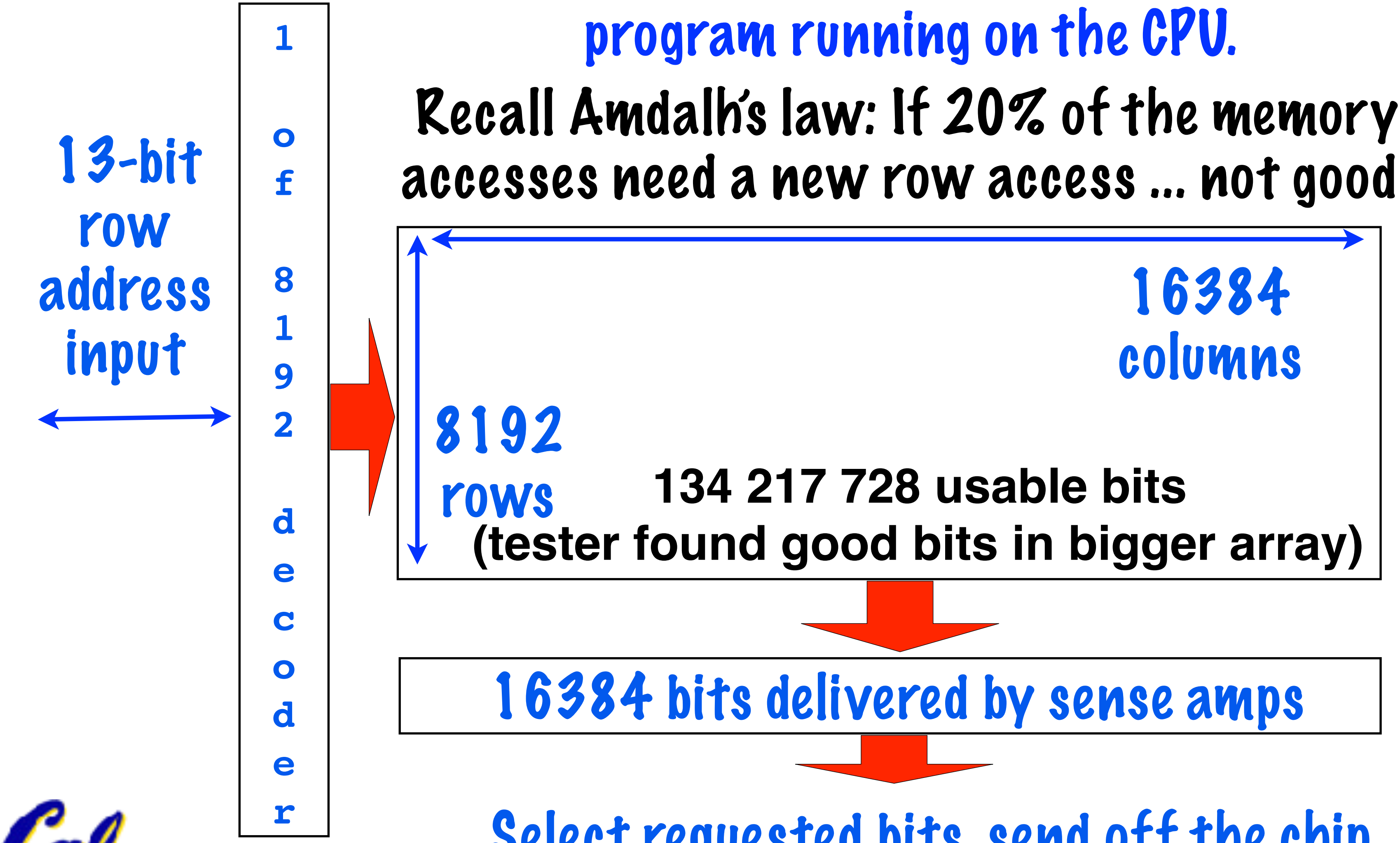
Select requested bits, send off the chip

Sadly, it's rarely this good ...

What if we want all of the 16384 bits?

The "we" for a CPU would be the program running on the CPU.

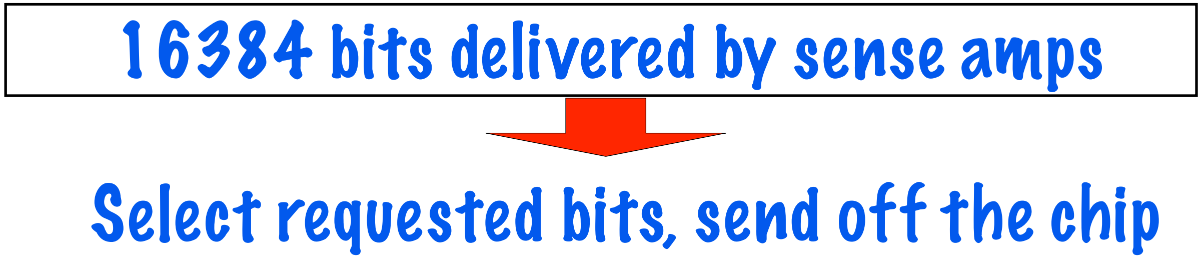
Recall Amdalh's law: If 20% of the memory accesses need a new row access ... not good.



Select requested bits, send off the chip

DRAM latency/bandwidth chip features

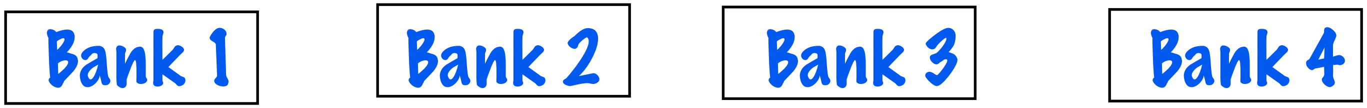
- **Columns:** Design the right interface for CPUs to request the subset of a column of data it wishes:



16384 bits delivered by sense amps

Select requested bits, send off the chip

- **Interleaving:** Design the right interface to the 4 memory banks on the chip, so several row requests run in parallel.

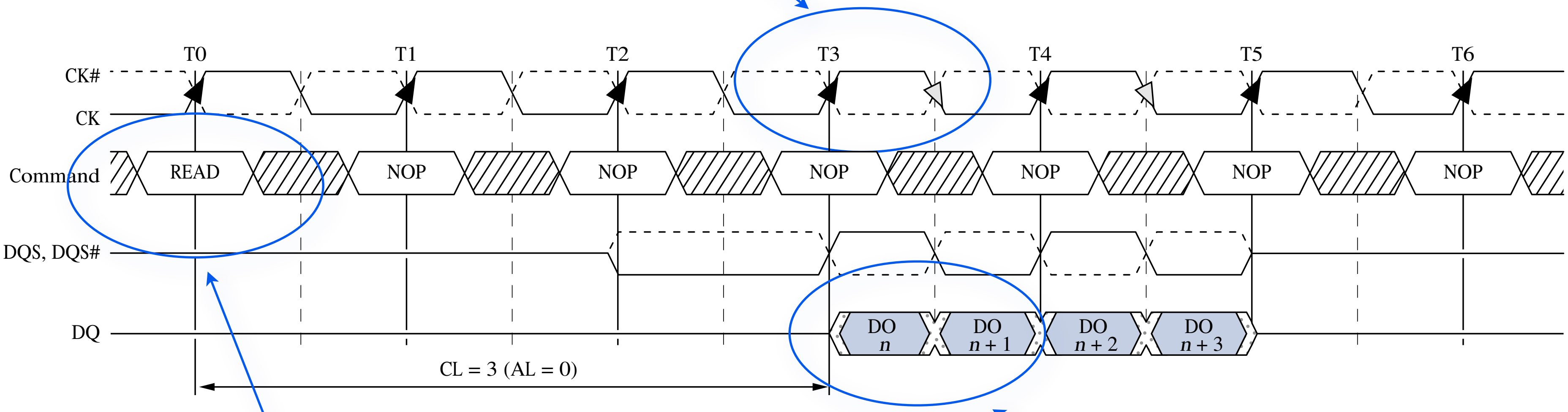


Bank 1 Bank 2 Bank 3 Bank 4

Off-chip interface for a Micron DDR part ...

A clocked bus: 200 MHz clock, data transfers on both edges (DDR).

Note! This example is best-case! To access a new row, a slow ACTIVE command must run before the READ.

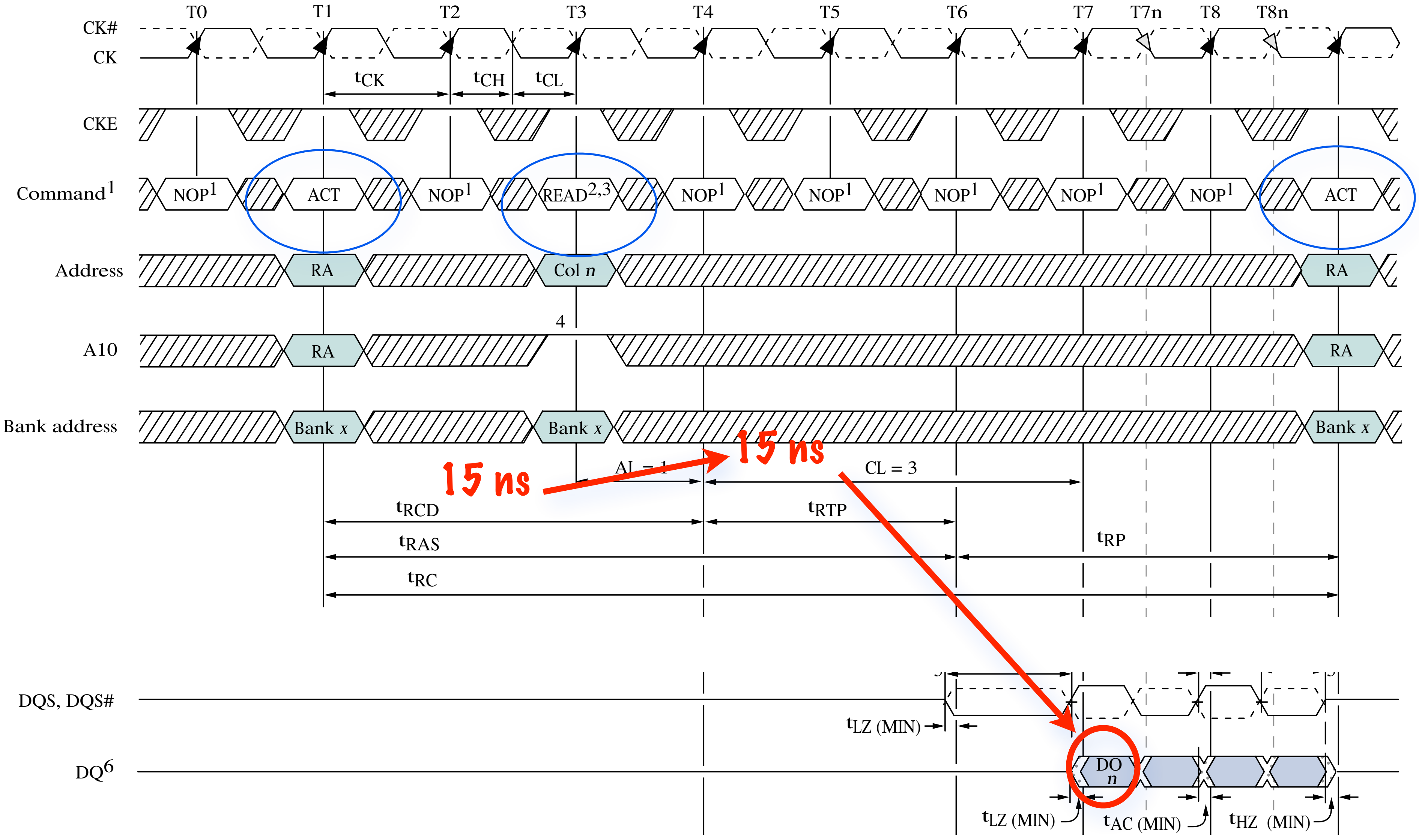


DRAM is controlled via commands (READ, WRITE, REFRESH, ...)

Synchronous data output.

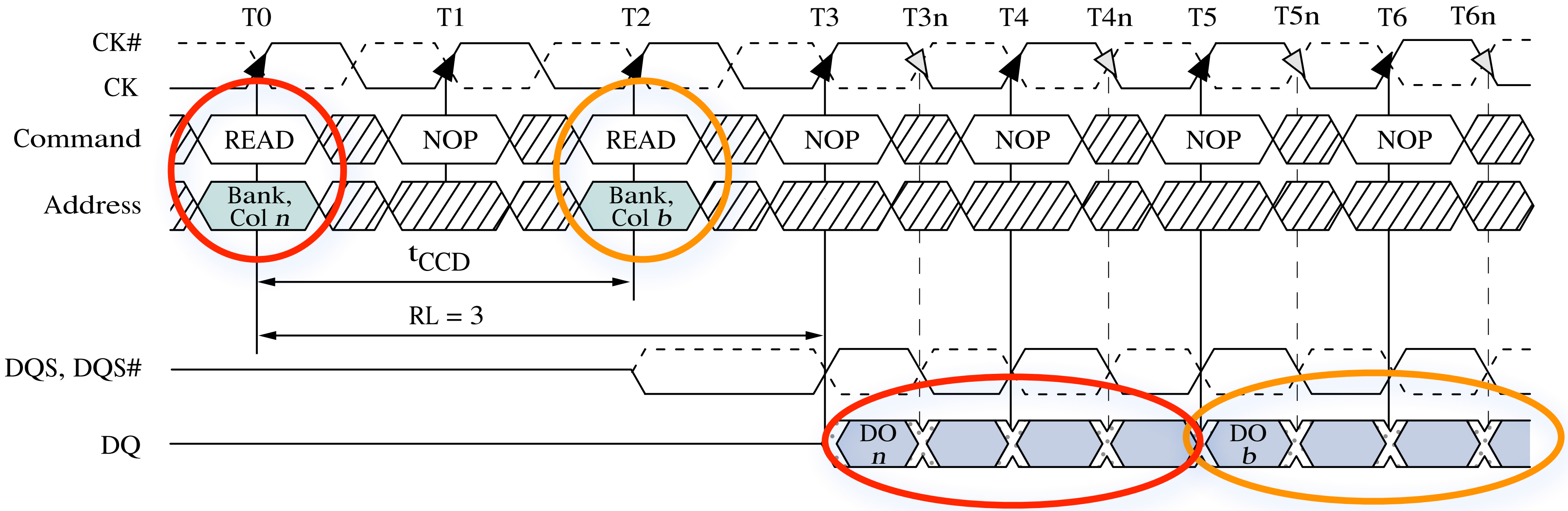
Opening a row before reading ...

Auto-Precharge READ



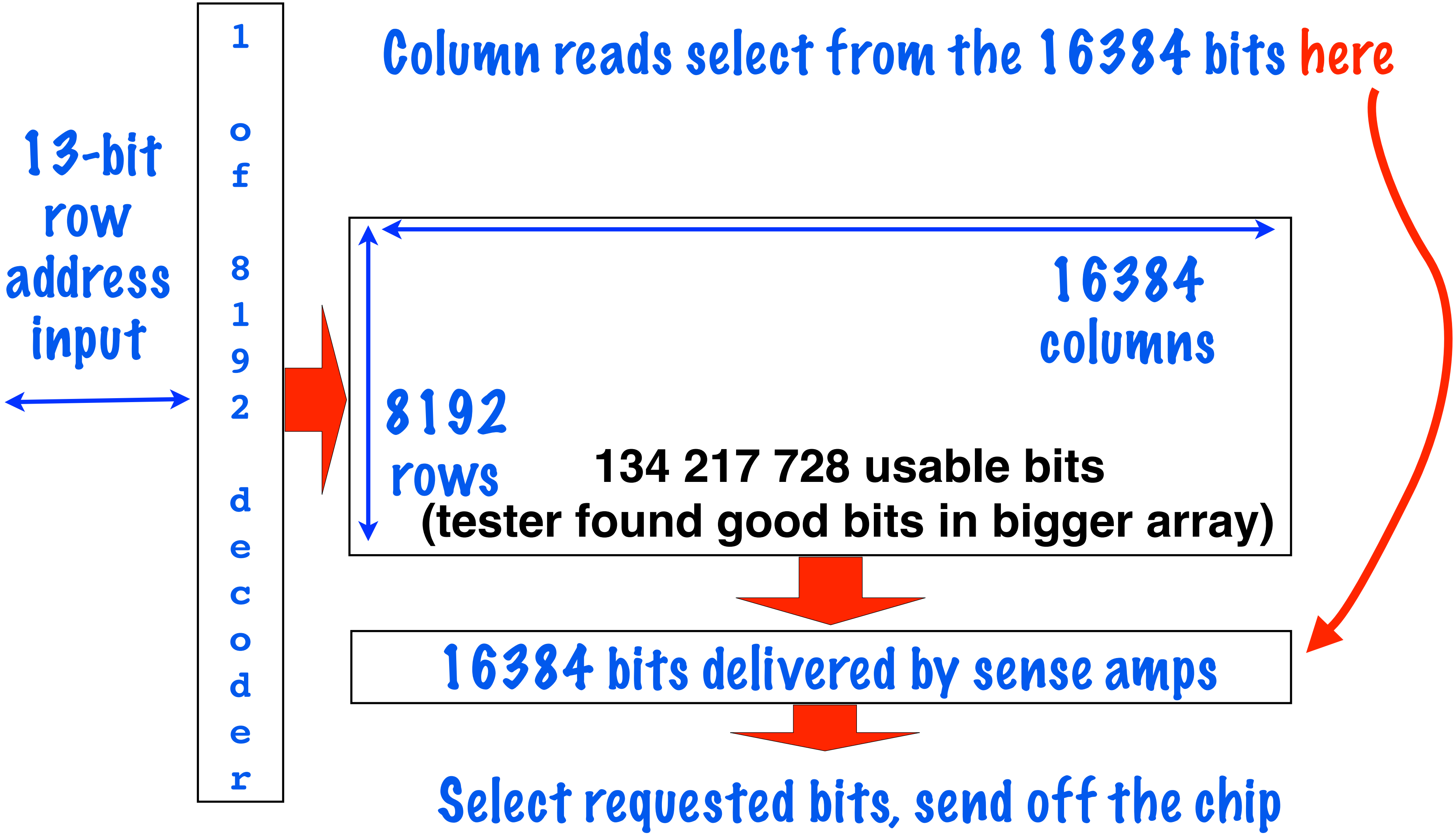
55 ns between row opens.

However, we can read columns quickly



Note: This is a "normal read" (not Auto-Precharge).
Both READs are to the same bank, but different columns.

Why can we read columns quickly?



Column reads select from the 16384 bits here

13-bit row address input

1 of 8192 decoder

16384 columns

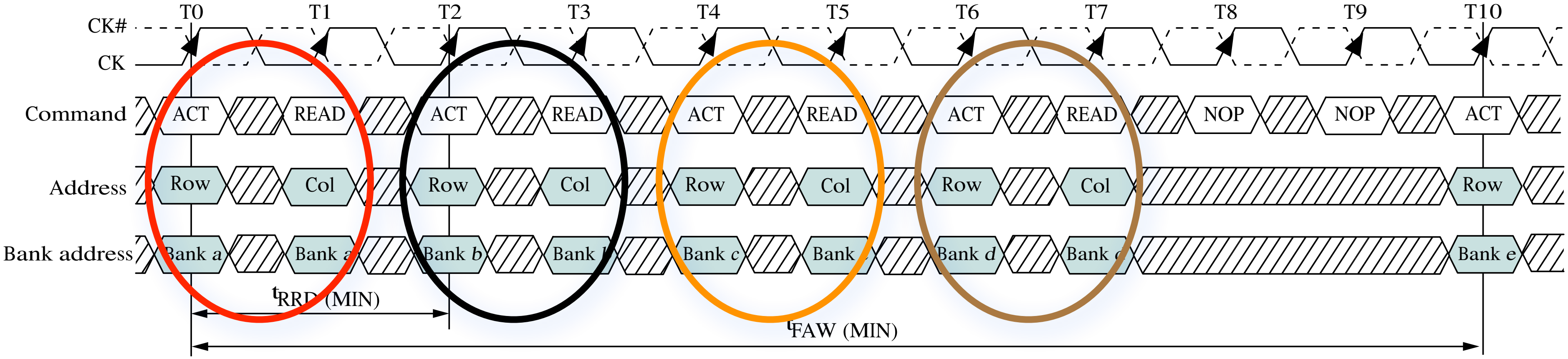
8192 rows

134 217 728 usable bits
(tester found good bits in bigger array)

16384 bits delivered by sense amps

Select requested bits, send off the chip

Interleave: Access all 4 banks in parallel



✳ **Interleaving:** **Design the right interface to the 4 memory banks on the chip, so several row requests run in parallel.**

Bank a

Bank b

Bank c

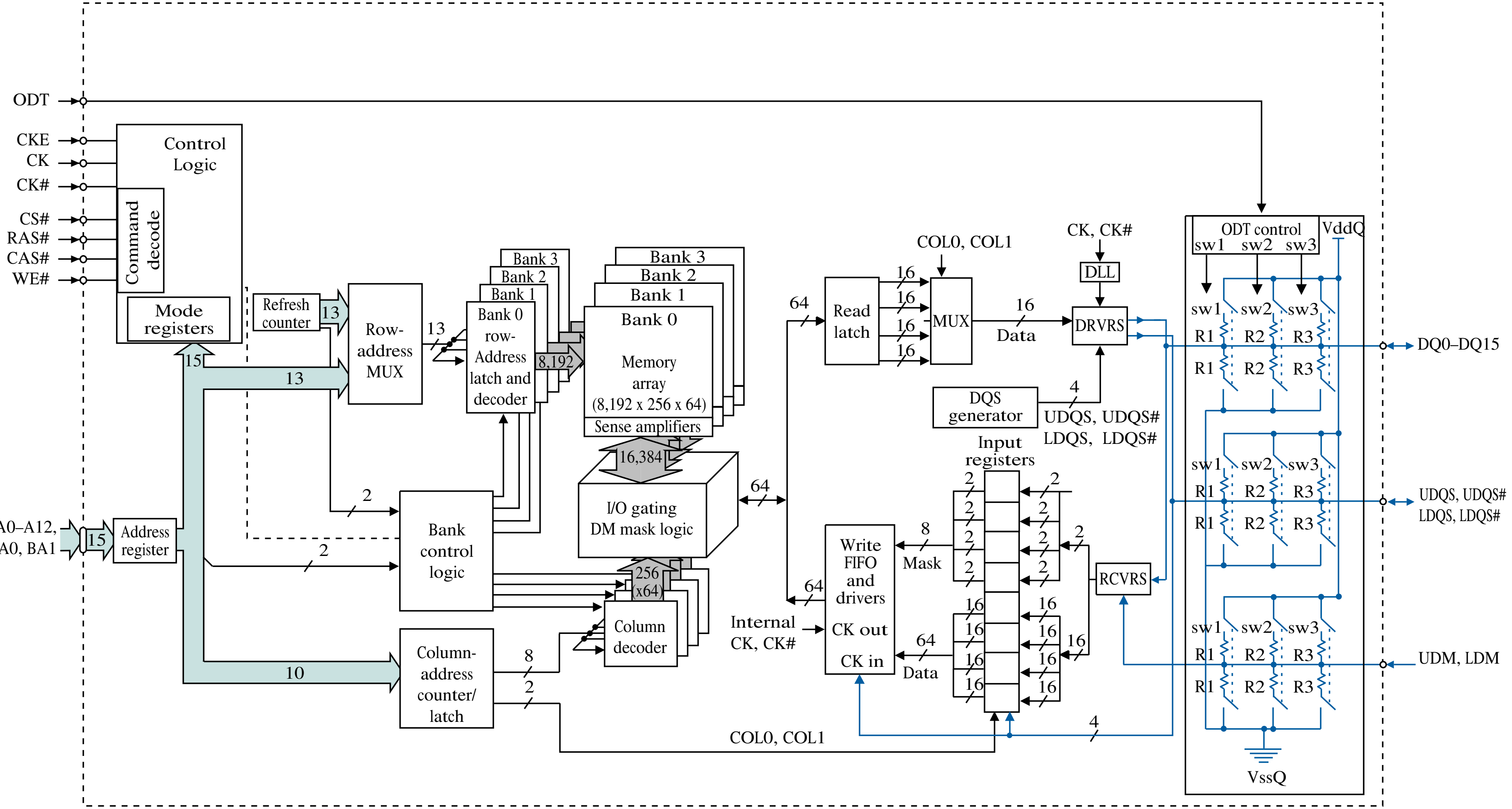
Bank d

Can also do other commands on banks concurrently.

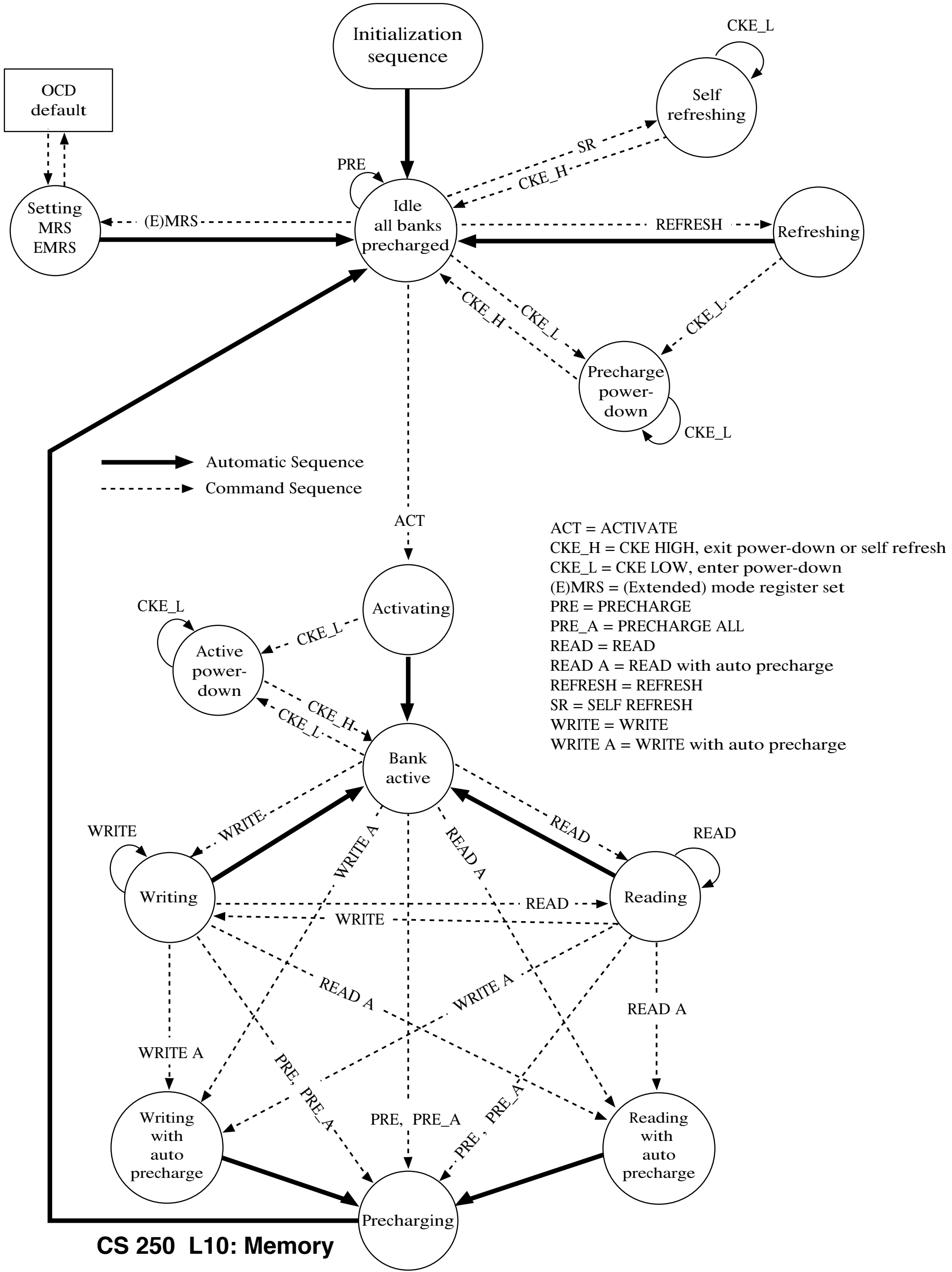
The SDRAM interface evolution

- Initially: 1b transferred per data line per clock cycle
- DDR: Double Data Rate: 2b per clock
  - Rising & falling edge
- DDR2: 4b per clock
- DDR3: 8b per clock
- DDR4: 16b per clock!
- But the latency hasn't really improved much in a decade+!!!
  - So if you are touching DRAM, try to make everything sequential

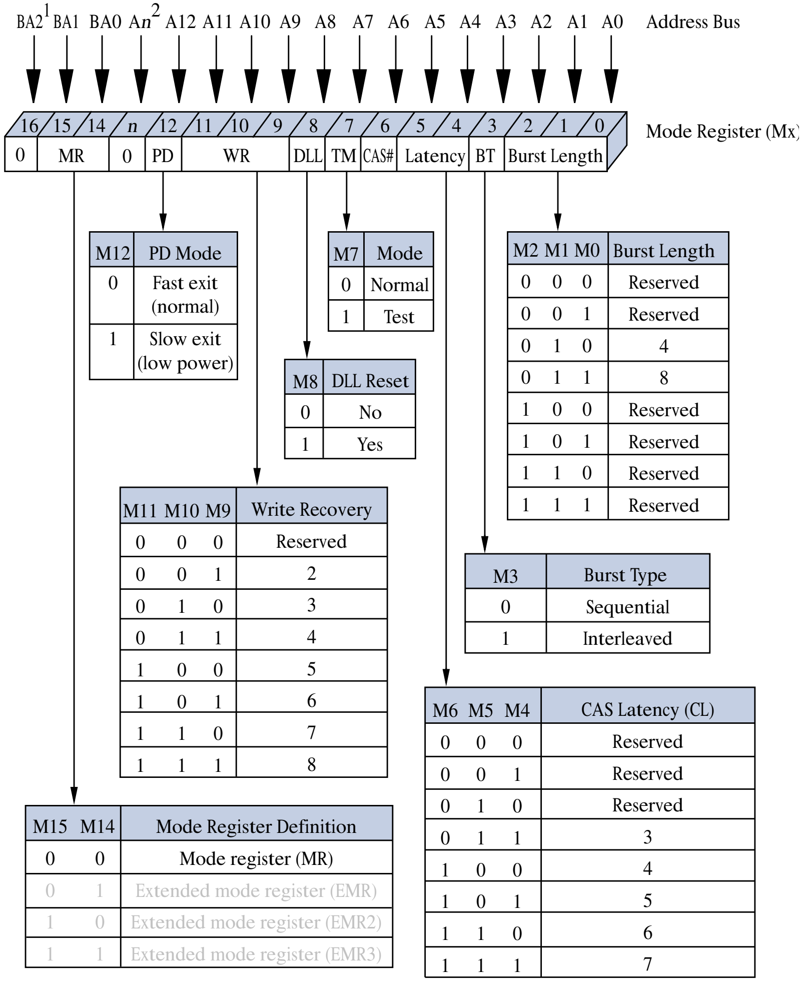
Only part of a bigger story ...



Only part of a bigger story ...



ACT = ACTIVATE
CKE_H = CKE HIGH, exit power-down or self refresh
CKE_L = CKE LOW, enter power-down
(E)MRS = (Extended) mode register set
PRE = PRECHARGE
PRE_A = PRECHARGE ALL
READ = READ
READ A = READ with auto precharge
REFRESH = REFRESH
SR = SELF REFRESH
WRITE = WRITE
WRITE A = WRITE with auto precharge


BA2[1] BA1 BA0 An[2] A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
Address Bus
16 15 14 n 12 11 10 9 8 7 6 5 4 3 2 1 0
0 MR 0 PD WR DLL TM CAS# Latency BT Burst Length
Mode Register (Mx)

M12	PD Mode
0	Fast exit (normal)
1	Slow exit (low power)

M7	Mode
0	Normal
1	Test

M8	DLL Reset
0	No
1	Yes

M2	M1	M0	Burst Length
0	0	0	Reserved
0	0	1	Reserved
0	1	0	4
0	1	1	8
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	Reserved

M11	M10	M9	Write Recovery
0	0	0	Reserved
0	0	1	2
0	1	0	3
0	1	1	4
1	0	0	5
1	0	1	6
1	1	0	7
1	1	1	8

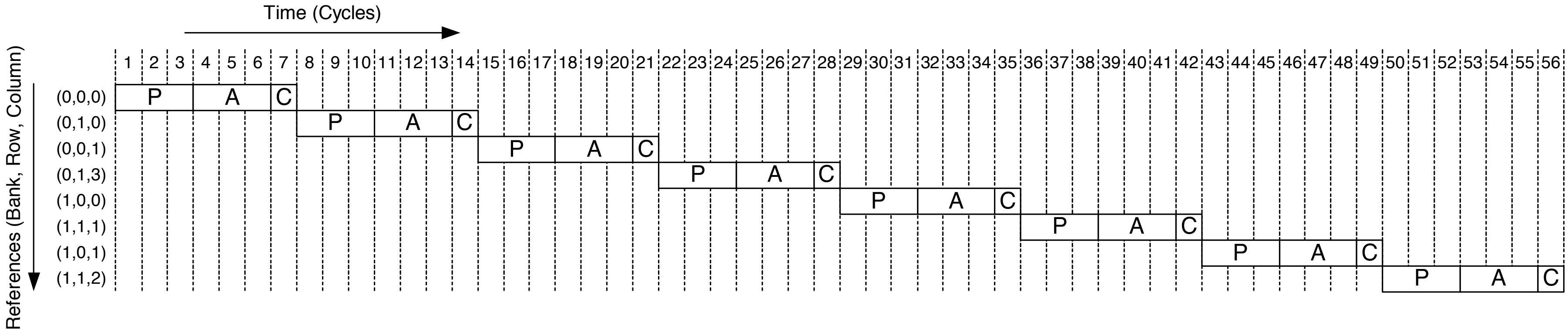
M3	Burst Type
0	Sequential
1	Interleaved

M6	M5	M4	CAS Latency (CL)
0	0	0	Reserved
0	0	1	Reserved
0	1	0	Reserved
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

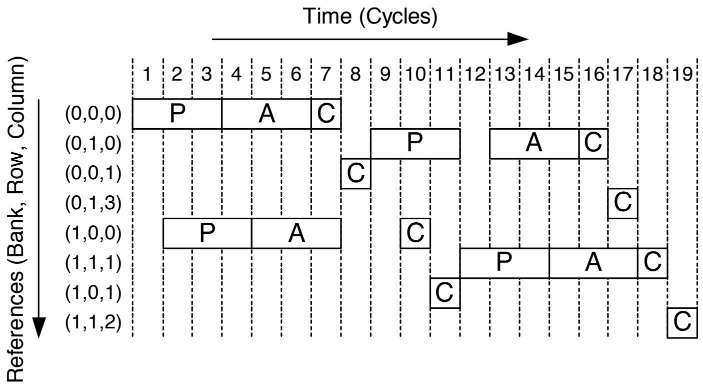
M15	M14	Mode Register Definition
0	0	Mode register (MR)
0	1	Extended mode register (EMR)
1	0	Extended mode register (EMR2)
1	1	Extended mode register (EMR3)

DRAM controllers: reorder requests

(A) Without access scheduling (56 DRAM Cycles)



(B) With access scheduling (19 DRAM Cycles)



DRAM Operations:

P: bank precharge (3 cycle occupancy)
A: row activation (3 cycle occupancy)
C: column access (1 cycle occupancy)

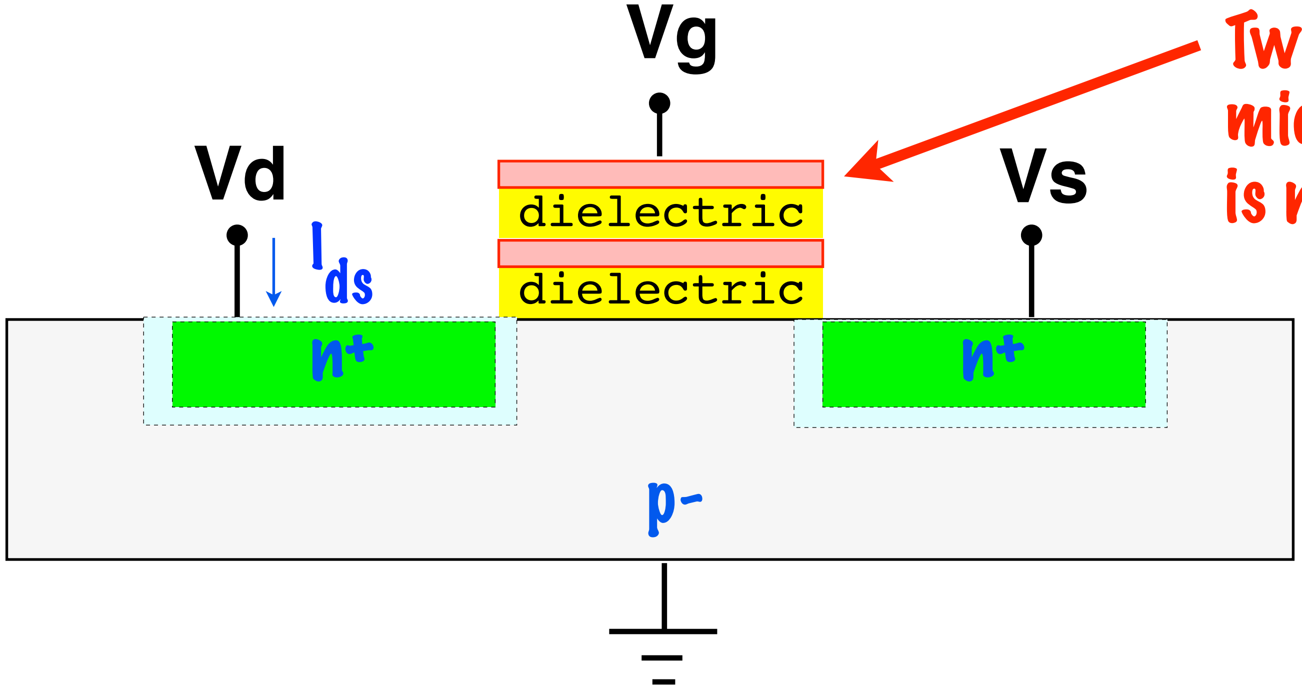
From: **Memory Access Scheduling**

Scott Rixner[1], William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens

And Rowhammer....

- These DRAM cells are not ***perfectly*** isolated...
  - In reading a row, there is a chance that it could flip a bit in a different row
- Rowhammer is a hardware attack
  - Repeatedly read the same DRAM row to cause bit-flips elsewhere
  - Asking the OS to effectively fill the memory with page tables...
    - And then when a bit flips, it will cause the page table to be mapped into the process address space...
    - And once you do that, you win!

The physics of FLASH memory

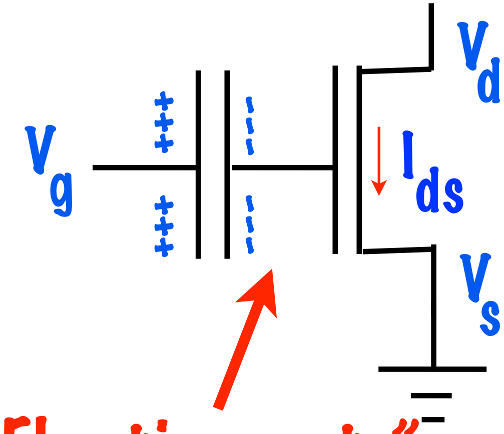


Two gates. But the middle one is not connected.

1. Electrons "placed" on floating gate stay there for many years (ideally).

2. 10,000 electrons on floating gate shift transistor threshold by 2V.

3. In a memory array, shifted transistors hold "0", unshifted hold "1".

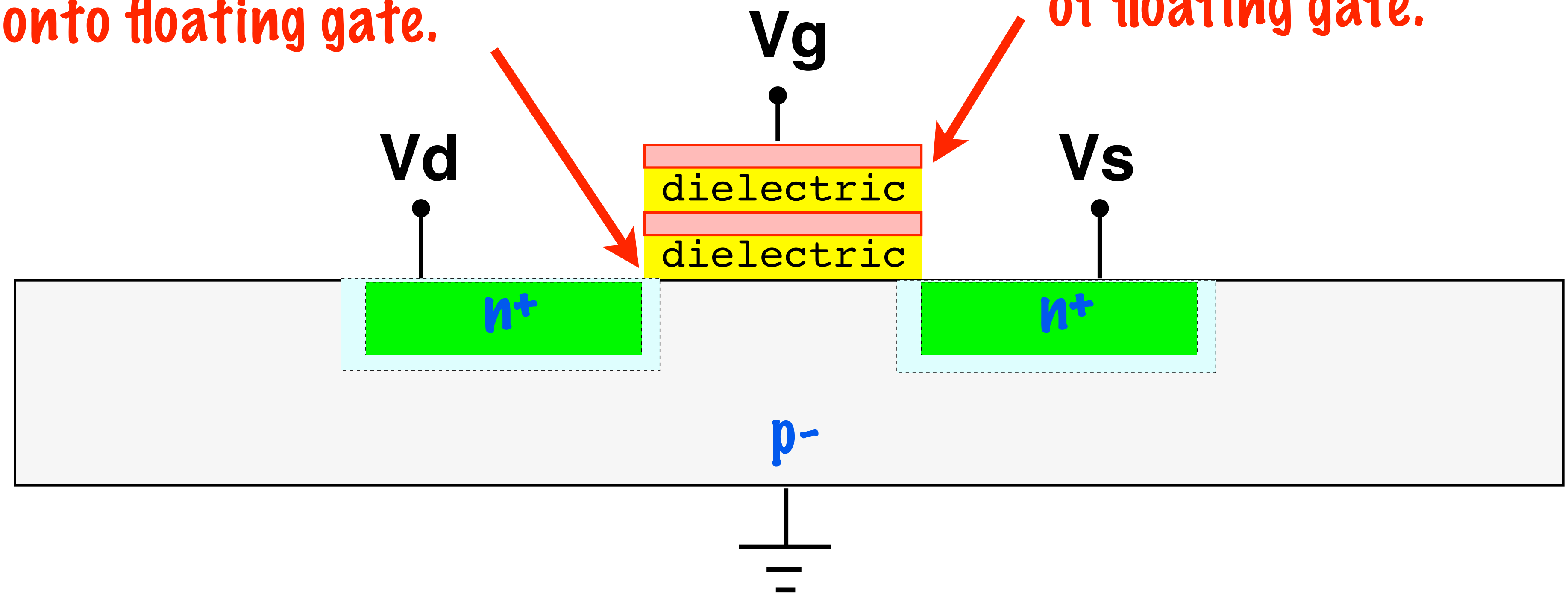


"Floating gate".

Moving electrons on/off floating gate

A high drain voltage injects "hot electrons" onto floating gate.

A high gate voltage "tunnels" electrons off of floating gate.



1. Hot electron injection and tunneling produce tiny currents, thus writes are slow.

2. High voltages damage the floating gate. Too many writes and a bit goes "bad".

NAND Flash Memory

Flash: Disk Replacement

Chip "remembers" for 10 years.

Presents memory to the CPU as a set of **pages.**

Page format:

2048 Bytes	+	64 Bytes
(user data)		(meta data)

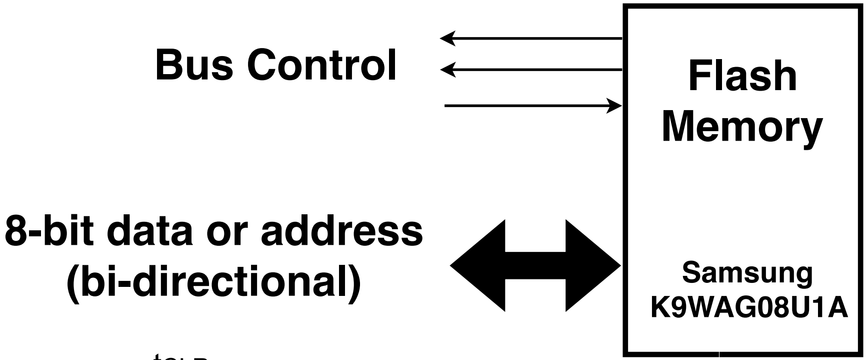
1GB Flash: 512K pages

2GB Flash: 1M pages

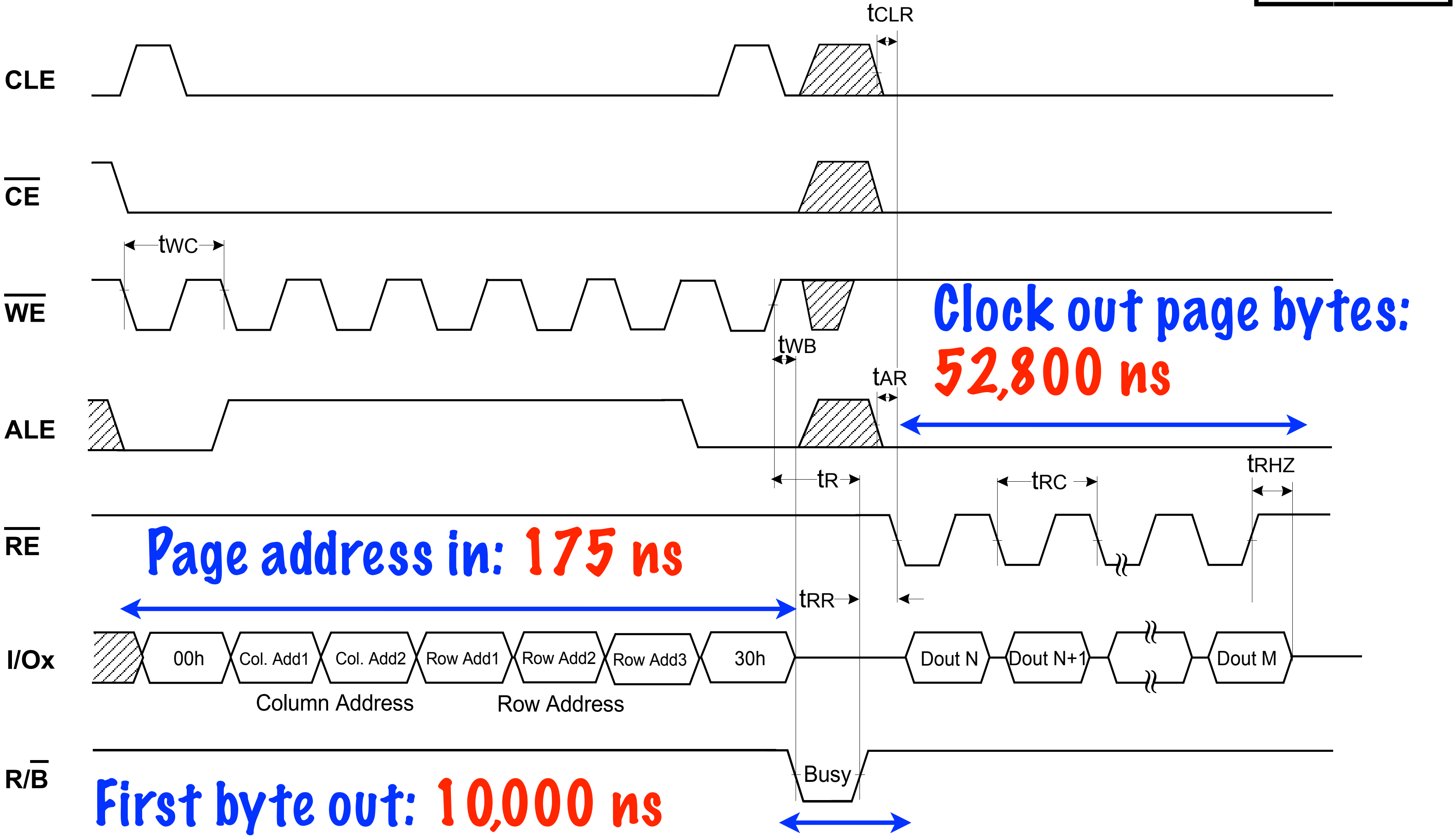
4GB Flash: 2M pages

Reading a Page ...

33 MB/s Read Bandwidth



Read Operation



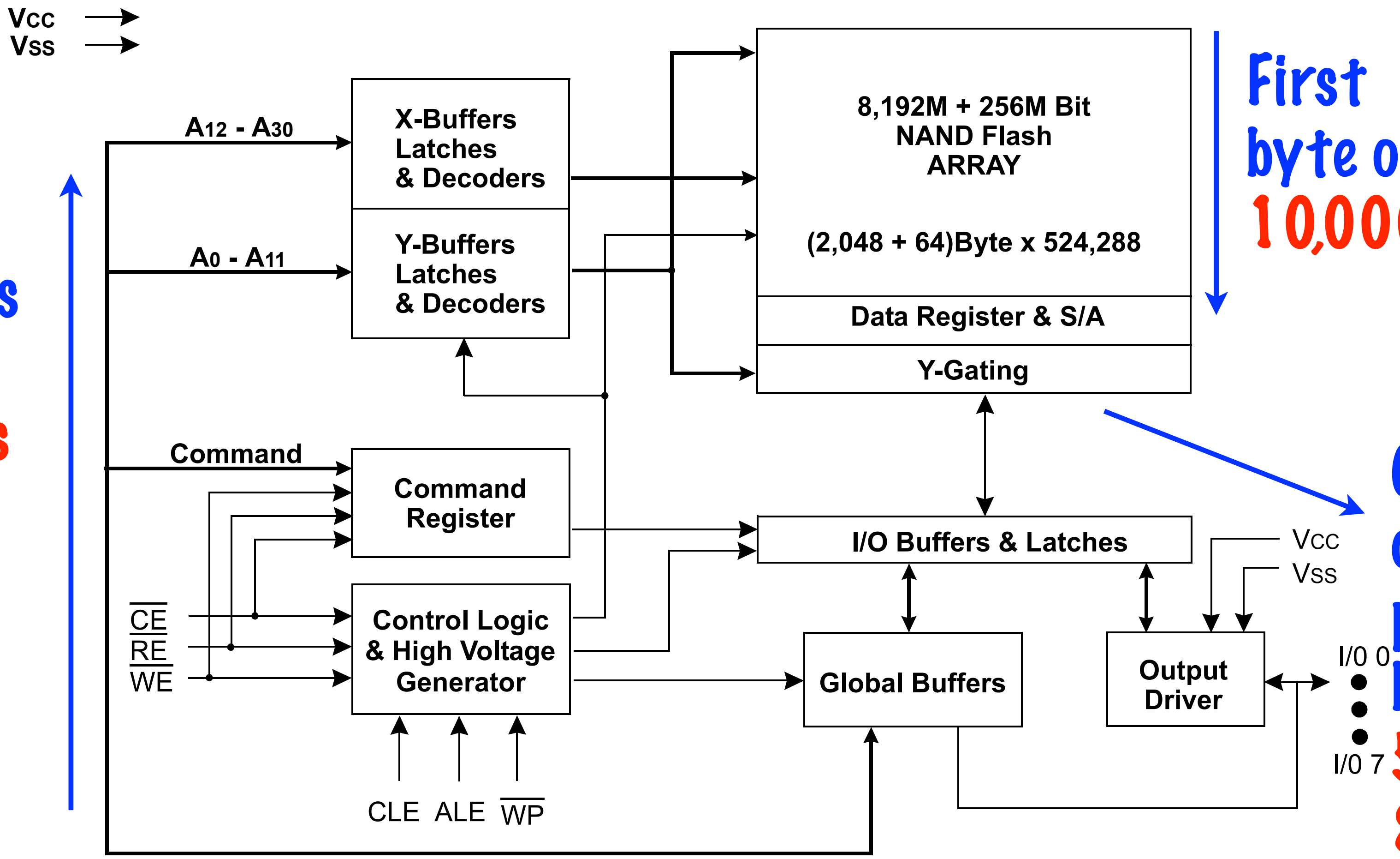
Clock out page bytes: 52,800 ns

Page address in: 175 ns

First byte out: 10,000 ns

Where Time Goes

Figure 1. K9K8G08U0A Functional Block Diagram


Vcc
Vss
A12 - A30
A0 - A11
X-Buffers Latches & Decoders
Y-Buffers Latches & Decoders
8,192M + 256M Bit NAND Flash ARRAY
(2,048 + 64)Byte x 524,288
Data Register & S/A
Y-Gating
Command
Command Register
I/O Buffers & Latches
Control Logic & High Voltage Generator
Global Buffers
Output Driver
Vcc
Vss
CE
RE
WE
CLE
ALE
WP
I/0 0
I/0 7

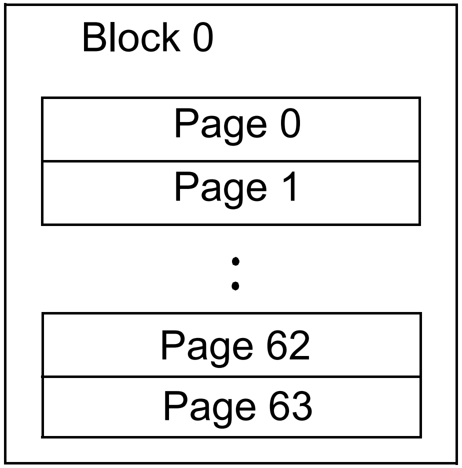
Page address in: 175 ns

First byte out: 10,000 ns

Clock out page bytes: 52, 800 ns

Writing a Page ...

A **page** lives in a **block** of **64 pages:**



1GB Flash: 8K blocks

2GB Flash: 16K blocks

4GB Flash: 32K blocks

To write a page:

1. Erase all pages in the block (cannot erase just one page).

Time: 1,500,000 ns

2. May program each page individually, exactly once.

Time: 200,000 ns per page.

Block lifetime: 100,000 erase/program cycles.

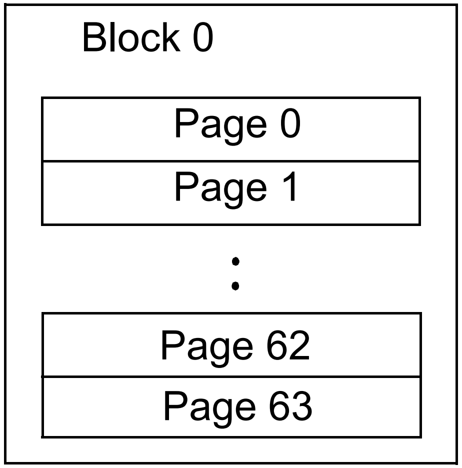
Block Failure

Even when new, not all blocks work!

1GB: 8K blocks, 160 may be bad.

2GB: 16K blocks, 220 may be bad.

4GB: 32K blocks, 640 may be bad.



Block 0

Page 0

Page 1

:

Page 62

Page 63

During factory testing, Samsung writes good/bad info for each block in the meta data bytes.

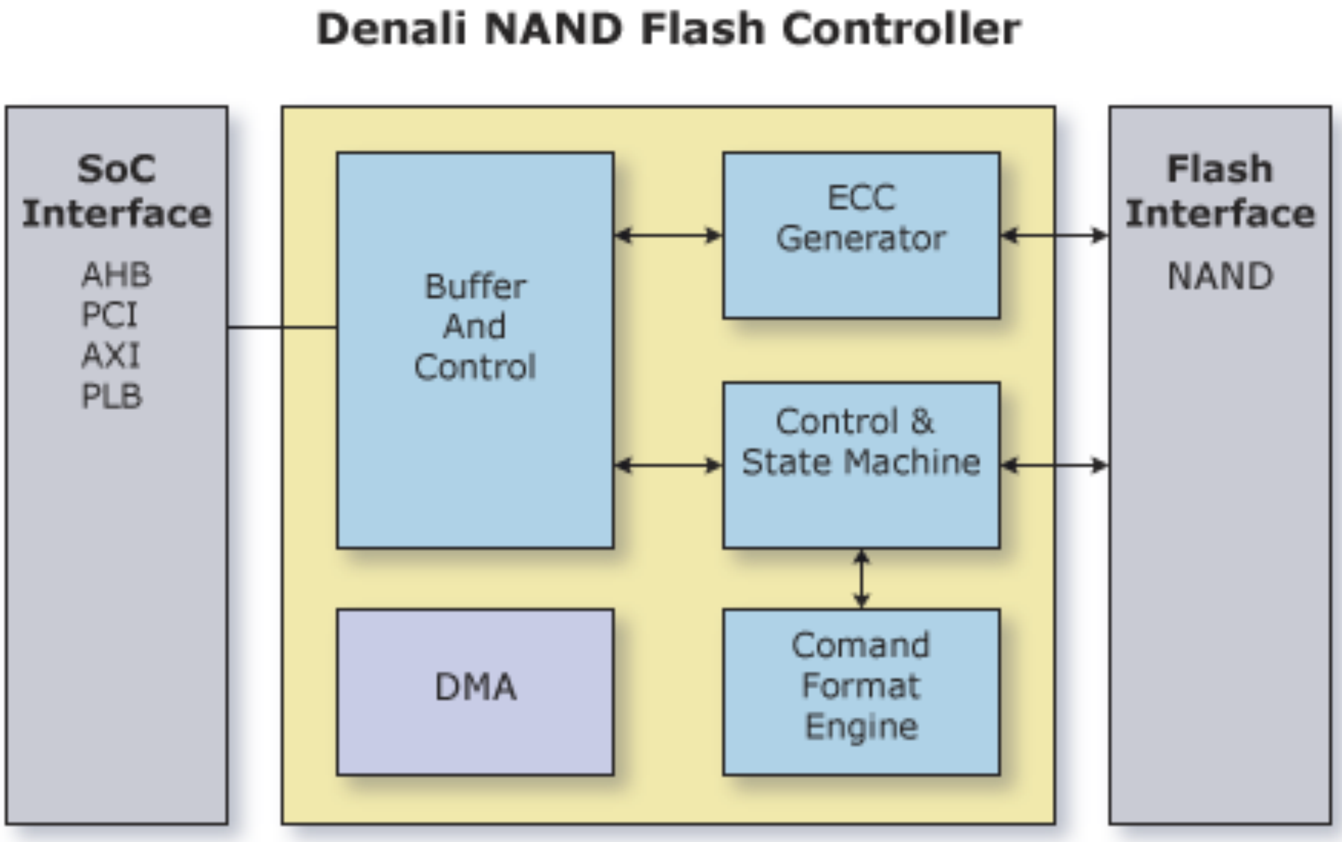
2048 Bytes + **64 Bytes**

(user data) **(meta data)**

After an erase/program, chip can say "write failed", and block is now "bad". OS must recover (migrate bad block data to a new block). Bits can also go bad "silently" (!!!).

Flash controllers: Chips or Verilog IP ...

Flash memory controller manages write lifetime management, block failures, silent bit errors ...



Software sees a “perfect” disk-like storage device.

Actually Using Memory

- Two options:
  - Directly instantiate memory blocks:
    Can either use IP generators or instantiate primitives directly
  - Write Verilog with something the tools can infer

    -
      ```
      parameter n = 4;
      parameter w = 8;
      reg [w-1:0] reg_array [2**n-1:0];
      always @ posed clk begin
         if (we) reg[write_addr] <= din;
      end
      always @* begin
        dout <= reg[read_addr]
      end
      ```
    - ***should*** be inferred as a simple dual-port memory:
      One synchronous write port, one asynchronous read port

Some Types of Memory...

- Single port:
  - One address port, one data in port, one data out port
  - Can read or write
- Simple dual port:
  - Two address ports, one for reading, one for writing
    - Very good for implementing FIFOs!
- True dual port:
  - Two address ports, both can be used for reading or writing
- Suggestion for Xilinx version of the project:
  - Best way to do the processor reg-file is instantiate simple-dual-port memories