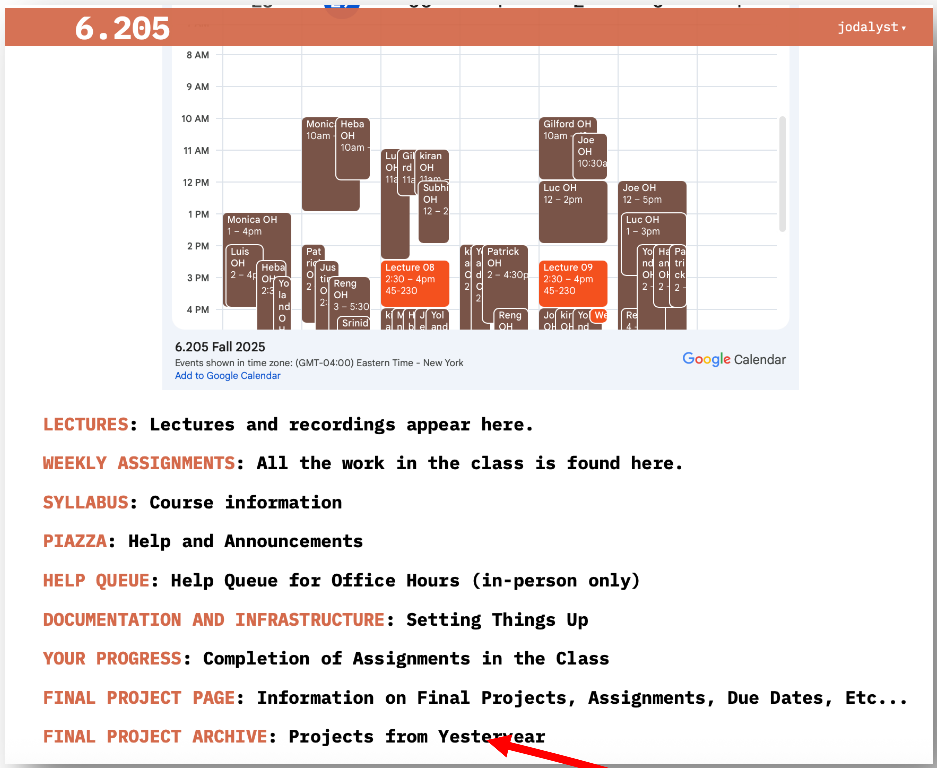
Administrative

- Week 04 Due tomorrow
- Week 05 Comes out on Thursday:
  - Using BRAMs to make image sprites
  - Video/Camera pipeline
  - Working with Camera to track objects



LECTURES: Lectures and recordings appear here.

WEEKLY ASSIGNMENTS: All the work in the class is found here.

SYLLABUS: Course information

PIAZZA: Help and Announcements

HELP QUEUE: Help Queue for Office Hours (in-person only)

DOCUMENTATION AND INFRASTRUCTURE: Setting Things Up

YOUR PROGRESS: Completion of Assignments in the Class

FINAL PROJECT PAGE: Information on Final Projects, Assignments, Due Dates, Etc...

FINAL PROJECT ARCHIVE: Projects from Yesteryear

- Final Project Dates and Schedule Will be Released Tomorrow/Thursday. Archive on site now
- Start Teaming! We will want to have teams by next week so we can either form teams or have you start formulating projects.

Memory

- Overview of Memories
- Memories on the FPGA
- Memories in Verilog
- External Memories
    - Flash
    - DRAM

Memories: The general state…

- **The good news: huge selection of technologies**
  - Small & faster vs. large & slower
  - Every year capacities go up and prices go down
- **The bad news: perennial system bottleneck**
  - Latencies (access time) haven't kept pace with cycle times
  - Often a separate technology from logic, so must communicate between silicon, so physical limitations (# of pins, R's and C's and L's) limit bandwidths
  - Likely one of the limiting factor in cost & performance of many digital systems (including your designs): designers spend a lot of time figuring out how to keep memories running at peak bandwidth

Memory in Hardware vs. Memory “in” Software

- There is a huge disconnect in software, particularly in higher level languages, to memory...in fact one of the reasons high-level languages exist is to facilitate that disconnect
- Python at first glance makes it seem like you can instantaneously access

```
for i in range(1000):
  print(y[i])
```

- This is absolutely 100% not the case!

In reality…

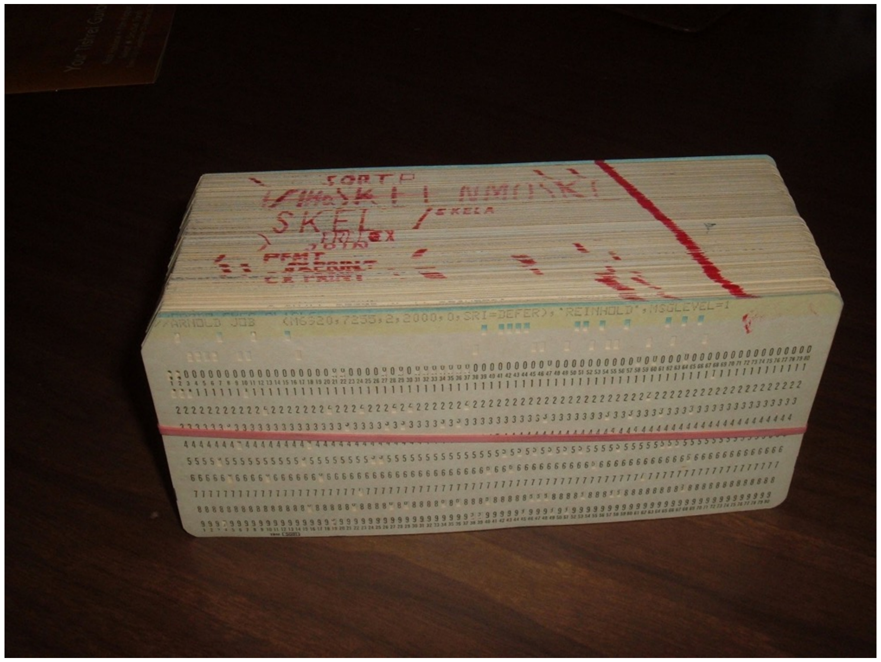
- Memory is often stored in very tightly packed, difficult to access arrays.
- Doing things with the data in that memory (reads/writes) inherently takes time (maybe many clock cycles)…"instant" access is often not easy
- In HW, "instant" access is actually instant. There's no Python interpreter there to lie to you and add cushy pillows around you. If you want something instantly, Verilog/Vivado will try to give it to you and it may be impossible…or very expensive to do.
- You need to be aware of that.

How do we Electrically Remember Things?

- We can convey/transfer information with voltages that change over time
- How can we store information in an electrically accessible manner?
- Store in either:
  - Electric Field
  - Magnetic Field

Early attempts:

- Punched Cards have existed as electromechanical program storage since ~1800s
- Switches would sense holes in card and interpret as 1's and 0's
- We're mostly concerned with rewritable storage mechanisms today (cards were true ROMs)



Computer program in punched card format

https://en.wikipedia.org/wiki/Computer_programming_in_the_punched_card_era

Electronic Memories in History

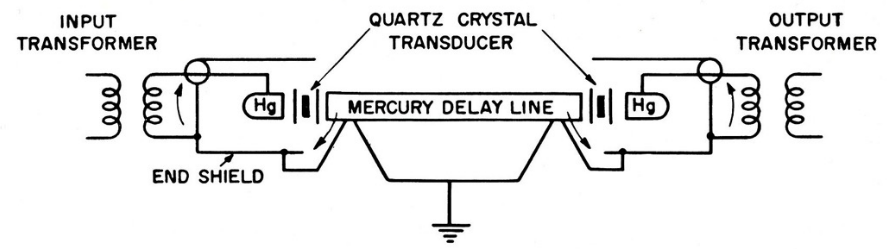
- Drum Memory:
  - Information stored magnetically on large rotating metallic cylinder
  - Could read/write to it
- Did not require periodic refresh

- Non-volatile (lasted after power cycles off)

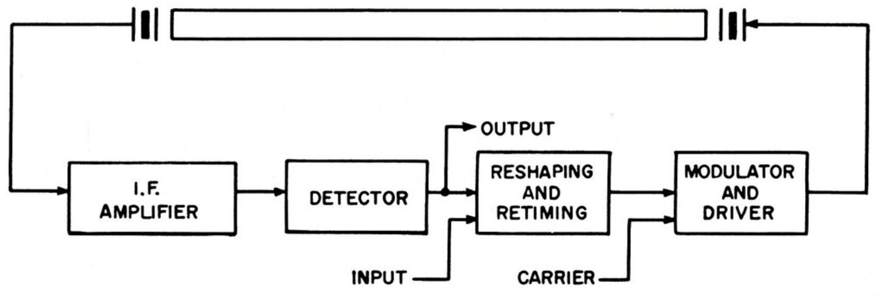
http://www.computerhistory.org/timeline/memory-storage/

Delay Line Memory

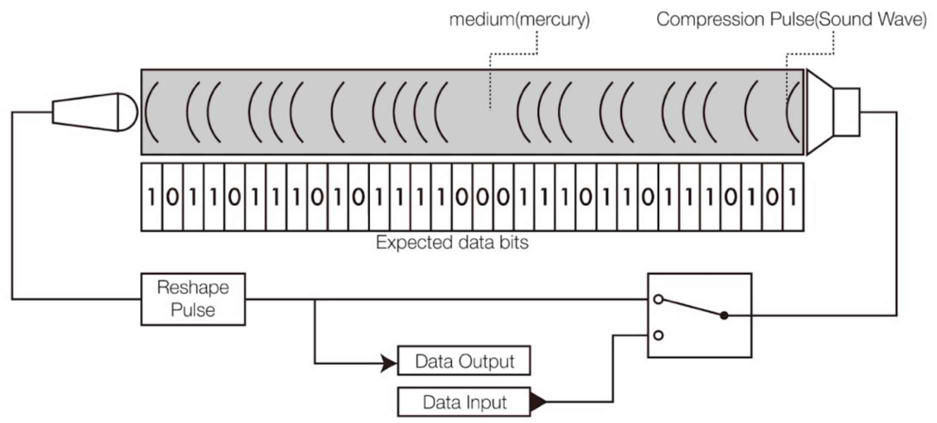
- Early form of FIFO memory
- Generate a wave pattern which exists for a few milliseconds in mercury
- Recover on the other end and either modify/reload or use
- Requires "refresh" circuitry
- Volatile (info lost soon after power cut)



Schematic diagram of circuit connections to the acoustic delay line used in NBS mercury memory.



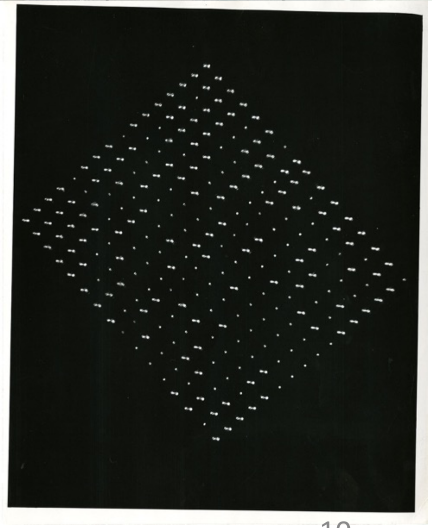
Block diagram of the mercury memory system.



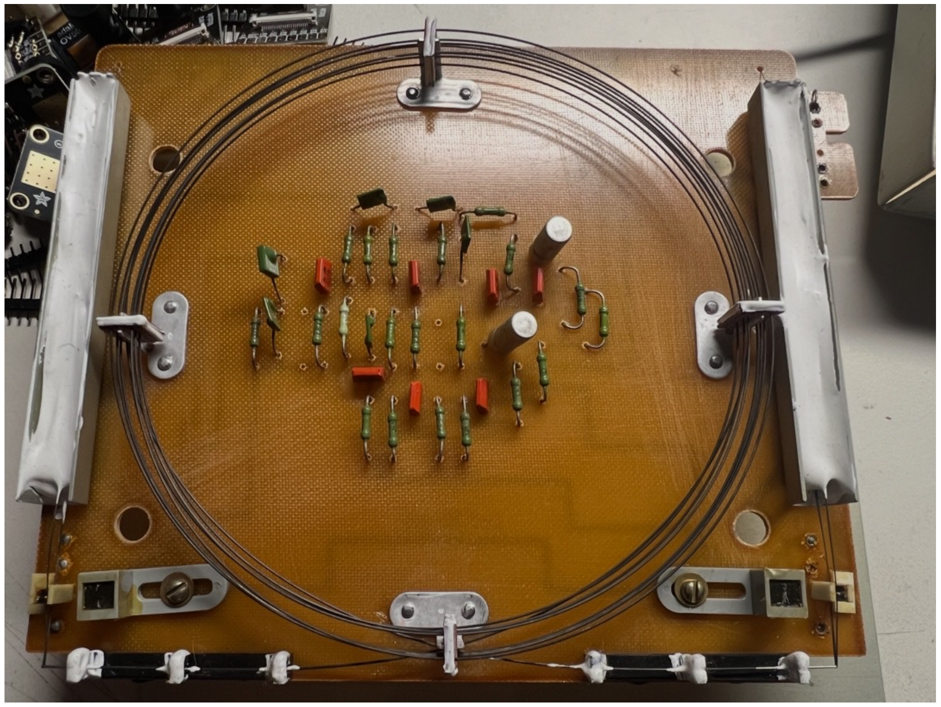
https://matsuuratomoya.com/en/works/post-past_sotsuten/

William’s Tube

- Take advantage of non-negligible decay time of phosphors on CRT to store data discussed in lecture 7
- Project data image
- Little bit later (milliseconds) recover it . Using a camera
- Either use it or re-project it for later use
- Again requires periodic refresh



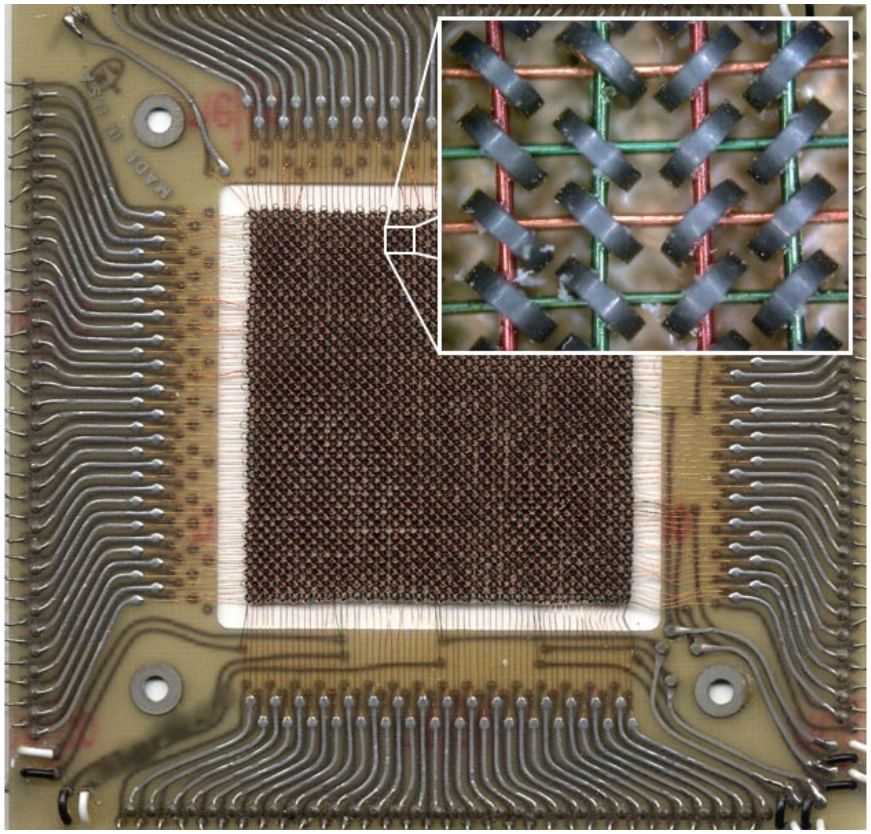
Mechanical Delay Line Memory


Soviet Delay Line Memory

- Store about 8,000 bits in the form of clockwise or counter-clockwise rotations applied to a very long piece of wire
- Required repeatedly reading them out and writing them back in like the Mercury delay line
- Turn off power twists get lost.

Core Memory

- Store 1’s and 0’s in the magnetic field of small toroids (magnetic cores)
- Where the term “core dump” comes from.
- Used up until mid 70’s
- Non *volatile*!



https://en.wikipedia.org/wiki/Magnetic-core_memory#/media/File:KL_Kernspeicher_Makro_1.jpg

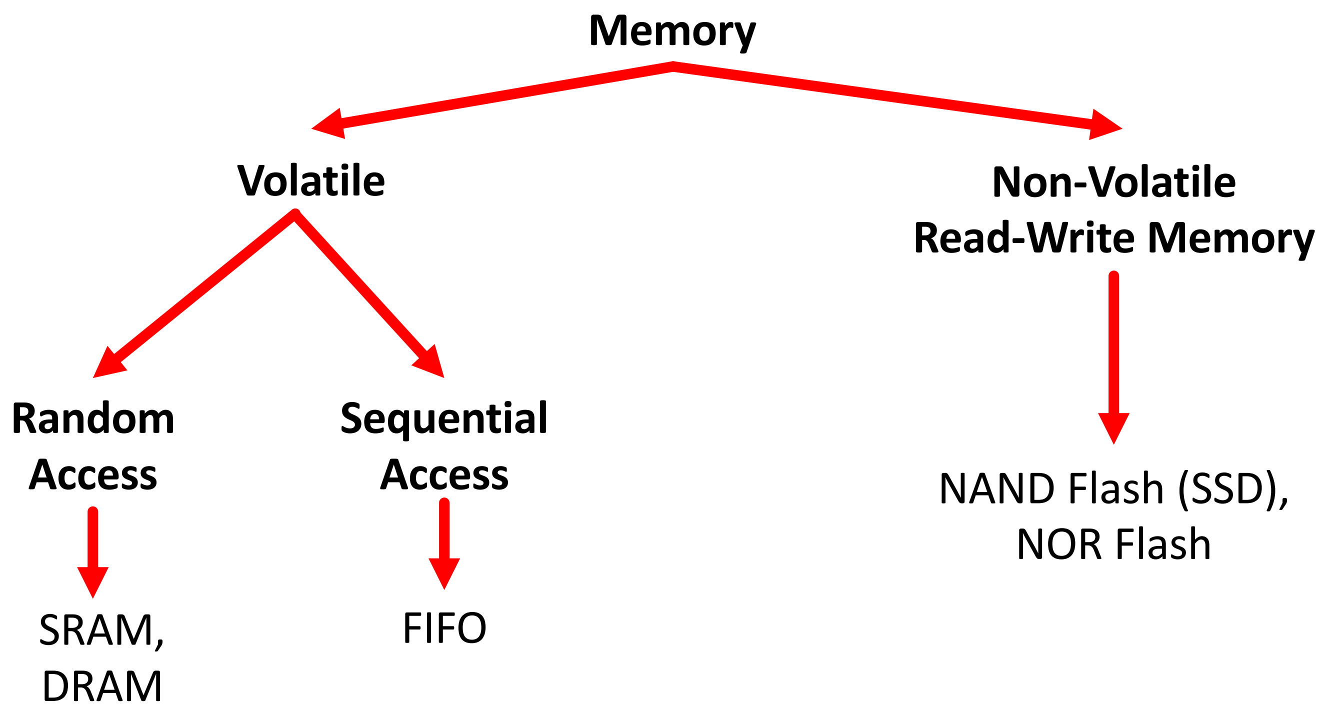
More Modern Memory

- Most modern memory uses some form of transistor-based structure to maintain data in either a long or short term

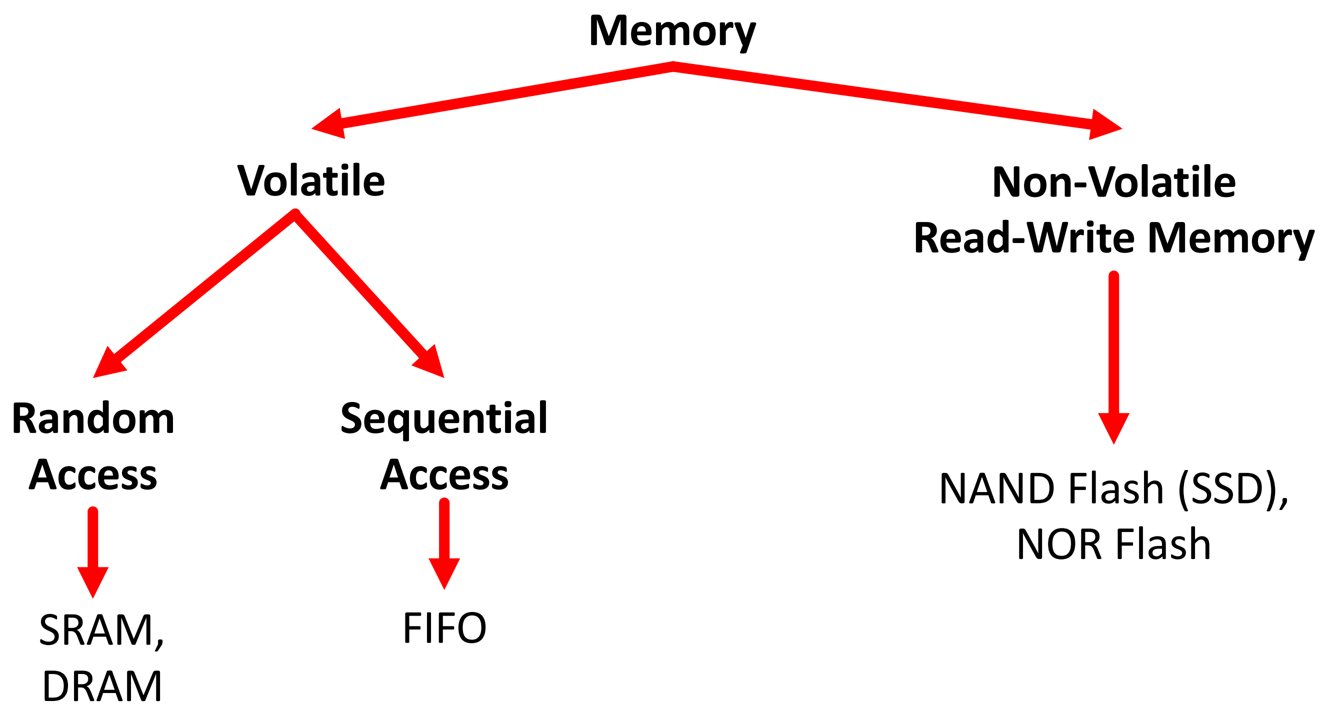
- How is it done?

- How does how it is done constrain how we can use and how much of it we have to use?

Modern Memory Classification

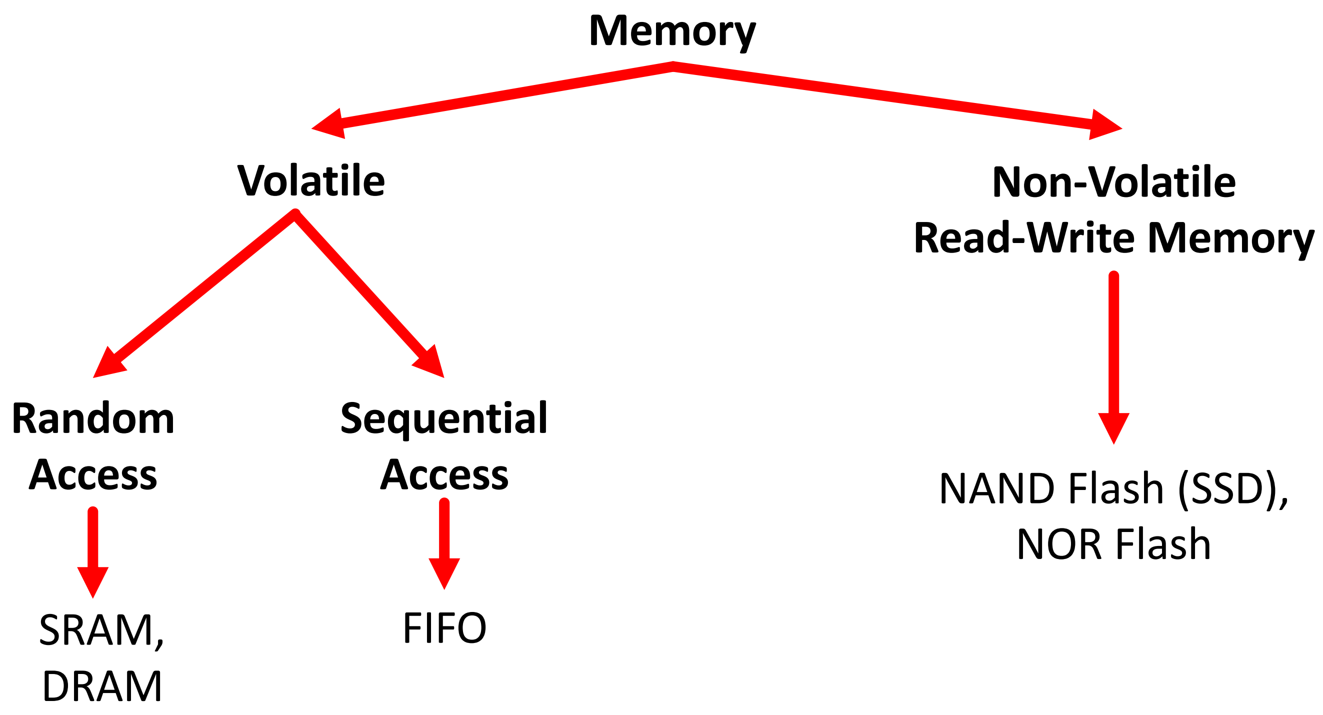


Modern Memory Classification



- **Random Access:** Give any address, get corresponding data. Access to memory need not be in a certain order
- **Sequential Access:** Put in values in an order, get them out in same order. Can't get or modify values at your desire…must wait for appropriate value to appear at ordered output (FIFO or shift buffer is an example)

Modern Memory Classification



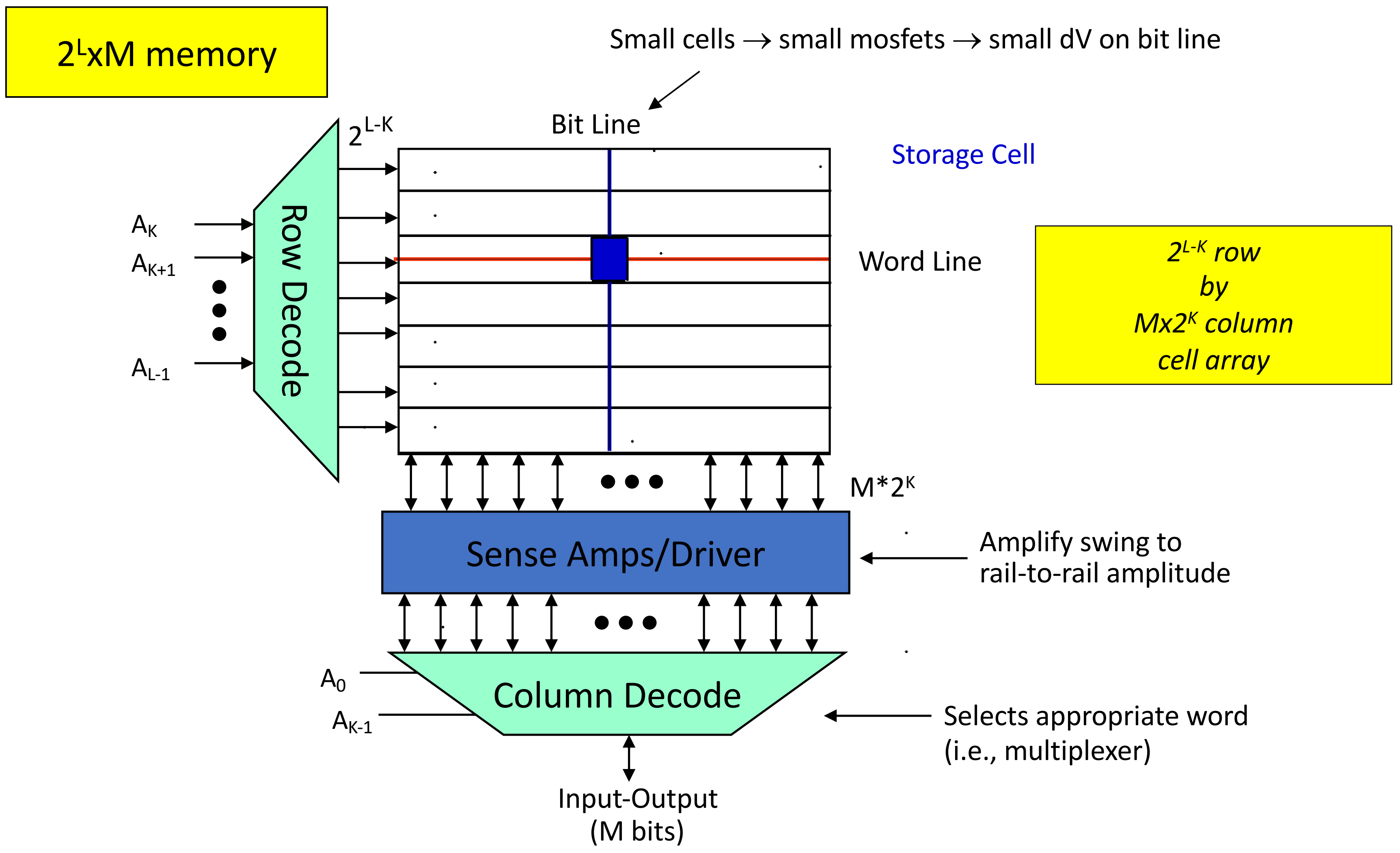
- **Volatile:** Maintains data only as long as power is applied

- **Non-Volatile:** Maintains data after power is applied!

Memory Density Tradeoff

- High-density memory technologies rarely enable “direct” access to anything inside of them.
- There’s just too many wires that would be needed and you wouldn’t be able to be very dense.
- Instead the memory-storage technology (transistors or whatever) are usually built into large grids which are accessed in a row-column format.
- This has implications for reading and writing!!!

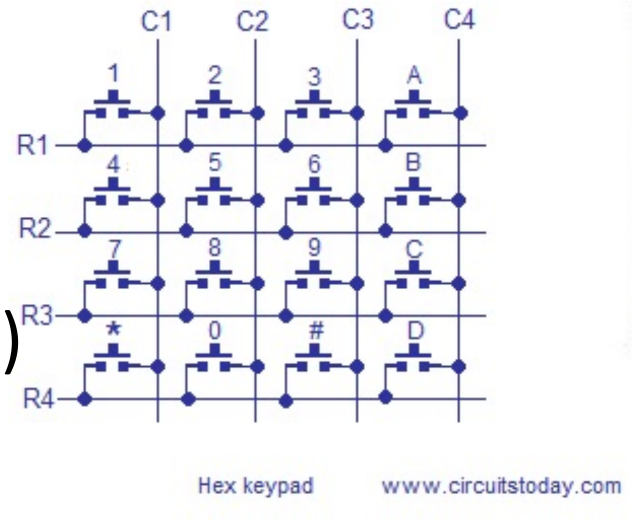
Memory Array Architecture (SRAM, Flash, DRAM)



2^LxM memory
Small cells → small mosfets → small dV on bit line
2^{L-K}
Bit Line
Storage Cell
Word Line
2^{L-K} row by $Mx2^K$ column cell array
A_K
A_{K+1}
A_{L-1}
Row Decode
$M*2^K$
Sense Amps/Driver
Amplify swing to rail-to-rail amplitude
A_0
A_{K-1}
Column Decode
Selects appropriate word (i.e., multiplexer)
Input-Output (M bits)

Memory Array's (Inspiration in Switches)

- If you have 16 switches, you can convey that using 16 independent wires (one-hot encoding)
- Alternatively if you assemble in an array/matrix, you can do with 8 wires (if you add some interfacing circuitry)

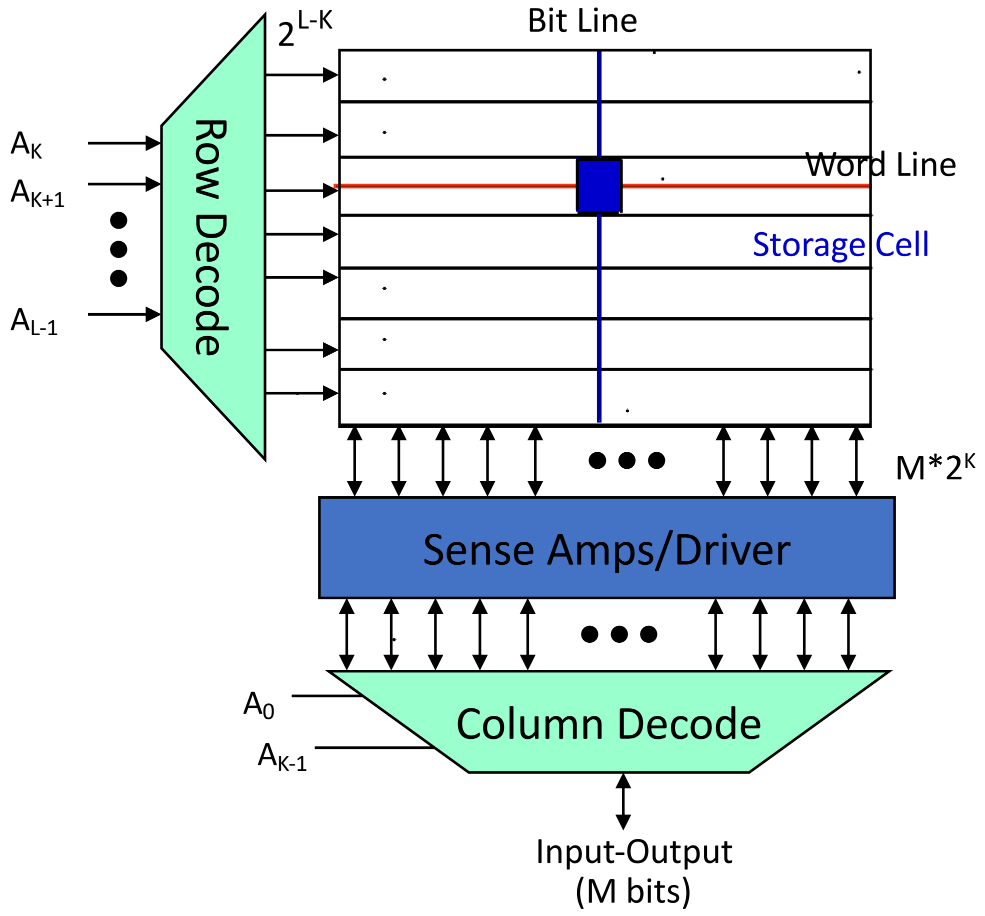


With correct interfacing you can still think of this as a 16X1 array of switches!!! Even though it isn't

- Same situation in most memory architectures

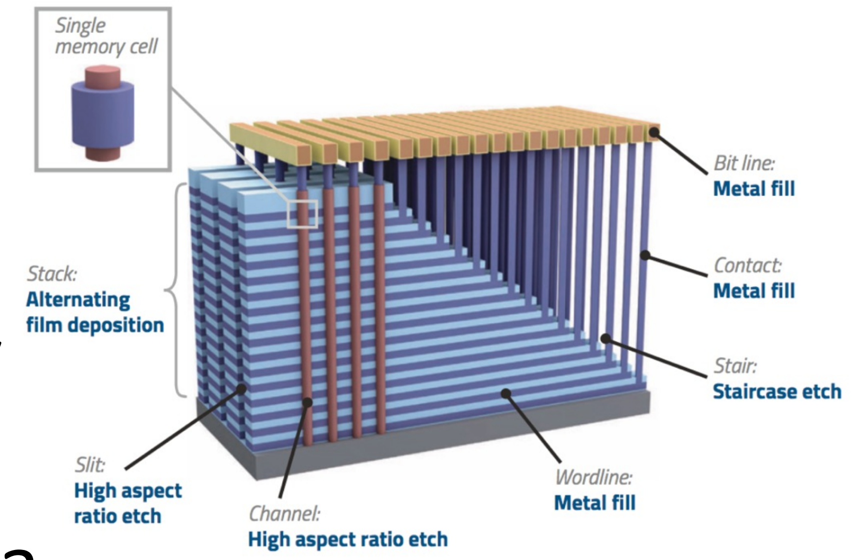
As a result…

- Can't simultaneously access multiple locations.
- In most technologies you can access one (or maybe two) entries at any point in time!
- In some layouts reading out two nearby addresses is easier/faster than reading out two addresses in different spots.



3D Memory

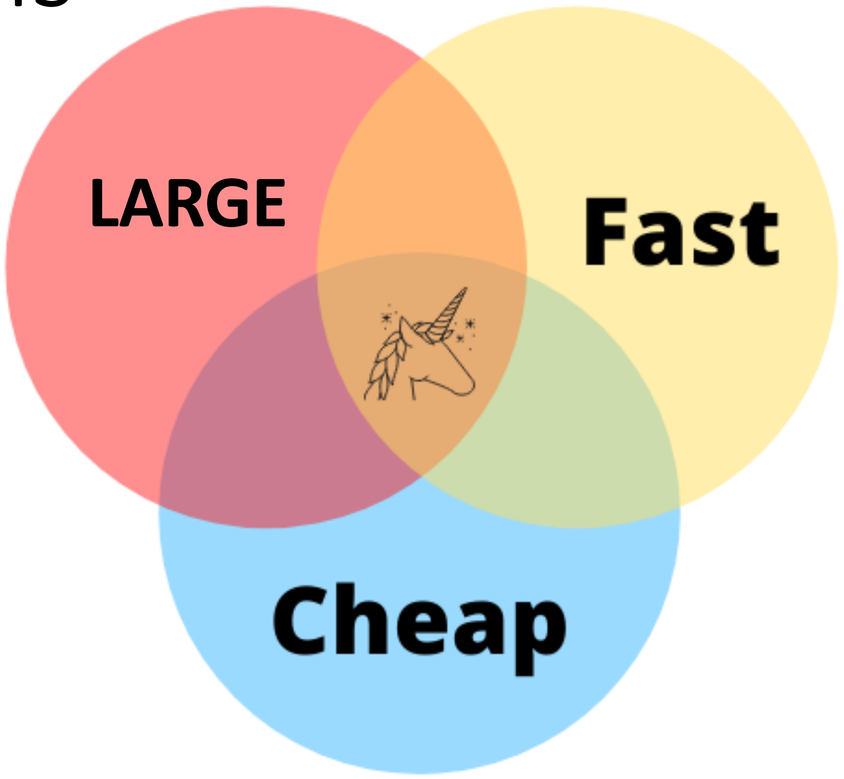
- Last decade has seen proliferation of 3D memory architectures.
- Same rough technology idea, but instead of planes, go to cubes of memory.
- Much higher densities.
- Still can only access a few spots at one time



https://sst.semiconductor-digest.com/2017/07/overcoming-challenges-in-3d-nand-volume-manufacturing/

Memory Limitations

- No memory does everything we want.
- Different types excel in different ways.
- Part of Digital Engineering is dealing with that.



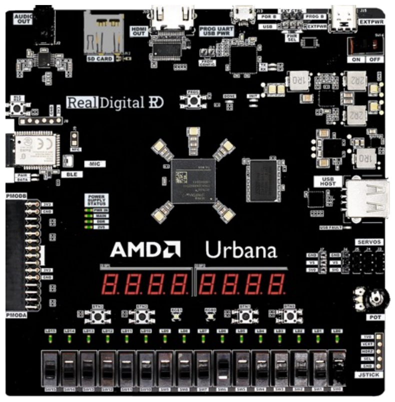
On our FPGA Board!

Inside the FPGA

- **Regular registers** in logic blocks
  - Operates at system clock speed, expensive (CLB utilization)
  - Configuration set by Verilog design (eg FIFO, single/dual port, etc)
- FPGA Distributed memory **(small SRAM)**
  - Operates at system clock speed
  - Uses LUTs (64 bits) for implementation, expensive (CLB utilization)
  - Requires significant routing for implementation
  - Configured using IP
  - Theoretical maximum: ~1Mbit
- FPGA Block RAM **(larger SRAM):**
  - 2,760K bits total (in 76/150 chunks)

Outside the FPGA

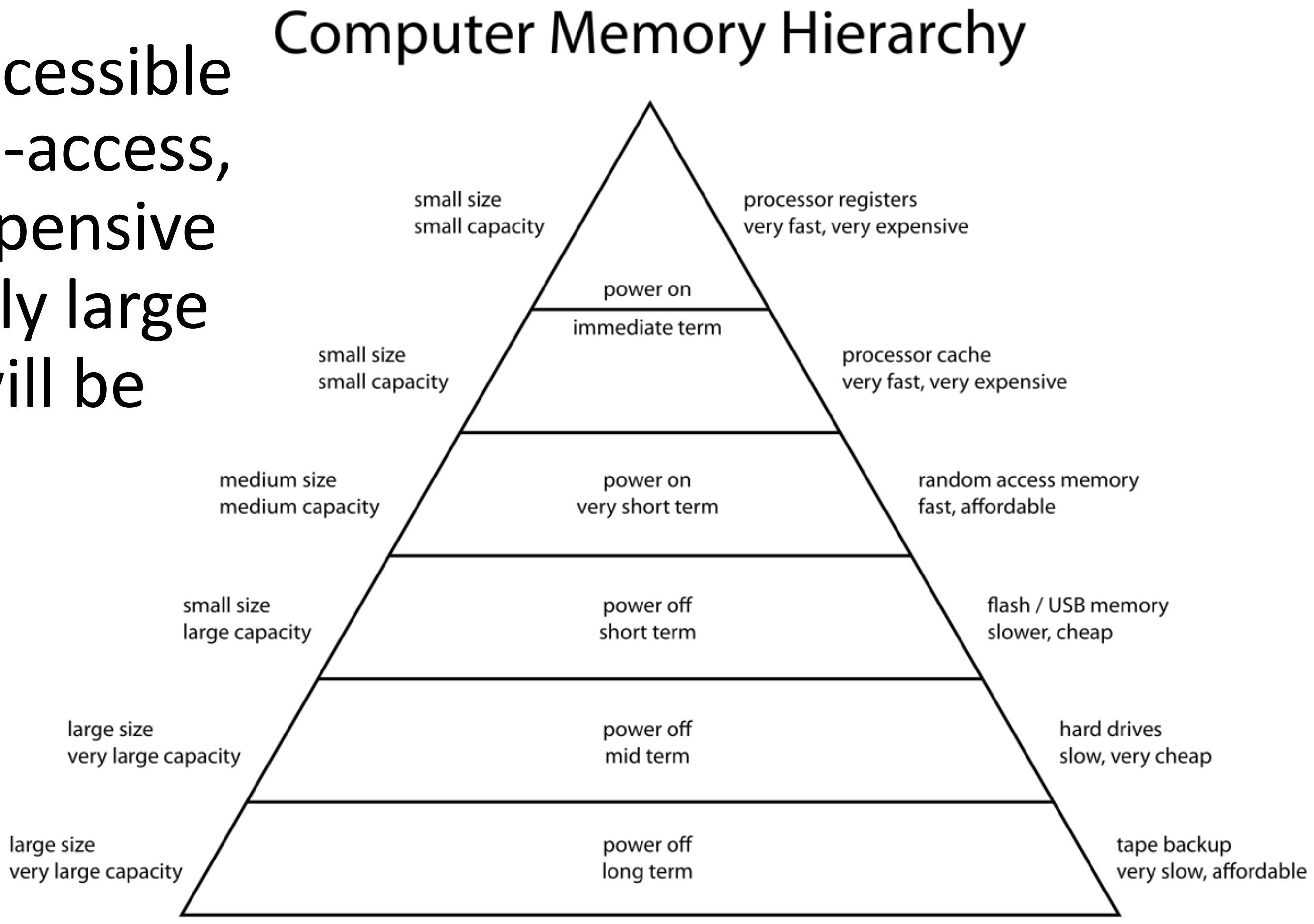
- DDR3 **SDRAM**
  - 1 GiB
  - Requires MIG (Memory Interface Generator)
- Flash memory **NAND storage**
  - 16MiB
  - Slow read access, even slower write access time!
- microSD port **larger NAND storage**
  - Different SD Card sizes (multi GB)



Notice the larger memory devices are ***outside*** *the FPGA*

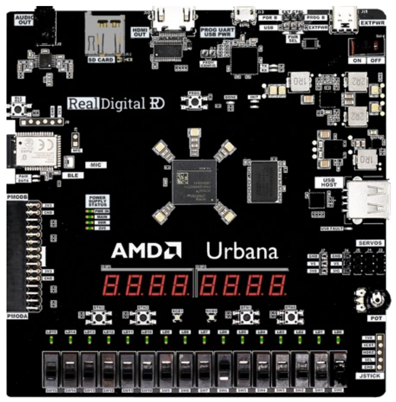
Same Issue with 6.191 Processor Design

- The more accessible and quick-to-access, the more expensive and physically large a memory will be



Good Coverage of Modern Types

- The memory types on our FPGA board provide a good coverage of *most* modern forms of digital memory, so we'll go through them now.



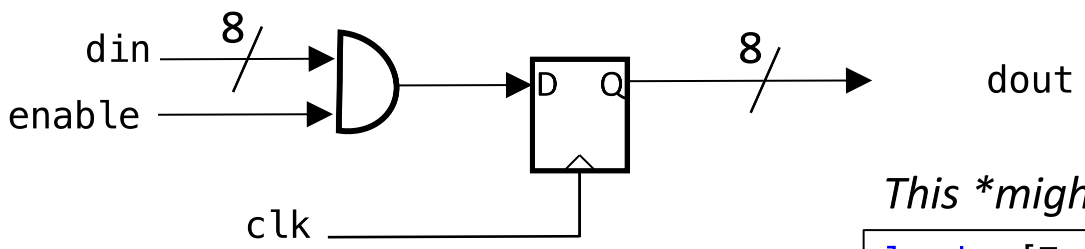
Memory ***IN*** the FPGA

FPGA Memory: Two Types

- The FPGA has two dedicated sets of resources (other than Flipflops) for storing information.
- All are comprised of SRAM (**S**tatic **R**andom-**A**ccess **M**emory)

Hold on…Aren’t FlipFloppies Memory?

- Yes, they are memory and you can use them like this:

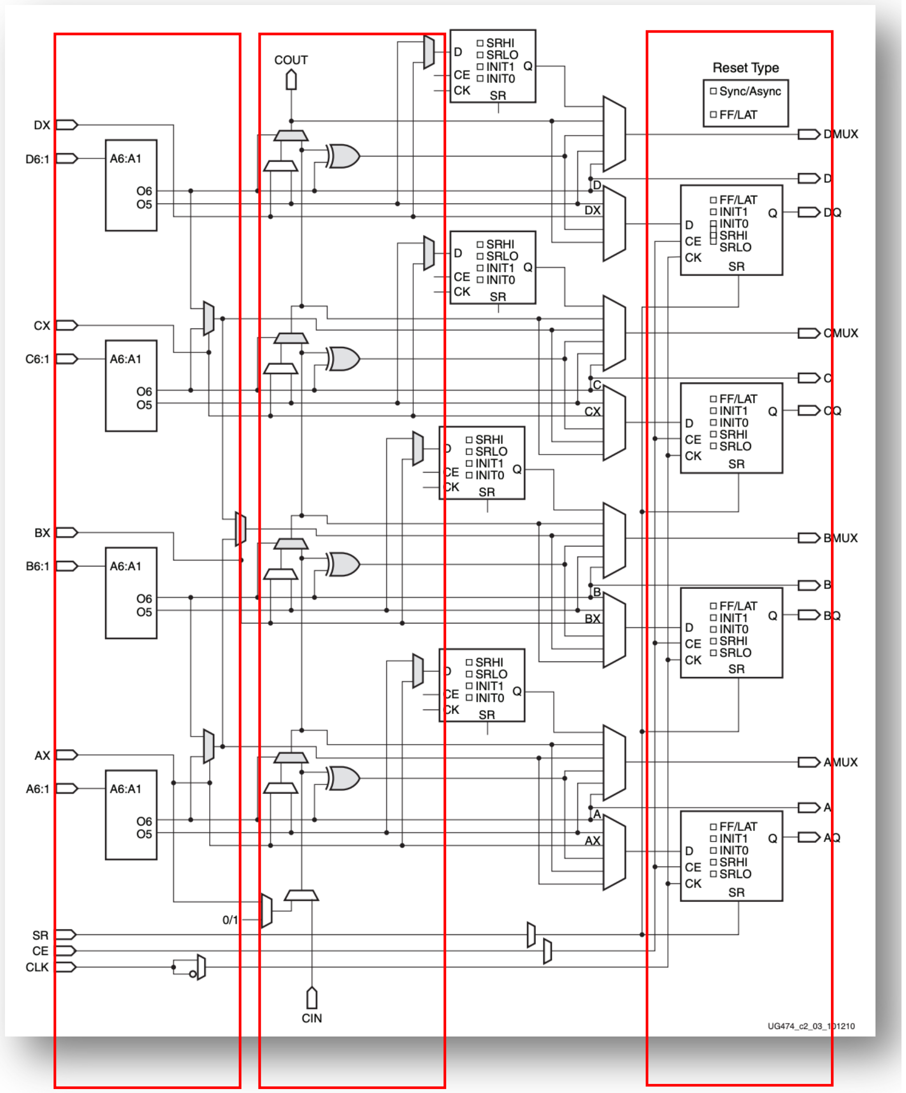


*This *might* synthesize using flip flops*

```
logic [7:0] storage;
logic [7:0] din;
logic enable;
logic [7:0] dout;
assign dout = storage;
always_ff @(posedge clk_in)begin
  if (enable)begin
    storage <= din;
  end
end
```

Flip Flops

- Flip flops are distributed all over the board in the logic cells
- Nearby for convenience
- Are meant for holding smaller temporary chunks of data
- Flip flops are not meant for bulk storage… (an image, for example)



LUTs

Used to synthesize all combinational stuff

Fast carry chain

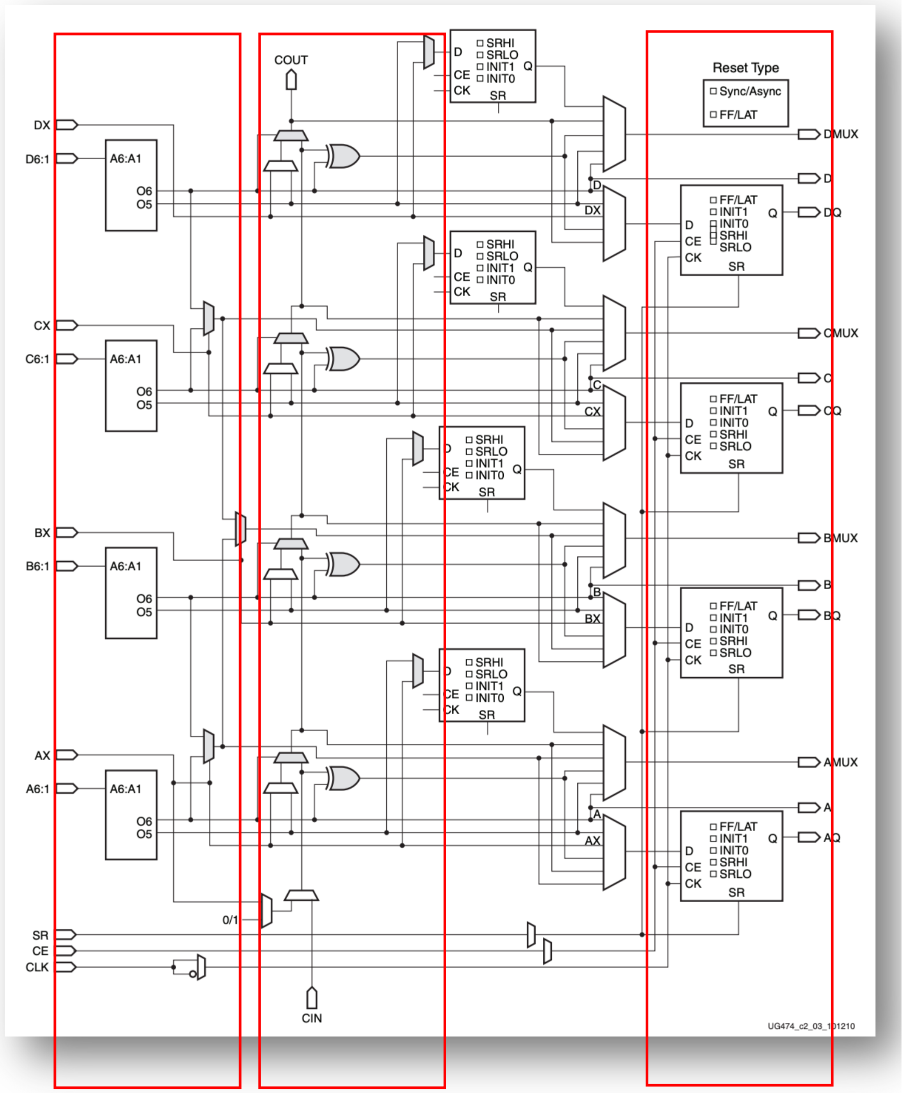
For mult-slice logic (addition, etc)

FF/Latches

Route through these for registers. Else bypass for purely combinational

Flip Flops

- Think of nearby flipflops as the registers you see in a processor
- Quick and relatively small memory access units
- Nearby so easy to route to
- Immediately accessible (not living in dense piles in which only one entry can be read at a time)
- But what about *more* memory?



LUTs
Used to synthesize all combinational stuff

Fast carry chain
For mult-slice logic (addition, etc)

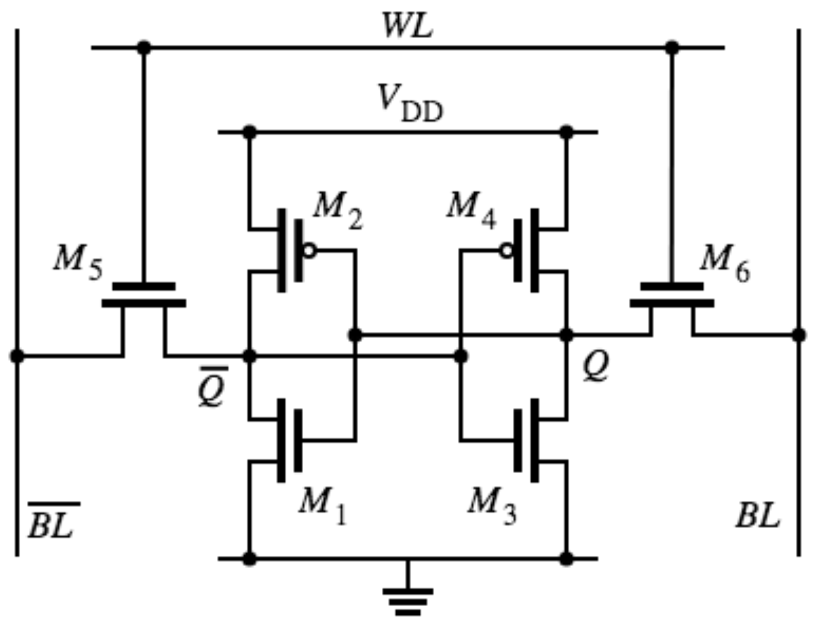
FF/Latches
Route through these for registers. Else bypass for purely combinational

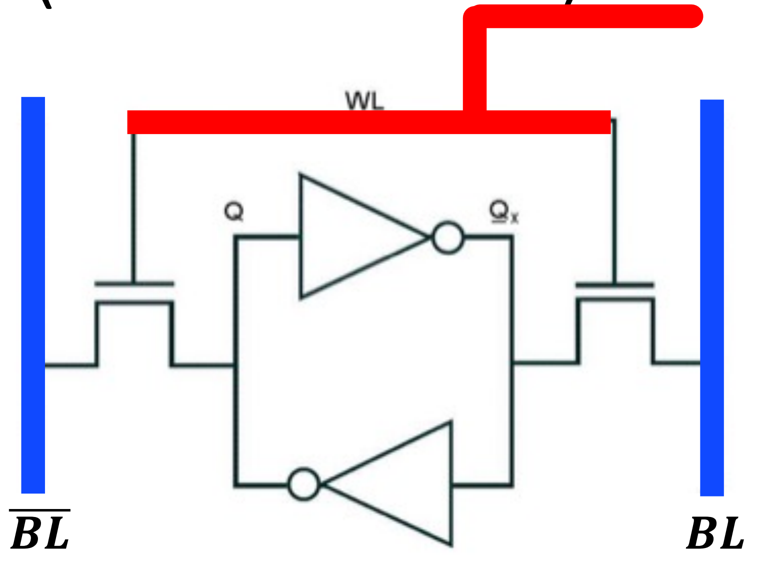
FPGA Internal Memory: Two Types

- The FPGA has two dedicated sets of resources for storing information in larger quantities
  - Block RAM
  - Distributed RAM

- Both are comprised of SRAM (Static Random-Access Memory)

Static RAM (SRAM) Cell (The 6-T Cell)





Write: Set BL, $\overline{\text{BL}}$ to (0,V_{DD}) or (V_{DD},0) then enable WL (= V_{DD})

Read: Disconnect drivers from BL and $\overline{\text{BL}}$, then enable WL (=V_{DD}). Sense a small change in BL or $\overline{\text{BL}}$

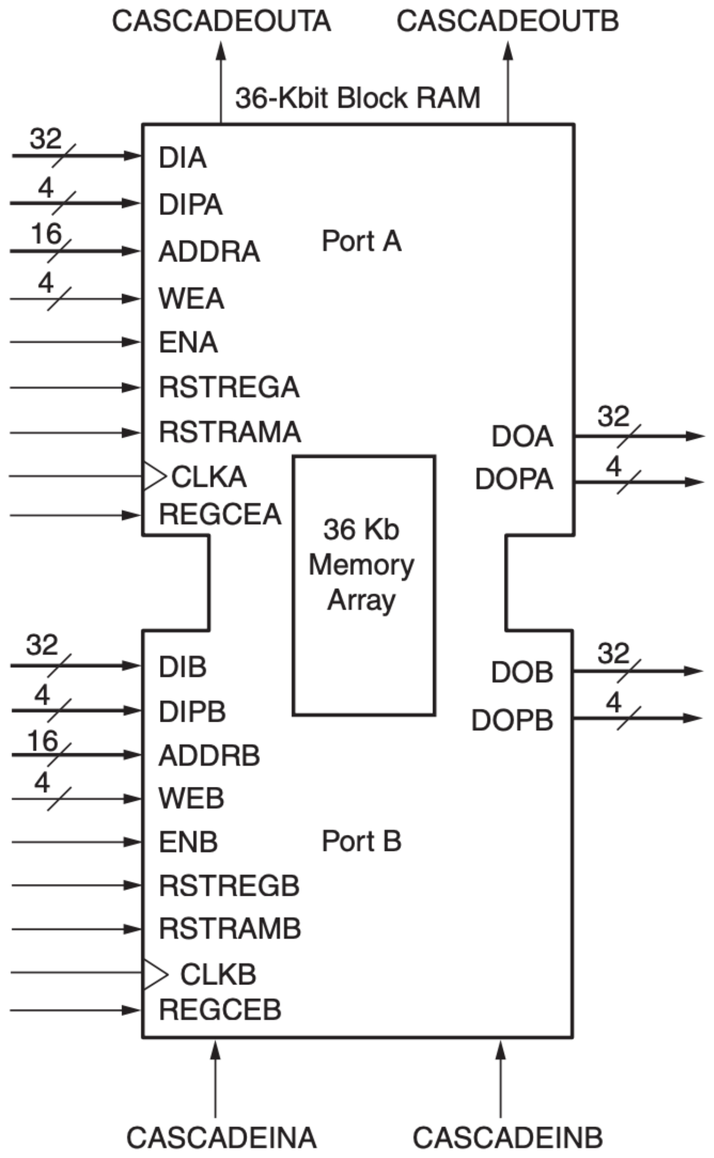
- **State held by cross-coupled inverters (M1-M4)**
- Retains state as long as power supply turned on
- Feedback must be overdriven to write into the memory

FPGA Memory: Two Types

- The SRAM in our FPGA (Xilinx 7S50T) is organized into two types (meant for using as memory explicitly):
- **Block RAM (BRAM):**
  - Large continuous chunks of SRAM
  - 36 kbits a piece
  - 75 of these on our particular FPGA
- **Distributed RAM:**
  - Of the ~32,000 LUTs on the FPGA, about 9,600 have 64 bits of SRAM in them that is usable
  - Can use this spread-out RAM as well (to squeeze another ~614.4 Kbits out of chip…but this takes away resources from your logic so you should use as last resort!

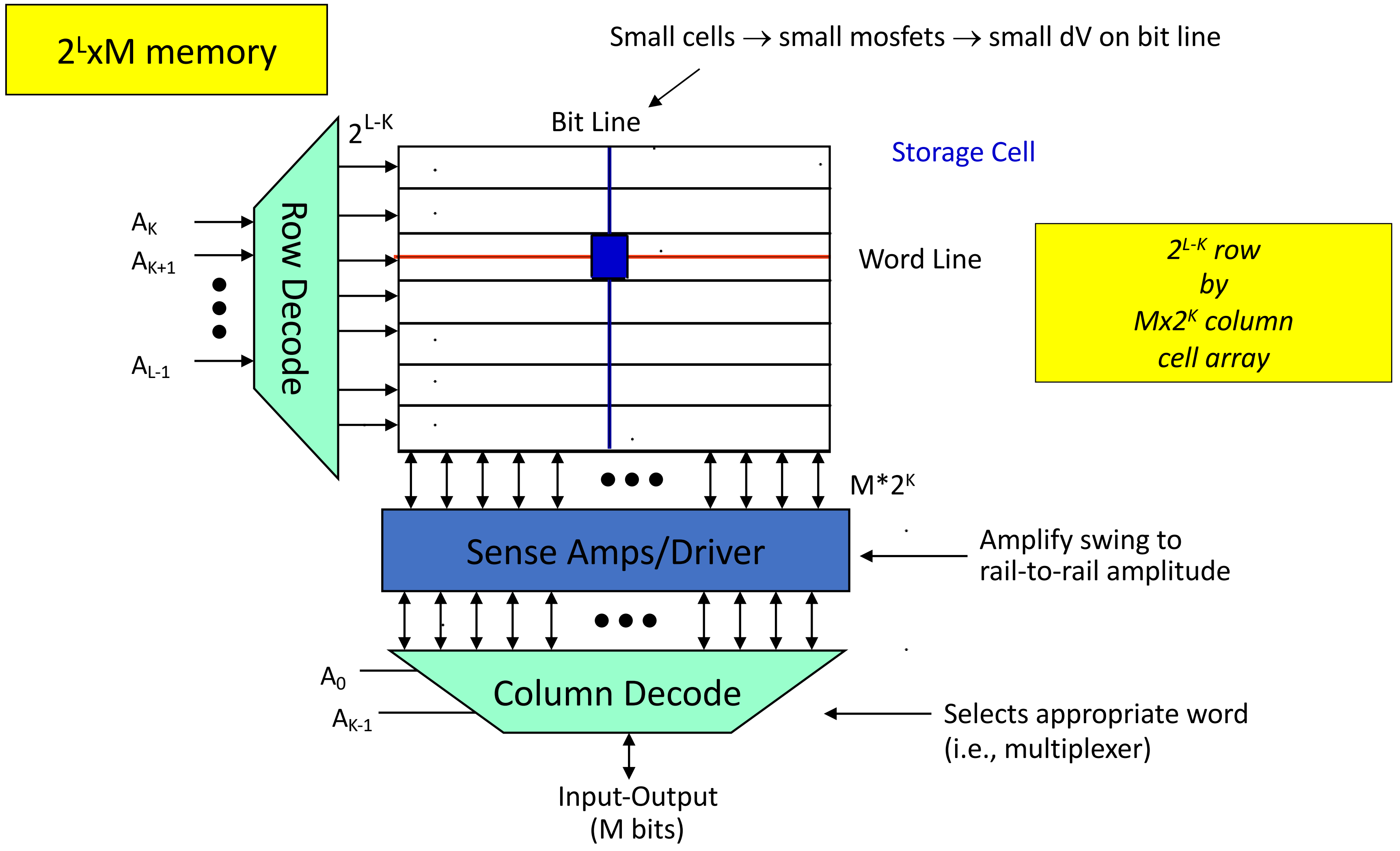
Block Memories (BRAMs)

There's 75 of these 36Kx1 bit SRAM arrays

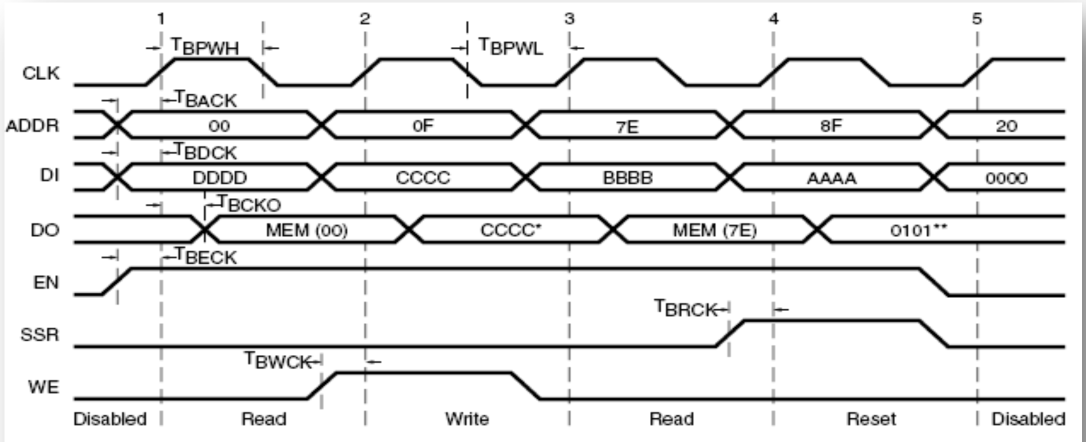


- Our FPGA has 75 dual-port SRAM modules
- Can write-to and lookup values using these two ports as needed
- Used these as audio storage in week 3, will use for video frame buffer in week 4 and beyond.

The BRAM is a dense array


2^LxM memory
Small cells → small mosfets → small dV on bit line
2^{L-K}
Bit Line
Storage Cell
Word Line
2^{L-K} row by $Mx2^K$ column cell array
A_K
A_{K+1}
A_{L-1}
Row Decode
$M*2^K$
Sense Amps/Driver
Amplify swing to rail-to-rail amplitude
A_0
A_{K-1}
Column Decode
Selects appropriate word (i.e., multiplexer)
Input-Output (M bits)

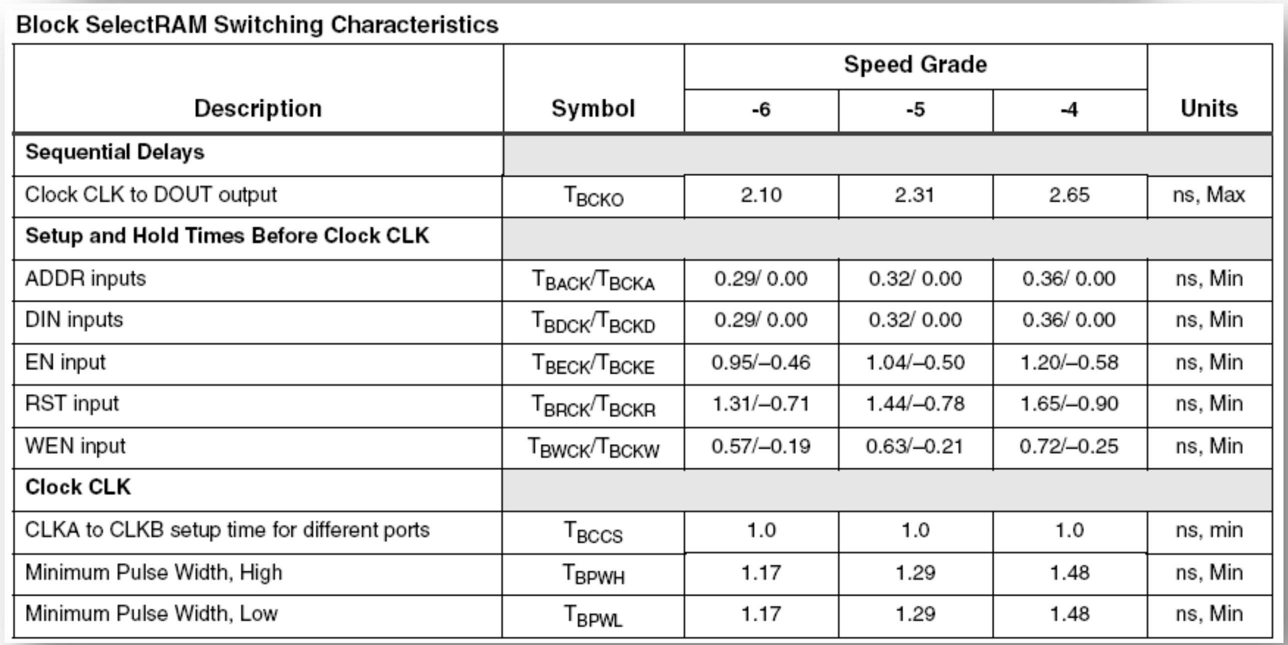
BRAM Timing



* Write Mode = "WRITE_FIRST"
** SRVAL = 0101

Block SelectRAM Timing Diagram

At best, a Block RAM will never provide asynchronous reads. You can get synchronous reads with a one-clock cycle delay

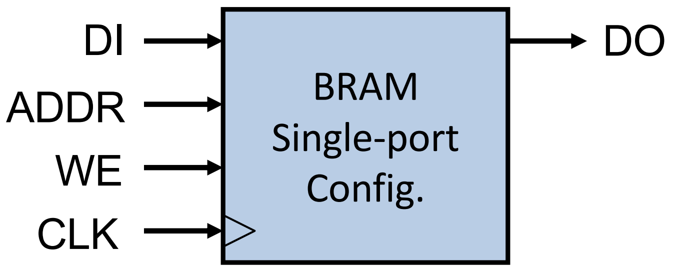


Block SelectRAM Switching Characteristics

Description	Symbol	Speed Grade			Units
		-6	-5	-4	
Sequential Delays					
Clock CLK to DOUT output	T_{BCKO}	2.10	2.31	2.65	ns, Max
Setup and Hold Times Before Clock CLK					
ADDR inputs	T_{BACK}/T_{BCKA}	0.29/ 0.00	0.32/ 0.00	0.36/ 0.00	ns, Min
DIN inputs	T_{BDCK}/T_{BCKD}	0.29/ 0.00	0.32/ 0.00	0.36/ 0.00	ns, Min
EN input	T_{BECK}/T_{BCKE}	0.95/–0.46	1.04/–0.50	1.20/–0.58	ns, Min
RST input	T_{BRCK}/T_{BCKR}	1.31/–0.71	1.44/–0.78	1.65/–0.90	ns, Min
WEN input	T_{BWCK}/T_{BCKW}	0.57/–0.19	0.63/–0.21	0.72/–0.25	ns, Min
Clock CLK					
CLKA to CLKB setup time for different ports	T_{BCCS}	1.0	1.0	1.0	ns, min
Minimum Pulse Width, High	T_{BPWH}	1.17	1.29	1.48	ns, Min
Minimum Pulse Width, Low	T_{BPWL}	1.17	1.29	1.48	ns, Min

It is strongly recommended to use them with a two-cycle delay though!

BRAM Operation



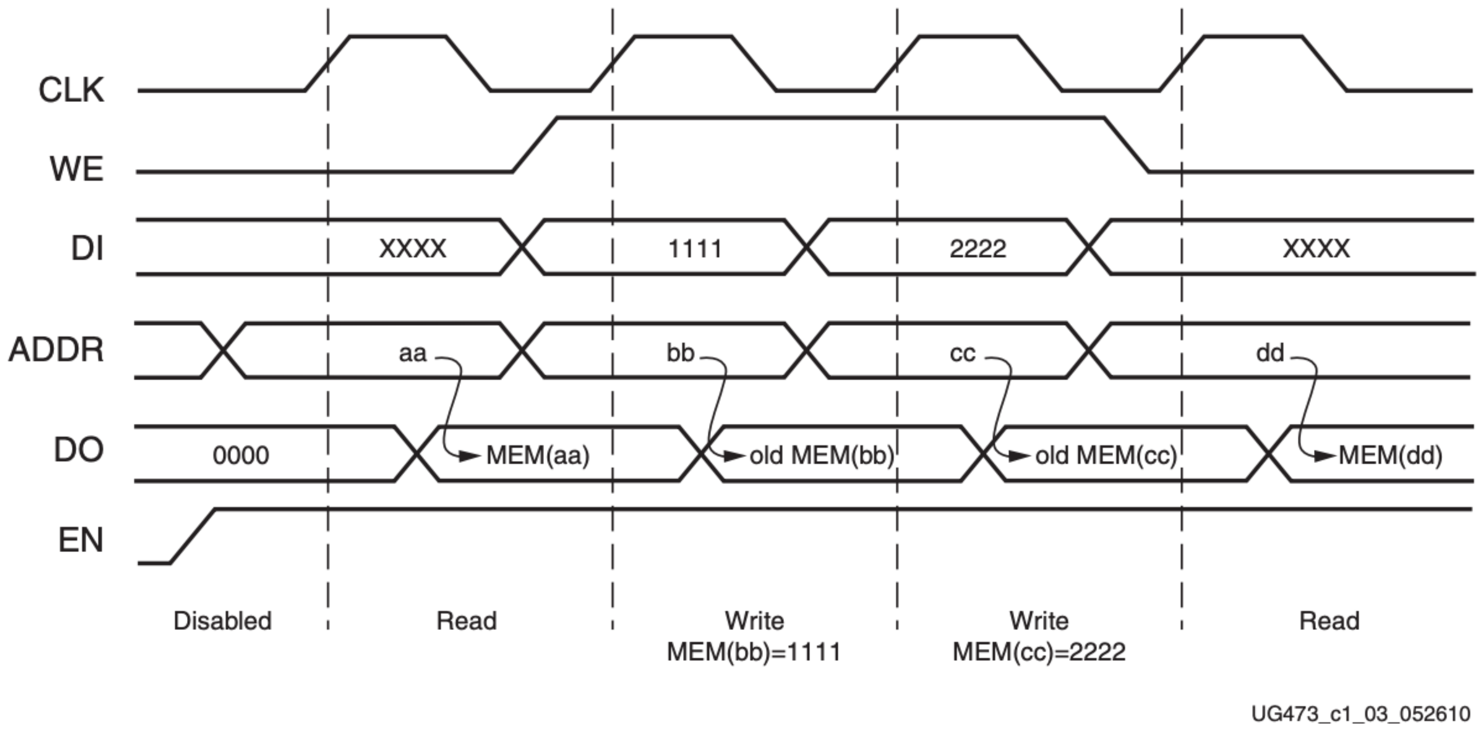


Figure 1-3: **READ_FIRST Mode Waveforms**

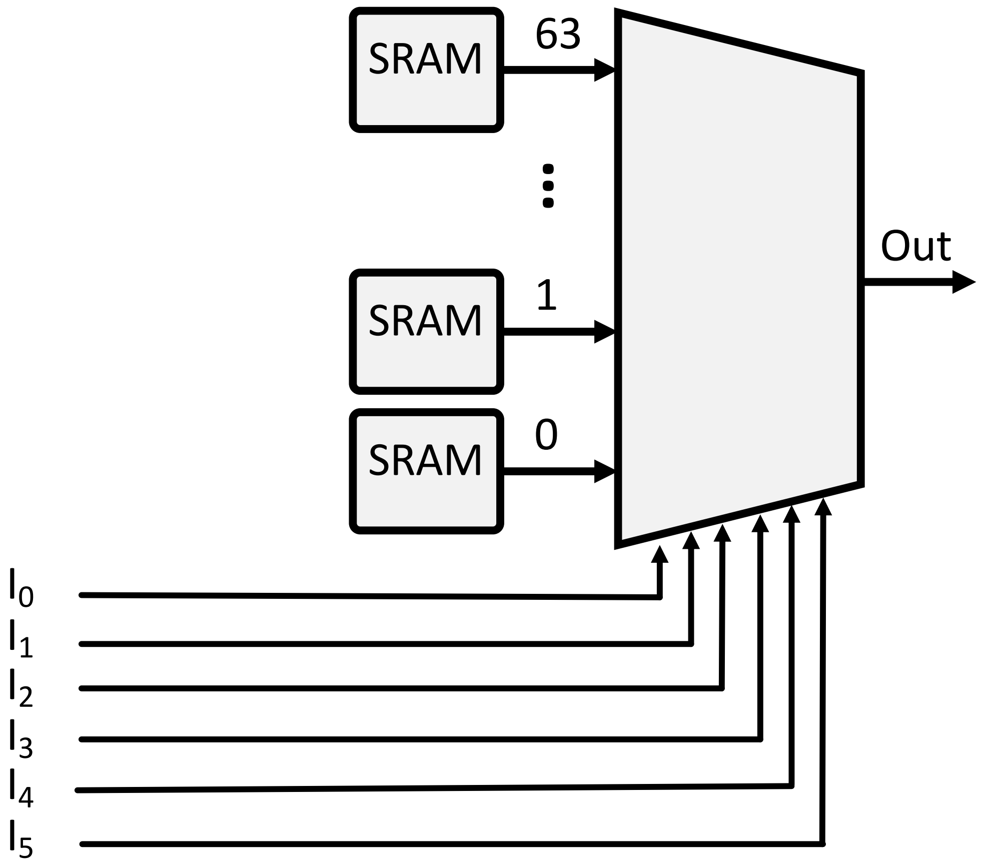
FPGA Memory: Two Types

- The SRAM in our FPGA (Xilinx 7S50T) is organized into two types (meant for using as memory explicitly):
- **Block RAM (BRAM):**
  - Large continuous chunks of SRAM
  - 36 kbits a piece
  - 75 of these on our particular FPGA
- **Distributed RAM:**
  - Of the ~32,000 LUTs on the FPGA, about 9,600 have 64 bits of SRAM in them that is usable for general memory.
  - Can use this spread-out RAM as well (to squeeze another ~614.4 Kbits out of chip…but this takes away resources from your logic so you should use as last resort!

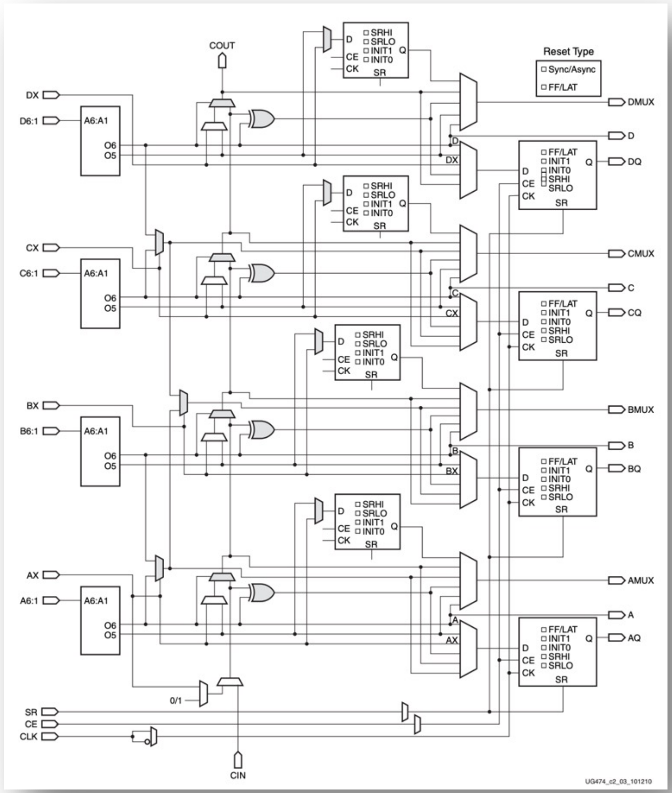
<u>Distributed RAM:</u> Each **Logic Cell** is made of Four Six-Input <u>L</u>ook<u>u</u>p <u>T</u>ables with inputs that can be set

- These LUTs are programmed to give us our logic functions and that program is set in SRAM...they can therefore synthesize any six-input lookup-table/function/Karnaugh Map

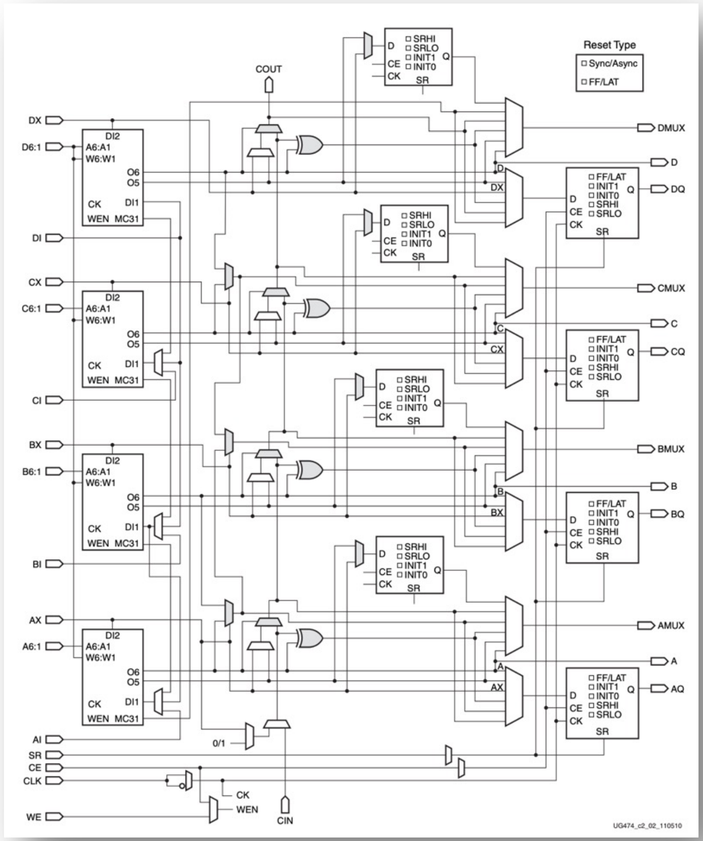
- In some logic cells, you can alternatively use this SRAM for regular generic memory!



SliceL vs. Slice M



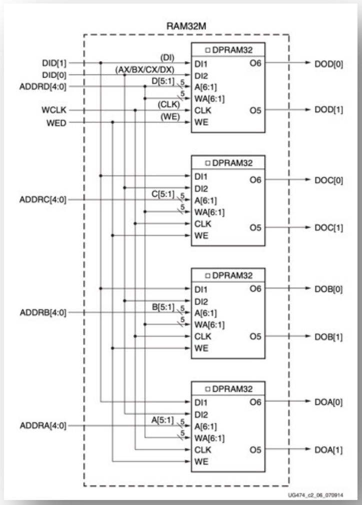
SliceL
LUTs programmed when bitfile written



SliceM
Memory used in LUT programming broken out and available

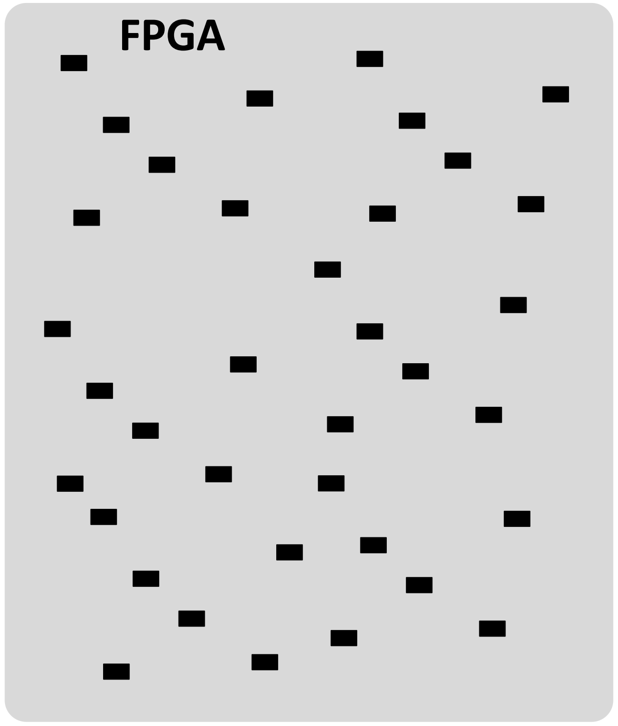
Distributed RAM is Distributed

- Each 64 bits of LUT specification is broken out…so each Slice (with Four LUT6's) has 256 bits of RAM



Four of these per slice

SliceM's are *distributed all over the FPGA:*



Distributed RAM vs. Block RAM

- Distributed RAM:
  - More flexible:
    - Smaller unit size (256 bits)
    - read multiple (>2) values at once
    - Single-cycle reads/writes
- Block RAM:
  - Less flexible:
    - Bigger unit size (18kbits or 36 kbits)
    - Dual-port ONLY (<=2)…can avoid using one if you want
    - Risky single-cycle reads/writes
    - No single-cycle bulk reset

Small Memory in Verilog

- 8X256 memory:
  - Synchronous write
  - *Asynchronous aka ~~combinational~~* read

```
module mem_one( input wire clk,
                input wire [7:0] w_idx, //write address
                input wire [7:0] din, //write value
                input wire we, //write enable
                input wire [7:0] r_idx, //read address
                output logic [7:0] dout); //read value

  logic [7:0] memory [0:255];
  always_ff @(posedge clk)begin
    if (we)begin
      memory[w_idx] <= din;
    end
  end
  assign dout = memory[r_idx];
endmodule
```

Result of Design:

- Because of what was asked, Vivado “inferred” the usage of Distributed RAM
- Why is this using 32 LUTs?

```
2. Slice Logic Distribution
---------------------------

+------------------------------------------+------+-------+------------+-----------+-------+
|                 Site Type                | Used | Fixed | Prohibited | Available | Util% |
+------------------------------------------+------+-------+------------+-----------+-------+
| Slice                                    |    8 |     0 |          0 |      8150 |  0.10 |
|   SLICEL                                 |    0 |     0 |            |           |       |
|   SLICEM                                 |    8 |     0 |            |           |       |
| LUT as Logic                             |    0 |     0 |          0 |     32600 |  0.00 |
| LUT as Memory                            |   32 |     0 |          0 |      9600 |  0.33 |
|   LUT as Distributed RAM                 |   32 |     0 |            |           |       |
|     using O5 output only                 |    0 |       |            |           |       |
|     using O6 output only                 |   32 |       |            |           |       |
|     using O5 and O6                      |    0 |       |            |           |       |
|   LUT as Shift Register                  |    0 |     0 |            |           |       |
| Slice Registers                          |    0 |     0 |          0 |     65200 |  0.00 |
|   Register driven from within the Slice  |    0 |       |            |           |       |
|   Register driven from outside the Slice |    0 |       |            |           |       |
| Unique Control Sets                      |    1 |       |          0 |      8150 |  0.01 |
+------------------------------------------+------+-------+------------+-----------+-------+
```

From post_place_util.rpt

Small Memory

- 8X256 memory:
  - Synchronous write
  - ***Synchronous*** read

```
module mem_one( input wire clk,
                input wire [7:0] w_idx, //write address
                input wire [7:0] din, //write value
                input wire we, //write enable
                input wire [7:0] r_idx, //read address
                output logic [7:0] val); //read value

  logic [7:0] memory [0:255];
  always_ff @(posedge clk)begin
    if (we)begin
      memory[w_idx] <= din;
    end
    dout <= memory[r_idx]; //moved
  end
endmodule
```

Result of Design

- No Longer Building with Distributed RAM, Instead Vivado Chose a Block RAM (because it has a tendency to choose BRAM when provided the option)

```
2. Slice Logic Distribution
---------------------------

+-------------------------------------------+------+-------+------------+-----------+-------+
|                 Site Type                 | Used | Fixed | Prohibited | Available | Util% |
+-------------------------------------------+------+-------+------------+-----------+-------+
| Slice                                     |    0 |     0 |          0 |      8150 |  0.00 |
|   SLICEL                                  |    0 |     0 |            |           |       |
|   SLICEM                                  |    0 |     0 |            |           |       |
| LUT as Logic                              |    0 |     0 |          0 |     32600 |  0.00 |
| LUT as Memory                             |    0 |     0 |          0 |      9600 |  0.00 |
|   LUT as Distributed RAM                  |    0 |     0 |            |           |       |
|   LUT as Shift Register                   |    0 |     0 |            |           |       |
| Slice Registers                           |    0 |     0 |          0 |     65200 |  0.00 |
|   Register driven from within the Slice   |    0 |       |            |           |       |
|   Register driven from outside the Slice  |    0 |       |            |           |       |
| Unique Control Sets                       |    0 |       |          0 |      8150 |  0.00 |
+-------------------------------------------+------+-------+------------+-----------+-------+
```

```
3. Memory
---------

+-------------------+------+-------+------------+-----------+-------+
|     Site Type     | Used | Fixed | Prohibited | Available | Util% |
+-------------------+------+-------+------------+-----------+-------+
| Block RAM Tile    |  0.5 |     0 |          0 |        75 |  0.67 |
|   RAMB36/FIFO*    |    0 |     0 |          0 |        75 |  0.00 |
|   RAMB18          |    1 |     0 |          0 |       150 |  0.67 |
|     RAMB18E1 only |    1 |       |            |           |       |
+-------------------+------+-------+------------+-----------+-------+
```

Small Memory

- 8X256 memory:
  - Synchronous write
  - ***Synchronous*** read
  - ***Bulk Resettable***

Only new thing compared to before is we can erase memory in one clock cycle

```
module mem_three( input wire clk,
                  input wire rst,
                  input wire [7:0] w_idx,
                  input wire [7:0] din,
                  input wire we,
                  input wire [7:0] r_idx,
                  output logic [7:0] dout);

  logic [7:0] memory [0:255];
  always_ff @(posedge clk)begin
    if (rst)begin
      for (int i=0; i<256; i=i+1)begin
        memory[i] <=0;
      end
    end else if (we)begin
      memory[w_idx] <= din;
    end
    dout <= memory[r_idx];
  end
endmodule
```

Results…

- Uh oh…

Being able to reset everything instantly is not possible with almost any denser memory technology easly. This design achieves this "performance" using the actual flip flops at massive expense of resources

```
2. Slice Logic Distribution
---------------------------

+-------------------------------------------+------+-------+------------+-----------+-------+
|                 Site Type                 | Used | Fixed | Prohibited | Available | Util% |
+-------------------------------------------+------+-------+------------+-----------+-------+
| Slice                                     | 1043 |     0 |          0 |      8150 | 12.80 |
|   SLICEL                                  |  718 |     0 |            |           |       |
|   SLICEM                                  |  325 |     0 |            |           |       |
| LUT as Logic                              |  841 |     0 |          0 |     32600 |  2.58 |
|   using O5 output only                    |    0 |       |            |           |       |
|   using O6 output only                    |  802 |       |            |           |       |
|   using O5 and O6                         |   39 |       |            |           |       |
| LUT as Memory                             |    0 |     0 |          0 |      9600 |  0.00 |
|   LUT as Distributed RAM                  |    0 |     0 |            |           |       |
|   LUT as Shift Register                   |    0 |     0 |            |           |       |
| Slice Registers                           | 2056 |     0 |          0 |     65200 |  3.15 |
|   Register driven from within the Slice   |    8 |       |            |           |       |
|   Register driven from outside the Slice  | 2048 |       |            |           |       |
|     LUT in front of the register is unused | 1697 |       |            |           |       |
|     LUT in front of the register is used   |  351 |       |            |           |       |
| Unique Control Sets                       |  257 |       |          0 |      8150 |  3.15 |
+-------------------------------------------+------+-------+------------+-----------+-------+


3. Memory
---------

+----------------+------+-------+------------+-----------+-------+
|    Site Type   | Used | Fixed | Prohibited | Available | Util% |
+----------------+------+-------+------------+-----------+-------+
| Block RAM Tile |    0 |     0 |          0 |        75 |  0.00 |
|   RAMB36/FIFO* |    0 |     0 |          0 |        75 |  0.00 |
|   RAMB18       |    0 |     0 |          0 |       150 |  0.00 |
+----------------+------+-------+------------+-----------+-------+
```

Distributed RAM vs. Block RAM

- **Distributed RAM:**
  - Trading logic for memory
  - Occupies a logic cell whenever you use it (more you use, the fewer logic cells you have for logic)
  - For large data structures/memories, needs to use lots of memory units (can be hard to route)
  - Too much usage can make it hard to place and route (long builds!)
- **Block RAM:**
  - Meant to be memory and nothing else
  - Relatively dense!
  - Use it or lose it. It is there…if you don’t use it, you don’t get free logic instead so you should use it.
  - For large things, will outperform distributed RAM in speed/latency
  - Generally won’t constrain place and route like distributed RAM usage will!

FFs vs. Distributed RAM vs. Block RAM

- Conclusions:
- Flip Flops (avoid for “memory” at all costs):
  - *Very small things…local variables (state, math, etc…)*
- Distributed RAM:
  - Small things (16 bit shift register), small memories of a few hundred bytes, few entries,
  - Design actually needs async/quick reads*
  - etc…
  - These are all best implemented with Distributed RAM:
- Block RAM:
  - Large things (images, large audio files, etc), large buffers, all best implemented with Block RAMs

for good reasons…not because you are lazy and don’t feel like dealing with issues.* ***Seriously, you will lose points on final projects if you make poor or lazy memory choices.

How do We Specify Which to Use?

- We do it with how we write Verilog.
- These “wants” will lead to the following inferences:
  - Async/quick reads?→Distributed RAM
  - Bulk resets? → FF’s
- Already showed some examples
- In reality memory of any medium-large size has delays with associated with it and we need to learn to expect that and build it into our designs
- Must have Verilog properly reflect that!

Building a Block RAM

- Use Verilog in a very particular way, Vivado can confidently “infer” block RAM usage
- Use pre-provided Verilog modules:
- These are guaranteed to simulate properly (right amount of delay and other issues)

```
module xilinx_single_port_ram_read_first #(
parameter RAM_WIDTH = 18, // Specify RAM data width
parameter RAM_DEPTH = 1024, // Specify RAM depth (number of entries)
parameter RAM_PERFORMANCE = "HIGH_PERFORMANCE", // Select "HIGH_PERFORMANCE" or "LOW_LATENCY"
parameter INIT_FILE = "" // Specify name/location of RAM initialization file if using one (leave blank if
not)
) (
input [clogb2(RAM_DEPTH-1)-1:0] addra, // Address bus, width determined from RAM_DEPTH
input [RAM_WIDTH-1:0] dina, // RAM input data
input clka, // Clock
input wea, // Write enable
input ena, // RAM Enable, for additional power savings, disable port when not in use
input rsta, // Output reset (does not affect memory contents)
input regcea, // Output register enable
output [RAM_WIDTH-1:0] douta // RAM output data
);

reg [RAM_WIDTH-1:0] BRAM [RAM_DEPTH-1:0];
reg [RAM_WIDTH-1:0] ram_data = {RAM_WIDTH{1'b0}};
```

Building a Block RAM

- If we use our Verilog in a very particular way, Vivado can confidently “infer” block RAM usage

```
xilinx_single_port_ram_read_first #(
     .RAM_WIDTH(8), // Specify RAM data width
     .RAM_DEPTH(WIDTH*HEIGHT*2), // Specify RAM depth (number of entries)
     .RAM_PERFORMANCE("HIGH_PERFORMANCE"), // Select "HIGH_PERFORMANCE" or "LOW_LATENCY"
     .INIT_FILE(`FPATH(image2.mem)) // Specify name/location of RAM initialization)
  image_brom (
     .addra(image_addr), // Address bus, width determined from RAM_DEPTH
     .dina(0), // RAM input data, width determined from RAM_WIDTH
     .clka(pixel_clk_in), // Clock
     .wea(0), // Write enable
     .ena(1), // RAM Enable, for additional power savings, disable port when not in use
     .rsta(rst_in), // Output reset (does not affect memory contents)
     .regcea(1), // Output register enable
     .douta(palette_lookup) // RAM output data, width determined from RAM_WIDTH
  );
```

This (week 05) leads to this usage:

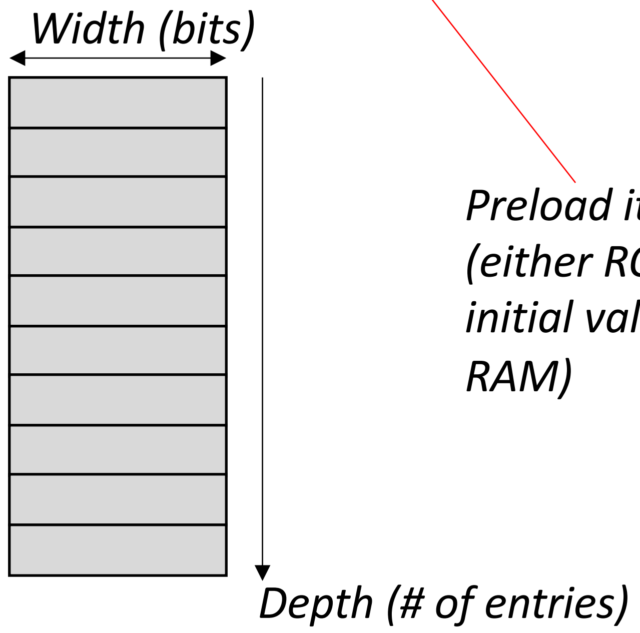
```
0. AAAAAA
---------

+-------------------+------+-------+-----------+----------+-------+
|     AAAA AAAA     | AAAA | AAAAA | AAAAAAAAAA | AAAAAAAAA | AAAA% |
+-------------------+------+-------+-----------+----------+-------+
| AAAAA AAA AAAA    | 00.0 |     0 |          0 |        00 | 00.00 |
|   AAAA00/AAAA*    |   00 |     0 |          0 |        00 | 00.00 |
|     AAAA00A0 AAAA |   00 |       |            |           |       |
|   AAAA00          |    0 |     0 |          0 |       000 |  0.00 |
|     AAAA00A0 AAAA |    0 |       |            |           |       |
+-------------------+------+-------+-----------+----------+-------+
```

Using a Block RAM

- If we use our Verilog in a very particular way, Vivado can confidently "infer" block RAM usage

```
xilinx_single_port_ram_read_first #(
     .RAM_WIDTH(8),
     .RAM_DEPTH(WIDTH*HEIGHT),
     .RAM_PERFORMANCE("HIGH_PERFORMANCE"), // "HIGH_PERFORMANCE" or "LOW_LATENCY"
     .INIT_FILE(`FPATH(image2.mem))
```



Preload it with files (either ROM or initial values of RAM)

*"High Performance" means more latency
But this also means it is much easier to build this design (fitting it into timing)*

Block RAM Uses

- Store Images (Week 5) :using as a ROM in that case
- Store Video (Week 5…second half): Frame buffer
- Store Audio (Week 3)
- Clock Domain Crossing (Week 5, 6, 7):
    - With dual-port RAM, you can write with one clock and read with another (can specify some settings to prevent race conditions, though no 100% guarantee)
- Store *anything you want! (rest of life)*

SRAM Summary

- Block RAM (and less so Distributed RAM) should be your first choice in storing information
- Quick and reliable:
  - Want measurement? Ask for it and get it:
    - One cycle later (some Distributed RAM configs)
    - One or two cycles later from BRAM
- Limited amounts of it on FPGA (0.5 Mbyte of BRAM, ~amount of Dist RAM)

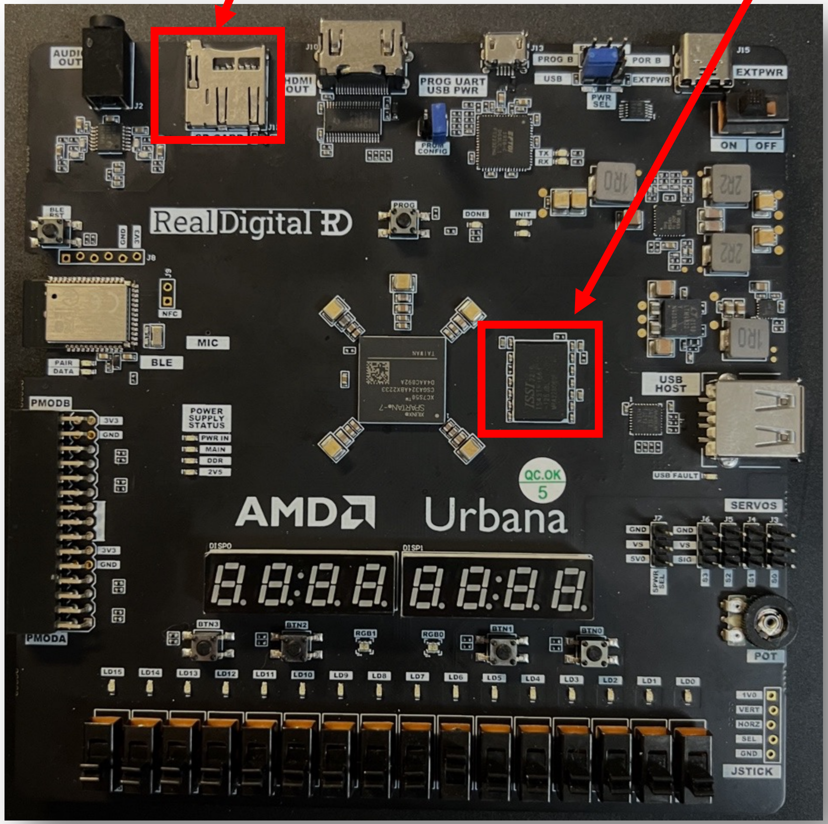
Memory ***<u>OFF</u>*** the FPGA

Off-FPGA Memories

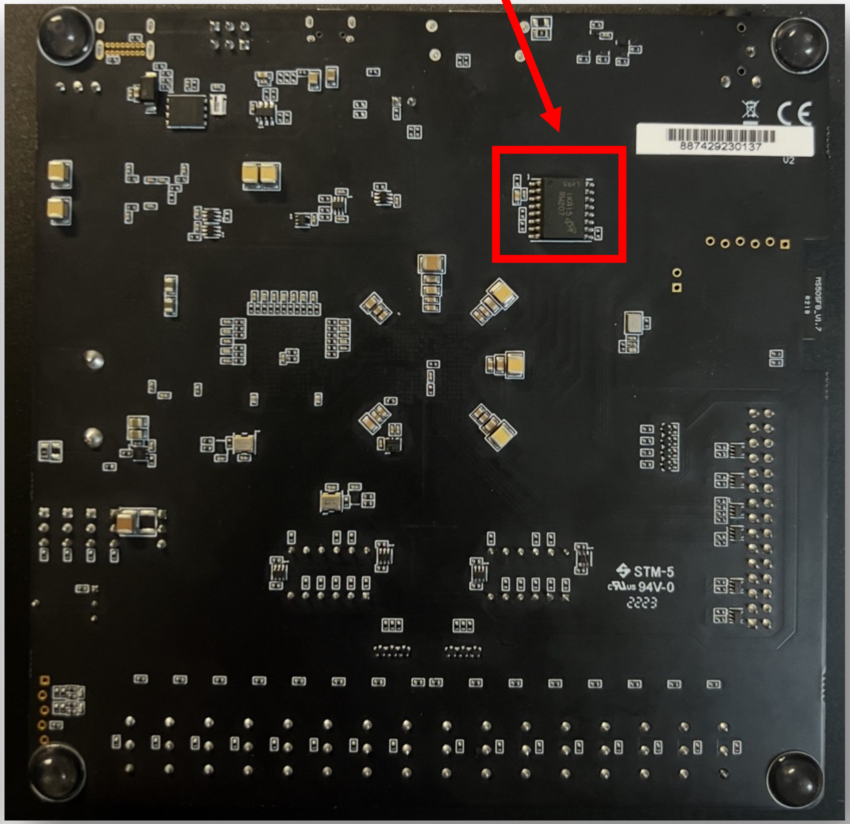
SD card (NAND Flash)

DDR3
Dynamic Memory

EEPROM



front



back

Two-ish Major Off-FPGA Options

- Flash/EEPROM:
  - Many different form factors, very slow to read/write, but **non-volatile**, meaning it will last beyond power cycles

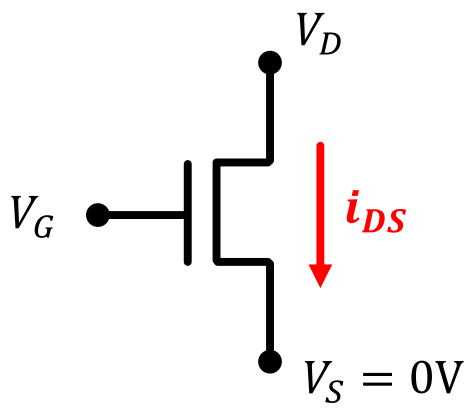
- Dynamic Random Access Memory (DRAM):
  - Potentially very high read-write rates
  - Needs to be constantly refreshed (dynamic)
  - Volatile…~100 ms after power-off, memory lost

EPROM Families

- Includes EPROM, EEPROM, Flash memory, (and SSDs)
- Utilize **Floating Gates**
- Different from SRAM!
- Instead of ~6 transistors per bit, you can do about 1 or so!
- Acts sorta like SRAM from outside but ***Non-Volatile*** and writes are *much* slower than reads
- Invented by Dov Frohman while at Intel ~1970ish

An early EPROM.
You'd program electrically and then shine UV onto it to erase it…don't use these anymore

Quick Review on MOSFETs



In sub-threshold mode:

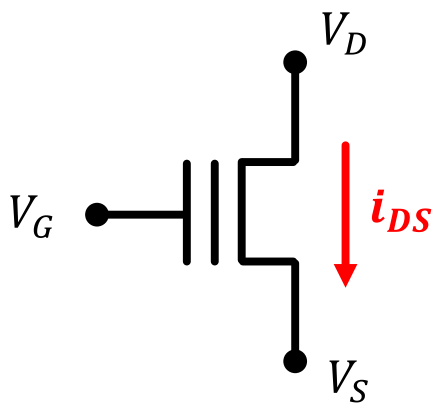
$$i_{DS} = K\left((V_G - V_T)V_D - \frac{V_D^2}{2}\right)$$

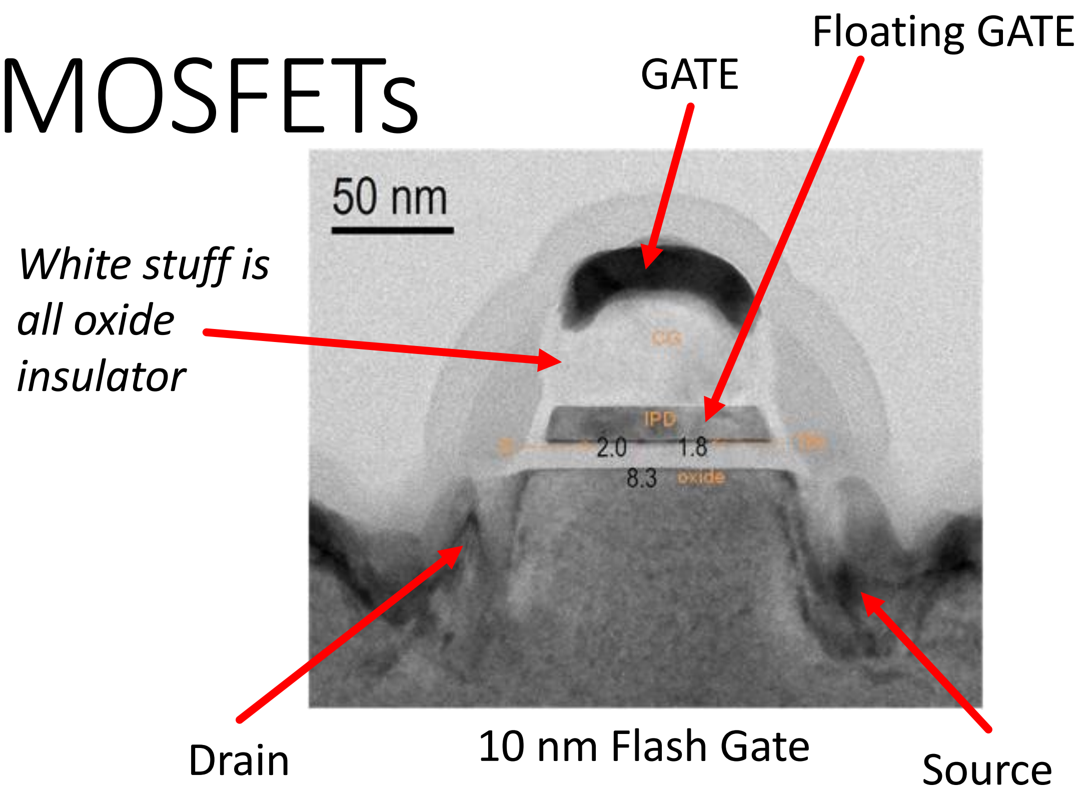
Above Threshold (saturation)

$$i_{DS} = K(V_G - V_T)^2$$

- Basically:
  - If V_G is > V_T you conduct (are ”on”)
  - If V_G is < V_T you do not conduct (are ”off”)
- Traditionally V_T is a function of doping, transistor dimensions, etc…
- BUT!....

Floating Gate MOSFETs





Presence or absence of carriers on floating gate affects the threshold voltage of MOSFET

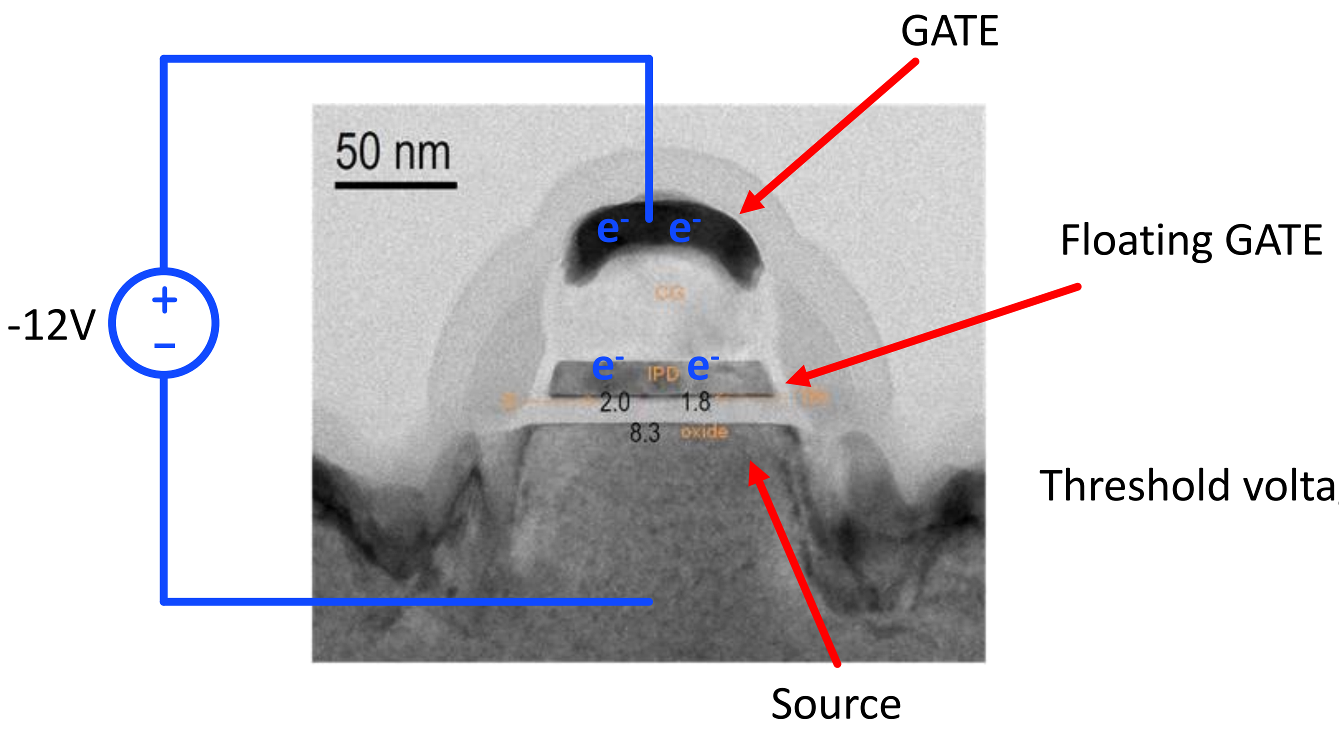
- Default ("binary 1")…Threshold voltage is lower V_{TL}
  - (no electrons trapped in gate)
- Programmed bit ("binary 0")…threshold voltage is higher V_{TH}
  - (electrons trapped in gate)

https://www.electronicsweekly.com/news/research-news/device-rd/iedm-hybrid-floating-gate-scales-flash-to-10nm-2012-12/

Hot Carrier Injection/Tunneling to Program/Reprogram

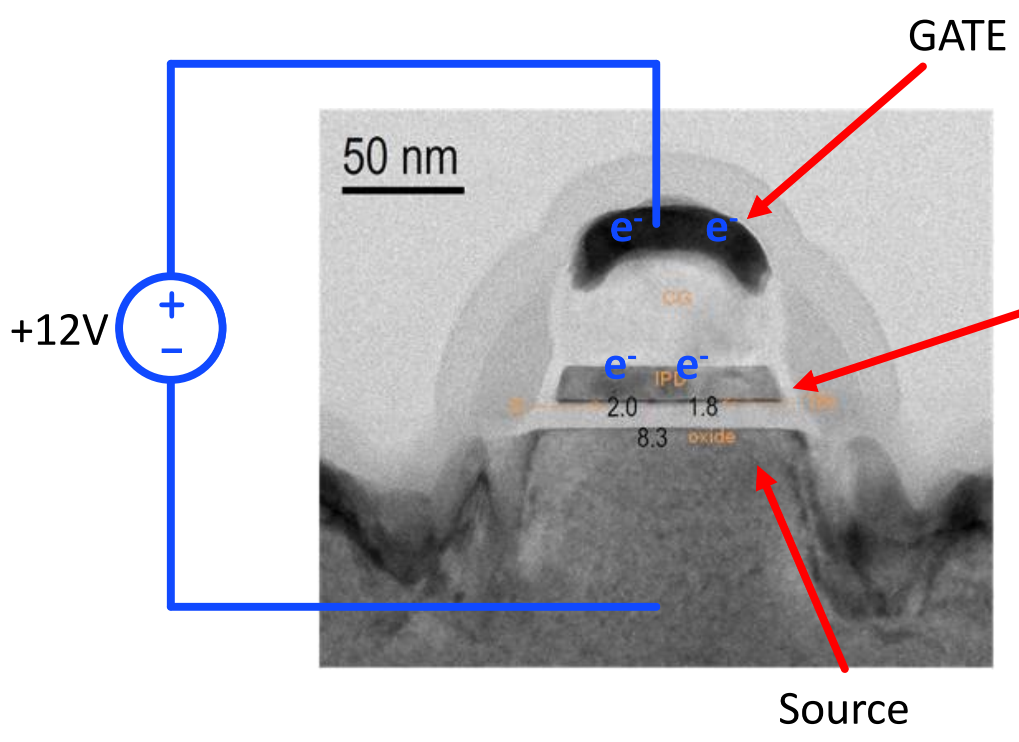
- To add or remove electrons to the floating gate you use a quantum tunneling phenomenon
- High voltage (~12V over 100’s of Angstroms) is used to force electrons to tunnel into floating gate… the term “hot” refers to high energies on electrons.
- A similar process is used in reverse to tunnel them out again
- High voltage is a potentially destructive process and will eventually ruin the device. Flash traditionally therefore has limits of ~ several 100,000’s of program/erase cycles
- Mitigate issues by wear-leveling (try to spread out usage across all of device…like rotating tires on a car)

To Program a 0



Threshold voltage is now Higher

To Program a 1 aka erase

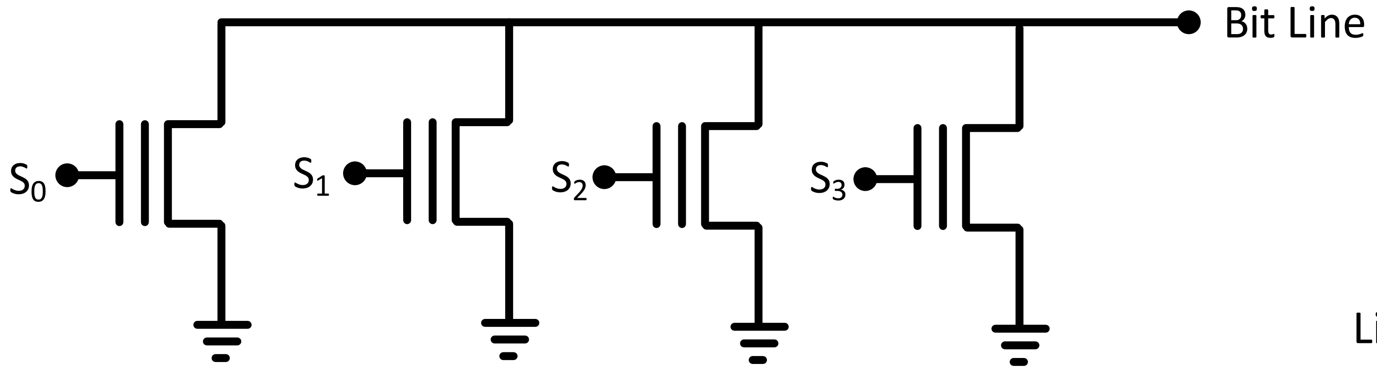


Threshold voltage is now Lower

NOR Flash

1 on the select bits means voltage:

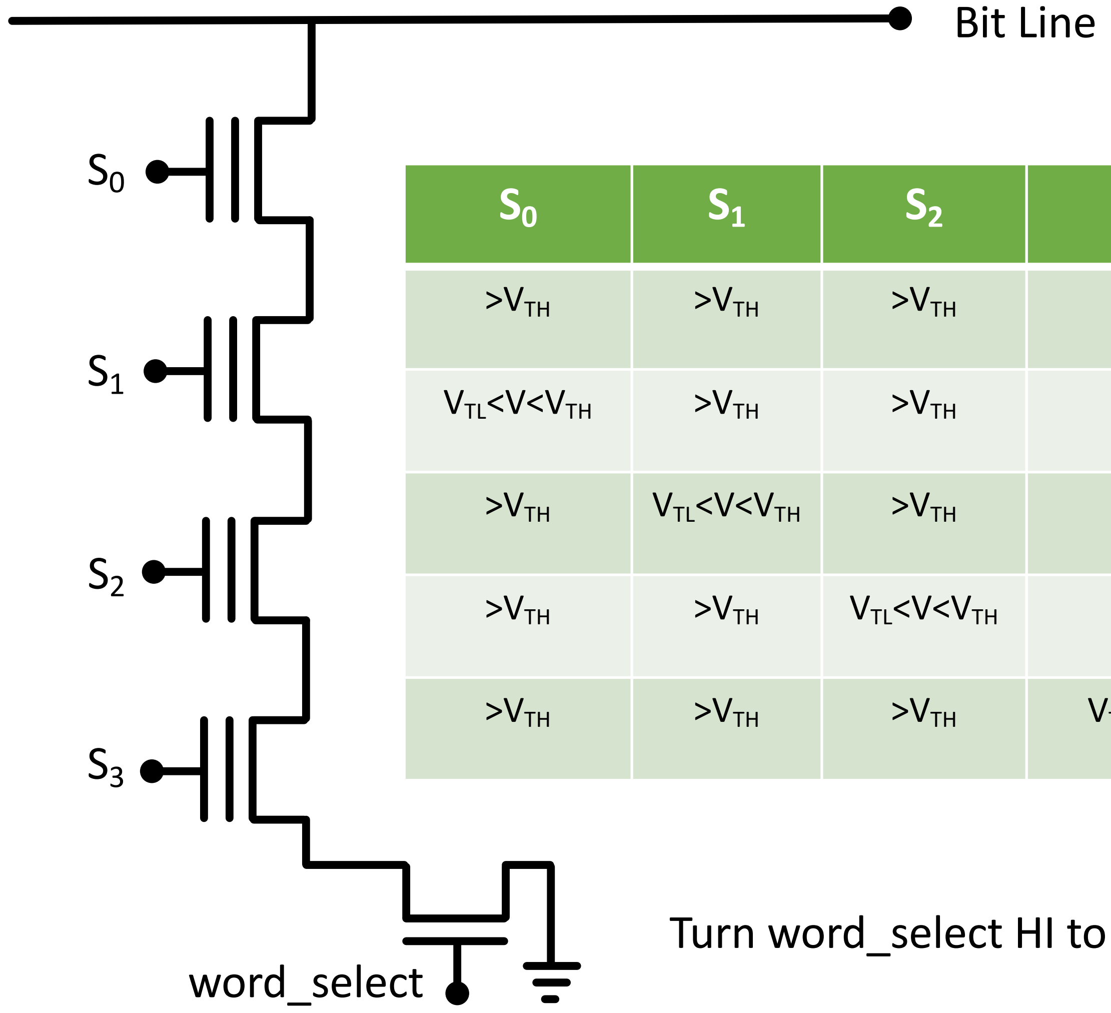
- greater than V_{TL} (low threshold from no trapped carriers)
- less than V_{TH} (high threshold from trapped carriers)



S_0	S_1	S_2	S_3	Out
0	0	0	0	1
1	0	0	0	0 if B_0==1, 1 if B_0==0
0	1	0	0	0 if B_1==1, 1 if B_1==0
0	0	1	0	0 if B_2==1, 1 if B_2==0
0	0	0	1	0 if B_3==1, 1 if B_3==0

Like a **NOR** gate since when all bits are 1, bit line goes low if any input is turned high…hence called **NOR Flash**

NAND Flash



Like a **NAND** gate since when all bits are 1, bit line goes low if tested input is also 1…hence called **NAND Flash**

S_0	S_1	S_2	S_3	Out
$>V_{TH}$	$>V_{TH}$	$>V_{TH}$	$>V_{TH}$	0
$V_{TL}<V<V_{TH}$	$>V_{TH}$	$>V_{TH}$	$>V_{TH}$	0 if B_0==1, 1 if B_0==0
$>V_{TH}$	$V_{TL}<V<V_{TH}$	$>V_{TH}$	$>V_{TH}$	0 if B_1==1, 1 if B_1==0
$>V_{TH}$	$>V_{TH}$	$V_{TL}<V<V_{TH}$	$>V_{TH}$	0 if B_2==1, 1 if B_2==0
$>V_{TH}$	$>V_{TH}$	$>V_{TH}$	$V_{TL}<V<V_{TH}$	0 if B_3==1, 1 if B_3==0

Turn word_select HI to enable whole word

NAND vs. NOR Flash?

- Have Pros cons related to r/w time, size, etc.

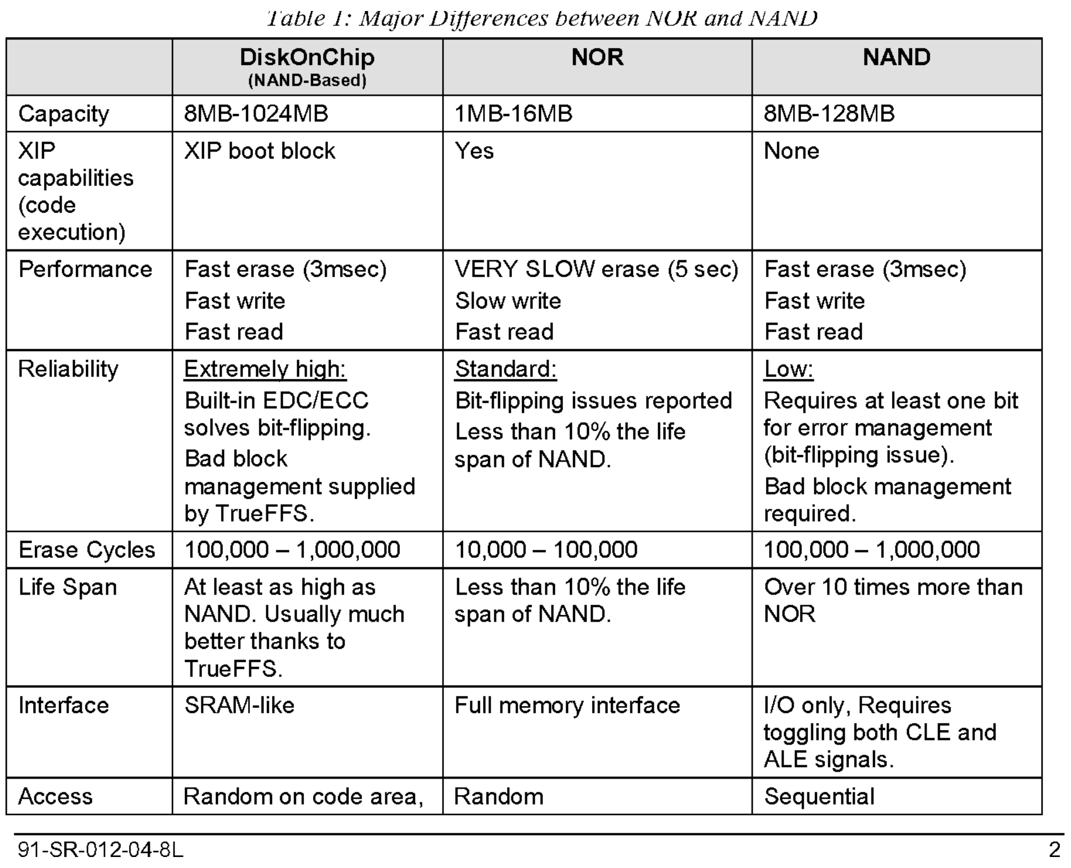


Table 1: Major Differences between NOR and NAND

	DiskOnChip (NAND-Based)	**NOR**	**NAND**
Capacity	8MB-1024MB	1MB-16MB	8MB-128MB
XIP capabilities (code execution)	XIP boot block	Yes	None
Performance	Fast erase (3msec) Fast write Fast read	VERY SLOW erase (5 sec) Slow write Fast read	Fast erase (3msec) Fast write Fast read
Reliability	Extremely high: Built-in EDC/ECC solves bit-flipping. Bad block management supplied by TrueFFS.	Standard: Bit-flipping issues reported Less than 10% the life span of NAND.	Low: Requires at least one bit for error management (bit-flipping issue). Bad block management required.
Erase Cycles	100,000 – 1,000,000	10,000 – 100,000	100,000 – 1,000,000
Life Span	At least as high as NAND. Usually much better thanks to TrueFFS.	Less than 10% the life span of NAND.	Over 10 times more than NOR
Interface	SRAM-like	Full memory interface	I/O only, Requires toggling both CLE and ALE signals.
Access	Random on code area,	Random	Sequential

91-SR-012-04-8L 2

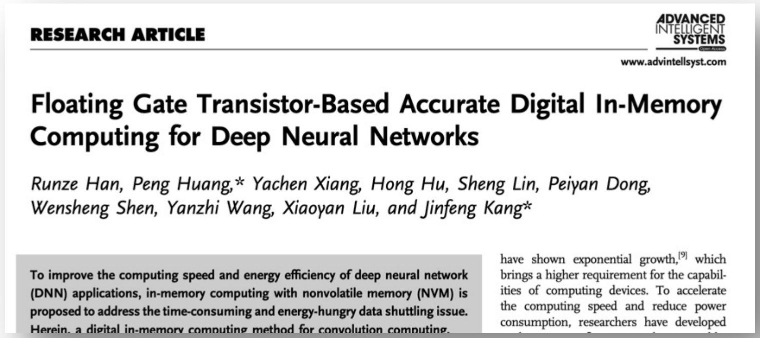
https://www.semanticscholar.org/paper/White-Paper-Two-Flash-Technologies-Compared-%3A-NOR-Tal/52f7d974a7be1911b33cb64c26ba4d7f5b337d9e/figure/0

Using Flash with FPGA Board

- You can use the 16MBits of Quad-SPI Flash to permanently “program” the board (the –f flag)...use wisely.
- About 80% is unused with full binary so you could use this for permanent storage (I never have, but it is doable...can also really mess stuff up)
- Can also interface directly to multi-GB. SD card (which is itself Flash just in a different format)

Floating Gates

- Some neat recent work using floating gates and their adjustable threshold capabilities
- Result is ability to adjust/teach a single transistor when to fire based on input signals!



RESEARCH ARTICLE

ADVANCED INTELLIGENT SYSTEMS

Open Access

www.advintellsyst.com

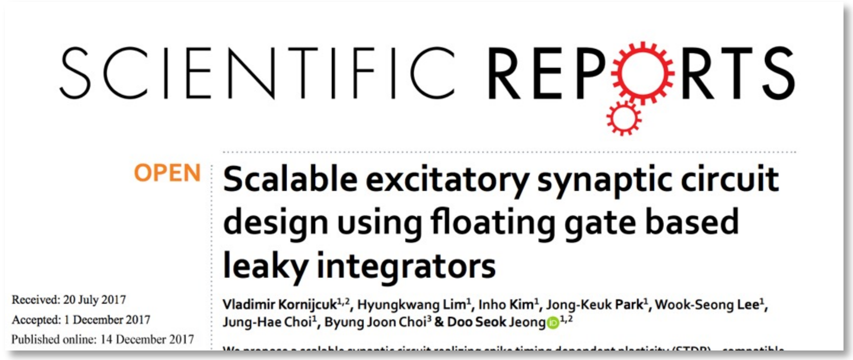
Floating Gate Transistor-Based Accurate Digital In-Memory Computing for Deep Neural Networks

Runze Han, Peng Huang, Yachen Xiang, Hong Hu, Sheng Lin, Peiyan Dong, Wensheng Shen, Yanzhi Wang, Xiaoyan Liu, and Jinfeng Kang**

To improve the computing speed and energy efficiency of deep neural network (DNN) applications, in-memory computing with nonvolatile memory (NVM) is proposed to address the time-consuming and energy-hungry data shuttling issue. Herein, a digital in-memory computing method for convolution computing,

have shown exponential growth,[9] which brings a higher requirement for the capabilities of computing devices. To accelerate the computing speed and reduce power consumption, researchers have developed

2022 improvements



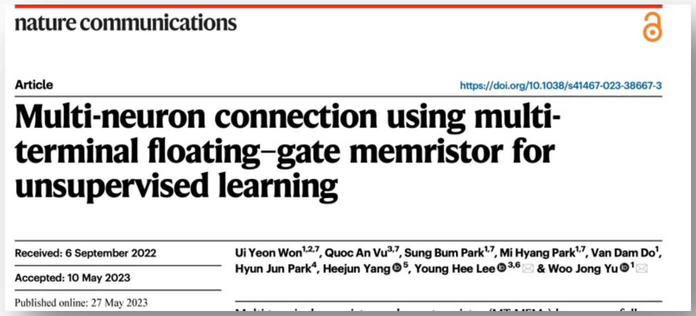
SCIENTIFIC REPORTS

OPEN **Scalable excitatory synaptic circuit design using floating gate based leaky integrators**

Received: 20 July 2017
Accepted: 1 December 2017
Published online: 14 December 2017

Vladimir Kornijcuk[1,2], Hyungkwang Lim[1], Inho Kim[1], Jong-Keuk Park[1], Wook-Seong Lee[1], Jung-Hae Choi[1], Byung Joon Choi[3] & Doo Seok Jeong[1,2]

Floating Gate neural-like implementations, 2017



nature communications

Article https://doi.org/10.1038/s41467-023-38667-3

Multi-neuron connection using multi-terminal floating–gate memristor for unsupervised learning

Received: 6 September 2022
Accepted: 10 May 2023
Published online: 27 May 2023

Ui Yeon Won[1,2,7], Quoc An Vu[3,7], Sung Bum Park[1,7], Mi Hyang Park[1,7], Van Dam Do[1], Hyun Jun Park[4], Heejun Yang[5], Young Hee Lee[3,6] & Woo Jong Yu[1]

2023 work

Two-ish Major Off-FPGA Options

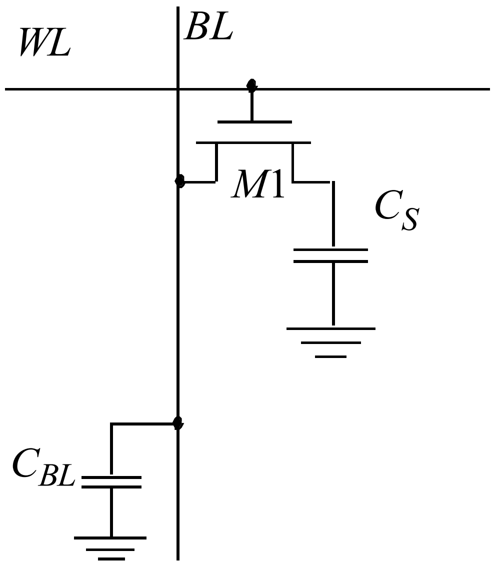
- Flash/EEPROM:
  - Many different form factors, very slow to read/write, but **non-volatile**, meaning it will last beyond power cycles

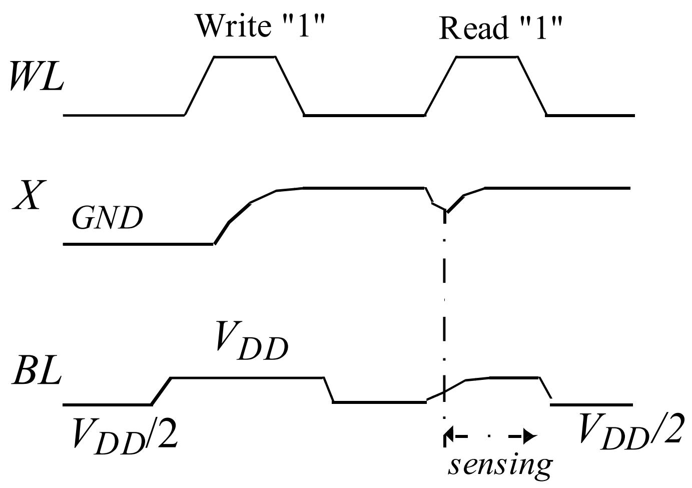
- Dynamic Random Access Memory (DRAM):
  - Potentially very high read-write rates
  - Needs to be constantly refreshed (dynamic)
  - Volatile…~100 ms after power-off, memory lost

DRAM

- Dynamic Random Access Memory!
- Single transistor and capacitor per bit (capacitor does the storage)
- Can be made *extremely* dense and therefore economical
- Are quite fast:
  - SRAM will have access time of down to 10ns or less (consistent)
  - **DRAM** will have access time from 50-250ns (variable)
  - EEPROM/Flash way slower (esp for writes)
- Capacitors decay rather quickly (especially since DRAM capacitors are about 10 femtoFarads) so need to be refreshed every 64 milliseconds.

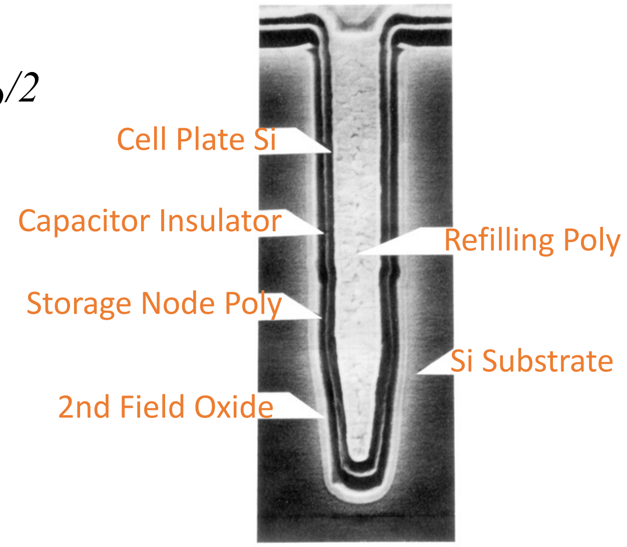
Dynamic RAM (DRAM) Cell





[Rabaey03]

DRAM uses Special Capacitor Structures

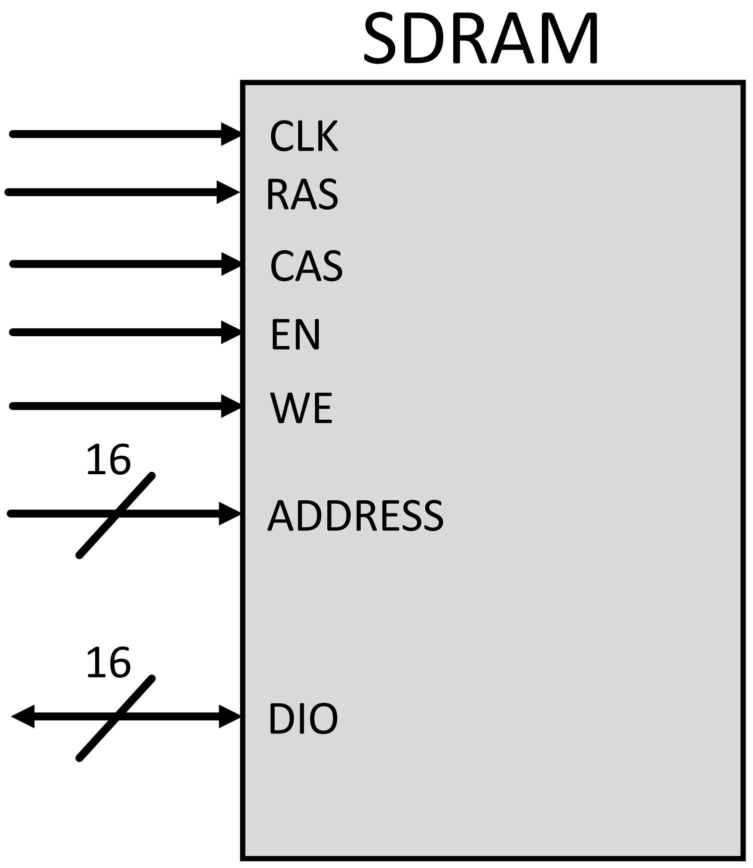


To Write: set Bit Line (BL) to 0 or V_{DD}
& enable Word Line (WL) (i.e., set to V_{DD})

To Read: set Bit Line (BL) to V_{DD} /2
& enable Word Line (i.e., set it to V_{DD})

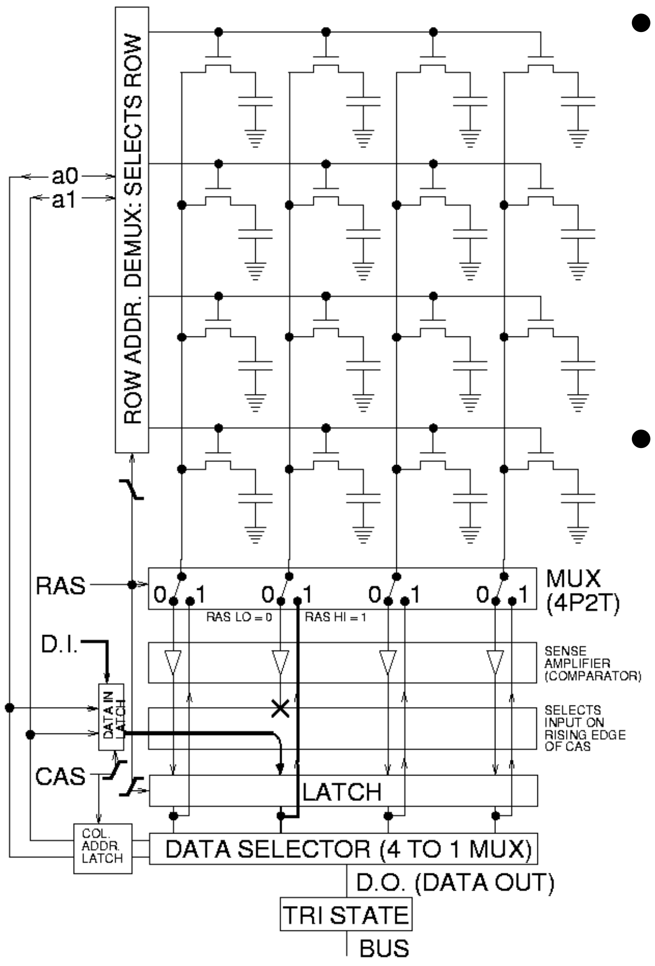
- DRAM relies on charge stored in a capacitor to hold state
- Found in all high density memories (one bit/transistor)
- Must be “refreshed” or state will be lost – high overhead

DRAM is inherently one-port



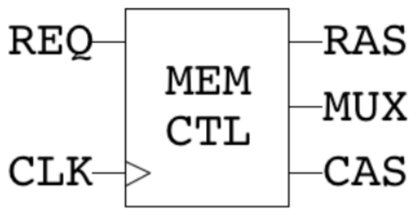
- DRAM is always a one-port entity.
- Have to read and write over same port

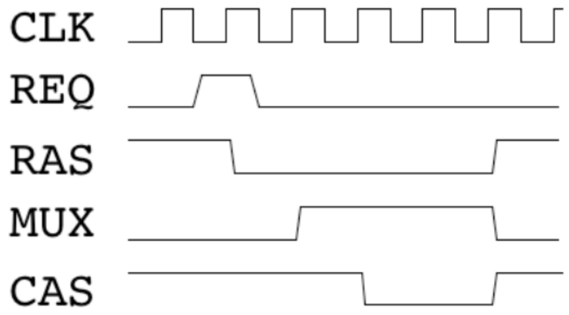
DRAM Memory and Controller



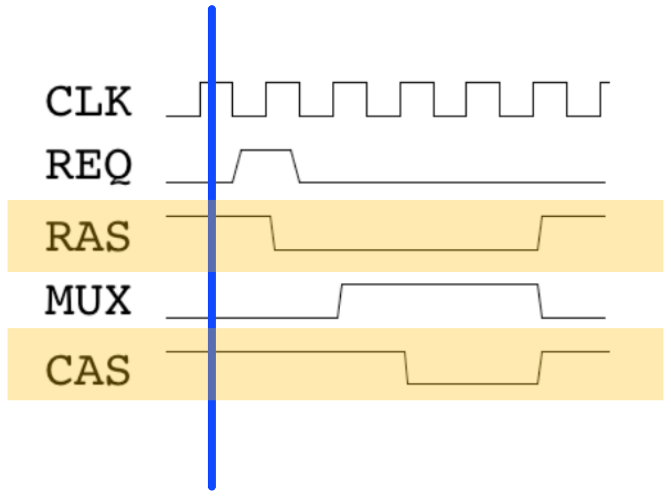
ROW ADDR. DEMUX: SELECTS ROW
a0
a1
RAS
MUX (4P2T)
RAS LO = 0
RAS HI = 1
D.I.
DATA IN LATCH
SENSE AMPLIFIER (COMPARATOR)
SELECTS INPUT ON RISING EDGE OF CAS
CAS
LATCH
COL. ADDR. LATCH
DATA SELECTOR (4 TO 1 MUX)
D.O. (DATA OUT)
TRI STATE
BUS

- Reading is destructive!
  - Data stored on small capacitor
  - To read it we must bleed the capacitor off
  - Therefore need to refresh
- Need to refresh even when not reading (every 100 ms)

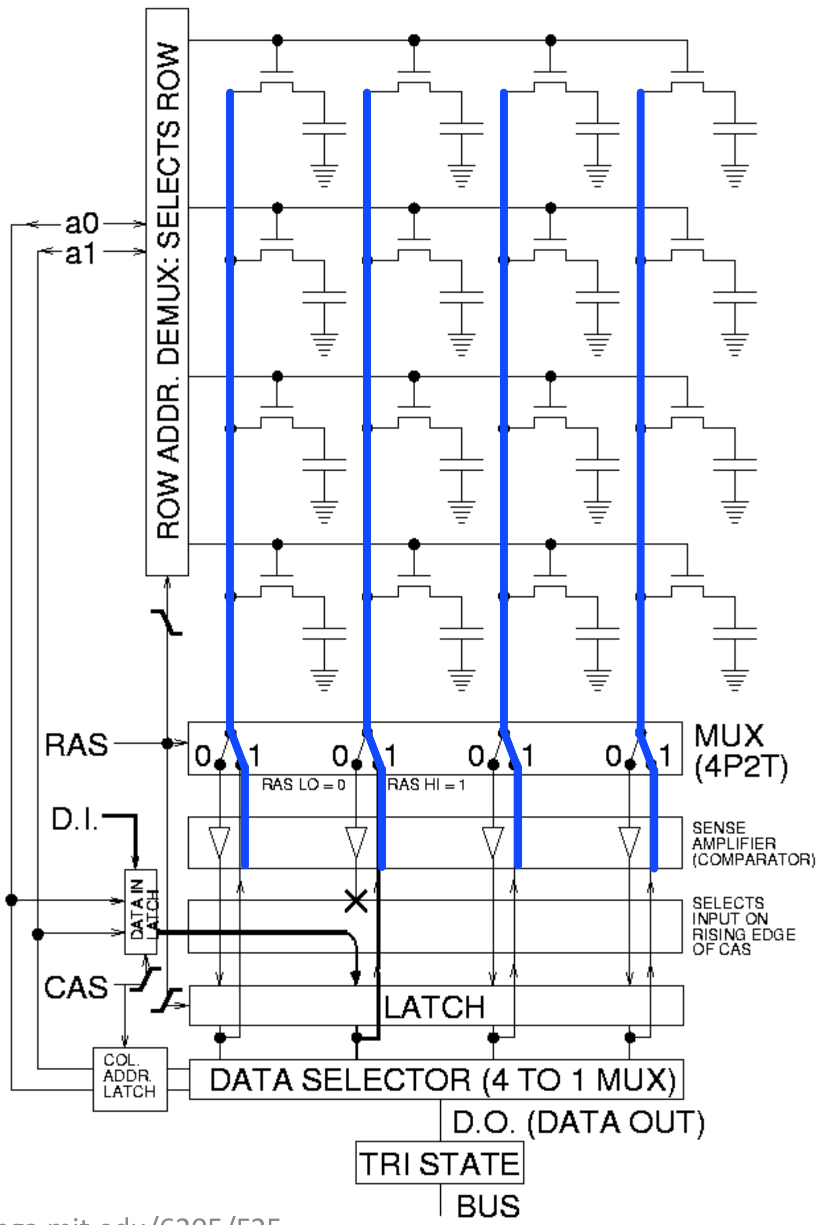




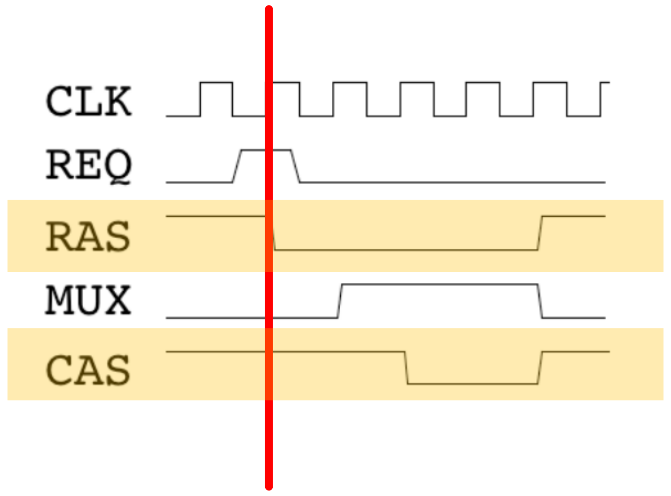
Asynchronous DRAM



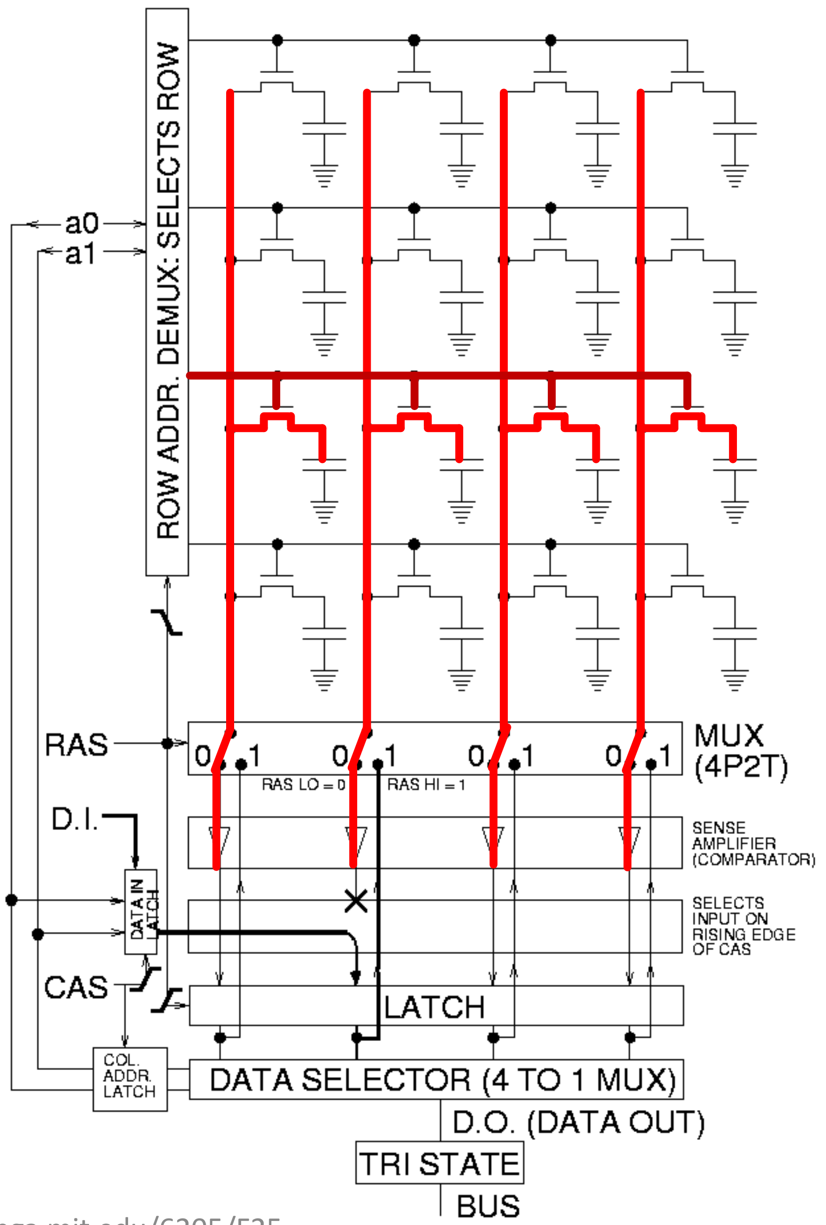
Column lines start charged to mid-voltage (RAS=1)


ROW ADDR. DEMUX: SELECTS ROW
a0
a1
RAS
D.I.
DATA IN LATCH
CAS
COL. ADDR. LATCH
MUX (4P2T)
RAS LO = 0
RAS HI = 1
SENSE AMPLIFIER (COMPARATOR)
SELECTS INPUT ON RISING EDGE OF CAS
LATCH
DATA SELECTOR (4 TO 1 MUX)
D.O. (DATA OUT)
TRI STATE
BUS

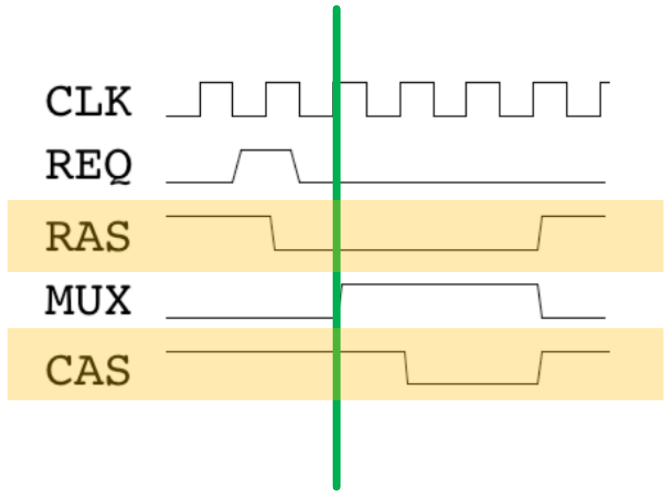
Asynchronous DRAM



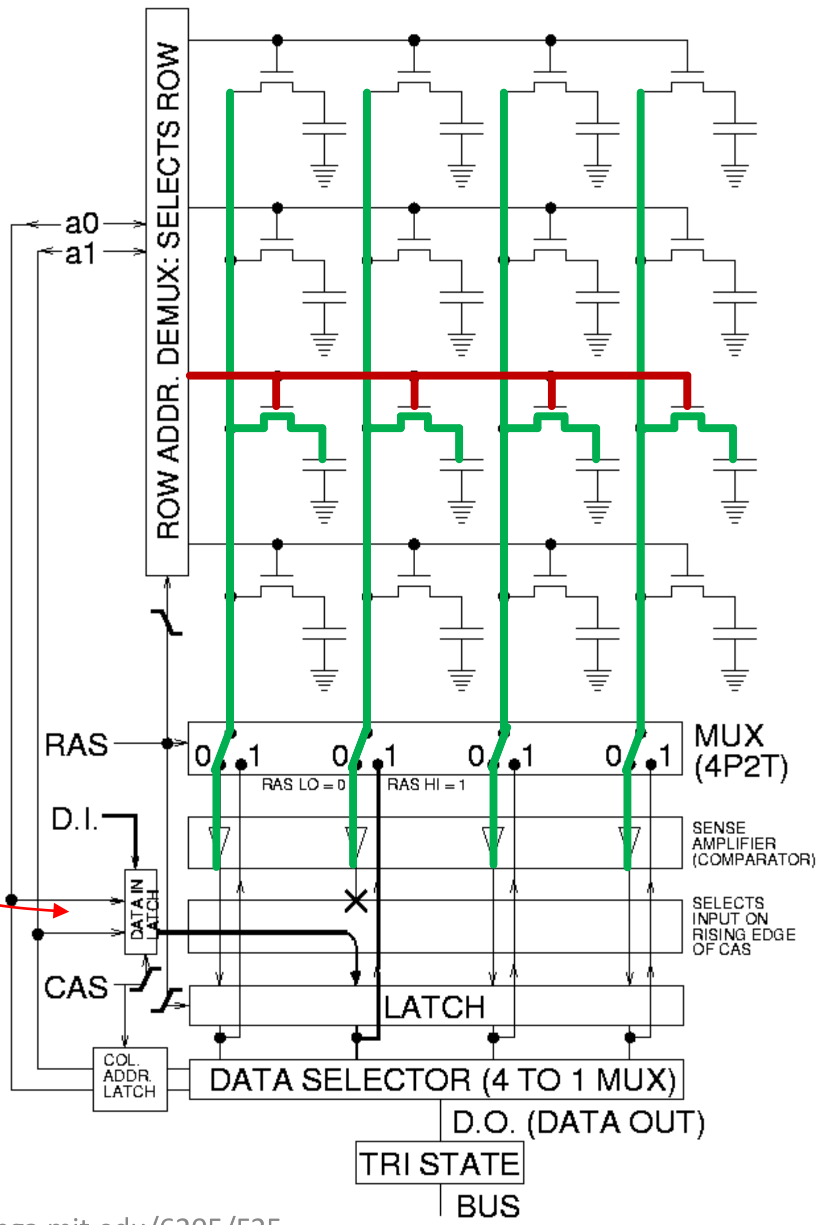
Select Row and feed output of each column into feedback amplifiers to sense/regenerate 1s and 0s


ROW ADDR. DEMUX: SELECTS ROW
a0
a1
RAS
D.I.
CAS
DATA IN LATCH
COL. ADDR. LATCH
MUX (4P2T)
RAS LO = 0
RAS HI = 1
SENSE AMPLIFIER (COMPARATOR)
SELECTS INPUT ON RISING EDGE OF CAS
LATCH
DATA SELECTOR (4 TO 1 MUX)
D.O. (DATA OUT)
TRI STATE
BUS

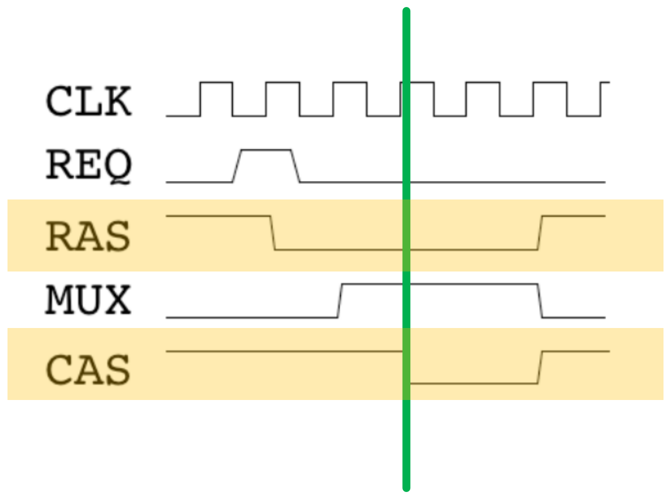
Asynchronous DRAM



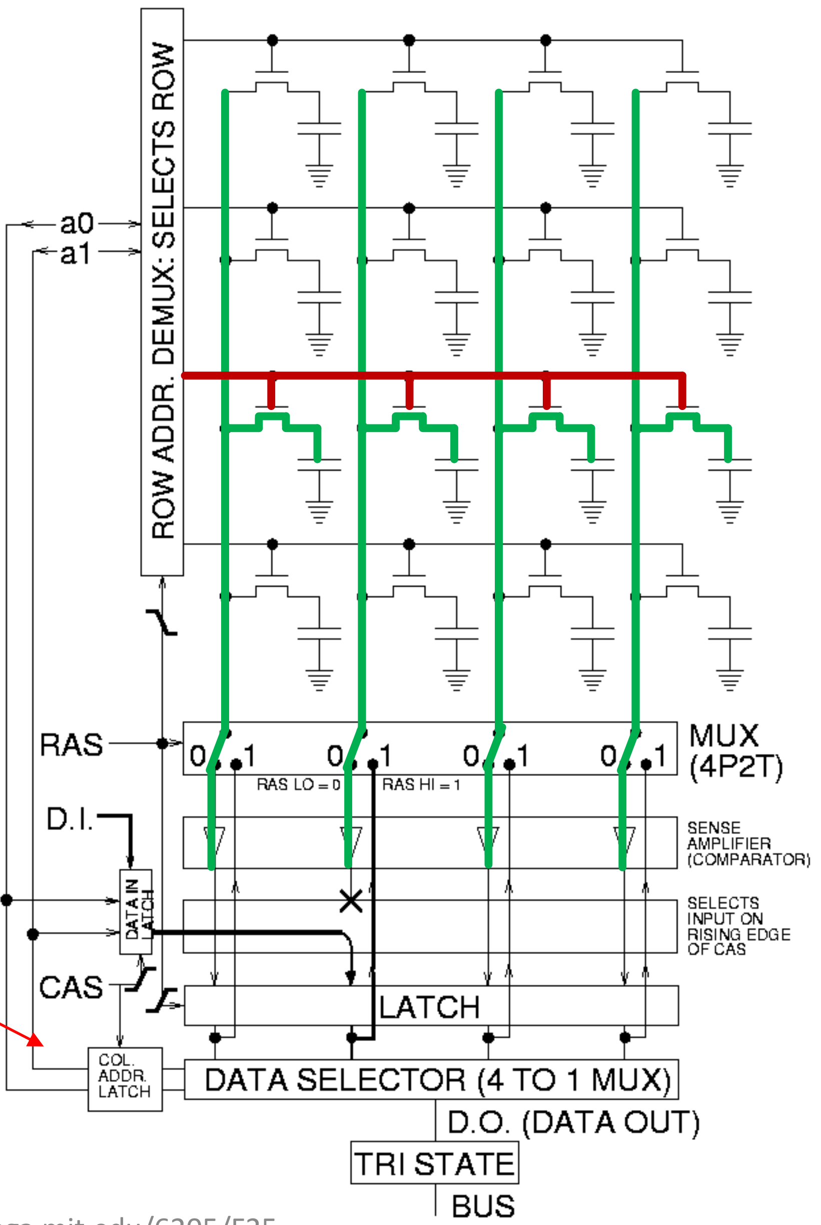
Read/Write by adjusting some input/output selector (associated with MUX signal)


ROW ADDR. DEMUX: SELECTS ROW
a0
a1
RAS
D.I.
DATA IN LATCH
CAS
COL. ADDR. LATCH
MUX (4P2T)
RAS LO = 0
RAS HI = 1
SENSE AMPLIFIER (COMPARATOR)
SELECTS INPUT ON RISING EDGE OF CAS
LATCH
DATA SELECTOR (4 TO 1 MUX)
D.O. (DATA OUT)
TRI STATE
BUS

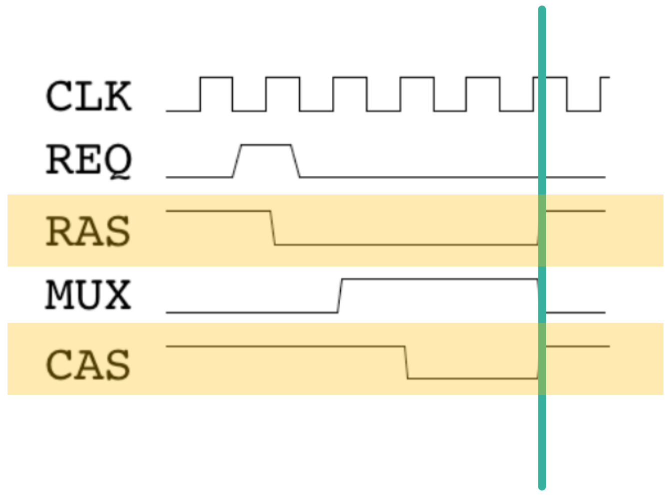
Asynchronous DRAM



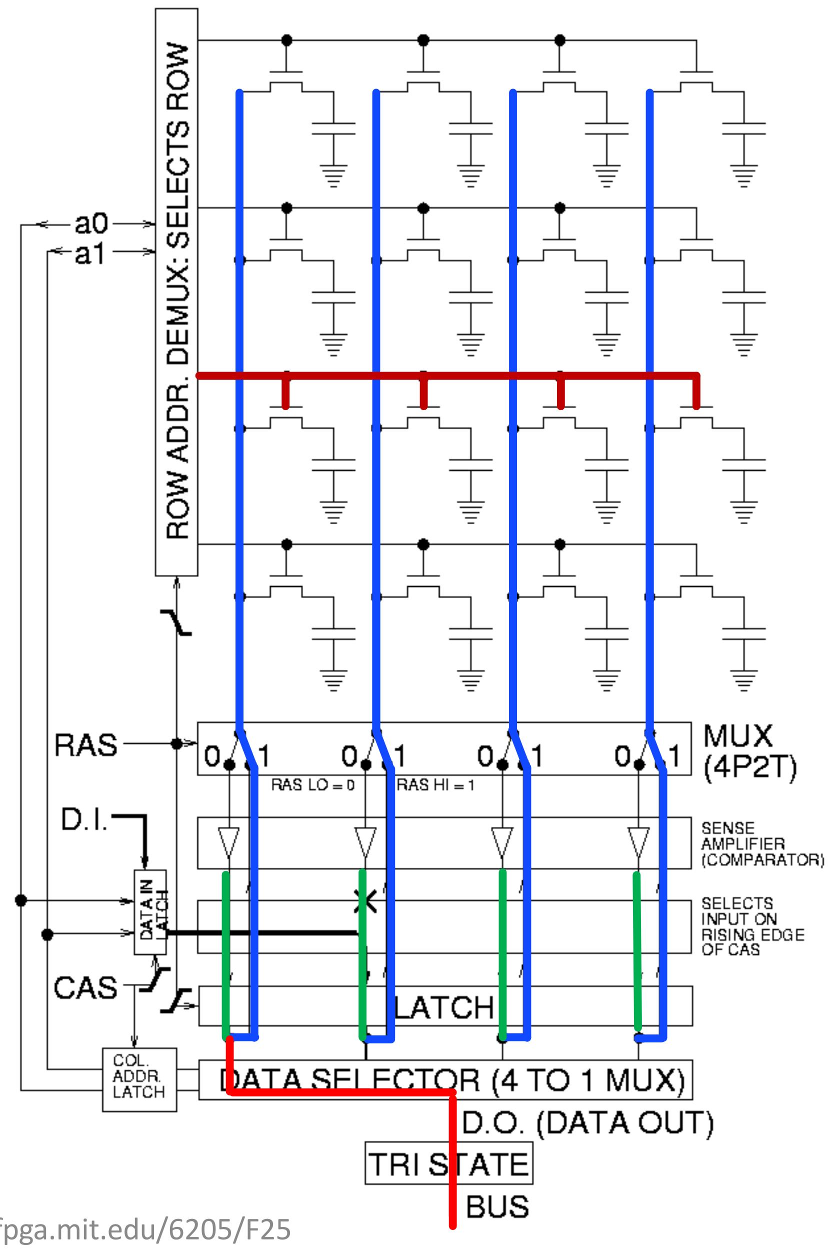
Select which Column to route out


ROW ADDR. DEMUX: SELECTS ROW
a0
a1
RAS
MUX (4P2T)
0
1
RAS LO = 0
RAS HI = 1
SENSE AMPLIFIER (COMPARATOR)
D.I.
DATA IN LATCH
SELECTS INPUT ON RISING EDGE OF CAS
CAS
LATCH
COL. ADDR. LATCH
DATA SELECTOR (4 TO 1 MUX)
D.O. (DATA OUT)
TRI STATE
BUS

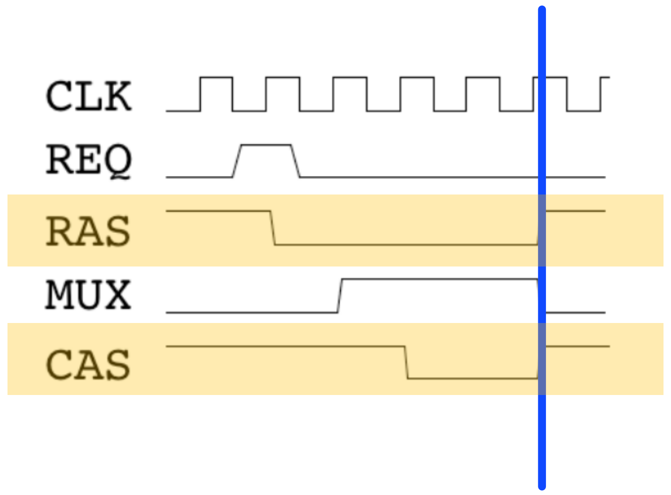
Asynchronous DRAM



Route out desired bits
ALSO
Redirect read out columns back up to recharge appropriate columns

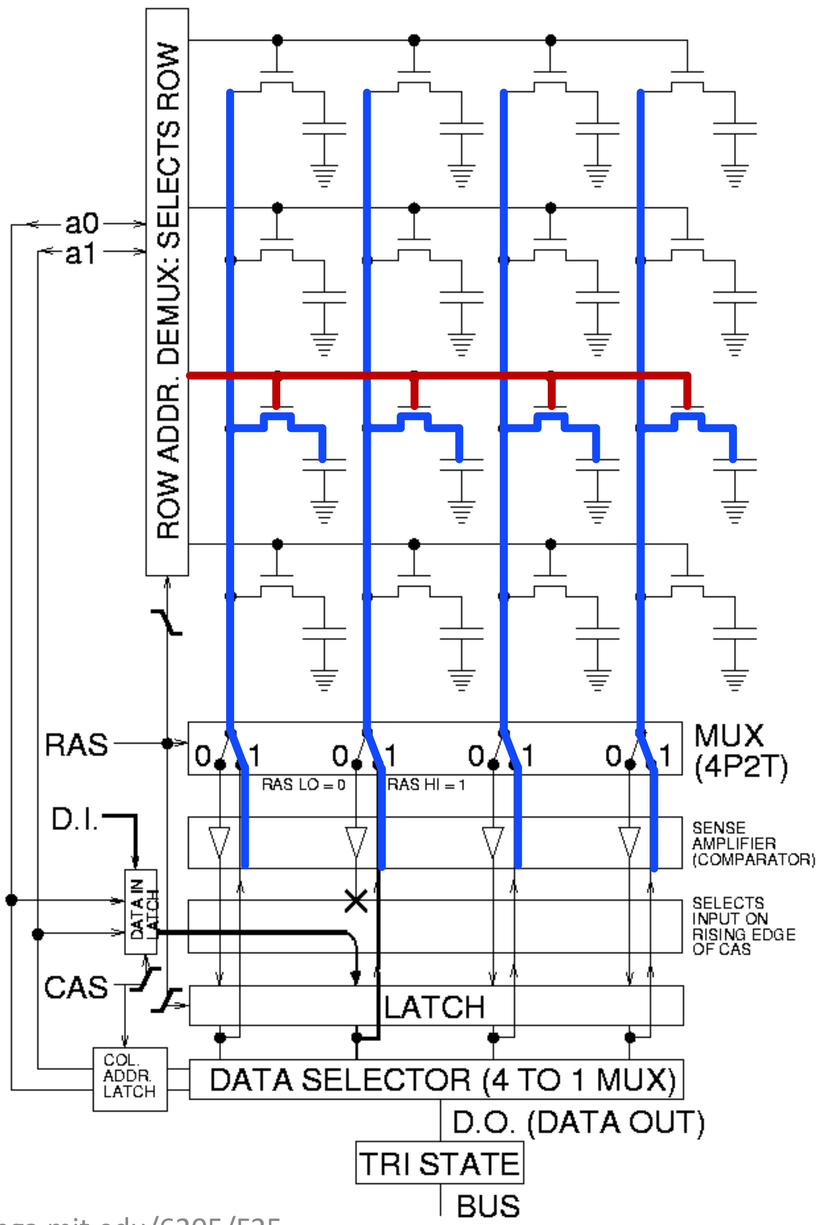

ROW ADDR. DEMUX: SELECTS ROW
a0
a1
RAS
MUX (4P2T)
RAS LO = 0
RAS HI = 1
SENSE AMPLIFIER (COMPARATOR)
D.I.
DATA IN LATCH
SELECTS INPUT ON RISING EDGE OF CAS
CAS
LATCH
COL. ADDR. LATCH
DATA SELECTOR (4 TO 1 MUX)
D.O. (DATA OUT)
TRI STATE
BUS

Asynchronous DRAM

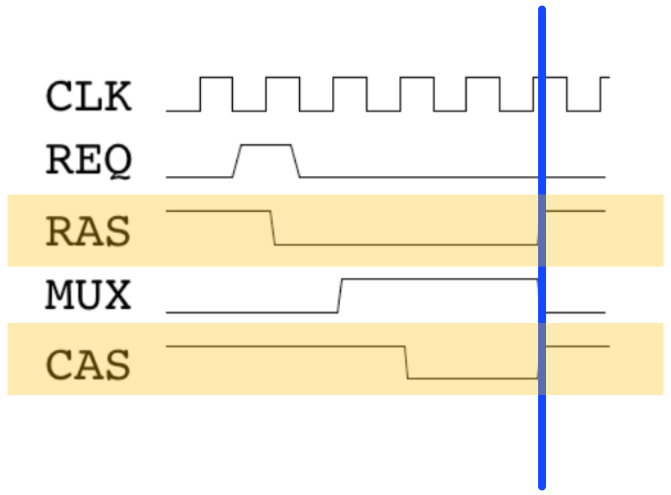


Recharge capacitors fully

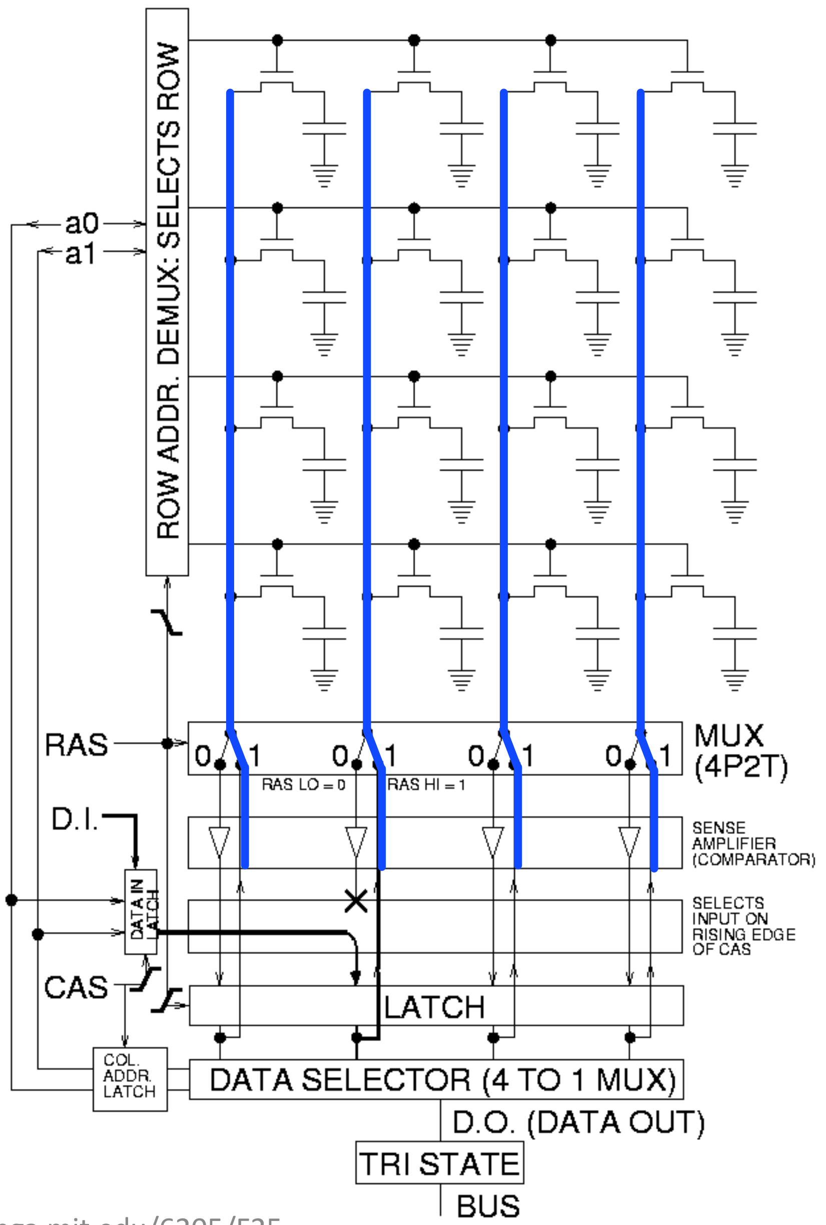
Then you're back to beginning


ROW ADDR. DEMUX: SELECTS ROW
a0
a1
RAS
0
1
RAS LO = 0
RAS HI = 1
MUX (4P2T)
SENSE AMPLIFIER (COMPARATOR)
D.I.
DATA IN LATCH
SELECTS INPUT ON RISING EDGE OF CAS
CAS
LATCH
COL. ADDR. LATCH
DATA SELECTOR (4 TO 1 MUX)
D.O. (DATA OUT)
TRI STATE
BUS

Asynchronous DRAM

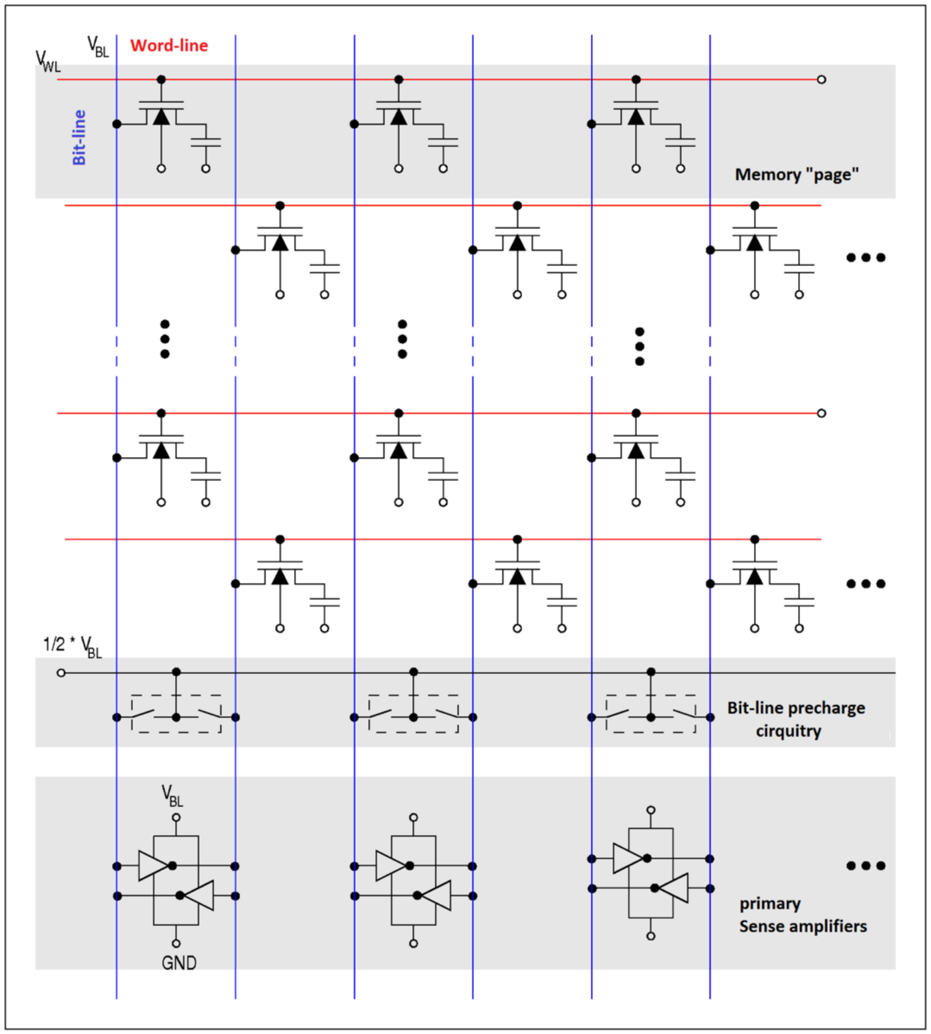


Then you're back to beginning


ROW ADDR. DEMUX: SELECTS ROW
a0
a1
RAS
D.I.
DATA IN LATCH
CAS
COL. ADDR. LATCH
MUX (4P2T)
RAS LO = 0
RAS HI = 1
SENSE AMPLIFIER (COMPARATOR)
SELECTS INPUT ON RISING EDGE OF CAS
LATCH
DATA SELECTOR (4 TO 1 MUX)
D.O. (DATA OUT)
TRI STATE
BUS

DRAM Cells are Staggered Physically

- The sense amplifiers use two parallel bit lines (one active and one for reference) to detect the slight perturbation when you discharge the capacitor

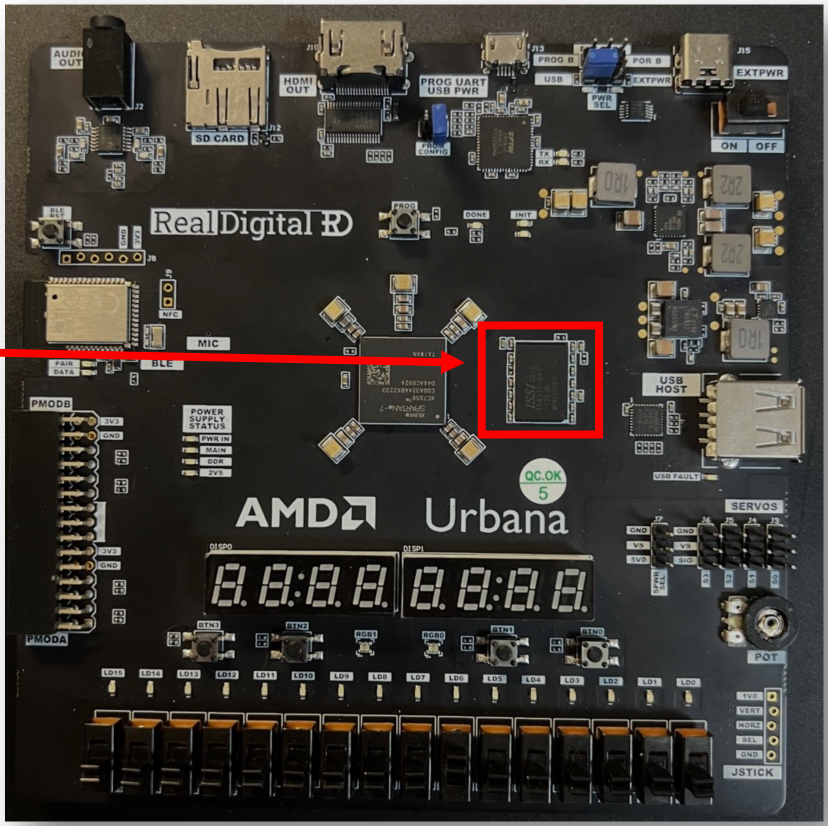


Many Flavors of DRAM

- DRAM (Asynchronous)
- SDRAM (Synchronous DRAM)
  - (one clock cycle per operation)
- Single Data Rate SDRAM (SDR SDRAM):
  - One R/W per clock Cycle
- Double-Data Rate SDRAM (DDR SDRAM)
  - Two R/W per clock cycle (called double pumping)
- Faster Double-Data Rate SDRAM (DDR2 SDRAM)
- And DDR3 and DDR4 (lower voltages, higher clocks)
- LDDR3 and LDDR4 (low power variants)
- DDR5…just faster in general 🤣…just keeps going…

DRAM

- DRAM is extremely dense.
- That tiny chip holds 1 billion yes/no decisions
- In a vast 2D array
- That is constantly bleeding out due to thermodynamics
- Requiring the entire thing to be rewritten every 64 ms
- Row then column must be specified



DRAM

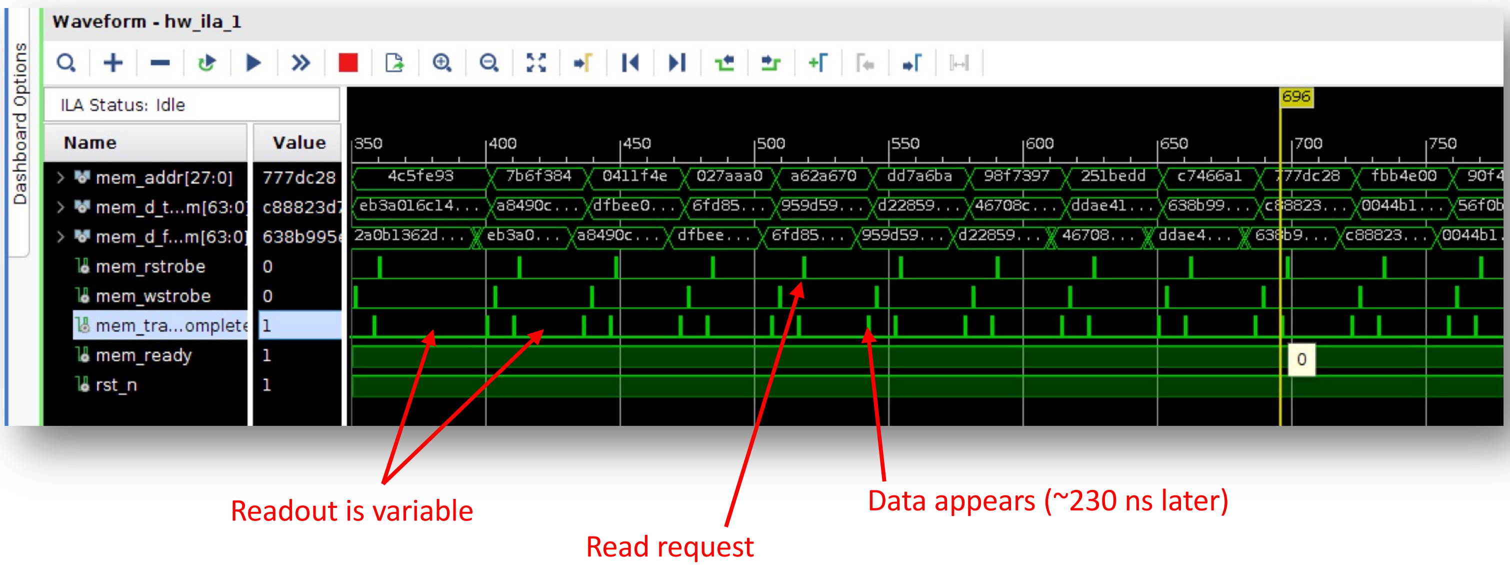
- The constant need for refreshing means getting info into and out of the DRAM is not an easy task…
- Even more complicated in modern devices because they'll have different banks/channels/buffers
- Requires something to handle all the needs for refreshes and balancing them with requests for reads/writes, etc…
- This is the job of a **Memory Interface/Controller**

Using DDR3 on Urbana Board

- Urbana board has ~128MB of DDR3
- We’ll use a Memory Controller to take care of the hard part of controlling the DDR3, though working with the memory controller is not super simple either.
- We’ll start using it in Week 6

Using DDR3 on Urbana Board

- DRAM is *FAST*, BUT:
  - A lot of delay from a read request to the data out
  - Response time can be *variable* since DRAM will be taken offline periodically to internally refresh its values…
  - if you're trying to access something that is in a bank getting refreshed, your readout (or write) will be delayed

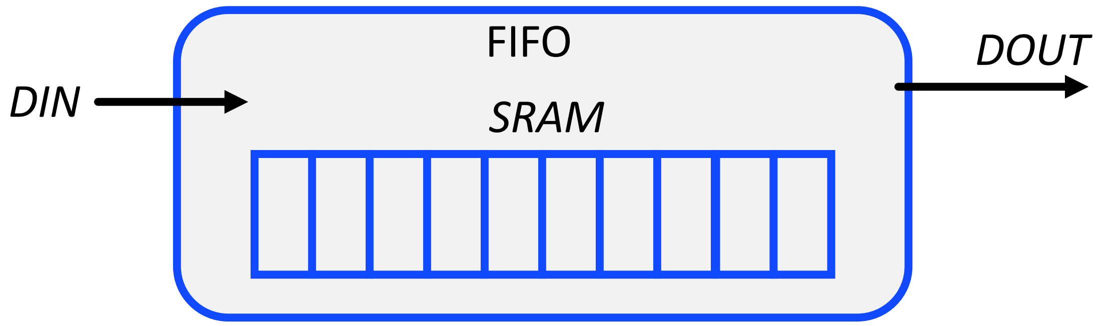


Often need to surround SDRAM with other memory

- There will be input and output buffers
- And you’ll have to wait for transactions to go through to get what you want.
- There’s no way around that.

FIFO (First-In-First-Out)

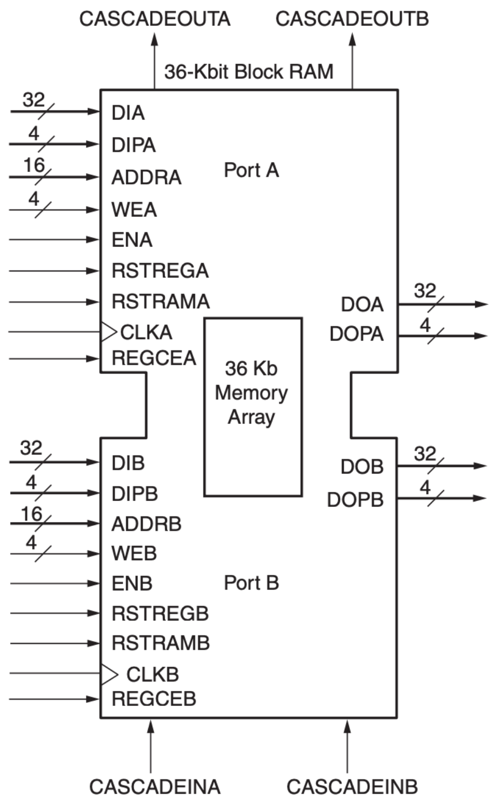
- Basically a Queue like you see in Python or something, but we can’t dynamically allocate storage space ahead of time at our low level!



- Data is not randomly accessed, but instead is accessed in the order it was provided
- Can generate either using Dist RAM or BRAM

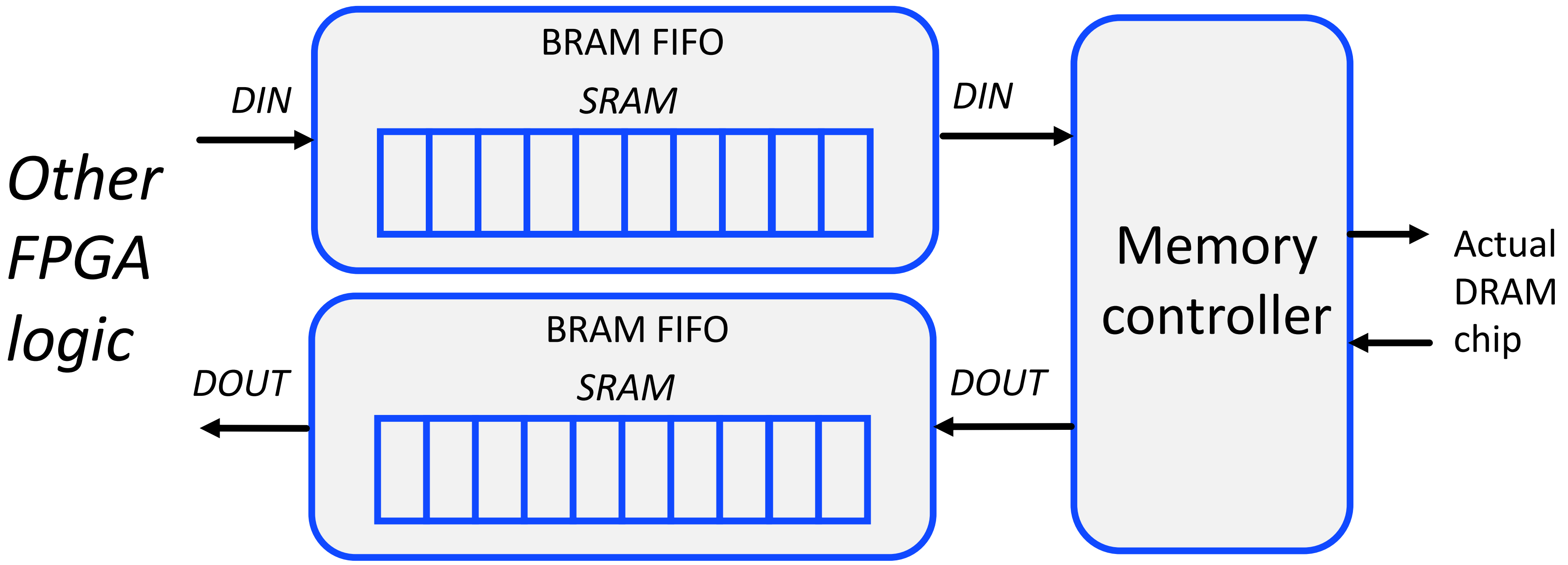
FIFO Implemented with BRAM:

- Remember structure of BRAM:
- Dual Port allows us to simultaneously read and write to different SRAM cells
- Add some logic around it to store and autoincrement the memory addresses and you've got a FIFO



Where to Use FIFOs?

- Anytime you have two modules sharing data (one providing data to another) and they may be producing/consuming in differing patterns (We'll



Using DDR3 on Urbana Board

- SDRAM is good for:
  - LOTs of data needed in random access at medium data rates
    - (~100 MBytes/s average R/W)
  - LOTS and LOTs of data needed in short high speed bursts
    - ~1.2 GBytes/s in small bursts

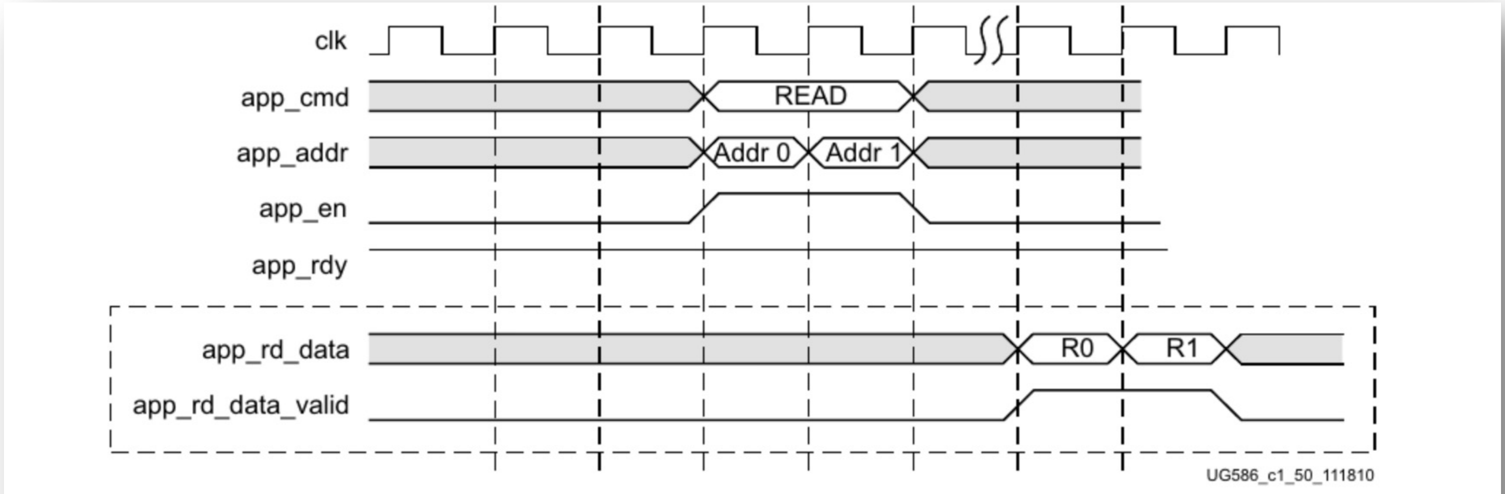


Figure 1-82: **2:1 Mode UI Interface Read Timing Diagram (Memory Burst Type = BL4 or BL8)**

And remember the disconnect!

- You cannot write your Verilog HDL with the same disconnect as you would with Python or C when it comes to memory.
- You need to think about and account for memory delays and give your system time to do what it needs to with memory.
- You must be careful what you ask for when it comes to memory! It may not be possible.