ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2023

Memory Hierarchy

Xin Li & Dawei Liu
Duke Kunshan University

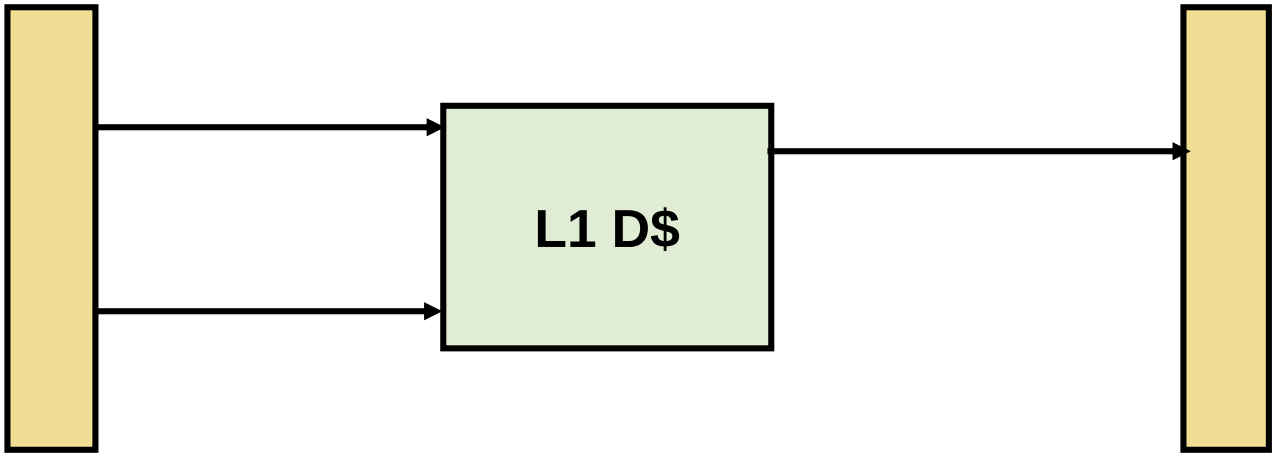
Slides are derived from work by
Andrew Hilton, Tyler Bletsch and Rabih Younes (Duke)

Write Issues

- So far we have looked at reading from cache
  - Insn fetches, loads
- What about writing into cache
  - Stores, not an issue for insn caches – we do not write insn cashes

- Several new issues
  - Must read tags first **before** writing data
    - Cannot be in parallel
  - Cache may have **dirty** data
    - Data which has been updated in this cache, but not lower levels
    - Must be written back to lower level before eviction

Recall Data Memory Stage of Datapath



- So far, have just assume D$ in Memory Stage...
  - Actually a bit more complex

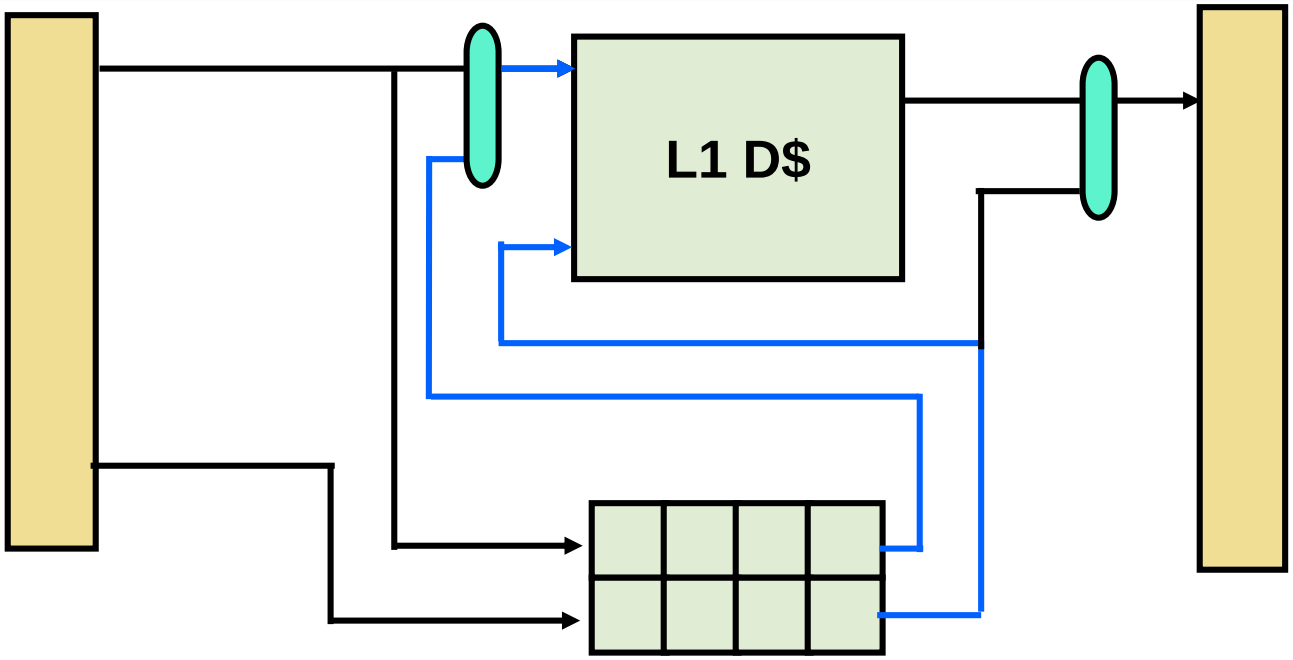
Problem with Writing #1: Store Misses

- Load instruction misses D$:
  - Have to stall datapath
  - Need missing data to complete instruction
- Store instruction misses D$:
  - Stall?
  - Would really like not to
  - Store is writing the data
  - Need rest of block because we cannot have part of a block
    - Generally do not support "these bytes are valid, those are not"
  - How to avoid stall?

Problem with Writing #2: Serial Tag/Data Access

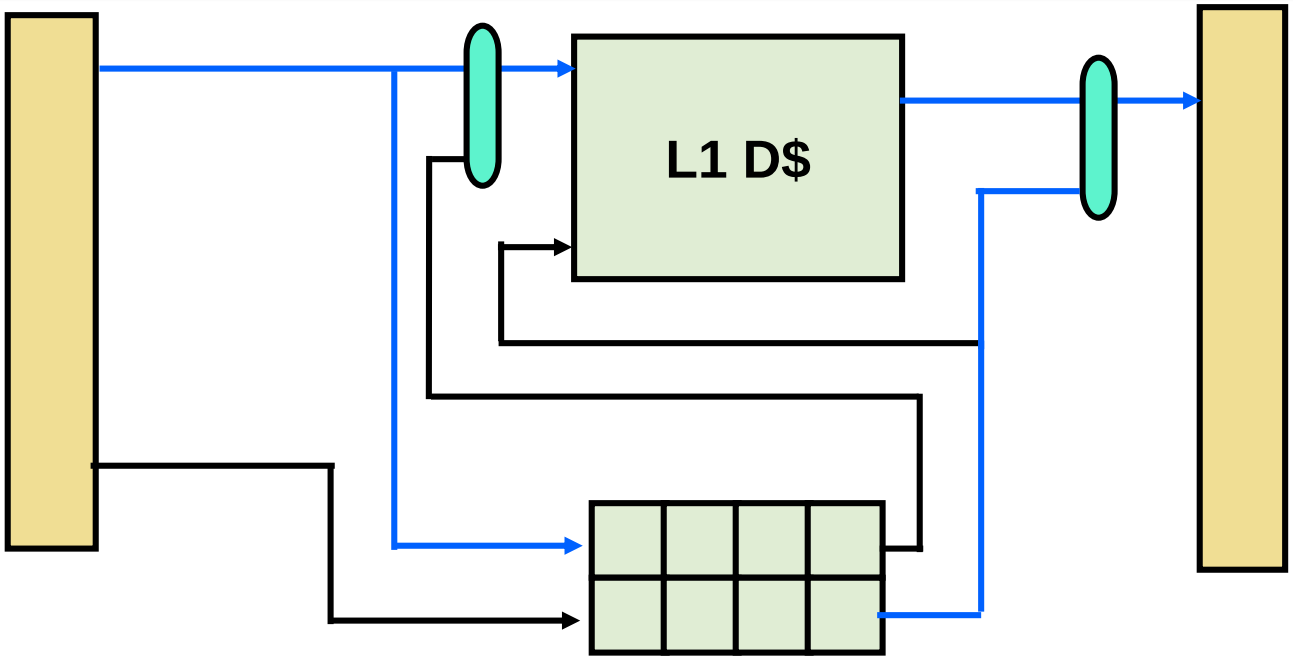
- Load can read tags/data in parallel
  - Read both SRAMs
  - Compare Tags -> Select proper way (if any)
- Stores cannot write tags/data in parallel
  - Read tags/write data array at same time??
    - How to know which way?
    - Or even if its a hit?
    - Incorrect -> overwrote data from somewhere else..
- Multi-cycle data-path:
  - Stores take an extra cycle? Increase CPI
- Pipelined data-path:
  - Tags in one stage, Data in the next?
  - Works for stores, but loads serialize tags/data -> higher CPI

Store Buffer



- Stores write into a **store buffer**
  - Holds address, size, data, of stores
  - Store data written from store buffer into cache

Store Buffer



- Loads search store buffer for matching store
  - Match? **Forward** data from the store
  - No match: Use data from D$
- Addresses are CAM: allow search for match

Store Buffer

- How does this resolve our issues?

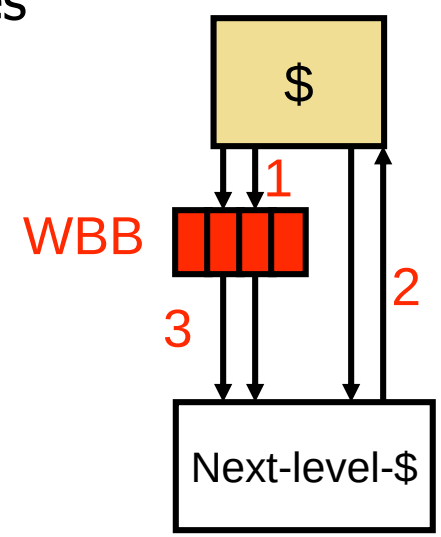
- Problem with Writing #1: Store misses
  - Stores write to store buffer and are done
  - FSM writes stores into D$ from store buffer
  - Misses stall store buffer -> D$ write (but not pipeline)
  - Pipeline will stall on full store buffer
- Problem with Writing #2: Tags -> Data
  - FSM that writes stores to D$ can check tags… then write data
  - Decoupled from data path's normal execution
    - Can happen whenever loads are not using the D$

Write Propagation

- When to propagate new value to (lower level) memory?
- **Write-thru**: immediately
  - Requires additional bus bandwidth
  - Not common
- **Write-back**: when block is replaced
  - Blocks may be **dirty** now
  - Dirty bit (in tag array)
    - Cleared on fill
    - Set by a store to the block

Write Back: Dirty Misses

- Writeback caches may have **dirty misses**:
  - Victim block (one to be replaced) is dirty
    - Must first writeback to next level
    - Then request data for miss
    - Slower :(
  - Solution:
    - Add a buffer on back side of cache: **writeback buffer**
      - Small full associative buffer, holds a few lines
    - Request miss data immediately
    - Put dirty line in WBB
    - Writeback later



What this means to the programmer

- If you're writing code, you want good performance.
- The cache is **crucial** to getting good performance.
- The effect of the cache is influenced by the **order of memory accesses**.

CONCLUSION:

The programmer can change the order of memory accesses to improve performance!

Cache performance matters!

- A **HUGE** component of software performance is how it interacts with cache
- Example:

  Assume that x[i][j] is stored next to x[i][j+1] in memory ("row major order").

  Which will have fewer cache misses?

```
for (k = 0; k < 100; k++)
   for (j = 0; j < 100; j++)
      for (i = 0; i < 5000; i++)
               x[i][j] = 2 * x[i][j];
```

A

```
for (k = 0; k < 100; k++)
   for (i = 0; i < 5000; i++)
      for (j = 0; j < 100; j++)
               x[i][j] = 2 * x[i][j];
```

B 

Blocking (Tiling) Example

```
/* Before */
 for(i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      for (k = 0; k < SIZE; k++)
        c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- Two Inner Loops:
  - Read all NxN elements of b[ ] (N = SIZE)
  - Read N elements of 1 row of a[ ] repeatedly
  - Write N elements of 1 row of c[ ]
- Capacity Misses a function of N & Cache Size:
  - 3 NxN => no capacity misses; otherwise ...
- Idea: compute on BxB submatrix that fits

Blocking (Tiling) Example

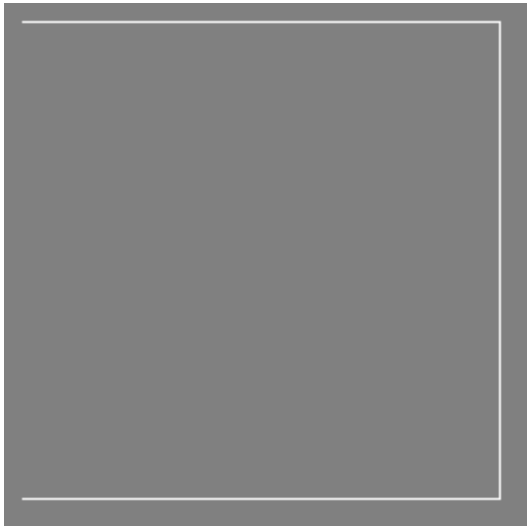
```
/* After */
 for(ii = 0; ii < SIZE; ii += B)
    for (jj = 0; jj < SIZE; jj += B)
      for (kk = 0; kk < SIZE; kk +=B)
        for(i = ii; i < MIN(ii+B-1,SIZE); i++)
          for (j = jj; j < MIN(jj+B-1,SIZE); j++)
            for (k = kk; k < MIN(kk+B-1,SIZE); k++)
               c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

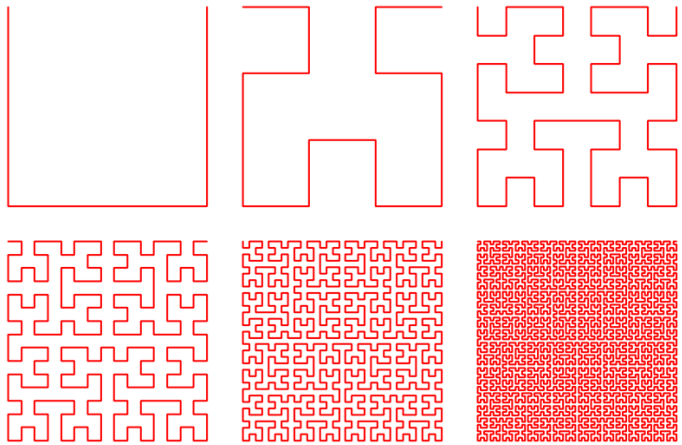
- Capacity Misses decrease

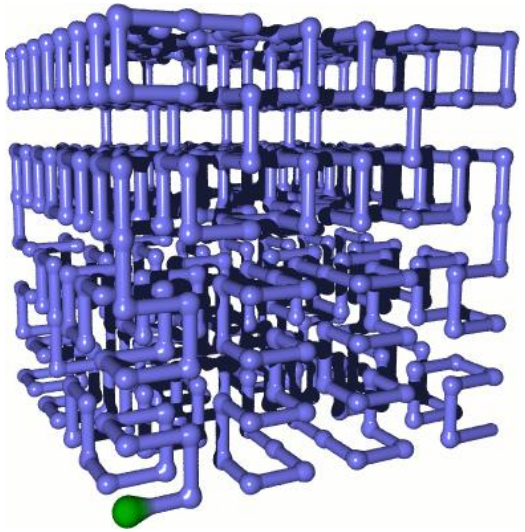
  $2N^3 + N^2$ to $2N^3/B + N^2$
- B called *Blocking Factor (Also called Tile Size)*

Hilbert curves: A fancy trick for matrix locality

- Turn a 1D value into an n-dimensional "walk" of a cube space (like a 2D or 3D matrix) in a manner that maximizes locality
- Extra overhead to compute curve path, but computation takes no memory, and cache misses are very expensive, so it may be worth it
- (Actual algorithm for these curves is simple and easy to find)







Brief History of DRAM

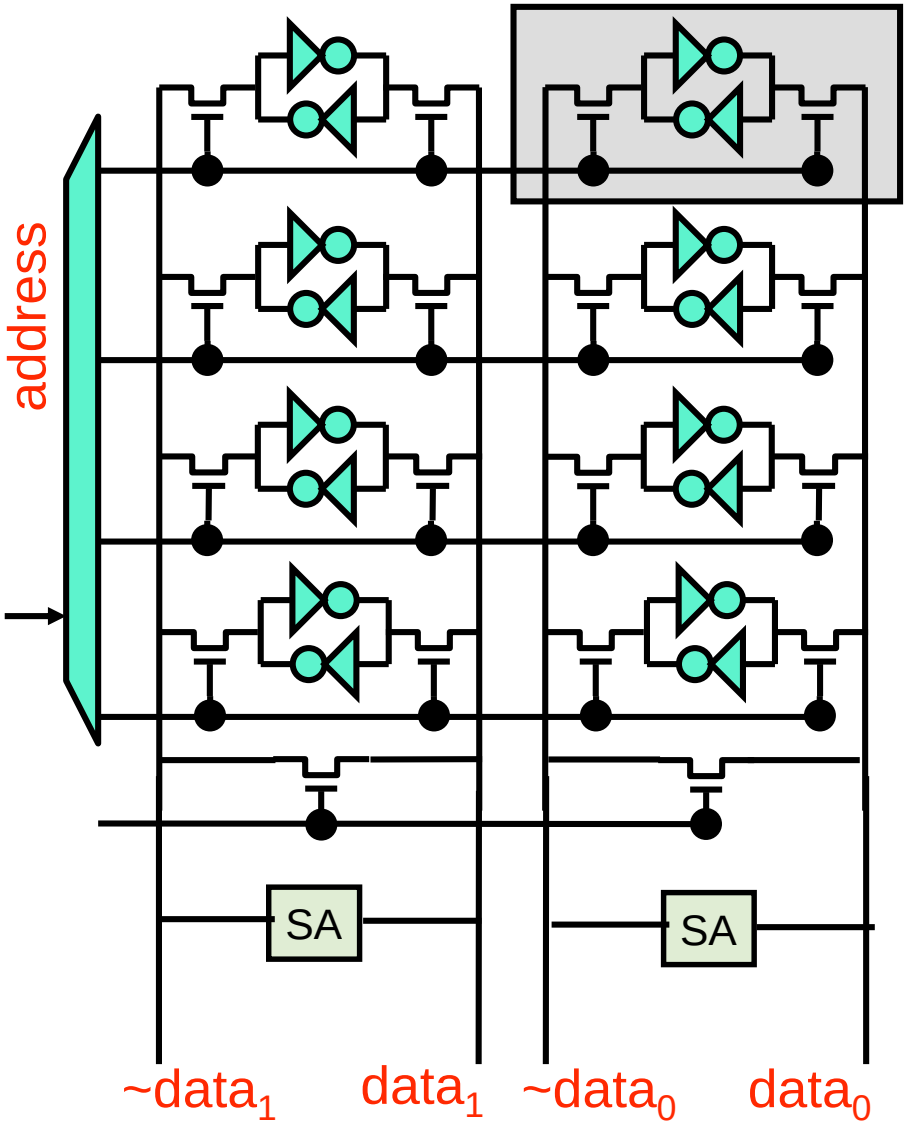
- DRAM (memory): a major force behind computer industry
  - Modern DRAM came with introduction of IC (1970)
  - Preceded by magnetic "core" memory (1950s)
    - More closely resembles today's disks than memory
  - And by mercury delay lines before that (ENIAC)
    - Re-circulating vibrations in mercury tubes

"the one single development that put computers on their feet was the invention of a reliable form of memory, namely the core memory… It's cost was reasonable, it was reliable, and because it was reliable it could in due course be made large"

Maurice Wilkes

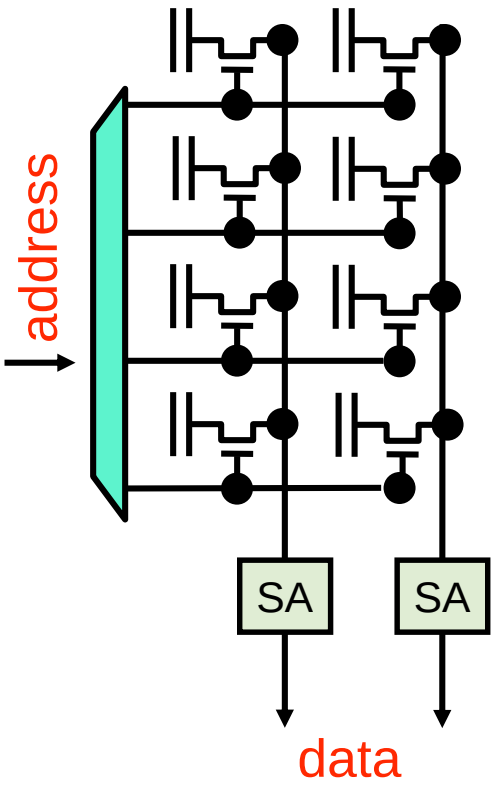
Memoirs of a Computer Programmer, 1985

SRAM



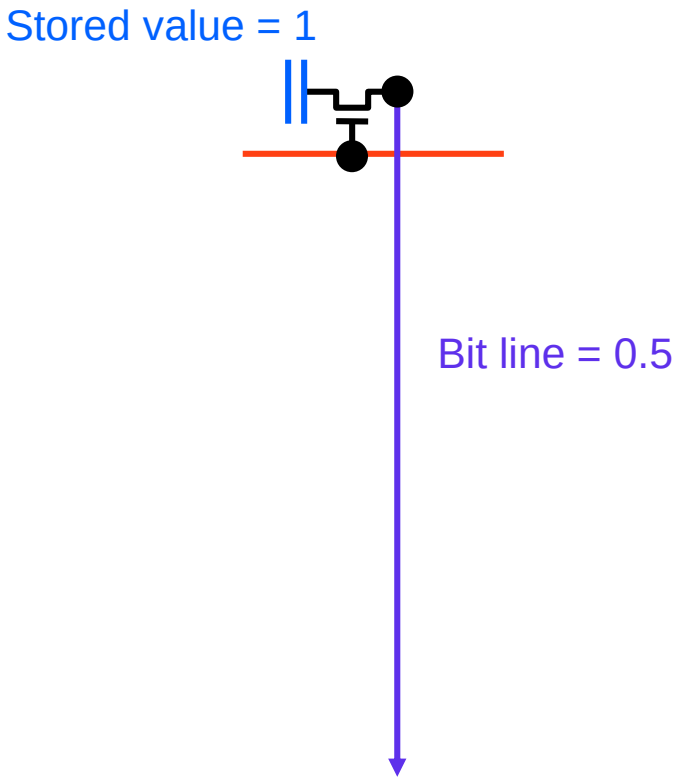
- SRAM: "6T" cells
  - 6 transistors per bit
    - 4 for the CCI
    - 2 access transistors
- **Static**
  - CCIs hold state
- To read
  - Equalize, swing, amplify
- To write
  - Overwhelm

DRAM



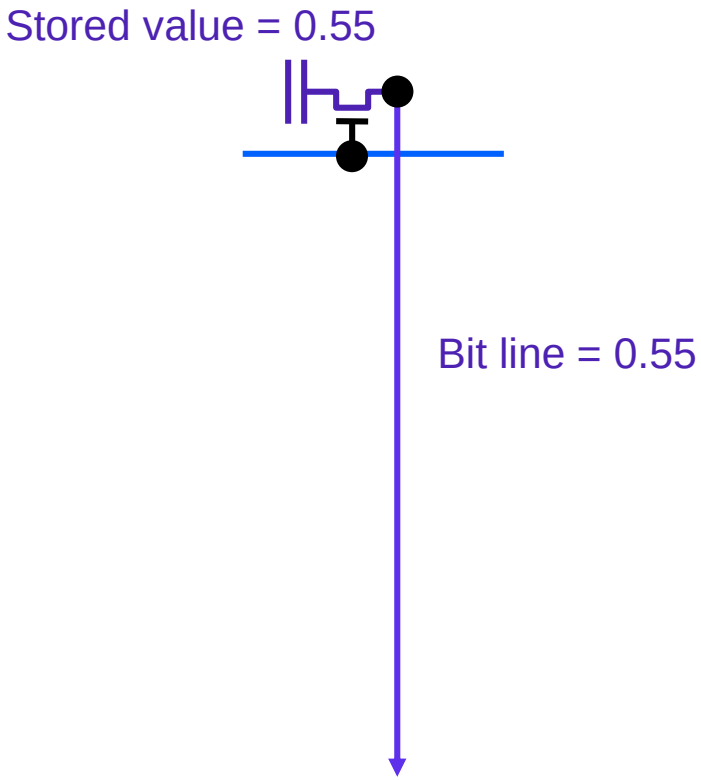
- **DRAM**: dynamic RAM
  - Bits as capacitors
  - Transistors as ports
  - "1T" cells: one access transistor per bit
- **"Dynamic"** means
  - Stored charge decays over time
  - Must be explicitly refreshed
- Designed for density
  - \+ ~6–8X denser than SRAM
  - – But slower too

DRAM Read (simplified version)



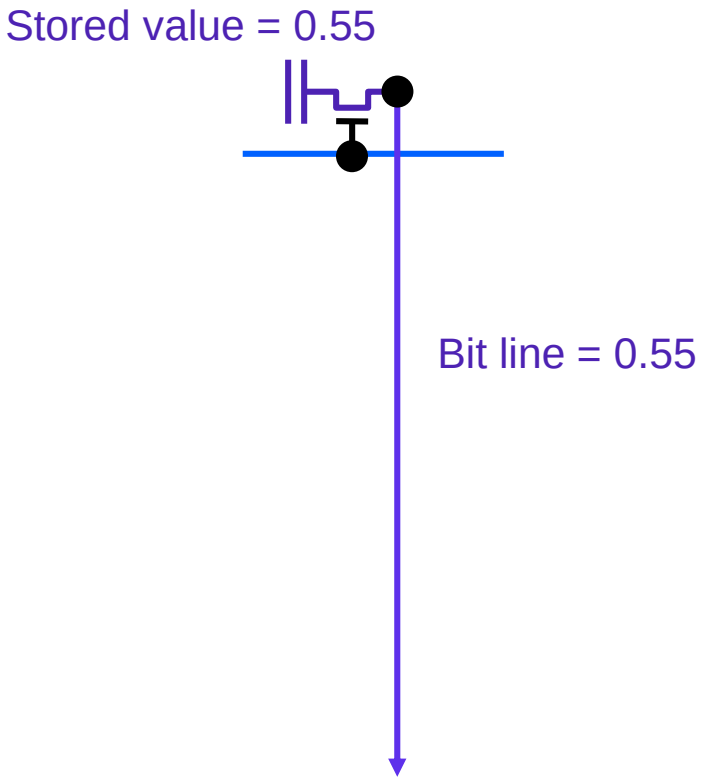
- Bit line pre-charged to 0.5
- Storage at 1

DRAM Read (simplified version)



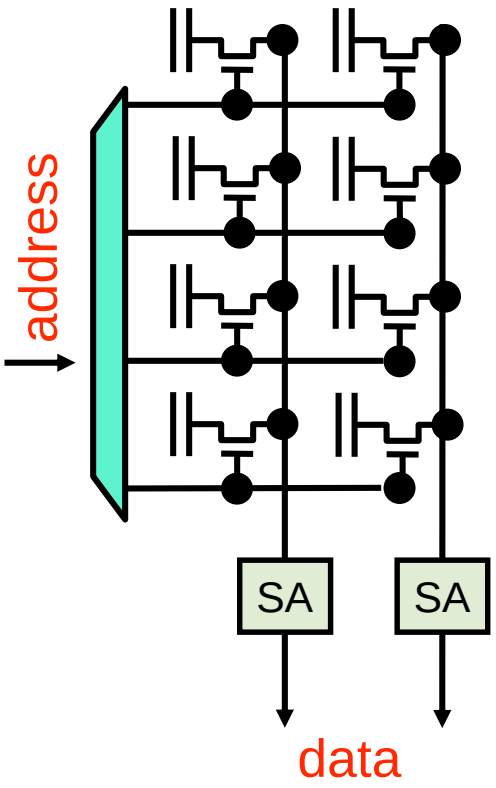
- Bit-line and capacitor equalize
- Settle out a bit above 0.5 if 1 was stored
  - A bit less if 0 was stored

DRAM Read (simplified version)



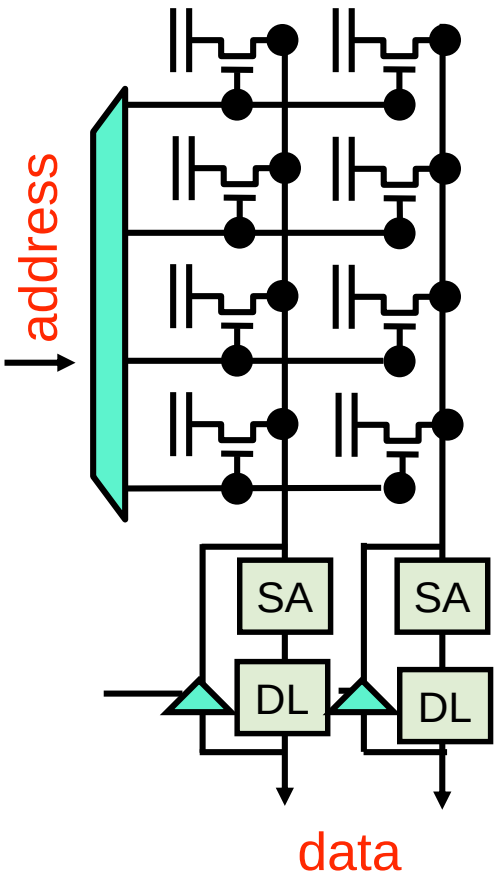
- Destroyed the stored value in the process
  - Could not read this again: change too small to detect

DRAM Operation I



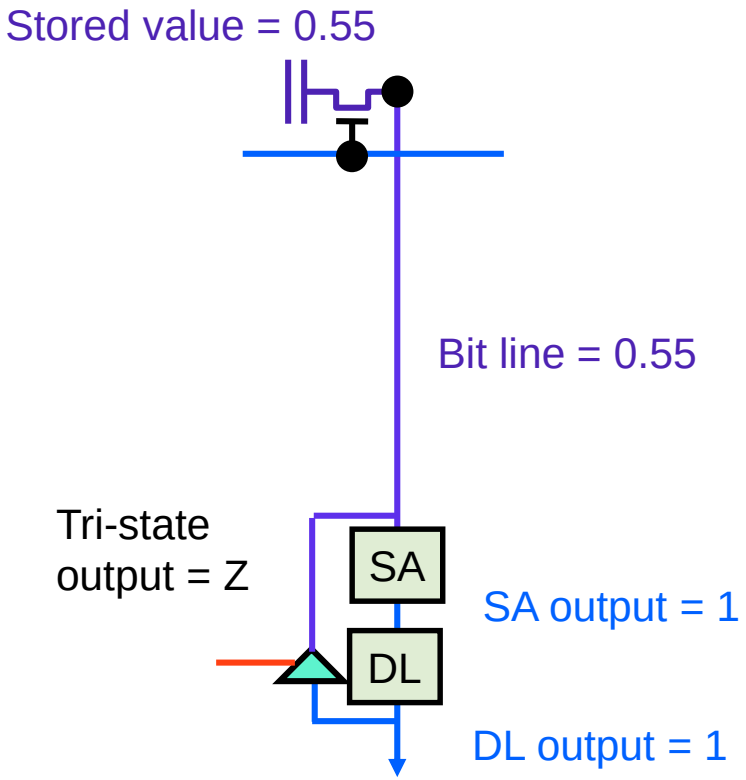
- Sense amps detect small swing
  - Amplify into 0 or 1
- This read: very slow
  - Why? No Vcc/Gnd connection in storage
- Need to deal with destructive reads:
  - Might want to read again…
- Also need to be able to write

DRAM Operation I



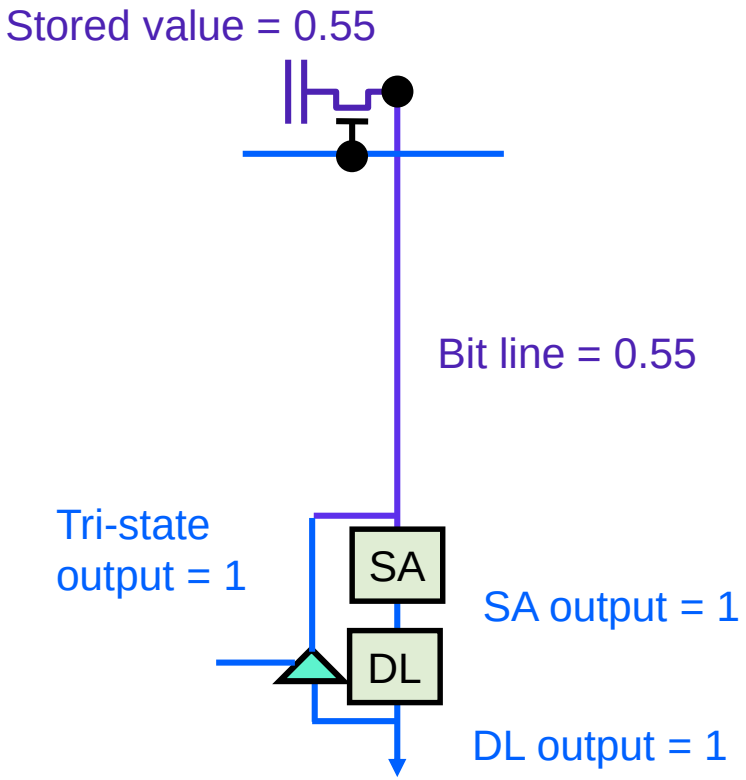
- Add some d-latches (**row buffer**)
  - Ok to use d-latches, not DFFs
  - No path from output->input when enabled
- Also add a tri-state path back
  - From the d-latch to the bit-line
  - Can drive the output of the d-latch onto bit lines
    - After we read, drive the value back
    - "Refill" (or re-empty) the capacitor

DRAM Read (better version)



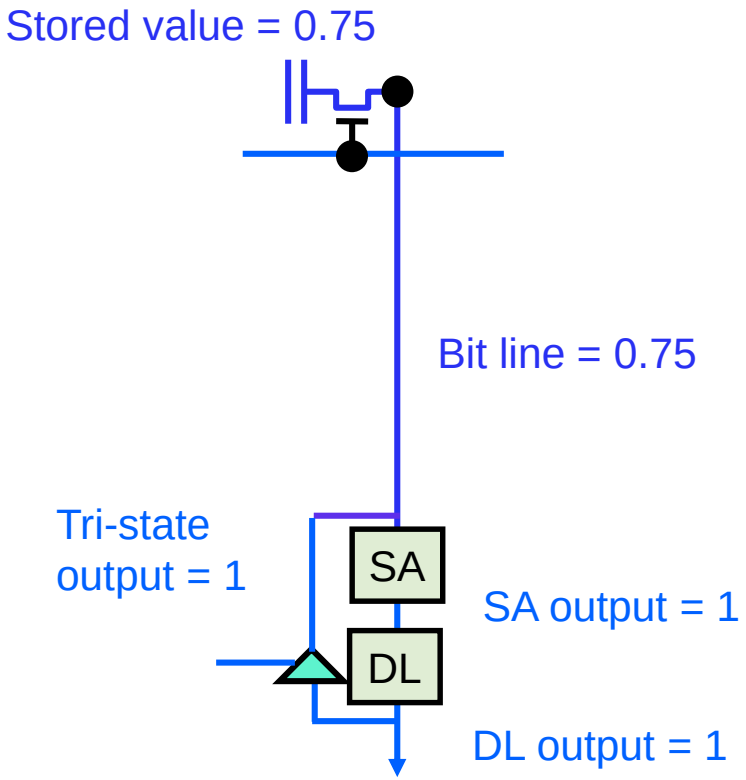
- SA amplifies 0.55 -> 1
- DL is enabled: latches the 1
- Tri-state disabled

DRAM Read (better version)



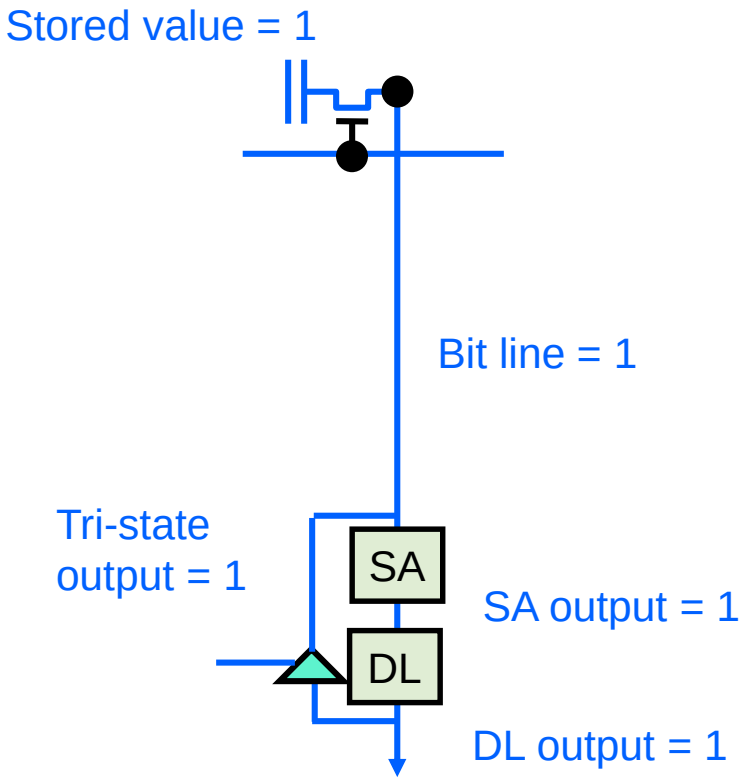
- Enable tri-state
  - Drives 1 back up bit-line

DRAM Read (better version)



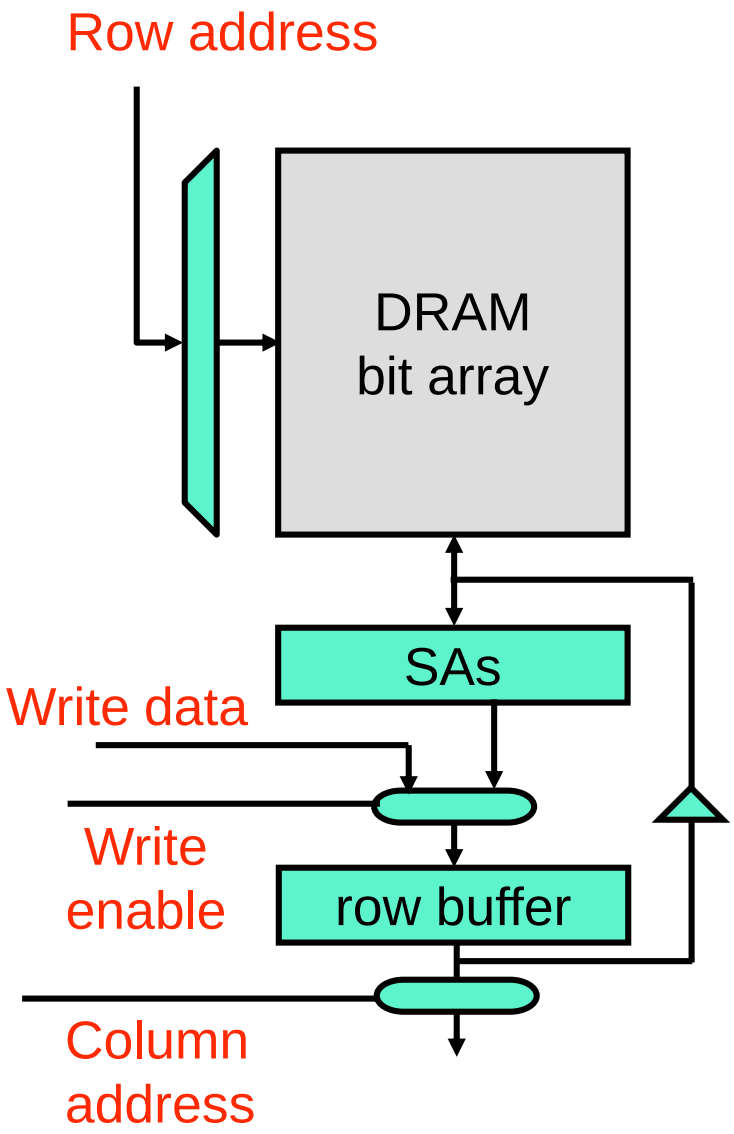
- Enable tri-state
  - Drives 1 back up bit-line
  - Starts to push value back up towards 1 (takes time)

DRAM Read (better version)



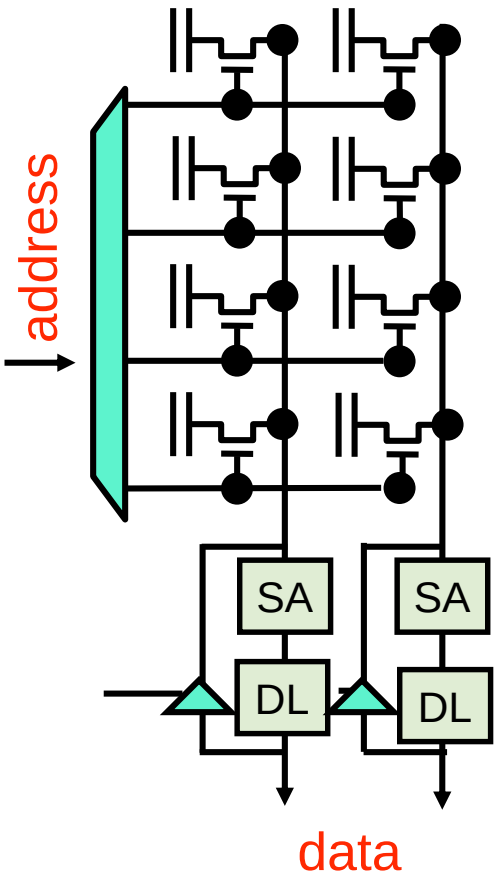
- Enable tri-state
  - Drives 1 back up bit-line
  - Starts to push value back up towards 1 (takes time)
  - Eventually restores value.

DRAM Operation



- Open row (read bits -> row buffer)
- Read "columns"
  - Mux selects right part of bits
  - Send data on bus -> processor
- Write "columns"
  - Change values in dlatches
- May read/write multiple columns
- Close row
  - Close access transistors
  - Pre-charge bit lines
- Row must remain open long enough
  - Must fully restore capacitors

DRAM Refresh



- DRAM periodically refreshes all contents
  - Loops through all rows
    - Open row (read -> RB)
    - Leave row open long enough
    - Close row
  - 1–2% of DRAM time occupied by refresh

Aside: Non-Volatile CMOS Storage

- Before we leave the subject of CMOS storage technology…
- Another important kind: **flash**
  - "Floating gate": no conductor/semi-conductor
    - Quantum tunneling involved in writing it
    - Effectively no leakage (key feature)
    - **Non-volatile**: remembers state when power is off
  - Slower than DRAM
  - Wears out with writes
    - Eventually writes just do not work

Memory Bus

- **Memory bus**: connects CPU package with main memory
  - Has its own clock
    - Typically slower than CPU internal clock: 100–500MHz vs. 3GHz
    - SDRAM operates on this clock
  - Is often itself internally pipelined
    - Clock implies bandwidth: 100MHz → start new transfer every 10ns
    - Clock doesn't imply latency: 100MHz !→ transfer takes 10ns
  - Bandwidth is more important: determines peak performance

Memory Latency and Bandwidth

- Nominal **clock frequency** applies to CPU and caches

- Careful when doing calculations
  - Clock frequency increases don't reduce memory or bus latency
  - May make misses come out faster
    - At some point memory bandwidth may become a **bottleneck**
    - Further increases in clock speed won't help at all

Clock Frequency Example

- Baseline setup
  - Processor clock: 1GHz.
    - 20% loads, 15% stores, 20% branches, 45% ALU
    - Branches: 3, ALU/stores: 4, Loads: 4 + t_{avgL1}
  - L1 D$: t_{hit} = 1 cycle, 10% miss
  - L2$: t_{hit} = 20 cycles, 5% miss
  - Memory: 200 cycles

Computation

t_{avgL2} = 20 + 0.05 * 200 = 30

t_{avgL1} = 1 + 0.10 * 30 = 4

Average load latency = 4 + 4 = 8

CPI = 0.2 * 8 + 0.15 * 4 + 0.2 * 3 + 0.45 * 4 = 4.6

The clock rate is 1GHz, or 1e9 cycles/second.
The CPI is 4.6 cycles/instruction.

Performance = (1e9 cycles/second) / (4.6 cycles/instruction)
= 217,391,304 instructions/second
= 217 MIPS

Clock Frequency Example

- Baseline setup
  - Processor clock: **2**GHz.
    - 20% loads, 15% stores, 20% branches, 45% ALU
    - Branches: 3, ALU/stores: 4, Loads: 4 + t_{avgL1}
  - L1 D$: t_{hit} = 1 cycle, 10% miss
  - L2$: t_{hit} = 20 cycles, 5% miss
  - Memory: **4**00 cycles

Computation

t_{avgL2} = 20 + 0.05 * **400** = **40**

t_{avgL1} = 1 + 0.10 * **40** = **5**

Average load latency = 4 + **5** = **9**

CPI = 0.2 * **9** + 0.15 * 4 + 0.2 * 3 + 0.45 * 4 = **4.8**

The clock rate is **2**GHz, or **2**e9 cycles/second.
The CPI is **4.8** cycles/instruction.

Performance = (**2e9** cycles/second) / (**4.8** cycles/instruction)
= **416,666,666** instructions/second
= **417** MIPS (91% speedup, for 100% freq increase)

Clock Frequency Example

- Baseline setup
  - Processor clock: **4**GHz.
    - 20% loads, 15% stores, 20% branches, 45% ALU
    - Branches: 3, ALU/stores: 4, Loads: 4 + t_{avgL1}
  - L1 D$: t_{hit} = 1 cycle, 10% miss
  - L2$: t_{hit} = 20 cycles, 5% miss
  - Memory: **8**00 cycles

Computation

t_{avgL2} = 20 + 0.05 * **800** = **60**

t_{avgL1} = 1 + 0.10 * **60** = **7**

Average load latency = 4 + **7** = **11**

CPI = 0.2 * **11** + 0.15 * 4 + 0.2 * 3 + 0.45 * 4 = **5.2**

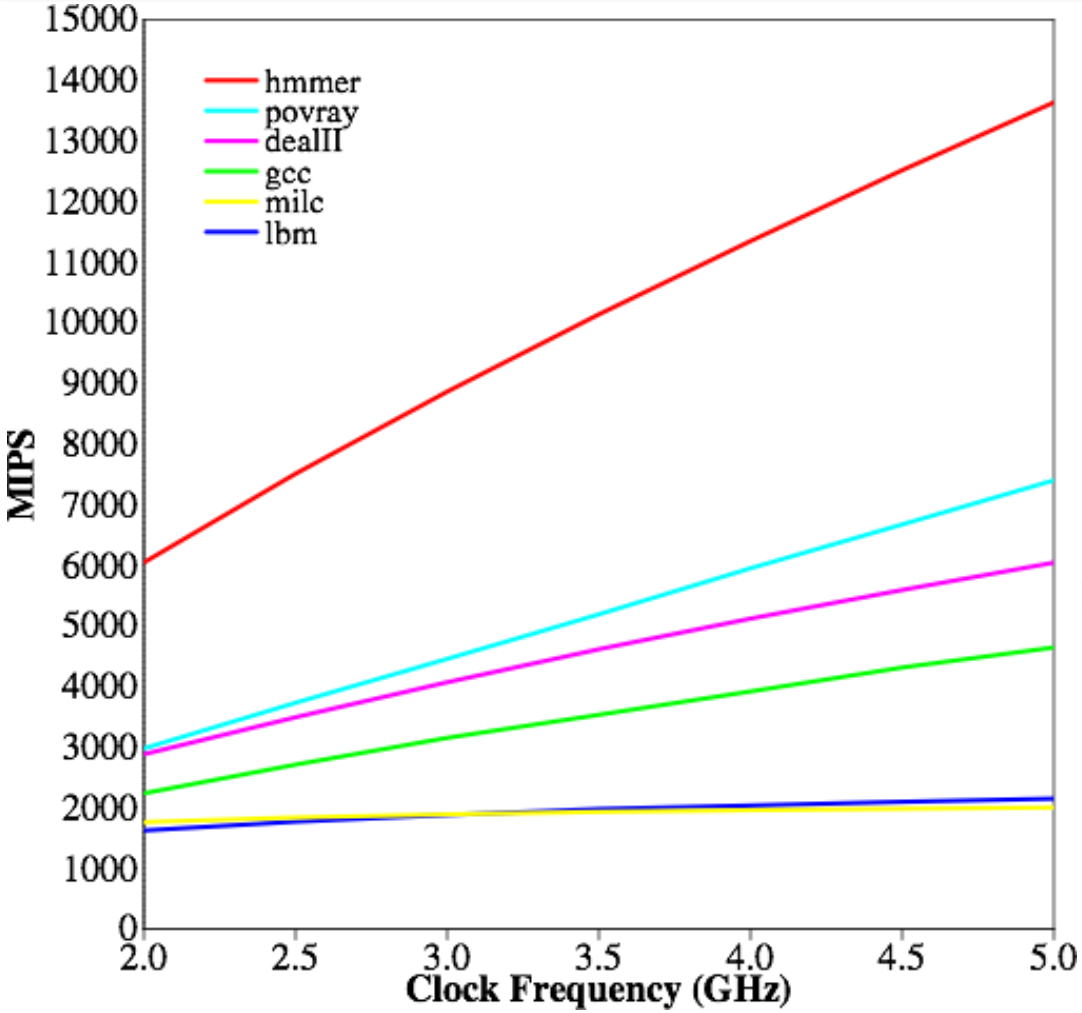
The clock rate is **4**GHz, or **4**e9 cycles/second.
The CPI is **5.2** cycles/instruction.

Performance = (**4e9** cycles/second) / (**5.2** cycles/instruction)
= **769,230,769** instructions/second
= **769** MIPS (84% speedup, for 100% freq increase)

Actually a Bit Worse..

- Only looked at D$ miss impact
  - Ignored store misses: assumed storebuffer can keep up
- Also have I$ misses
- At some point, become bandwidth constrained
  - Effectively makes t_{miss} go up (think of a traffic jam)
  - Also makes things we ignored matter
    - Storebuffer may not be able to keep up as well -> store stalls
  - Data we previously prefetched may not arrive in time
    - Effectively makes %miss go up

Clock Frequency and Real Programs

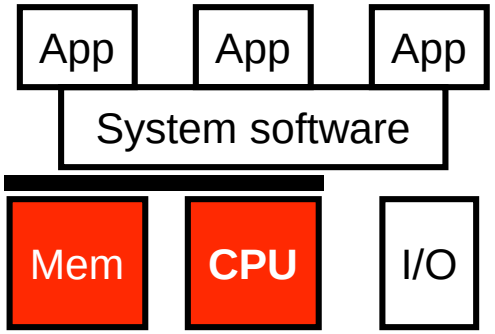


Detailed Simulation Results
- Includes all caches, bandwidth,...
- Has L3 on separate clock
- Real programs
- 2.0 Ghz -> 5.0 Ghz (150% increase)

hmmer:
- Very low %miss
- Good performance for clock
- **125%** speedup

lbm, milc:
- Very high %miss
- Not much performance gained
- lbm: **32%**
- milc: **14%**

Summary



- $t_{avg} = t_{hit} + \%_{miss} * t_{miss}$
  - t_{hit} and $\%_{miss}$ in one component? Difficult
- Memory hierarchy
  - Capacity: smaller, low t_{hit} → bigger, low$\%_{miss}$
    - 10/90 rule, temporal/spatial locality
  - Technology: expensive→cheaper
    - SRAM →DRAM→Disk: reasonable total cost
- Organizing a memory component
  - ABC, write policies
  - 3C miss model: how to eliminate misses?
- Technologies:
  - DRAM, SRAM, Flash