COMP311: *COMPUTER ORGANIZATION!*

Lecture 24: Caching

tinyurl.com/comp311-sp26

Logistics Check-in

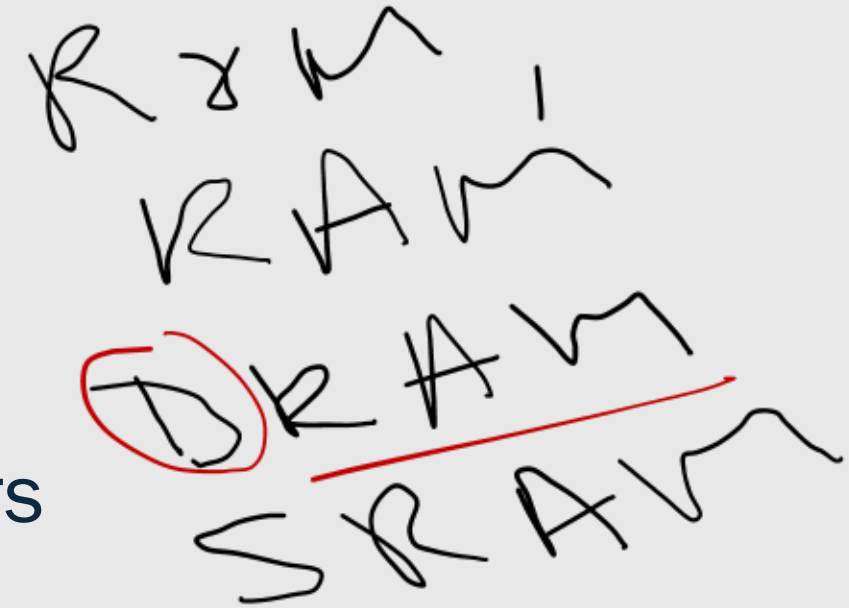
- Lab 5 due LDOC (4/27)
- Office hours ending LDOC
- Final review session on the first reading day (4/28)
  - *Time TBD*
- Monday was not Earth Day
  - *Today is!* ☺

MEMORY: AN OVERVIEW

These slides adapted from material from Leonard McMillian & Don Porter

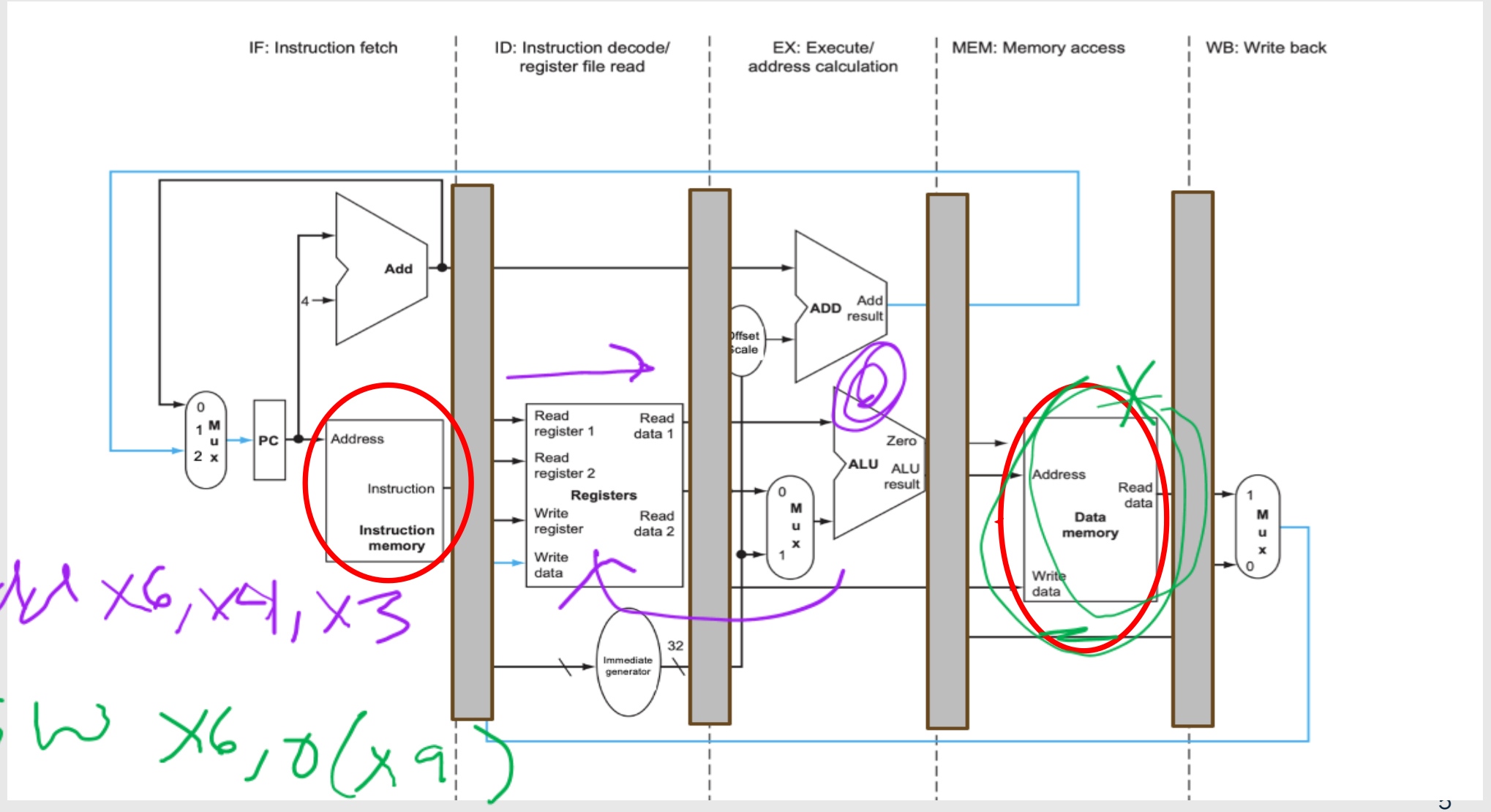
Topics

- Memory Arrays
- ~~Transparent Latches~~
- Edge-Triggered Registers
- Finite State Machines

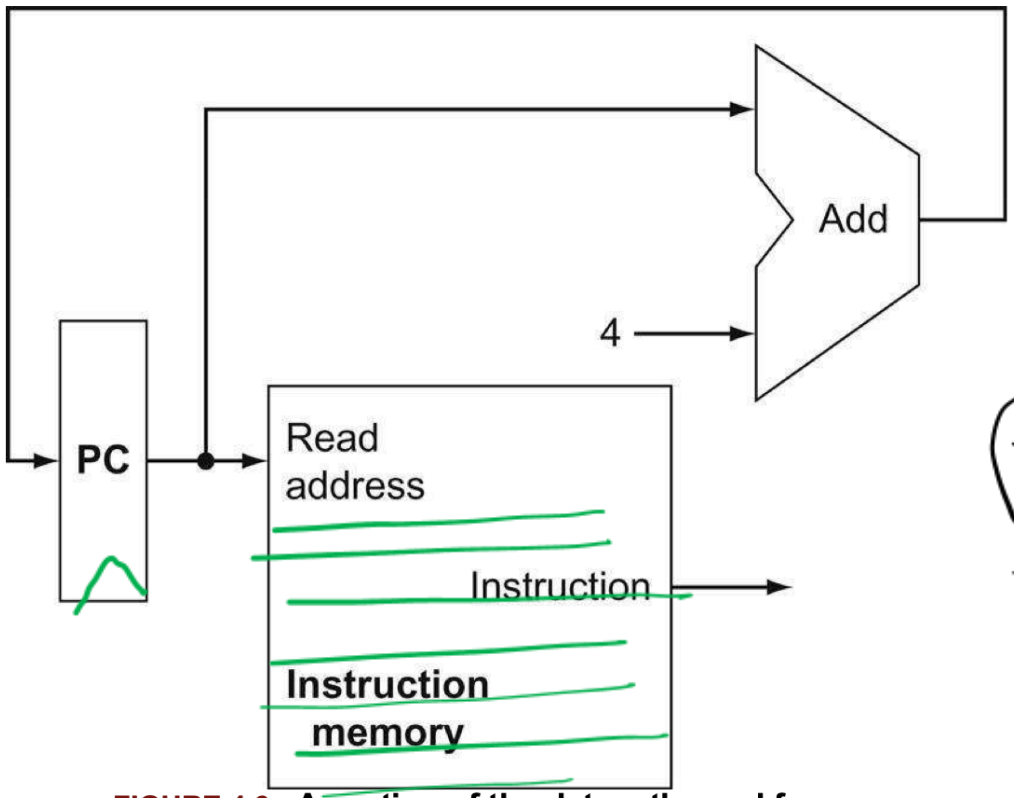


caching :)

Memory in the RISC-V Datapath. See Ch. 4!



RISC-V Datapath. See Chapter 4 & Appendix

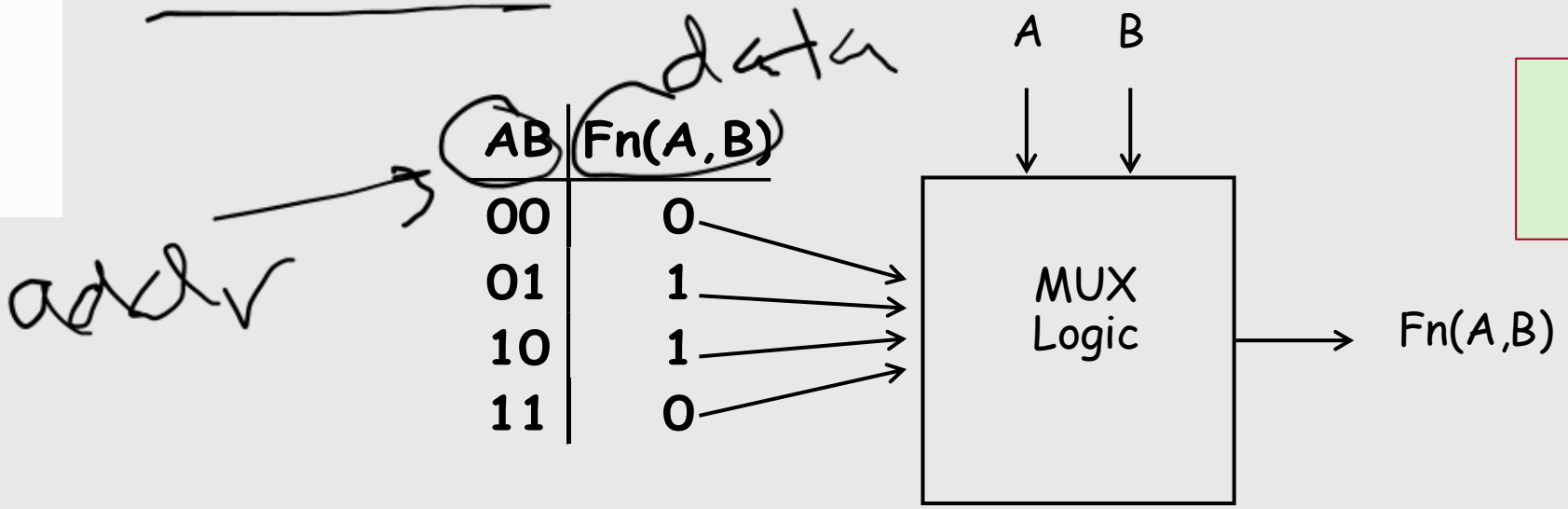


PC + 4

FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter.
The fetched instruction is used by other parts of the datapath.

In your lab, instruction memory is a ***ROM***

Read-Only Memory = a Lookup Table

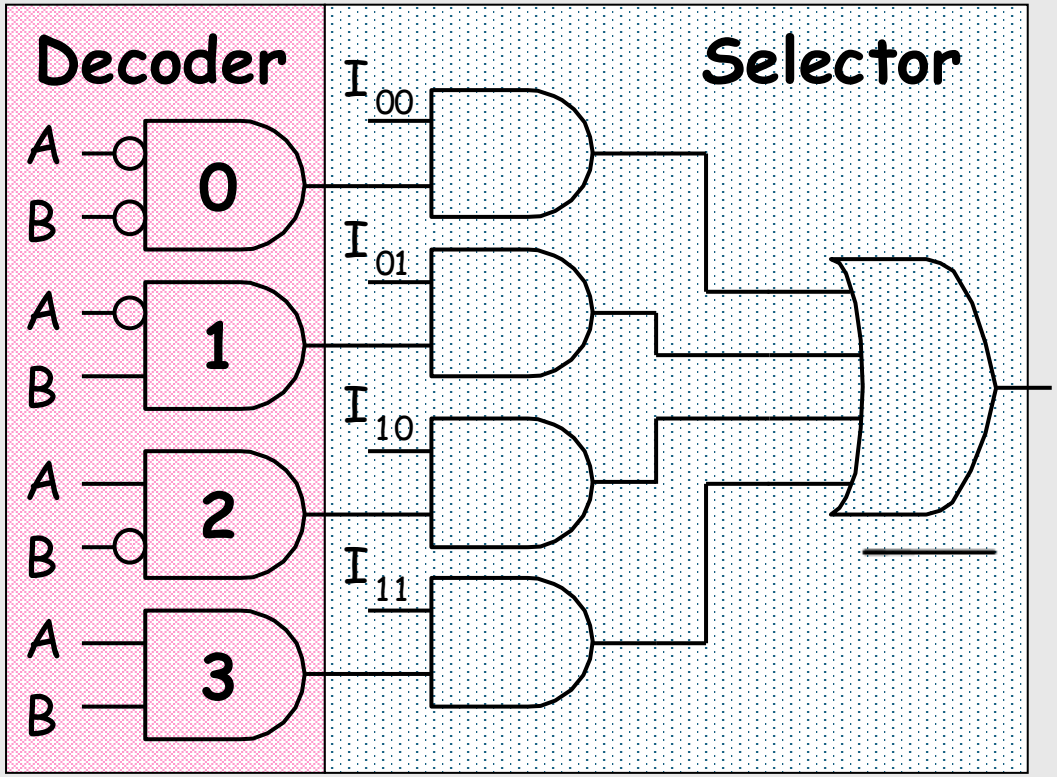


Read-Only Memory (ROM)

- A multiplexer can implement a lookup table:
  - *for an N-input function we need a 2^N input multiplexer*
  - *simply select one row out of 2^N*
  - *2^N values go into the mux, only one comes out*
  - *N-bit select input is now the "address" of data being accessed*
- Limitations:
  - *mux can get really big: 10-bit address → 1024 rows!*
  - *read-only; we also would like to write!*

Takeaway: ROM → "hardcoded truth table"
MUX → Hardware that selects correct output based on the input

A Mux's Guts: Decoder + Selector



A decoder generates all possible product terms for a set of inputs

Multiplexers can be partitioned into two sections.

A DECODER that identifies the desired input,and

a SELECTOR that enables that input onto the output.

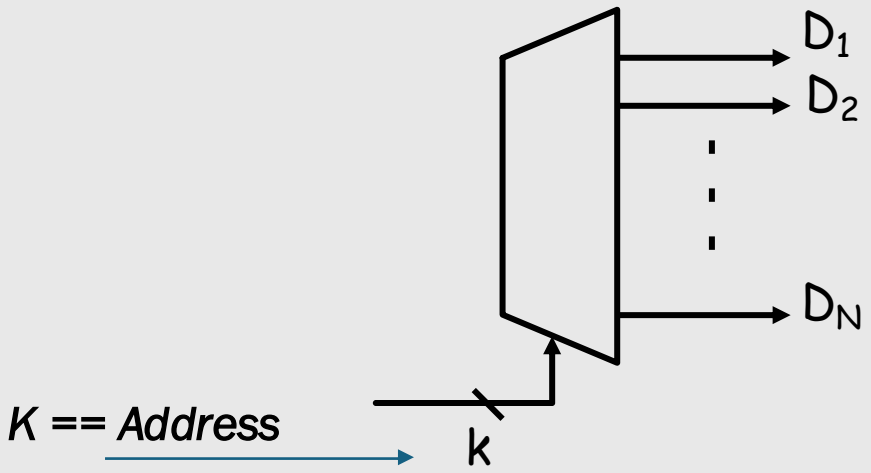
addressing

Key idea behind building ROMs: by **sharing the decoder** part of the logic MUXs could be adapted to make lookup tables with **any number of outputs**

Decoder

4-to-1 mux

1-to-4 mux



DECODER:

k SELECT inputs,

N = 2^k DATA OUTPUTs.

Selected D_j HIGH; all others LOW.

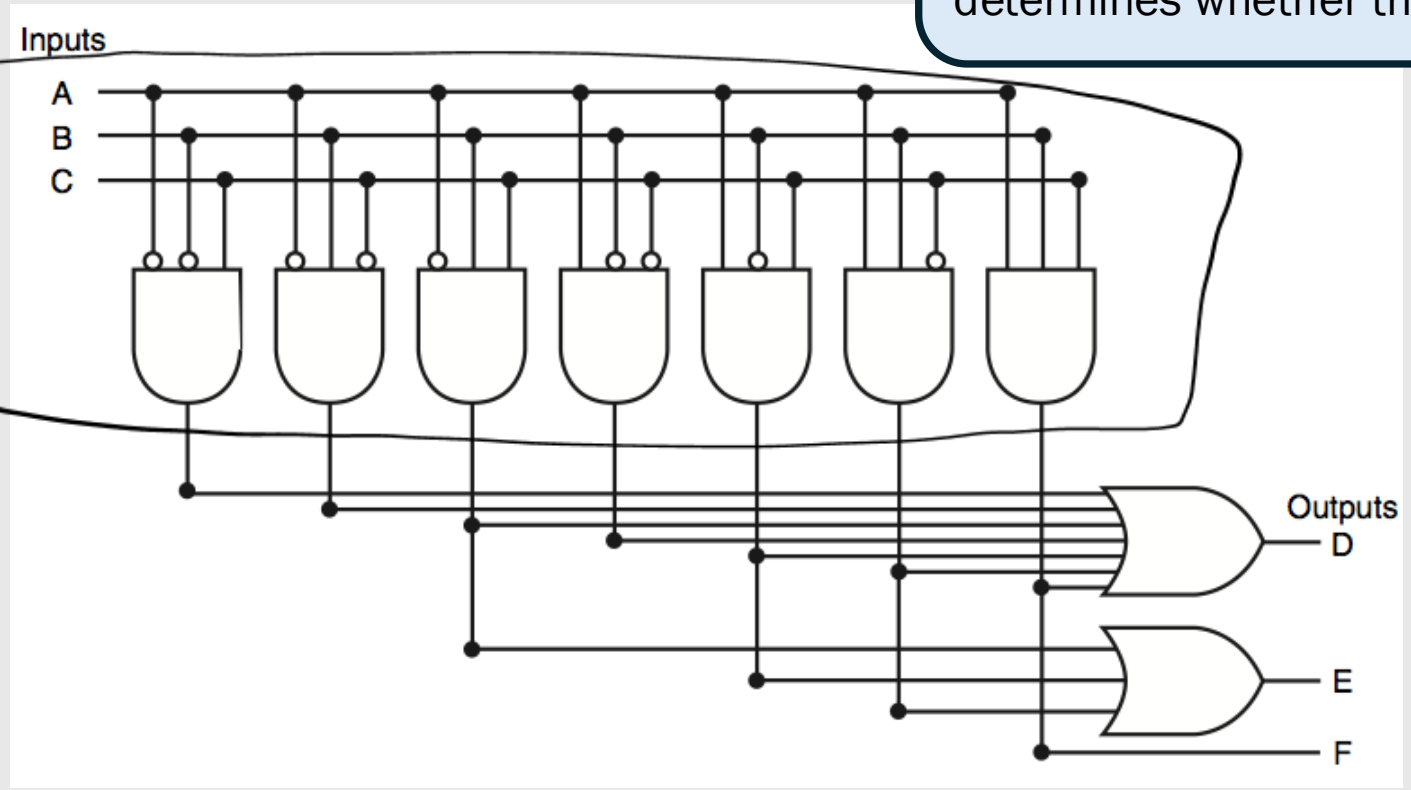
One-hot signal to identify row we are working with

NOW, we are well on our way to building a general purpose table-lookup device.

We can build a 2-dimensional ARRAY of decoders and selectors as follows by arranging them in a grid…

Shared Decoding Logic

This structure is exactly what a ROM does: the decoder selects a row (the address), and each output corresponds to a column that determines whether that row produces a 1 or 0.



We can build a general purpose “table-lookup” device called a Read-Only Memory (ROM), from which we can implement any truth table and, thus, any combinational device

Logic According to ROMs

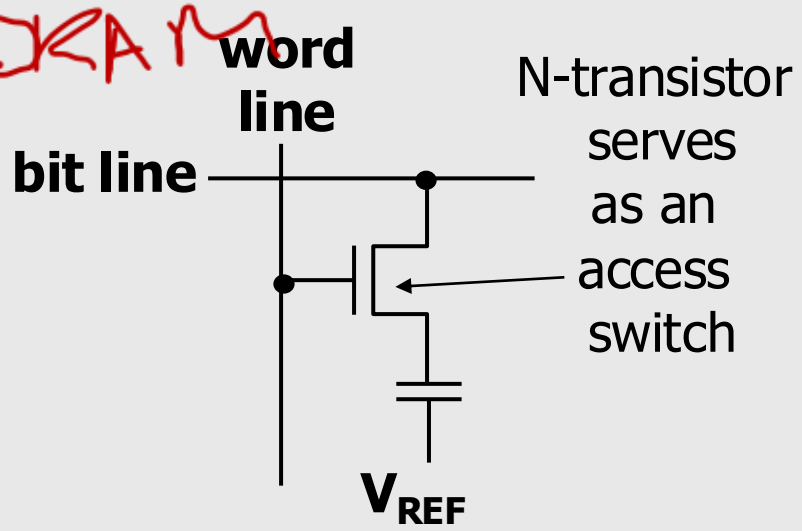
Store fixed values to be read

- Different ROM implementation technologies
  - *Mask (old) → ROM*
    - read-only memory
  - *Fuses (old) → PROM*
    - programmable read-only memory
  - *Erasable → EPROM*
    - erasable programmable read-only memory
  - *Electrically erasable → EEPROM*
    - electrically-erasable programmable read-only memory
    - → today called **FLASH**! Used everywhere!
- Model: LOOK UP value of function in truth table…
  - *Inputs: "ADDRESS" of a truth table entry*
  - *ROM SIZE = # truth table entries…*
    - … for an N-input boolean function, size = $2^N \times \#\text{outputs}$

2^N

Read-Write Memories *(RAM)*

- Read-Write often called **random-access memories (RAM)**
- Dynamic RAM uses analog storage:
  - *encode information using voltages*
  - *from physics: we can "store" a voltage as "charge" on a capacitor*

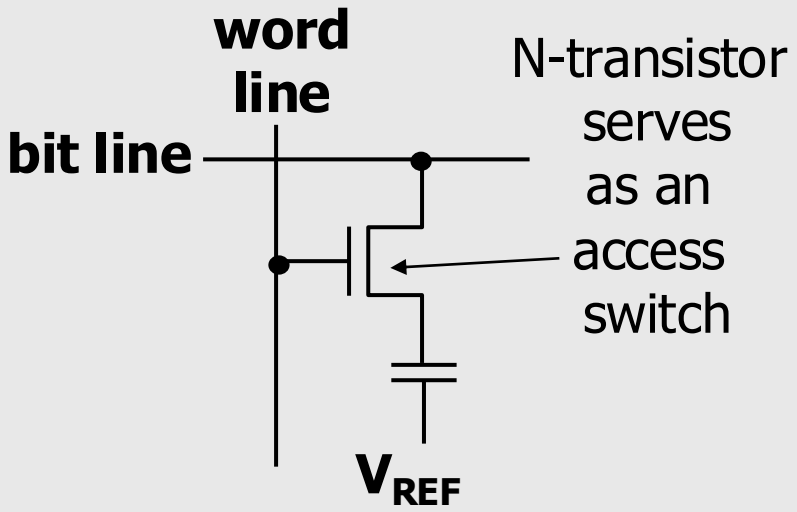


To write:
Drive bit line, turn on access transistor, force storage cap to new voltage

To read:
precharge bit line, turn on access trans., detect (small) change in bit line voltage

Read-Write Memories *(RAM)*

- Dynamic RAM uses analog storage:
  - *encode information using voltages*
  - *from physics: we can "store" a voltage as "charge" on a capacitor*
  - *Pros:*
    - compact!
  - *Cons:*
    - it leaks! -> refresh
    - complex interface
    - reading a bit, destroys it
      - *(you have to rewrite the value after each read)*
    - it's NOT a digital circuit



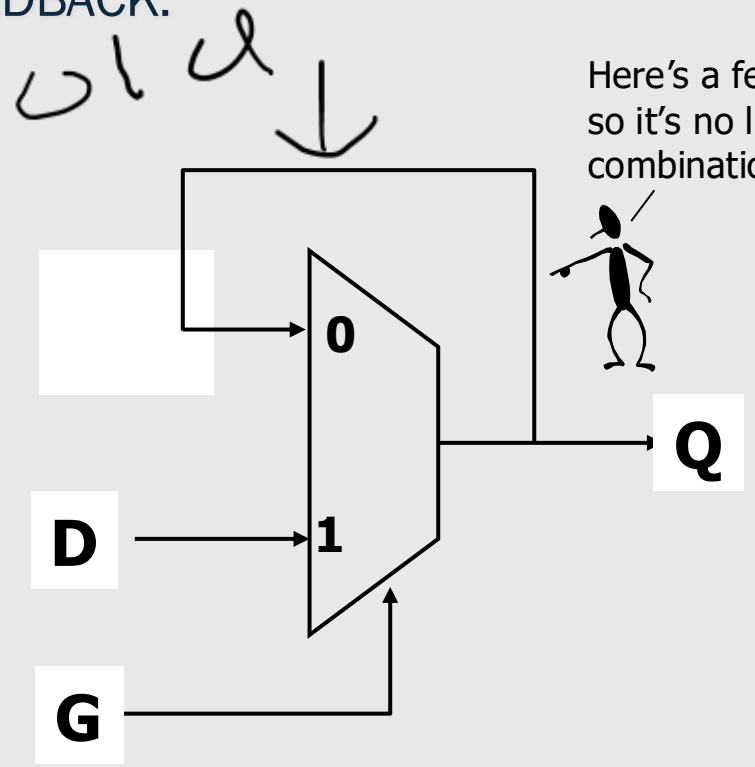
To write:
Drive bit line, turn on access transistor, force storage cap to new voltage

To read:
precharge bit line, turn on access trans., detect (small) change in bit line voltage

This storage circuit is the basis for commodity DRAMs

Static RAM (SRAM): A "Digital" Storage Element

- It's also easy to build a DIGITAL storage element (called a latch) using a MUX and FEEDBACK:

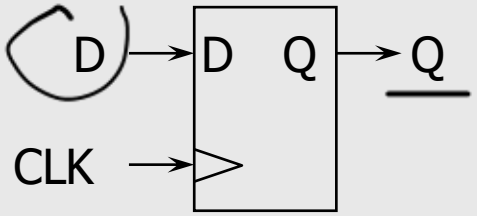


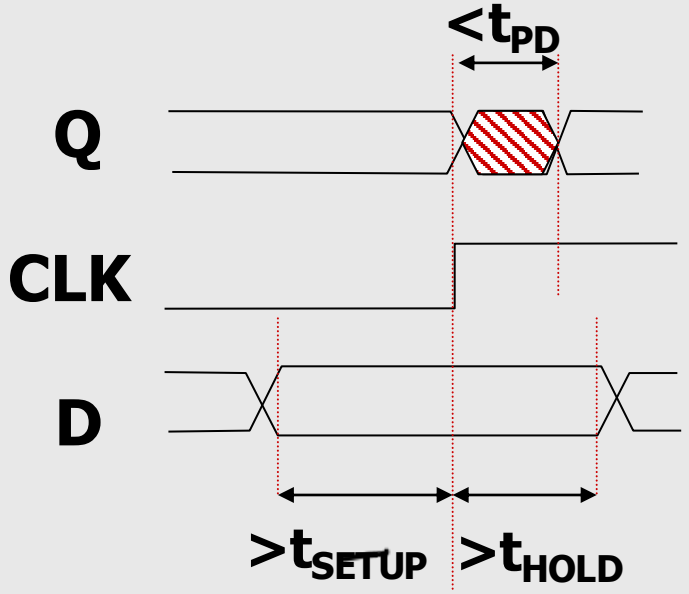
Here's a feedback path, so it's no longer a combinational circuit.

"state" signal appears as both input and output

G	D	Q_{IN}	Q_{OUT}	
0	--	0	0	Q stable
0	--	1	1	
1	0	--	0	Q follows D
1	1	--	1	

Flip Flop Timing





clk-to-q dly

t_{PD}: maximum propagation delay, CLK→Q

t_{SETUP}: setup time

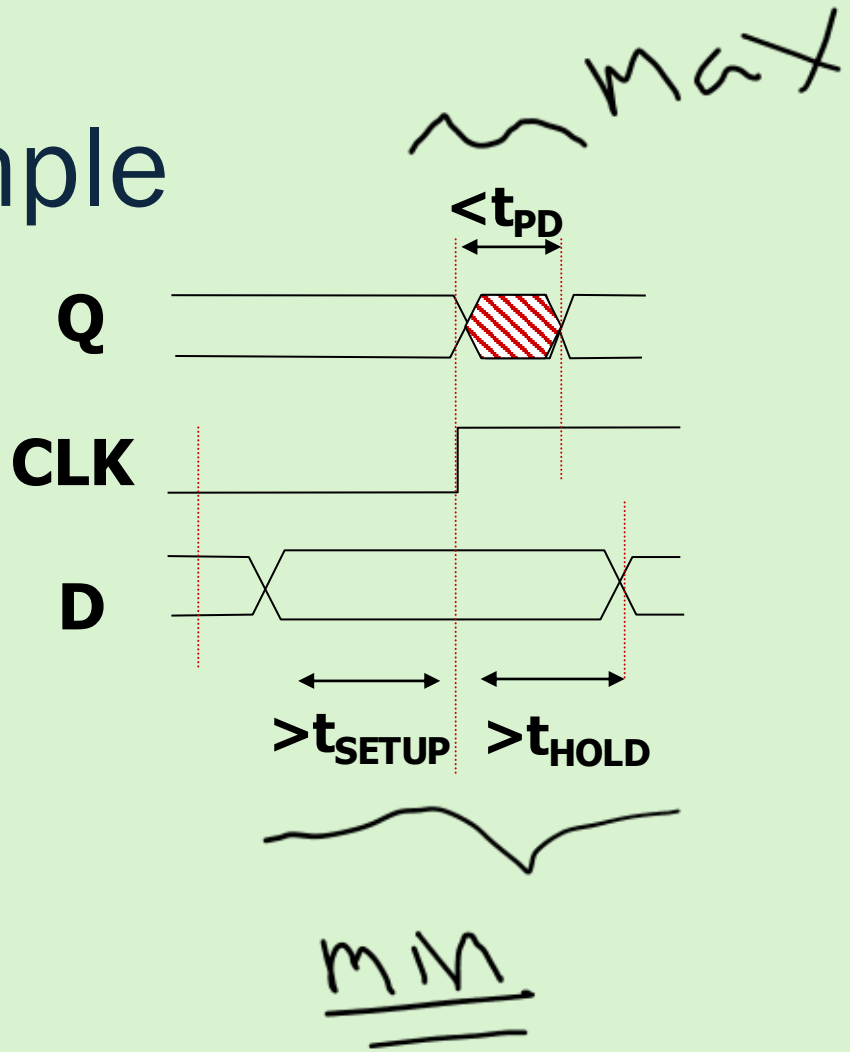
D must be stable for at least this long before the clock edge so the flip-flop can correctly capture it

t_{HOLD}: hold time

D must remain stable for at least this long after the clock edge so the value is reliably restored

Intuition: There is a small “no-change” window around the clock edge (before: setup, after: hold)
If D, changes in this window, flip-flop may capture the wrong value.

Flip Flop Timing Example



A flip-flop has the following timing parameters:
t_{setup} = 2 ns, t_{hold} = 1 ns, t_{PD} = 3 ns.

A rising clock edge occurs at 10 ns.

1. During what time interval must D remain stable so the value is captured correctly?

2. By what latest time should Q have updated after the clock edge?

Flip Flop Timing Example

A flip-flop has the following timing parameters:
t_{setup} = 2 ns, t_{hold} = 1 ns, t_{PD} = 3 ns.

A rising clock edge occurs at 10 ns.

1. During what time interval must D remain stable so the value is captured correctly?

2. By what latest time should Q have updated after the clock edge?

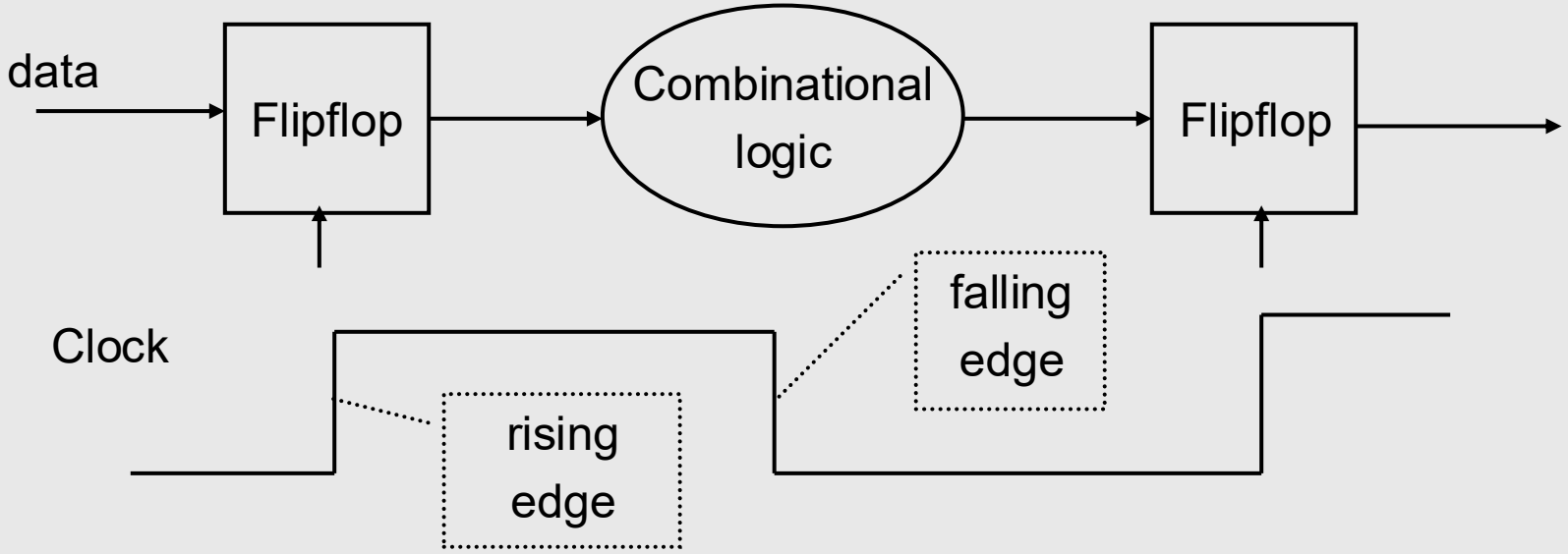
1. 8 – 11 ns.

2. 10 + 3 ⟹ 13 ns

SYNCHRONOUS SYSTEMS

(Reminder: Designing digital systems using a clock)

Synchronous Systems



On the leading edge of the clock, the input of a flipflop is transferred to the output and held.

We must be sure the output of the combinational logic has *settled* before the next leading clock edge.

Clock period must be "long enough"

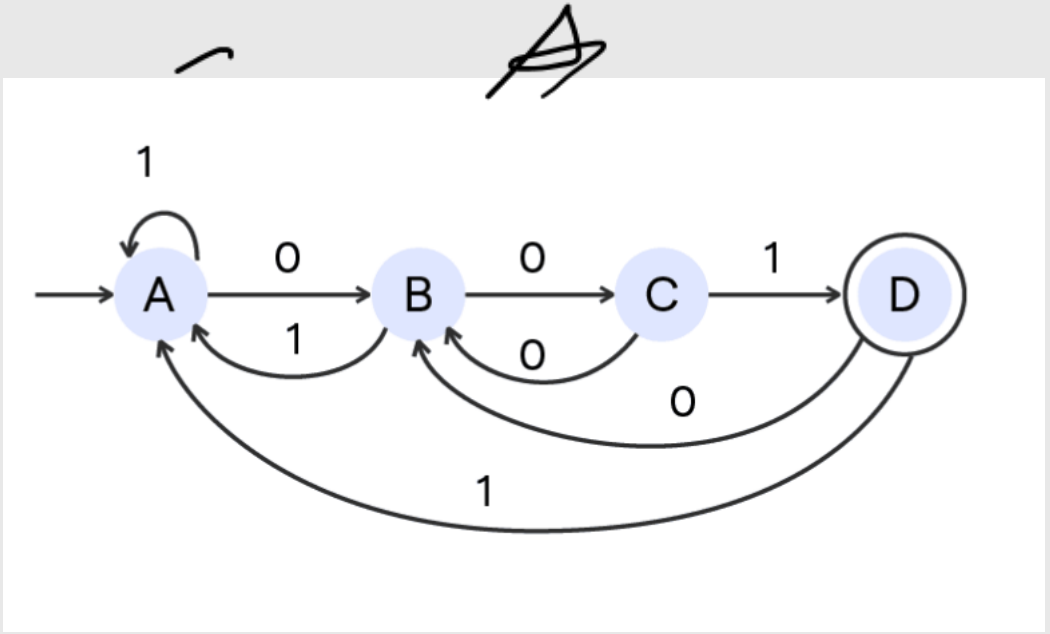
Clock Period >= $t_{FF} + t_{logic} + t_{setup}$

Setup time is the minimum amount of time the input (D) must be stable *before* the clock edge so the flip-flop can reliably capture it.

Finite State Machines

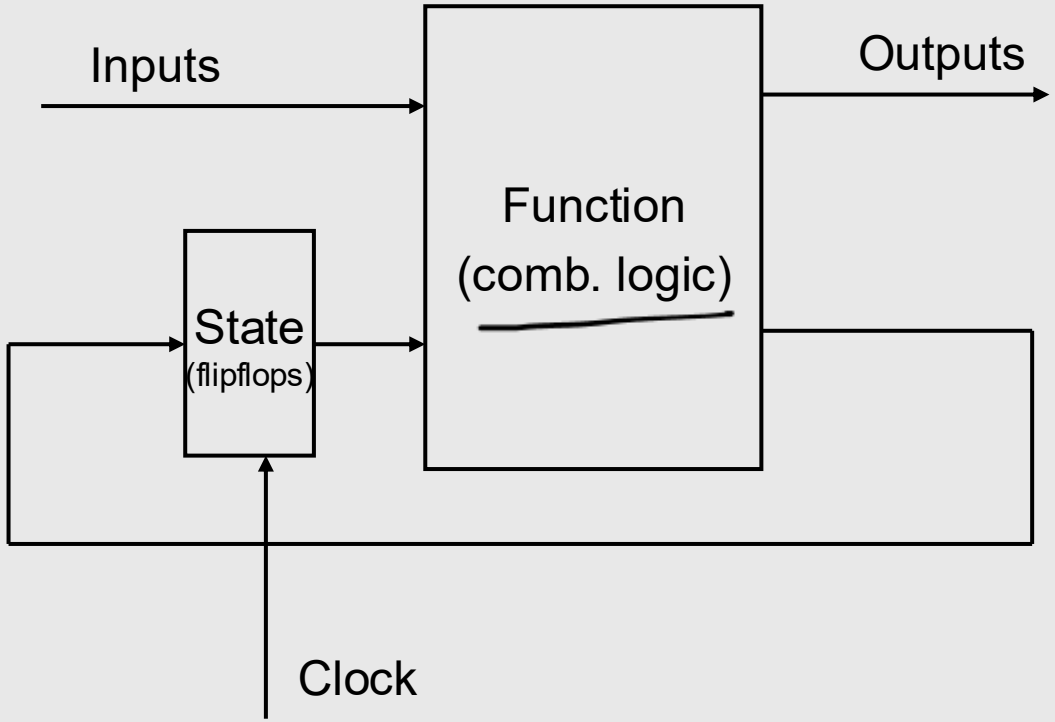
- What is a State Machine?
  - *Automata from 455* ☺
  - *It is defined by the following:*
    - Set of STATES
    - Set of INPUTS
    - Set of OUTPUTS
    - A mapping from (STATES, INPUTS) to …

      … the next STATE and an OUTPUT
  - *STATE represents memory!*



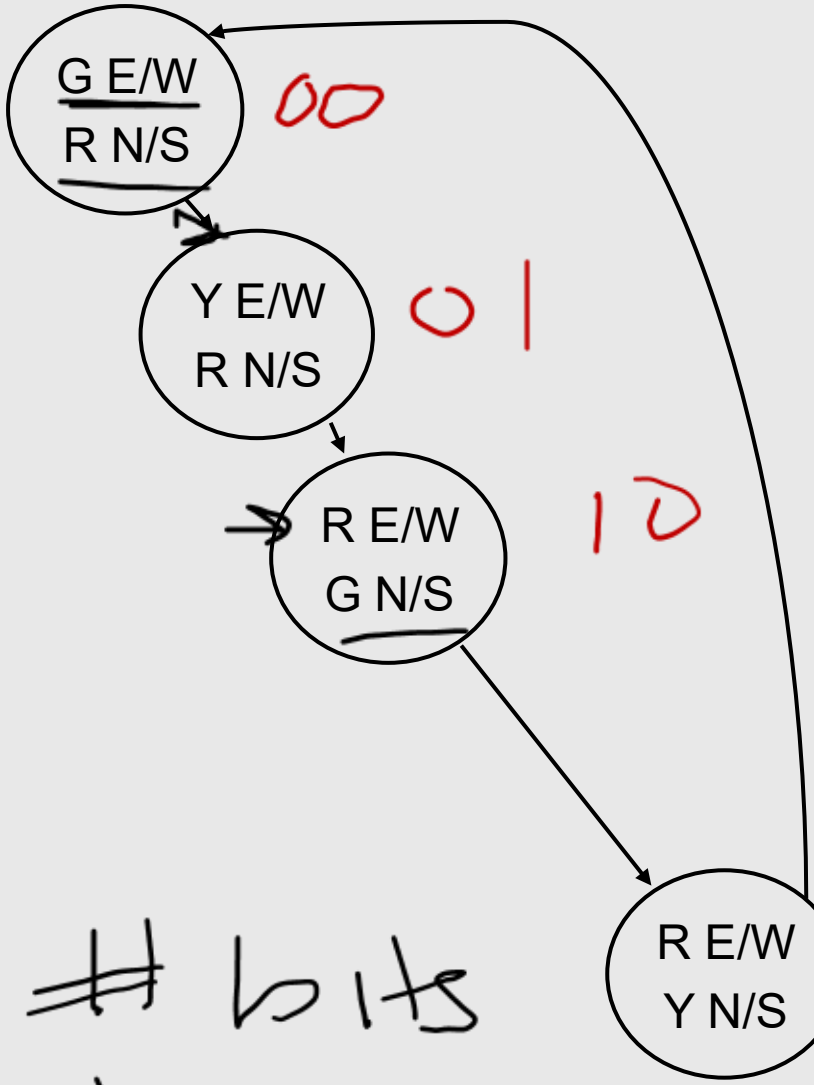
Implementing an FSM

- Use flipflops to represent state (i.e., memory)
- Use combinational logic to compute:
  - *output as a function of inputs + state*
  - *next state as a function of inputs + state*



Example: Traffic Light Controller

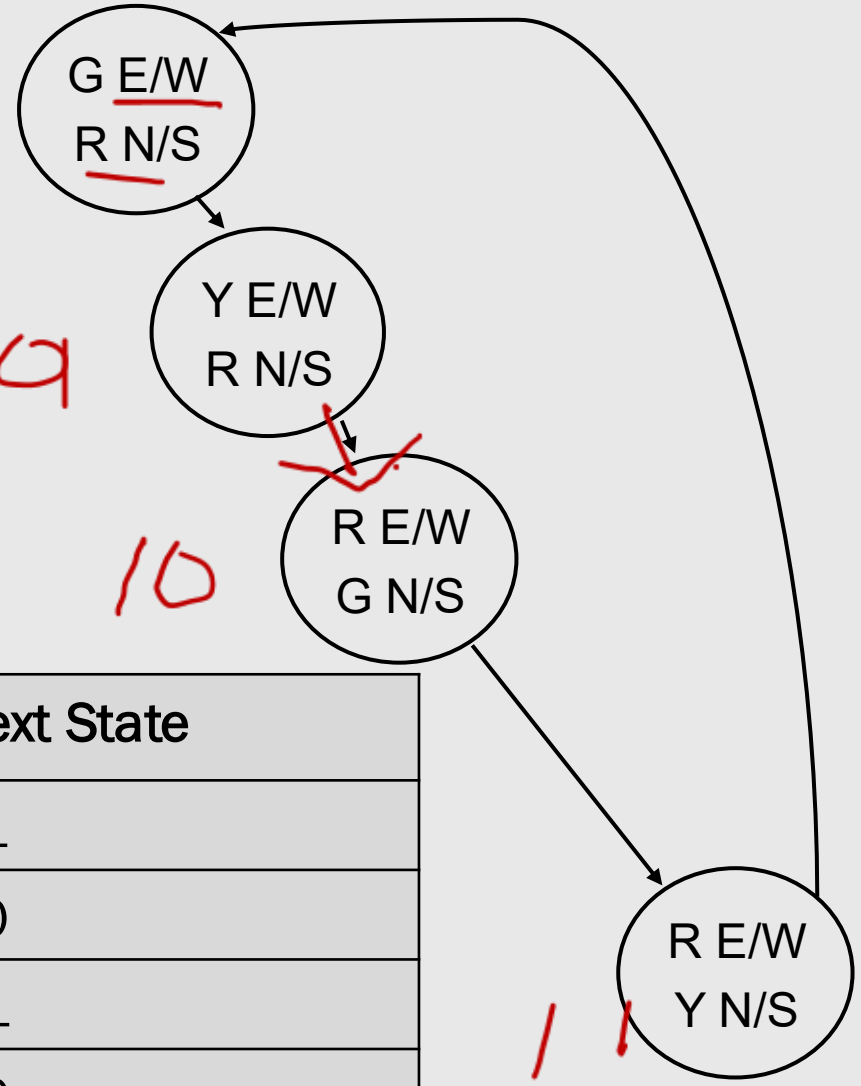
- 4 states
  - *each corresponding to a circle*
  - *represent states using 2 bits*
    - 00, 01, 10, 11
- 4 bits of output
  - *2 bits for east/west light*
  - *2 bits for north/south light*
    - Red (00)
    - Yellow (01)
    - Green (10)



$\lceil \log_2(\text{states}) \rceil = \#$ bits to encode

Example: Traffic Light Controller

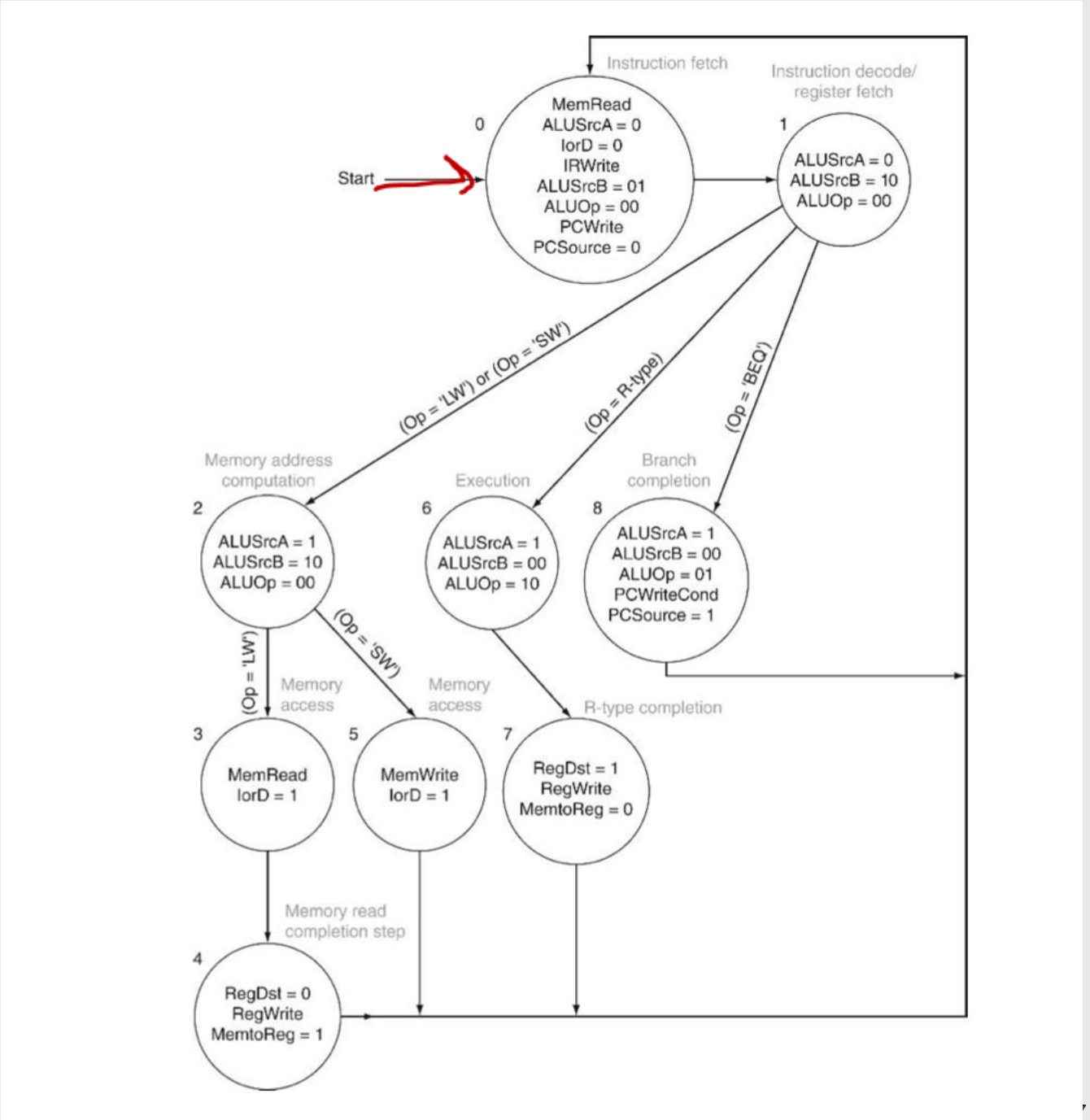
- 4 states
  - 00, 01, 10, 11
  - need 2 flipflops to represent state
- 4 bits of output
  - R (00), Y (01), G (10)
- Truth table:
  - this defines the combinational logic part of the FSM



Input (State)	Output	Next State
00	10 00	01
01	01 00	10
10	00 10	11
11	00 01	00

FSMs in the CPU

The complete finite-state machine control for the RISC-V datapath



Summary

- Regular Arrays can be used to implement arbitrary logic functions
- Memories
  - *ROMs are HARDWIRED memories*
  - *RAMs include storage elements that are read-write*
    - dynamic memory: compact, only reliable short-term
    - static memory: controlled use of positive feedback
- For static storage:
  - *Some timing constraints: setup and hold times*
- Finite State Machines:
  - *Implement using flip-flops for memory, and combinational logic to compute output, next state*

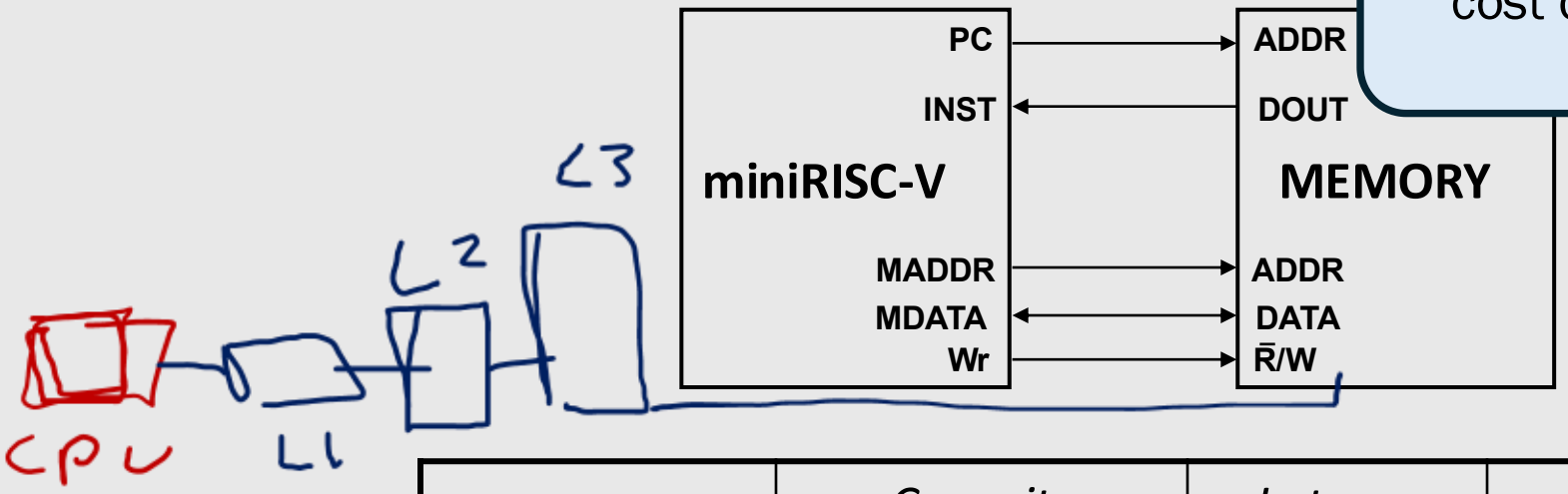
MEMORY HIERARCHY … INTRO TO CACHING!)

Topics

- Memory Flavors
- Principle of Locality
- What is Caching
- Associativity

What Do We Want in a M

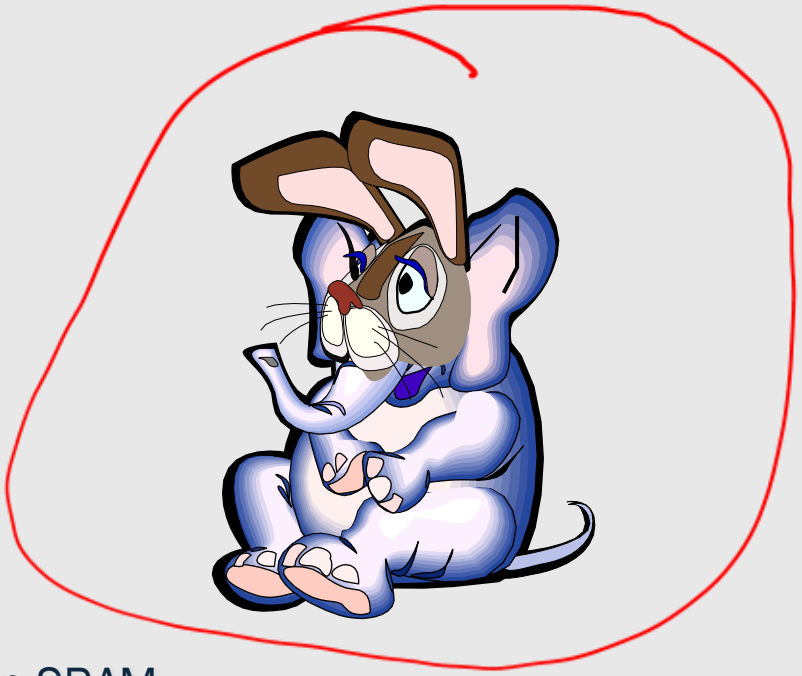
Observation: As capacity goes up and cost does down, latency gets worse.



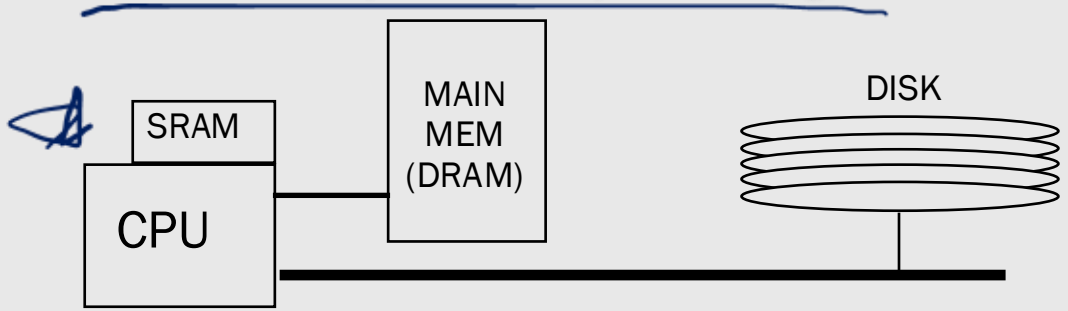
	Capacity	*Latency*	*Cost*
Register	1000s of bits	10 ps	$$$$
SRAM	1-8 MBytes	0.5-1 ns	$2K/GB
DRAM	1-8 GBytes	50 ns	$20/GB
Hard disk*	100s GBytes	10 ms	20¢/GB
Want?	Huge	Low (fast)	Cheap!

*non-volatile: Typically magnetic disk drive, but also SSD/flash now

Best of Both Worlds



- What we REALLY want: A BIG, FAST memory!
  - *Keep everything within instant access*
- We'd like to have a memory system that
  - *Stores 2-16 GB of data like DRAM would*
    - typical SRAM sizes are in MB, not GB
  - *Performs fast like SRAM would*
    - typical DRAMs are order of magnitude slower than SRAM
  - *SURPRISE: We can (nearly) get our wish!*
    - Key Idea: Use a hierarchy of memory technologies:



Principle of Locality

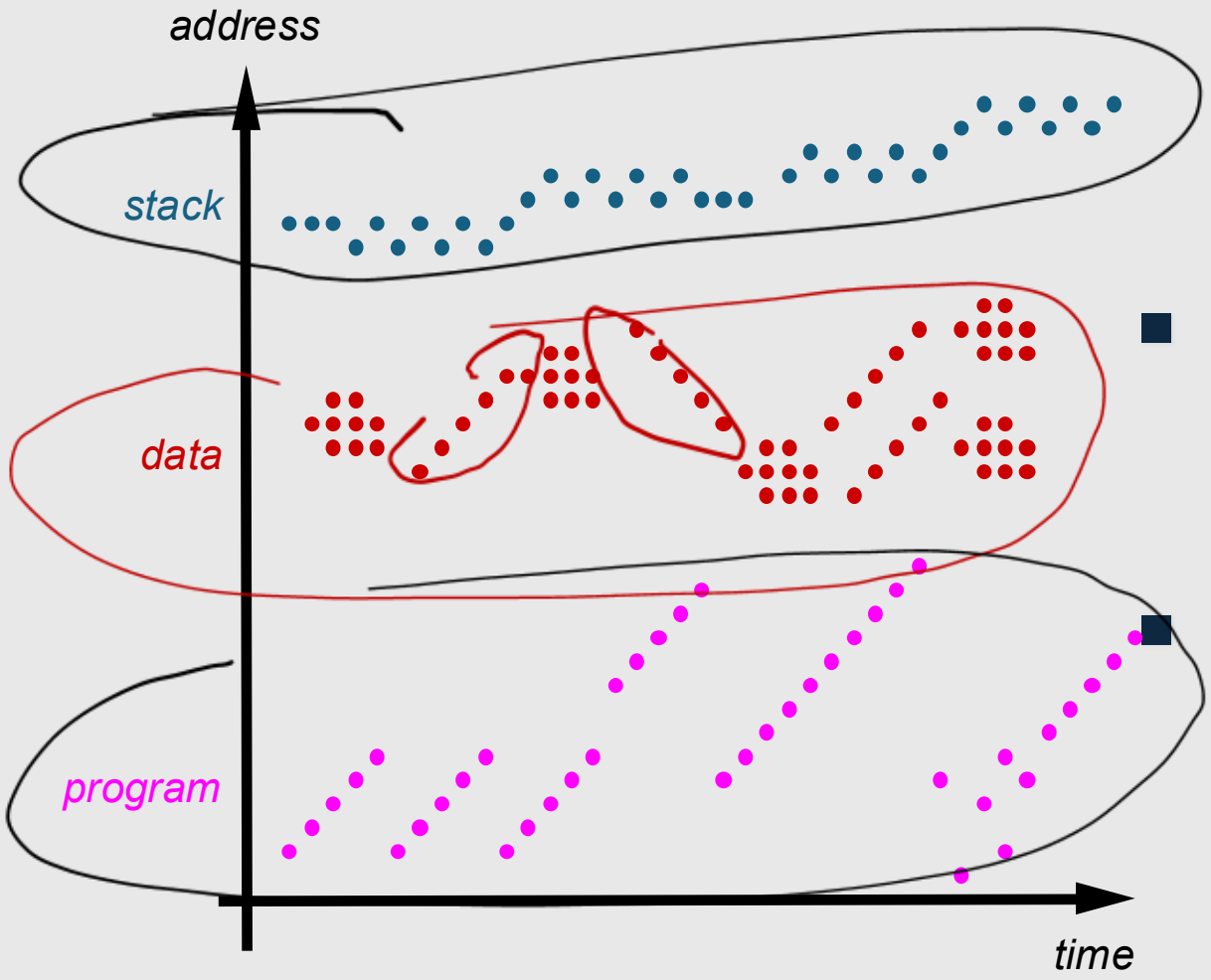
- Key Idea: Exploit "Principle of Locality"
  - *Keep data used often in a small fast SRAM*
    - called "CACHE", often on the same chip as the CPU
  - *Keep all data in a bigger but slower DRAM*
    - called "main memory", usually separate chip
  - *Access Main Memory only rarely, for remaining data*
  - *The reason this strategy works: LOCALITY*
    - if you access something now, you will likely access it again (or its neighbors) soon

Locality of Reference:

Reference to location X at time t implies that reference to location $X+\Delta X$ at time $t+\Delta t$ is likely for small ΔX and Δt.

Put simply: If a location is accessed in memory, the **same location** or its **neighbors** will likely be accessed soon.

Typical Memory Reference Patterns



- Memory Trace
  - *A temporal sequence of memory references (addresses) from a real program.*
- Temporal Locality
  - *If an item is referenced, it will tend to be referenced again soon*
- Spatial Locality
  - *If an item is referenced, nearby items will tend to be referenced soon.*

Cache

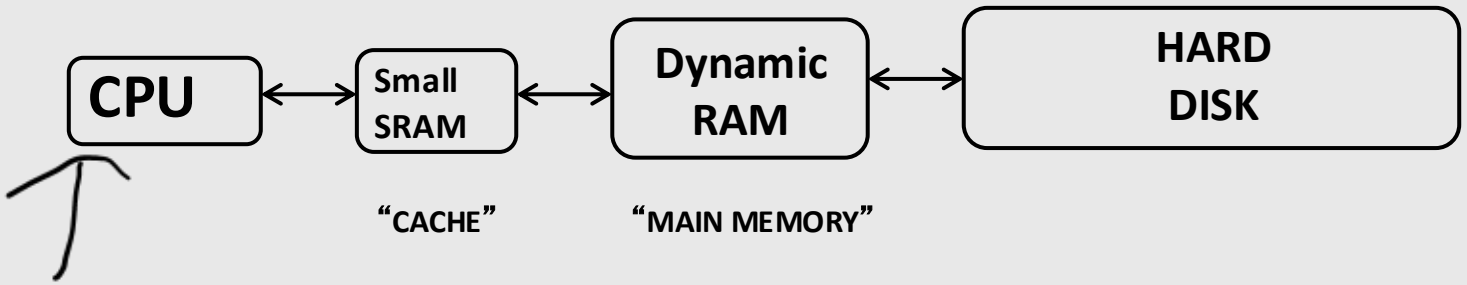
- cache (kash)
  n.
  - *A hiding place used especially for storing provisions.*
  - *A place for concealment and safekeeping, as of valuables.*
  - *The store of goods or valuables concealed in a hiding place.*
  - *Computer Science. A fast storage buffer in the central processing unit of a computer. In this sense, also called cache memory.*
- *v. tr.* cached, cach·ing, cach·es.
  - *To hide or store in a cache.*

Cache Analogy

- You are writing a term paper for your history class at a table in the library
  - *As you work you realize you need a book*
  - *You stop writing, fetch the reference, continue writing*
  - *You don't immediately return the book, maybe you'll need it again*
  - *Soon you have a few books at your table, and you can work smoothly without needing to fetch more books from the shelves*
  - *The table is a CACHE for the rest of the library*
- Now you switch to doing your biology homework
  - *You need to fetch your biology textbook from the shelf*
  - *If your table is full, you need to return one of the history books back to the shelf to make room for the biology book*

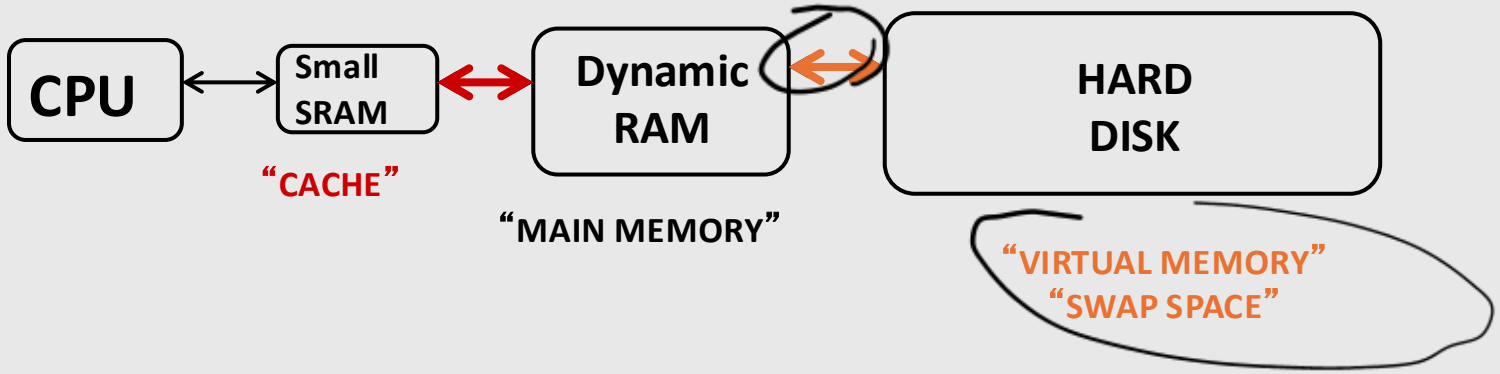
Exploiting the Memory Hierarchy

- Memory Hierarchy is hidden from programmer!
  - *Programming model: SINGLE kind of memory, single address space.*
  - *Transparent to programmer: Machine AUTOMATICALLY assigns locations, depending on runtime usage patterns.*
    - programmer doesn't (***cannot***) know where the data is actually stored!



Exploiting the Memory Hierarchy

- CPU speed is dominated by memory performance
  - *More significant than: ISA, circuit optimization, pipelining, etc.*

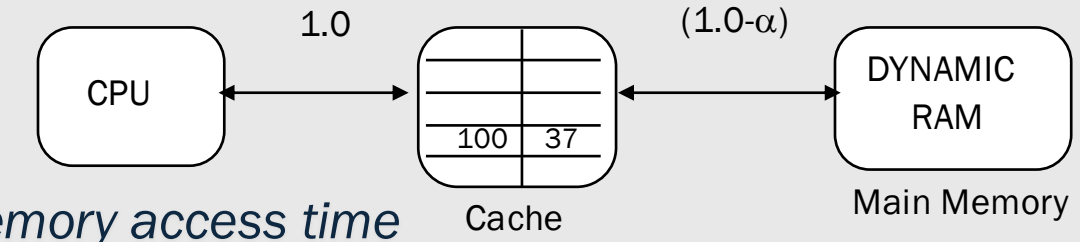


- *TRICK #1: Make slow MAIN MEMORY appear faster*
  - Technique: CACHING
- *TRICK #2: Make small MAIN MEMORY appear bigger*
  - Technique: VIRTUAL MEMORY

The Cache Idea

Program-Transparent Memory Hierarchy:

- *Cache contains TEMPORARY COPIES of selected main memory locations…*
  - e.g. Mem[100] = 37



- Two Goals:
  - *Improve the average memory access time*
    - HIT RATIO (α): Fraction of refs found in cache
    - MISS RATIO (1- α): Remaining references
    - avg total access time depends on these parameters

$$t_{ave} = \alpha t_c + (1-\alpha)(t_c + t_m) = t_c + (1-\alpha)t_m$$

$$t_c = \text{time to access cache}$$

$$t_m = \text{time to access main memory}$$

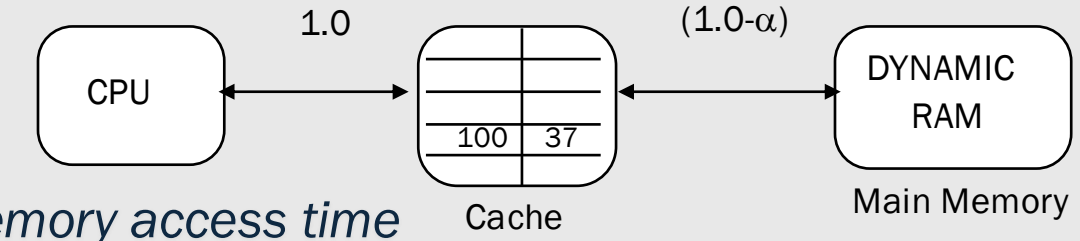
  - *Transparency (compatibility, programming ease)*

Challenge:
To make the hit ratio as high as possible.

The Cache Idea

Program-Transparent Memory Hierarchy:

- *Cache contains TEMPORARY COPIES of selected main memory locations…*
  - e.g. Mem[100] = 37



- Two Goals:
  - *Improve the average memory access time*
    - HIT RATIO (α): Fraction of refs found in cache
    - MISS RATIO (1- α): Remaining references
    - avg total access time depends on these parameters

$$t_{ave} = \alpha t_c + (1-\alpha)(t_c + t_m) = t_c + (1-\alpha)t_m$$

$$t_c = \text{time to access cache}$$

$$t_m = \text{time to access main memory}$$

  - *Transparency (compatibility, programming ease)*

Challenge:
To make the hit ratio as high as possible.

How High of a Hit Ratio?

- Suppose we can easily build a cache with a 0.8 ns access time, but the fastest RAM we can buy for main memory has access time of 10 ns. How high of a hit rate do we need to sustain an average total access time of 1 ns?

$$t_{ave} = \alpha t_c + (1-\alpha)(t_c + t_m) = t_c + (1-\alpha)t_m$$

t_c = time to access cache

t_m = time to access main memory

How High of a Hit Ratio?

- Suppose we can easily build a cache with a 0.8 ns access time, but the fastest RAM we can buy for main memory has access time of 10 ns. How high of a hit rate do we need to sustain an average total access time of 1 ns?

$$\alpha = 1 - \frac{t_{ave} - t_c}{t_m} = 1 - \frac{1 - 0.8}{10} = 98\%$$

Wow, a cache really needs to be good!

Cache

- Sits between CPU and main memory
- Basically a fast table that stores a TAG and DATA
  - *TAG is the memory address*
  - *DATA is a copy of memory contents at the address given by TAG*

Tag	Data
1000	17
1040	1
1032	97
1008	11

Cache

Main Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Cache Access

- On load (lw) we look in the TAG entries for the address we're loading
  - *Found → a HIT, return the DATA*
  - *Not Found → a MISS, go to memory for the data and put it and the address (TAG) in the cache*

Tag	Data
1000	17
1040	1
1032	97
1008	11

Cache

Main Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Cache Lines

- Usually get more data than requested
  - *a LINE is the unit of memory stored in the cache*
  - *usually much bigger than 1 word, 32 bytes per line is common*
  - *bigger LINE means fewer misses because of spatial locality*
  - *but bigger LINE means longer time on miss*

Tag	Data	
1000	17	23
1040	1	4
1032	97	25
1008	11	5

Cache

Main Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

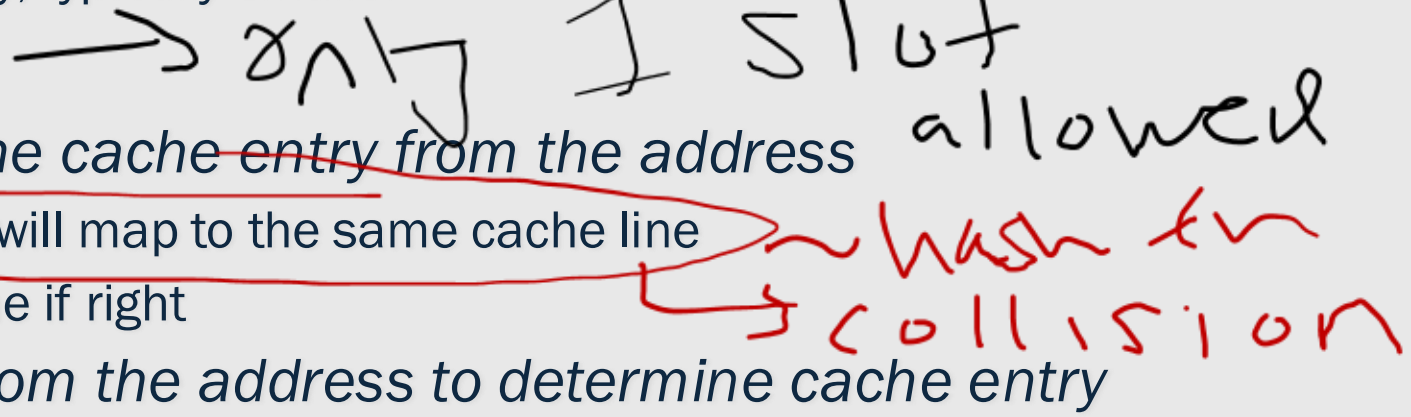
Finding the TAG in the Cache

(addr)

- Fully Associative:
  - *Requested data could be **anywhere** in the cache*
  - *Fully Associative cache uses hardware to compare the address to the tags in parallel but it is expensive*
    - 1 MB is thus unlikely, typically smaller

Finding the TAG in the Cache

- Fully Associative:
  - *Requested data could be **anywhere** in the cache*
  - *Fully Associative cache uses hardware to compare the address to the tags in parallel but it is expensive*
    - 1 MB is thus unlikely, typically smaller
- Direct Mapped Cache:
  - *Directly computes the cache entry from the address*
    - multiple addresses will map to the same cache line
    - use TAG to determine if right
  - *Choose some bits from the address to determine cache entry*
    - low 5 bits determine which byte within the line of 32 bytes
    - we need 15 bits to determine which of the 32k different lines has the data
    - which of the 32 – 5 = 27 remaining bits should we use?



Direct-Mapping Example

Addresses split into 3 parts:

Offset (lowest bits): which byte within the block (here 3 bits for 8-byte blocks)

Index (next bits): which cache line to use (here 2 bits for 4 lines)

Tag (remaining bits): which memory block is actually stored there

- Suppose: 2 words/line, 4 lines, bytes are being read
  - *With 8 byte lines, bottom 3 bits determine byte within line*
  - *With 4 cache lines, next 2 bits determine which line to use*
    - 1024d = 10000**00**000b → line = 00b = 0d
    - 1000d = 01111**10**1000b → line = 01b = 1d
    - 1040d = 10000**10**000b → line = 10b = 2d

Tag | Data

1024	44	99
1000	17	23
1040	1	4
1016	29	38

Cache

4 lines

Main memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Direct Mapping Miss

Addresses split into 3 parts:

Offset (lowest bits): which byte within the block (here 3 bits for 8-byte blocks)

Index (next bits): which cache line to use (here 2 bits for 4 lines)

Tag (remaining bits): which memory block is actually stored there

- What happens when we now ask for address 1008?
  - 1008d = 0111111**10**000b → line = 10b = 2d
  - *...but earlier we put 1040d there*
    - 1040d = 1000001**0**000b → line = 10b = 2d
  - *...so evict 1040d, put 1008d in that entry*

	Tag	Data	
00	1024	44	99
01	1000	17	23
10 →	1008	11	5
11	1016	29	38

Cache

Memory	
1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

Miss Penalty and Rate

- How much time do you lose on a Miss?
  - *MISS PENALTY is the time it takes to read the main memory if data was not found in the cache*
    - 50 to 100 clock cycles is common
  - *MISS RATE is the fraction of accesses which MISS*
  - *HIT RATE is the fraction of accesses which HIT*
  - *MISS RATE + HIT RATE = 1*

Final

Miss Penalty and Rate Example

Suppose a particular cache has a MISS PENALTY of 100 cycles and a HIT RATE of 95%. The CPI for load on HIT is 5 but on a MISS it is 105. What is the average CPI for load?

Avg CPI = $5 * .95 + 105 * .05$

$\Rightarrow 10$

What if MISS PENALTY = 120 cycles?

$= 5 * 0.95 + (120 + 5) * 0.05$

$125 * .05$

$\Rightarrow 11$

Miss Penalty and Rate Example

Suppose a particular cache has a MISS PENALTY of 100 cycles and a HIT RATE of 95%. The CPI for load on HIT is 5 but on a MISS it is 105. What is the average CPI for load?

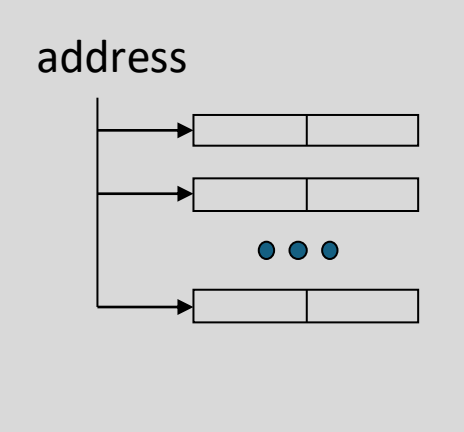
*Average CPI = 5 * 0.95 + 105 * 0.05 = 10*

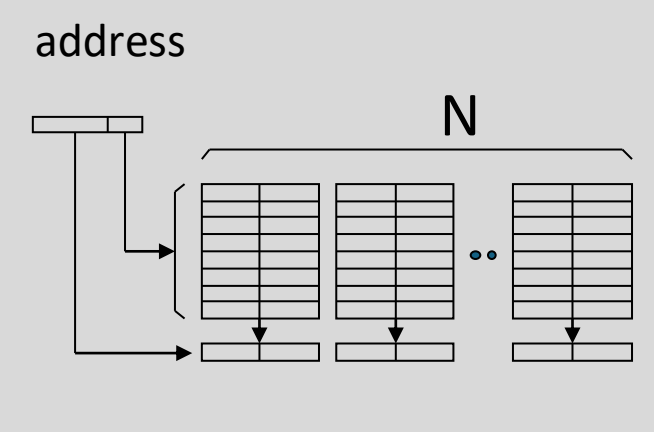
What if MISS PENALTY = 120 cycles?

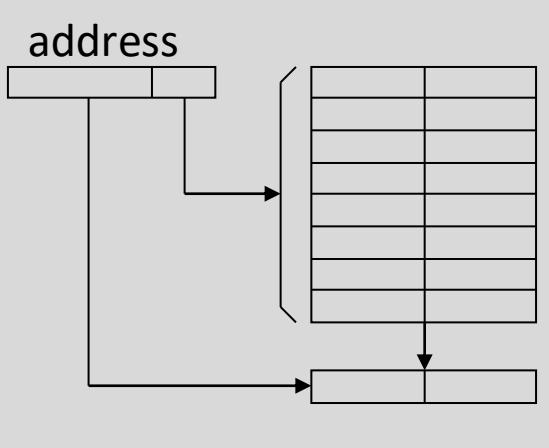
*Average CPI = 5 * 0.95 + (120+5) * 0.05 = 11*

Continuum of Associativity

Fully associative | N-way set associative | Direct-mapped







Compares addr with ALL tags simultaneously location A can be stored in any cache line.

Compares addr with N tags simultaneously. Data can be stored in any of the N cache lines belonging to a "set" like N direct-mapped caches.

Compare addr with only ONE tag. Location A can be stored in exactly one cache line.

What happens on a MISS?