EECS151 : Introduction to Digital Design and ICs

Lecture 24 – Decoders

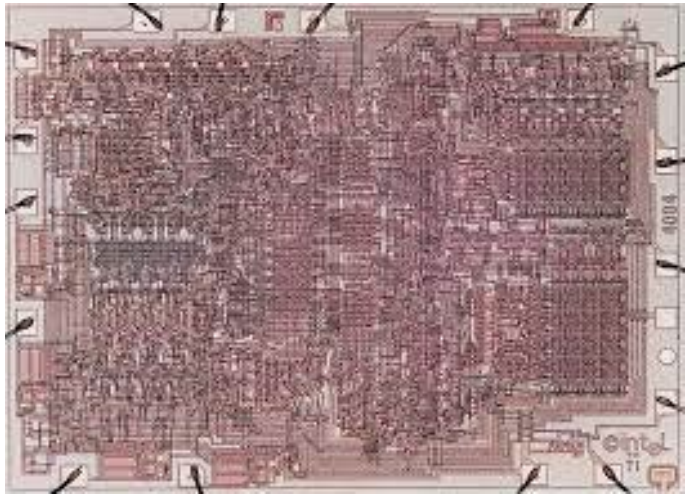
Bora Nikolić

Happy 50th birthday Intel 4004!

The **Intel 4004** is a 4-bit central processing unit (CPU) released by Intel Corporation in 1971. Sold for US$60, it was the first commercially produced microprocessor, and the first in a long line of Intel CPUs. Launched November 15, 1971.

www.wikipedia.org

https://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html



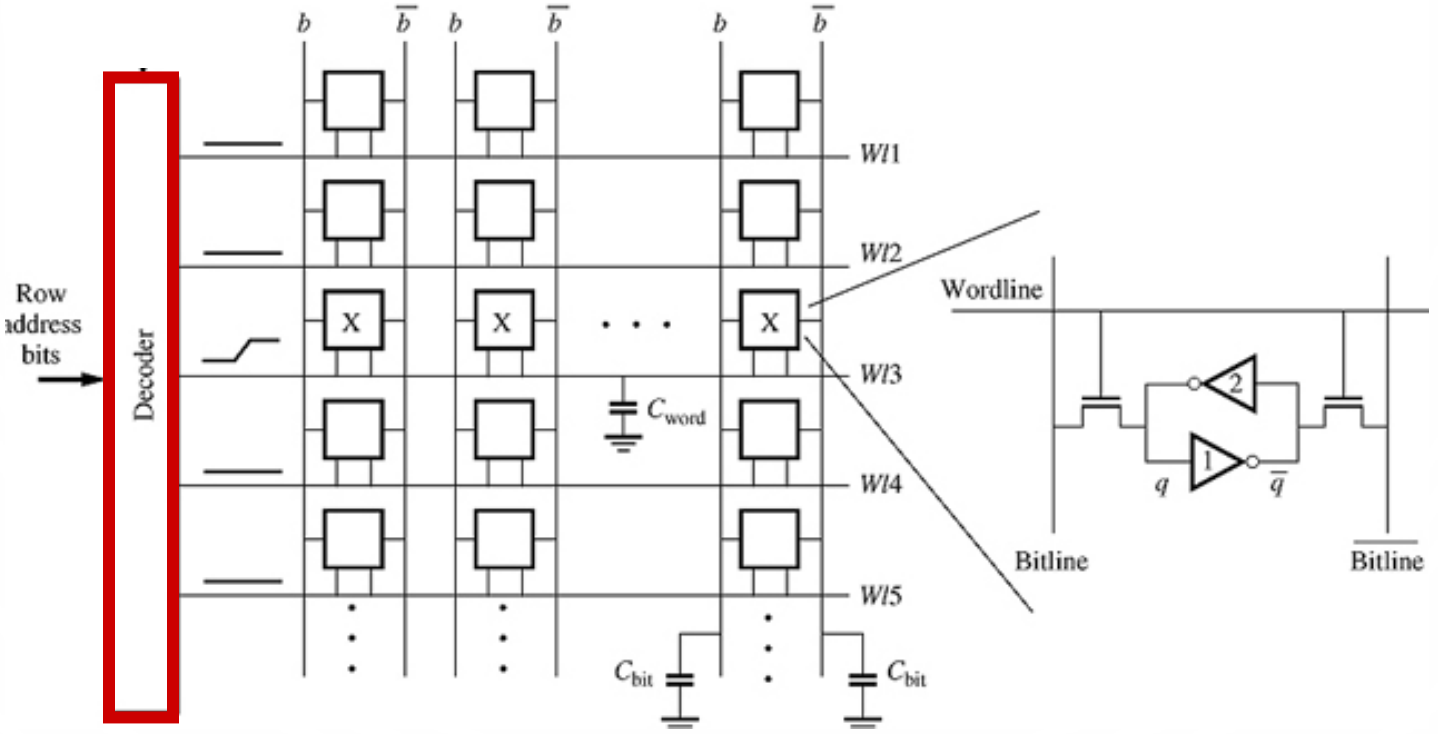
Review

- Dense memories are built as arrays of memory elements
  - SRAM is a static memory
- SRAM has unique combination of density, speed, power
- SRAM cells sized for stability and writeability
- SRAM and regfile cells can have multiple R/W ports

Memory Decoders

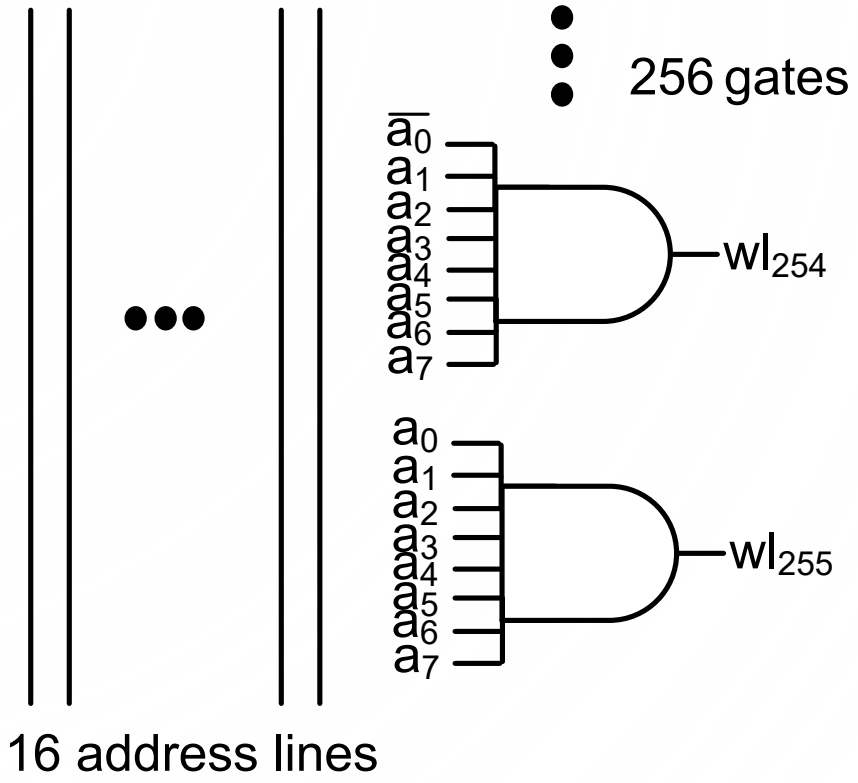
Decoder Design Example

- Look at decoder for 256x256 memory block (8KBytes)



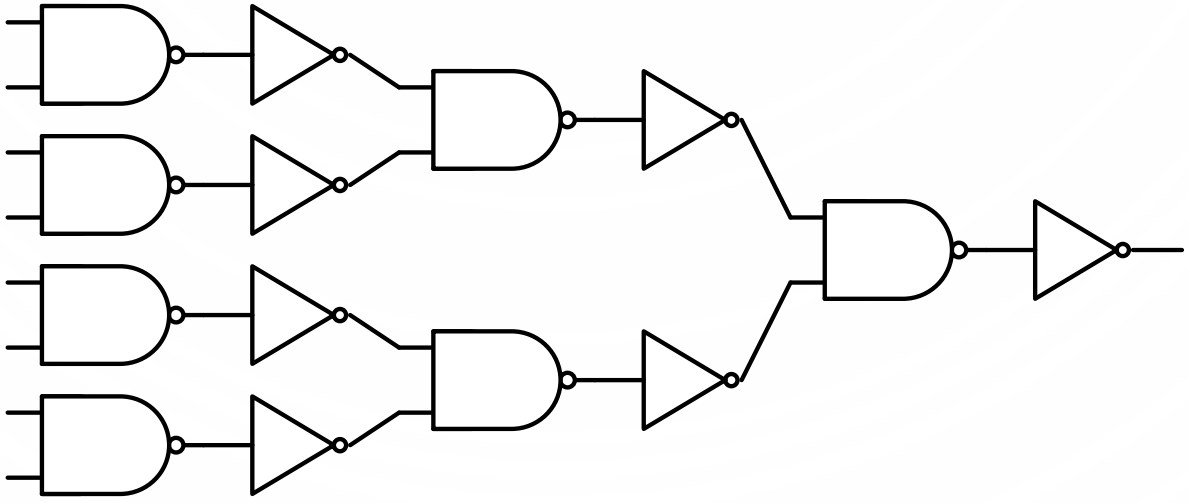
Possible Decoder

- 256 8-input AND gates
  - Each built out of a tree of NAND gates and inverters
- Need to drive a lot of capacitance (SRAM cells and wordline wire)
  - What's the best way to do this?



Possible AND8

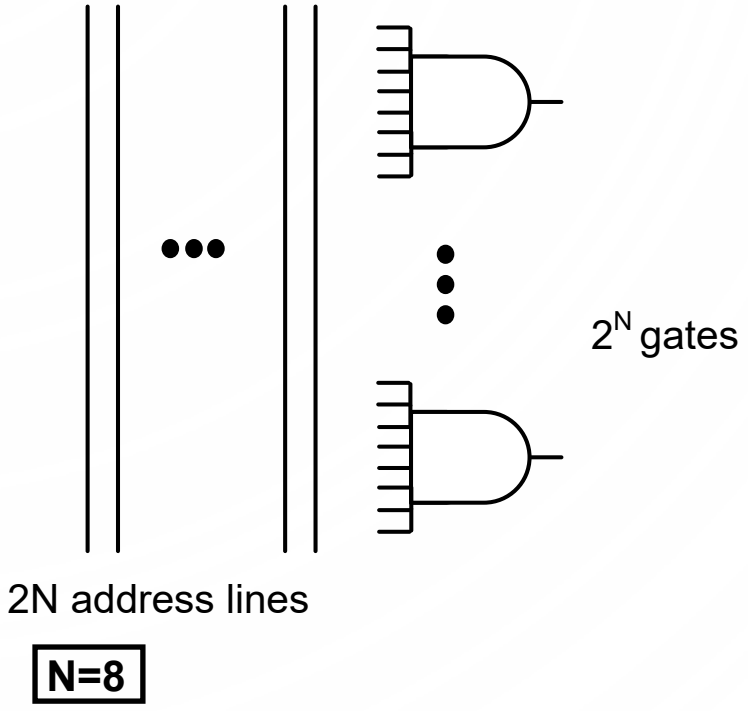
- Build 8-input NAND gate using 2-input gates and inverters
- Is this the best we can do?
- Is this better than using fewer NAND4 gates?



Problem Setup

- Goal: Build fastest possible decoder with static CMOS logic

- What we know
  - Basically need 256 AND gates, each one of them drives one word line



Problem Setup (1)

- Each wordline has 256 cells connected to it
- $C_{WL} = 256*C_{cell} + C_{wire}$
  - Ignore wire for now
- Assume that decoder input capacitance is $C_{address}=4*C_{cell}$

Problem Setup (2)

- Each address bit drives $2^8/2$ AND gates
  - A0 drives ½ of the gates, A0_b the other ½ of the gates

- Total fanout on each address wire is:

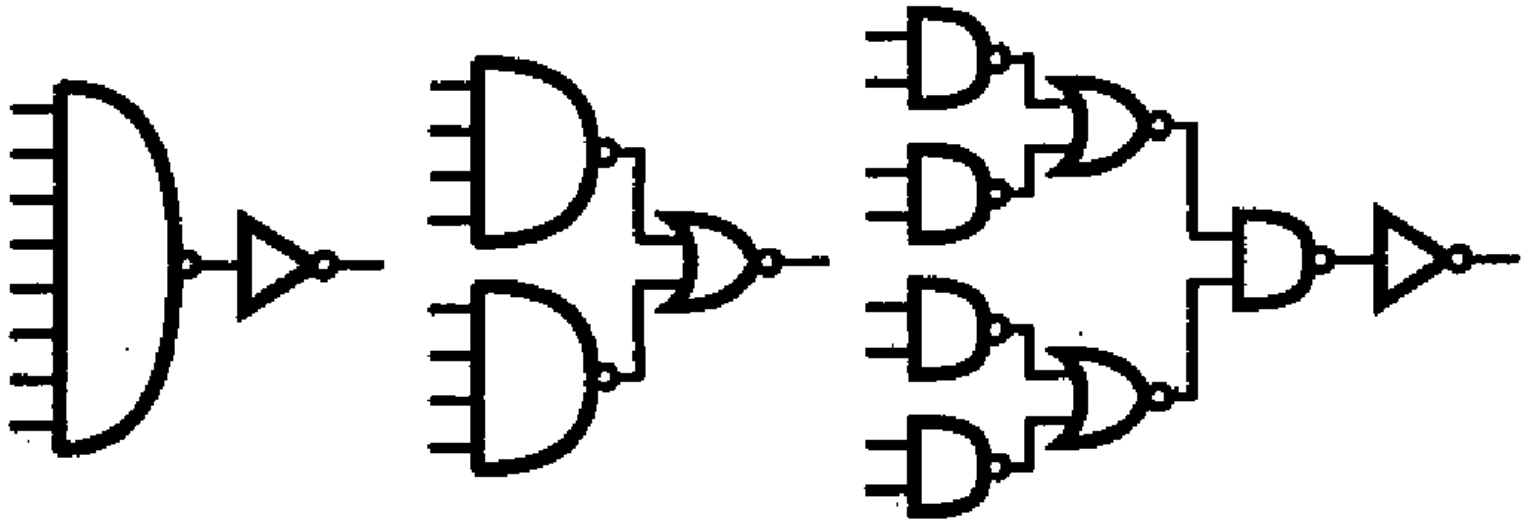
$$F = \Pi B \frac{C_{load}}{C_{in}} = 128 \frac{(256 C_{cell})}{4 C_{cell}} = 2^7 \frac{(2^8 C_{cell})}{2^2 C_{cell}} = 2^{13}$$

Decoder Fan-Out

- F of 2^{13} means that we will want to use more than $\log_4(2^{13}) = 6.5$ stages to implement the AND8

- Need many stages anyways
  - So what is the best way to implement the AND gate?
  - Will see next that it's the one with the most stages and least complicated gates

8-Input AND



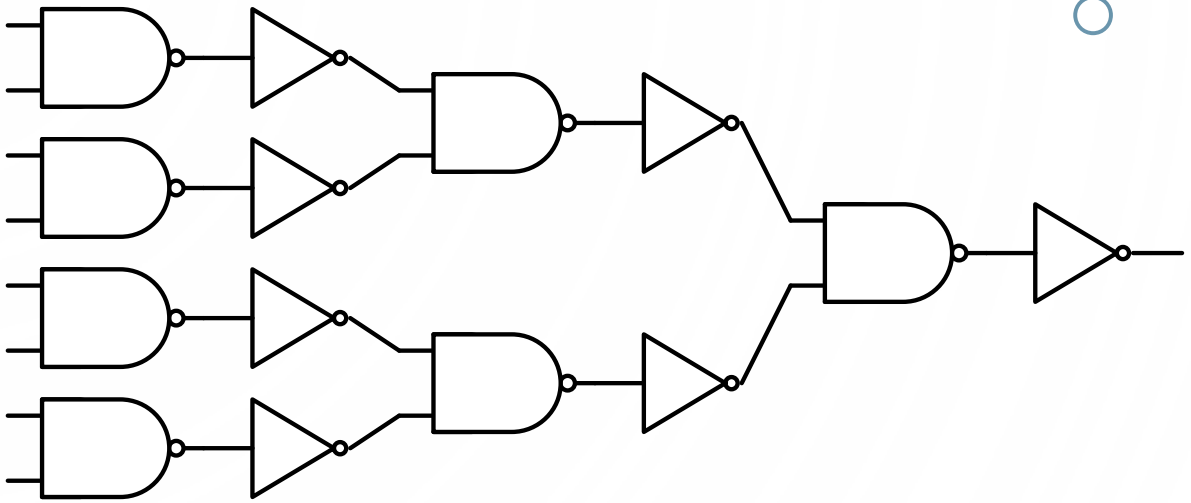
g : 9/2 1
G = 9/2
P = 8 + 1

g : 5/2 3/2
G = 15/4
P = 4 + 2

g : 3/2 3/2 3/2 1
G = 27/8
P = 2 + 2 + 2 + 1

8-Input AND

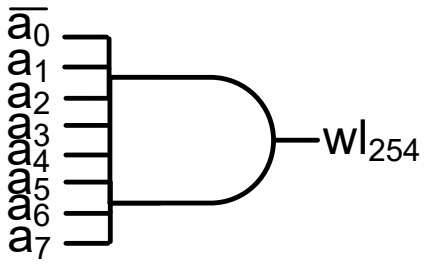
- Using 2-input NAND gates
  - 8-input gate takes 6 stages
- Total G is $(3/2)^3 \approx 3.4$
- So H is $3.4*2^{13}$ – optimal N of ~7.4

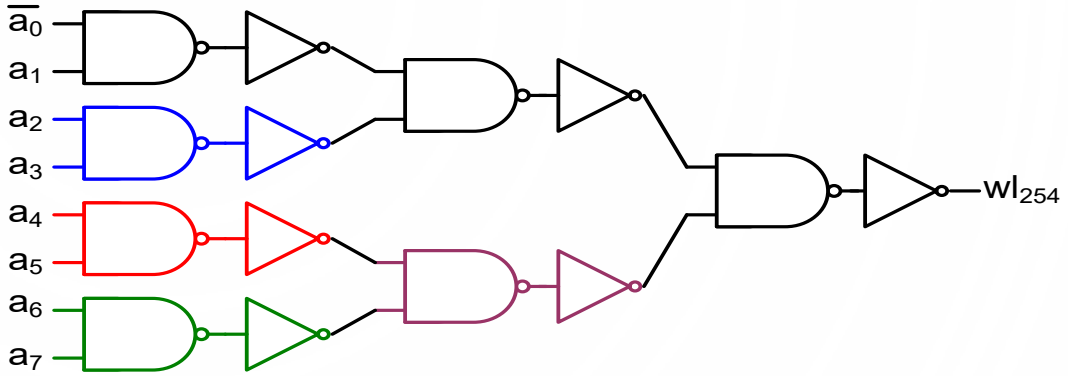


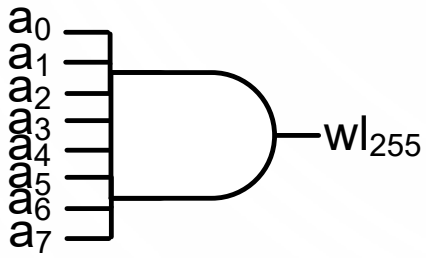
Decoder So Far

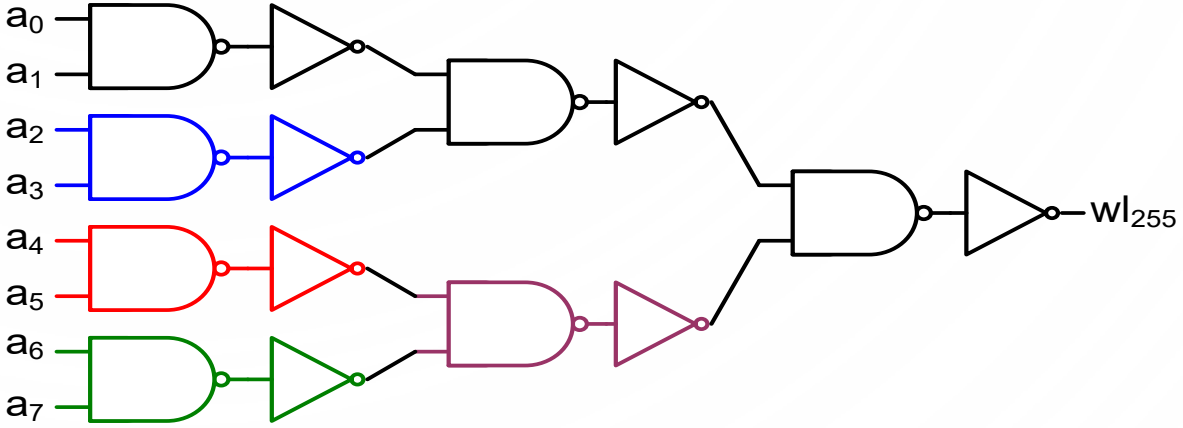
- 256 8-input AND gates
  - Each built out of tree of NAND gates and inverters
- Issue:
  - Every address line has to drive 128 gates (and wire) right away
  - Forces us to add buffers just to drive address inputs

Look Inside Each AND8 Gate





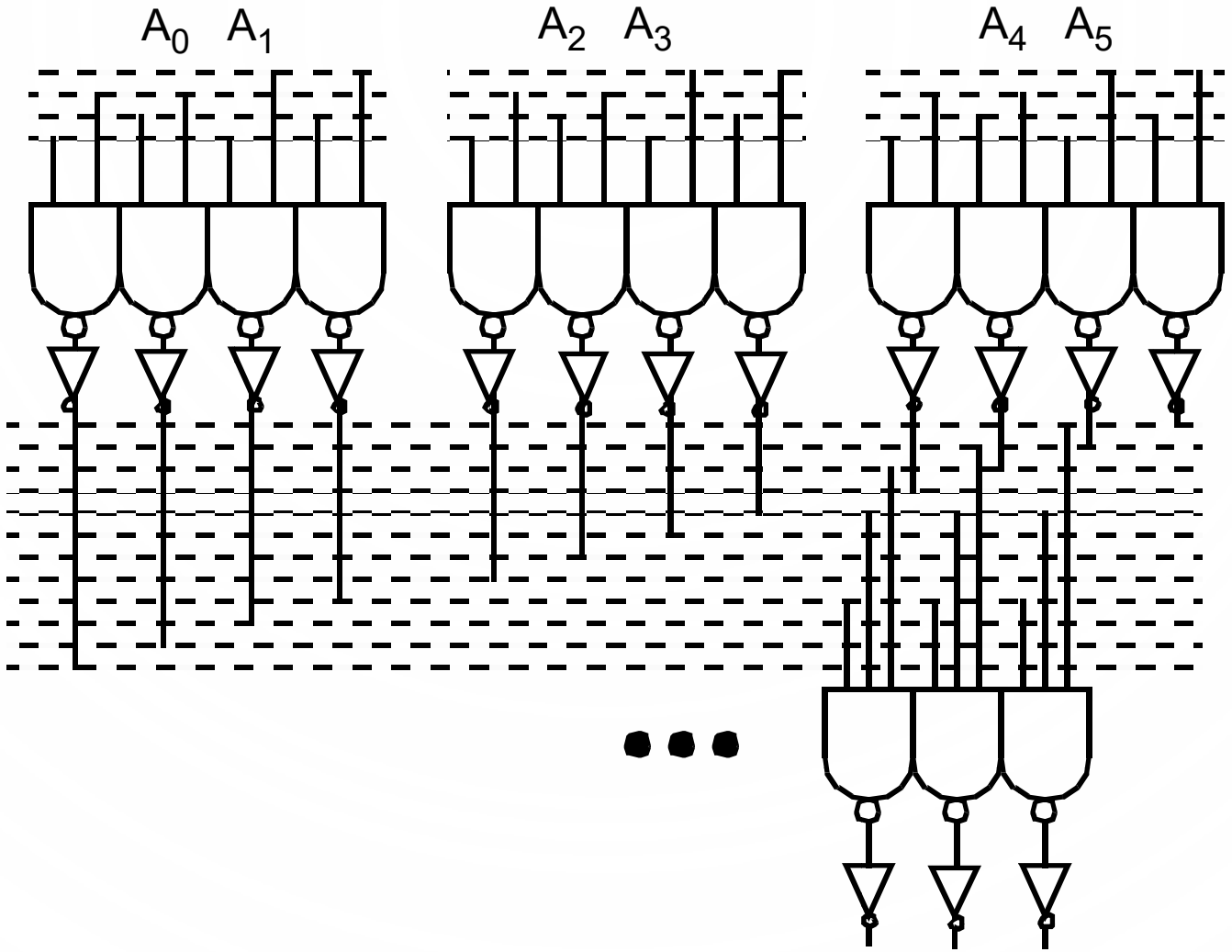




Predecoders

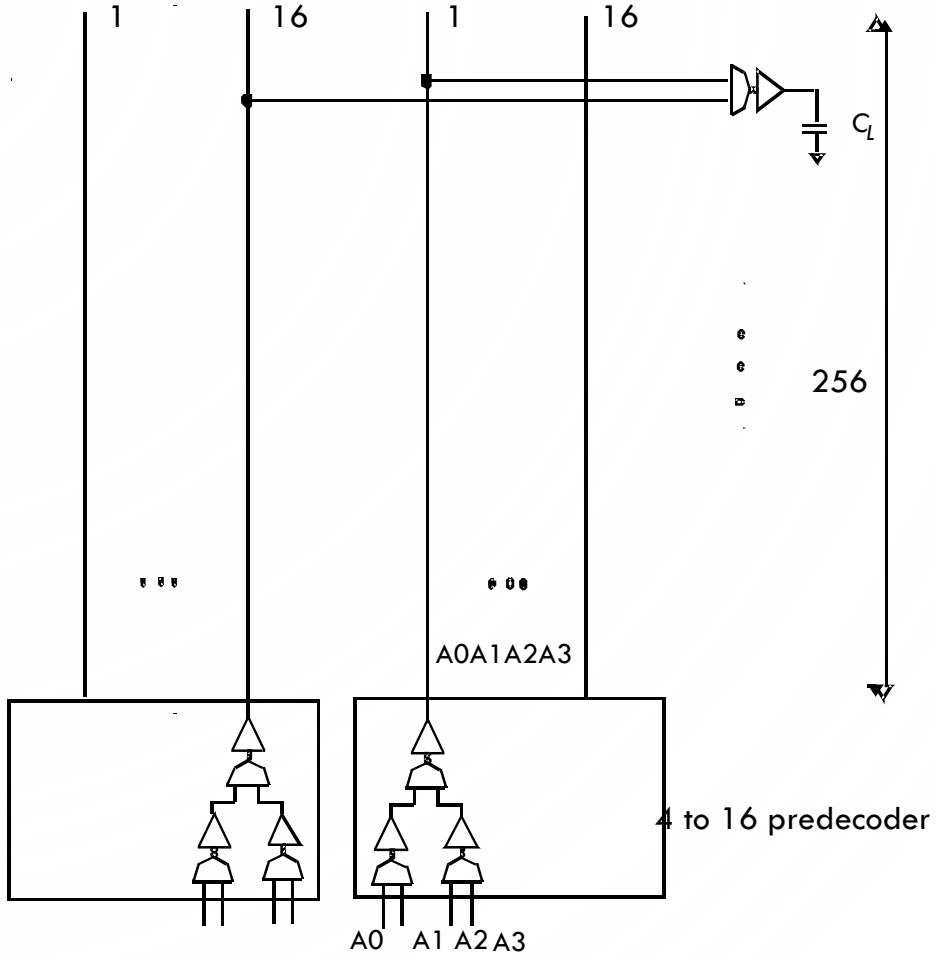
- Use a single gate for each of the shared terms
  - E.g., from A_0, $\overline{A_0}$, A_1, and $\overline{A_1}$, generate four signals: A_0A_1, $\overline{A_0}A_1$, $A_0\overline{A_1}$, $\overline{A_0}\overline{A_1}$
- In other words, we are decoding smaller groups of address bits first
  - And using the "predecoded" outputs to do the rest of the decoding

Predecoder and Decoder



Predecode Options

- Larger predecode usually better:
- More stages before the long wires
  - Decreases their effect on the circuit
- Fewer number of long wires switches
  - Lower power
- Easier to fit 2-input gate into cell pitch

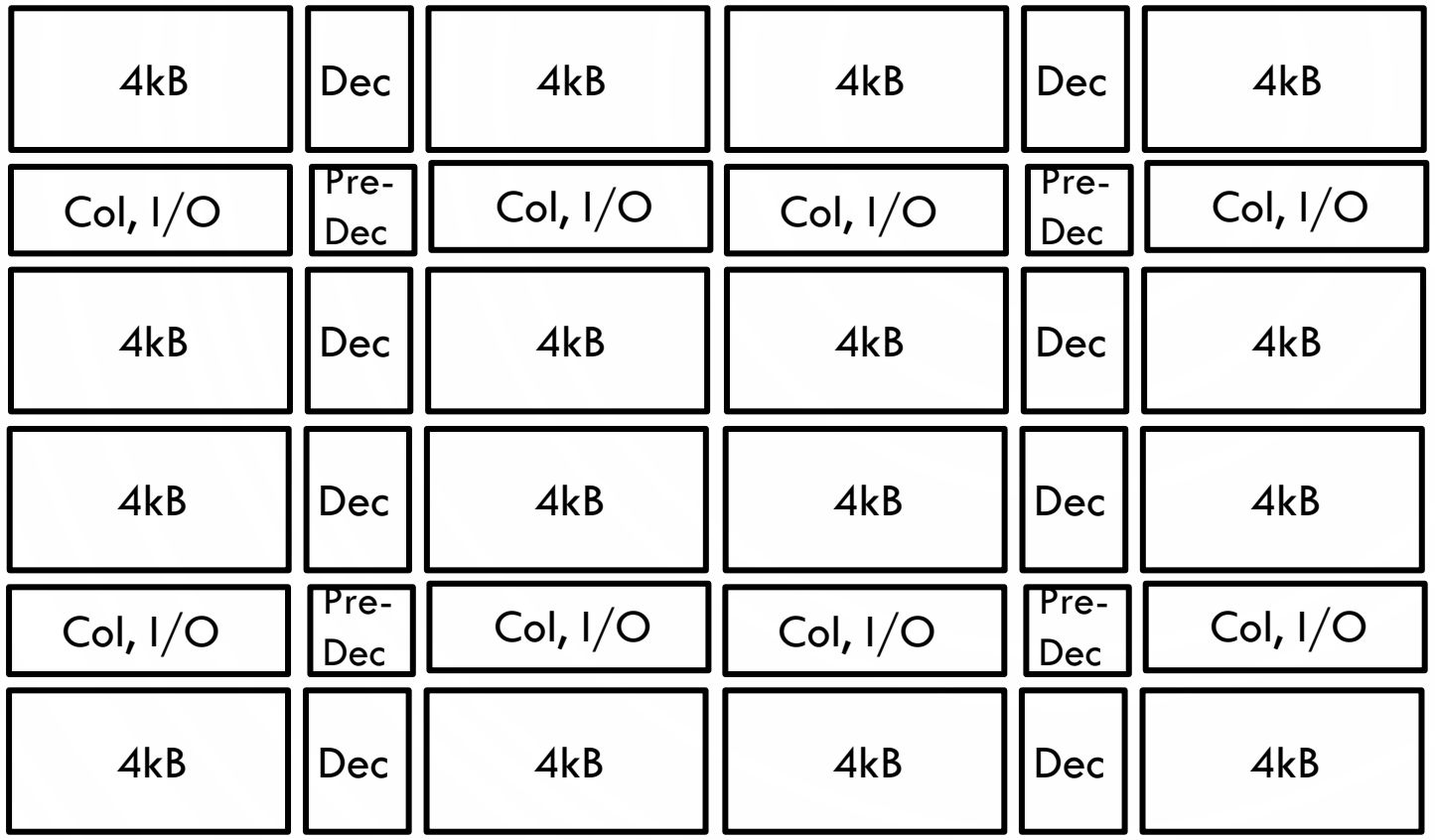


Administrivia

- Homework 10 posted on Friday, due 11/22
  - No homework during Thanksgiving
- Project checkpoints #2/#3 this week
- Next week:
  - Class lab and discussion on Monday
  - No classes/labs/discussions Tu and We

Building Larger Arrays

Building Larger Custom Arrays

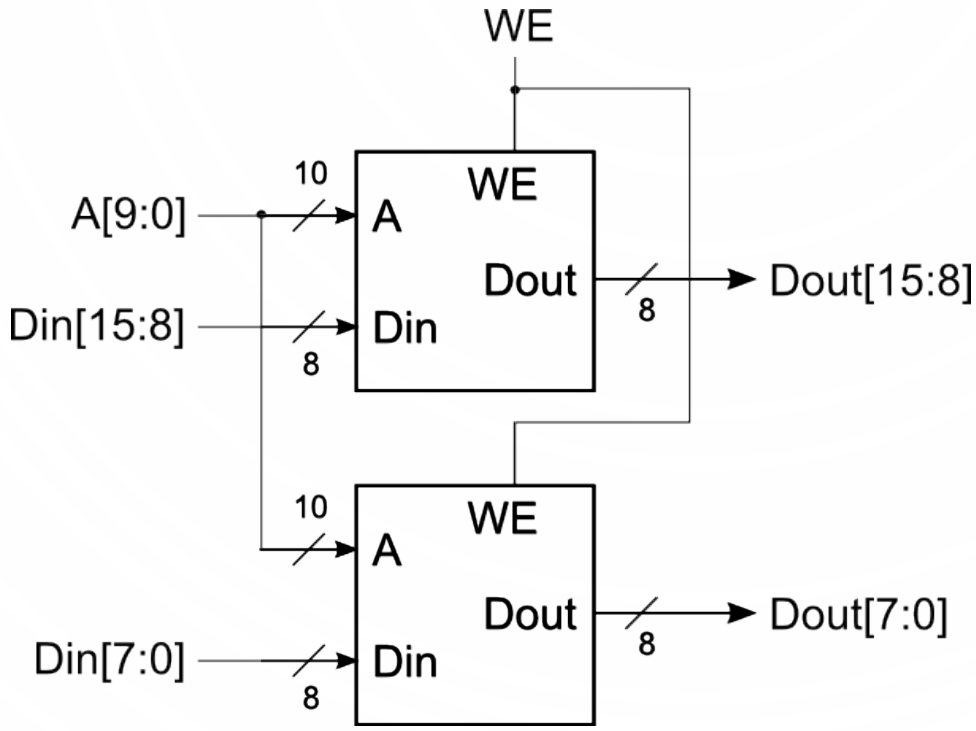


- Each subarray is 2-8kB
- Hierarchical decoding
- Peripheral overhead is 30-50%
- Delay is wire dominated
- Scratchpads, caches, TLBs

Cascading Memory-Blocks

How to make larger memory blocks out of smaller ones.

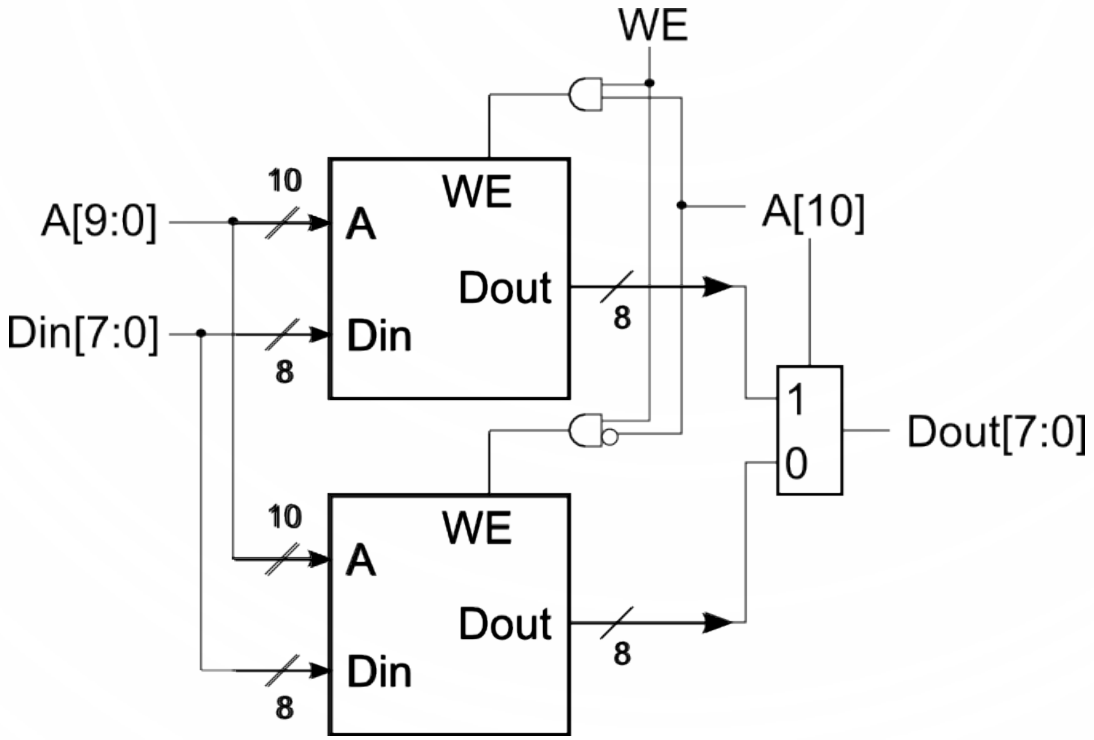
Increasing the width. Example: given 1Kx8, want 1Kx16



Cascading Memory-Blocks

How to make larger memory blocks out of smaller ones.

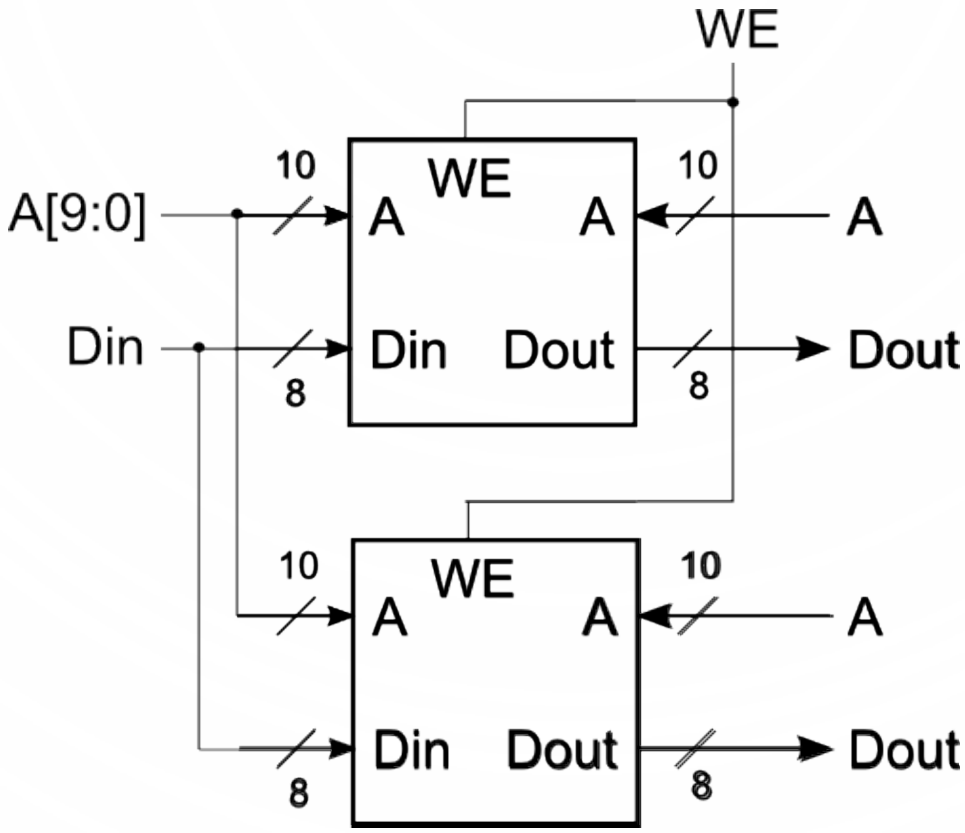
Increasing the depth. Example: given 1Kx8, want 2Kx8



Adding Ports to Primitive Memory Blocks

Adding a read port to a simple dual port (SDP) memory.

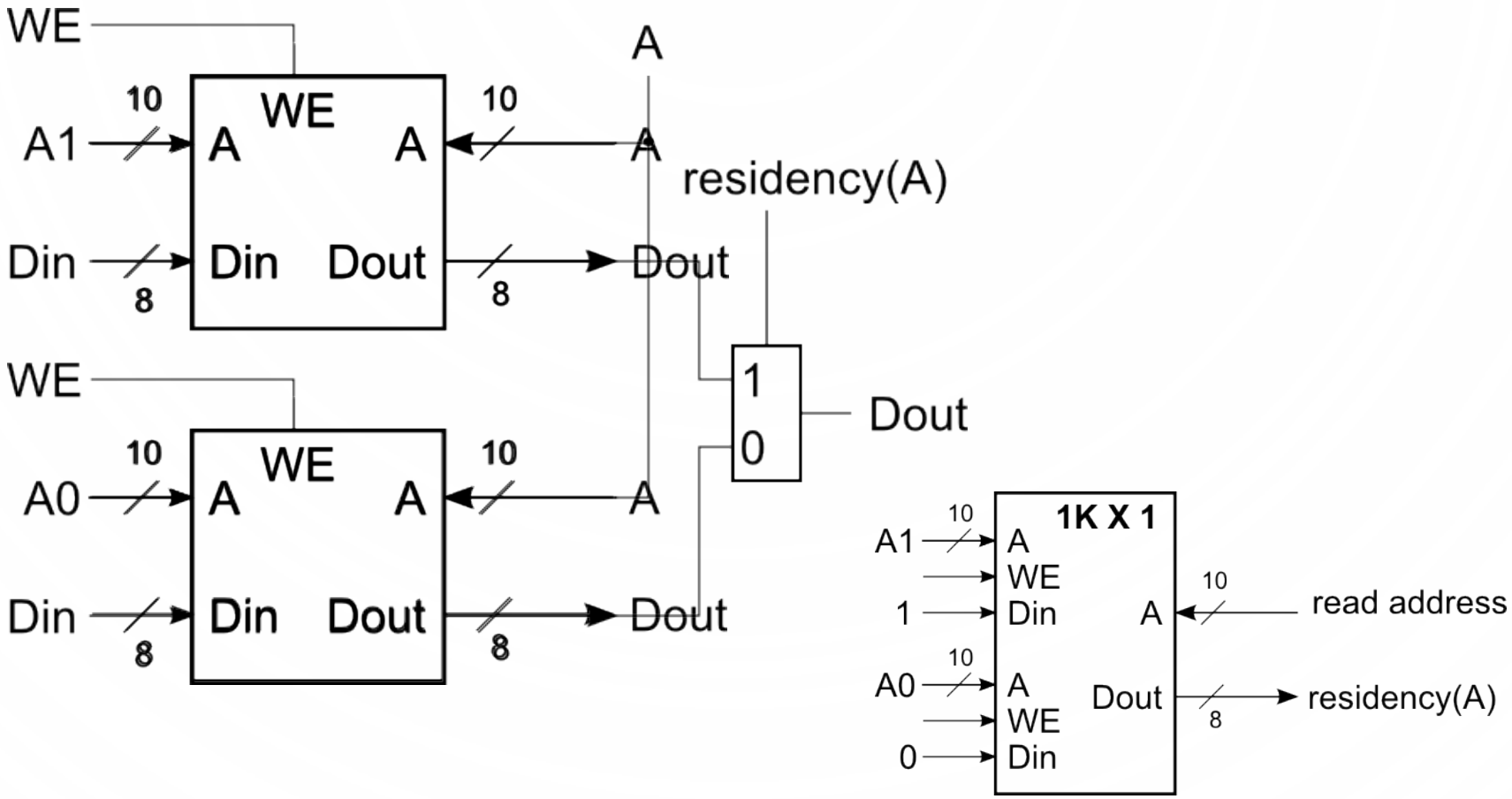
Example: given 1Kx8 SDP, want 1 write & 2 read ports.



Adding Ports to Primitive Memory Blocks

How to add a write port to a simple dual-port memory.

Example: given 1Kx8 SDP, want 1 read & 2 write ports.



Caches

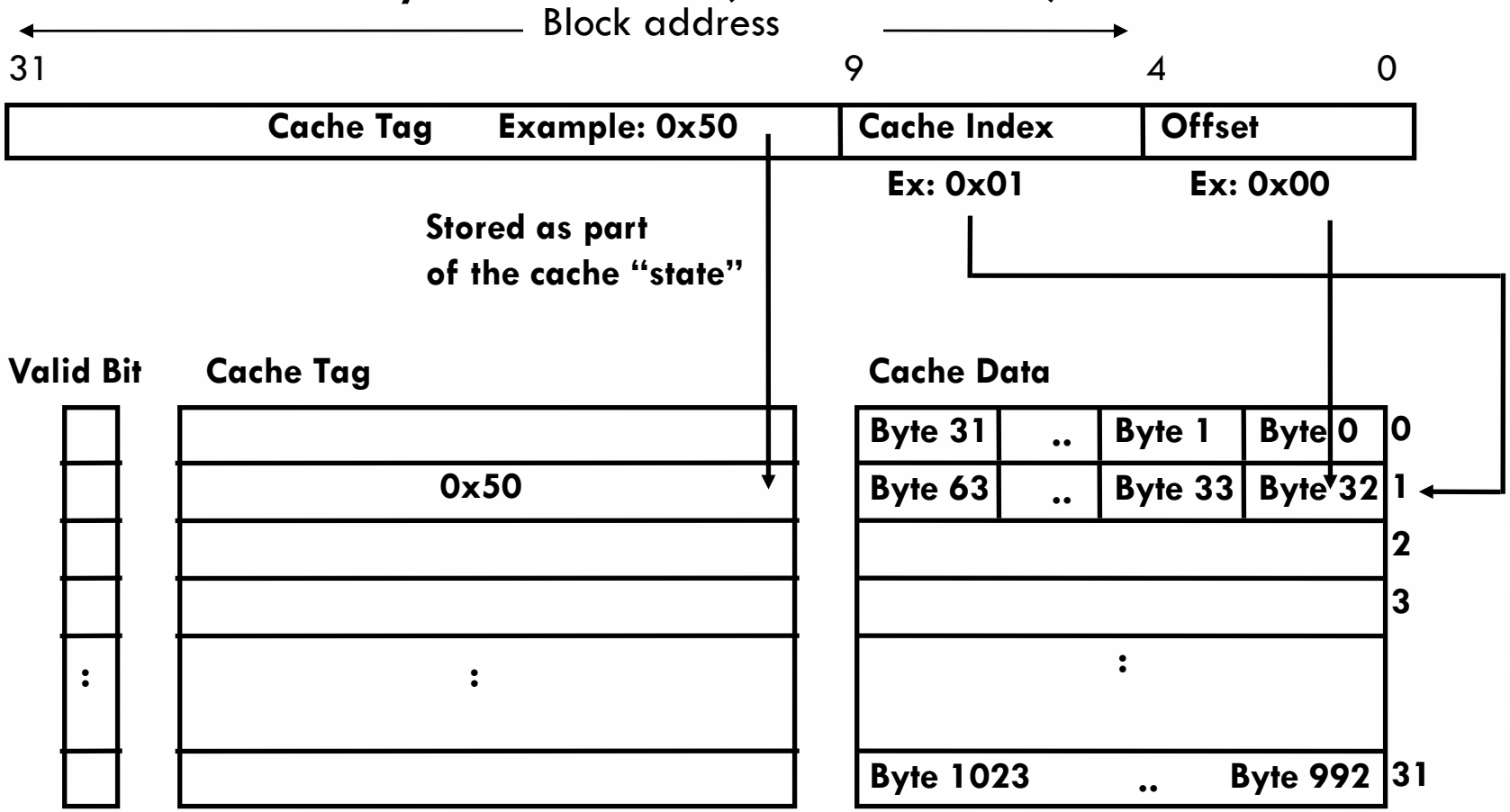
Caches (Review from 61C)

- **Two Different Types of Locality:**
  - **Temporal locality (Locality in time): If an item is referenced, it tends to be referenced again soon.**
  - **Spatial locality (Locality in space): If an item is referenced, items whose addresses are close by tend to be referenced soon.**
- **By taking advantage of the principle of locality:**
  - **Present the user with as much memory as is available in the cheapest technology.**
  - **Provide access at the speed offered by the fastest technology.**
- **DRAM is slow but cheap and dense:**
  - **Good choice for presenting the user with a BIG memory system**
- **SRAM is fast but expensive and not as dense:**
  - **Good choice for providing the user FAST access time.**

Example: 1 KB Direct Mapped Cache with 32-B Blocks

For a 2^N-byte cache:

- **The uppermost (32 - N) bits are always the Cache Tag**
- **The lowest M bits are the byte-select offset (Block Size = 2^M)**

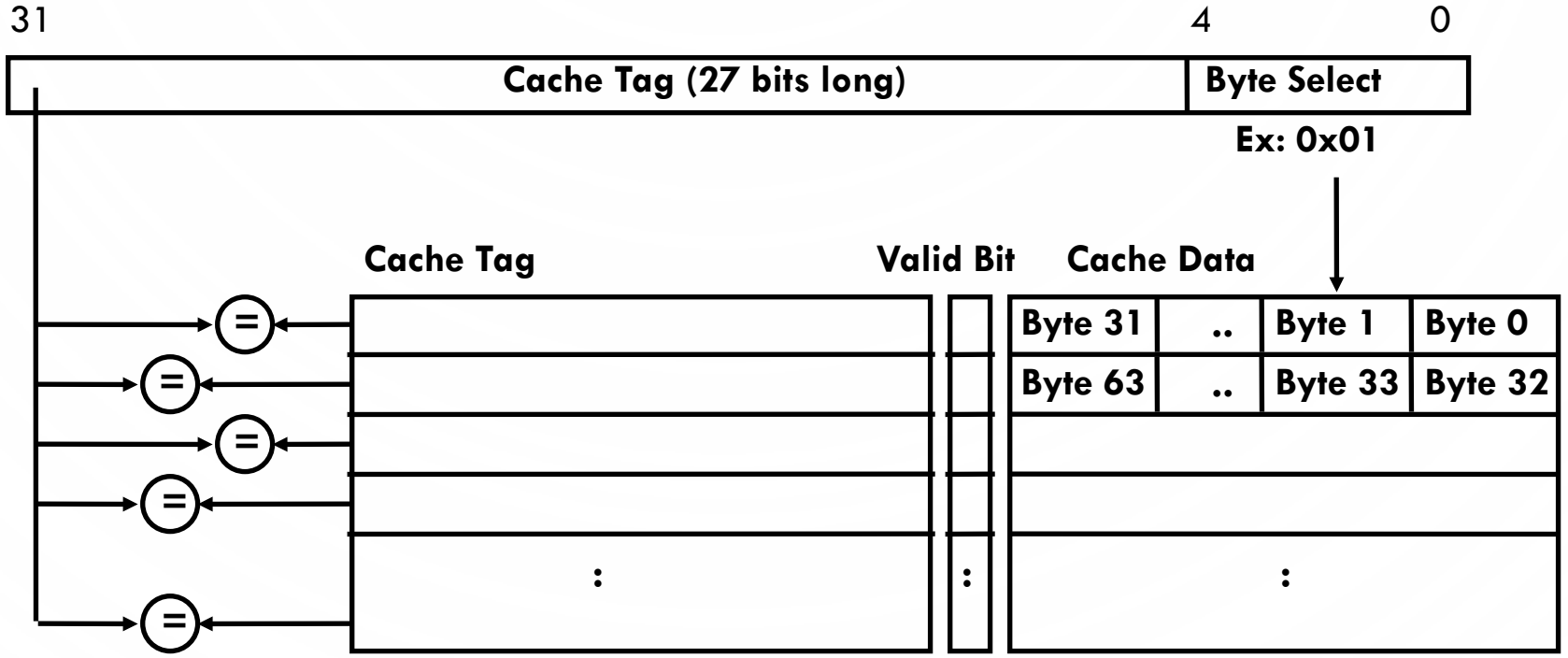


Fully Associative Cache

Fully Associative Cache

- Ignore cache Index for now
- Compare the Cache Tags of all cache entries in parallel (expensive…)
- Example: Block Size = 32 B blocks, we need N 27-bit comparators

By definition: Conflict Miss = 0 for a fully associative cache



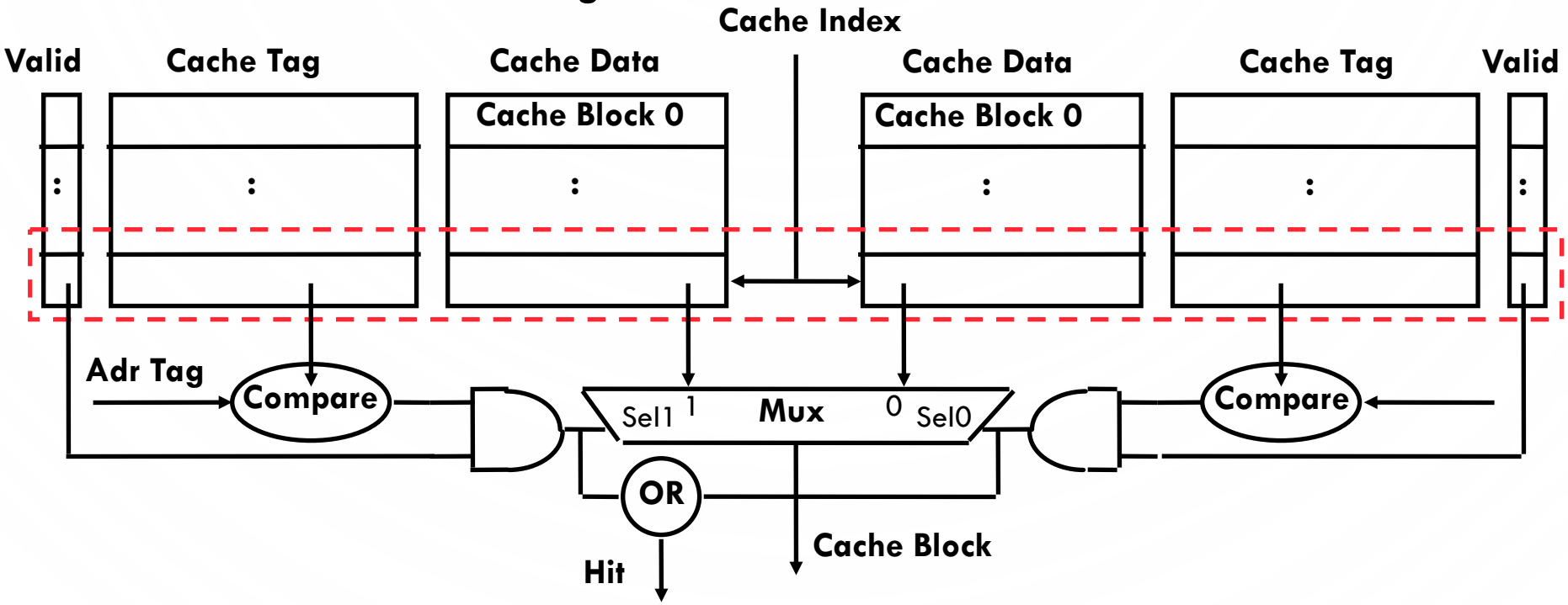
Set Associative Cache

N-way set associative: N entries for each Cache Index

- N direct mapped caches operate in parallel

Example: Two-way set associative cache

- Cache Index selects a "set" from the cache
- The two tags in the set are compared to the input in parallel
- Data is selected based on the tag result



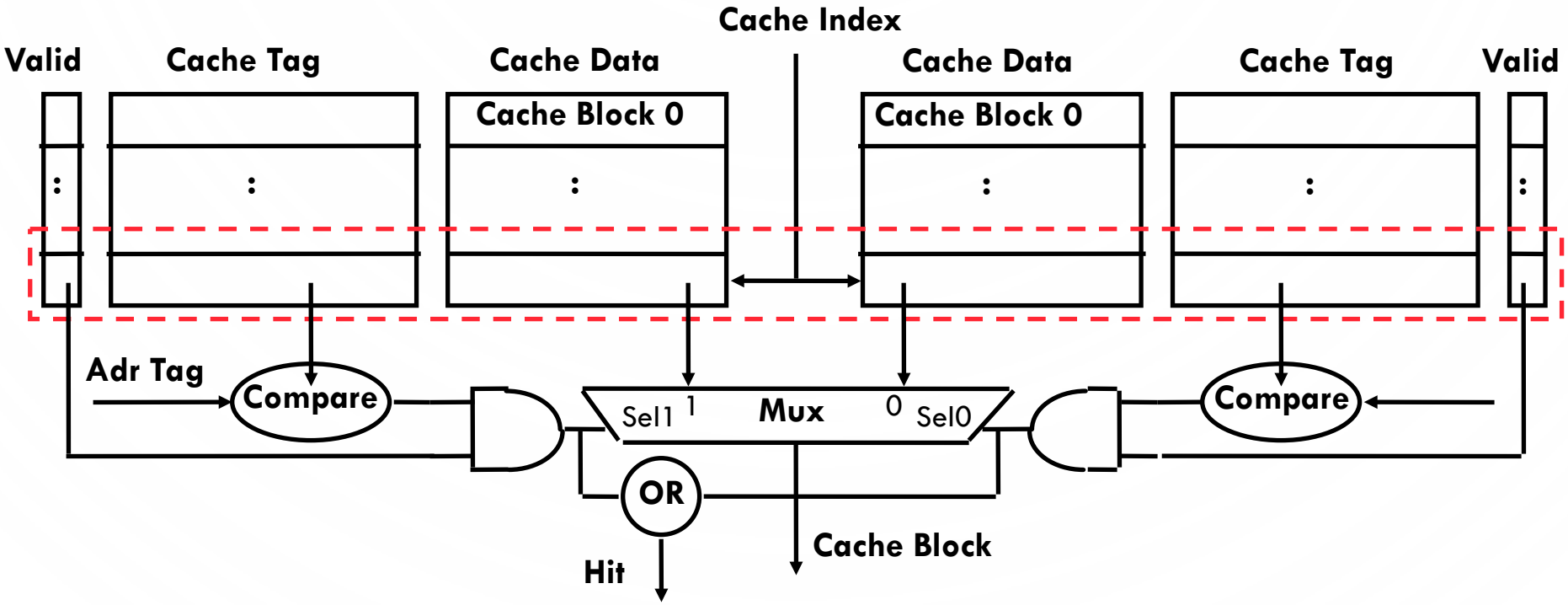
Disadvantage of Set Associative Cache

N-way Set Associative Cache versus Direct Mapped Cache:

- **N comparators vs. 1**
- **Extra MUX delay for the data**
- **Data comes AFTER Hit/Miss decision and set selection**

In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:

- **Possible to assume a hit and continue. Recover later if miss.**



Block Replacement Policy

- Direct-Mapped Cache
  - index completely specifies position which position a block can go in on a miss
- N-Way Set Assoc
  - index specifies a set, but block can occupy any position within the set on a miss
- Fully Associative
  - block can be written into any position
- Question: if we have the choice, where should we write an incoming block?
  - If there's a valid bit off, write new block into first invalid.
  - If all are valid, pick a replacement policy
    - rule for which block gets "cached out" on a miss.

Block Replacement Policy: LRU

- LRU (Least Recently Used)
  - Idea: cache out block which has been accessed (read or write) least recently
  - Pro: temporal locality ➔ recent past use implies likely future use: in fact, this is a very effective policy
  - Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires more complicated hardware and more time to keep track of this

Summary

- Memory decoding is done hierarchically
  - Wire-limited in large arrays
- Multiple cache levels make memory appear both fast and big
- Direct mapped and set-associative cache