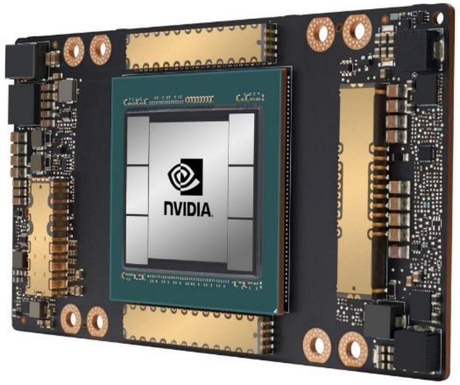
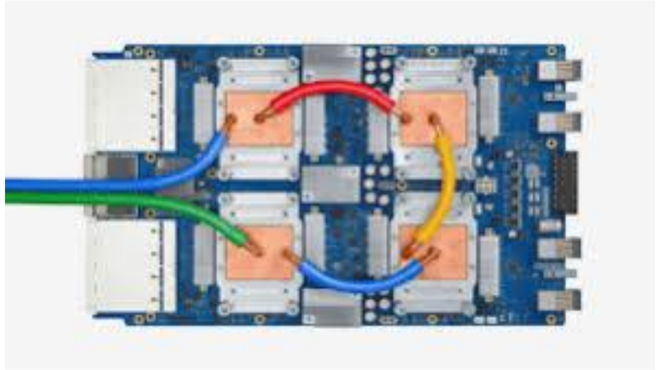
CSCB58: Computer Organization



Prof. Gennady Pekhimenko

University of Toronto

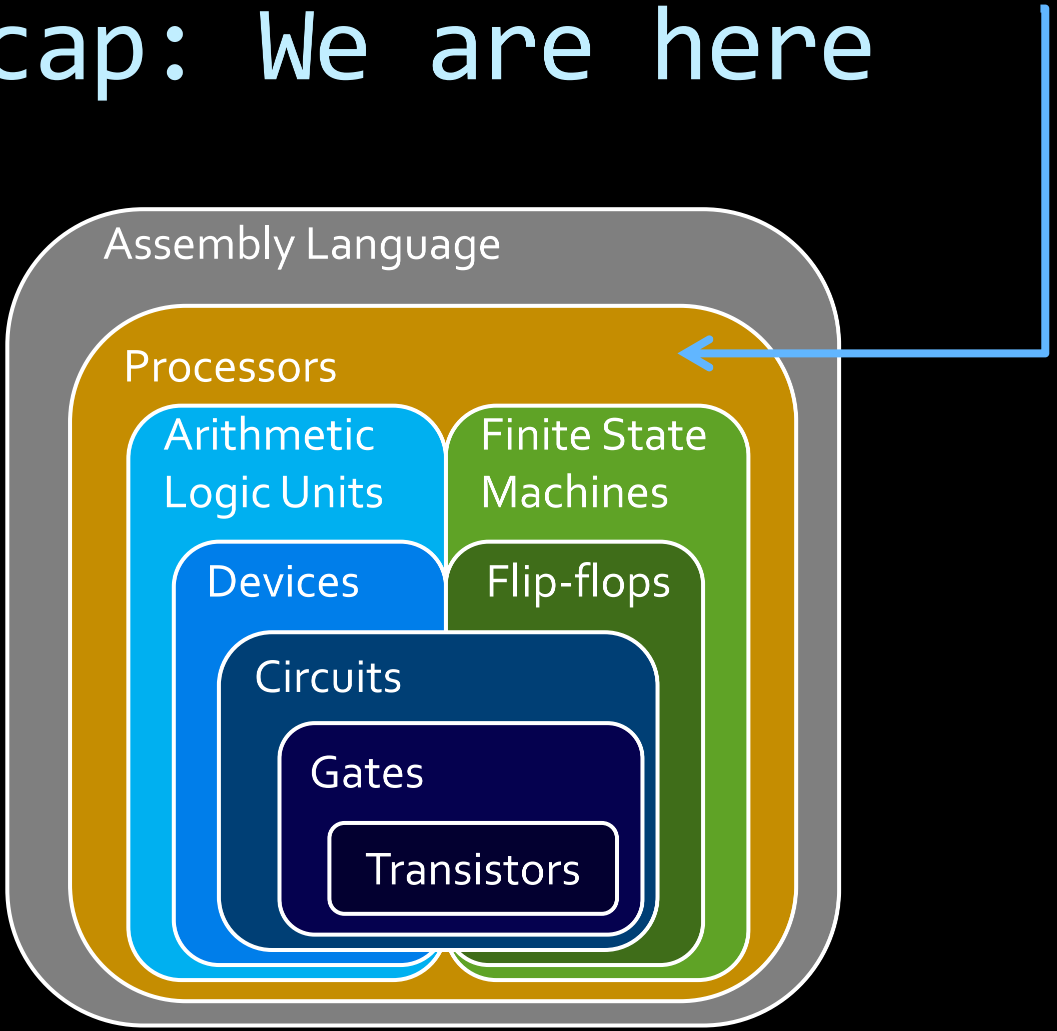
Fall 2020

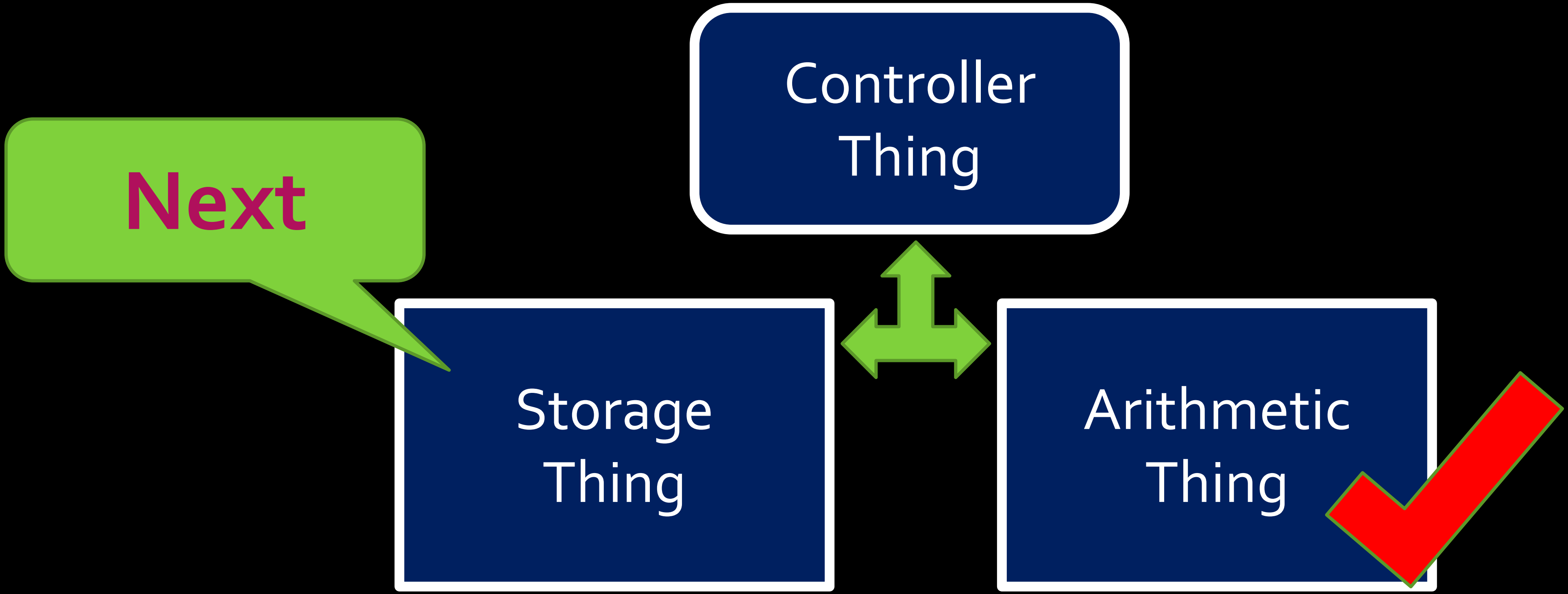


The content of this lecture is adapted from the lectures of Larry Zheng and Steve Engels

CSCB58 Week 8

Recap: We are here





Next

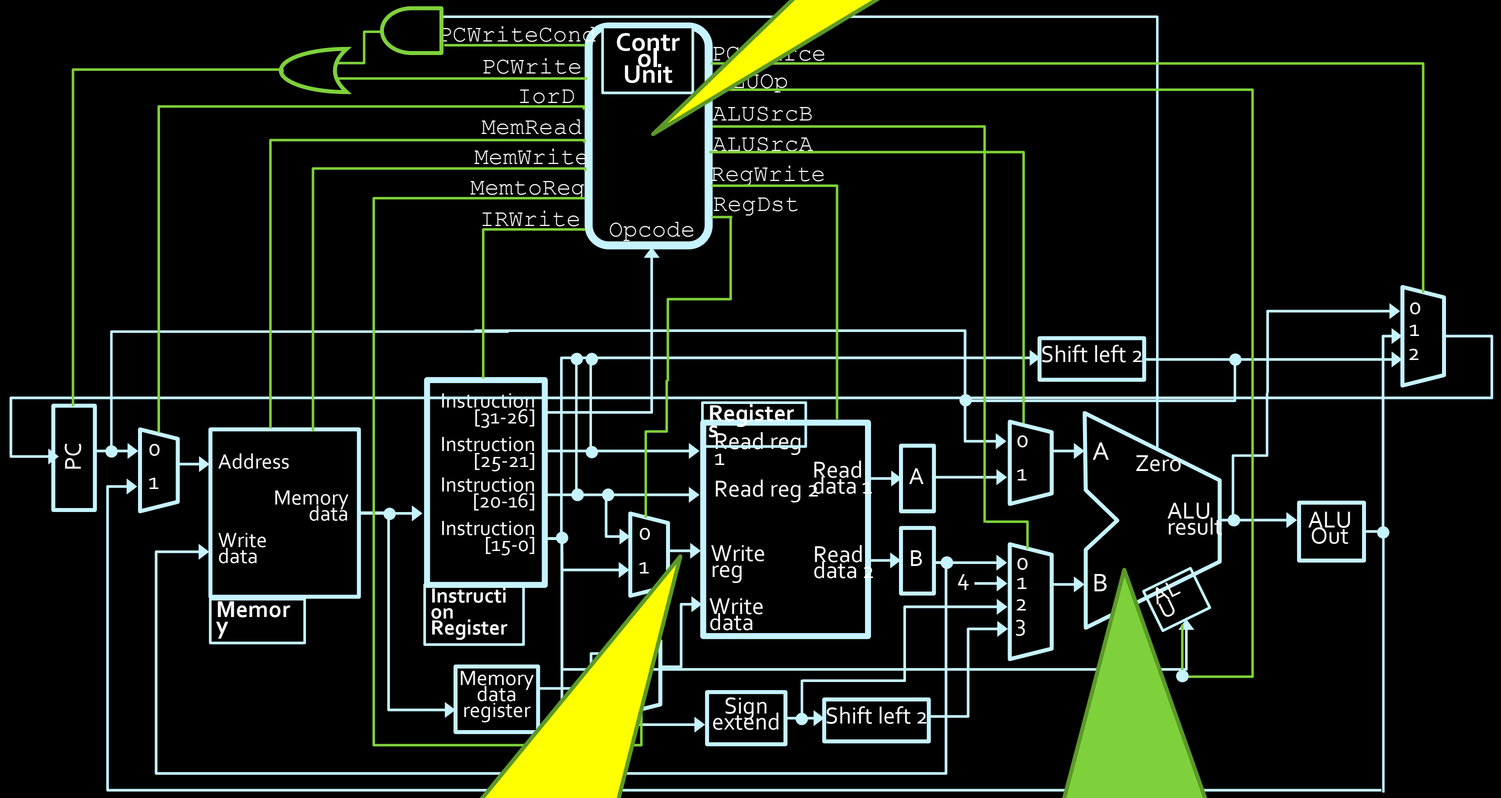
Controller Thing

Storage Thing

Arithmetic Thing

The Blueprint of a microprocessor

The Controller Thing

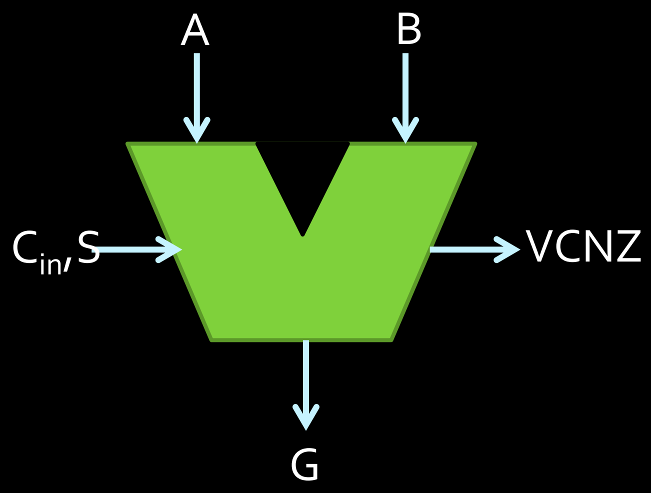


The Storage Thing

The Arithmetic Thing

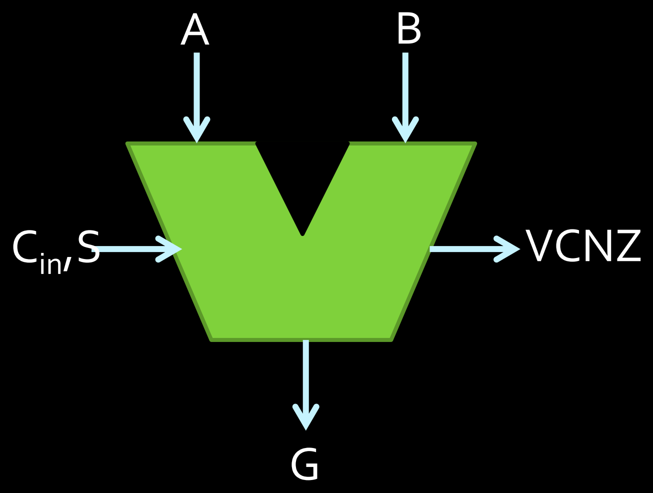
We've learned the arithmetic thing

ALU



With ALU, we can do addition, subtraction, logic operations, etc.

So this is the ALU

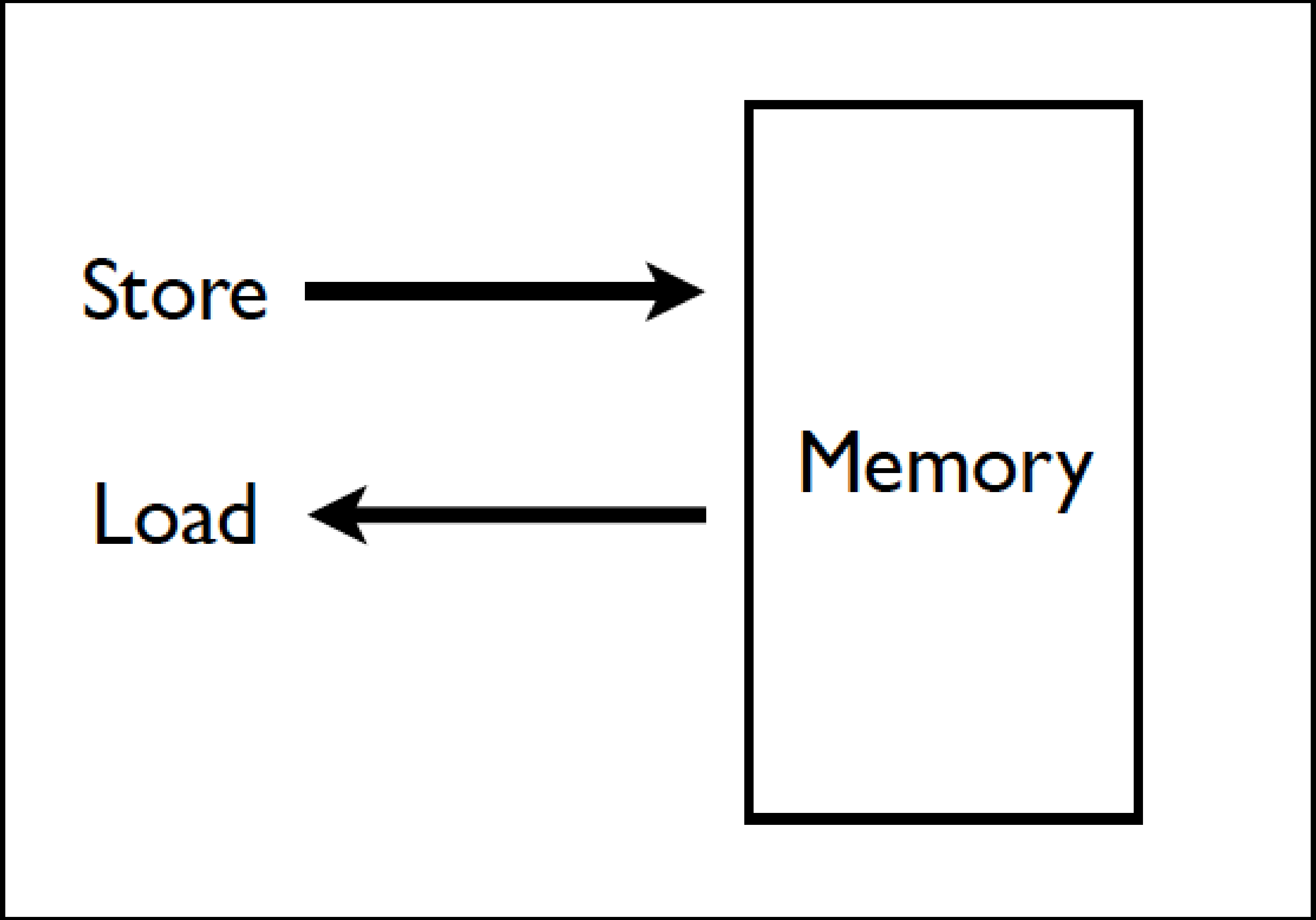


So where do A and B come from?

The “Storage Thing”

aka: the register file and main memory

Memory (Programmer’s View)



Computer memory hierarchy

Sorted by data access speed

- Registers: in the processor
- Cache: several levels, the closest is next to the processor
- Memory: off-chip
- Hard disk: for virtualization; requires OS support to access
- Network: not for process execution …

Computer memory hierarchy

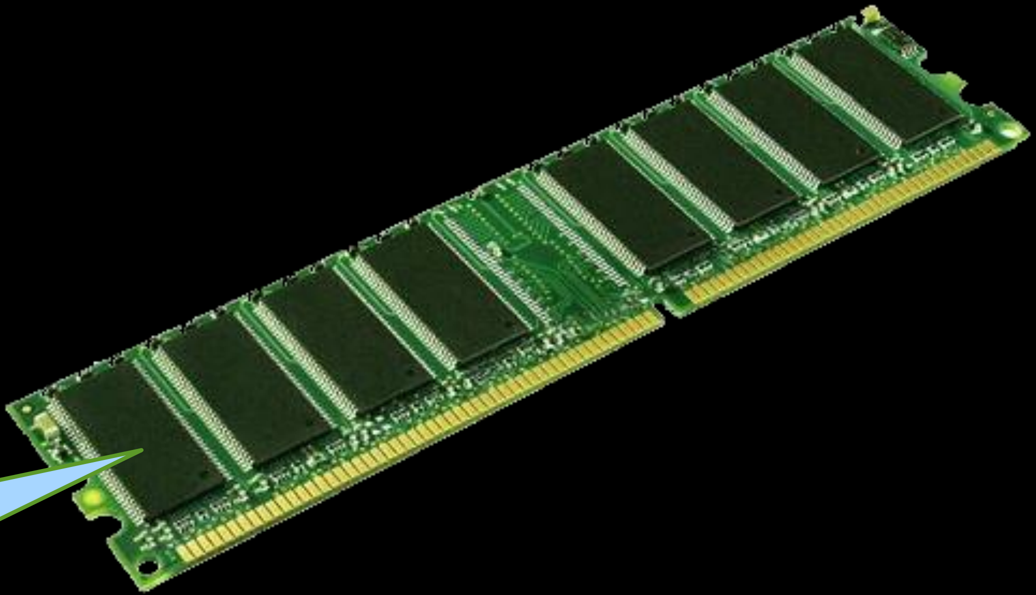
Sorted by data (food) access speed

- Registers: that plate in front of you
- Cache: the fridge in the kitchen
- Memory: the grocery store downstairs
- Hard disk: the farm in the suburb
- Network: the farm in a different country

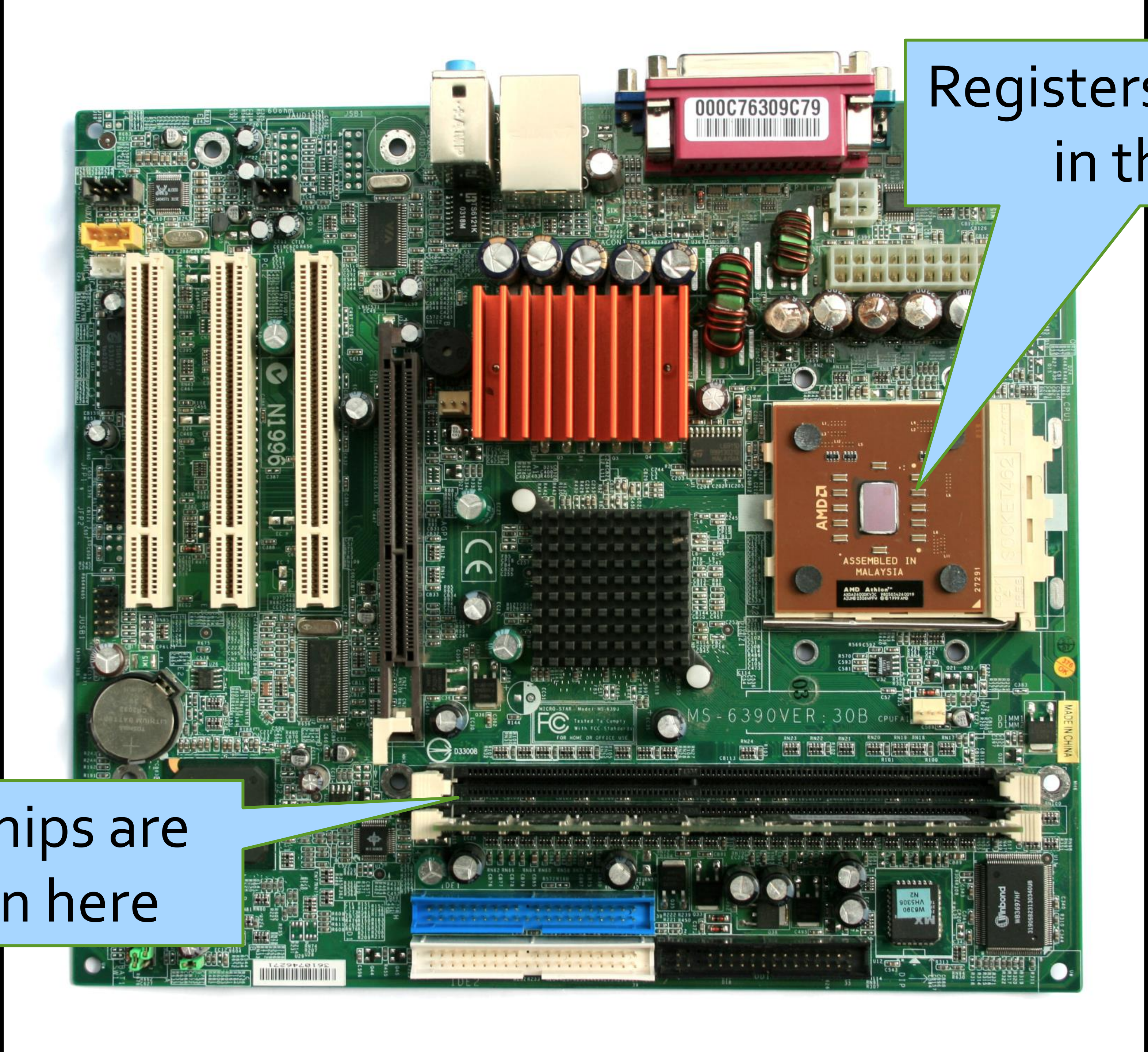
Memory and registers

- There are units in the CPU that store multiple data values for use by the CPU:
  - Registers: Small number of fast memory units that allow multiple values to be read and written simultaneously.
  - Main memory: Larger grid of memory cells that are used to store the main information to be processed by the CPU.

Registers are in here

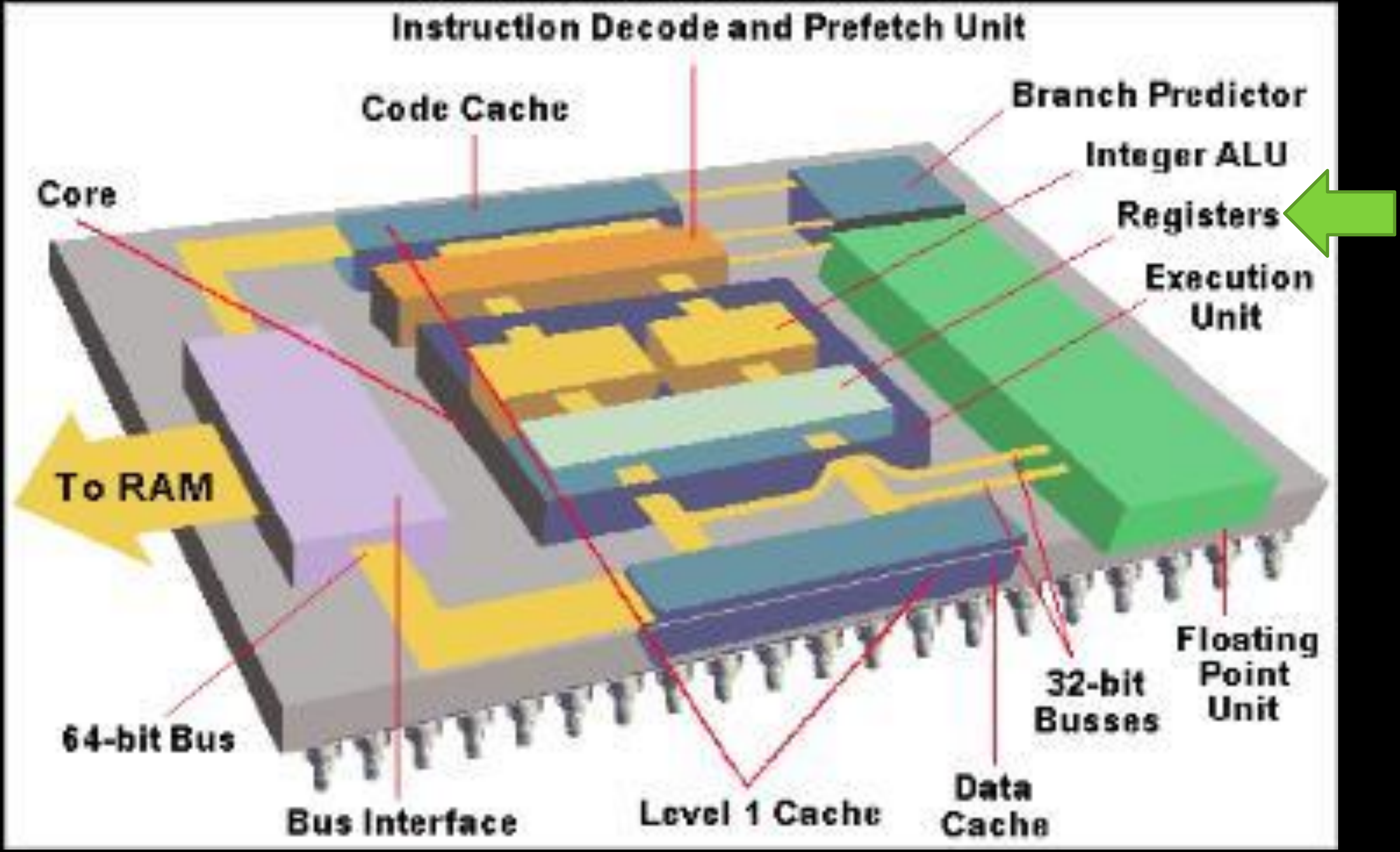


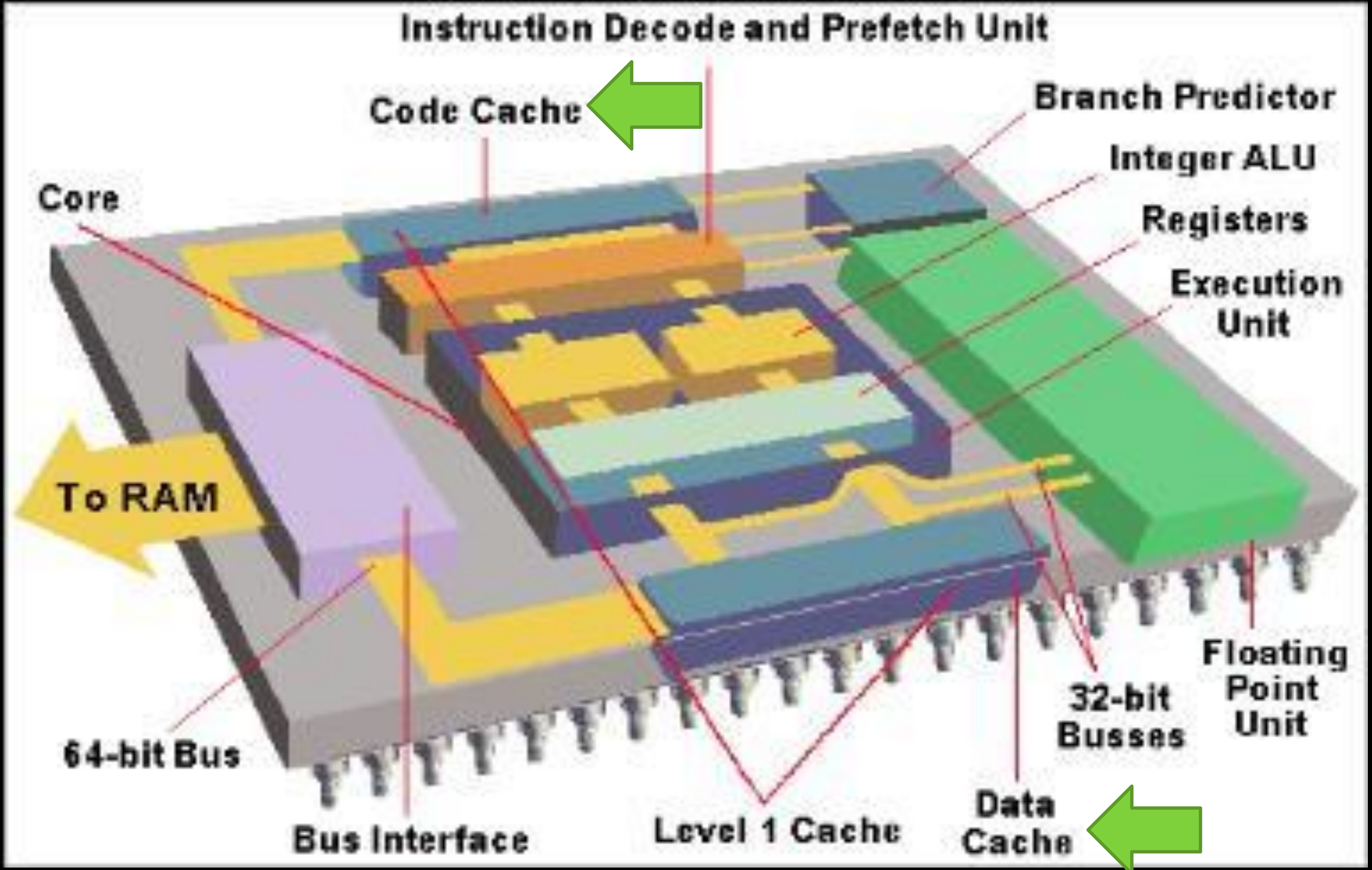
Memory is in here



Registers are in here in the CPU

Memory chips are plugged in here


Instruction Decode and Prefetch Unit
Code Cache
Branch Predictor
Integer ALU
Core
Registers
Execution Unit
To RAM
Floating Point Unit
32-bit Busses
64-bit Bus
Data Cache
Bus Interface
Level 1 Cache

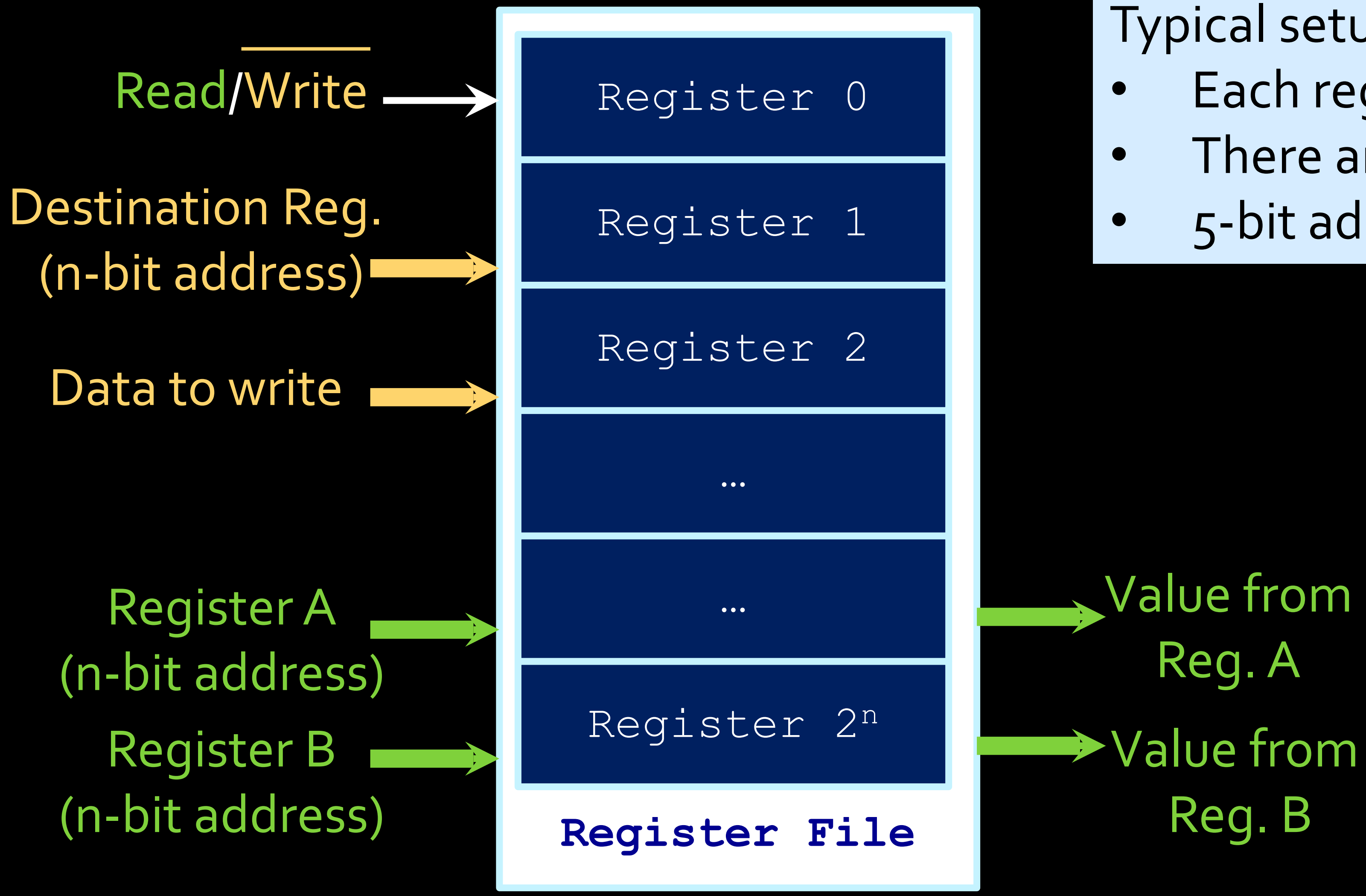


Instruction Decode and Prefetch Unit
Code Cache
Branch Predictor
Core
Integer ALU
Registers
Execution Unit
To RAM
Floating Point Unit
32-bit Busses
64-bit Bus
Data Cache
Bus Interface
Level 1 Cache

Register file

An array of registers in the CPU

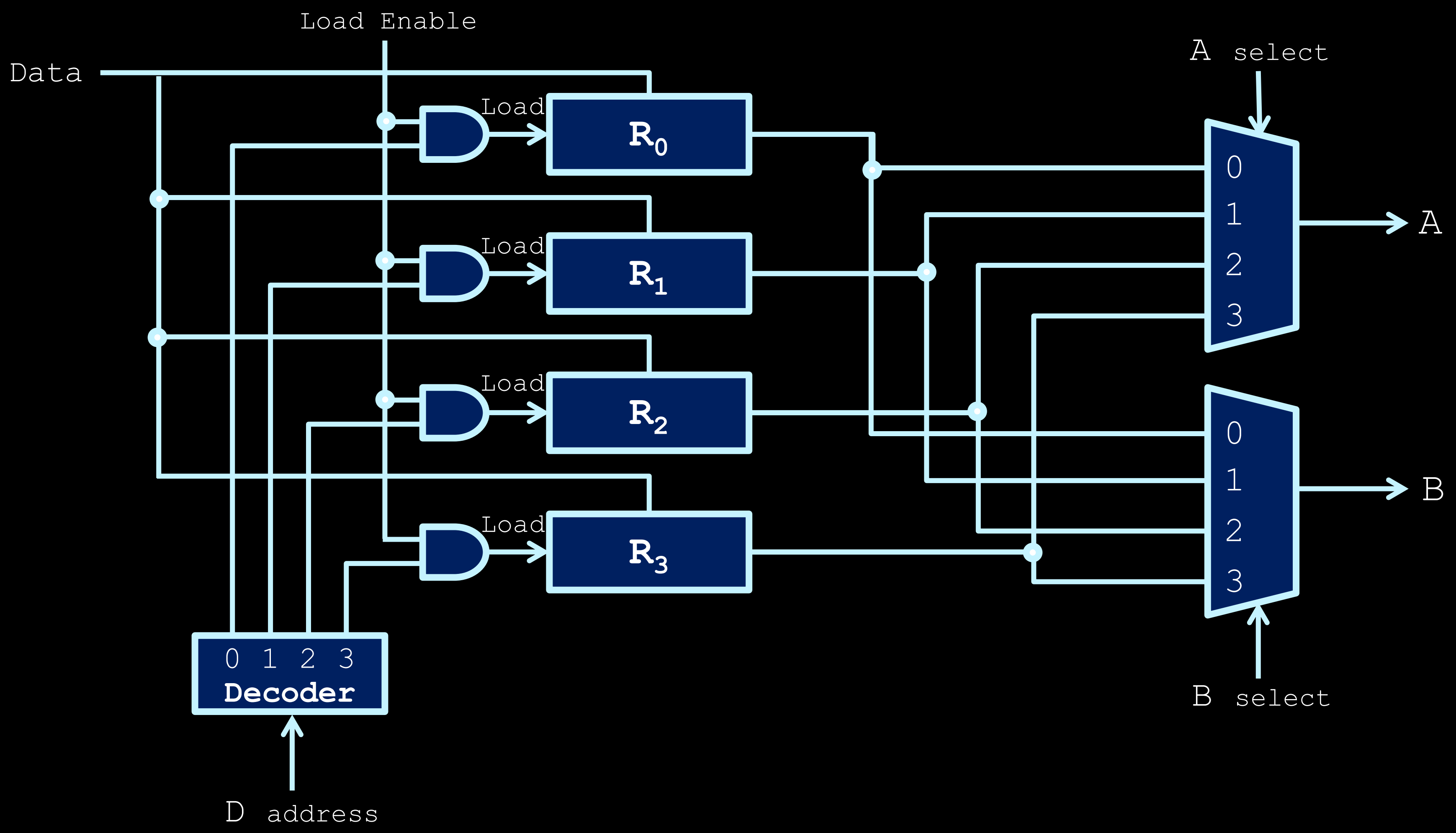
Register File Functionality



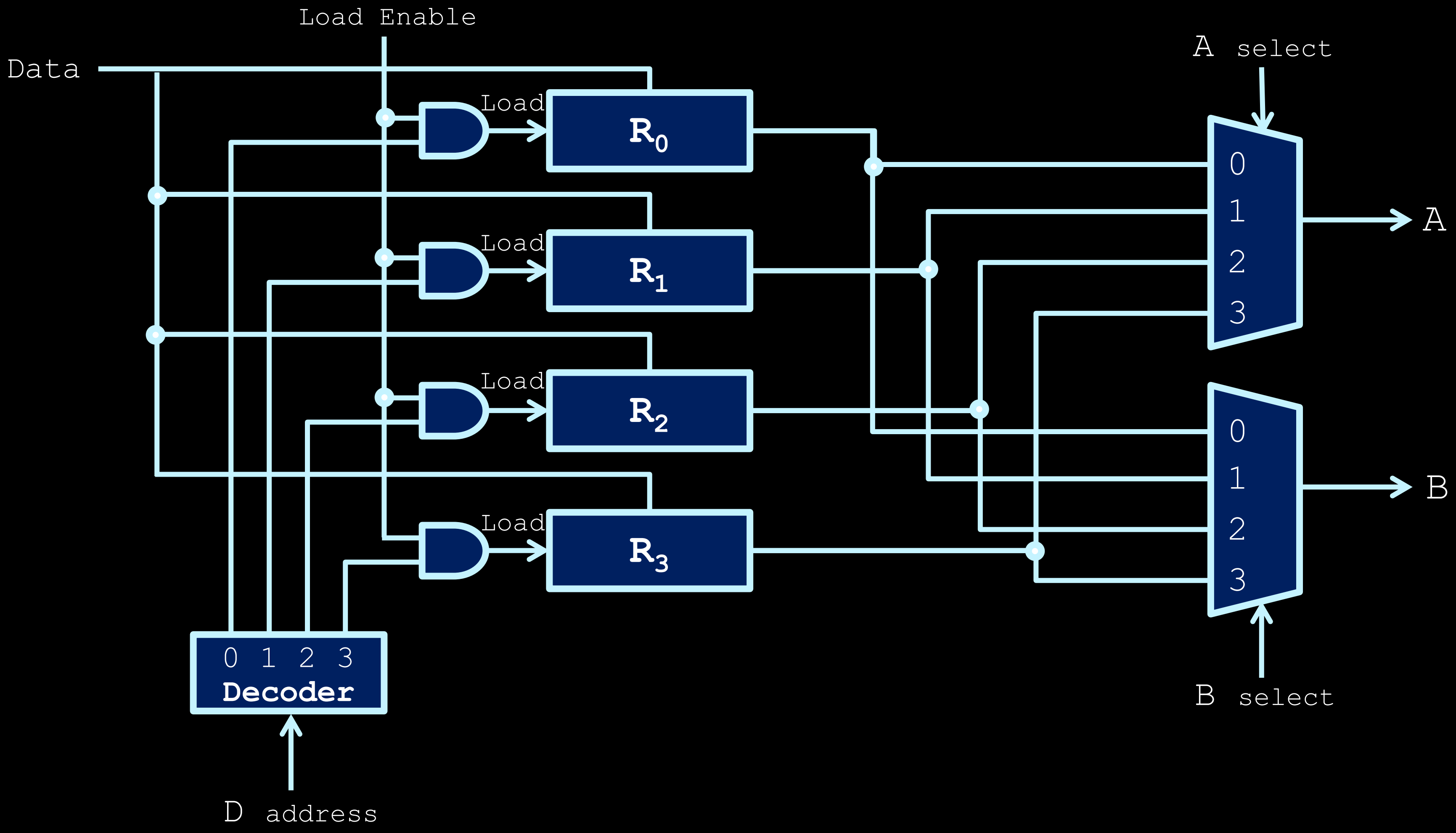
Typical setup (MIPS):

- Each register is 32-bit
- There are 32 registers.
- 5-bit address

Register File – Write Operation



Register File – Read Operation

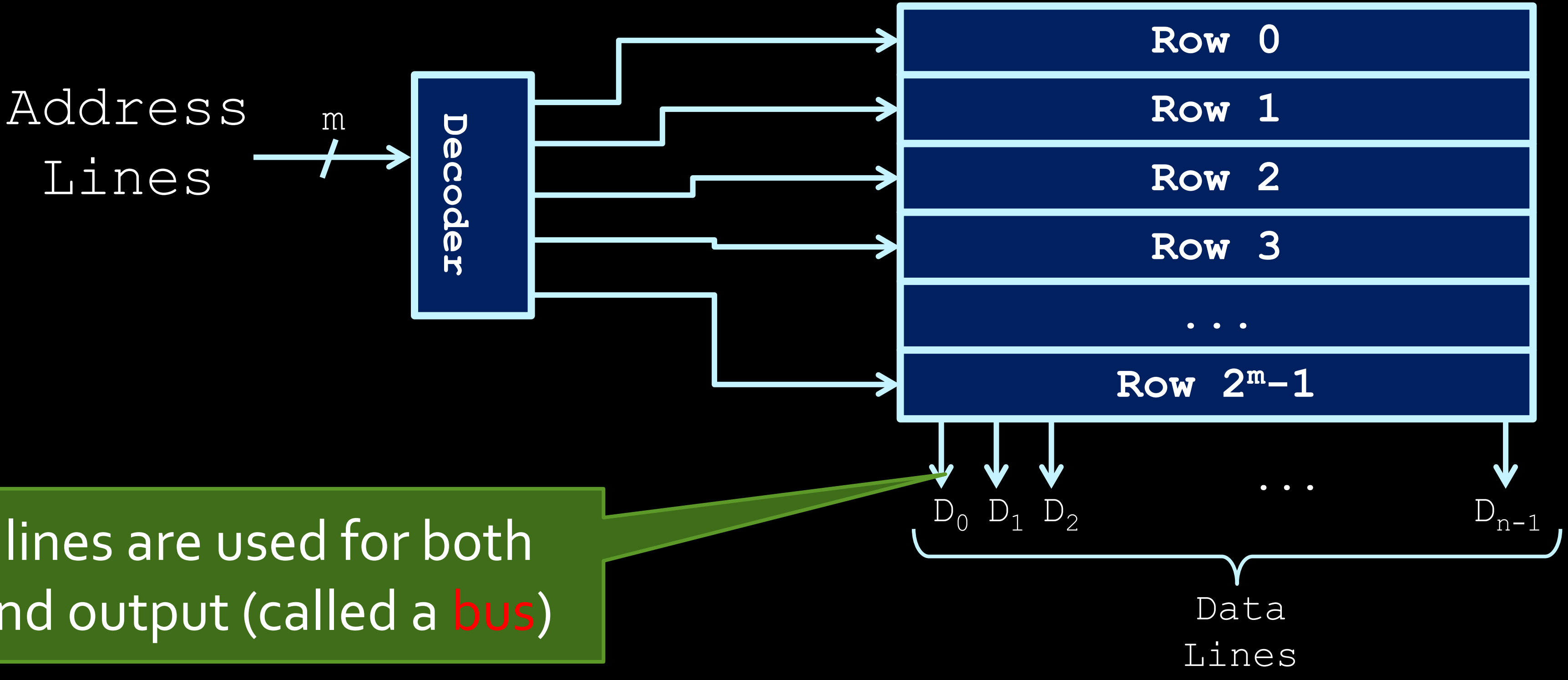


The main memory

An array of memory units

Electronic Memory

- Like register files, main memory is made up of a decoder and rows of memory units.



These lines are used for both input and output (called a bus)

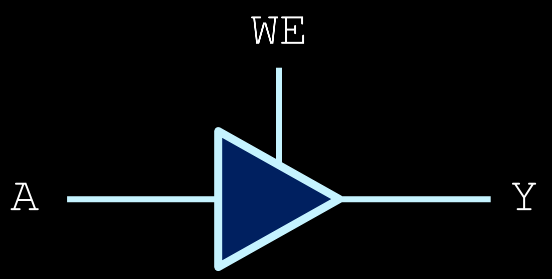
One-hot decoder

- The decoder takes in the m-bit binary address and activates a single row in the memory array.

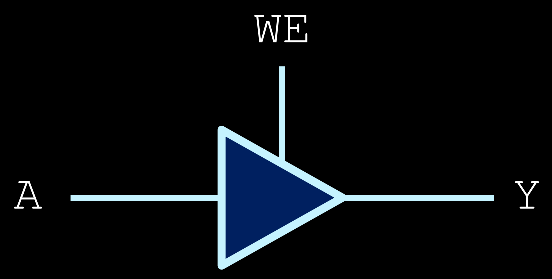
A_2	A_1	A_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
…			…							
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Controlling the flow

- Since some lines (buses) will now be used for both input and output, we use a tri-state buffer (remember them from the mux example?)
- When WE (write enable) signal is low, buffer output is a high impedance signal (connected to neither high voltage or ground).

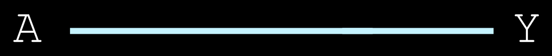


WE	A	Y
0	x	z
1	0	0
1	1	1

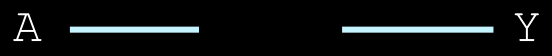


WE	A	Y
0	x	z
1	0	0
1	1	1

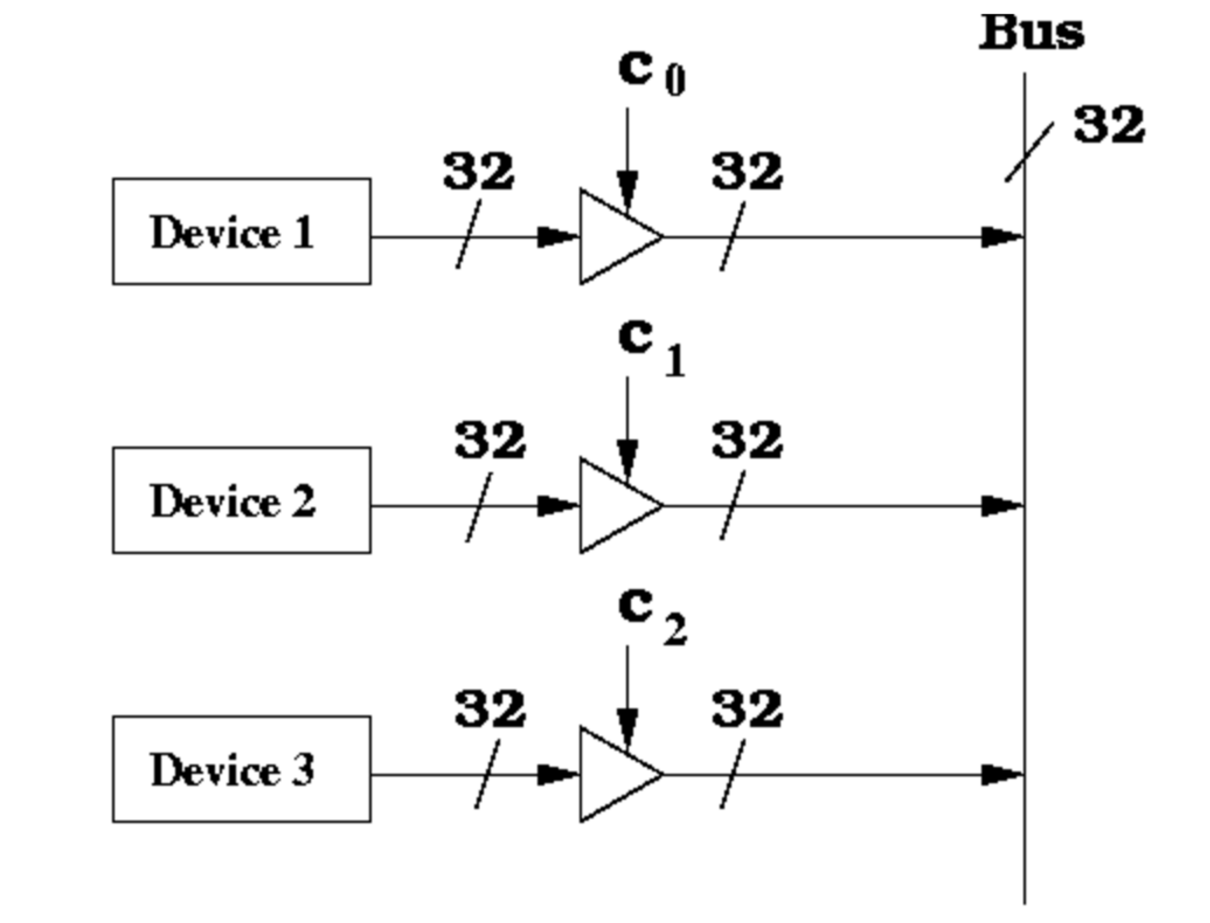
WE = 1



WE = 0



Control the flow using tri-state buffer

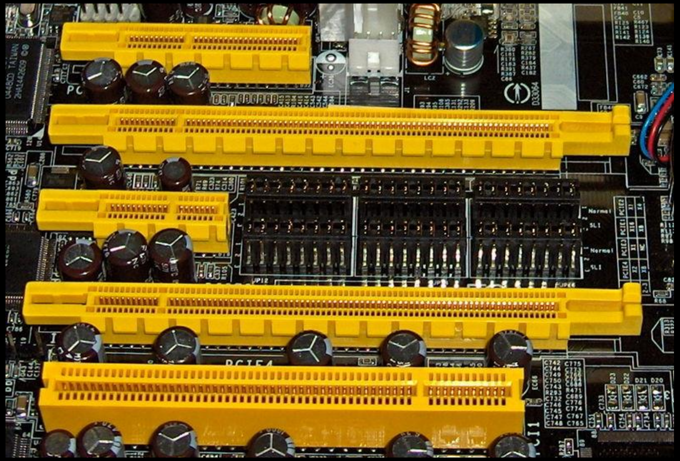


Control c0, c1 and c2 so that only one of the devices output is written to the bus.

In general, the bus can be read by multiple devices at the same but can only be written by one device at a time.

Data Bus

- Communication between components takes place through groups of wires called a bus (or data bus).
  - Multiple components can read from a bus, but only one can write to a bus at a time.
  - Each component has a tri-state buffer that feeds into the bus. When not reading or writing, the tri-state buffer drives high impedance onto the bus.

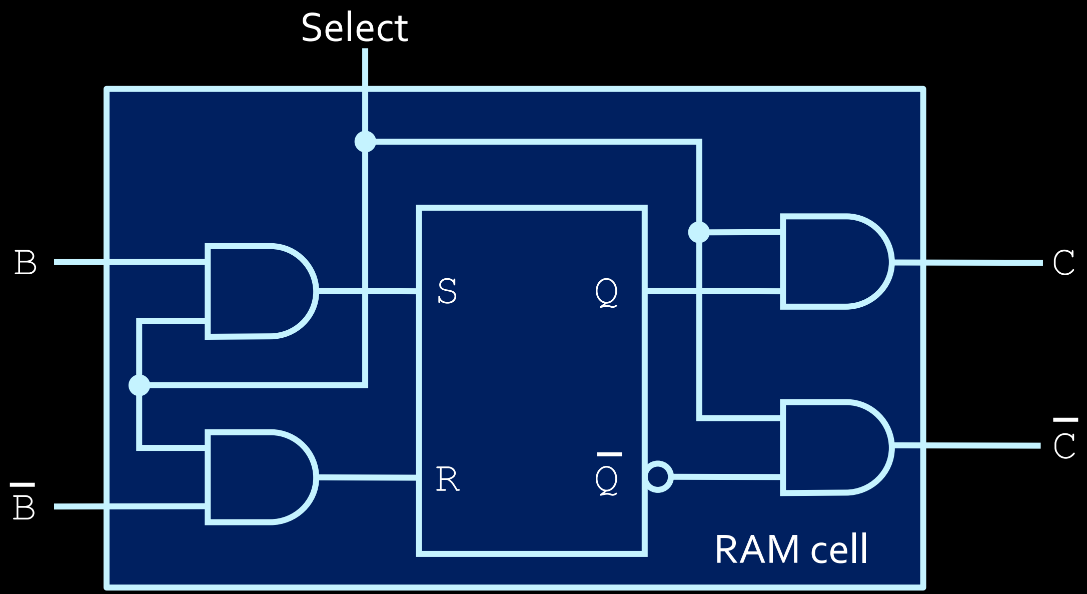


RAM Storage Cell

what stores each bit of the memory

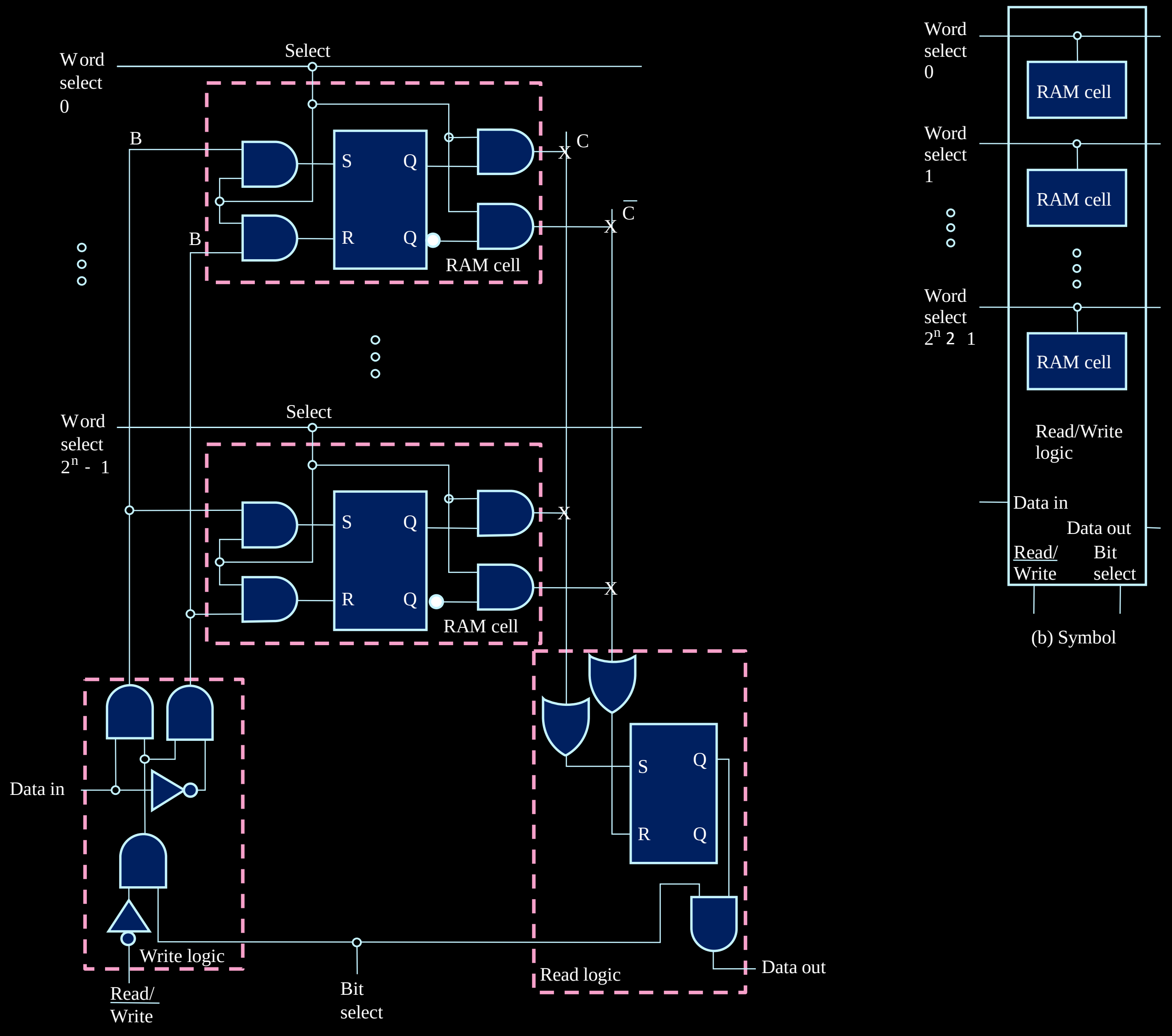
Storage cells

- For storing a single bit
- Each row is made of an array of storage cells.
- Multiple ways of representing these cells.
  - e.g. RAM cell (basically a gated latch)



RAM slice model

- Word select signals determine which row to send out on the C lines.



(b) Symbol

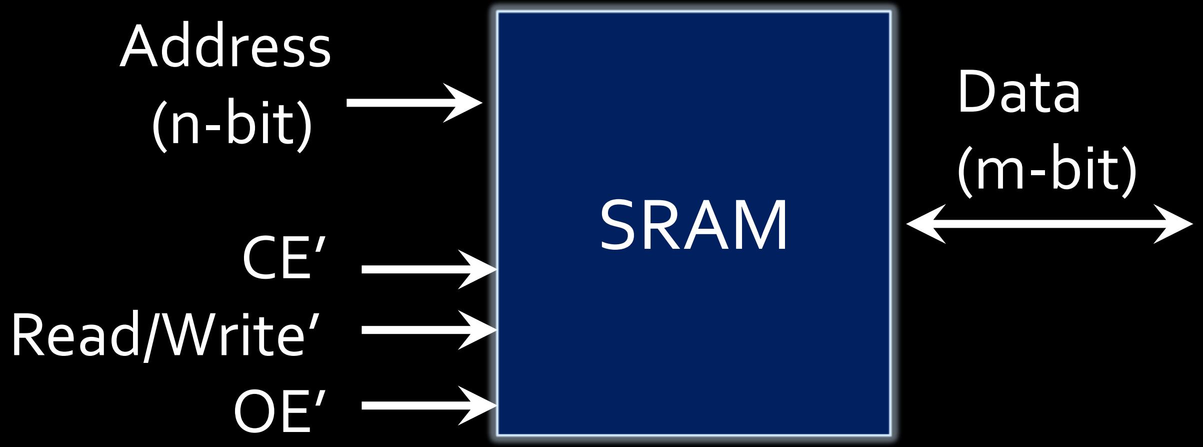
Example:

SRAM

Static Random Access Memory

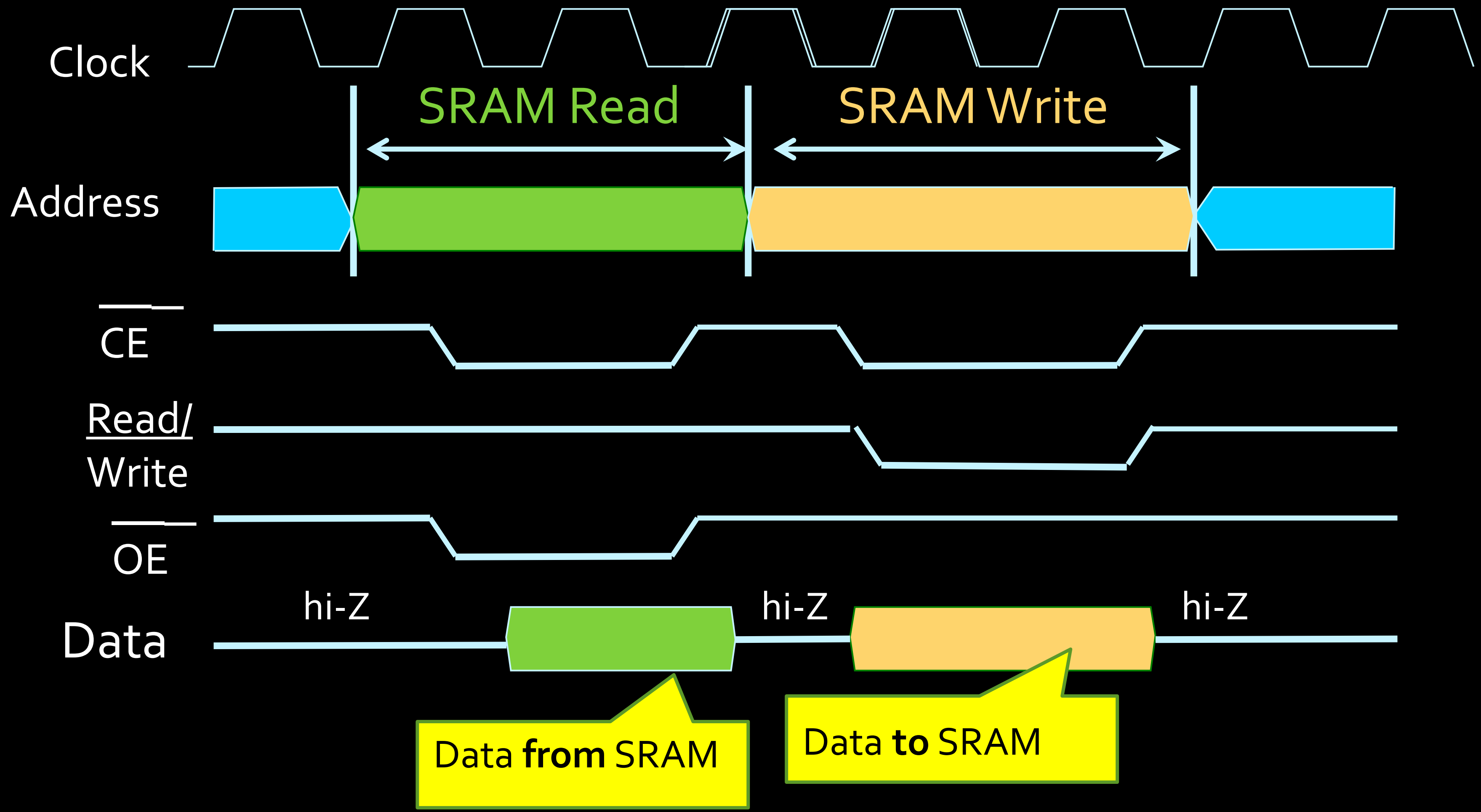
There are other types of RAMs such as DRAM, SDRAM, DDR SDRAM, RDRAM, VRAM, etc.

Asynchronous SRAM Interface – An example



Chip Enable’ (CE’)	Read/Write’	Output Enable’ (OE’)	Access Type
0	0	X	SRAM Write
0	1	0	SRAM Read
1	X	X	SRAM not enabled

Read/Write SRAM - Timing waveforms



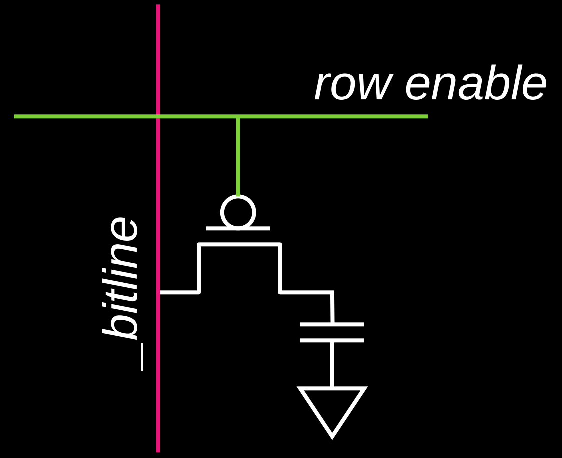
- Reading and writing of signals takes time. To make sure things are read/written correctly, we must control the timing carefully.

Memory vs registers

- Memory houses most of the data values being used by a program.
- Registers are more local data stores, meant to be used to execute an instruction.
  - Registers are not meant to host memory between instructions (like scrap paper for a calculation).
  - Exception is the stack pointer register, which is sometimes in the same register file as the others.

Memory Technology: DRAM

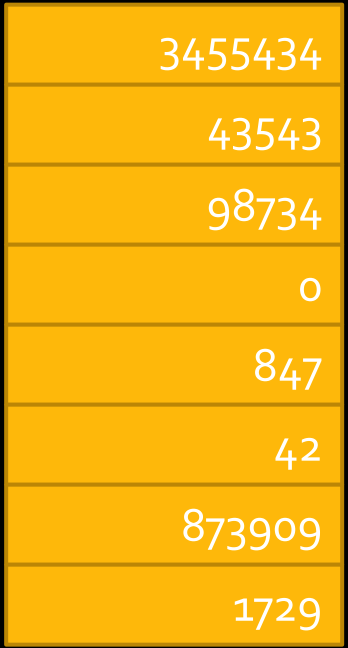
- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor



- Capacitor leaks through the RC path
  - DRAM cell loses charge over time
  - DRAM cell needs to be refreshed

DRAM – Dynamic Random Access Memory

Array of Values



3455434

43543

98734

0

847

42

873909

1729

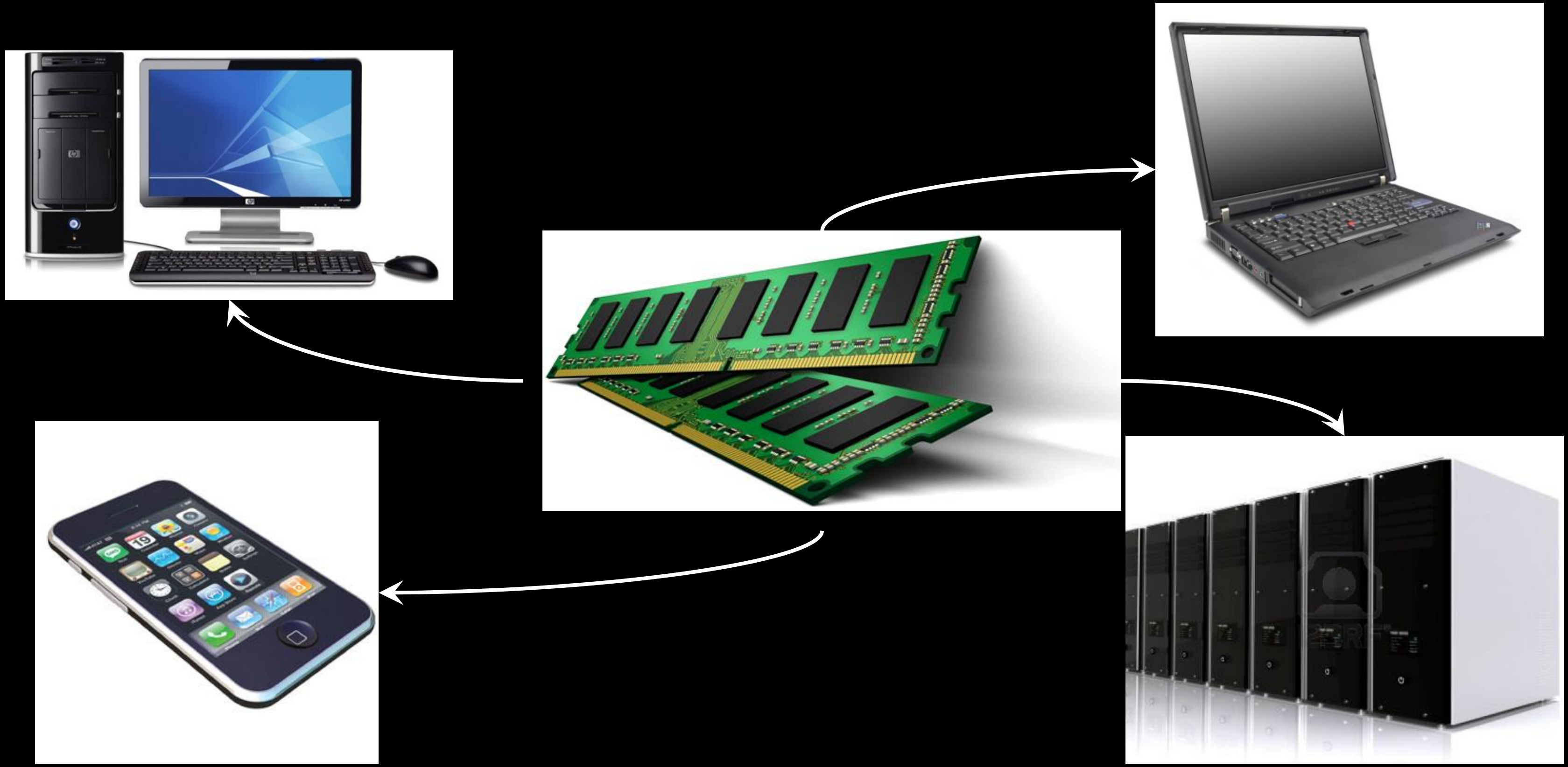
`READ` *address*

`WRITE` *address*, value

Accessing any location takes the same amount of time

Data needs to be constantly refreshed

DRAM in Today’s Systems

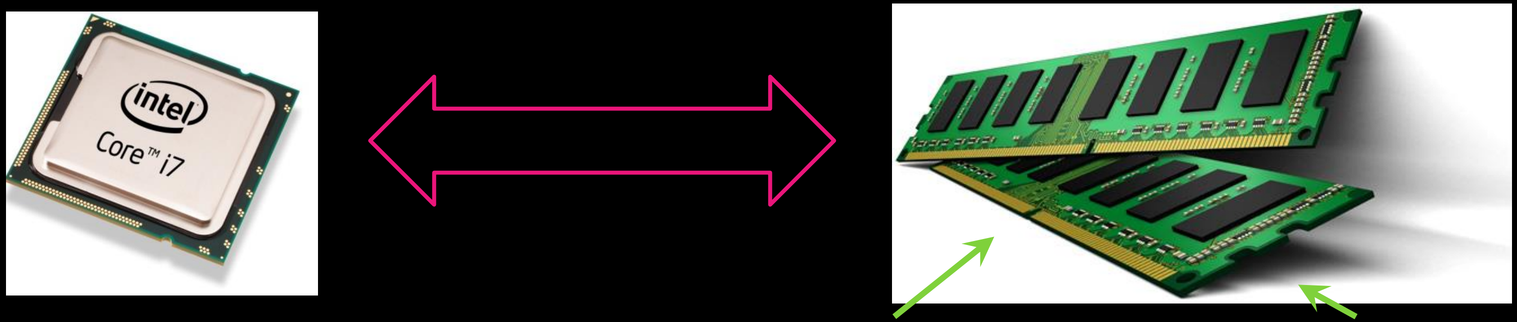


Why DRAM? Why not some other memory?

Von Neumann Model

Processor

Memory



Program	Data
T1 = Read Data[1]	3
T2 = Read Data[2]	4
T3 = T1 + T2	8
Write Data[3], T3	2

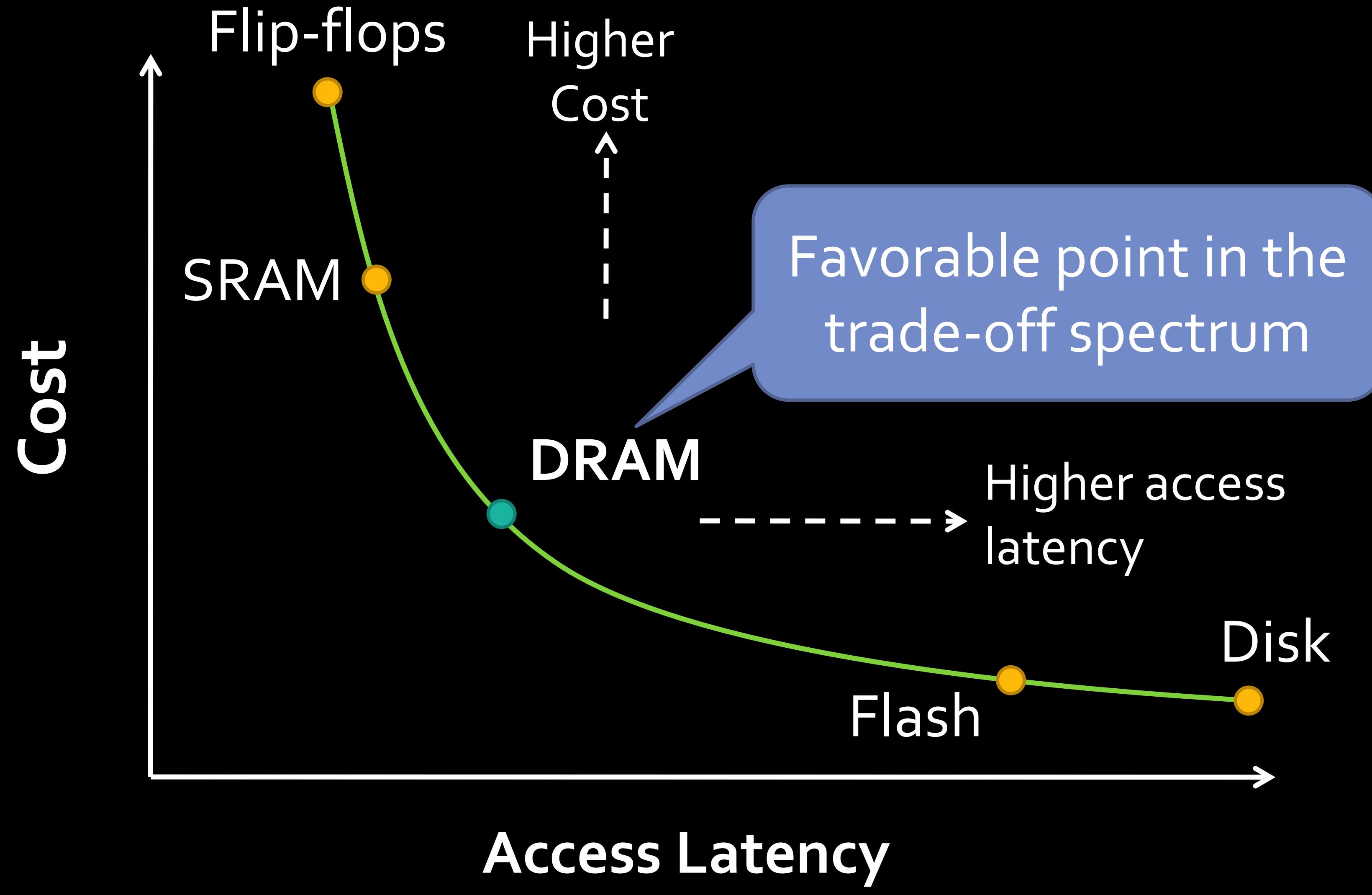
4 instruction accesses +
3 data accesses

Memory performance is important

Factors that Affect Choice of Memory

1. Speed
   - Should be reasonably fast compared to processor
2. Capacity
   - Should be large enough to fit programs and data
3. Cost
   - Should be cheap

Why DRAM?



Favorable point in the trade-off spectrum

Is DRAM Fast Enough?

Processor

Core

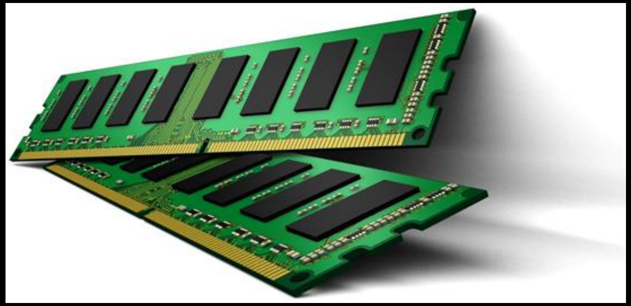
Core

Core

Core

Latency

Commodity DRAM



3 GHz, 2 Instructions / cycle

50ns

300 Instructio

300 Instructions

300 Instructions

300 Instructions

Parallelism

Request

Request

Request

Request

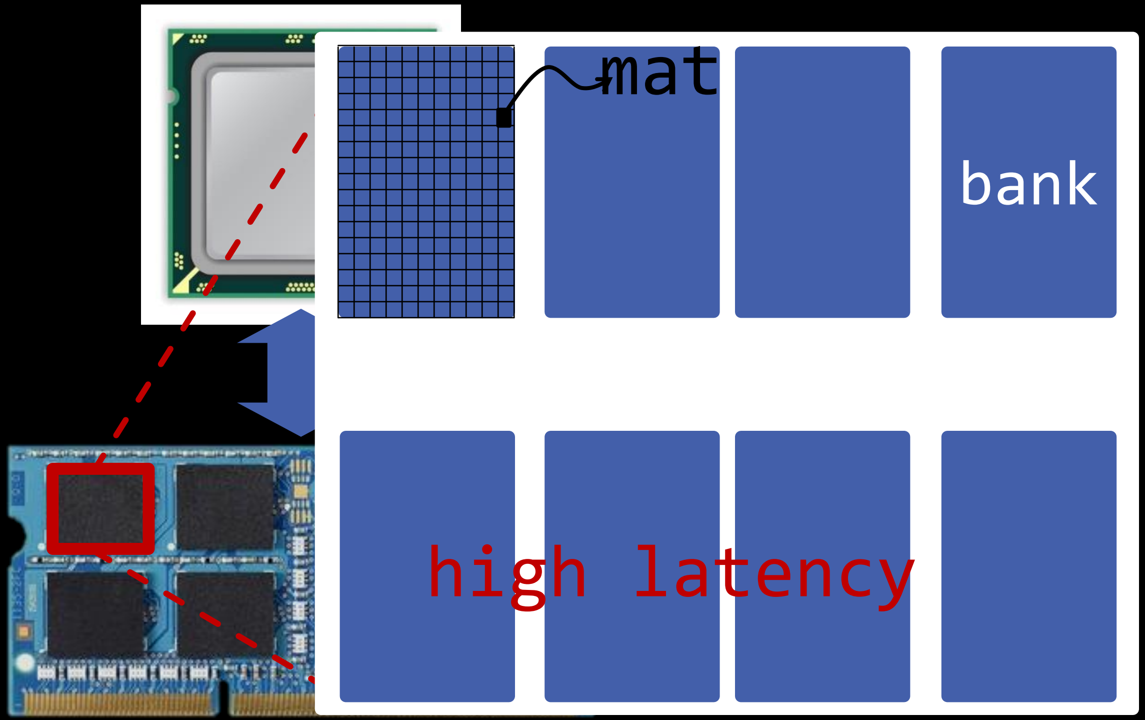
Independent programs

Served in parallel?

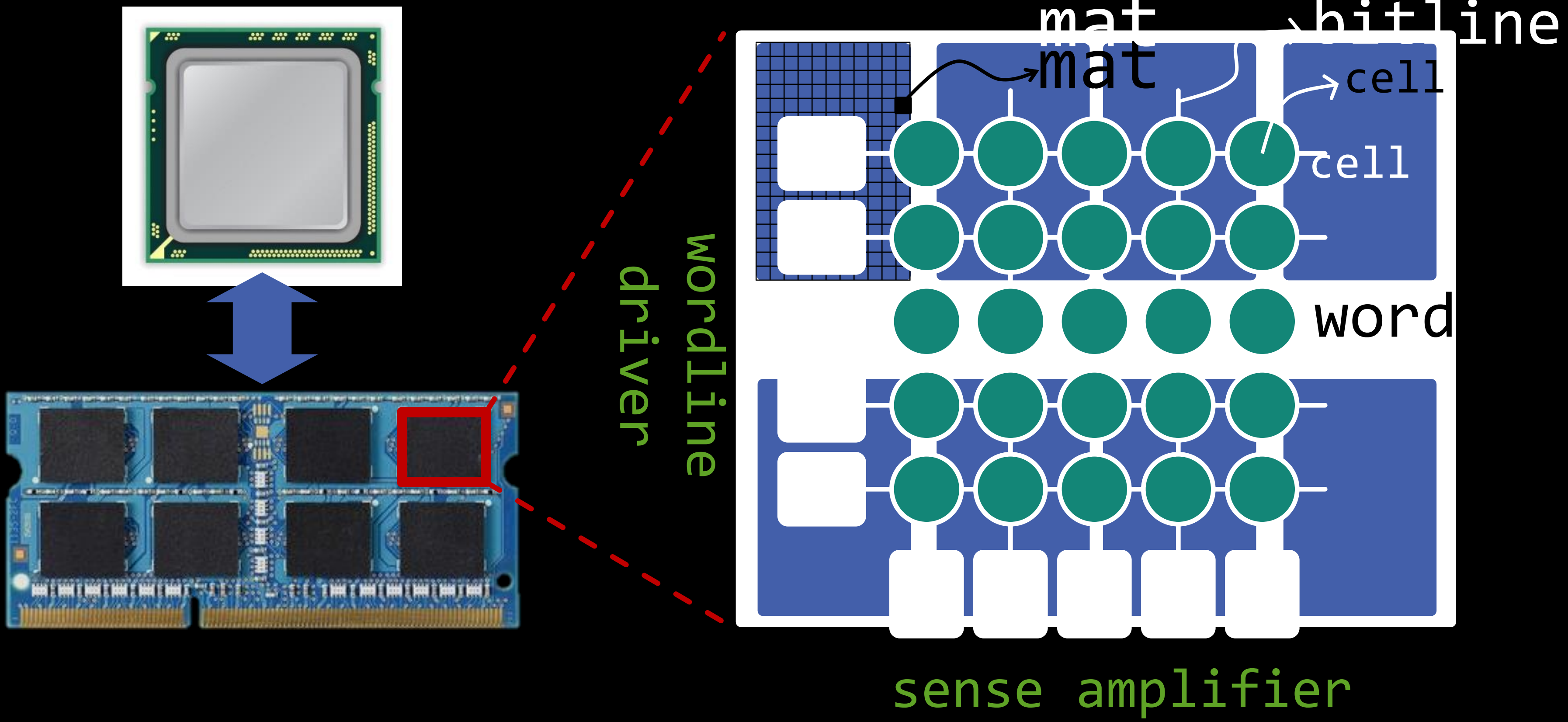
DRAM Organization

processor

main memory

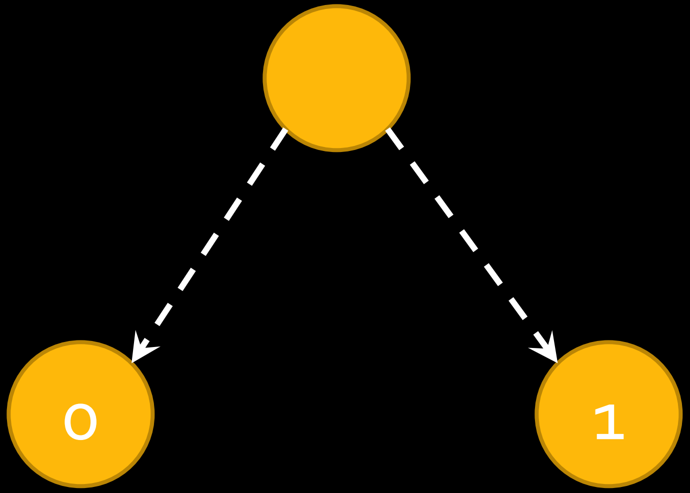


DRAM Cell Array: Mat



Memory Element (Cell)

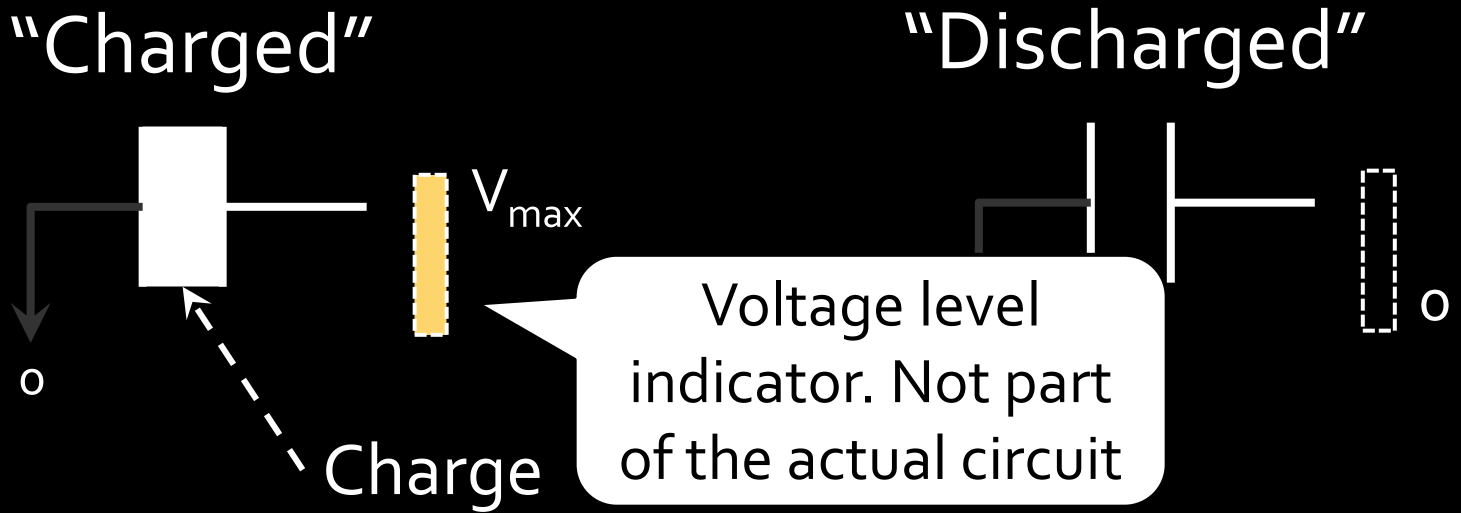
Component that can be in at least two states



Can be electrical, magnetic, mechanical, etc.

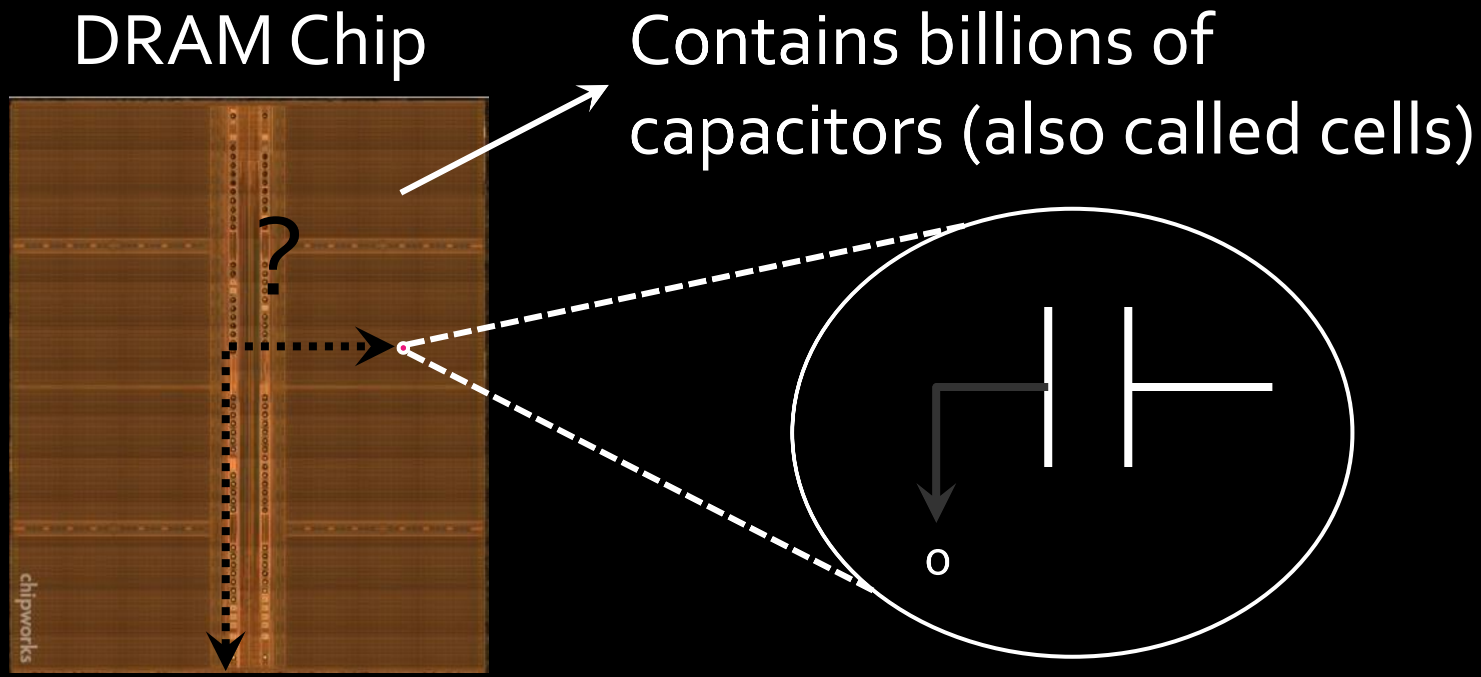
DRAM → Capacitor

Capacitor – Bucket of Electric Charge



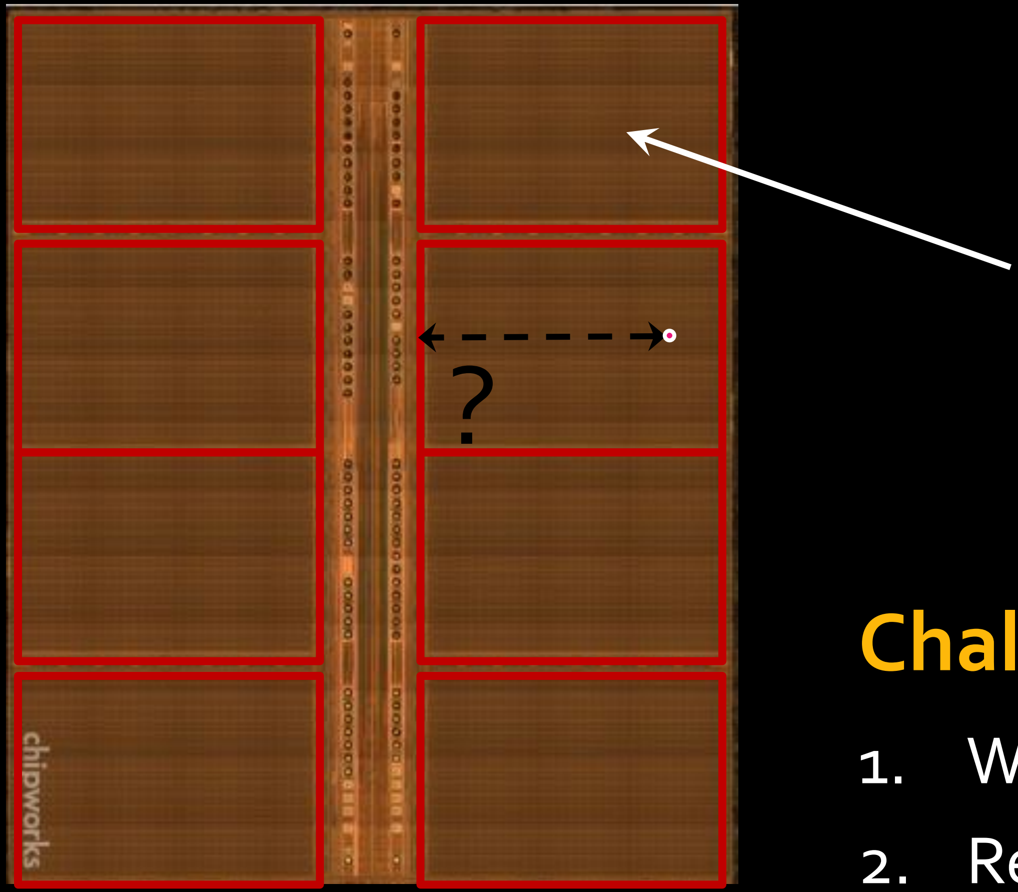
Voltage level indicator. Not part of the actual circuit

DRAM Chip



Contains billions of capacitors (also called cells)

Divide and Conquer

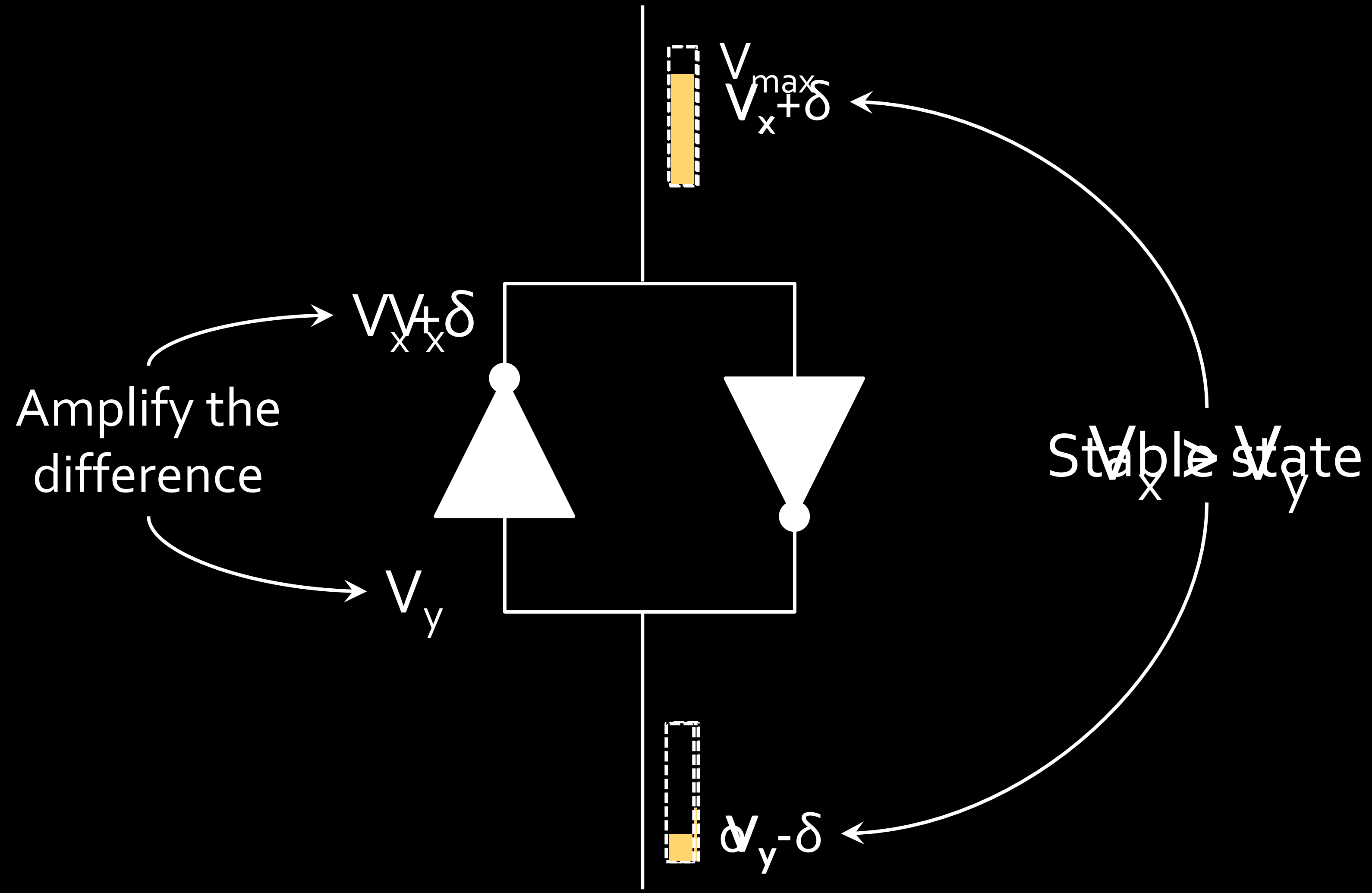


DRAM Bank

Challenges

1. Weak
2. Reading destroys state

Sense-Amplifier



But … memory is far away

- Most processor spend most of their time waiting.
  - … often for memory. This delay is referred to as the “*memory wall*”.
- No matter how fast we make a processor, if memory is too far away, we’ll just spend more time waiting.
  - As processors get faster, more processor cycles can be executed before a *load* completes.
- As a result, *Amdahl’s Law* tells us memory access time is an aspect of performance that has become increasingly important.

Aside: Amdahl's Law

- *Amdahl's law* is a statement in computer architecture about system performance.

The performance gain in task achievable by optimizing a subproblem is limited by the proportion of time spent in that subproblem in the unoptimized task.

- Ex: If you only spend 1/100 of your time doing task A, then even if you reduce the time required for A to zero, you'll only save 1%.

Aside: Amdahl’s Law

- *Amdahl’s law* is a statement in computer architecture about system performance.

The performance gain in task achievable by optimizing a subproblem is limited by the proportion of time spent in that subproblem in the unoptimized task.

Conclusion: Focus your optimization efforts on the tasks that consume the most time.

Amdahl's Law

- f: Parallelizable fraction of a program
- P: Number of processors

$$\frac{1}{1 - f + \frac{f}{P}}$$

- Maximum speedup limited by serial portion: Serial bottleneck

But … memory is far away

- **Most processor spend most of their time waiting.**
  - … often for memory. This delay is referred to as the "*memory wall*".

- No matter how fast we make a processor, **if memory is too far away, we'll just spend more time waiting**.
  - As processors get faster, more processor cycles can be executed before a *load* completes.

- As a result, *Amdahl's Law* tells us memory access time is an aspect of performance that has become increasingly important.

Scaling the Memory Wall

- ***Caches*** are a structure that makes it appear that memory is closer than it is.

- Every load to memory fetches more than just the value that is loaded.
  - In fact, a lot of values -- a block (or line) -- is brought from the memory to a location close to the processor.
  - Why? The cost of a bus is its length. Making it wider is inexpensive.
  - The closer location is called a *cache*. It stores the value that was loaded and the values near it, in case they are needed soon.

Big Idea: Locality

Caches rely on *spatial and temporal locality*.

This is a Big Idea in computing. Basically: if we used something recently, we're likely to use it again (or something near it) soon.

- "It or something near it" is *spatial locality.*
- "… soon" is *temporal locality.*

Examples of Locality

- "Iterating over an array" exhibits both temporal and spatial locality.
- "Executing code" often exhibits temporal and spatial locality.
- "Accessing items from a dictionary" does not: the items in the dictionary may not be close to each other in memory.
- Linked lists and other dynamically allocated structures can also cause locality problems.

First, some key terms …

- Address
- Tag
- Block
- Set
- Associativity
- Hit rate (and miss rate)
- Average Memory Access Time (AMAT)

(Read Textbook Chapter 8.3 for detailed definitions!)

First, some key terms …

- The cache has a few sets of blocks
- In a *direct mapped* cache, each set has one block
- In a *N-way set associative* cache, each set has N blocks.
- A *fully associative* cache has one set with all the blocks.
- A memory address gets “hashed” to a set.
- Different memory addresses may be hashed to the same set.

Addresses and Caches

- Each *load* fetches an entire *cache block* -- not just a single value.
  - The size of a cache “block” is dependent on the cache.
  - A “block” is a set of words with closely related addresses.
  - Why fetch a whole block when you just need part of it?
    - spatial locality

- The easiest way to define a block is to look at its *mask*.

Bit Masking

- A *bit vector* is an integer that should be interpreted as a sequence of bits.
  - We can think of an address as a bit vector.
- A *mask* is a value that can be used to turn specific bits in a bit vector on or off.

- For example, let's set a mod-16 mask.

```
value = ....
mod_16 = 15               # 0x000000F
print (value & mod_16)    # Only the bottom 4 bits
                          # "&" is "bitwise and"
```

A small example

- Consider a 8-bit memory address (byte-addressable)
- 10101010, 256 different addresses
- What if we divide 256 addresses into 8-byte blocks?
- How many blocks are there?
- The address is now "hierarchical":
  - block number
  - offset within the block
- 10101010
- block number, block offset

Exercise: Cache Masking

Given a 32-bit address space, identify the tag, set, and block offset for a (direct mapped) cache that stores 16 32-byte blocks.

```
00000000 00000000 00000000 00000000  <-  32 bits
```

Exercise: Cache Masking

Given a 32-bit address space, identify the tag, set, and block offset for a (direct mapped) cache that stores 16 32-byte blocks.

In a direct-mapped cache, we use part of the address as an index into the cache. Since there are 16 storage locations in this cache, we need 4 (2^4 = 16) bits from the address as the index.

Exercise: Cache Masking

Given a 32-bit address space, identify the tag, set, and block offset for a (direct mapped) cache that stores 16 32-byte blocks.

```
00000000 00000000 00000000 00000000  <-   32 bits
                              ^^^^^  <- offset into a block
                         ^ ^^^       <- set
```

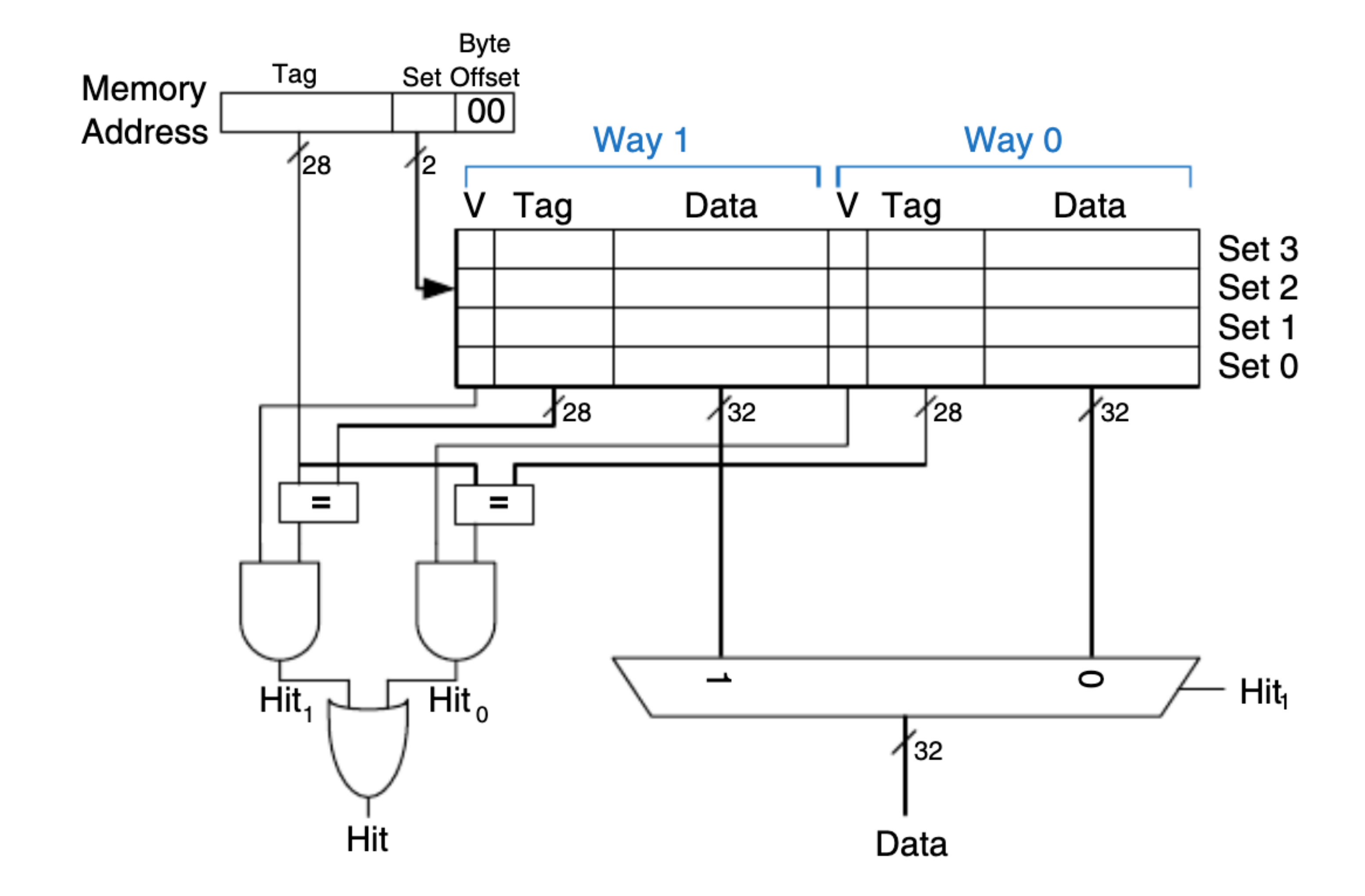
Everything else is the tag.

We match the tag to make sure the memory address matches.

Associativity

- Most caches use some form of hashing.
  - The caches are smaller than the memory they are caching from, so they can't store everything!
- If two blocks hash to the same value, they can't both be stored. To avoid that, caches are often *associative*.
  - A 2-way set associative cache can store two blocks that hash to the same value.
  - A fully associative cache doesn't have to worry about hash collisions at all.

2-way set associative cache: how it's done in hardware



Cache Loading and Evicting

- Each cache has a finite size.
  - It can store some maximum number of blocks.
  - Based on its *associativity*, it can store a set number of blocks with a specific hash.

- Every time a load is performed from memory, the block must be stored.
  - This means that another block might need to be *evicted.*

How do we choose what to evict?

- Ideally, we'd kick out data we never need again.

- But we can't see the future, so we do the next best thing. We rely on *locality* and kick out … something old.

- The most common heuristic is *"least recently used" (LRU).*
    - The cache block that was accessed the longest time ago is dropped.
- Other heuristics include *"first in first out" (FIFO)*, *"least frequently used"*, and others.

Discussion

What is the effect of increasing:

(a) block size
(b) associativity
(c) cache size

Let’s vary these parameters to see what happens.

Exercise: Cache Example

Simulate the performance of a cache on the following (hex) address *loads*.

```
40 48 4c 40 50 58 5c 40 60 48 4c 44 40 60 58 5c
```

The cache is direct-mapped and stores 4 words. It uses a FIFO eviction policy.

0	60
1	44
2	58
3	5c

16 accesses, and only one 1(!) hit

Exercise: Cache Example

Simulate the performance of a cache on the following (hex) address *loads*.

```
40 48 4c 40 50 58 5c 40 60 48 4c 44 40 60 58 5c
```

The cache is direct-mapped and stores 2 blocks of two words. It uses a FIFO eviction policy.

60	64
58	5c

16 accesses, and now 6 hits

Exercise: Cache Example

Simulate the performance of a cache on the following (hex) address *loads*.

```
40 48 4c 40 50 58 5c 40 60 48 4c 44 40 60 58 5c
```

The cache is 2-way set associative and stores 4 words. It uses a FIFO eviction policy.

60	58
44	5c

16 accesses, and now 3 hits

Exercise: Cache Example

Simulate the performance of a cache on the following (hex) address *loads*.

```
40 48 4c 40 50 58 5c 40 60 48 4c 44 40 60 58 5c
```

The cache is 2-way set associative and stores 4 words. It uses an LRU eviction policy.

60	58
5c	44

16 accesses, and now 2 hits

Exercise: Cache Example

Simulate the performance of a cache on the following (hex) address *loads*.

```
40 48 4c 40 50 58 5c 40 60 48 4c 44 40 60 58 5c
```

The cache is fully associative and stores 4 words. It uses a LRU eviction policy.

5c	60	58	40

Discussion

What is the effect of increasing:

(a) block size
(b) associativity
(c) cache size

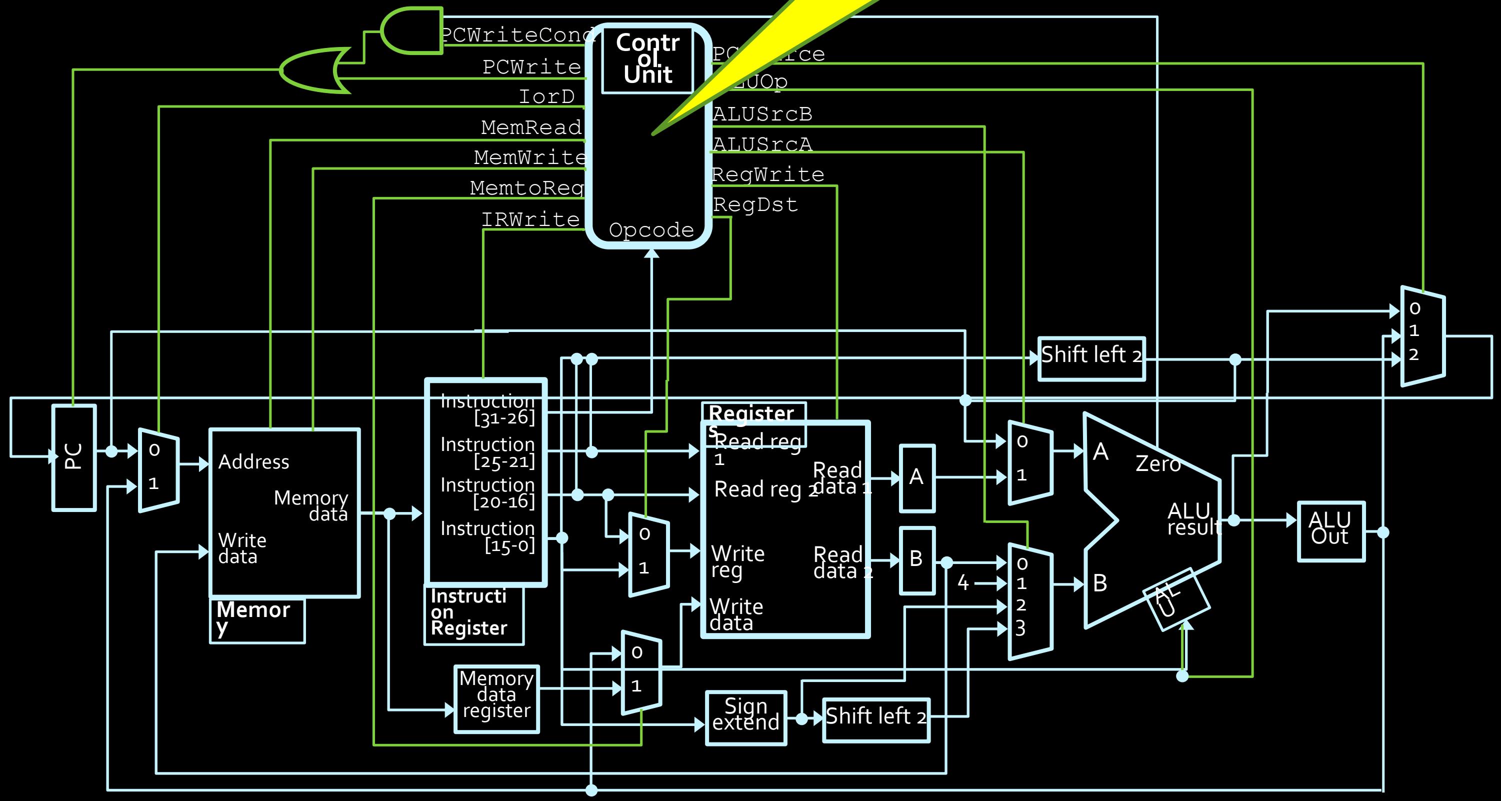
(Read Textbook Chapter 8.3 for more discussion!)

Cache Consistency

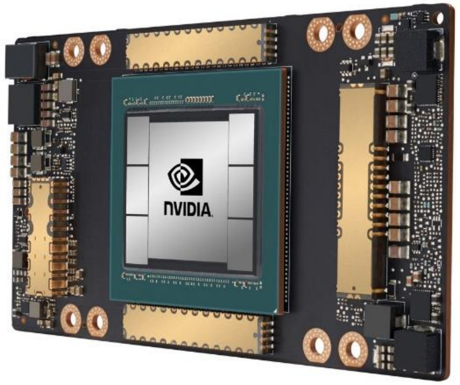
- Consistency is a huge problem -- and we won't discuss it.
- Imagine a multi-processor system: each processor has its own L1.
    - What happens when one of the processors issues a store? Is it written back to memory? To the other processor's L1?
    - How quickly does that write-back happen? What will occur if the second processor accesses that memory address before the write-back?

Next…

The Controller Thing



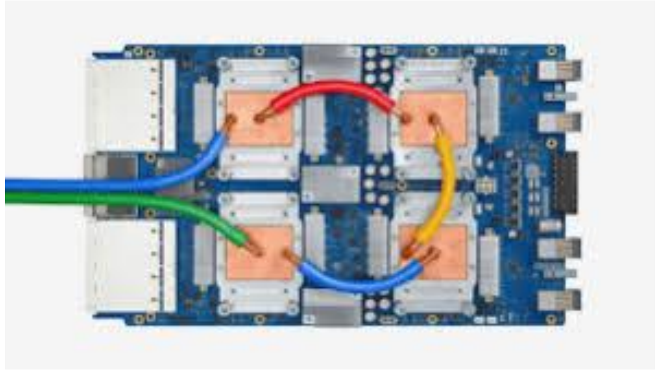
CSCB58:
Computer Organization



Prof. Gennady Pekhimenko

University of Toronto

Fall 2020



The content of this lecture is adapted from the lectures of Larry Zheng and Steve Engels