

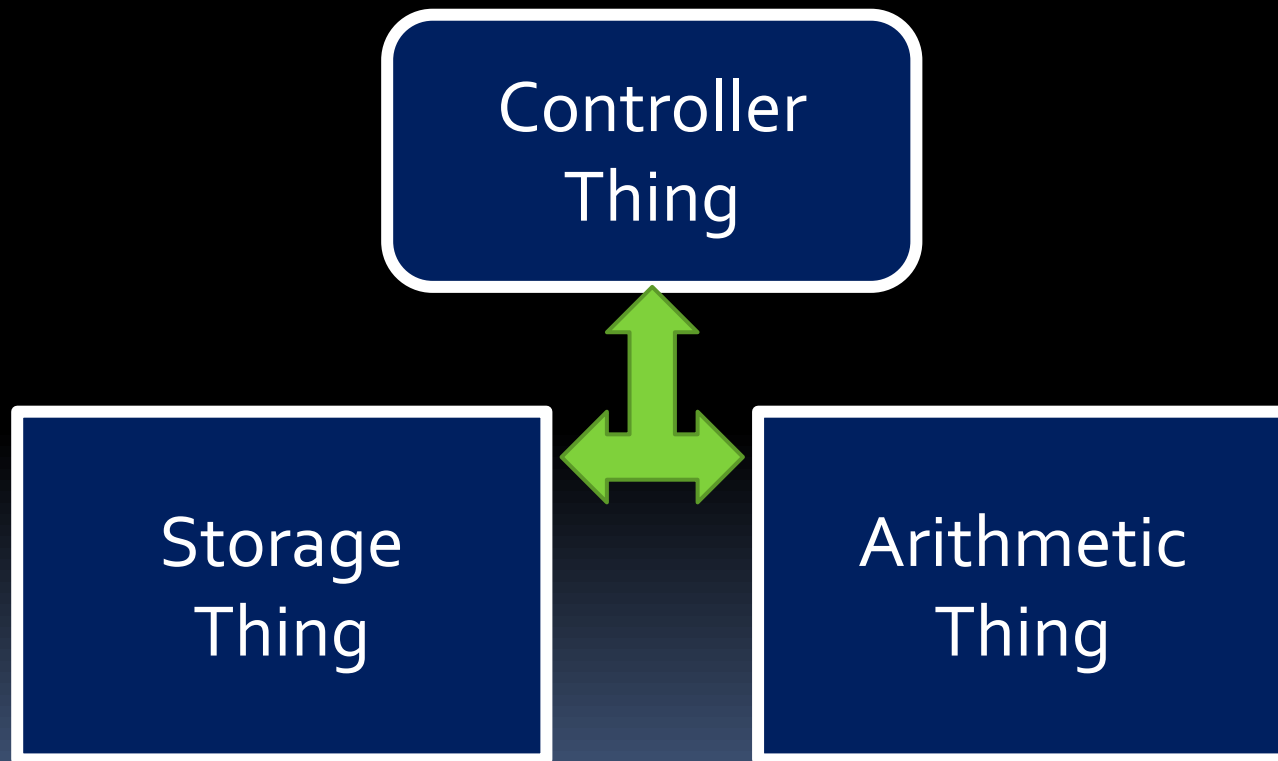


# Week 6 Review



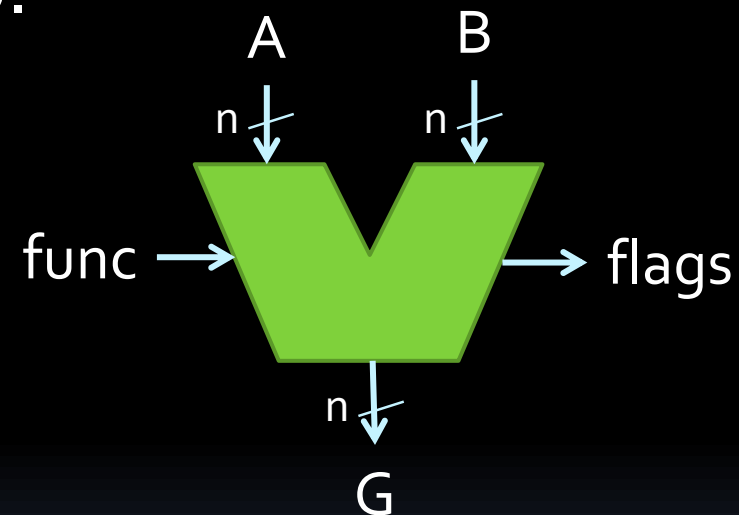
# Deconstructing Processors

- Simpler at a high level:



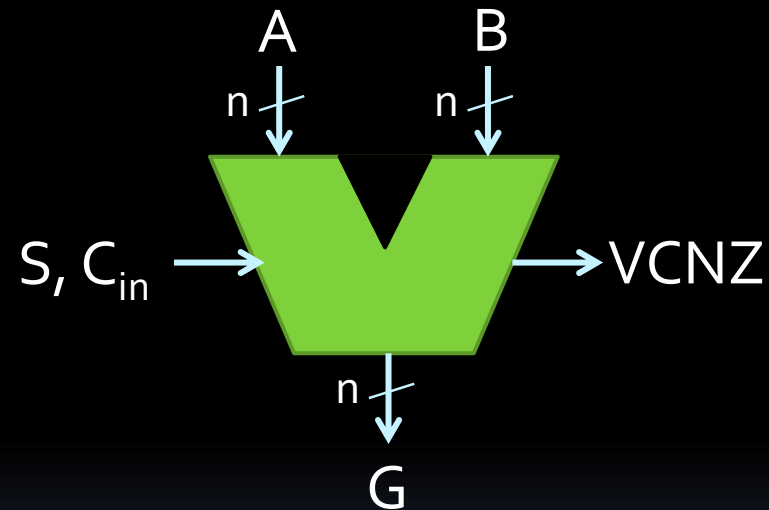
# Arithmetic Logic Unit

- The ALU is responsible for all data processing in a basic CPU.
- Usually looks like this.
  - A, B are **inputs**
  - G is the **result**
  - func tells it **what to compute**
  - flags indicate **conditions** of the result.
    - Examples: overflow, result is zero, result is negative, etc.



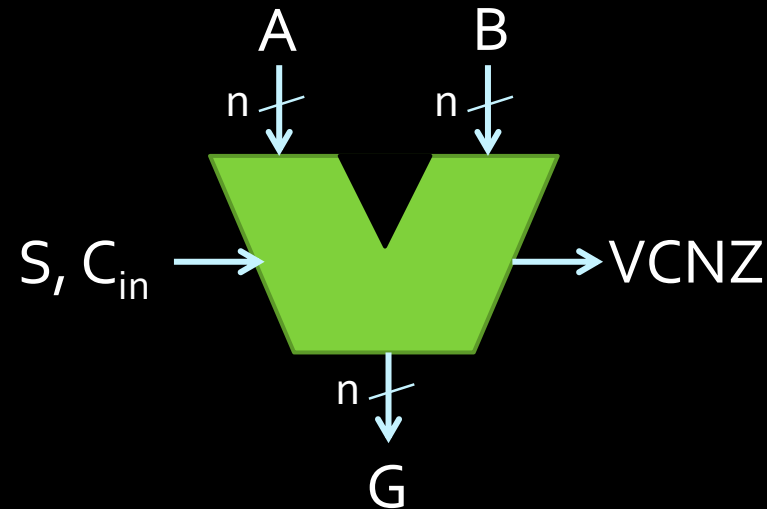
# Our Simple ALU

- Input **S** represents **3 select bits** that specify which operation to perform.
  - For example: S2 is a mode select bit, indicating whether the ALU is in arithmetic or logic mode
- The **carry-in bit  $C_{in}$**  will be useful for incrementing an input value or the overall result.
  - Go directly into the adder.

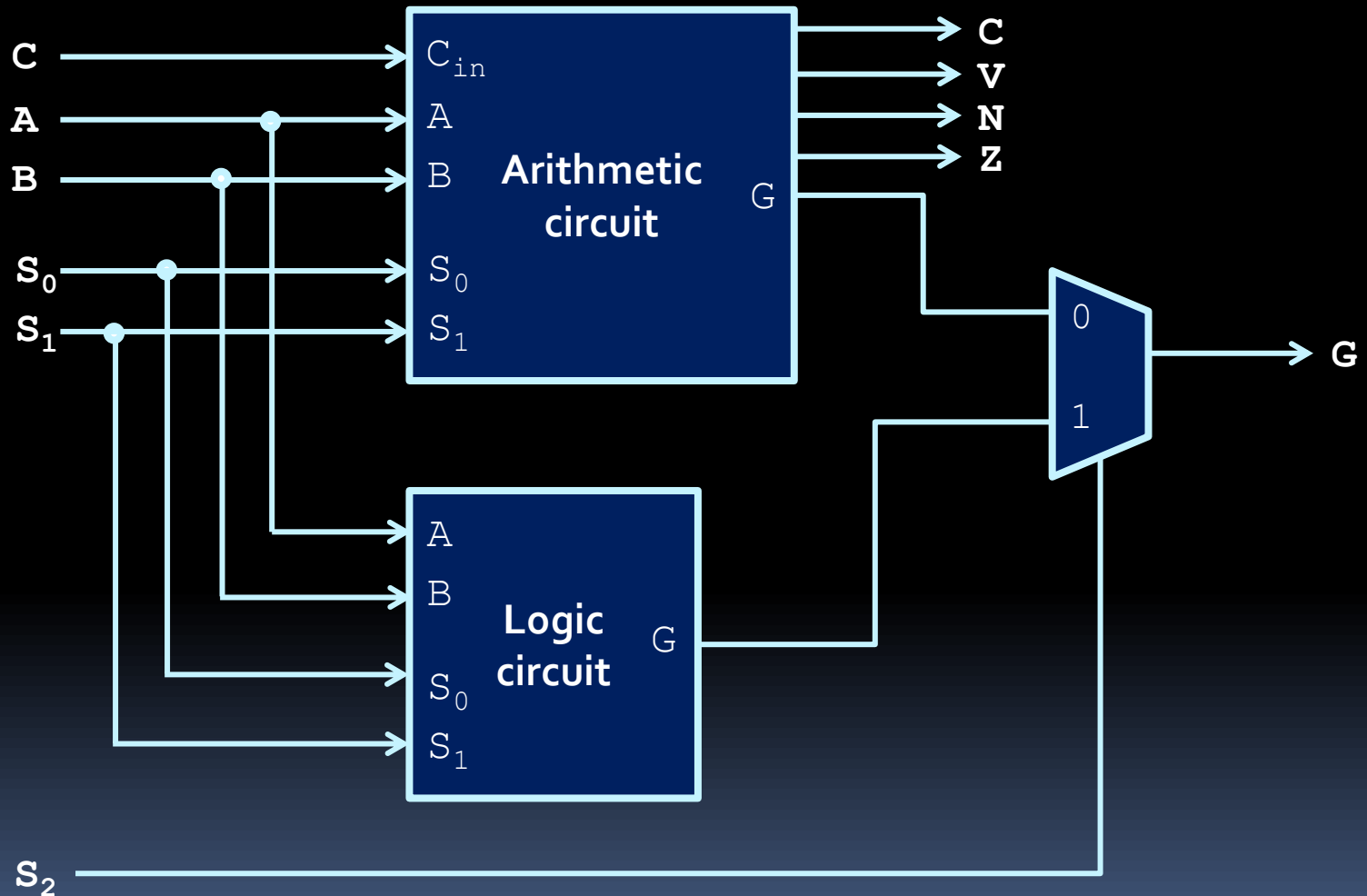


# Our Simple ALU

- **G** is the result.
- Output signals **V, C, N, Z** are flags: they indicate special conditions in the result:
  - **V: overflow condition**
    - 1 if the result of the operation could not be stored in the  $n$  bits of  $G$ , meaning that the result is incorrect.
  - **C: carry-out** bit (basically  $C_{out}$ )
  - **N: 1** if the result is **negative**
  - **Z: 1** if the result is **zero**



# ALU Diagram



# Overflow

- How do we know the result overflows?
- For **unsigned numbers**,  $C_{out}=1$  indicates overflow
- For **signed numbers**:
  - For  $G=A+B$ , overflow if and only if:
    - A and B have the same sign AND...
    - ... G has opposite sign.
  - For  $G=A-B$ , overflow if and only if:
    - A and B have different sign AND...
    - ... G has the same sign as B.

Overflow if...

(pos A) + (pos B)  $\rightarrow$  (neg G)

(neg A) + (neg B)  $\rightarrow$  (pos G)

Overflow if...

(pos A) - (neg B)  $\rightarrow$  (neg G)

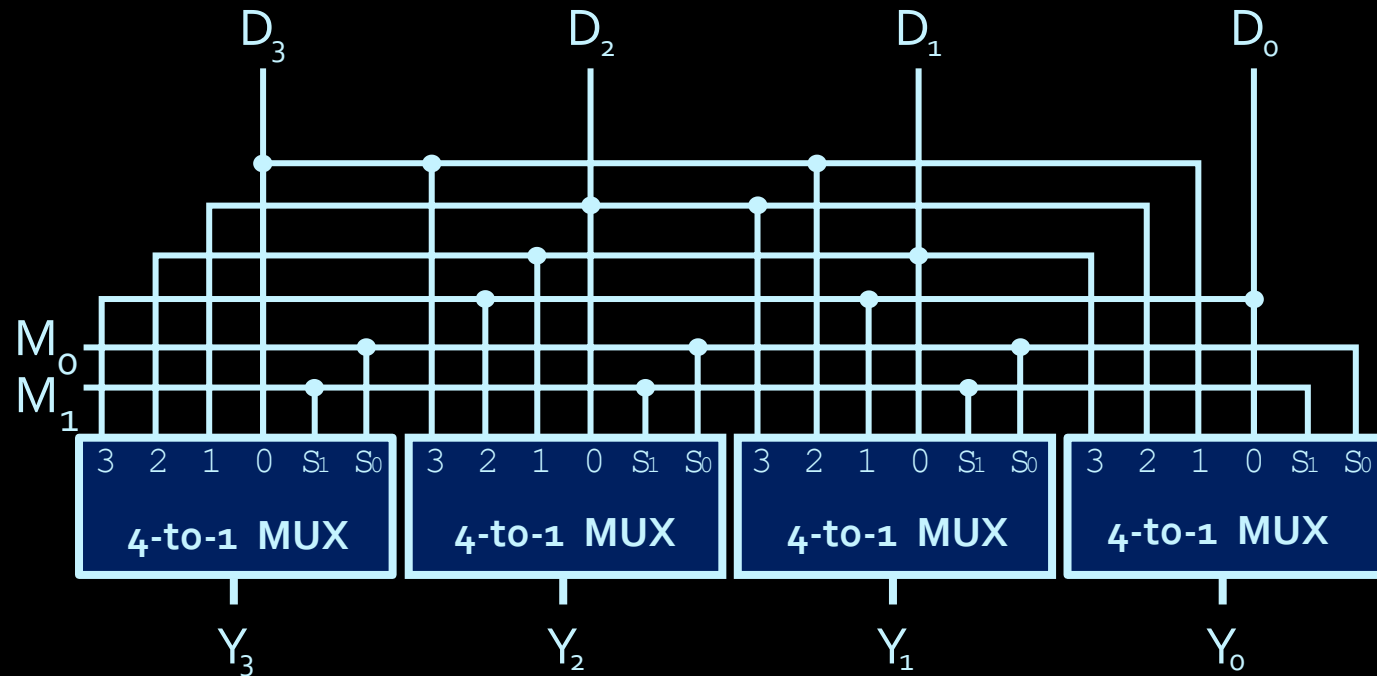
(neg A) - (pos B)  $\rightarrow$  (pos G)

# Barrel Shifter

- Early CPUs could only shift 1 bit every clock cycle.
- What if we want to shift by  $m$  bits?
  - Do a loop
- Shifting is important, so new CPUs have a **barrel shifter**.
  - Shift in one cycle...
  - ...by variable number of bits (determined by input)
- Disadvantage: cost (area)



# New Component: Barrel Shifter



- This barrel shifter **shifts and rotates**  $D$  to the left by  $S$  bits.
  - ▣ If  $M=1$  ( $M_1M_0$  is 01)  $\rightarrow Y = D_2D_1D_0D_3$  (a little like shift left by 1)
  - ▣ If  $M=3$  ( $M_1M_0$  is 11)  $\rightarrow Y = D_0D_3D_2D_1$  (a little like shift right by 1)
- Extra logic to convert rotate to shift (to zero bits in the result).
- A **purely combinational circuit** that works in **one cycle** (unlike shift registers).

# Multiplication

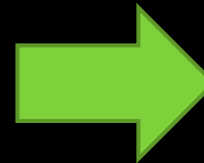
- Revisiting grade 3 math...

$$\begin{array}{r} 123 \\ \times 456 \\ \hline \end{array}$$

$$\begin{array}{r} 12 \phantom{3} \\ \times 456 \\ \hline 1368 \phantom{00} \end{array}$$

$$\begin{array}{r} 1 \phantom{2} 3 \\ \times 456 \\ \hline 1368 \\ 912 \phantom{00} \end{array}$$

$$\begin{array}{r} 1 \phantom{2} 3 \\ \times 456 \\ \hline 1368 \\ 912 \\ 456 \phantom{00} \end{array}$$

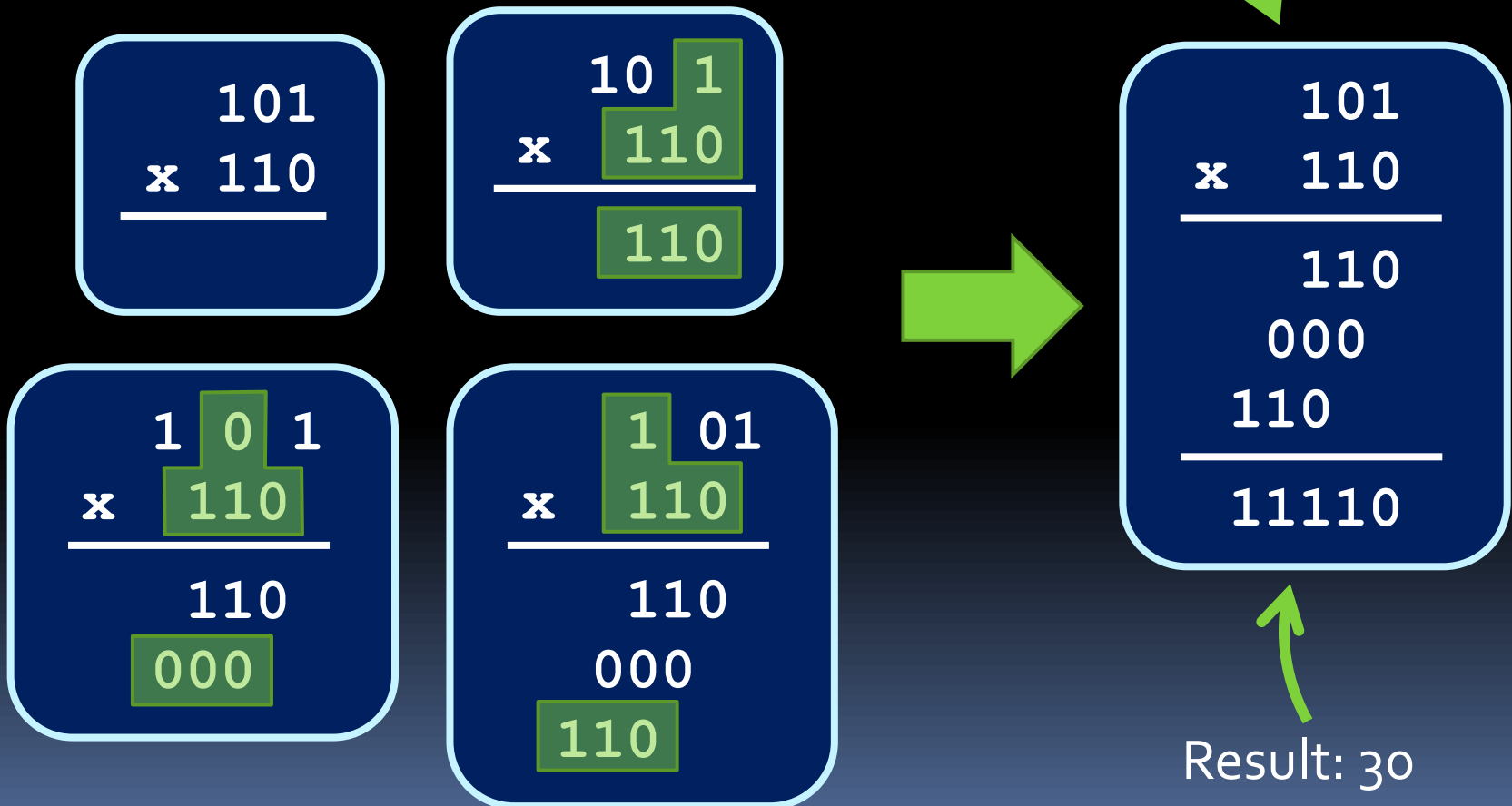


$$\begin{array}{r} 123 \\ \times 456 \\ \hline 1368 \\ 912 \\ 456 \\ \hline 56088 \end{array}$$

# Binary Multiplication

- And now, in binary...

5\*6 (unsigned)



# Binary Multiplication

- Or seen another way....

$$\begin{array}{r} 101 \\ \times 110 \\ \hline \end{array}$$

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 000 \end{array}$$

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 000 \\ 101 \end{array}$$

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 000 \\ 101 \\ 101 \end{array}$$



$$\begin{array}{r} 101 \\ \times 110 \\ \hline 000 \\ 101 \\ 101 \\ \hline 11110 \end{array}$$

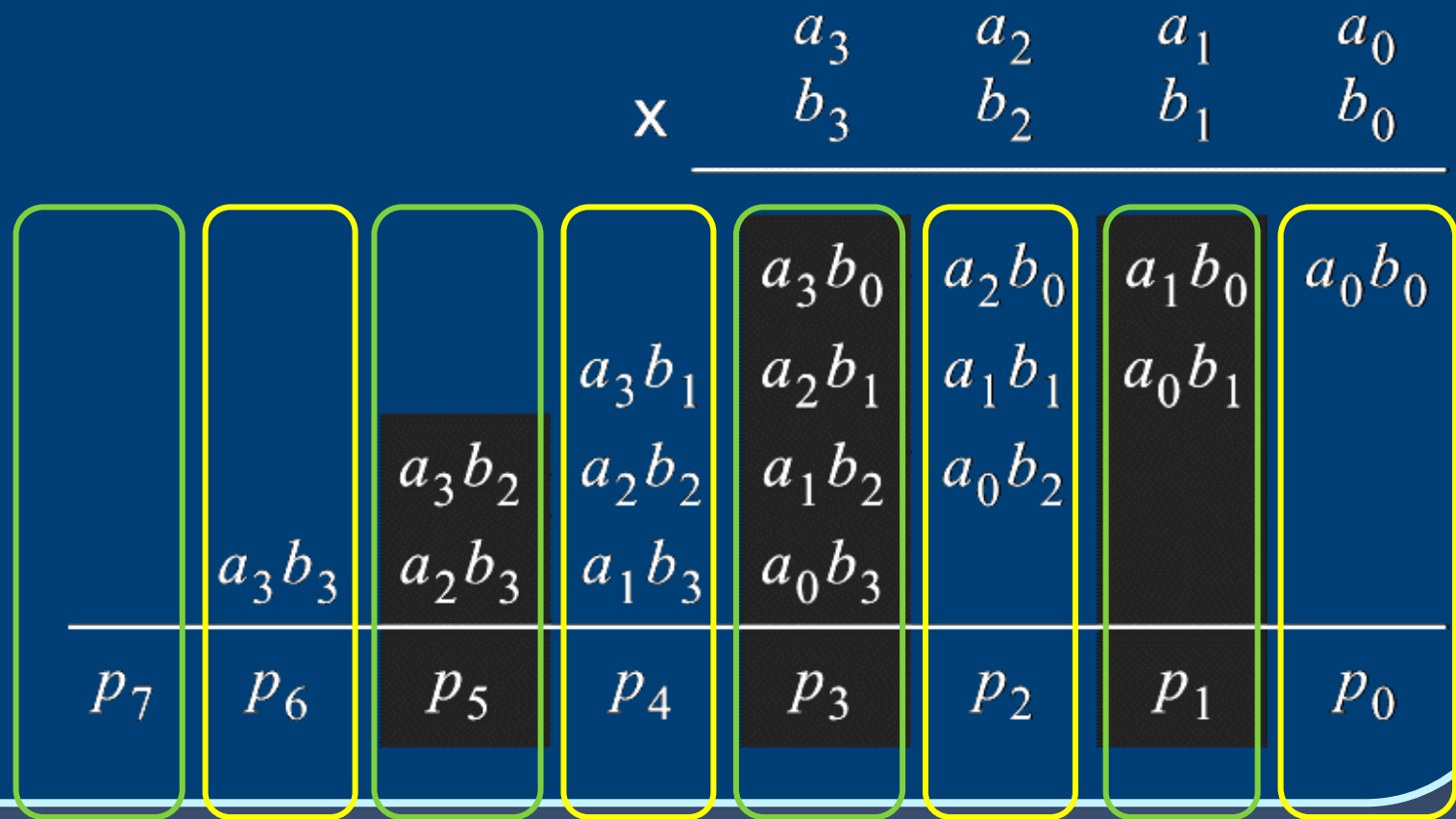
# Array Multiplier

				$\times$	$a_3$ $b_3$	$a_2$ $b_2$	$a_1$ $b_1$	$a_0$ $b_0$
					$a_3b_0$	$a_2b_0$	$a_1b_0$	$a_0b_0$
			$a_3b_1$		$a_2b_1$	$a_1b_1$	$a_0b_1$	
		$a_3b_2$	$a_2b_2$		$a_1b_2$	$a_0b_2$		
	$a_3b_3$	$a_2b_3$	$a_1b_3$		$a_0b_3$			
$p_7$	$p_6$	$p_5$	$p_4$	$p_3$	$p_2$	$p_1$	$p_0$	

# Array Multiplier

				x	$a_3$ $b_3$	$a_2$ $b_2$	$a_1$ $b_1$	$a_0$ $b_0$
			$a \times b_0 \rightarrow$		$a_3b_0$	$a_2b_0$	$a_1b_0$	$a_0b_0$
		$a \times b_1 \rightarrow$	$a_3b_1$		$a_2b_1$	$a_1b_1$	$a_0b_1$	
	$a \times b_2 \rightarrow$	$a_3b_2$	$a_2b_2$		$a_1b_2$	$a_0b_2$		
$a \times b_3 \rightarrow$	$a_3b_3$	$a_2b_3$	$a_1b_3$	$a_0b_3$				
	$p_7$	$p_6$	$p_5$	$p_4$	$p_3$	$p_2$	$p_1$	$p_0$

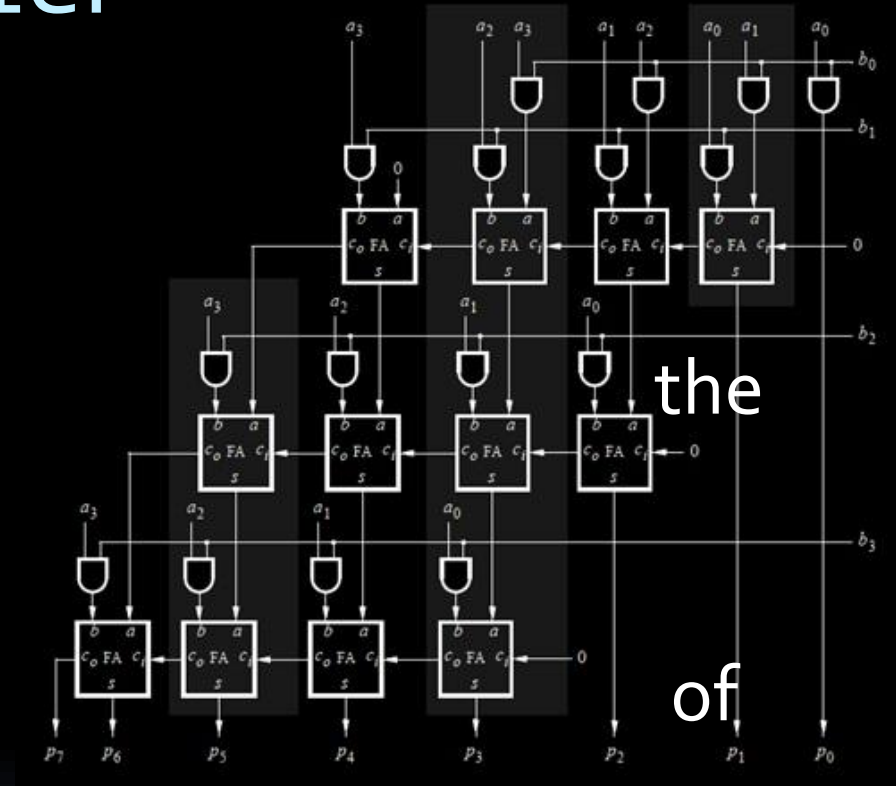
# Array Multiplier



$$p_7 = \sum \quad p_6 = \sum \quad p_5 = \sum \quad p_4 = \sum \quad p_3 = \sum \quad p_2 = \sum \quad p_1 = \sum \quad p_0 = \sum \dots$$

# Array Multiplier

- This implementation results in an array of adder circuits to make multiplier circuit.
- This can get a little expensive as the size of the operands grows.
  - N-bit numbers  $\rightarrow O(1)$  clock cycles, but  $O(N^2)$  size.
- Is there an alternative to this circuit?





# Accumulator circuits

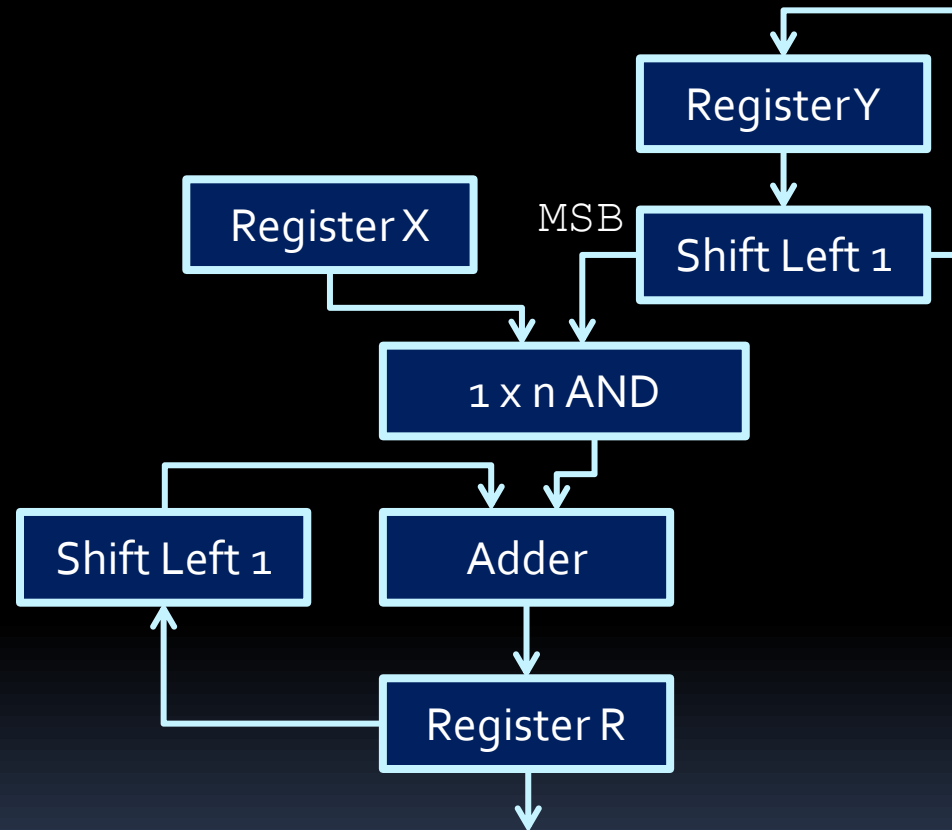
- We can implement the same algorithm in hardware:
  - Use shift register and check the MSB to get bits of **101**
  - Use shift register with parallel load to store and shift **R**
    - Or use load register with shifter circuit
  - Use adder to compute:  
 $R = R + 110 \times 1$

A hand-drawn style binary multiplication diagram on a dark blue background with rounded corners. It shows the multiplication of 101 by 110. The multiplier 101 is at the top, and the multiplicand 110 is below it, preceded by an 'x' symbol. A horizontal line separates the multiplicand from the partial products. The first partial product is 11000, where the first three digits are green. The second partial product is 0000, where the first three digits are pink. The third partial product is 110, where the first three digits are yellow. A second horizontal line separates the partial products from the final result, 11110.

$$\begin{array}{r} \phantom{x} 101 \\ x \phantom{00} 110 \\ \hline 11000 \\ 0000 \\ 110 \\ \hline 11110 \end{array}$$

# Accumulator circuit

- This circuit needs only a single row of adders and a couple of shift registers.
- But  $O(n)$  cycles
- How wide does register R have to be?
  - When multiplying  **$n$ -bit** number by  **$k$ -bit** number, result may need up to  **$n+k$  bits**.



# Booth's Algorithm

- In real life we often see sequences of bits: 00111000
- Designed to take advantage of the fact that in circuits, **shifting is cheaper than adding** and space is at a premium.
  - Based on the premise that when multiplying by certain values (e.g. 99), it can be easier to think of this operation as a difference between two products.
- Consider the shortcut method when multiplying a given decimal value  $X$  by 9999:
  - $X * 9999 = X * 10000 - X * 1$
- Now consider the equivalent problem in binary:
  - $X * 001111 = X * 010000 - X * 1$

# Booth's Algorithm

- This idea is applied when two neighboring digits in an operand are different.
- Go through digits from  $n-1$  to  $0$ 
  - If digits at  $i$  and  $i-1$  are  $0$  and  $1$ , the multiplicand is added to the result at position  $i$ .
  - If digits at  $i$  and  $i-1$  are  $1$  and  $0$ , the multiplicand is subtracted from the result at position  $i$ .
- The result is always a value whose size is the sum of the sizes of the two multiplicands.

# Booth's Algorithm

- Steps in Booth's Algorithm:
  1. Designate the two multiplicands as A & B, and the result as some product P.
  2. Add an extra zero bit to the right-most side of A.
  3. Repeat the following for each original bit in A:
    - a) If the last two bits of A are the same, do nothing.
    - b) If the last two bits of A are 01, then add B to the highest bits of P.
    - c) If the last two bits of A are 10, then subtract B from the highest bits of P.
    - d) Perform one-digit arithmetic right-shift on both P and A.
  4. The result in P is the product of A and B.

# Question

- Use Booth's Algorithm to find  $14d * -5d$ , in binary
- We need  
5 + 5 =  
10 bits

# Question

- Use Booth's Algorithm to find  $14d * -5d$ , in binary
- We need  $5 + 5 = 10$  bits

```
01110      A = 01110
x 11011    B = 11011
          -B = 00101
```

A:            011100 ("virtual zero")

```
-----
      ↓      ↓
  1110110000  add +B (sign extend)
+0000001010  add -B (sign extend)
-----
  1110111010
```

- → -70

# Question

- Swap A and B
- Remember:  
we add a  
“virtual  
zero” to  
the right  
of A
  - No such  
zero on  
the left  
however!



# Question

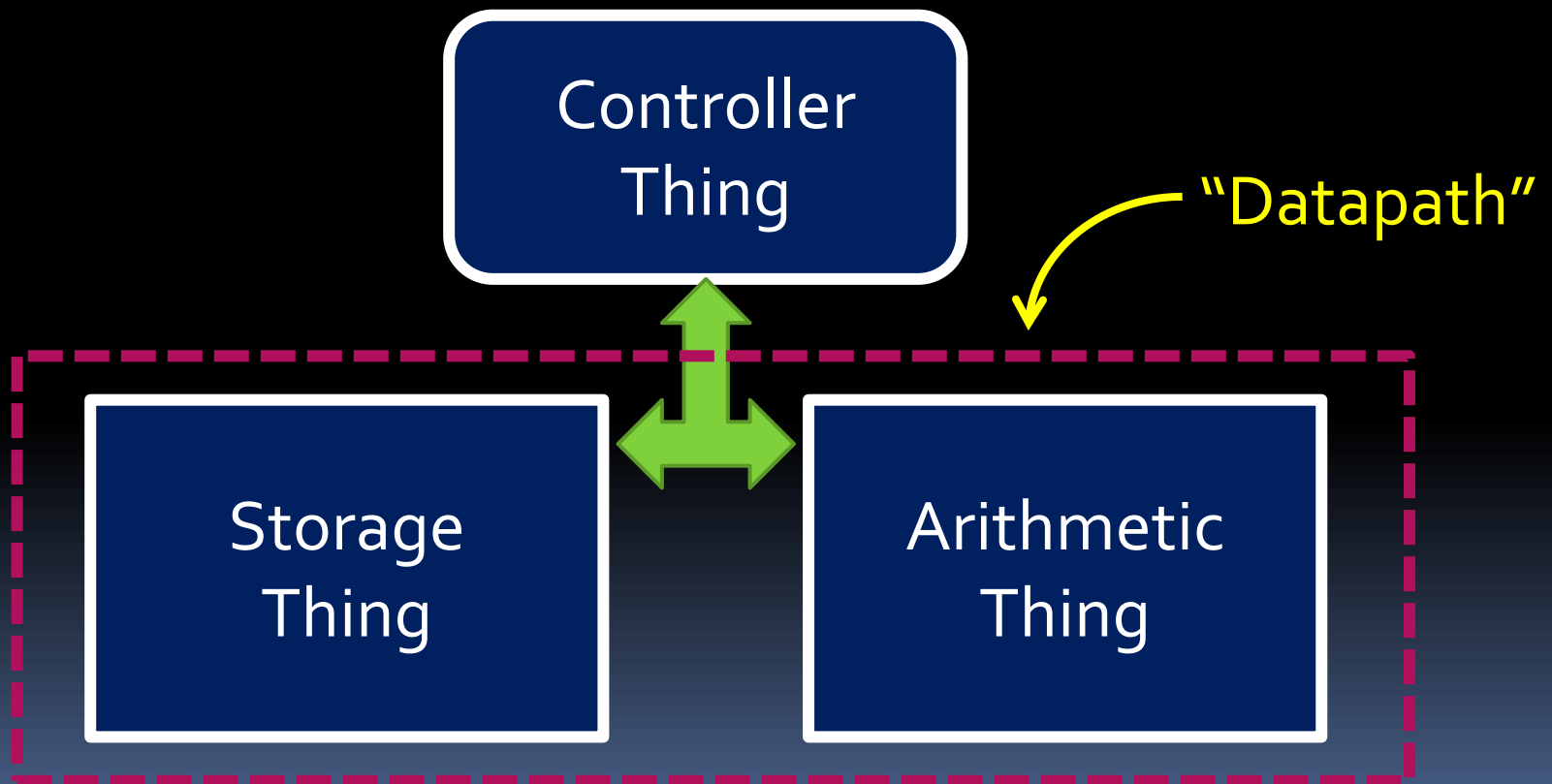
- Swap A and B
- Remember:  
we add a  
“virtual  
zero” to  
the right  
of A
  - No such  
zero on  
the left  
however!
- → -70

```
      11011      A = 11011
x   01110      B = 01110
          -B = 10010

A:      110110 ("virtual zero")
      -----
           ↓ ↓ ↓
      1110010    add -B (sign extend)
      00001110   add  B (sign extend)
      1111110010 add -B (sign extend)
      -----
      1110111010
```

# Datapath

- Where and how does computation happen?



# Datapath vs. Control

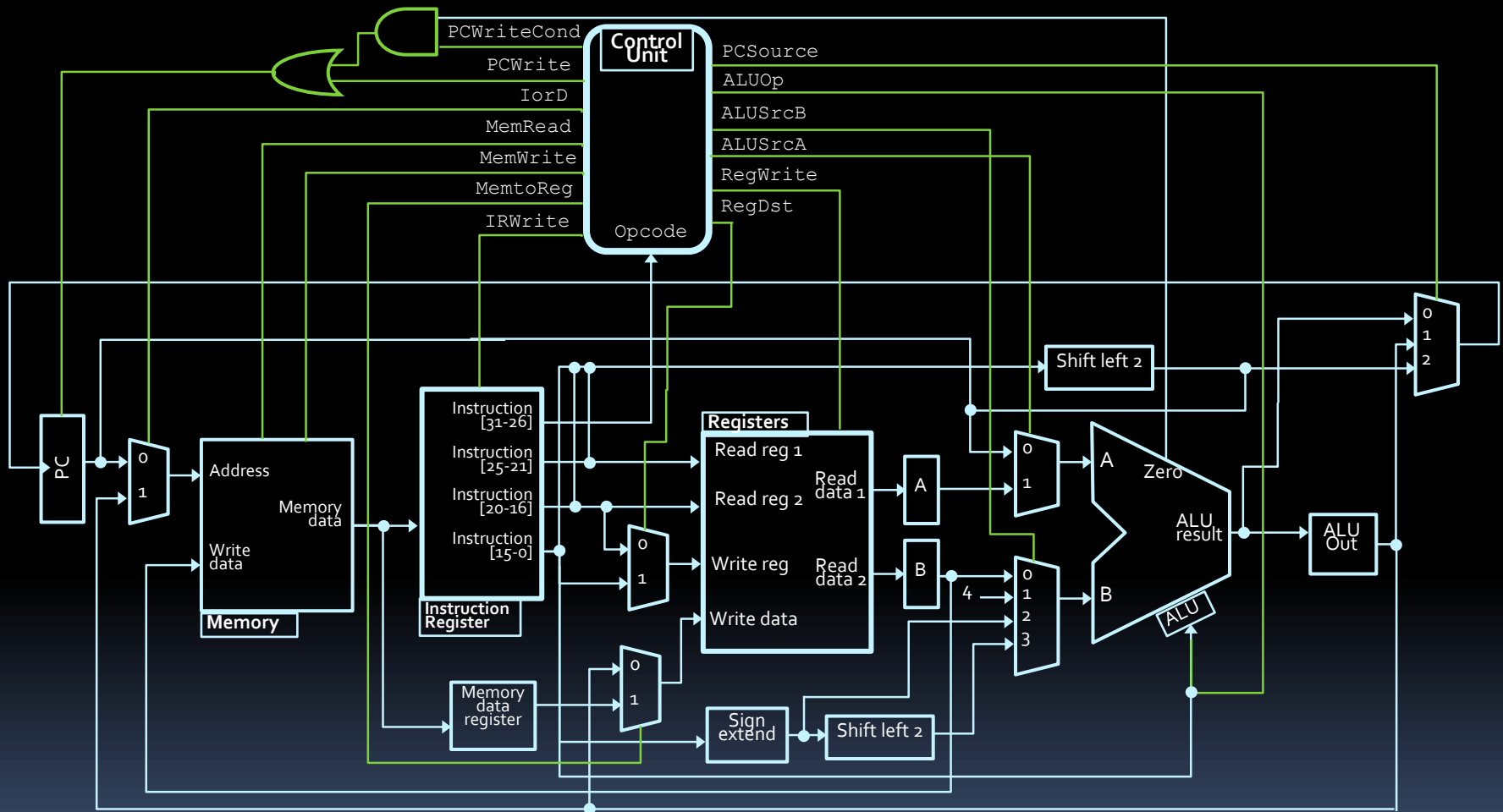
- **Datapath:** where all data computations take place.
  - Usually: registers, computational units, and a bunch of wires and muxes to connect them
- **Control unit:** orchestrates the actions that take place in the datapath.
  - The control unit is a big finite-state machine that instructs the datapath to perform all appropriate actions.

# Week 7:

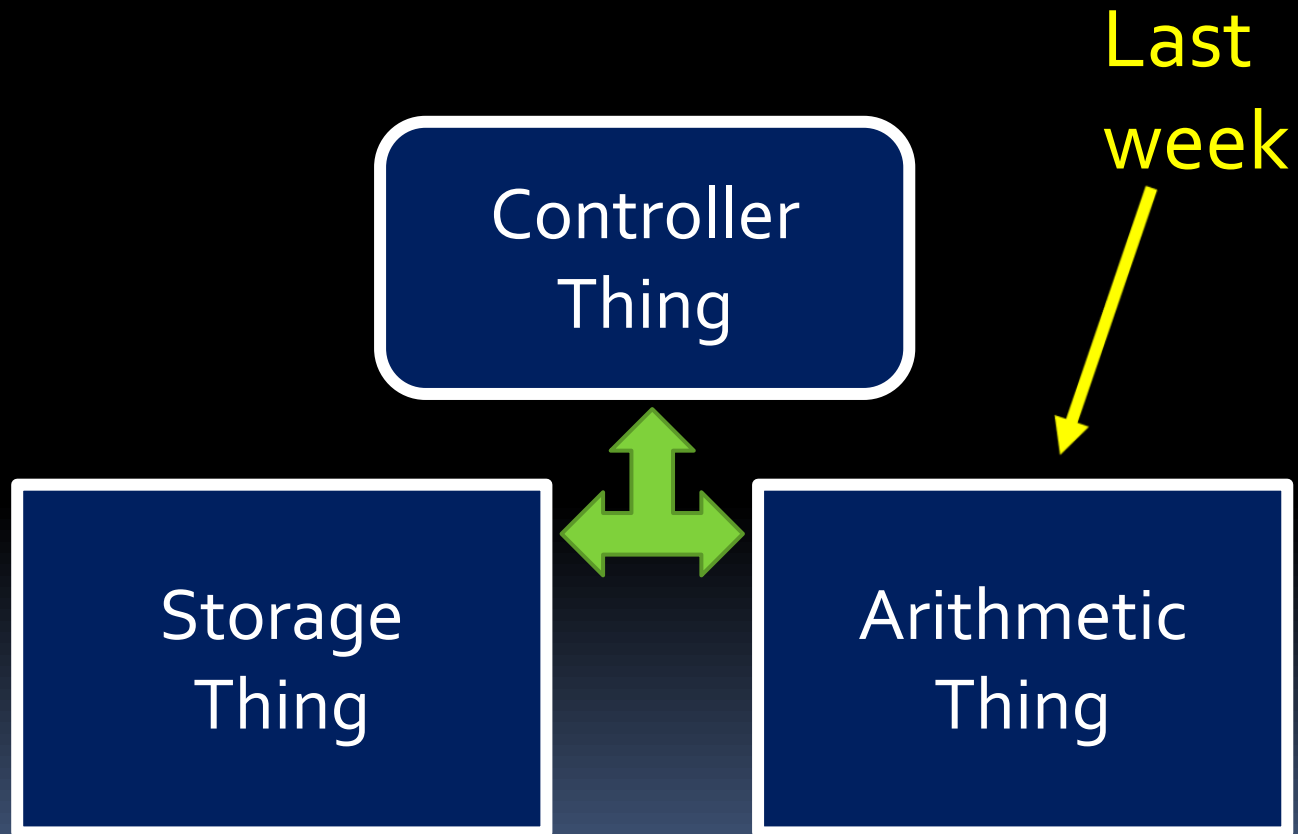
# Processor Storage

# and Control

# Building a Microprocessor



# Deconstructing processors



# The “Storage Thing”

aka: the register file and main memory

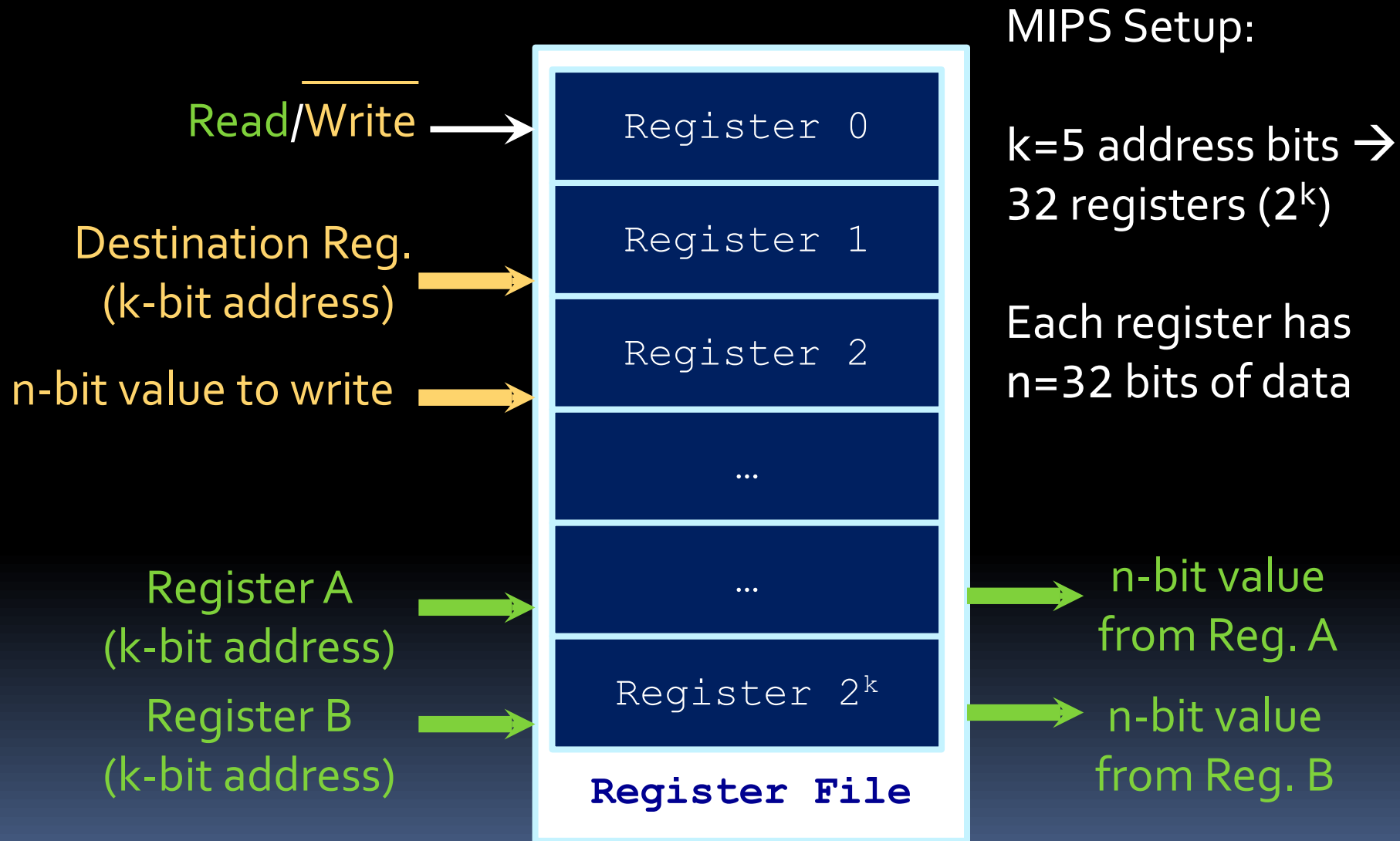


# Memory and registers

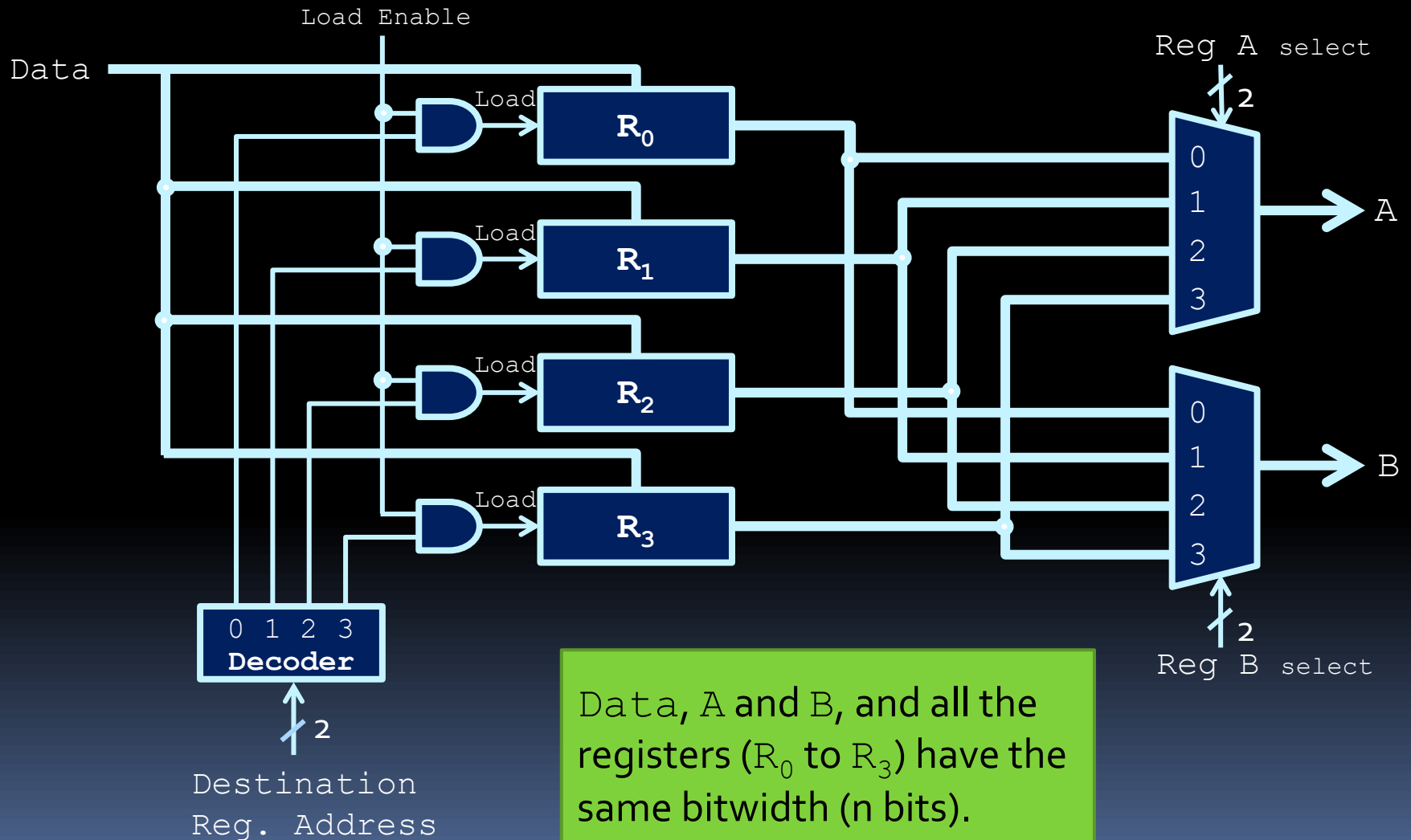
- The CPU has registers that store a single value (program counters, instruction registers, etc.)
- There are also units in the CPU that store large amounts of data for use by the CPU:
- **Register file**: Small number of fast memory units.
  - Allows multiple values to be read and written simultaneously.
- **Main memory**: Larger grid of slower memory cells.
  - Used to store the main information to be processed by the CPU.



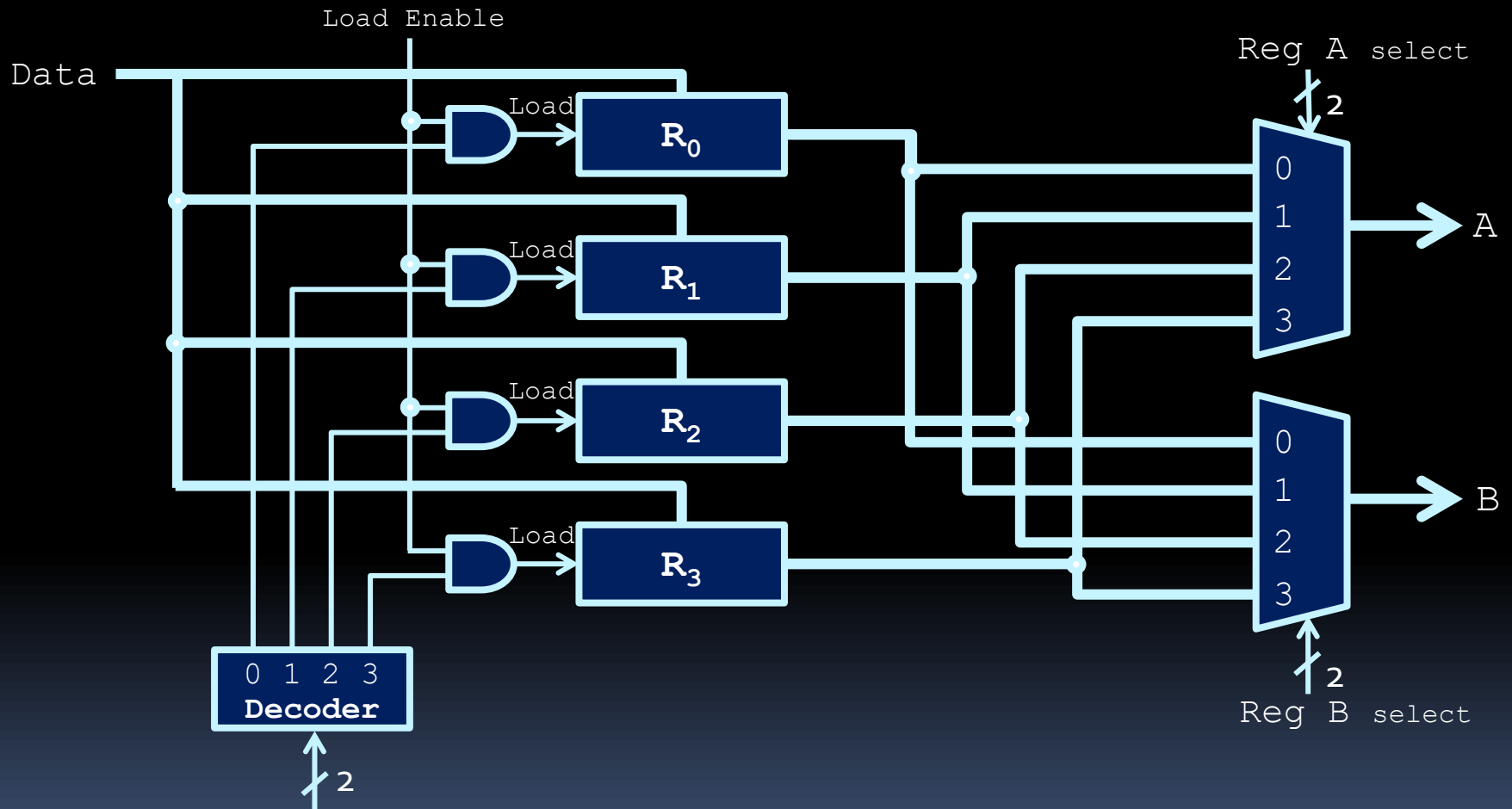
# Register File Functionality



# Register File Structure

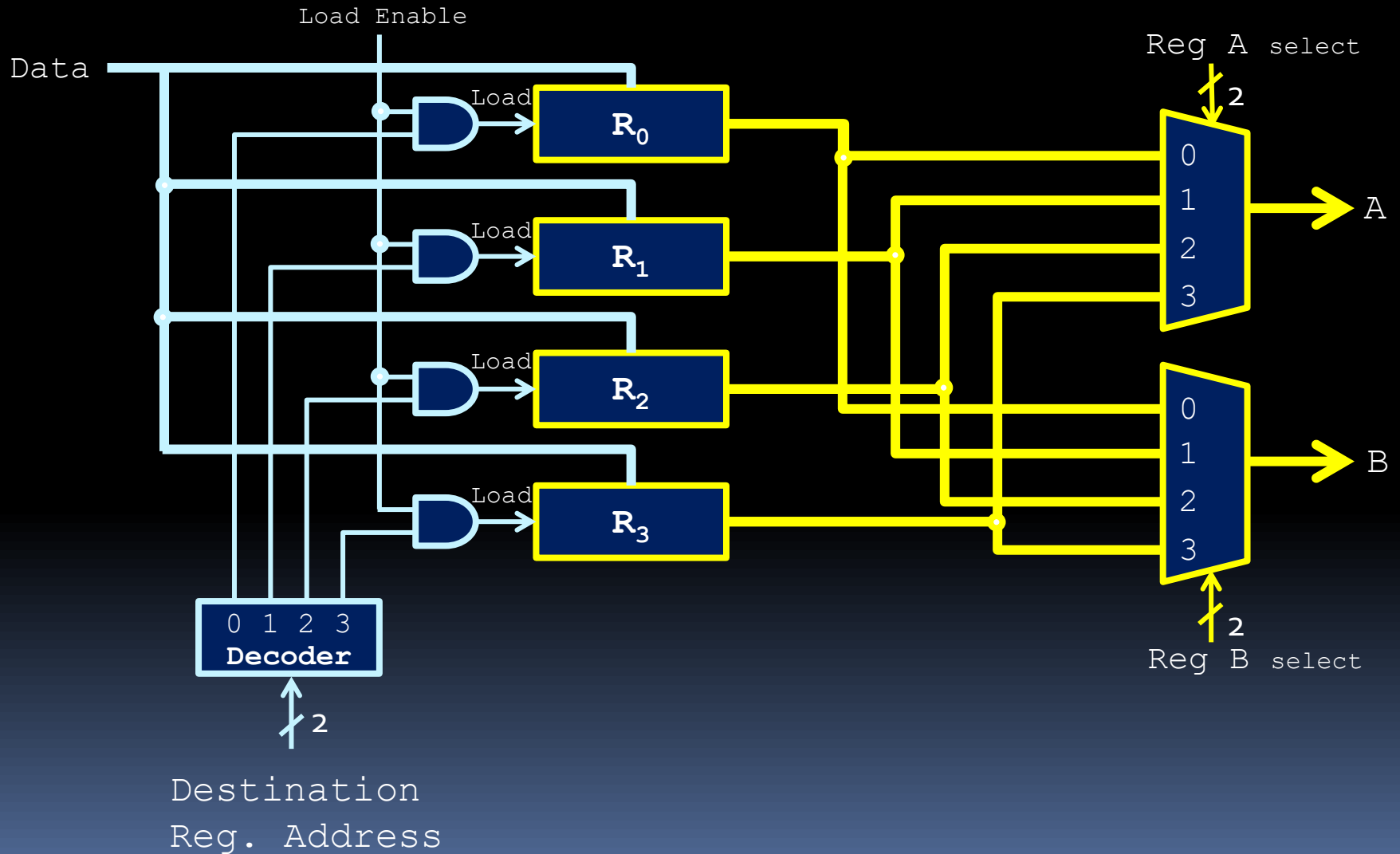


# Register File – Read Operation

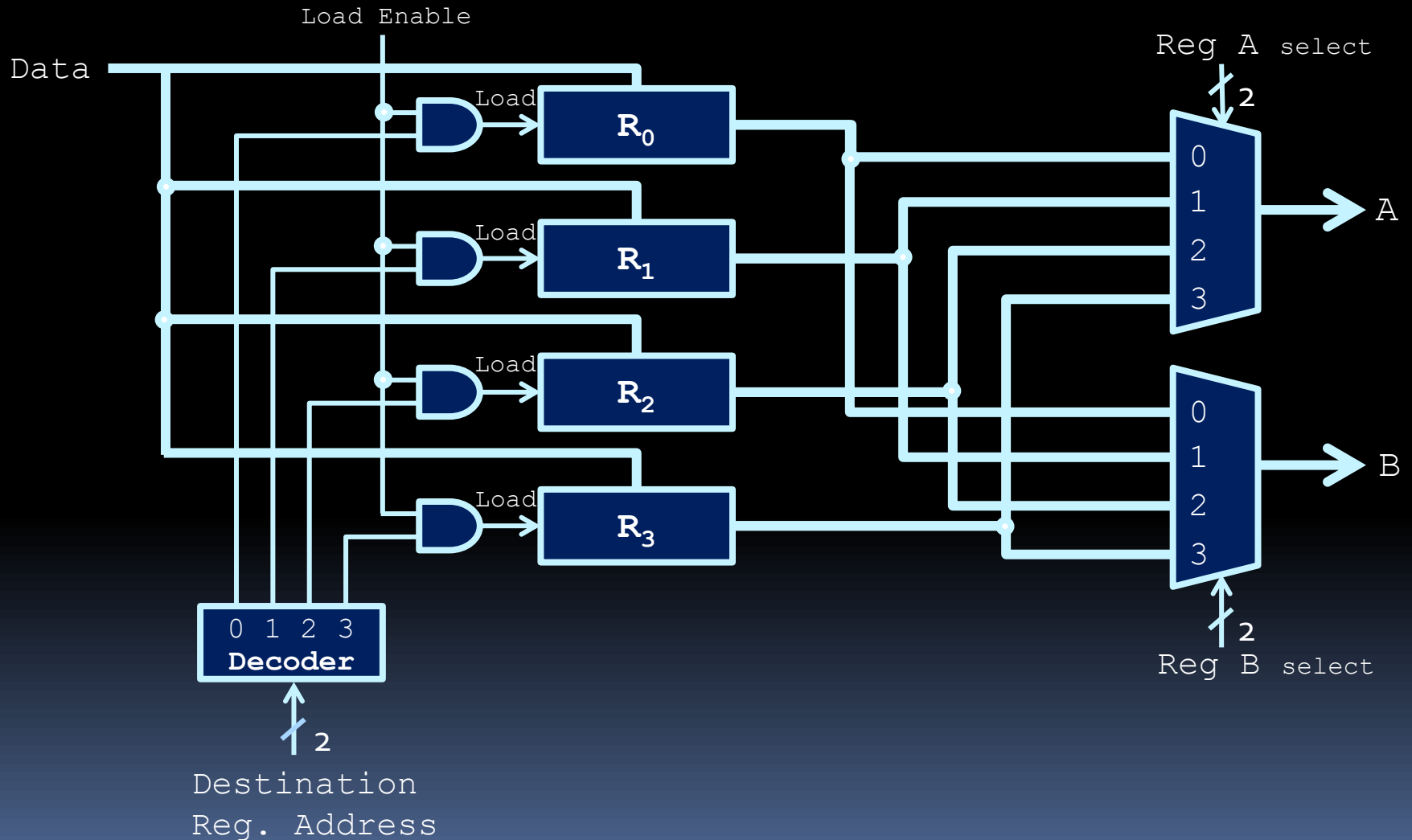


Destination  
Reg. Address

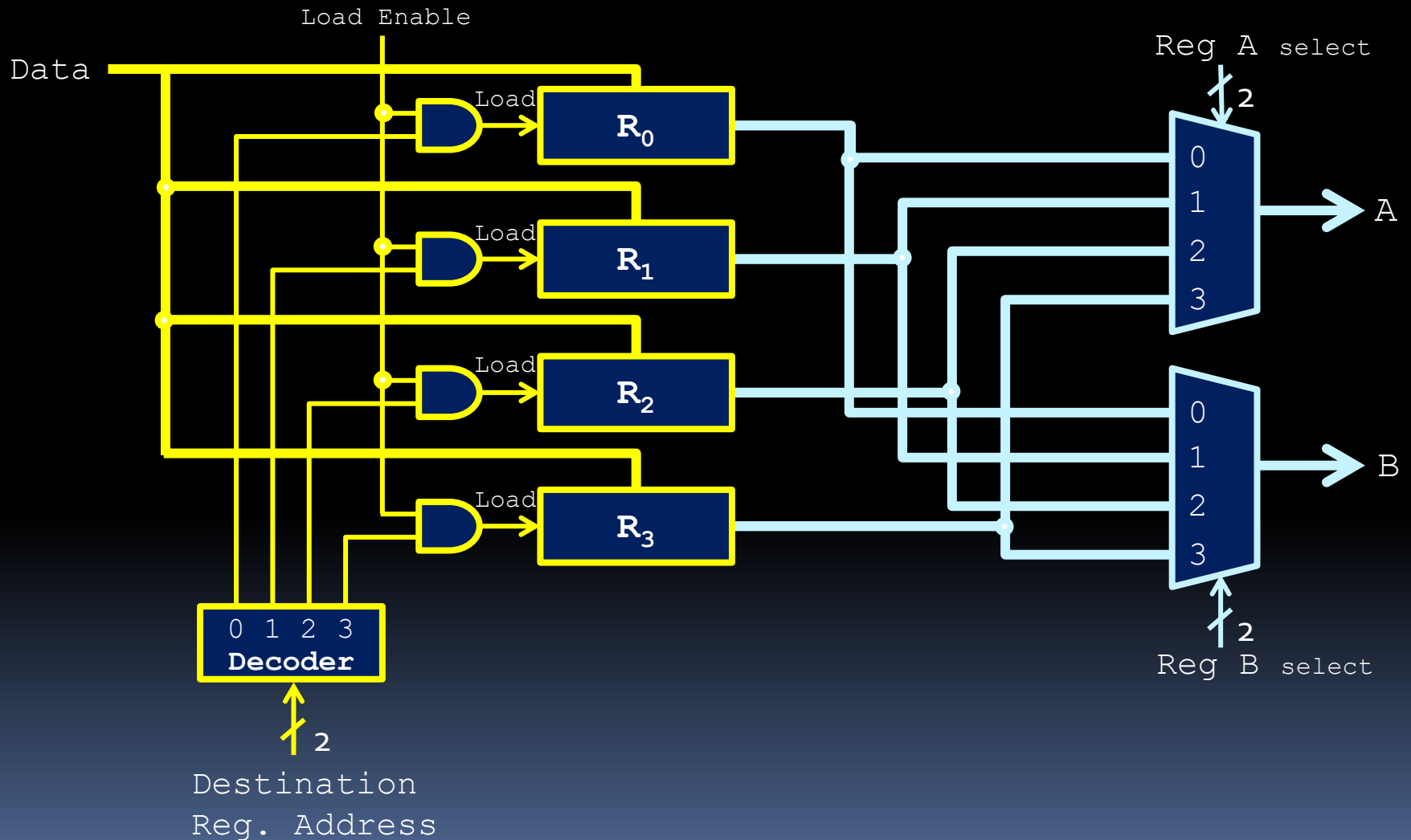
# Register File – Read Operation



# Register File - Write Operation



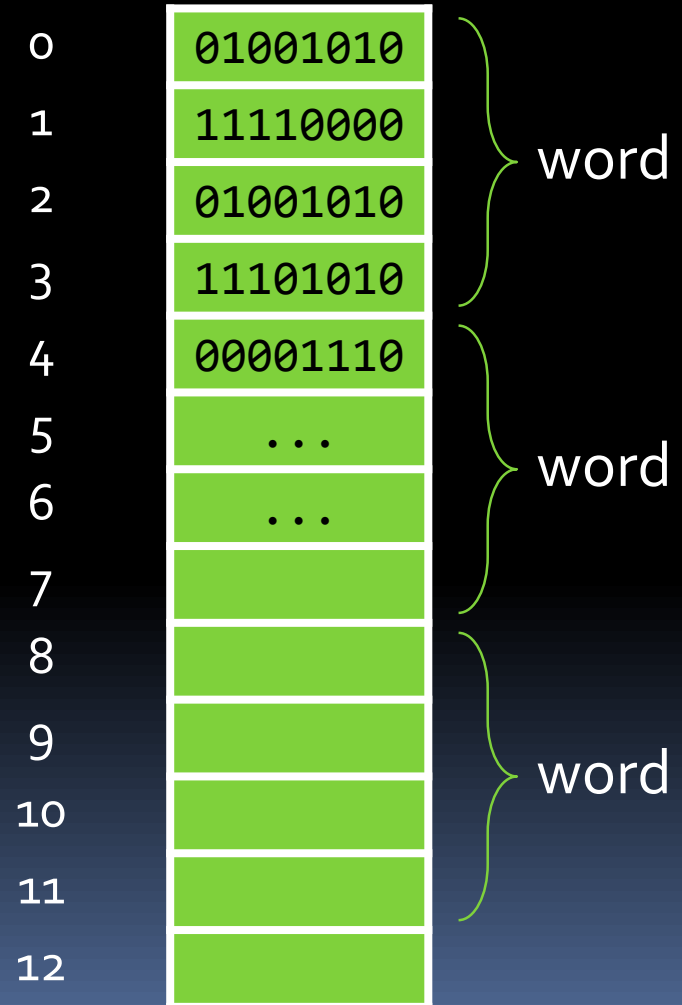
# Register File - Write Operation



# Main Memory and Addressing

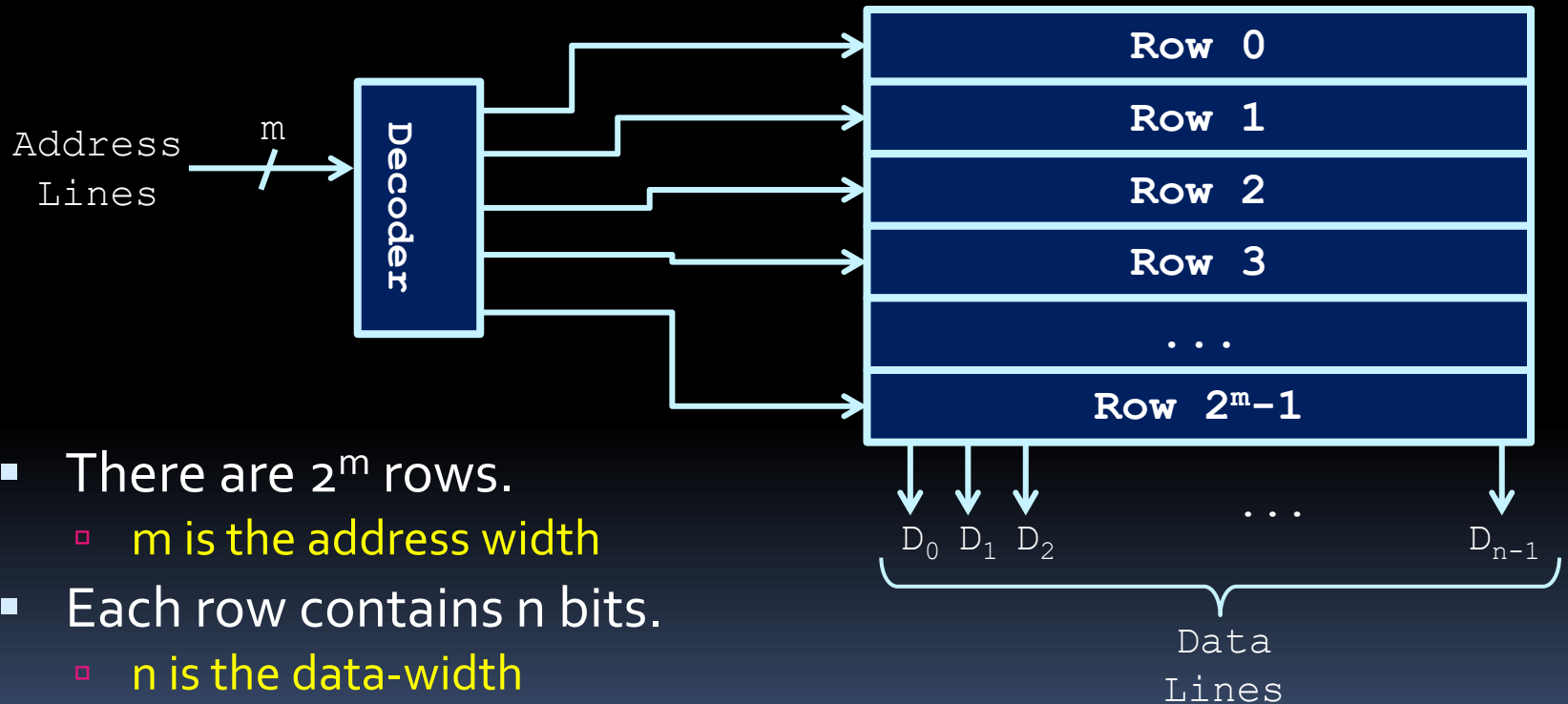
- Registers files are fast but too costly for storing lots of data.
- Instead store data in main memory.
- Main memory is **addressed in units of bytes** (8-bits)
- Every group of 4 bytes is one 32-bit **word**.

Address



# Electronic Memory

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

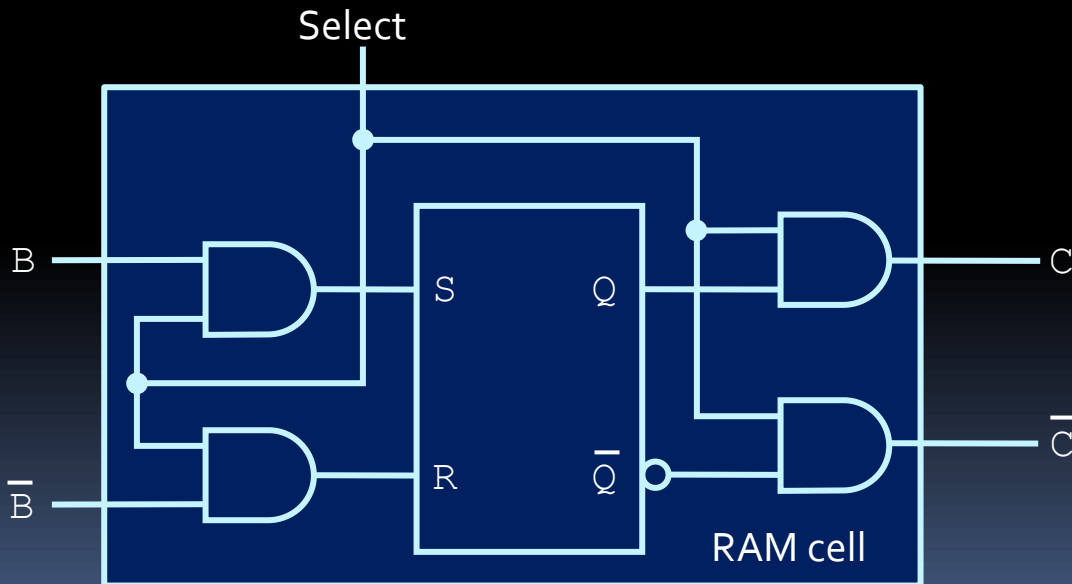


- There are  $2^m$  rows.
  - $m$  is the address width
- Each row contains  $n$  bits.
  - $n$  is the data-width
- What's the size of this memory?
  - $2^m * n$  bits  $\Rightarrow 2^m * n / 8$  Bytes

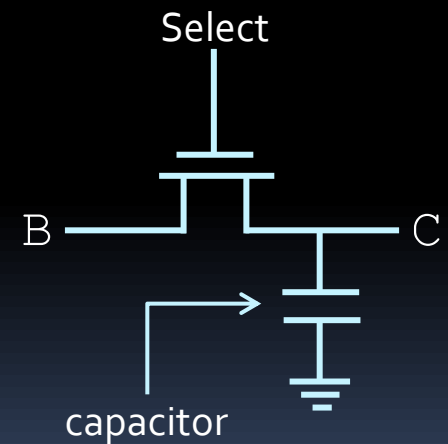


# Storage cells

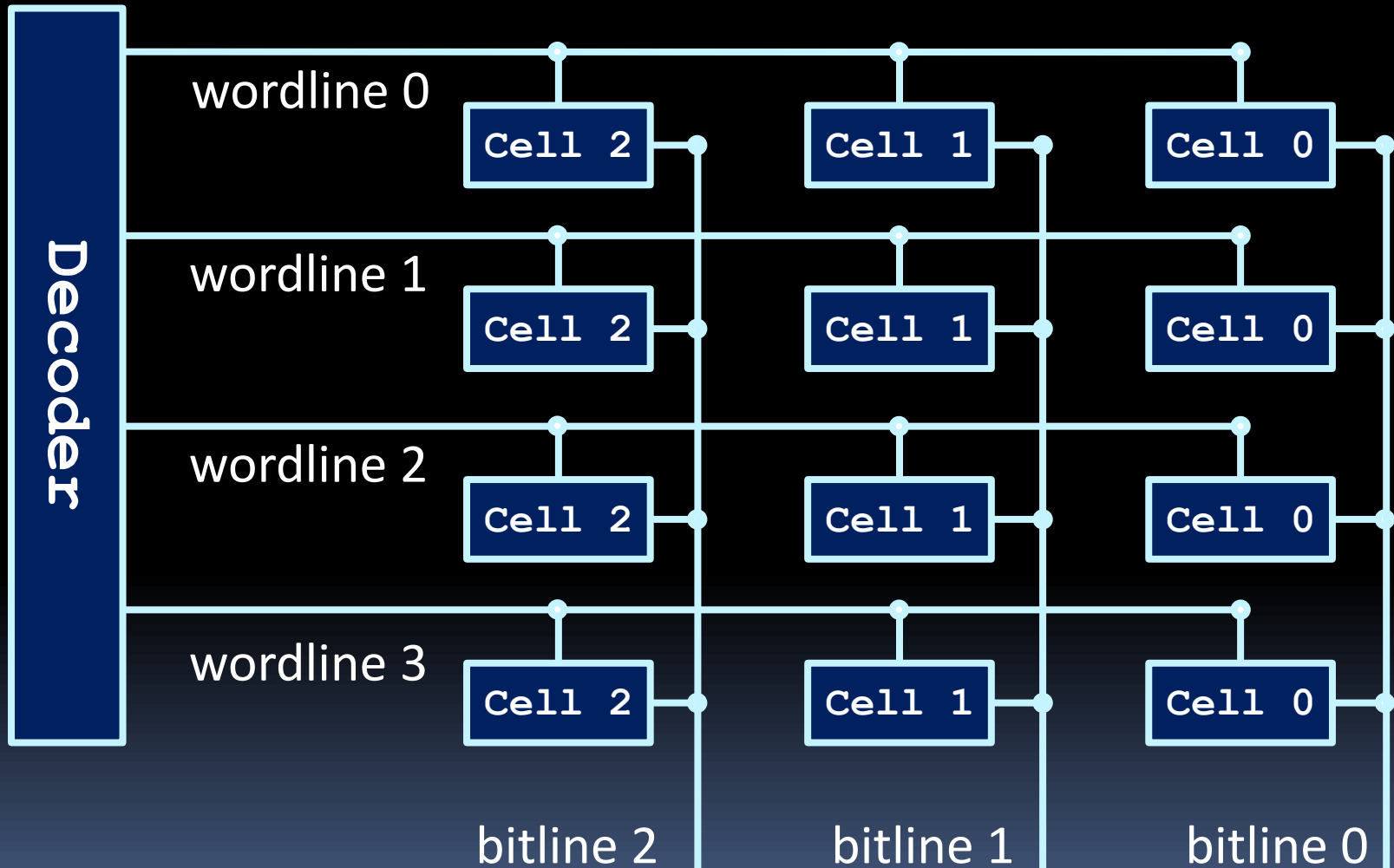
- Each row is made of  $n$  storage cells.
  - Each cell stores a single bit of information.
- Multiple ways of building these cells.
  - e.g. RAM cell



DRAM IC cell



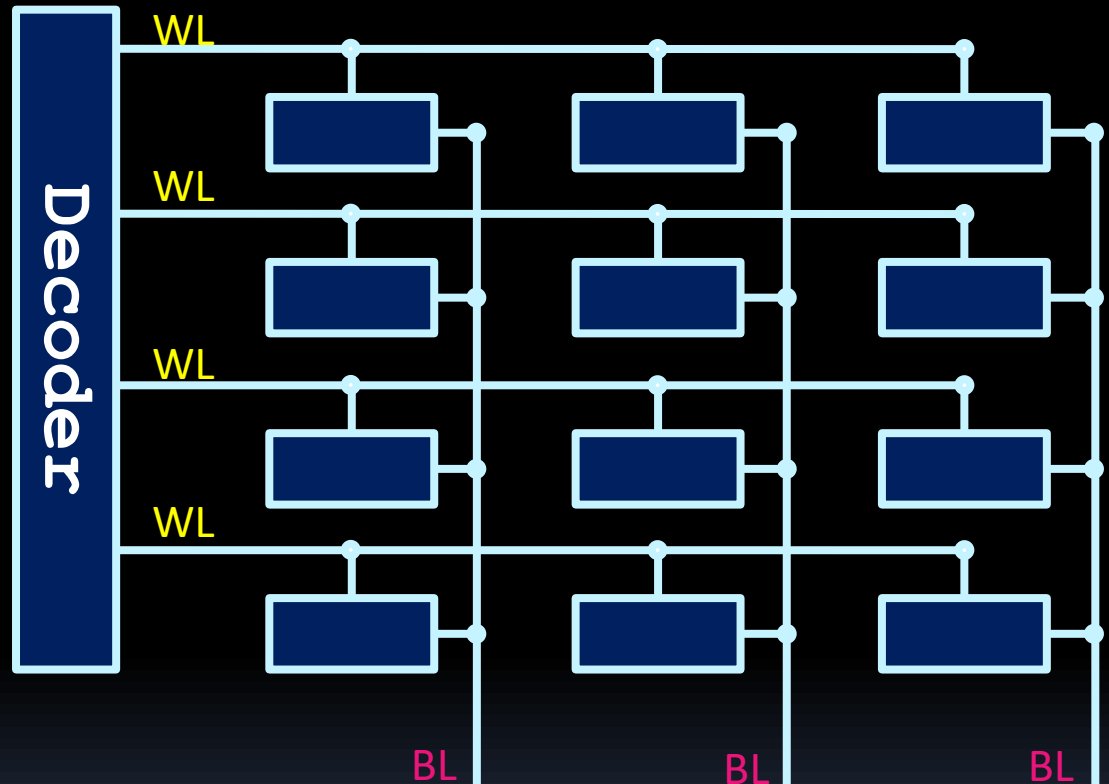
# Memory Array



# Memory Array – Main Signals

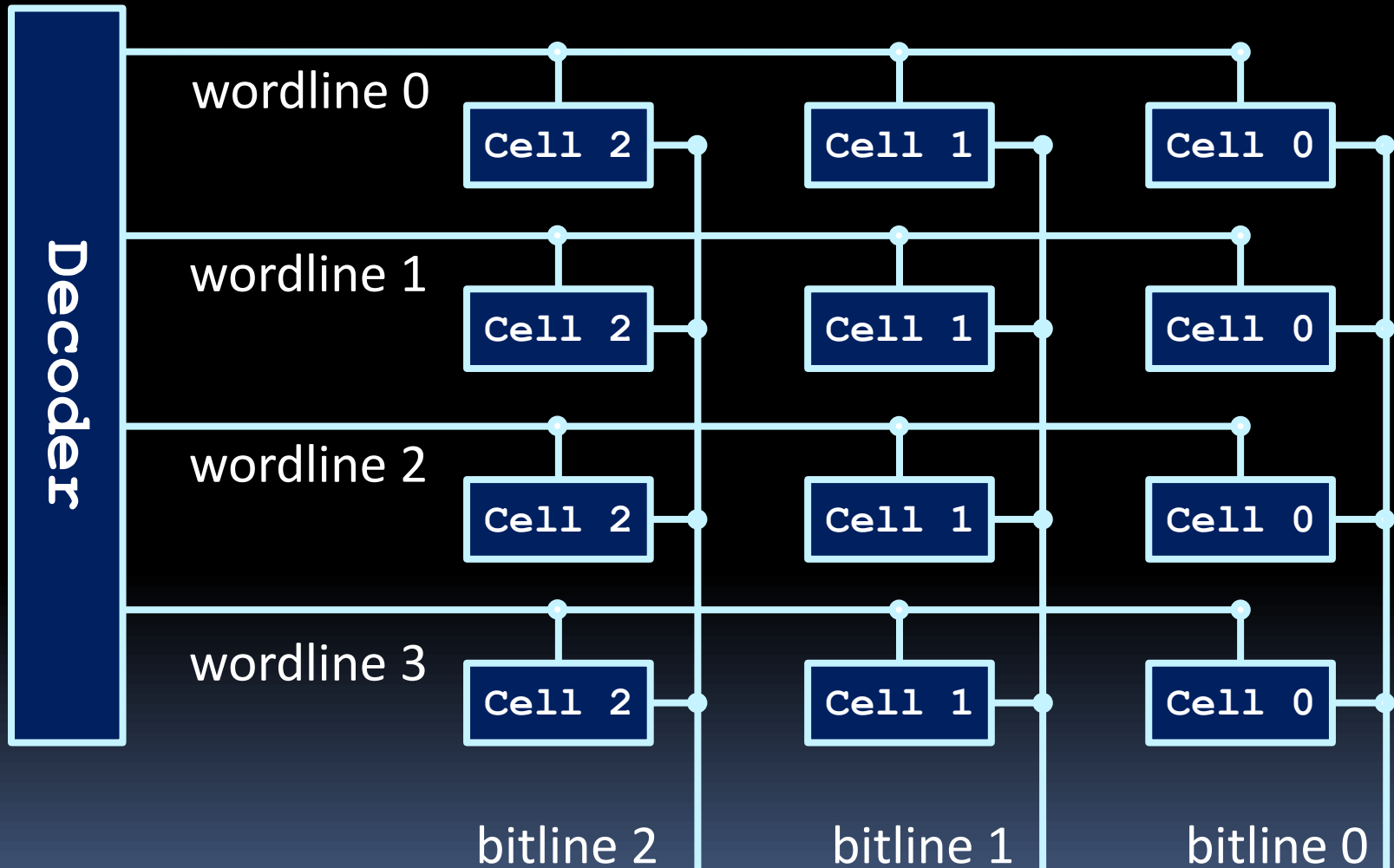
**Wordline:**  
which memory  
row (word) to  
read/write

**Bitline:**  
read/write data

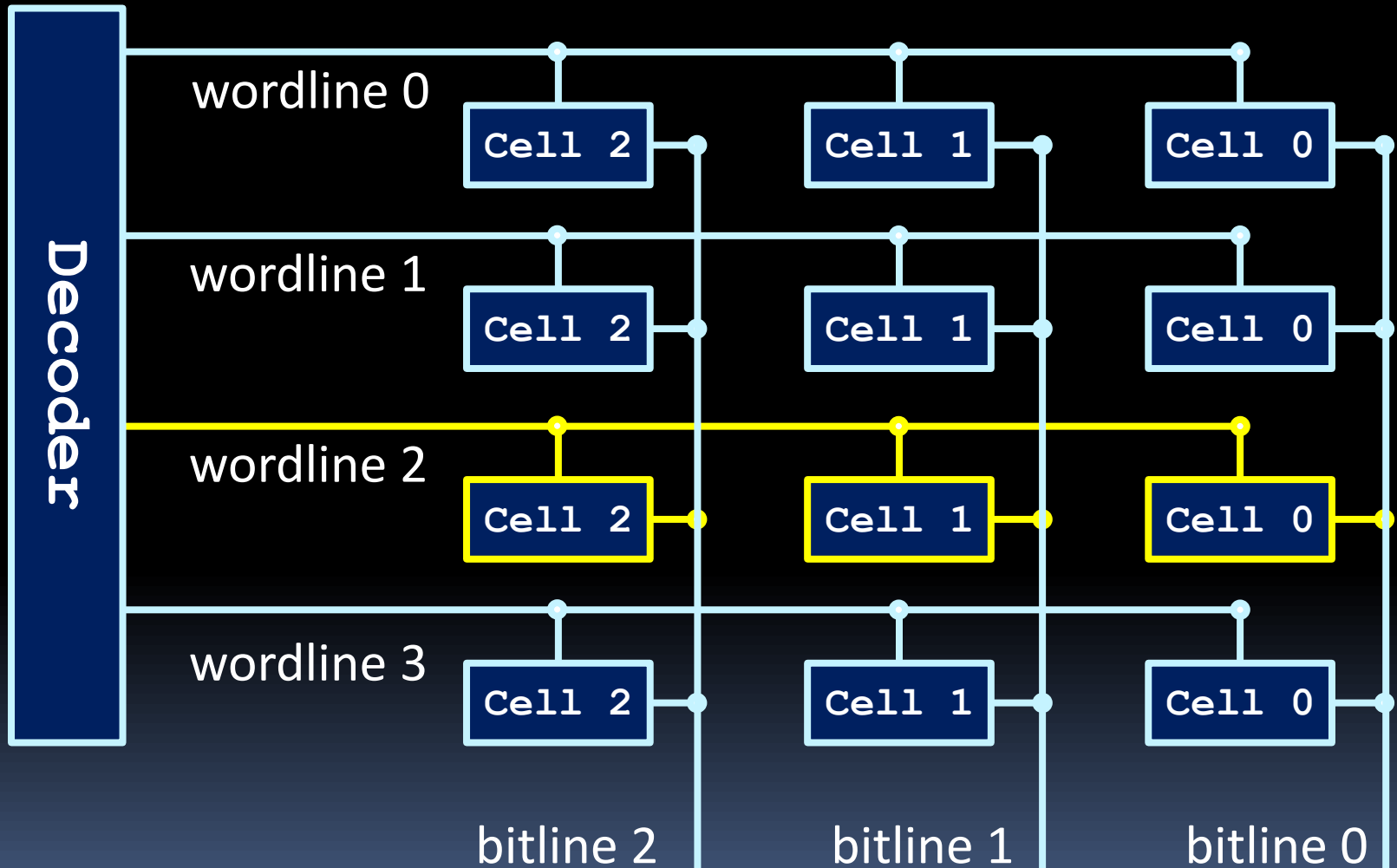


Also **read/write** signal: are we reading or writing?  
Can add memory enable, column select line.

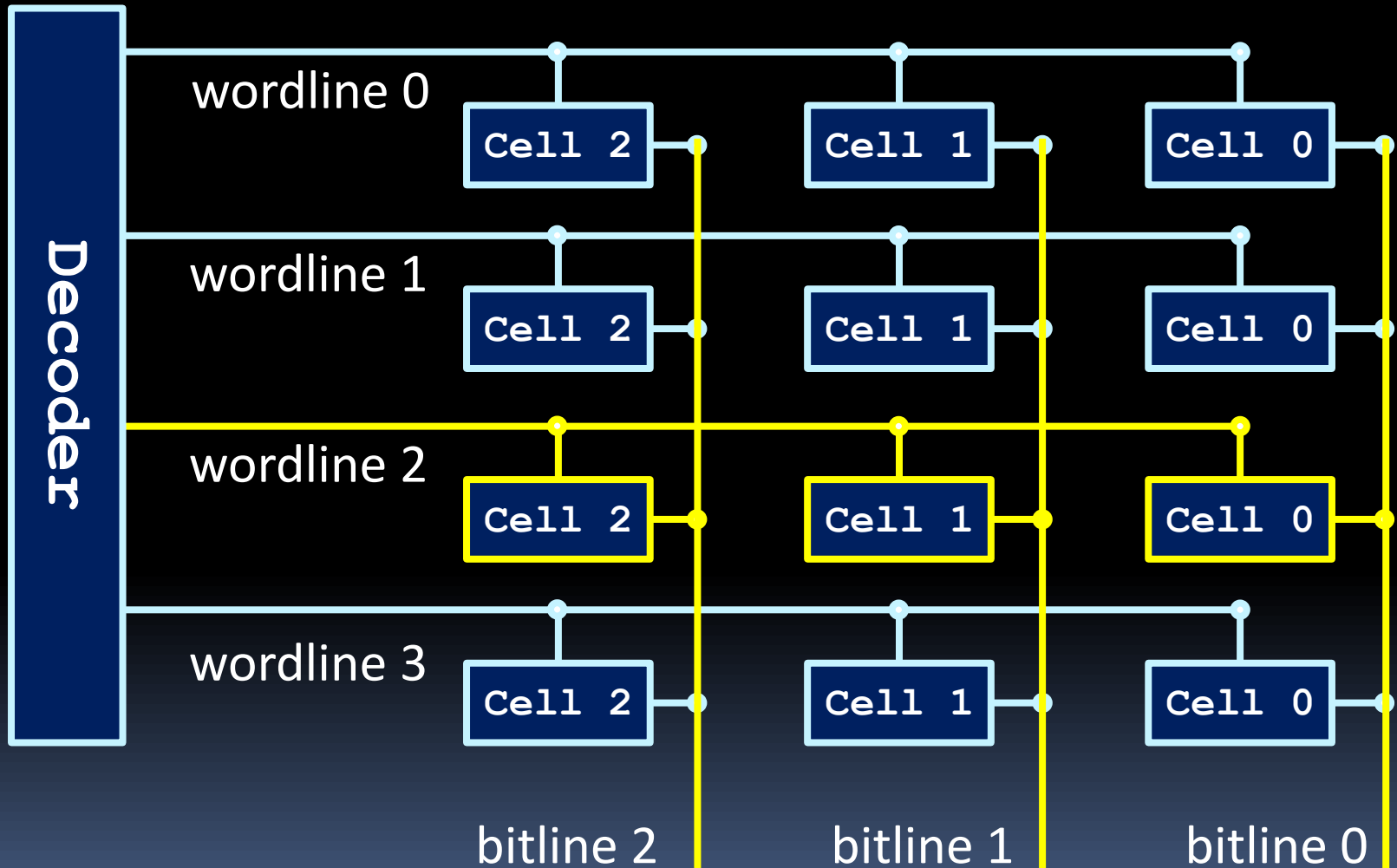
# How we read from wordline 2



# How we read from wordline 2

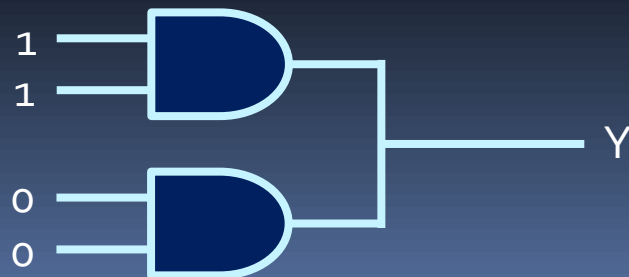
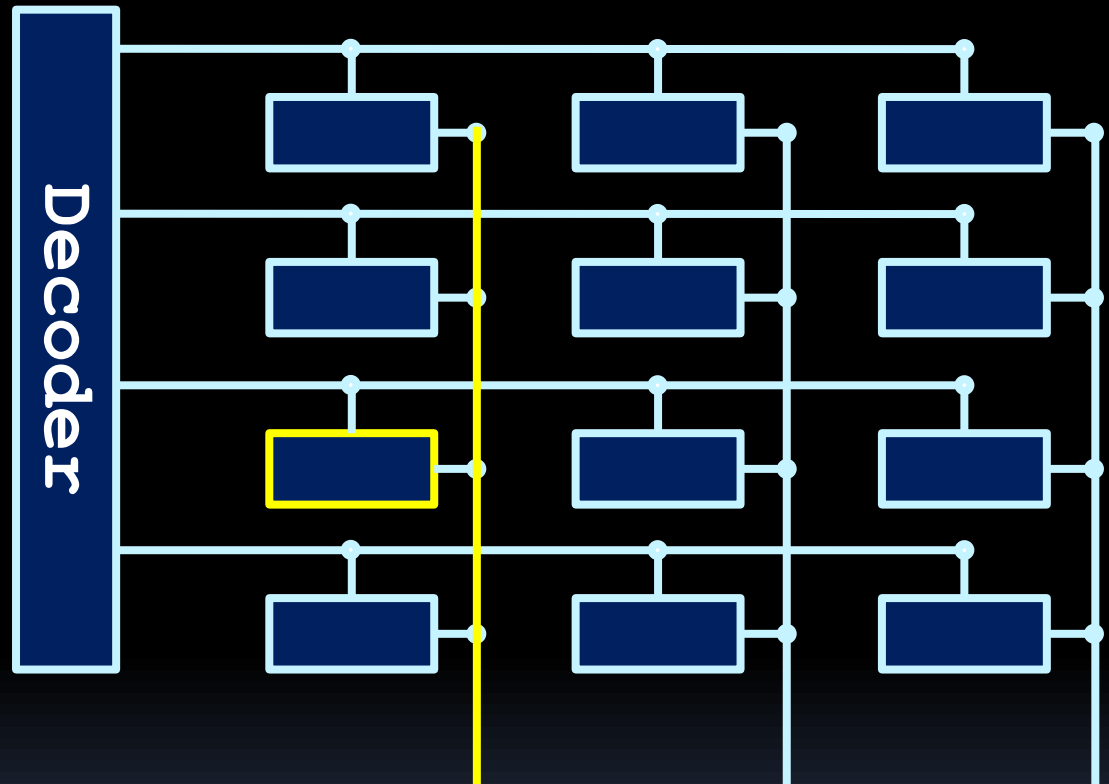


# How we read from wordline 2



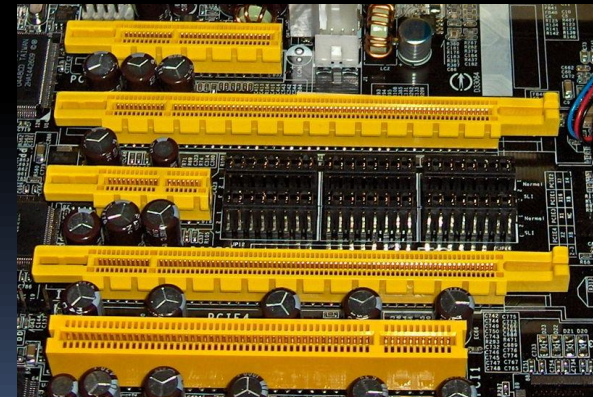
# Wait a Second!

- Outputs of all cells connected to same bitline!
- Cannot have multiple gates driving the same output!



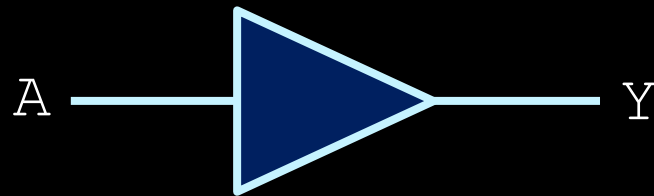
# Data Bus

- Communication between components takes place through groups of shared wires called a **shared bus** (or **data bus**).
- Multiple components can read from a bus at the same time.
- Only one can **write** to a bus at the same time.
  - Also called a **bus driver**.
- How do we make it so?





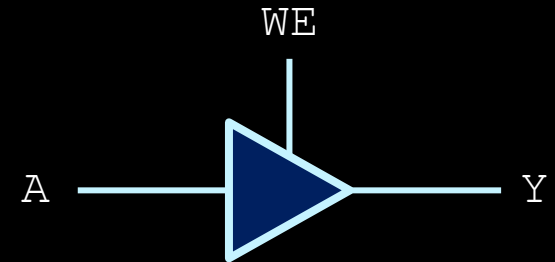
# Buffer



A	Y
0	0
1	1

# Controlling the Flow

- Since some lines (buses) will now be used for both input and output, we introduce a (sort of) new gate called the **tri-state buffer**.
- When WE (write enable) signal is low, buffer output is a **high impedance** "signal" Z.
  - The output is **floating**: neither connected to high voltage or to the ground.
  - This is called "**high Z**"

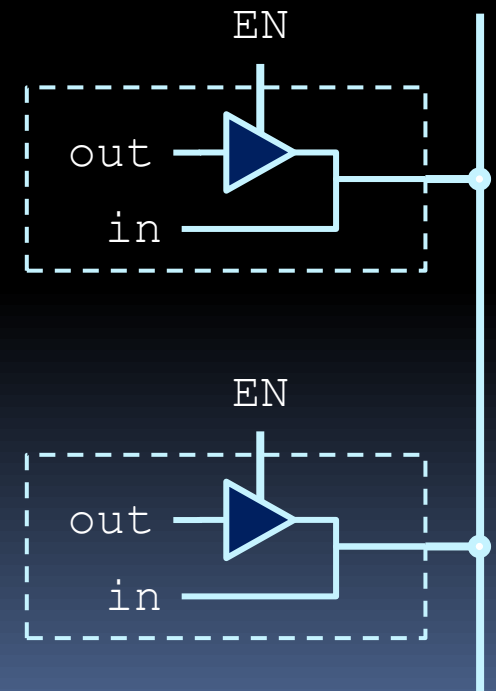
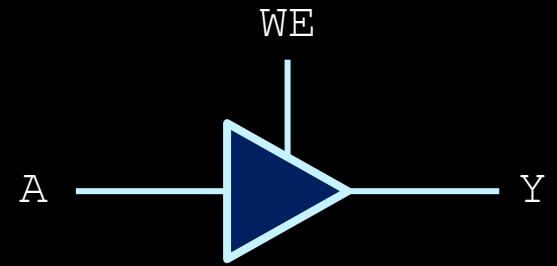


WE	A	Y
0	x	z
1	0	0
1	1	1

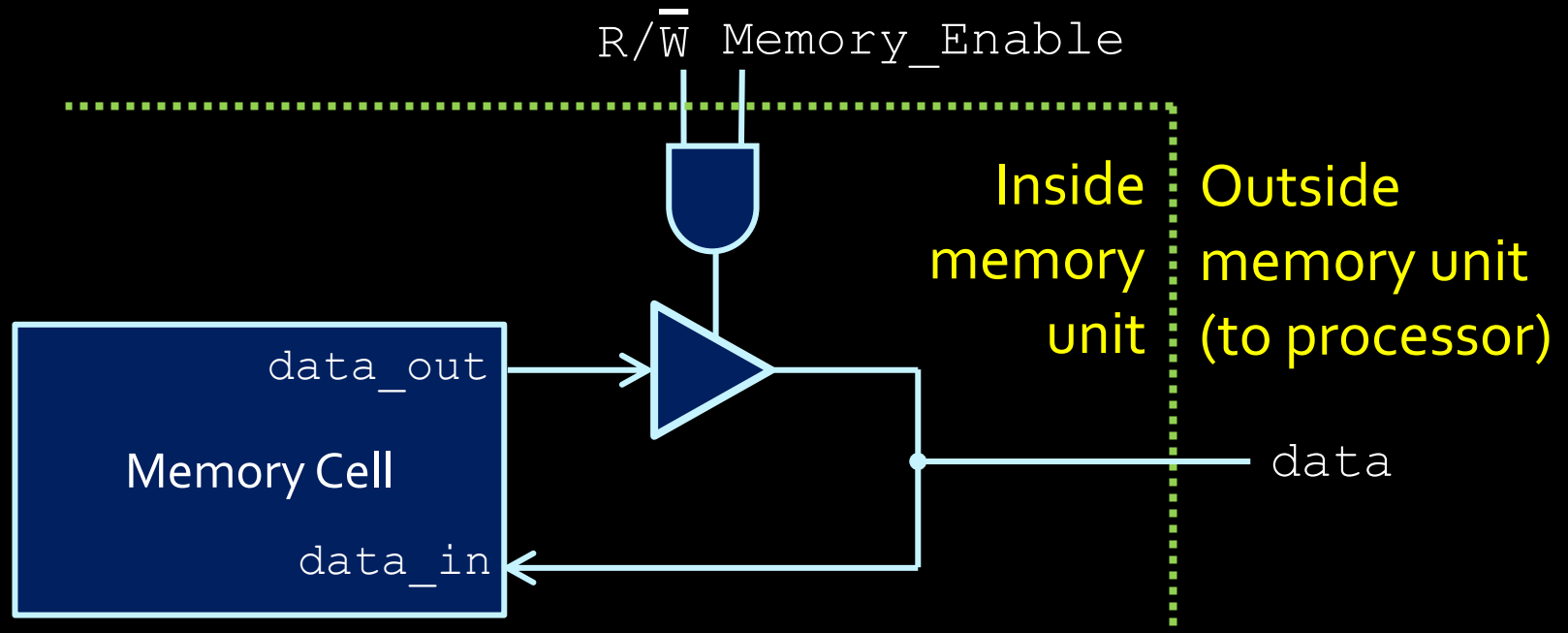


# Controlling the Flow

- Summary:
  - $WE = 1$ : Y is connected to A
  - $WE = 0$ : Y is disconnected
- We use this to make sure only one device writes to the bus at any time.
  - Each component has a **tristate buffer** that feeds into the bus.
  - When not reading or writing, the tristate buffer drives high impedance onto the bus



# Tri-state Buffer in Memory



- If  $R/\overline{W}$  is 1 (read) & `Memory_Enable` is 1
  - `data` behaves as output to processor
- Otherwise if  $R/\overline{W}$  is 0
  - `data` behaves as input from processor to memory since the tri-state buffer is disabled.

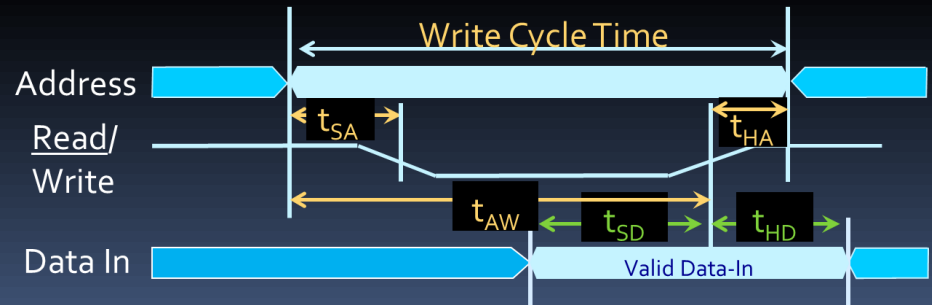
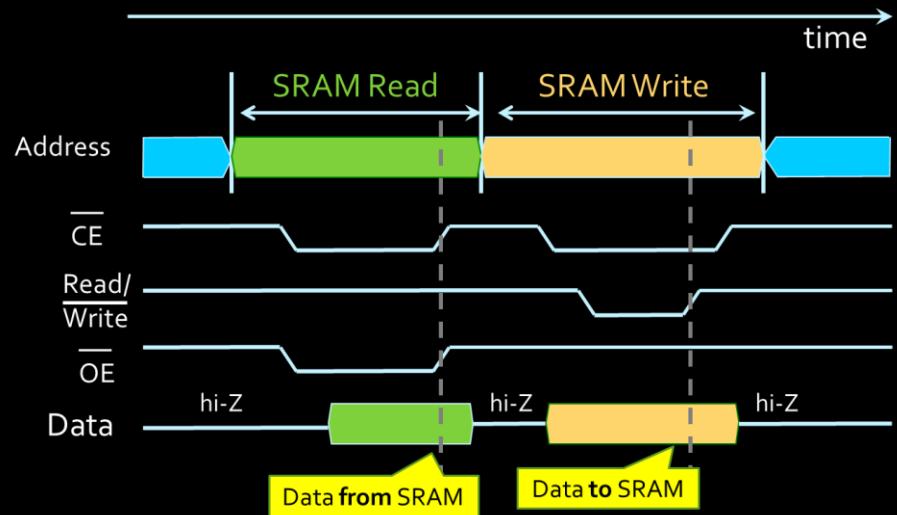
# Timing Is Everything

- RAM is slow.
- Flipflops store and read data in a single clock cycle.
- RAM is slower and further away from the CPU.
- We need to coordinate when to read and write data, addresses.



# Asynchronous SRAM Interface

- There is a protocol for reading and writing from RAM.
  - Set up addresses, data, and control signals in the right order
  - With some timing constraints.
- It is not complicated, but we prefer to focus on other things.



# Summary: Memory vs Registers



- Memory houses most of the data values being used by a program.
  - And the program instructions themselves!
- Registers are for local / temporary data stores, meant to be used to execute an instruction.
  - Registers are can host memory between instructions (like scrap paper for a calculation).
  - Some have special purpose or used to control execution, like the stack pointer register

# Load-Store Architecture

- The MIPS processor architecture we are building is a **load-store architecture**.
  - We load data from main memory to registers
  - Process them using ALU
  - Store back in main memory
- We do either ALU or memory, not both.
- This simplifies design of datapath and instruction set.

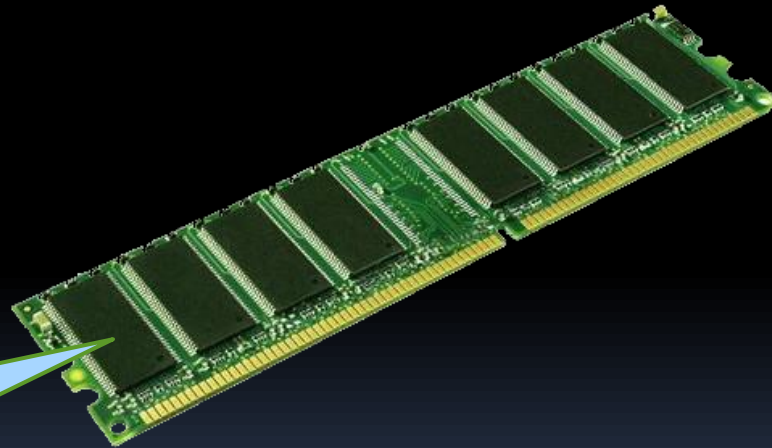


# Memory Hierarchy

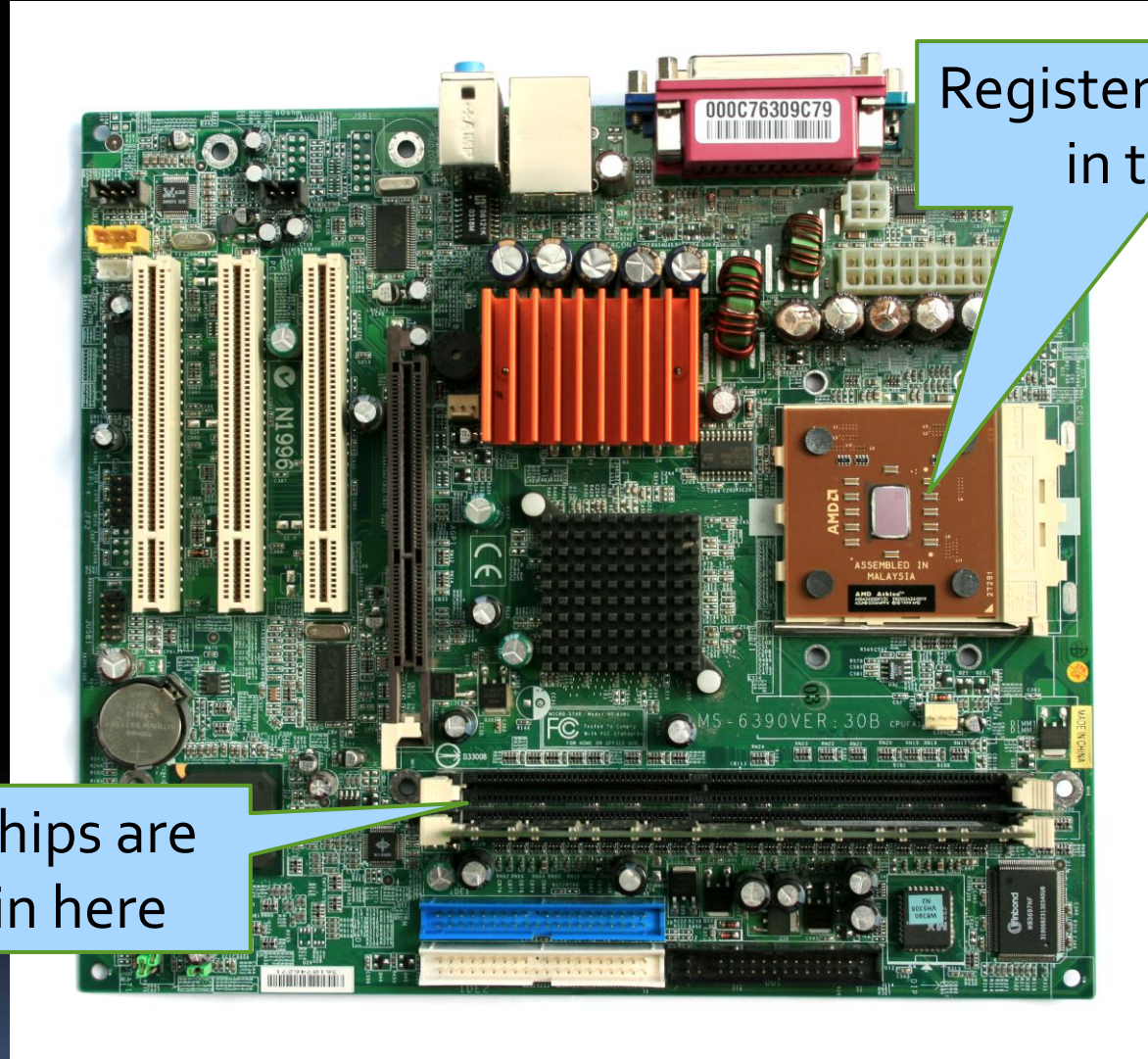
- There are in fact multiple levels of memory.
  - We saw only two.
- Memory sorted by access speed:
  1. **Register file** or “registers”  In this course
  2. Cache (several levels)
  3. **RAM** or “memory”: off-chip  In this course
  4. Hard disk: requires OS support
  5. Network: quite slow



Registers are in here



Memory is in here



Registers are in here  
in the CPU

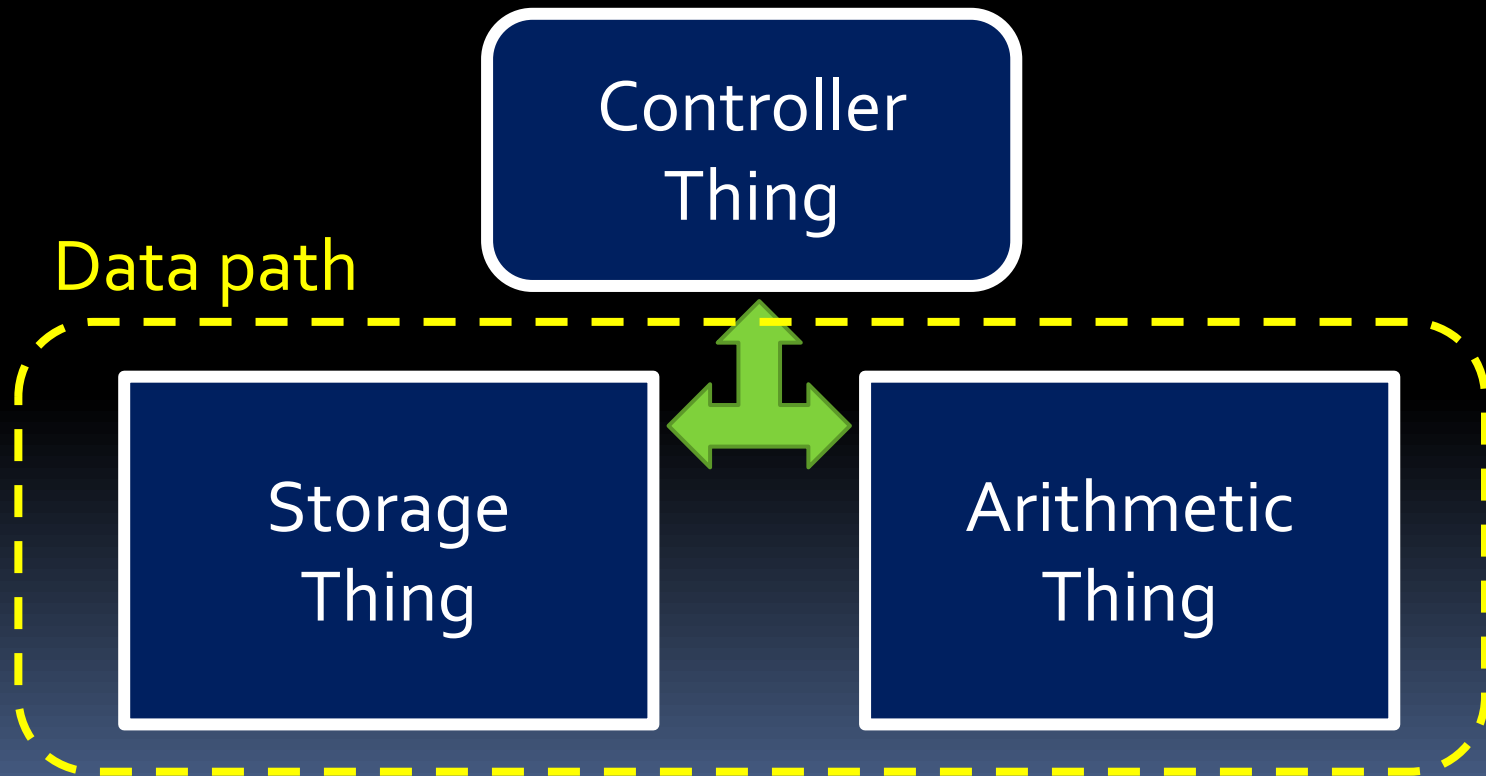
Memory chips are  
plugged in here

# Memory Hierarchy as Food

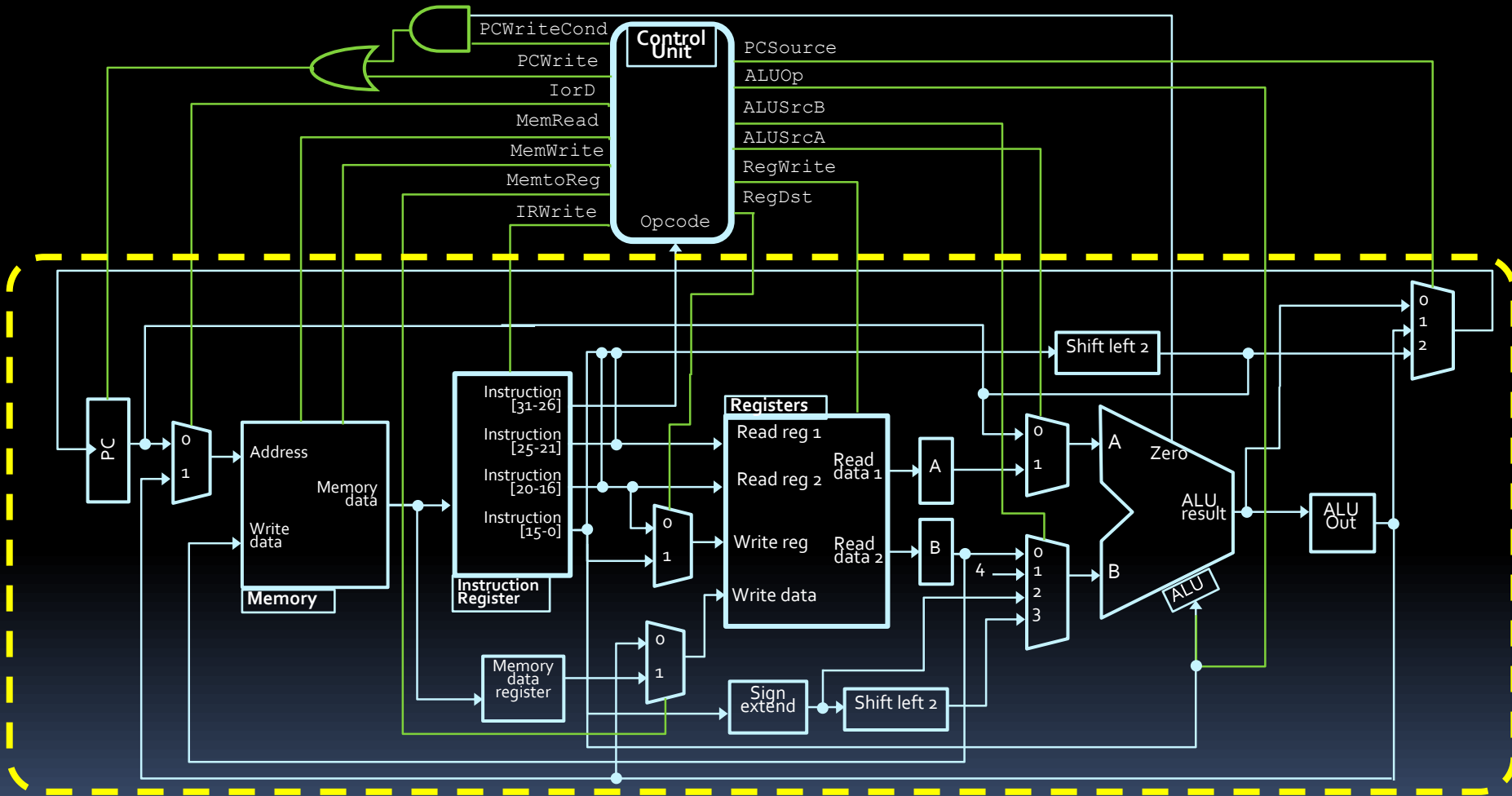
(In terms of access speed)

- **Registers**: food in your mouth, ready for chewing
- Cache: food on your plate
- **Memory**: food in your fridge
- Hard disk: grocery store down the street
- Network: the farm

- Now we know what the Arithmetic and Storage Things do



# Processor Datapath Diagram



Data path