CS 240
Foundations of Computer Systems

WELLESLEY

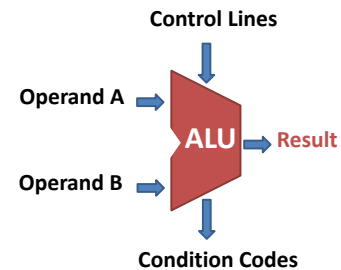Latch: CC-BY Rberteig@flickr

Sequential Logic and State

Output depends on inputs *and stored values*.
(vs. combinational: output depends only on inputs)

Elements to store values: latches, flip-flops, registers, memory

https://cs.wellesley.edu/~cs240/

1

Motivation

Control Lines

Operand A →

ALU → **Result**

Operand B →

Condition Codes

Now that we have ALUs to perform computations, how do we store the results?

How do we calculate different results over time?

Answer: we need circuits that depend not just on inputs, but also on *prior state*
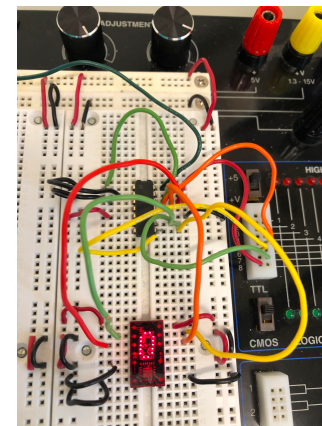= *Sequential Logic*

2

Can you think of an example from lab of a sequential circuit you used?
Hint: previous button pushes are past state.

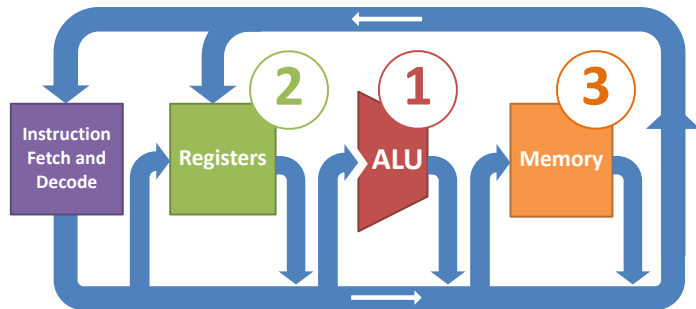Nobody has responded yet.

Hang tight! Responses are coming in.

Example from previous lab

4
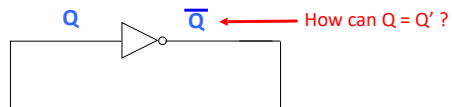
Processor: Data Path Components

Goal for this section

Design a circuit state that **holds** a state over time

- We should be able to set the value to 0 or 1
- We should be able to read the value off the circuit
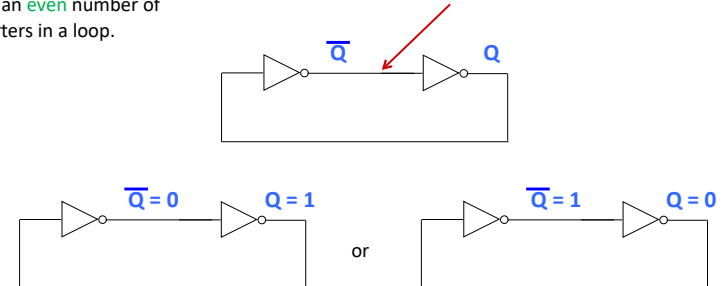
First attempt: Unstable circuit

Q $\overline{Q}$ ← How can Q = Q' ?

Have this issue with any odd number of inverters in a loop.

Second attempt: stable circuit?

Things are more sensible with an even number of inverters in a loop.
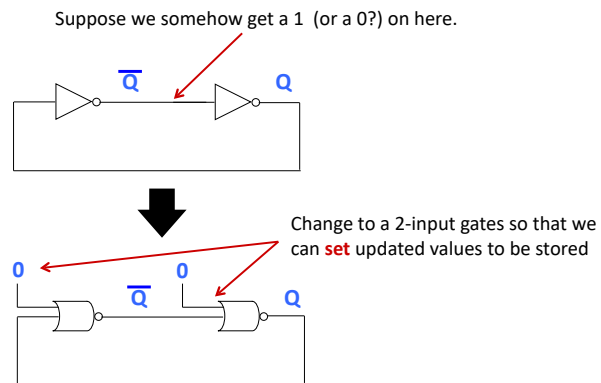
Suppose we somehow get a 1 (or a 0?) on here.

$\overline{Q}$ Q

$\overline{Q} = 0$ Q = 1   or $\overline{Q} = 1$ Q = 0

Now stable, but how do we set the value?

Bistable latches

Things are more sensible with an even number of inverters in a loop.

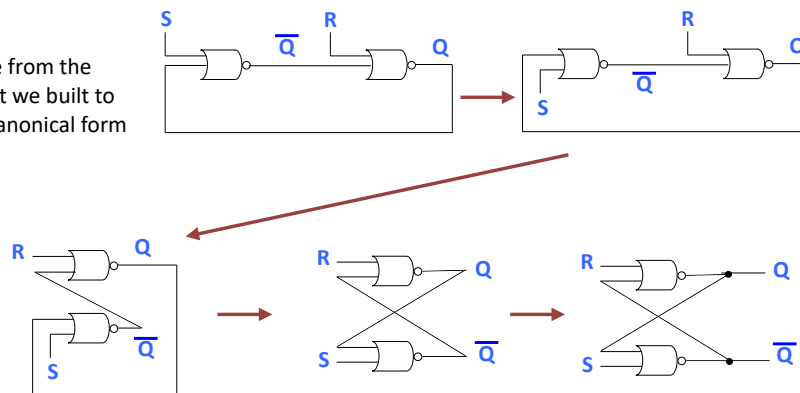Suppose we somehow get a 1 (or a 0?) on here.

$\overline{Q}$ Q

Change to a 2-input gates so that we can **set** updated values to be stored

0   0

$\overline{Q}$ Q

9

SR latch

Set   Reset
S   R

$\overline{Q}$ Q
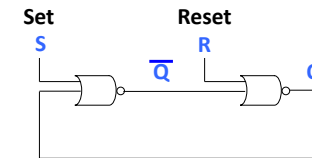
S	R	Q_prev	Q'_prev	Q_next (stable)	Q'_next (stable)
0	0	0	1	0	1
0	0	1	0	1	0
1	0	any	any	1	0
0	1	any	any	0	1
1	1	any	any	0	0

Violates invariant that Q and Q' are inverses!

10

SR latch

Move from the circuit we built to the canonical form

S   R   Q
$\overline{Q}$

R   Q
$\overline{Q}$
S

R   Q
$\overline{Q}$
S

R   Q
$\overline{Q}$
S

R   Q
$\overline{Q}$
S

11

SR latch

R   Q
S $\overline{Q}$

Figure 3.3 SR latch schematic

R — N1 — Q
S — N2 — $\overline{Q}$
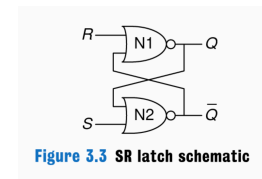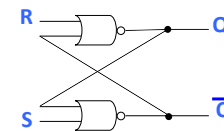
Meets our goals:
• Able to set the value to 0 or 1
• Able to read the value off the circuit

12

Figure 3.3 SR latch schematic

S = 0; R = 0

S = 1; R = 0

S = 0; R = 1

S = 1; R = 1

None of the above
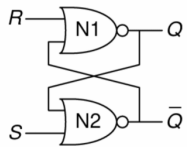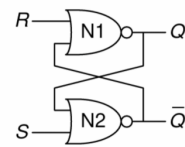

Figure 3.3 SR latch schematic

S = 0; R = 0   0%

S = 1; R = 0   0%

S = 0; R = 1   0%

S = 1; R = 1   0%

None of the above   0%
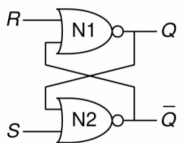

Figure 3.3 SR latch schematic

S = 0; R = 0   0%

S = 1; R = 0   0%

S = 0; R = 1   0%

S = 1; R = 1   0%
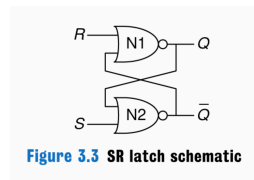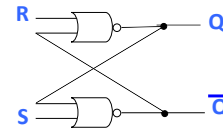
None of the above   0%
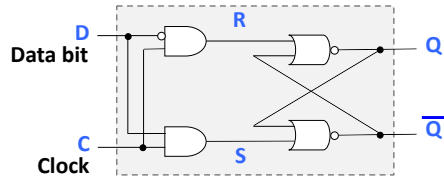

SR latch

Figure 3.3 SR latch schematic

Meets our goals:
- Able to set the value to 0 or 1
- Able to read the value off the circuit

But:
- Ambiguous when S = 1 and R = 1
- No distinction between new value and timing

D latch

Goals:
- Only 1 bit for data
- Control over timing

D Data bit
R
Q
C Clock
S
$\overline{Q}$

if C = 0, then SR latch stores current value of Q.

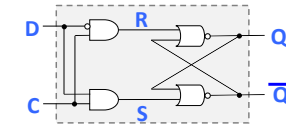if C = 1, then D flows to Q:

  if D = 0, then R = 1 and S = 0, Q = 0
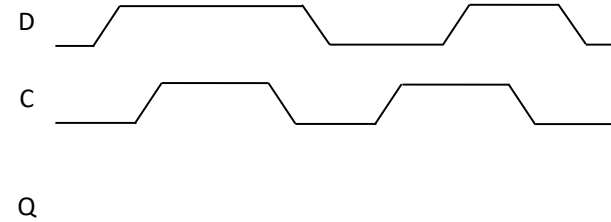
  if D = 1, then R = 0 and S = 1, Q = 1

Notes:
- Data bit D replaces S & R: it's the bit value we want to store when Clock = 1
  - Internally, Data bit D prevents bad case of S = R = 1
- This logic is **level-triggered**; as long as Clock = 1, changes to D flow to outputs

---

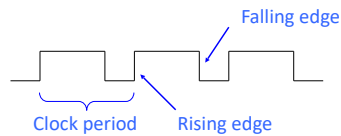# Time matters!

ex

D

C

Q

Assume Q has an initial state of 0

In general: clocks

Clock: free-running signal
with fixed cycle time = clock period = T.

Clock frequency = 1 / clock period

Falling edge

Clock period   Rising edge

A clock controls when to update a sequential logic element's state.

Aside: "Clock frequency"

Clock frequency
 = 1 / period = 1 / s = Hz

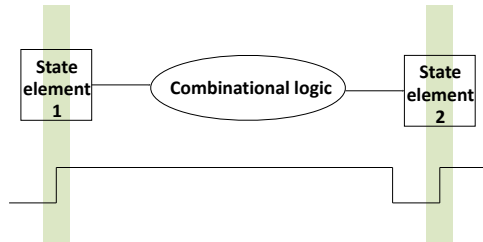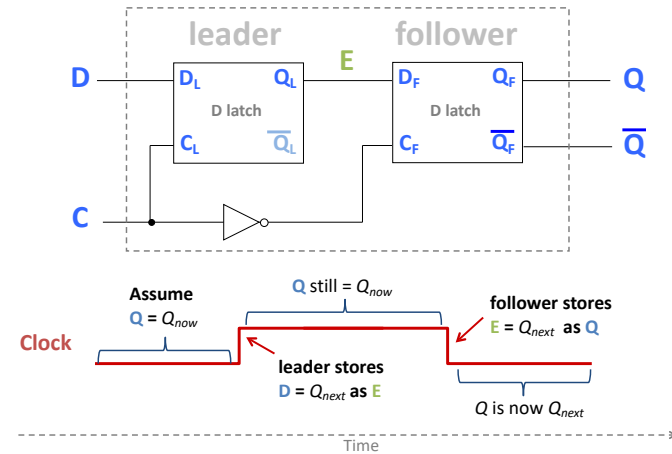Typical CPU: 3-4 GHz

Microprocessor Clock Speed

Logarithmic Plot

(Hz)

$r^2 = 0.9699$

1975 1980 1985 1990 1995 2000 2005 2010 2015 2020

Doubling time: 3 years

Year
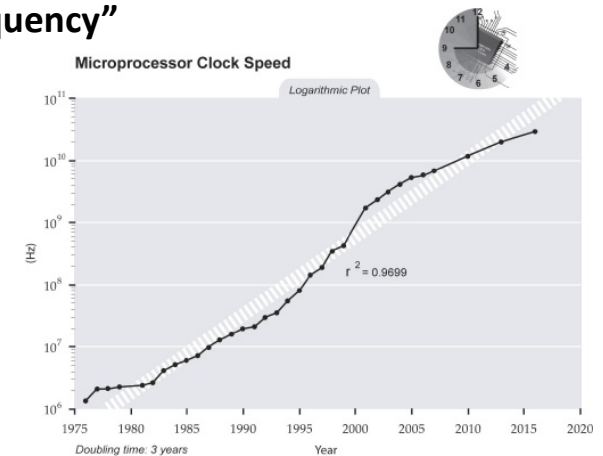
Synchronous systems

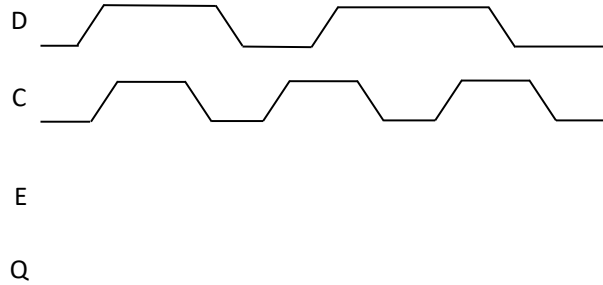Inputs to state elements must be valid on active clock edge.

D flip-flop with *falling*-edge trigger

Time matters! D flip-flop with *falling*-edge trigger

ex

D

C

E

Q

Assume Q and E have an initial state of 0

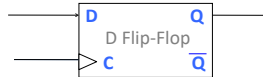## Reading and writing in the same cycle

Clock

D Flip-Flop

D   Q

C $\overline{Q}$

Assume Q is initially 0.
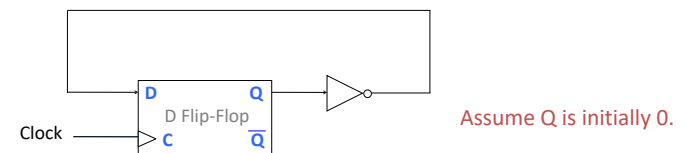
Moral: It's OK to use the current output Q of a flip-flop as part of the the next data input D to the same flip-flop.
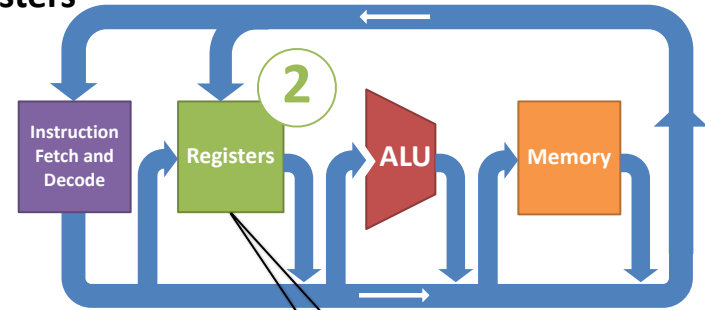
D flip-flop = one bit of storage

The bit value of D when C has a falling edge is remembered at Q until the next falling edge of C.
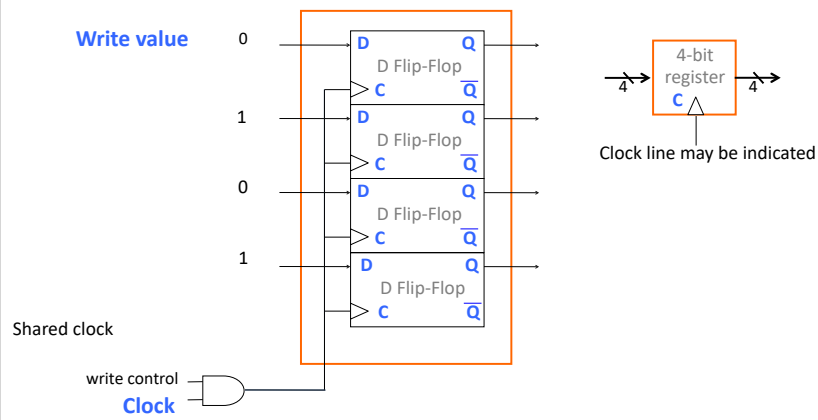
Registers

Assembly code (later this semester):   `addq %rdi,%rsi`

A 1-nybble* register   *Half a byte!

(a 4-bit hardware storage cell)

Write value

Clock line may be indicated

Shared clock

write control

Clock

Register file

Register address #1

Register address #2

Register address #3

Write data

Read data 1

Read data 2

**Read ports
Why 2?**

Write port   Write?

0 = read
1 = write

$r = \log_2$ number of registers
w = bits in word
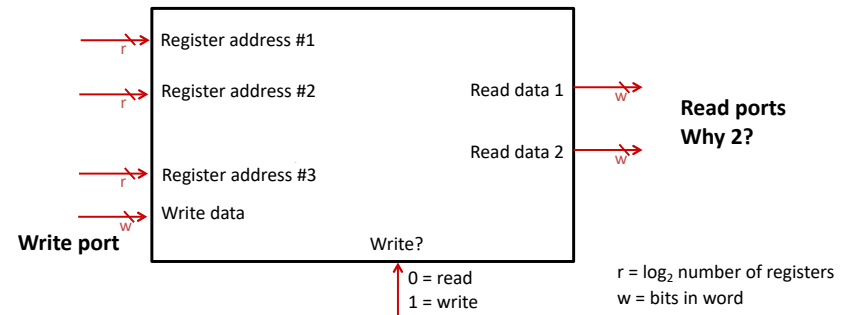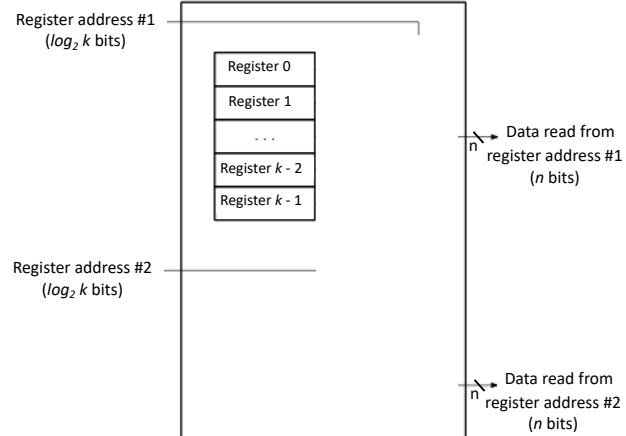
Array of registers, with register selectors, write/read control, input port for writing data, output ports for reading data.

Read ports (data out)
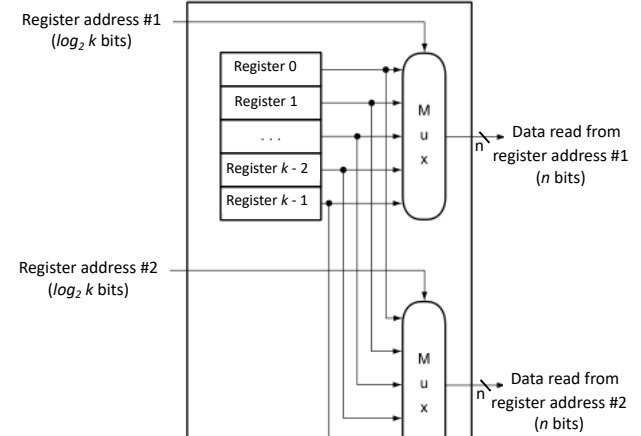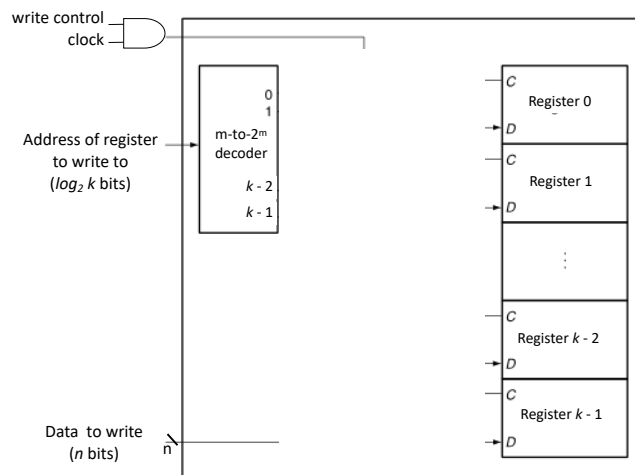
Register address #1
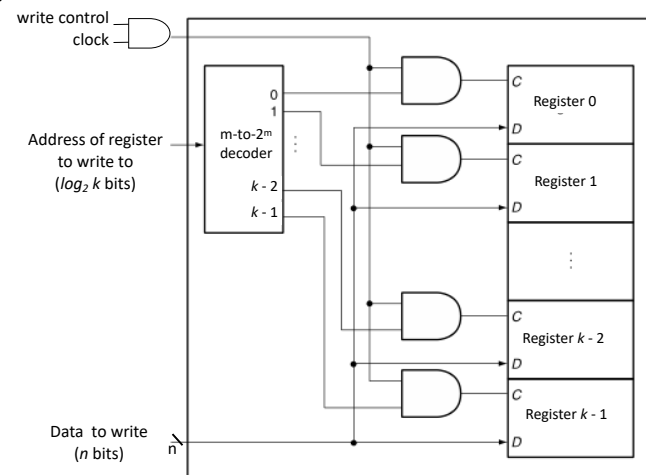($log_2 k$ bits)

Register 0
Register 1
. . .
Register k - 2
Register k - 1

n Data read from register address #1 (n bits)

Register address #2
($log_2 k$ bits)

n Data read from register address #2 (n bits)

29

Read ports (data out)

Register address #1
($log_2 k$ bits)

Register 0
Register 1
. . .
Register k - 2
Register k - 1

Mux

n Data read from register address #1 (n bits)

Register address #2
($log_2 k$ bits)

Mux

n Data read from register address #2 (n bits)

30

Write port (data in)

write control
clock

Address of register to write to
($log_2 k$ bits)

m-to-2^m decoder
0
1
k - 2
k - 1

C
Register 0
D

C
Register 1
D

C
Register k - 2
D

C
Register k - 1
D

Data to write (n bits)
n

31

Write port (data in)

write control
clock

Address of register to write to
($log_2 k$ bits)

m-to-2^m decoder
0
1
k - 2
k - 1

C
Register 0
D

C
Register 1
D

C
Register k - 2
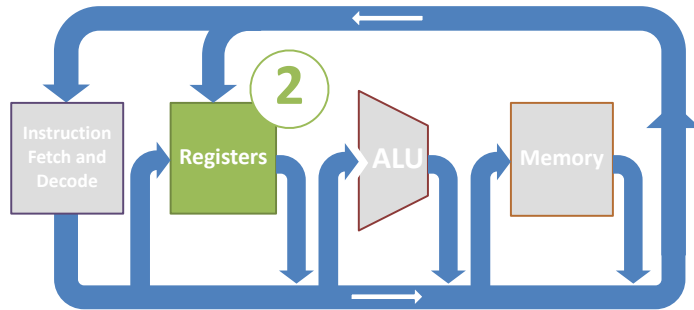D

C
Register k - 1
D

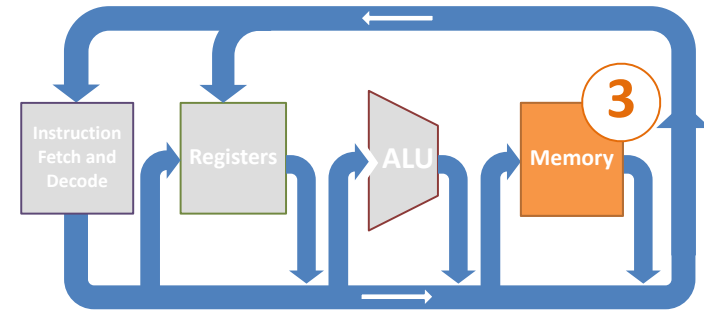Data to write (n bits)
n

32

Registers summary

- For our purposes: implemented with flip-flops
- Very fast access
- Limited in size:
  - Need an m-to-2^m decoder
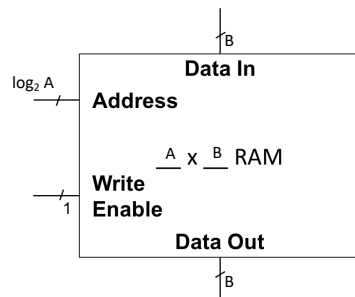 - CPUs typically have ~10s of words of register storage

Registers summary

- We'll think about at a higher level of abstraction
- Designed to handle a much larger amount of data
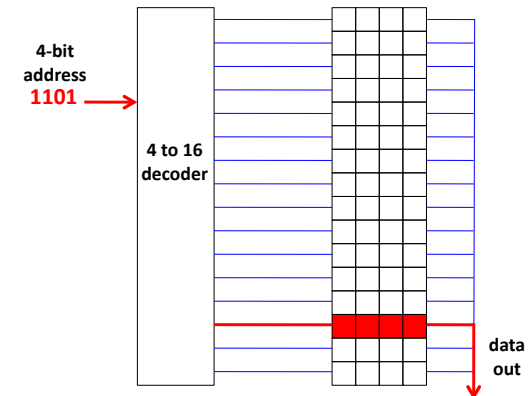 - CPUs can have millions-billions of words of memory storage

RAM (Random Access Memory)

- A is number of words in RAM
- Specify the desired word by an address of size $\log_2 A$
- B is the width of each word (in bits)

16 x 4 RAM