CSE 390B, Autumn 2022

Building Academic Success Through Bottom-Up Computing

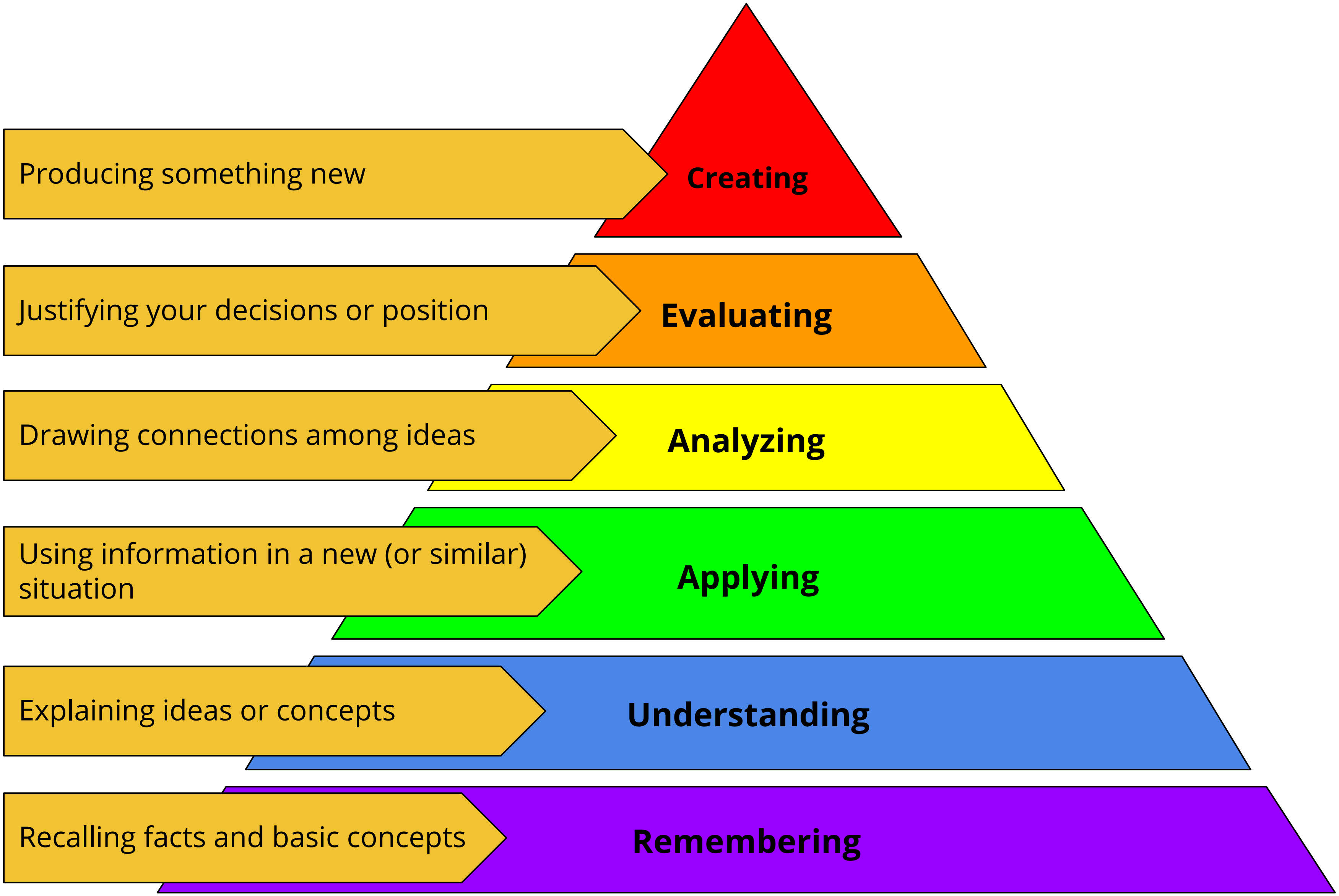
Cornell Note-taking & Building Memory

Cornell Note-taking Method, Storing Data, Representing and Building Memory, Program Counter Overview

Lecture Outline

- **Cornell Note-taking Method**
  - **System for Taking, Organizing, and Reviewing Notes**
- Review of Sequential Logic and DFFs
- Storing Data: Bit
  - Bit Overview and Implementation
- Representing and Building Memory
  - Array Abstraction, Building From the Bit
- Program Counter (PC) Overview
  - Control Flow of Computer Programs

Recap: Bloom's Taxonomy



Cornell Note Taking Method

Questions	**Notes**
	I. Main Topic ○ Sub point ○ <u>definition</u> ○ example **
Compose a question that corresponds to the notes you took	
In what ways is object-oriented programming more extensible than functional programming?	II. Object-Oriented Programming ○ Encapsulates the data and the operations for a given data type ○ Provides abstractions - you don't need to know how a car is implemented in order to use it ○ Extensibility - easier to <u>add new data types</u>
	III. Functional Programming ○ Extensibility - easier to <u>add new operations</u>

Summary

Object-oriented programming and functional programming are two types of programming paradigms...

Cornell Note Taking Method

Questions

Compose a question that corresponds to the notes you took

In what ways is object-oriented programming more extensible than functional programming?

Summary

Object-oriented programming and functional programming are two types of programming paradigms…

Applying the Cornell Note-Taking Method

- Try it during today's technical lecture!
- Next week, we'll provide you with an opportunity to discuss your notes with your classmates
- You will also practice it with one of your other classes as part of Project 4

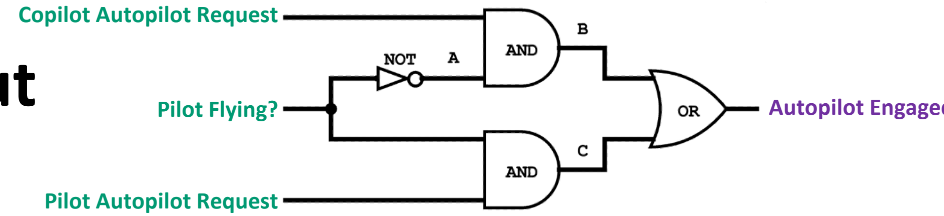
Lecture Outline

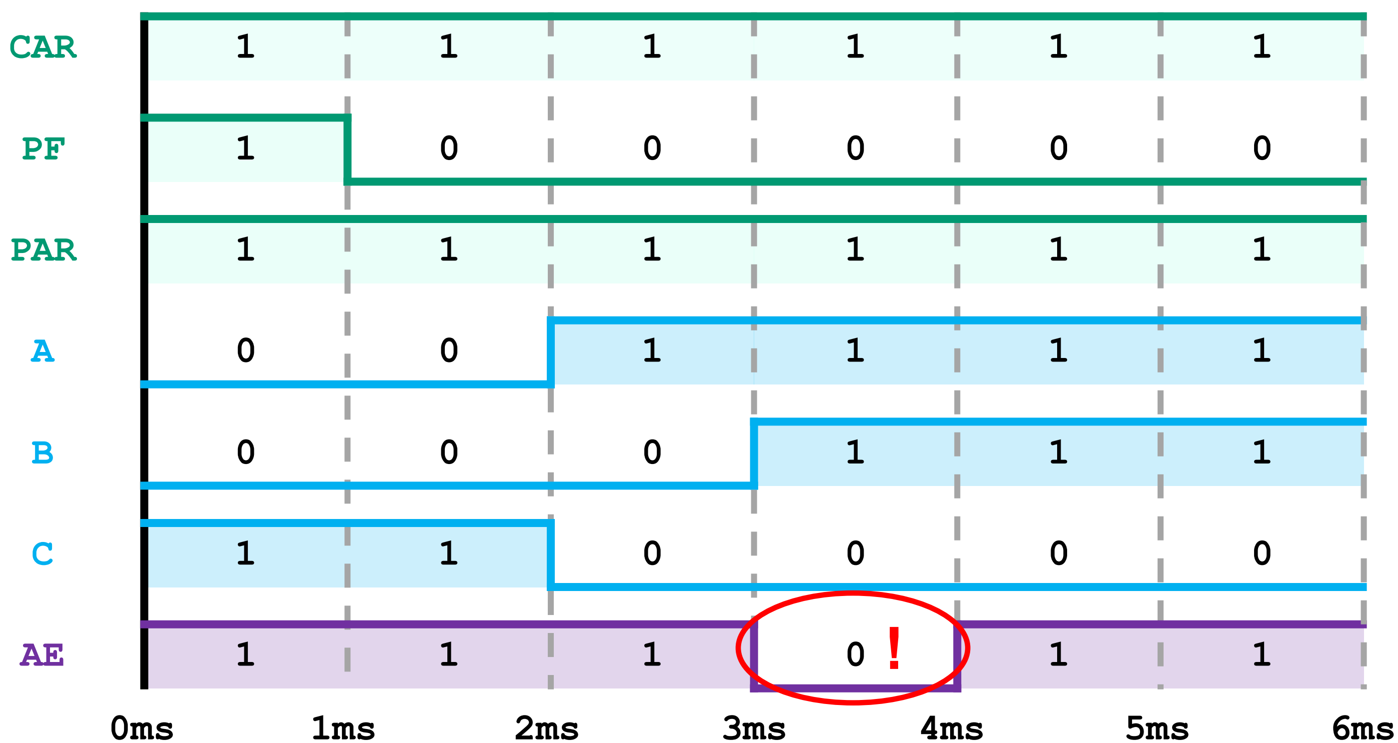
- Cornell Note-taking Method
  - System for Taking, Organizing, and Reviewing Notes
- **Review of Sequential Logic and DFFs**
- Storing Data: Bit
  - Bit Overview and Implementation
- Representing and Building Memory
  - Array Abstraction, Building From the Bit
- Program Counter (PC) Overview
  - Control Flow of Computer Programs

Poll Everywhere

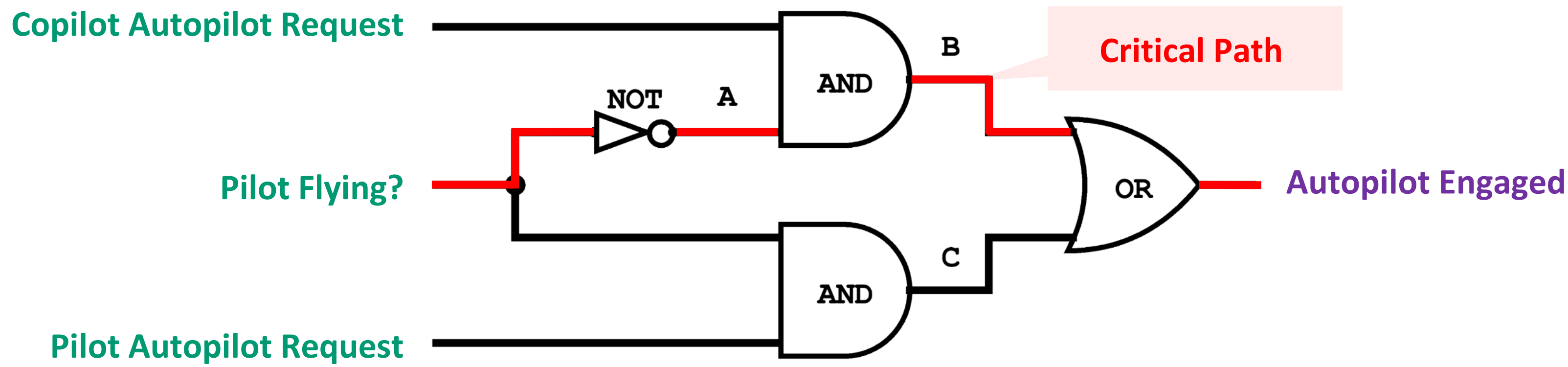
Vote at https://pollev.com/cse390b

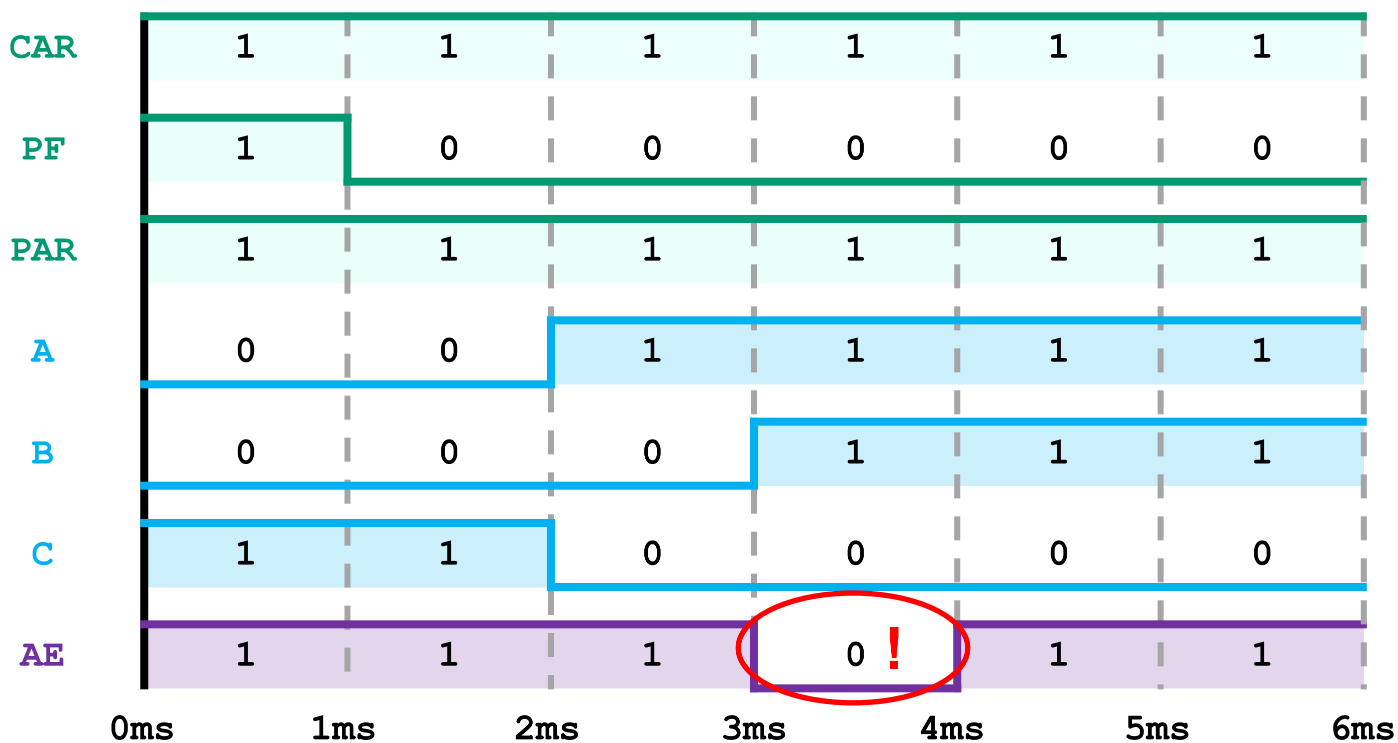
Describe the behavior of the Autopilot Engaged (AE) output between 1ms to 6ms.





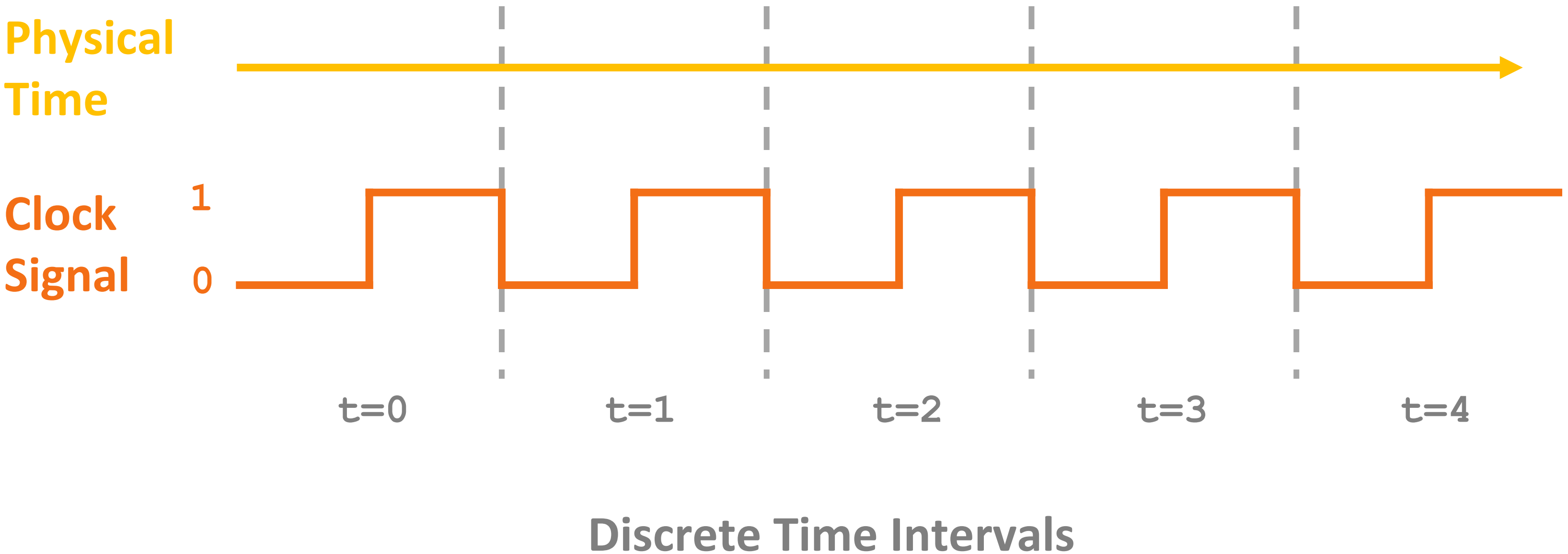
Autopilot Control Circuit Example





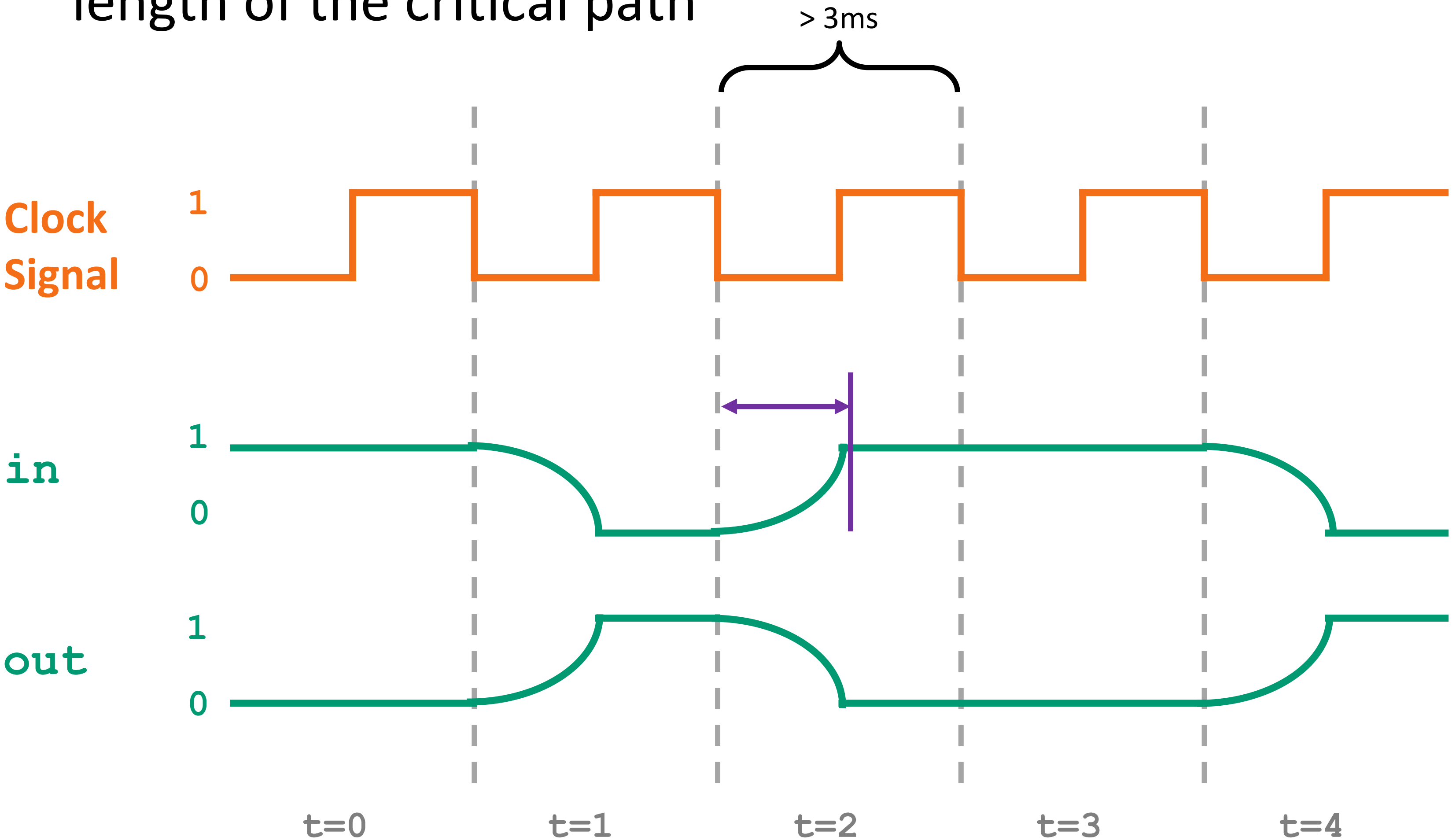
Physical Timekeeping

- Hardware keeps track of time using an alternating signal
  - Creates the idea of **discrete time:** state changes only occur in discrete intervals, right when signal alternates



Adding a Clock: Clock Cycles

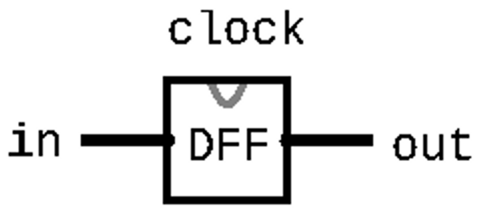
- Choose a clock cycle length slightly longer than the delay length of the critical path



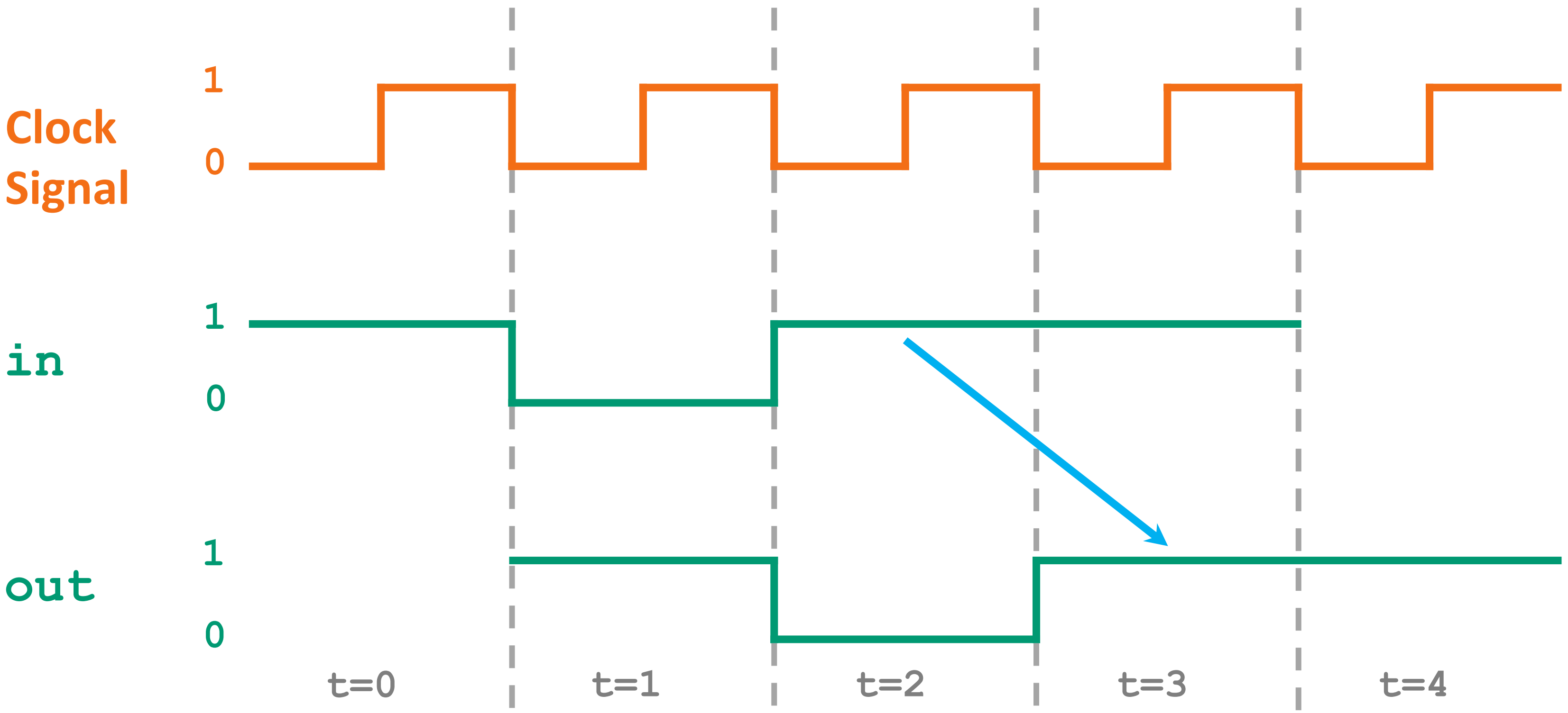
The Data Flip-Flop Gate

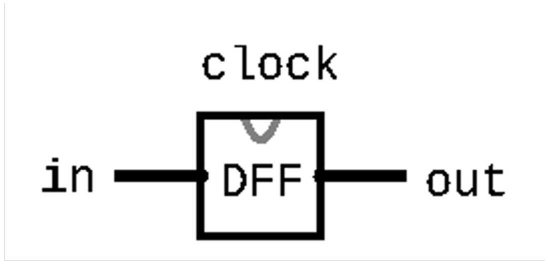
- Simplest state-keeping component
  - 1-bit input, 1-bit output
  - Wired to the clock signal
  - Always outputs its previous input: **`out(t) = in(t-1)`**

- Implementation: a gate that can flip between two stable states (remembering 0 vs. remembering 1)
  - Gates with this behavior are “Data Flip Flops” (DFFs)



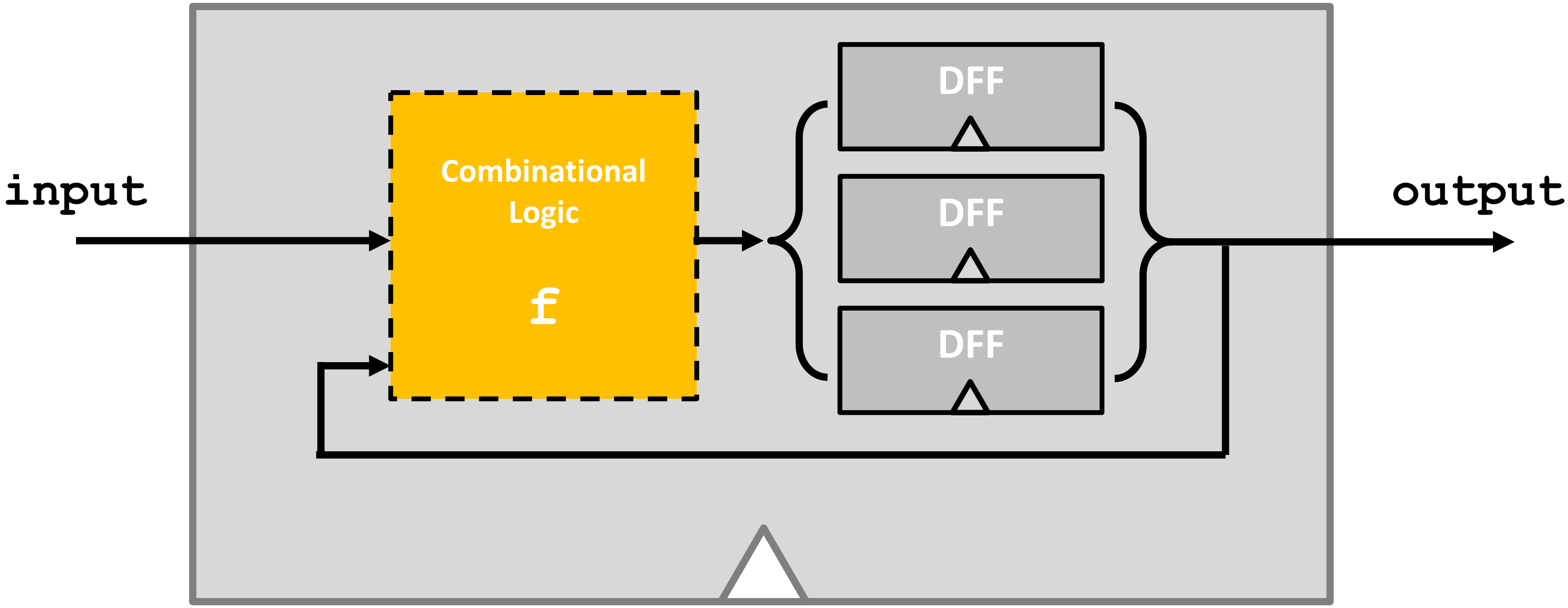
Data Flip-Flop (DFF) Behavior



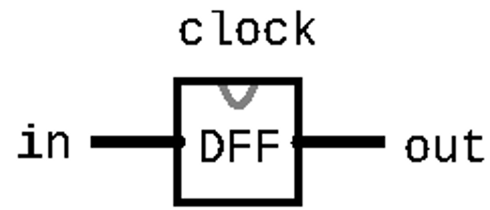


Sequential Chips

`output(t) = f(state(t-1), input(t))`



D Flip-Flop: Time Series



- DFF Specification:

`out(t) = in(t-1)`

in	0	0	1	1	0	1	0	...
out	0	0	0	1	1	0	1	...
time	t=0	t=1	t=2	t=3	t=4	t=5	t=6	...

Example: `out(t=3)` = `in(t=2)`

DFF Example 1: Specification

- Example specification:

  `out(t) = Xor(a(t-1), b(t-1))`

- Takes two inputs, `a` and `b`, and outputs the `Xor` of them
  - Note that out at time `t` is determined by `a` and `b` at time `t-1`
  - We will need to use a DFF

- Exercise: Draw out the corresponding circuit diagram and HDL implementation

DFF Example 1: Time Series

- Example specification:
  `out(t) = Xor(a(t-1), b(t-1))`

a	0	0	1	1	1	0	0	...
b	0	1	0	1	1	1	0	...
out	0	0	1	1	0	0	1	...
time	t=0	t=1	t=2	t=3	t=4	t=5	t=6	...

- Example: `out(t=3) = Xor(a(t=2), b(t=2))`

DFF Example 1: Circuit Diagram & HDL

- Example specification:

  `out(t) = Xor(a(t-1), b(t-1))`

- Circuit diagram:

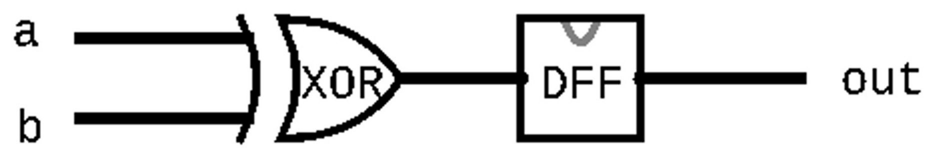
- HDL:

DFF Example 1: Circuit Diagram & HDL

- Example specification:

  `out(t) = Xor(a(t-1), b(t-1))`

- Circuit diagram:



- HDL:

```
CHIP Example1 {
    IN a, b;
    OUT out;

    PARTS:
    Xor(a=a, b=b, out=xorout);
    DFF(in=xorout, out=out);
}
```

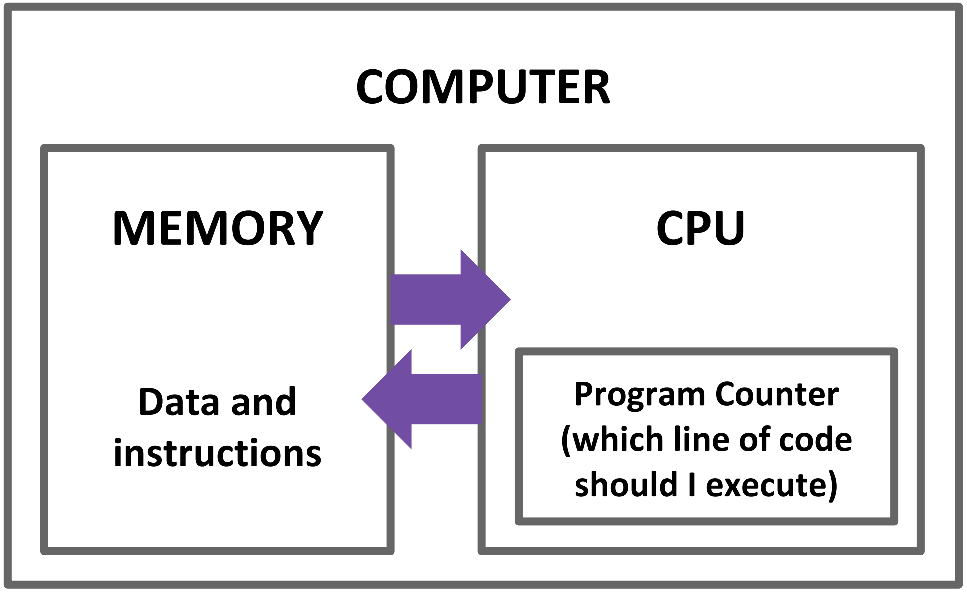
Lecture Outline

- Cornell Note-taking Method
  - System for Taking, Organizing, and Reviewing Notes
- Review of Sequential Logic and DFFs
- **Storing Data: Bit**
  - **Bit Overview and Implementation**
- Representing and Building Memory
  - Array Abstraction, Building From the Bit
- Program Counter (PC) Overview
  - Control Flow of Computer Programs

Computer Overview

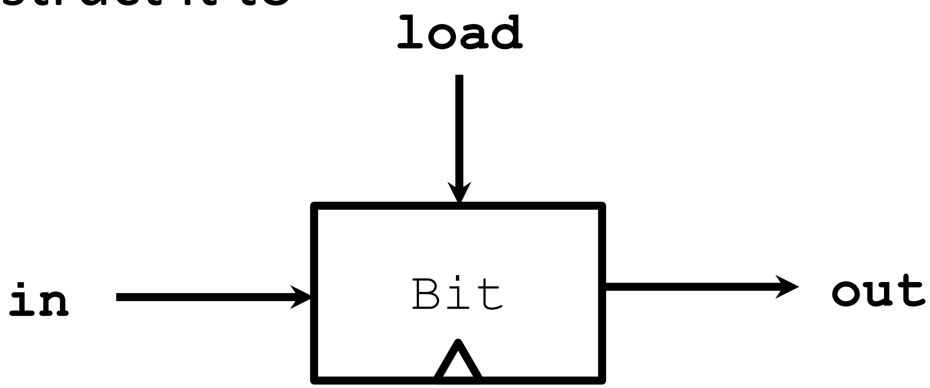
- CPU is the "brain" of our computer
  - Does necessary computations (add, subtract, multiply, etc.)

- Memory is used to store values for later use
  - Requires persistence across multiple computations
  - Needs to change values at our discretion



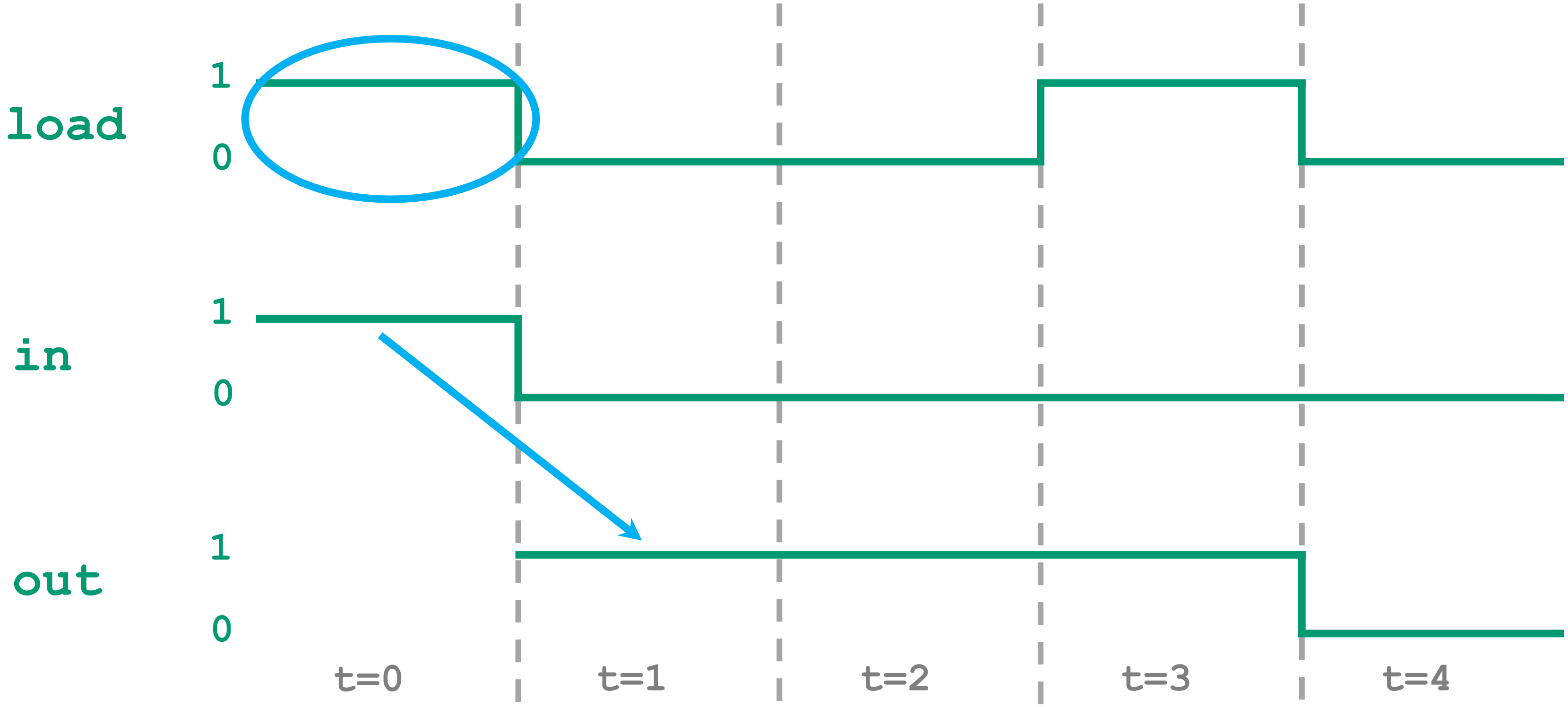
Storing Data: Bit

- A Flip-Flop changes state *every* clock cycle
- We will build the abstraction of a "Bit" that only changes when we instruct it to

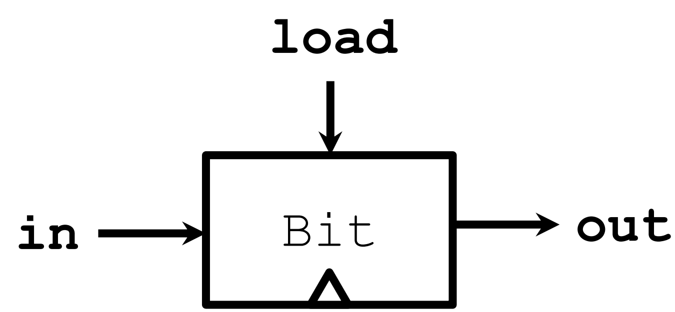



```
if load(t-1)      out(t) = in(t-1)
else              out(t) = out(t-1)
```

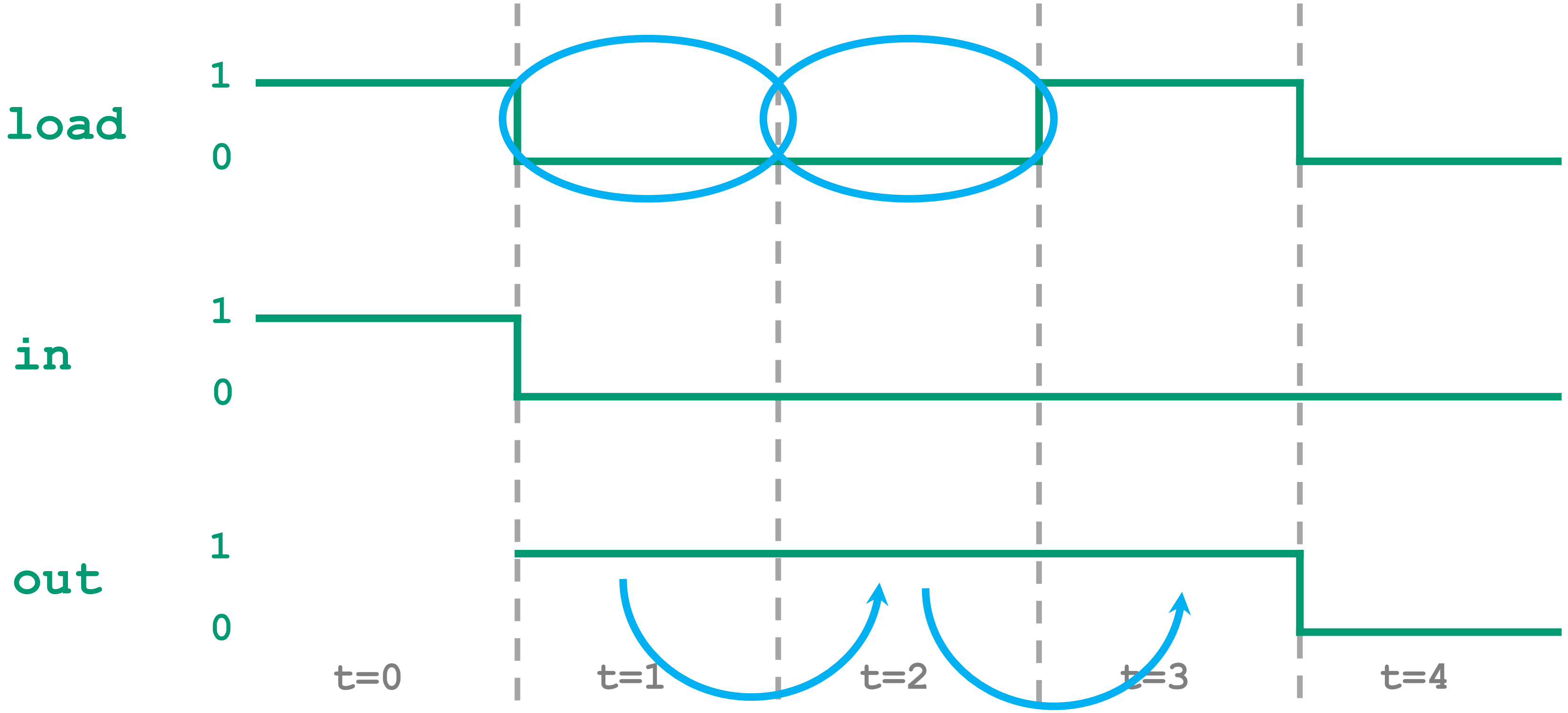
Bit Behavior



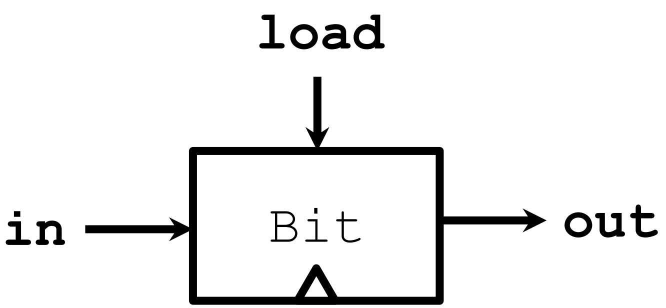

```
if load(t-1)     out(t) = in(t-1)
else             out(t) = out(t-1)
```



Bit Behavior




```
if load(t-1)     out(t) = in(t-1)
else             out(t) = out(t-1)
```



Bit Time Series

- Bit Specification:

```
if (load(t-1)): out(t) = in(t-1)
          else: out(t) = out(t-1)
```

load	1	0	0	1	1	1	0	...
in	1	0	0	0	1	0	1	...
out	0	1	1	1	0	1	0	...
time	**t=0**	**t=1**	**t=2**	**t=3**	**t=4**	**t=5**	**t=6**	...

Example 1: `load(t=0) == 1` so `out(t=1) = in(t=0)`

Bit Time Series

- Bit Specification:

```
if (load(t-1)): out(t) = in(t-1)
          else: out(t) = out(t-1)
```

load	1	0	0	1	1	1	0	...
in	1	0	0	0	1	0	1	...
out	0	1	1	1	0	1	0	...
time	**t=0**	**t=1**	**t=2**	**t=3**	**t=4**	**t=5**	**t=6**	**...**

- Example 1: `load(t=0)` `==` `1`, so `out(t=1)` `=` `in(t=0)`

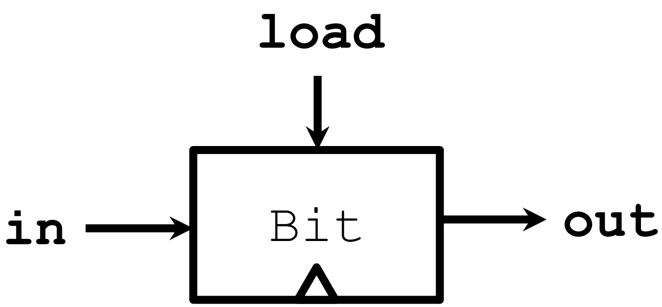
- Example 2: `load(t=2)` `==` `0`, so `out(t=3)` `=` `out(t=2)`

Poll Everywhere

Vote at https://pollev.com/cse390b

Which gates will we need to implement a Bit? Select all that apply.

A. Mux
B. Xor
C. And
D. DFF
E. We're lost…




```
if load(t-1)      out(t) = in(t-1)
else              out(t) = out(t-1)
```

Implementing a Bit

- Bit Specification:

```
if load(t-1)      out(t) = in(t-1)
else              out(t) = out(t-1)
```

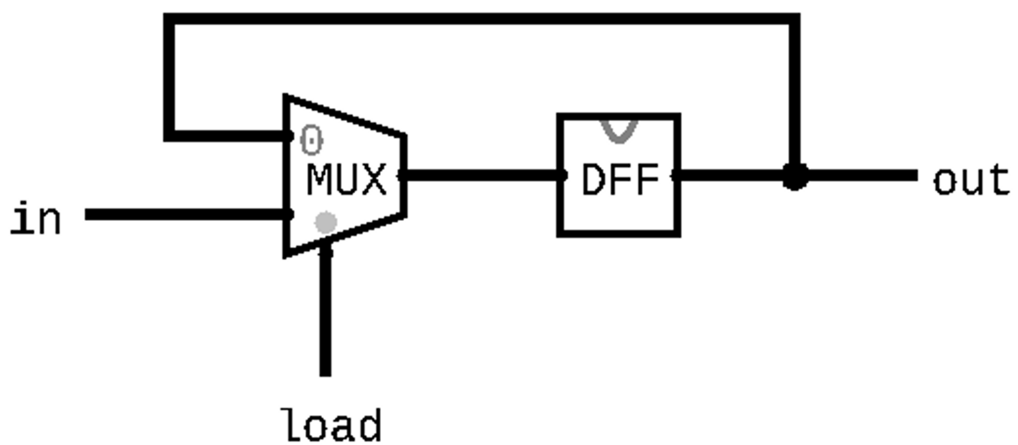
- Exercise: fill in the connections to the gates to create a circuit diagram of Bit

Implementing a Bit

- Bit Specification:

```
if load(t-1)     out(t) = in(t-1)
else             out(t) = out(t-1)
```

- Exercise: fill in the connections to the gates to create a circuit diagram of Bit



Five-minute Break!

- Feel free to stand up, stretch, use the restroom, drink some water, review your notes, or ask questions

- We'll be back at:

Lecture Outline

- Cornell Note-taking Method
  - System for Taking, Organizing, and Reviewing Notes
- Review of Sequential Logic and DFFs
- Storing Data: Bit
  - Bit Overview and Implementation
- **Representing and Building Memory**
  - **Array Abstraction, Building From the Bit**
- Program Counter (PC) Overview
  - Control Flow of Computer Programs

Memory Representation

- Memory can be abstracted as one huge array
- **Addresses** are indices into different memory slots
  - The width of an address is fixed for the system
  - The nand2tetris project will use 16-bit addresses
- Each **value** in memory takes up a fixed width
  - Not the same as address width
  - The nand2tetris project uses 16-bit slots (values) in memory

Memory Representation

- Can read and write to memory by specifying an address
  - More details next week
- Example: `x = memory[01...00]`
  - Reads the value in memory at address `01...00` and stores it in `x`
- Example: `memory[01...00] = 7`
  - Writes the value 7 in the memory slot at address `01...00`

Building Memory: Register

- Bits store a single value (0 or 1)
  - In memory, we need to store 16-bit values
- Registers are conceptually the same as a Bit
  - Allows us to store and change 16-bit values
  - Groups together 16 individual bits that share a load signal

```
// if (load(t-1)): out(t) = in(t-1)
//            else: out(t) = out(t-1)
CHIP Register {
      IN in[16], load;
      OUT out[16];
      ...
}
```

RAM: Random Access Memory

- Abstraction of Computer Memory: just a giant array

- Goal: create hardware that can provide that abstraction

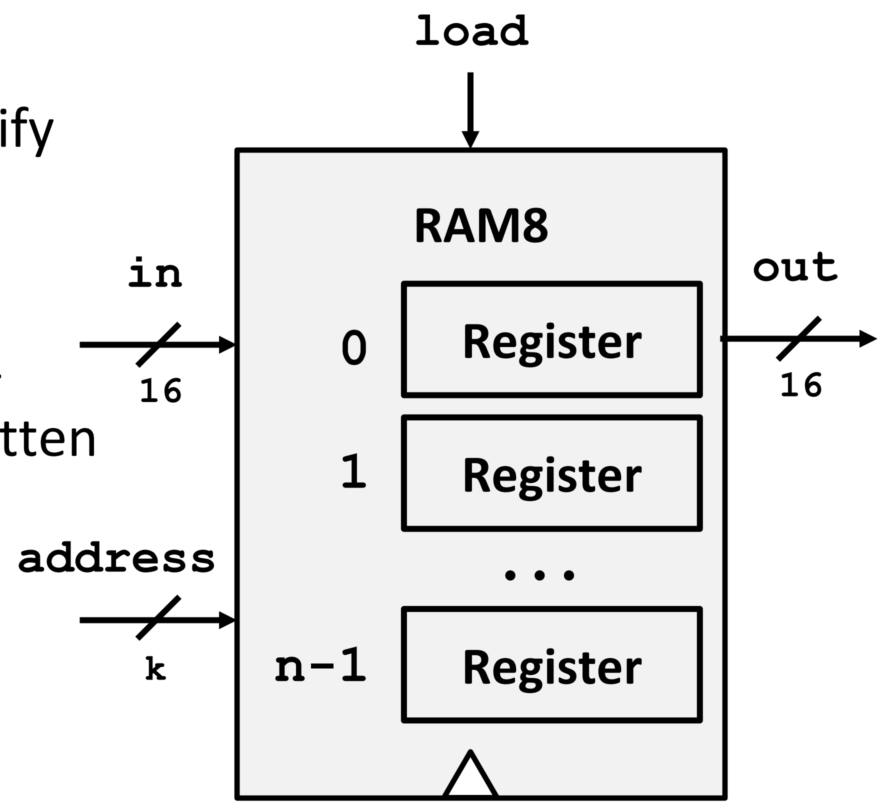
	24 11000	25 11001	26 11010	27 11011	28 11100	29 11101	30 11110	31 11111	
...	0 0000000	0 0000000	-1 1111111	25 0011001	124 1111100	0 0000000	9 0001001	-15 1110001	...

- Key attribute of arrays: "random access" lets us index into them at any point

```
memory[26] = -1;
```

Building Memory: RAM8 From Registers

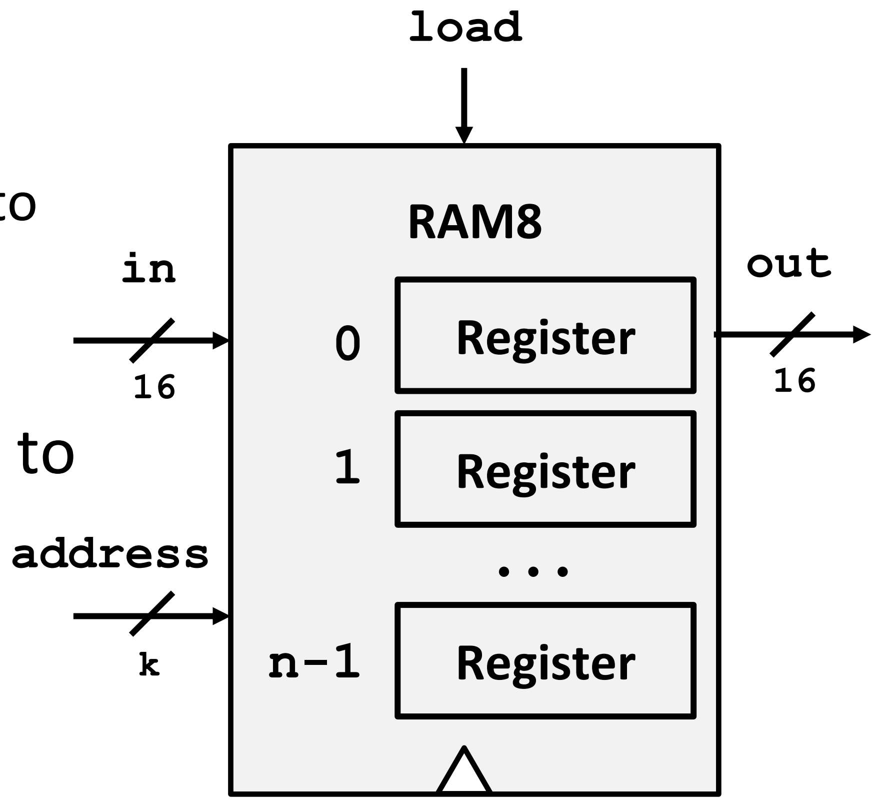
- RAM interface:
  - **`address`**: address used to specify memory slot
  - **`in`**: 16-bit input used to update specified memory slot if load is 1
  - **`load`**: if 1, then in should be written to specified memory slot
  - **`out`**: 16-bit output from the slot specified by address



- RAM8 can be built from 8 registers
  - address width is $\log_2(8) = 3$ bits

Building Memory: RAM8 From Registers

- Step 1: Route **in** to every register
  - We don't want to update every register, however
  - Solution: choose which register to enable with **address**
- Step 2: Choose which register to use for the output
- When we think about making *choices* in hardware, we want to think about Mux and DMux



Building Memory: The Rest of RAM

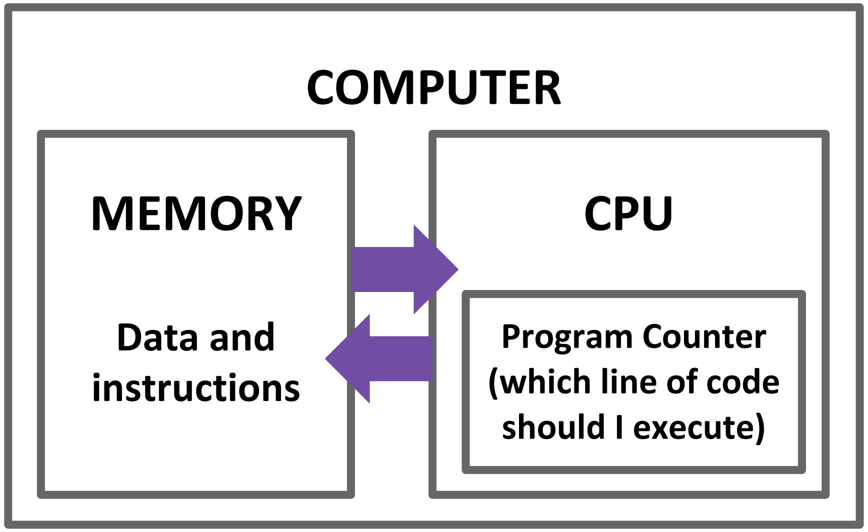
- After RAM8, can build larger RAM chips from a combination of smaller RAM chips
  - For example, RAM64 can be built using eight RAM8 chips
- Technique is similar to RAM8 but will have to use different portions of the address
- The blocks section of the reading will be helpful
  - For example, can think of each RAM8 as a block of RAM64

Lecture Outline

- Cornell Note-taking Method
  - System for Taking, Organizing, and Reviewing Notes
- Review of Sequential Logic and DFFs
- Storing Data: Bit
  - Bit Overview and Implementation
- Representing and Building Memory
  - Array Abstraction, Building From the Bit
- **Program Counter (PC) Overview**
  - **Control Flow of Computer Programs**

Program Counter (PC)

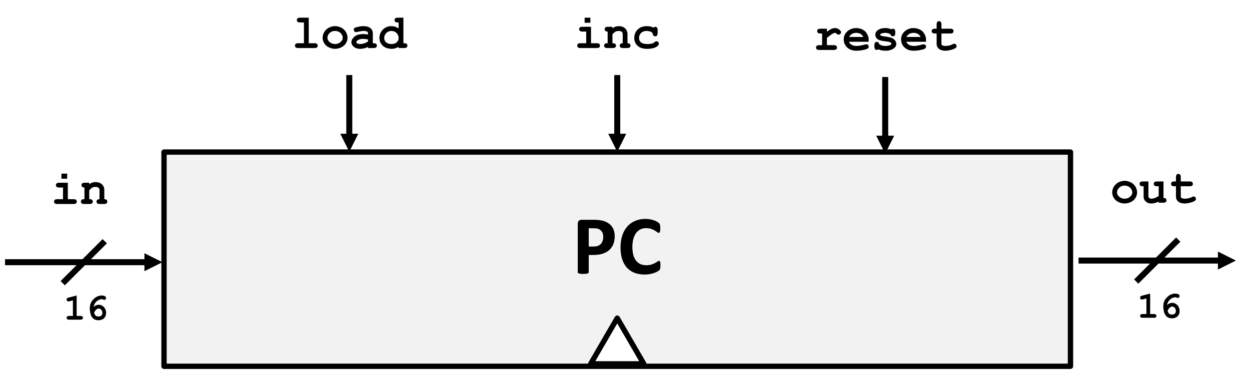
- Memory is used to store data as well as code
- Instructions and operations are stored at different addresses in memory
- Program Counter in the CPU keeps track of which address contains the instruction that should be executed next



Program Counter (PC)

- Keeps track of what instruction we are executing
  - If the PC outputs 24, on the next clock cycle the computer runs the instruction at address 24 in the code segment
- Program counter specification:

```
if      (reset[t] == 1) out[t+1] = 0
else if (load[t] == 1)  out[t+1] = in[t]
else if (inc[t] == 1)   out[t+1] = out[t] + 1
else                    out[t+1] = out[t]
```



Project 4 Overview

- Part I: Cornell Note Taking
  - Practice taking detailed notes in another class
  - Think critically about the technique
- Part II: Building Memory
  - Memory & Sequential Logic: Build our first sequential chips, from a 1-bit register to a 16K RAM module
  - Program Counter: Build counter that tracks where we are in a program, with support for several operations we'll need later
  - *Note: Folder split for performance reasons only*
- Part III: Project 4 Reflection

Post-Lecture 7 Reminders

- **Project 3 due tonight (10/20) at 11:59pm**
- Project 4 (Cornell Note Taking & Building Memory) released today, due next Thursday (10/27) at 11:59pm
- Course Staff Support
  - Eric has office hours in CSE2 153 today after lecture
  - Post your questions on the Ed discussion board