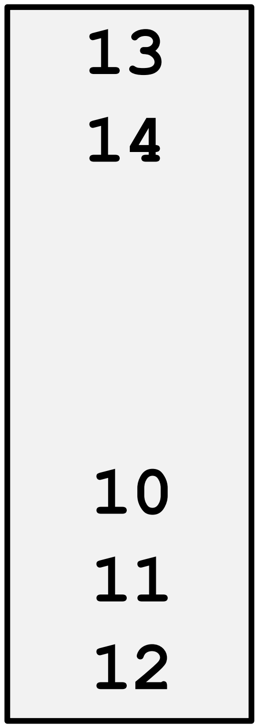
FIFOs (First-In First-Out)

- A FIFO is a first-in, first-out memory queue
- Rather than an addressable memory, where any address can be read or written to, a FIFO holds data ***in order***
  → The first word written will be the first word read
- FIFOs are used in many kinds of hardware applications where data needs to be temporarily buffered
- Your Project (Part 2) will use a FIFO to temporarily buffer output values before they are transmitted using the output AXI-Stream interface

FIFO Basic Example

(in-class animation)

- 1. Imagine we write values 5, 6, 7, 8 to the FIFO
- 2. Then we read two values – we get 5 and 6
- 3. Then concurrently write 9 and read a value (7)
- 4. Then write 10, 11, 12, 13
- 5. Then read (8)
- 6. Then write 14 while reading (9)



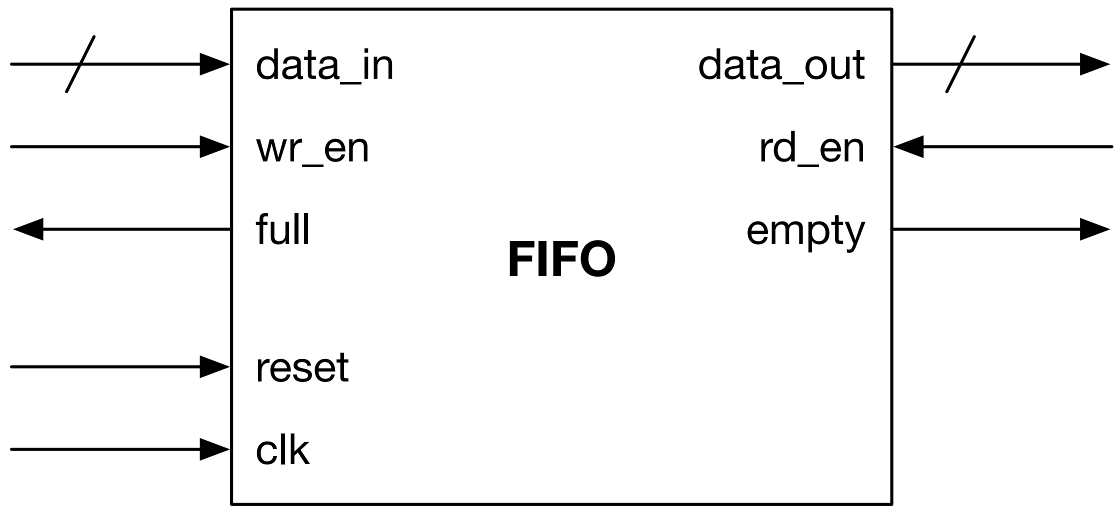
- Basic idea: the FIFO’s memory holds values; the FIFO contains control logic and registers to keep track of addresses for writing and reading

FIFO Parameters and Options

- We a FIFO, we need to define:
  - WIDTH: how many bits per word?
  - DEPTH: how many words can the FIFO hold?

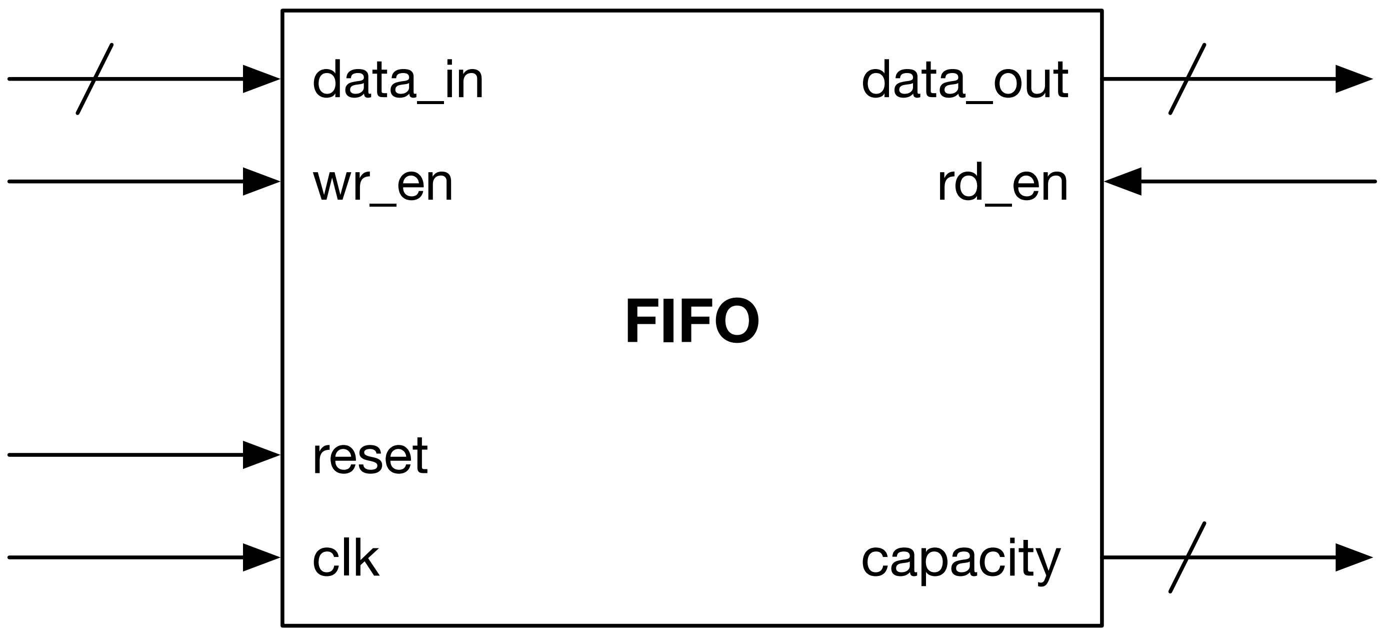
- We also need to define its interfaces
  - Some examples on the next slides
  - but many variations are possible

Example FIFO Interface (1)



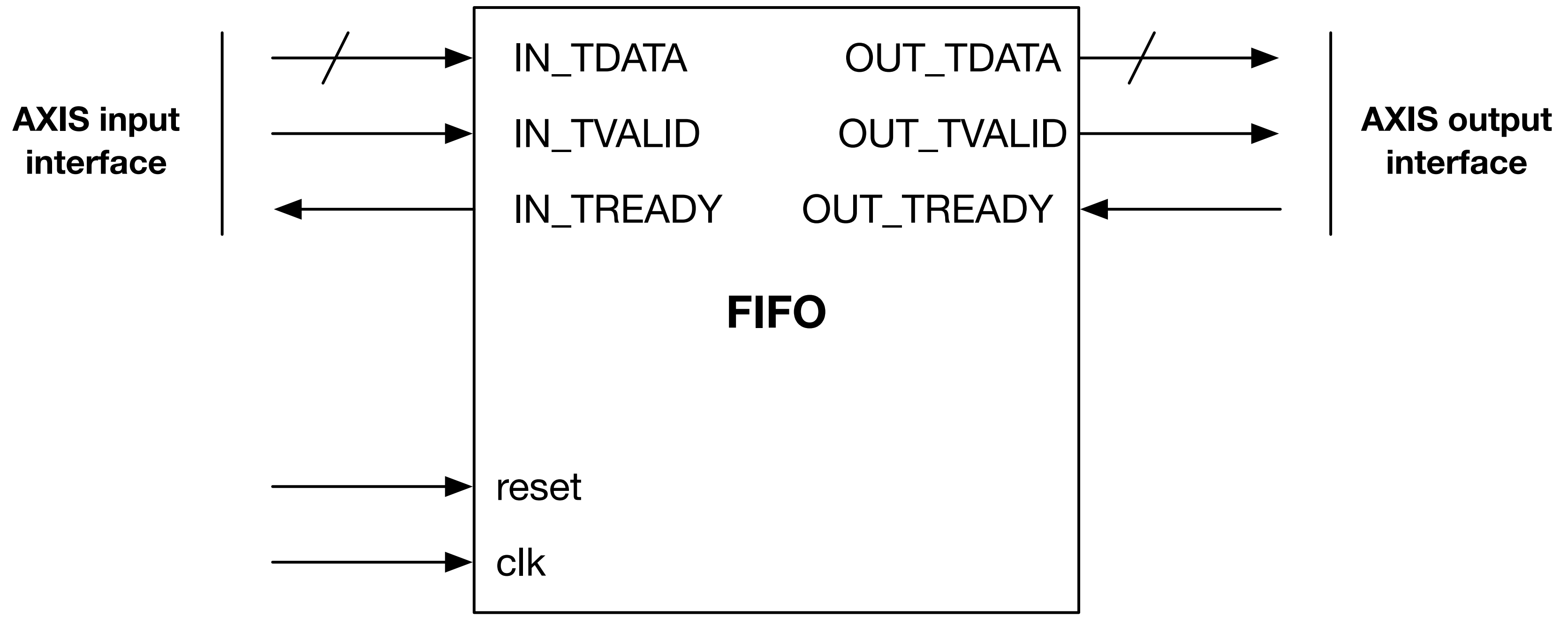
- If (wr_en==1) on a positive clock edge, then value on data_in is written to the FIFO
- If (rd_en==1) on a positive clock edge, then the oldest value stored in the FIFO is read on the data_out port
- empty and full signals show when the FIFO is empty/full
- Don’t write when it is full! Don’t read when it is empty!
  - What happens if you try to?

Example FIFO Interface (2)



- Only difference: a capacity signal that shows how much space is in the FIFO:
  - If capacity == DEPTH, the FIFO is empty
  - If capacity == 0, then the FIFO is full
- This requires slightly more complex logic than the previous example, but gives more information to other modules

Example FIFO Interface (3)



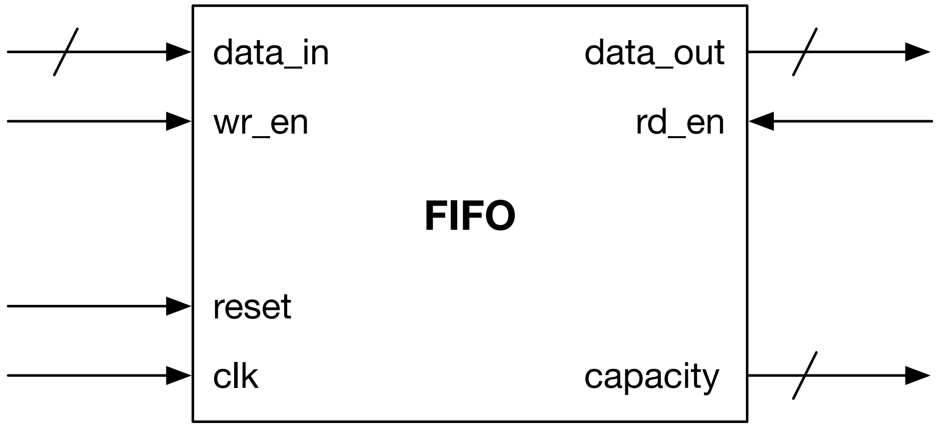
- Uses AXI-Stream interfaces (see Topic 6 for AXI-Stream)
- Very similar to first example:
  - Assert IN_TVALID to write
  - IN_TREADY == 0 if FIFO is full, otherwise 1
  - Assert OUT_TREADY to read
  - OUT_TVALID == 0 if FIFO is empty, otherwise 1

So How Do We Build a FIFO?

- First, we use a memory for the storage
- Then, we use registers to keep track of the memory addresses we should read and write from
- Then, we add logic to deal with empty/full or capacity or TVALID/TREADY signals, as needed
- Many options for how to do each of these steps
    - Let’s look at the logic for Example 2 from slide 25

Detailed FIFO Design (Example 2)

- Assume the FIFO has WIDTH=16 and DEPTH=256 (although it should be easy to generalize this for any parameter values)

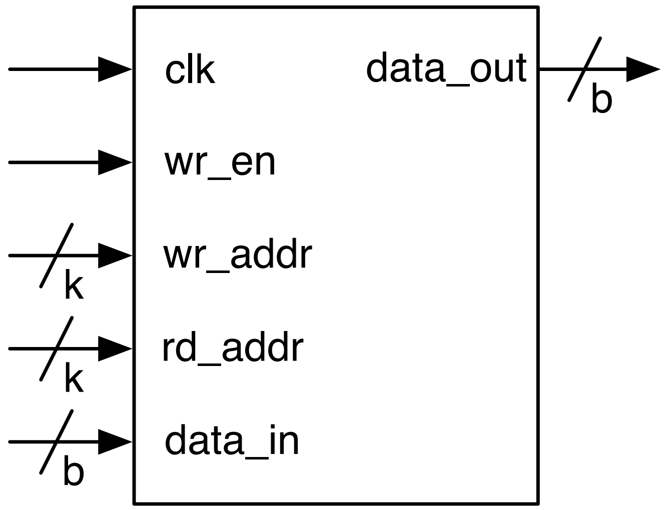


- So step 1: let's use a memory with DEPTH entries and WIDTH bits per entry

- Hmm… we talked about many varieties of memory before. Which should we use?
  - Any of them *could* work correctly
  - An easy choice: let's use the dual-port memory with bypass

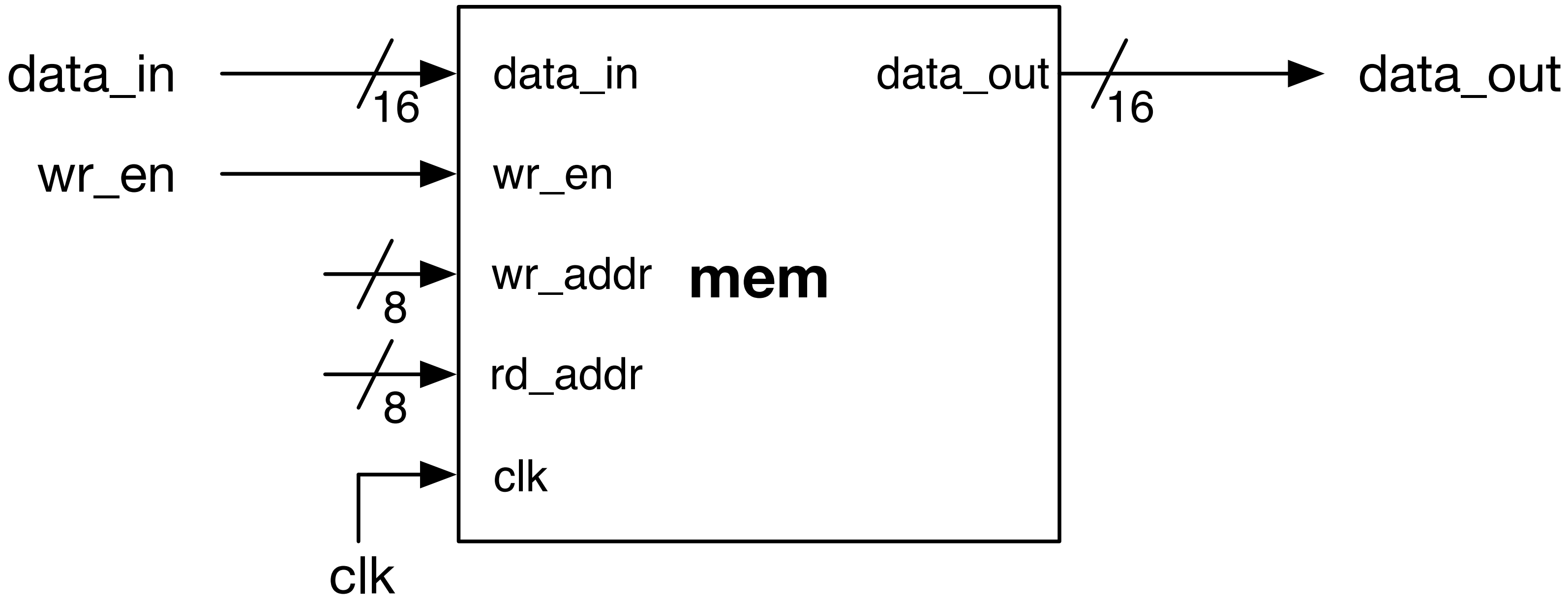
Recall: Dual-Port Memory with Bypass

```
module memory_dual_port #(
      parameter  WIDTH=16, SIZE=64,
      localparam LOGSIZE=$clog2(SIZE)
   )(
      input [WIDTH-1:0] data_in,
      output logic [WIDTH-1:0] data_out,
      input [LOGSIZE-1:0] wr_addr, rd_addr,
      input clk, wr_en);

   logic [SIZE-1:0][WIDTH-1:0] mem;

   always_ff @(posedge clk) begin
      data_out <= mem[rd_addr];

      if (wr_en) begin
         mem[wr_addr] <= data_in;
         if (rd_addr == wr_addr)
            data_out <= data_in;
      end
   end
endmodule
```



Step 1: Memory

- One instance of the memory from the previous slide, with WIDTH=16 and SIZE=256



- What's missing? The addresses and the capacity logic

FIFO "Head" and "Tail"

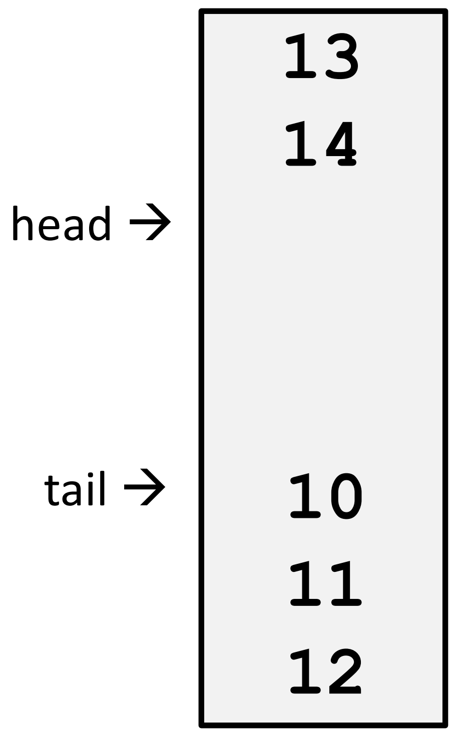
- The ***head*** and ***tail*** are addresses which we will use to keep track of where valid data is in the FIFO
  - held in registers

- ***head***: holds the next "free" address in the RAM
  - The next time we write, we write at the head address
- ***tail***: holds the oldest valid item stored in the FIFO
  - the FIFO should read from the tail

- Let's revisit our prior example, now with the head and tail included

FIFO Basic Example with Head and Tail

- 1. Write 5, 6, 7, 8
- 2. Read twice (5, 6)
- 3. Write 9 and read (7)
- 4. Write 10, 11, 12, 13
- 5. Read (8)
- 6. Write 14 and read (9)



(in-class animation)

The head represents the write address
The tail represents the read address

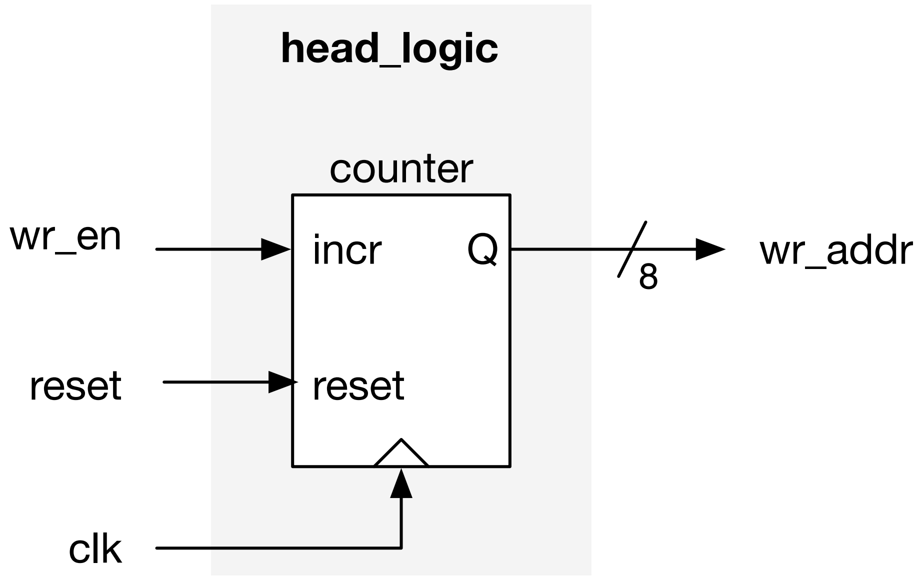
(Note: the read address is slightly more complex—we'll come back to this)

Head (Write Address) Logic

- The memory's write address is the head value

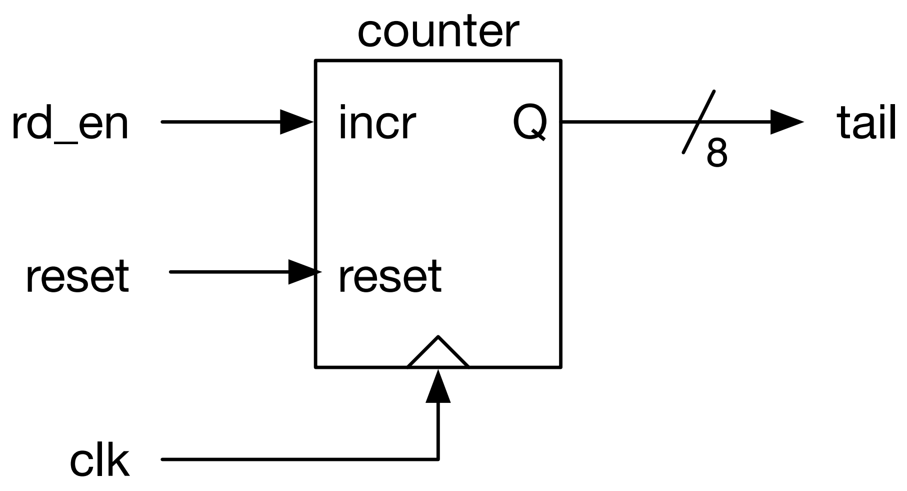
- The head is held in a simple counter:
  on a positive clock edge, if wr_en==1, then increment head
  - Some added complexity here if DEPTH is not a power of two—I'll show this later

```
// assuming DEPTH is power of 2
always_ff @(posedge clk) begin
   if (reset == 1)
      wr_addr <= 0;
   else if (wr_en == 1)
      wr_addr <= wr_addr+1;
end
```



Tail (Read Address) Logic

- The tail register works the same was as the head:
  if rd_en is 1 on a positive clock edge, increment the tail

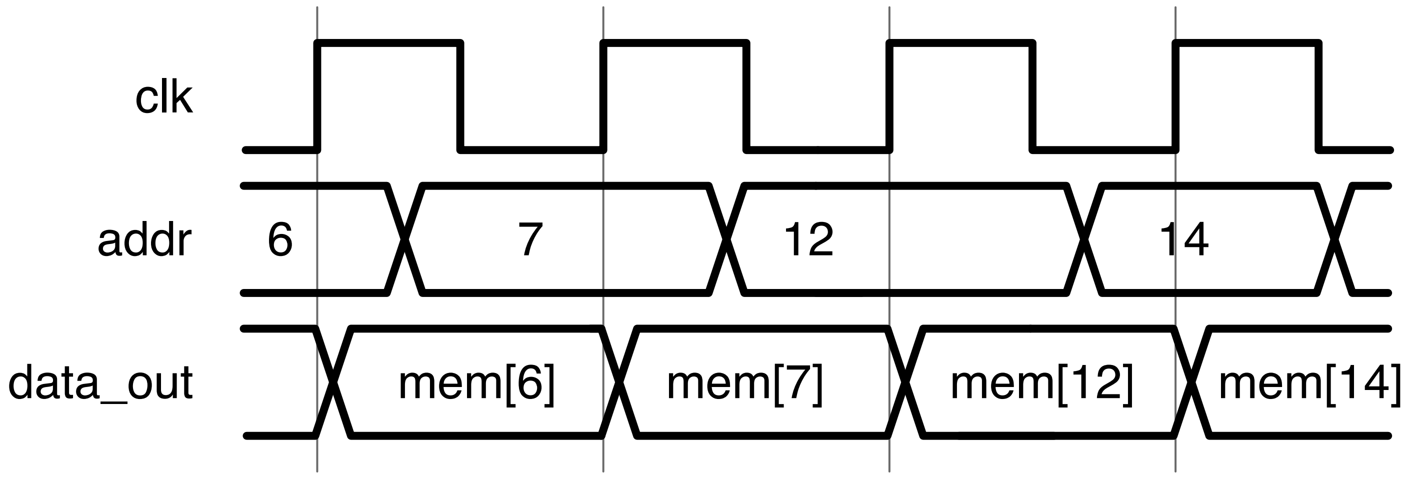


- However, we can't directly use the tail value as the read address. Why? Because the read address tells the memory what value to read ***on the next positive clock edge***
  - (More on next slide)

Remember: Sequential Memory Reads

- Our memory has sequential reads

- The address I give it ***now*** determines what the memory will output at the ***next positive clock edge***

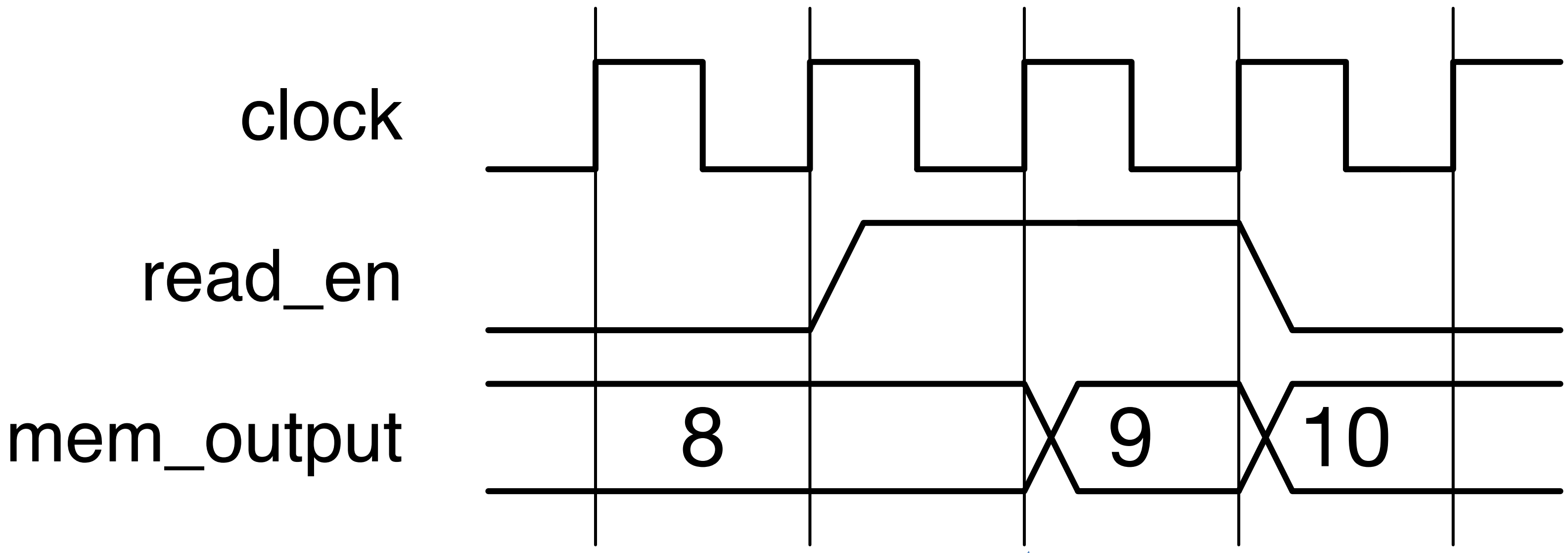


- This creates a problem in the FIFO: we need to set the read address to point to the data you will want ***on the next positive clock edge***

Example and Desired Behavior

- Assume DEPTH=4 and we start with TAIL=1
- Let's look at the expected output given the input

address	data
0	11
1	8
2	9
3	10



the FIFO already outputs 8 even though we aren't yet ready to read it

here we read 8

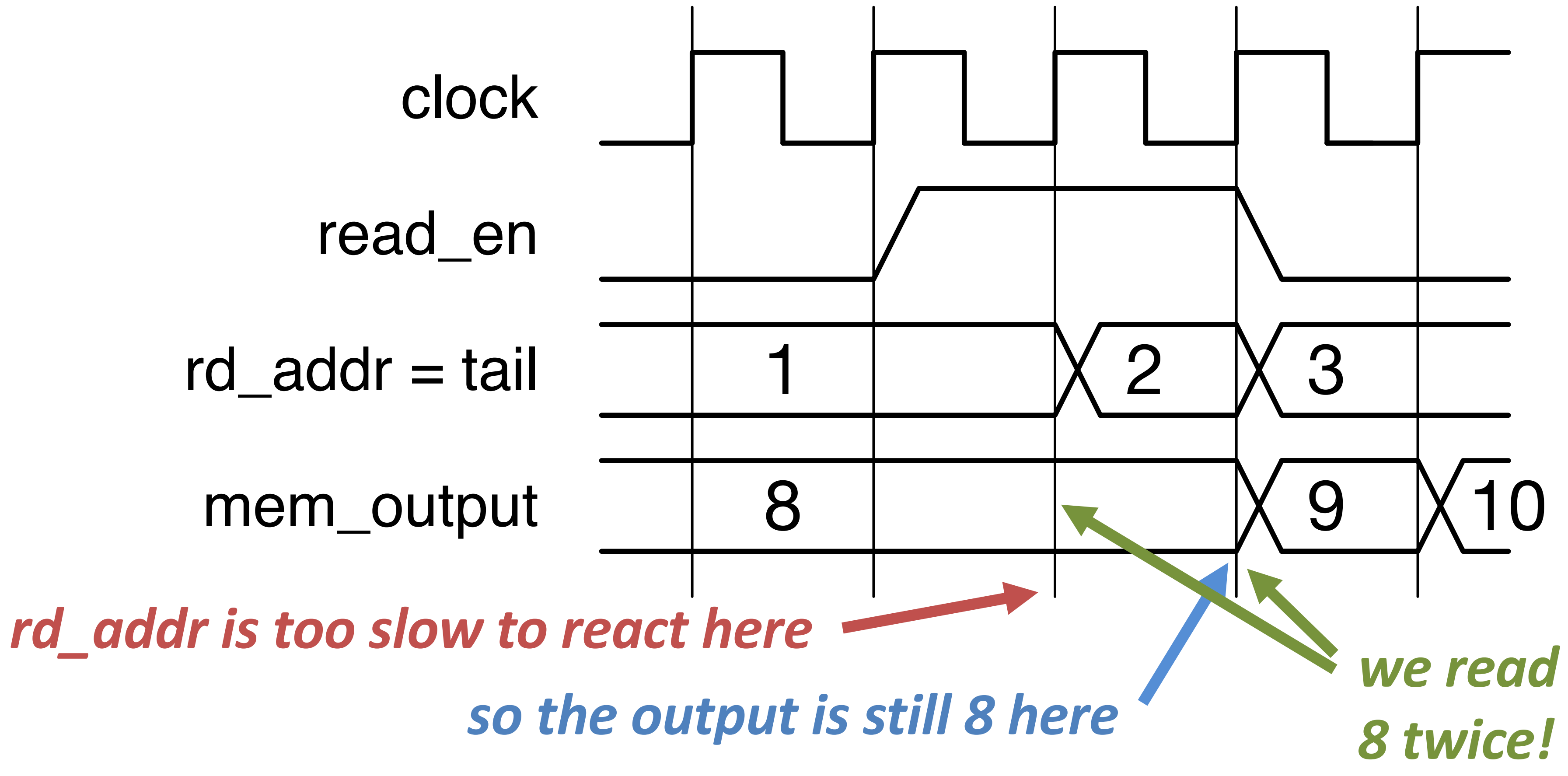
here we read 9

the FIFO prepares 10

What Would Go Wrong if We Set the Read Address = Tail?

- If we use the tail as the read address, we get the data too late!
- Here, we incorrectly read 8 twice!

address	data
0	11
1	8
2	9
3	10


clock
read_en
rd_addr = tail
1
2
3
mem_output
8
9
10
rd_addr is too slow to react here
so the output is still 8 here
we read 8 twice!

Read Address Logic

- The trick: if I am reading right now, I need to set the read address to tail+1 to prepare the output data for ***the next clock cycle***

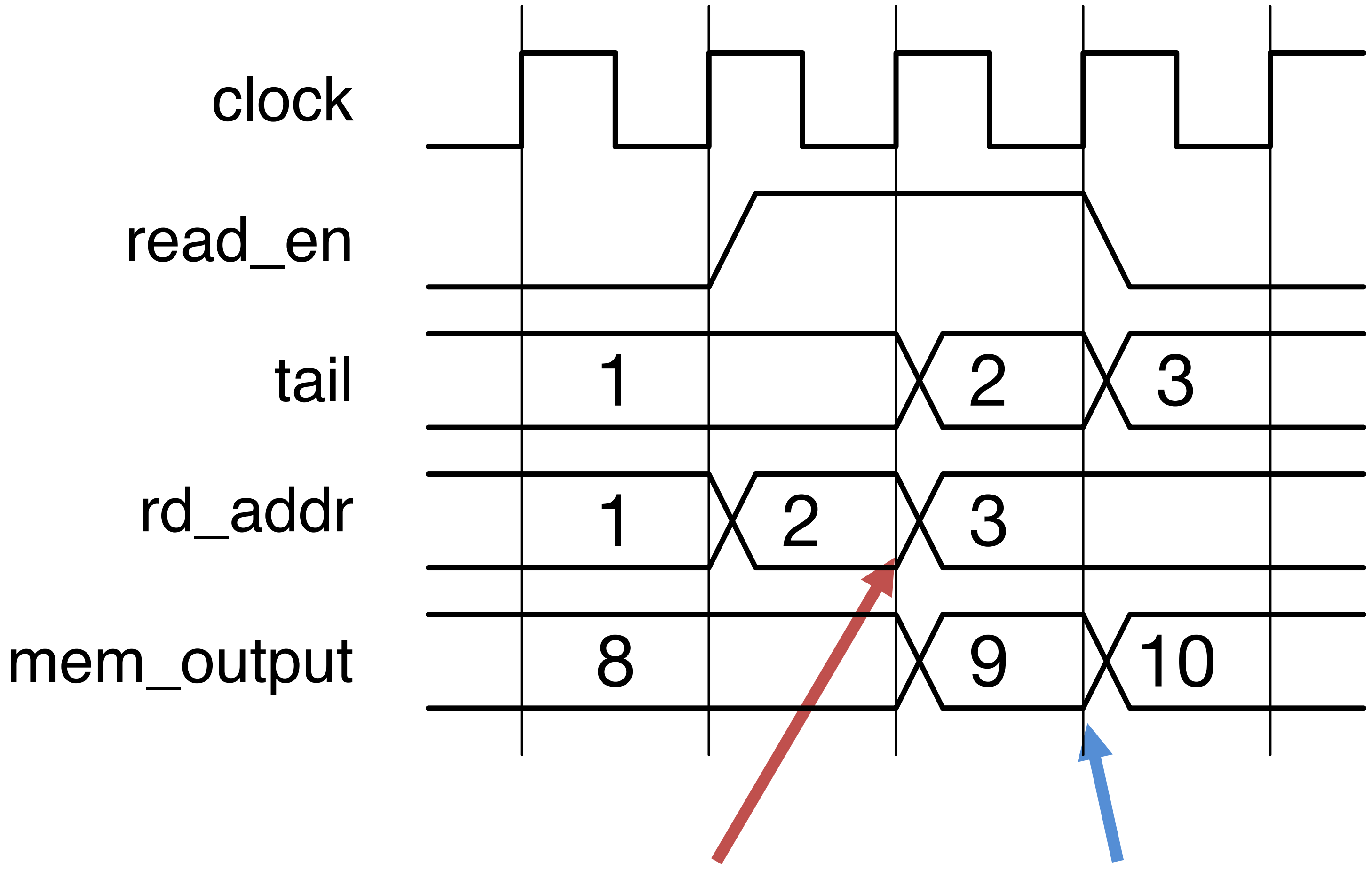
- The solution is the following combinational logic:

```
// assume DEPTH is a power of two
always_comb begin
   if (rd_en == 0)
      rd_addr = tail;
   else
      rd_addr = tail+1;
end
```

- Visual example on next slide

Correct Read Address Timing

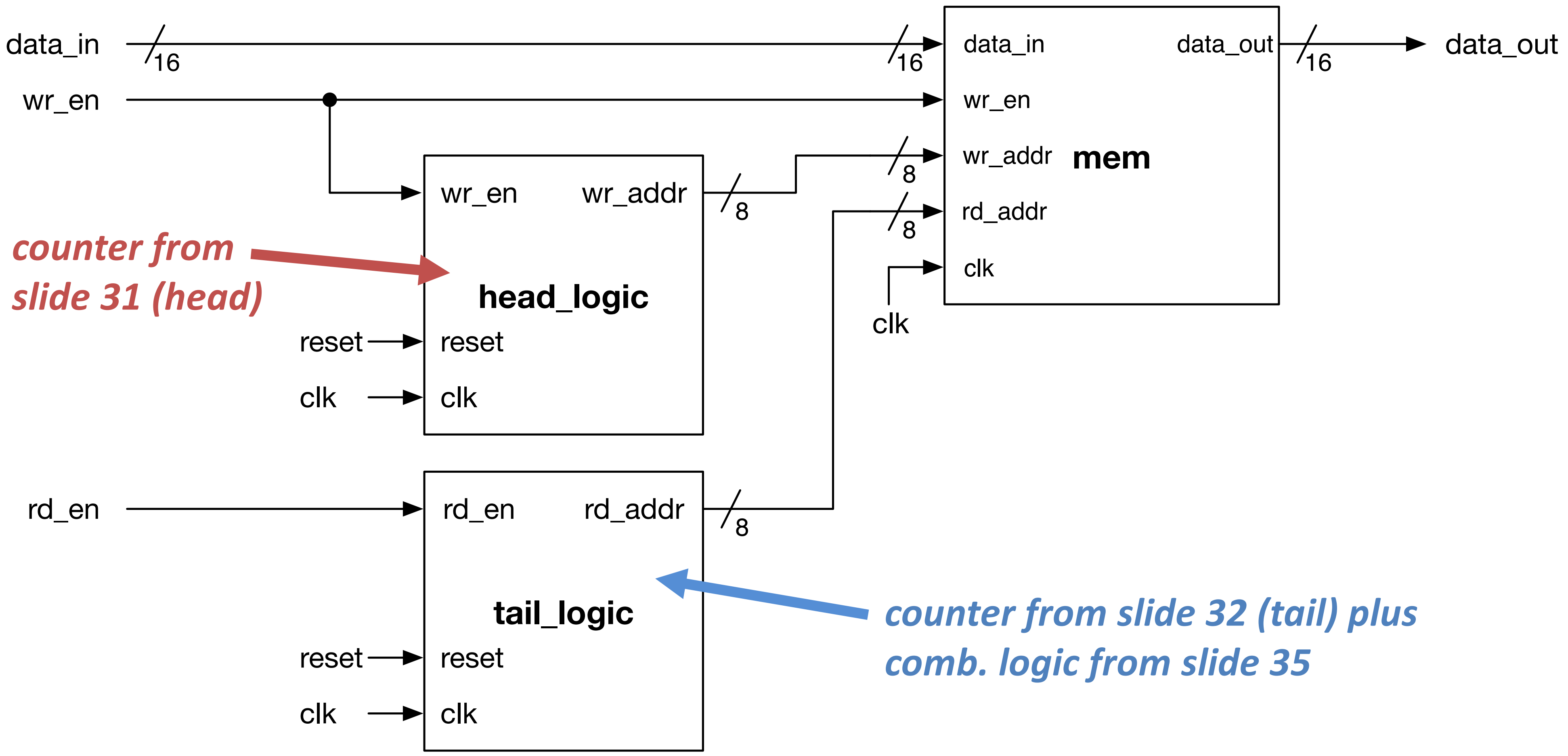
address	data
0	11
1	8
2	9
3	10



rd_addr changes here

so the output is correct here

FIFO With Head and Tail Logic



- What's missing? The logic to track the capacity

Capacity

- Remember, FIFOs have many ways they can provide status information

- Some can be quite simple — e.g., you can determine whether a FIFO is empty or full by comparing the head and tail

- In this example, we will track the ***capacity***, which shows the amount of free space in the FIFO
  - capacity == 0 → the FIFO is full
  - capacity == DEPTH → the FIFO is empty

- The FIFO in your project uses capacity (because this extra status information will be helpful in Parts 4/5)

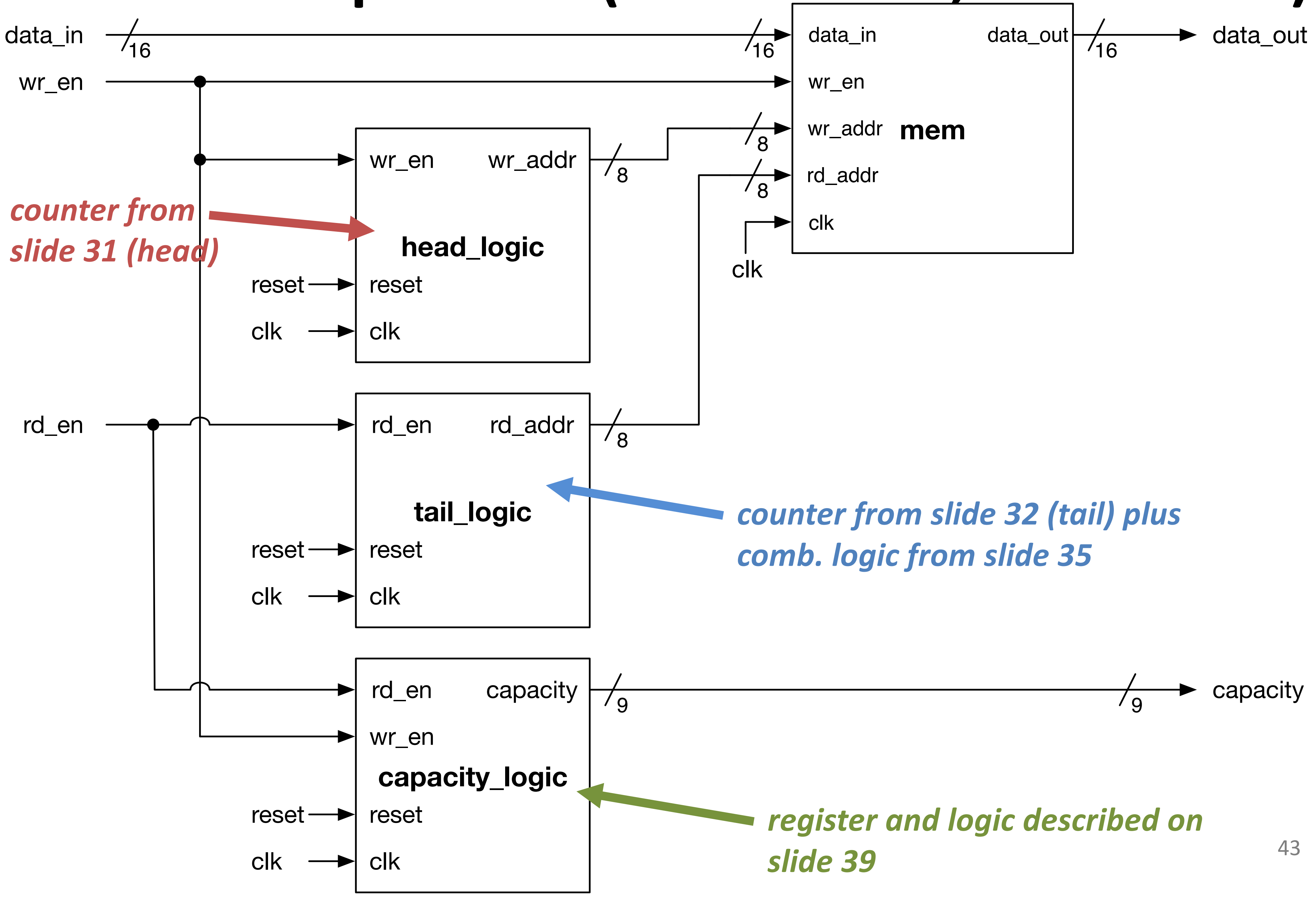
Capacity

- The logic for keeping track of the capacity is easy.

- Use a register with $clog2(DEPTH+1) bits
  - Why DEPTH+1? Because the capacity is between 0 and DEPTH (not 0 and DEPTH–1)

- When you reset the system, initialize the register to DEPTH
- On every positive clock edge, check the values of rd_en and wr_en:
  - If you are reading, but not writing, increase the capacity by 1
  - If you are writing, but not reading, decrease the capacity by 1
  - If you are reading and writing, don’t change the capacity
  - If you are doing nothing, don’t change the capacity

Final Example FIFO (DEPTH=256, WIDTH=16)



counter from slide 31 (head)

counter from slide 32 (tail) plus comb. logic from slide 35

register and logic described on slide 39

What if DEPTH Is Not a Power of Two?

- In our FIFO, there are several places where we compute head+1 or tail+1
  - In the logic for the head register, the tail register, and the read combinational logic
- We have to make sure these values “wrap around”
  - if head == DEPTH–1, then we need to make sure that head+1 → 0
- If DEPTH is a power of two, this is easy
  - E.g., if DEPTH=256, then head is 8 bits, so 255+1 → 0
- But if DEPTH is not a power of two, we need to check, e.g.:

```
if (wr_addr == DEPTH-1)
   wr_addr <= 0;
else
   wr_addr <= wr_addr+1;
```

Memories in Your Project

- You will use memories in two places in your project:

- In Part 2, you will build a FIFO similar to the one we just discussed, except its output will use AXI-Stream
  - Use the same memory from our example (dual-port with bypass)

- In Part 3, you will build the “input memory” module that uses two memories (one for the matrix and one for the sparse vector)