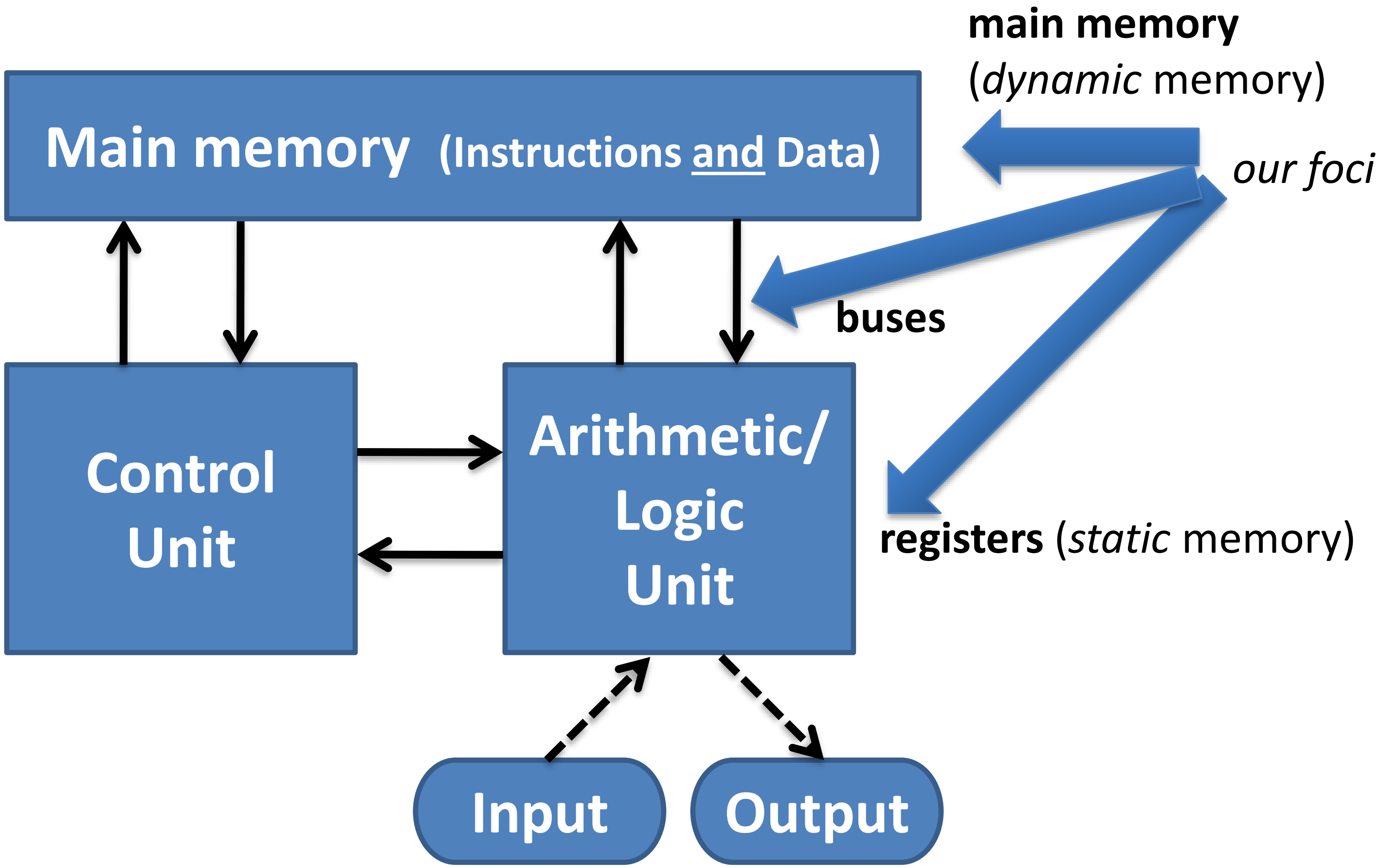
Lancaster University

# SCC131: Digital Systems

## Topic 7: Building memory

(and: connecting it to the rest of the system)

# Reminder of the von Neumann architecture



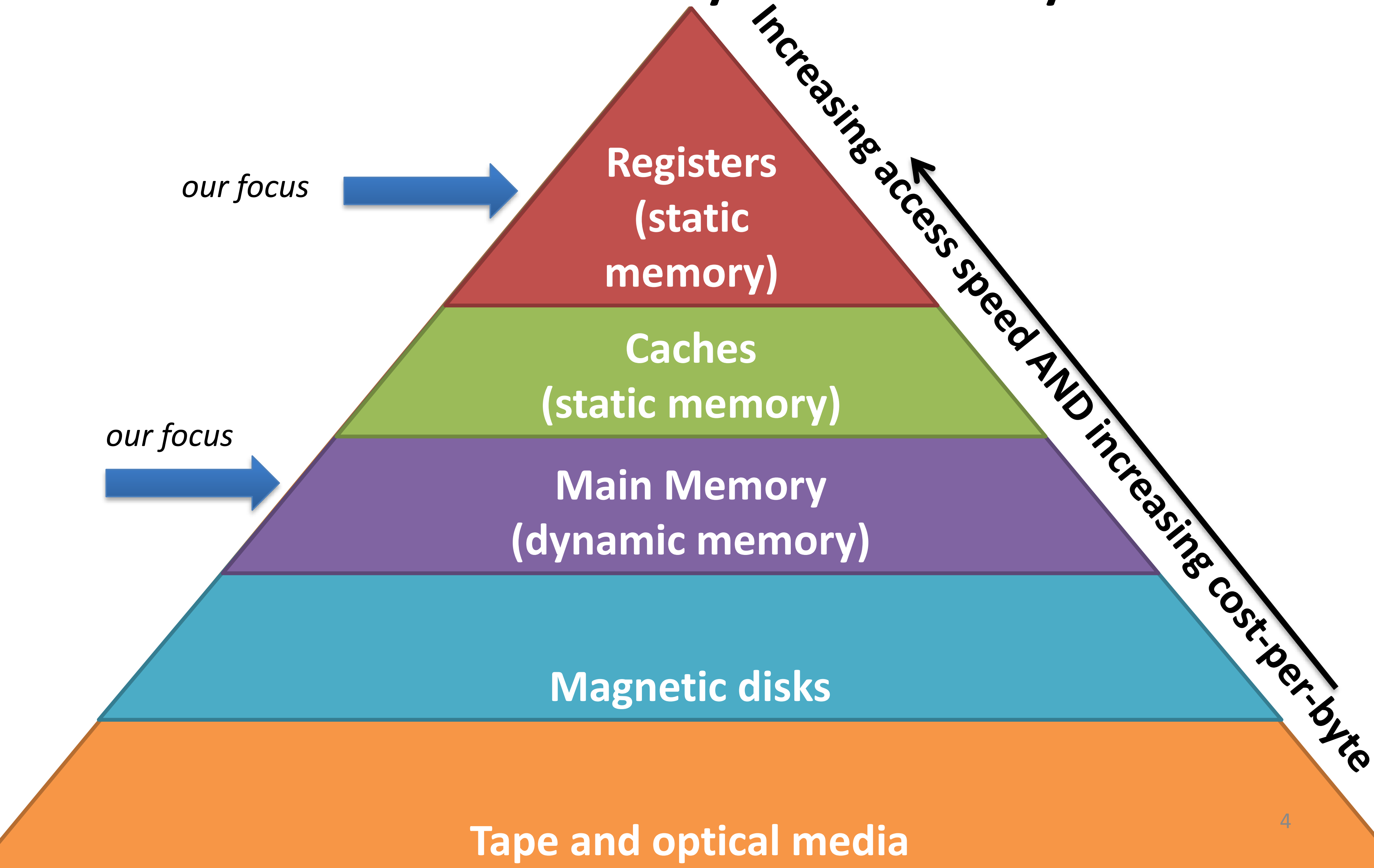
# What does memory do?

- Stores bits :-)

- We are discussing *volatile* memory here (i.e. not disks or read-only memory – *non-volatile* memory )

- There are two types of volatile memory
  - Dynamic memory
    - Used for main memory
    - Slower than static memory, but relatively cheap
  - Static memory
    - Used for registers and caches
    - Fast but relatively expensive

# The memory hierarchy

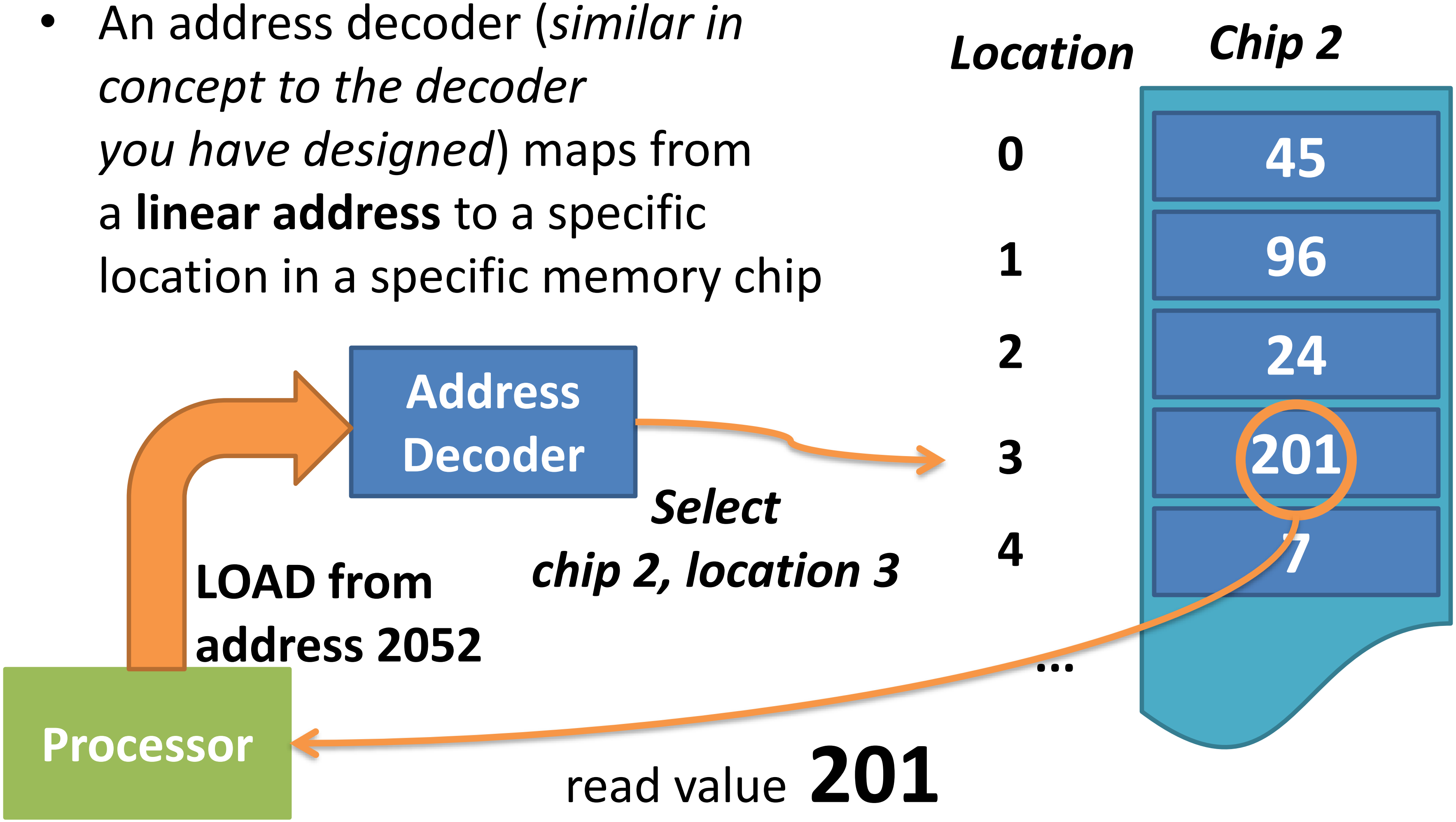


# Organising main (dynamic) memory

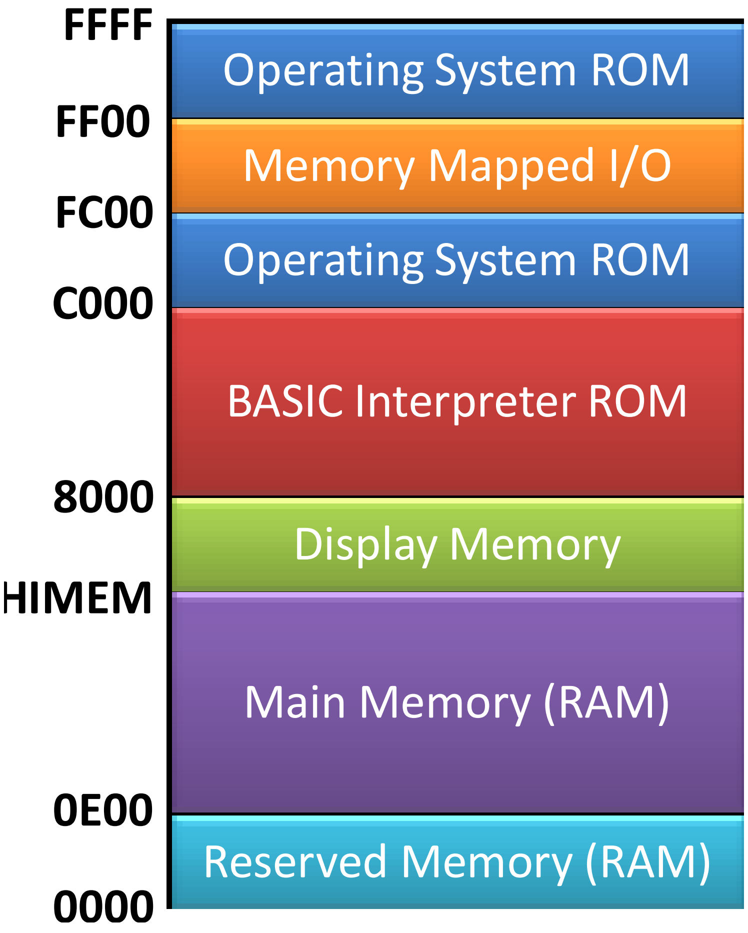
- Abstractly, main memory looks just like “pigeon holes”
  - Each “hole”, or *location*, holds one unit of information or data
  - Each location has an “address” for identification
- Processors usually generate a so-called **linear address**, 0…n
  - In implementation, this typically maps to multiple memory chips

# Address decoding

- An address decoder (*similar in concept to the decoder you have designed*) maps from a **linear address** to a specific location in a specific memory chip

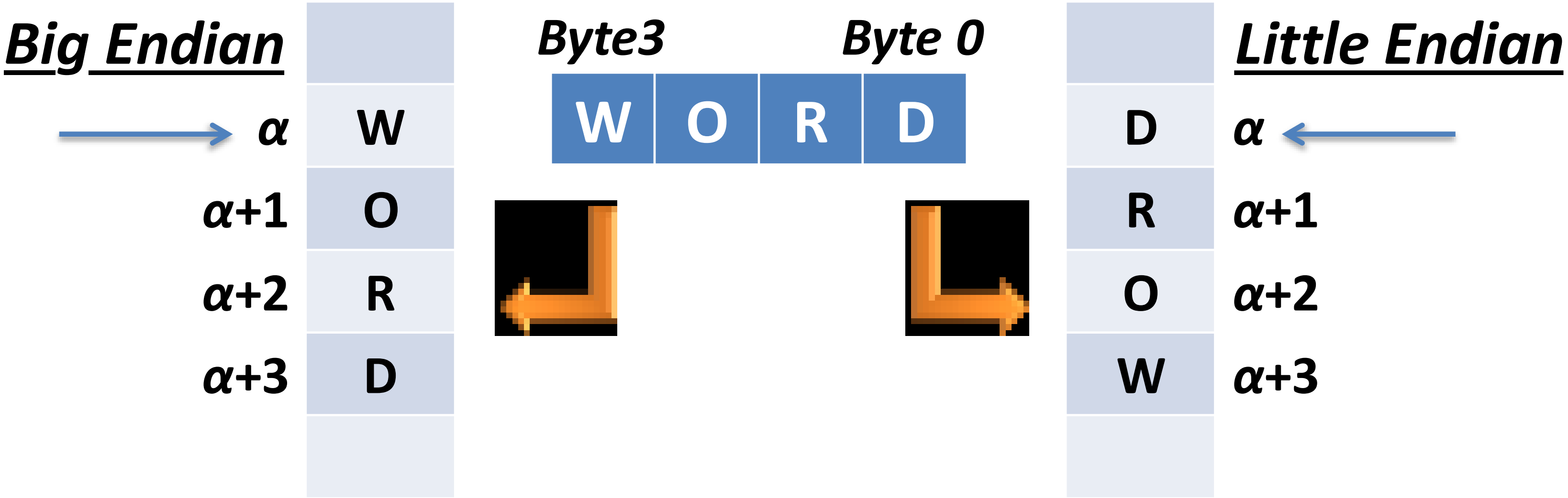


# An example ‘memory map’ for main memory



# Byte ordering in multi-byte words

- Different machine architectures may organise multi-byte words differently in memory
  - *Big-endian*: within a multi-byte word (e.g. 4 bytes for a 32-bit integer), the location (byte) with the lowest memory address holds the **most**-significant-byte (MSByte)
  - *Little-endian*: within a multi-byte word (e.g. 4 bytes for a 32-bit integer) the location (byte) with the lowest memory address holds the **least**-significant-byte (LSByte)



***Big Endian***

*α* W

*α*+1 O

*α*+2 R

*α*+3 D

***Byte3*** ***Byte 0***

W O R D

D *α*

R *α*+1

O *α*+2

W *α*+3

***Little Endian***

# How can we tell if we are running on a big-endian or little-endian machine?

```
/* Here is a C function that returns 1 if the
 * machine it is running on is big endian,
 * or 0 if it is little endian
 */

int endian()
{
  int i = 1; char *p = (char *)&i;

  /* p points to the byte at the leftmost end of
   * the 0000000000000000000000000000000001 word
   * (i.e. lowest address);
   * if this holds 0, we are on a big endian
   * machine; otherwise little endian
   */
  if(*p==(char)0) return 1; else return 0;
}
```

# Static memory

- As we’ve previously seen, stored bits are organised into multi-bit storage slots called *registers*
- We use networks of logic components to build storage for individual data bits
- But how can a collection of, say, NAND gates create memory???

***** combinatorial logic → sequential logic *****

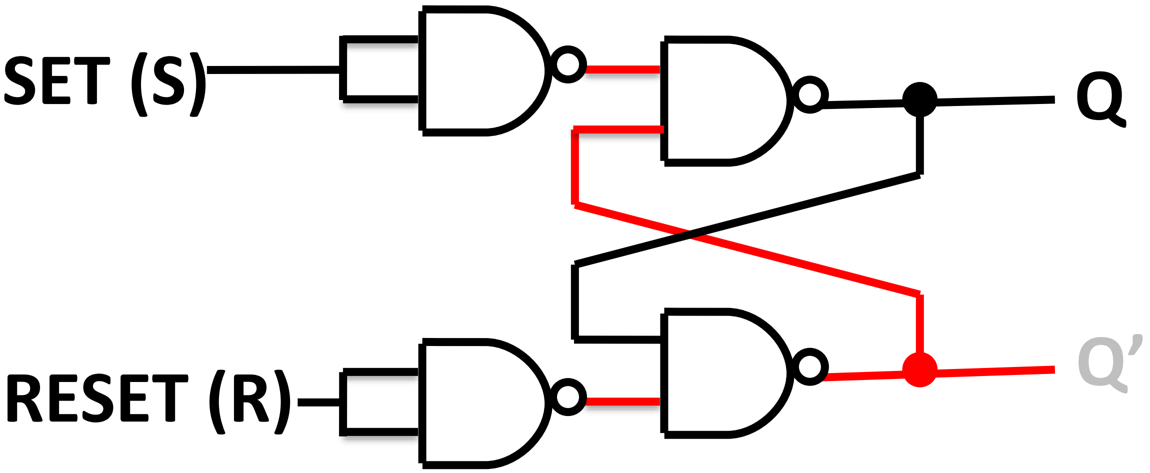
# Combinatorial logic and sequential logic

- The defining feature of combinatorial logic is that its outputs are **purely a function of its inputs**
- The defining feature of sequential logic is that its outputs are **a function of its inputs <u>and</u> of its current outputs**
  - This involves **feedback** of the outputs to the inputs
  - This feedback is the hook on which we hang memory
  - We have looked at combinatorial logic design; we now look at sequential logic design in the following slides

# A simple sequential logic circuit: the S-R flip-flop

| $Q_0$ | S | R | Q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |

- This “remembers” which of 2 possible *states* it is in
  - $Q_0$ is the ‘present’ state; Q is the ‘next’ state
- Q depends on the S and R inputs:
  - A high pulse on **S** always takes us to the Q=1 state (set)
    A high pulse on **R** always takes us to the Q=0 state (reset)
    (we’re “not allowed” to have S and R both high simultaneously)
  - When S or R returns to low after the high pulse, Q **stays where it is**
    - and so the flip-flop “remembers” if it is in state Q=1 or state Q=0

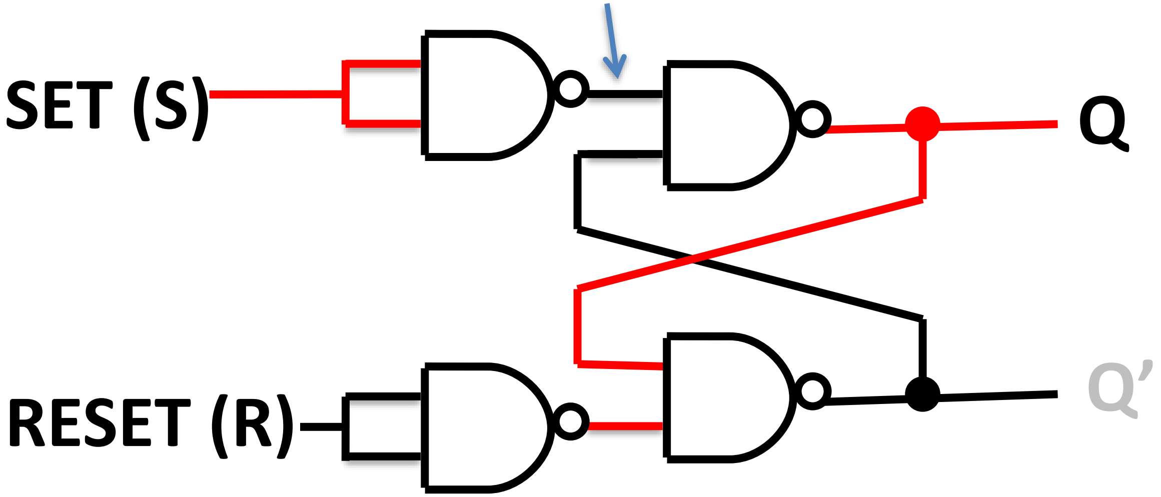


RED = high

# Setting an S-R flip-flop

| $Q_0$ | S | R | Q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |

Forced low by set – forces the output of the following NAND to be high

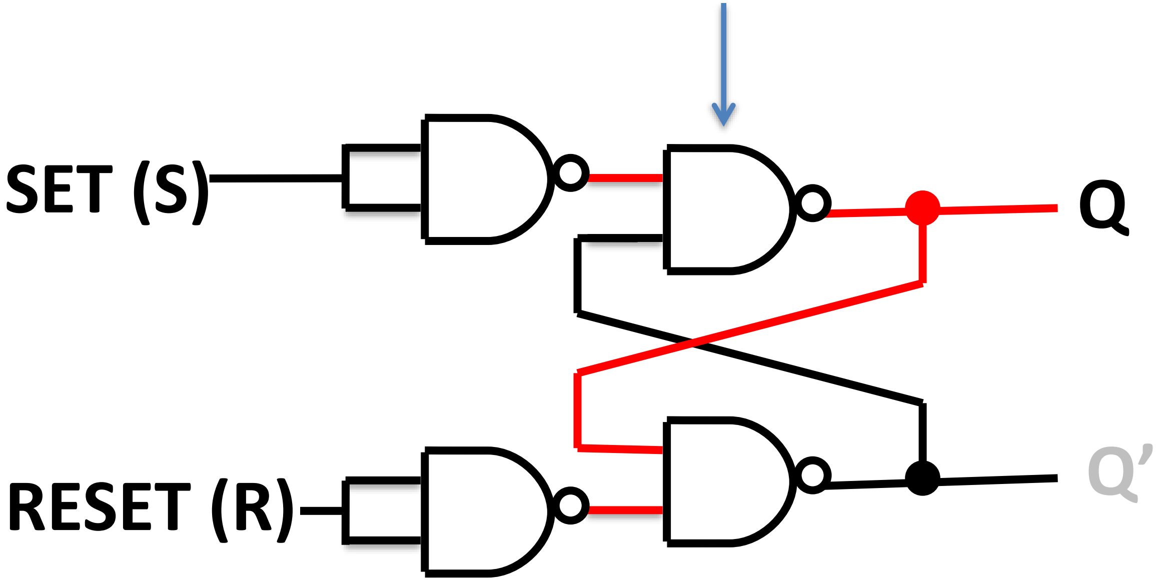


(Recall that it takes *two high inputs to force the output of a NAND gate low* – every other combination results in a high output)

# Post-setting

| $Q_0$ | S | R | Q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |

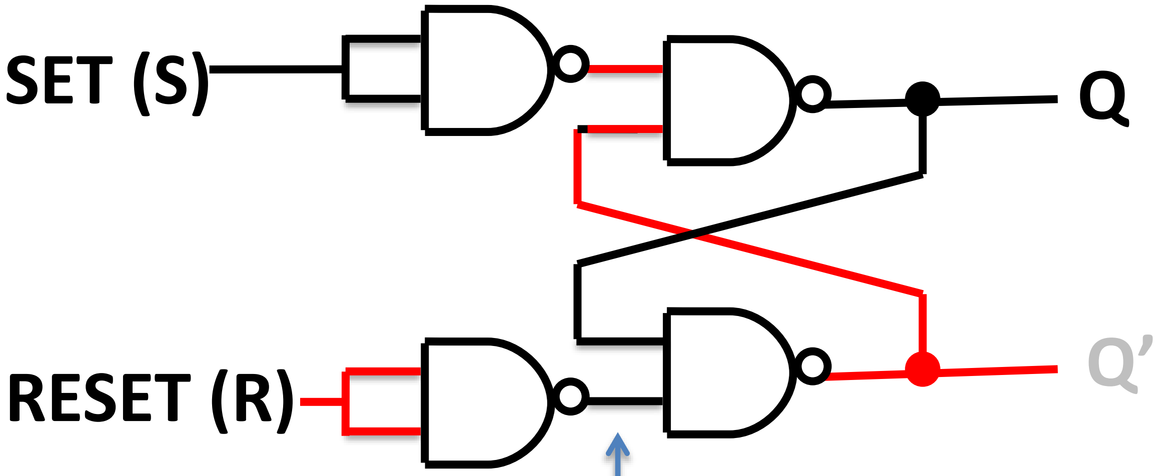
Output of the NAND **stays** high
(would need *both* inputs high to force it low)



(Recall that it takes *two high inputs to force the output of a NAND gate low* – every other combination results in a high output)

# Resetting an S-R flip-flop

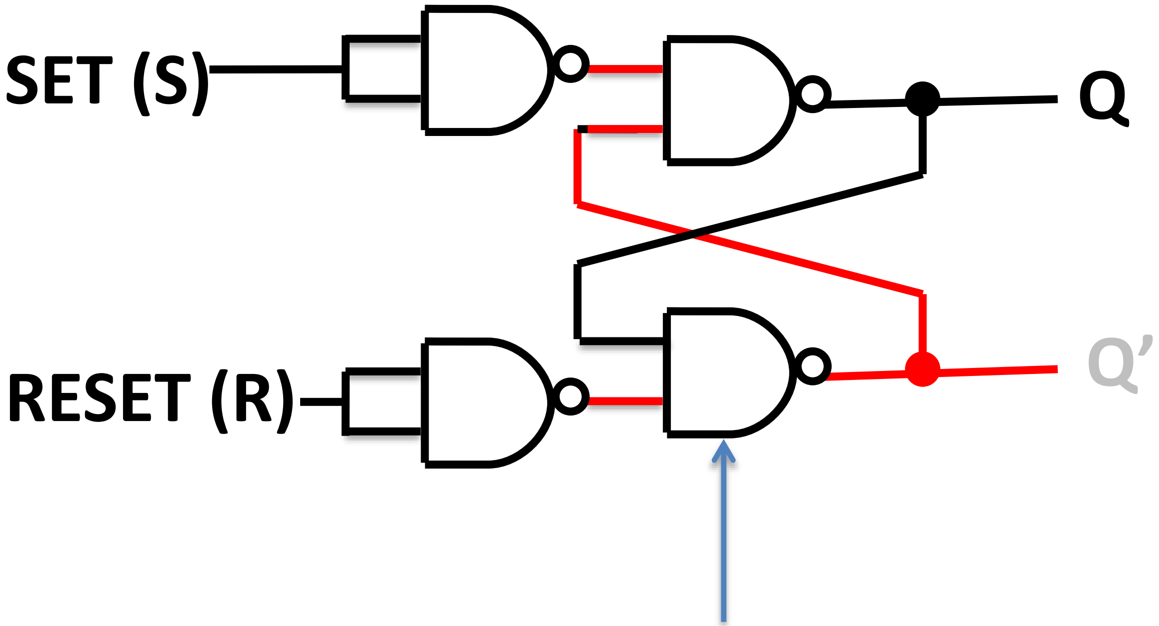
| $Q_0$ | S | R | Q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |



Forced low by reset – forces the output of the following NAND to be high

# Post-resetting

| $Q_0$ | S | R | Q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |



Output of the NAND **stays** high
(would need *both* inputs high to force it low)

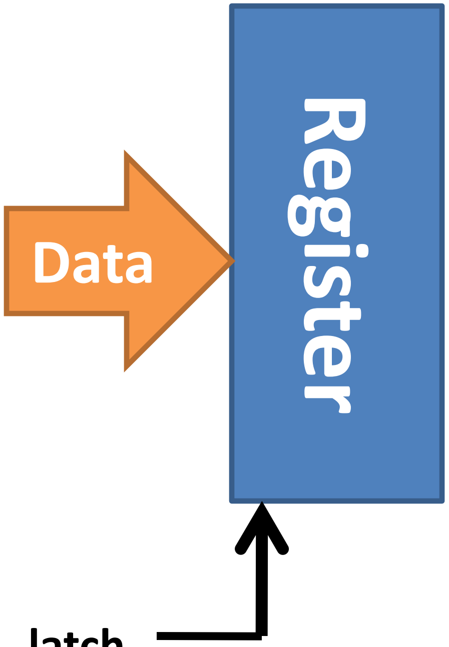
# Limitations of the S-R flip-flop

- There are two problems in using an S-R flip-flop to implement a bit in a register

1. It has distinct SET and RESET inputs
   – We'd ideally like just a **single input** that "sets" the state if it's 1 and "resets" the state if it's 0

2. We have no way of telling the flip-flop exactly **when** it should store input data
   – We'd like a "latch" signal for this to work in a practical system under the supervision of a control unit

- Enter the *Clocked D-type flip-flop*…



(Store input data ***now***)

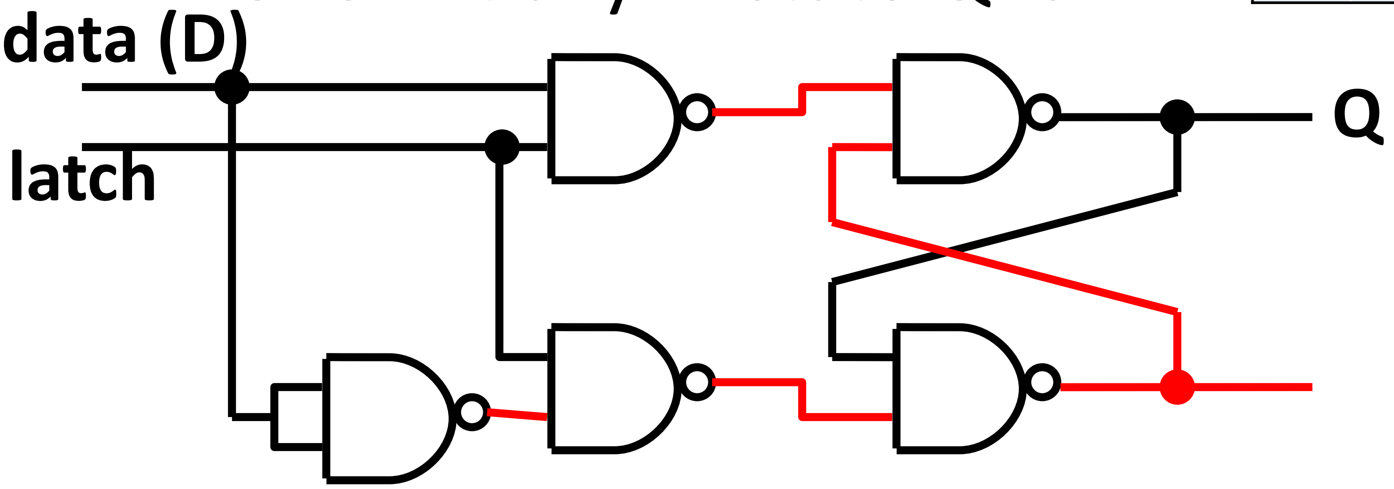
# Clocked D-type flip-flop

- To understand how this works, let’s consider the two distinct cases of latching a 1, and then latching a 0…
- We’re initially in state Q=0…

| $Q_0$ | D | L | Q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

} “latch” a 0

} “latch” a 1



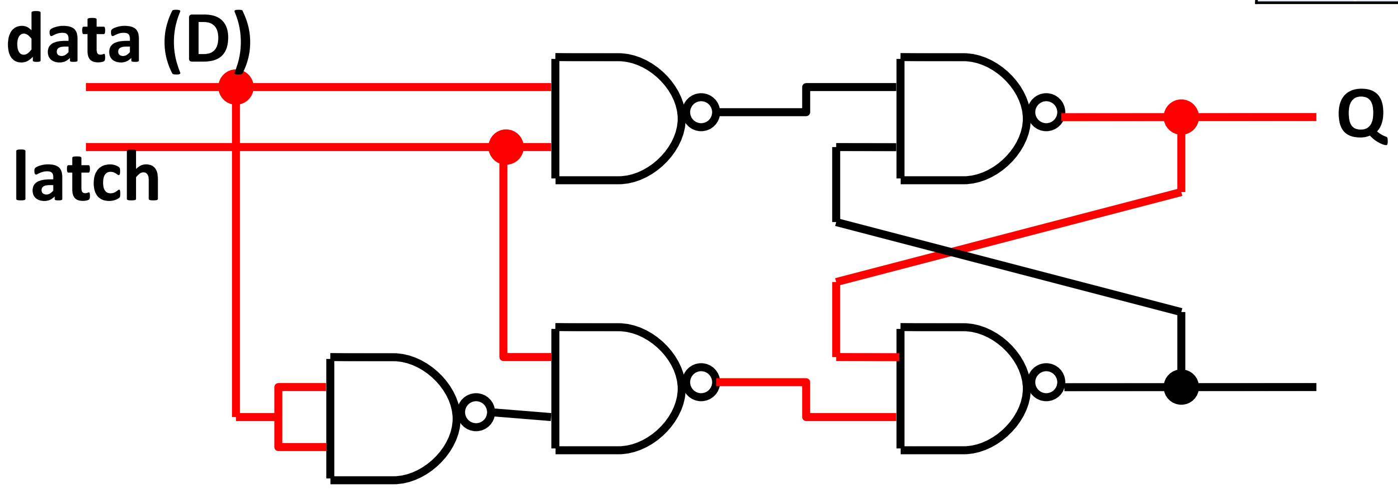
# Latching in a 1

- (Recall that it takes *two high inputs to force the output of a NAND gate low*)

| $Q_0$ | D | L | Q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

“latch” a 0

“latch” a 1



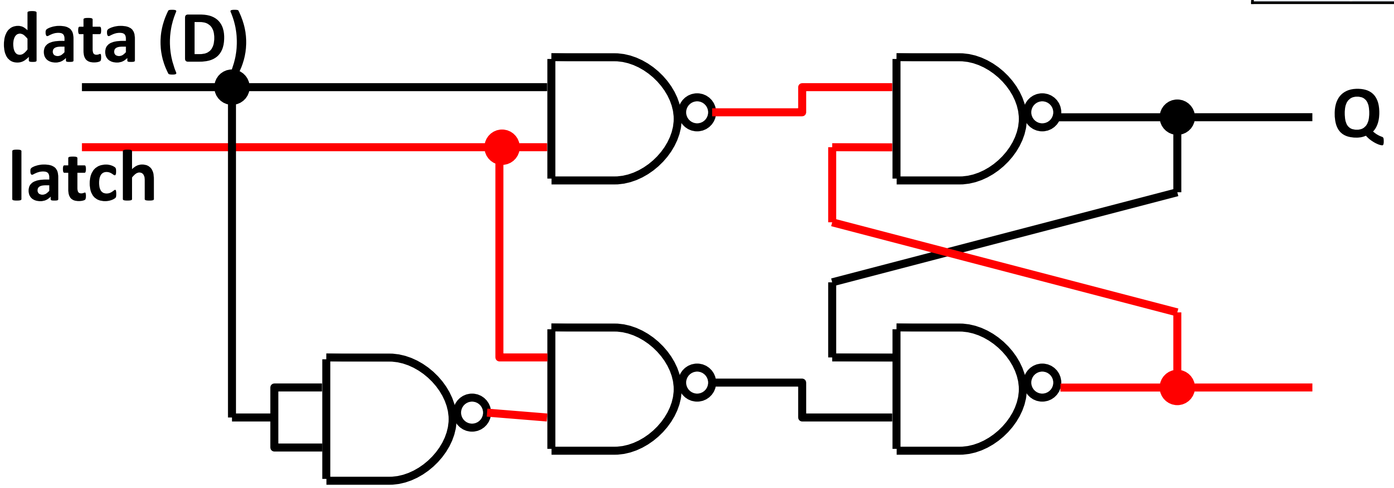
# Latching in a 0

- (Recall that it takes *two high inputs to force the output of a NAND gate low*)

| $Q_0$ | D | L | Q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

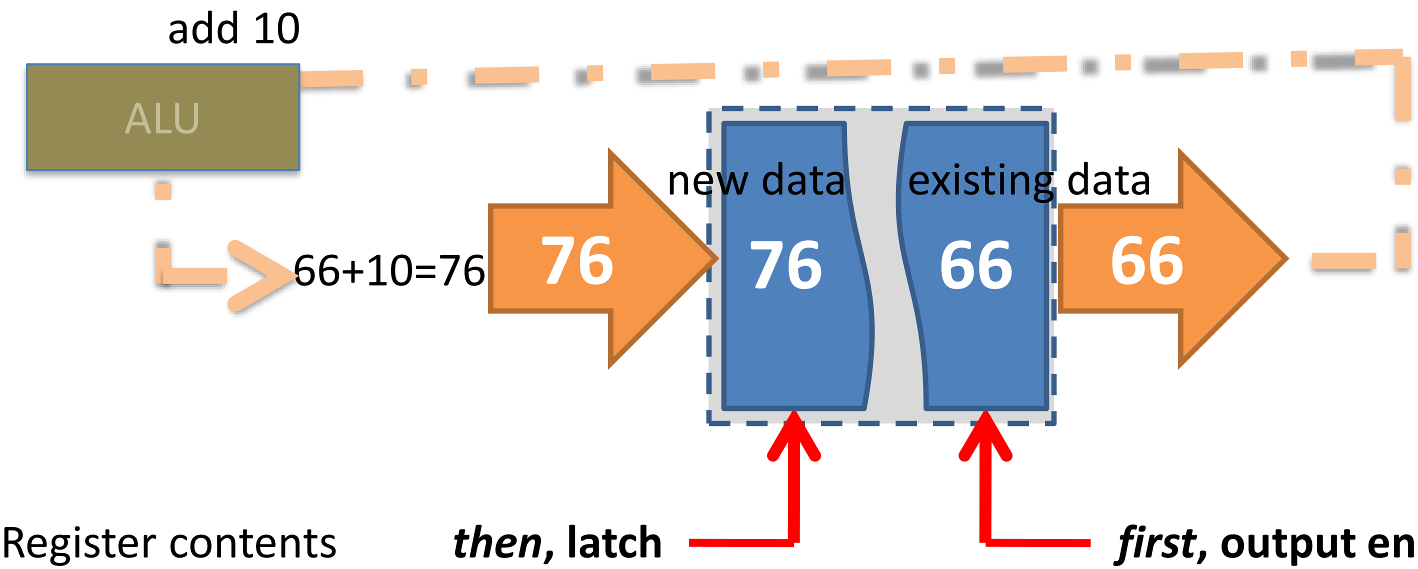
} “latch” a 0

} “latch” a 1



# A final tweak…

- So, with a Clocked D-type, we can **latch** a data value into a flip-flop – but we'd additionally like an analogous facility on the output side: **output enable (OE)**
- Will be useful in supporting register operations like "ADD A, A, 10" - where A is both a source and a destination
  - We need close control over when existing data "leaves" the flip-flop and new data "arrives"—if we don't get the timing right we might overwrite the stored bits before we've read them!
  - We'll understand the motivation here more clearly when we consider buses and the control unit
- Enter the *master-slave flip-flop*…



(Register contents replaced with data presented at input - release latch at end of cycle)

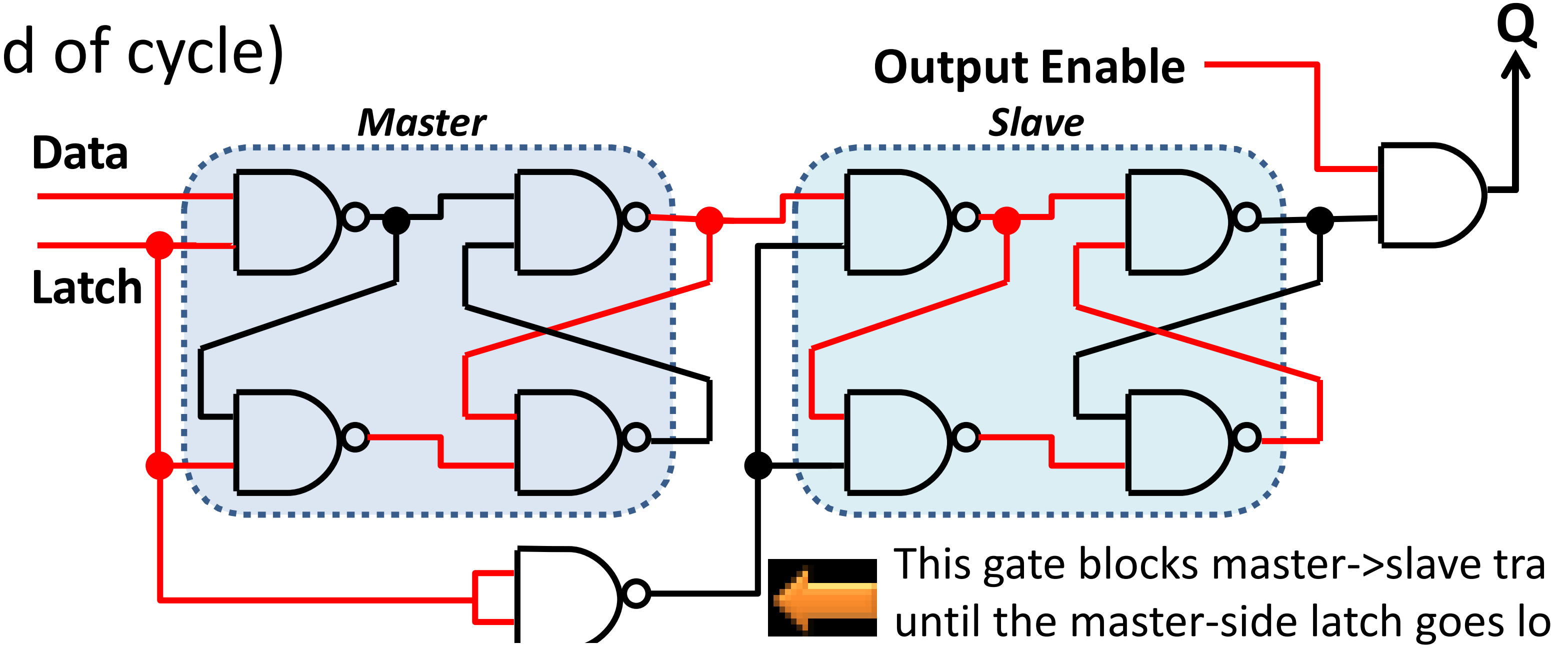
***then*, latch**

***first*, output enable**

(Register contents made 'visible' at output)

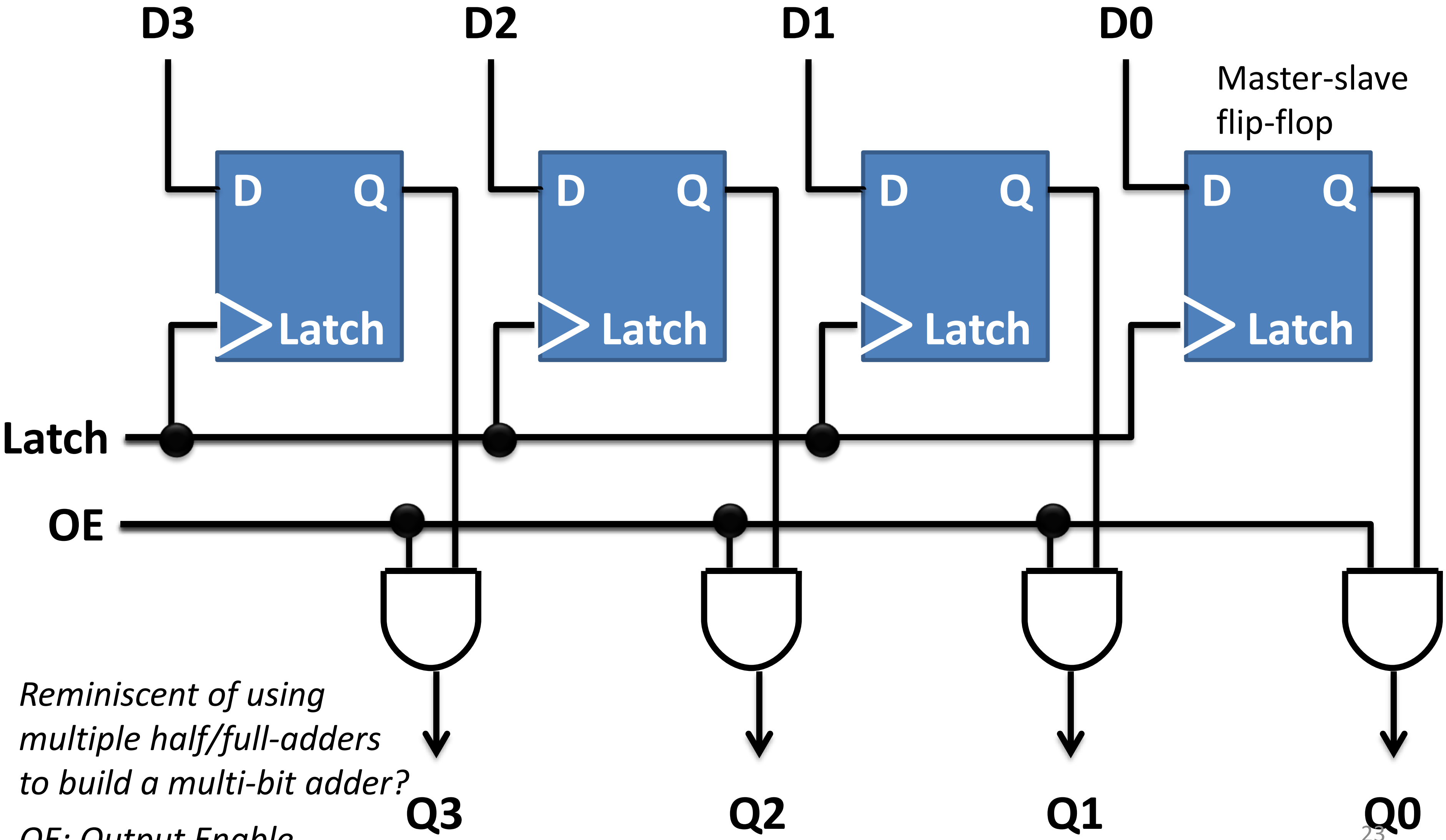
# Building a “master-slave flip-flop” from two D-type flip-flops

- Q (on the slave side) is only readable if *output enable* is **high**
- When *latch* (on the master side) pulses **high**, the *data* signal is stored in the master side, but the slave side is untouched
- When latch returns to **low**, the data in master moves to slave (at end of cycle)



This gate blocks master->slave transfer until the master-side latch goes low

# Implementing a register using multiple master-slave flip-flops



*Reminiscent of using multiple half/full-adders to build a multi-bit adder?*

*OE: Output Enable*

# Buses: connecting the blocks

- Buses are the “glue” of the computer architecture
  - They connect the elements of the von Neumann architecture: the ALU, the control unit, memory chips, and I/O devices
  - They are essentially bundles of **wires**, one for each bit

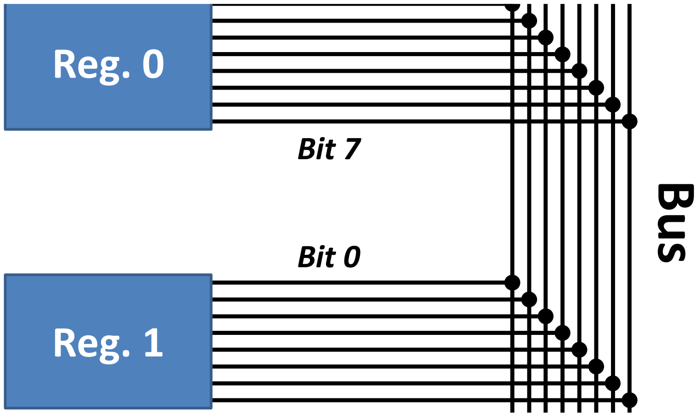
- There are two types of bus

  - An ***address bus*** runs between the control unit and main memory
    - Used to tell main memory to access a specific address (whether for reading or writing)
    - The “width” of the address bus corresponds to the amount of addressable memory
      - Examples: 16 bits → 65536 bytes ($2^{16}$); 32 bits → 4 GB ($2^{32}$)

  - ***Data buses*** carry data around the computer
    - An “n-bit” processor will have a data bus *n* bits wide (e.g. n=8 or 32 or 64)
      - This means we can read/ write *n* bits from/ to memory in one go
    - Some processors have distinct internal and external data buses
      - External bus may be narrower to reduce the number of external connections/ pins, and thus cost

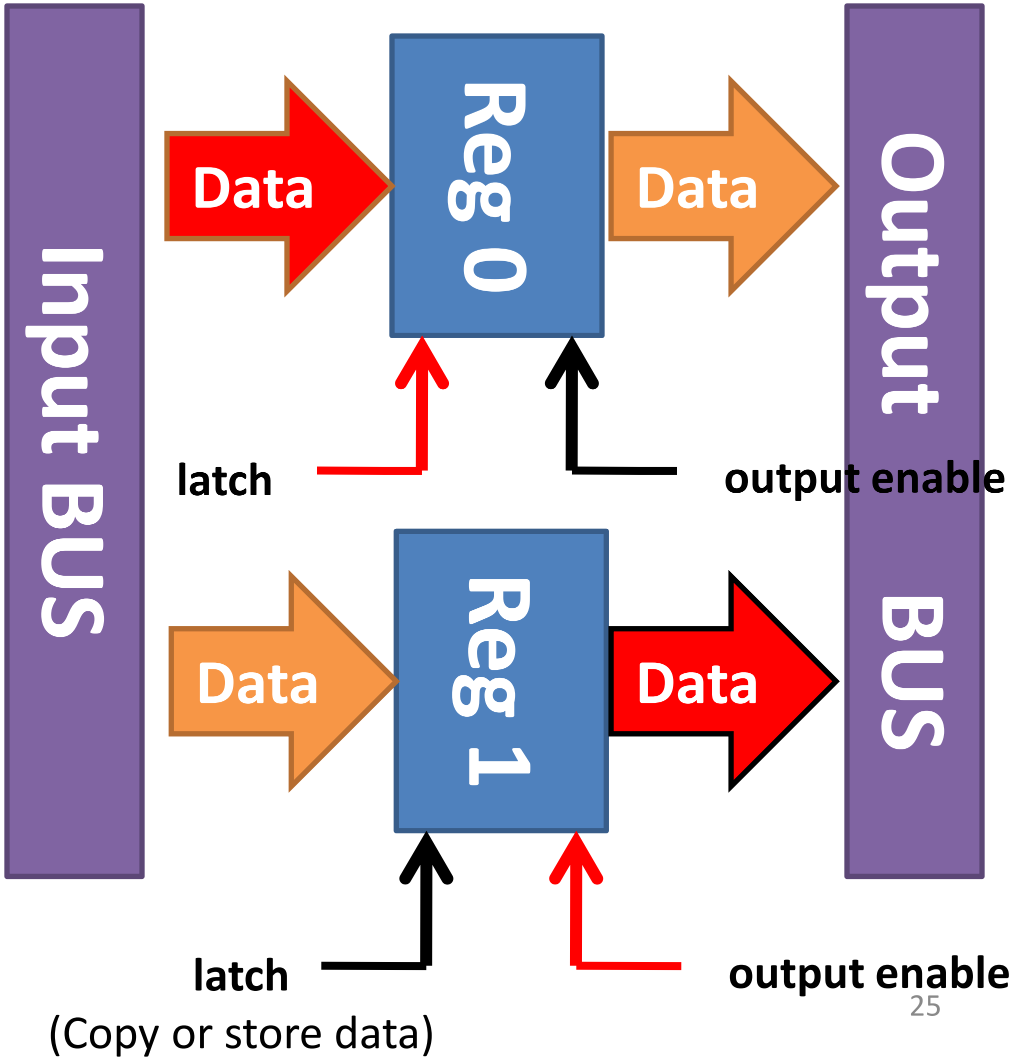
# Data buses: connecting to registers

- Bus wires are *shared*



... so we must ensure that there is only one active output at a given time

– A job for **output enable**...



# Summary

- We know the main characteristics of both dynamic and static memory
- We know, in outline, how the processor interacts with main memory (dynamic memory)
- We know the difference between combinatorial and sequential logic
- We know how logic components can be used to build *one bit* of static storage (static memory)
  - … and we know how these bits can then be combined into multi-bit registers
- We understand how the elements of the computer architecture can be glued together using buses