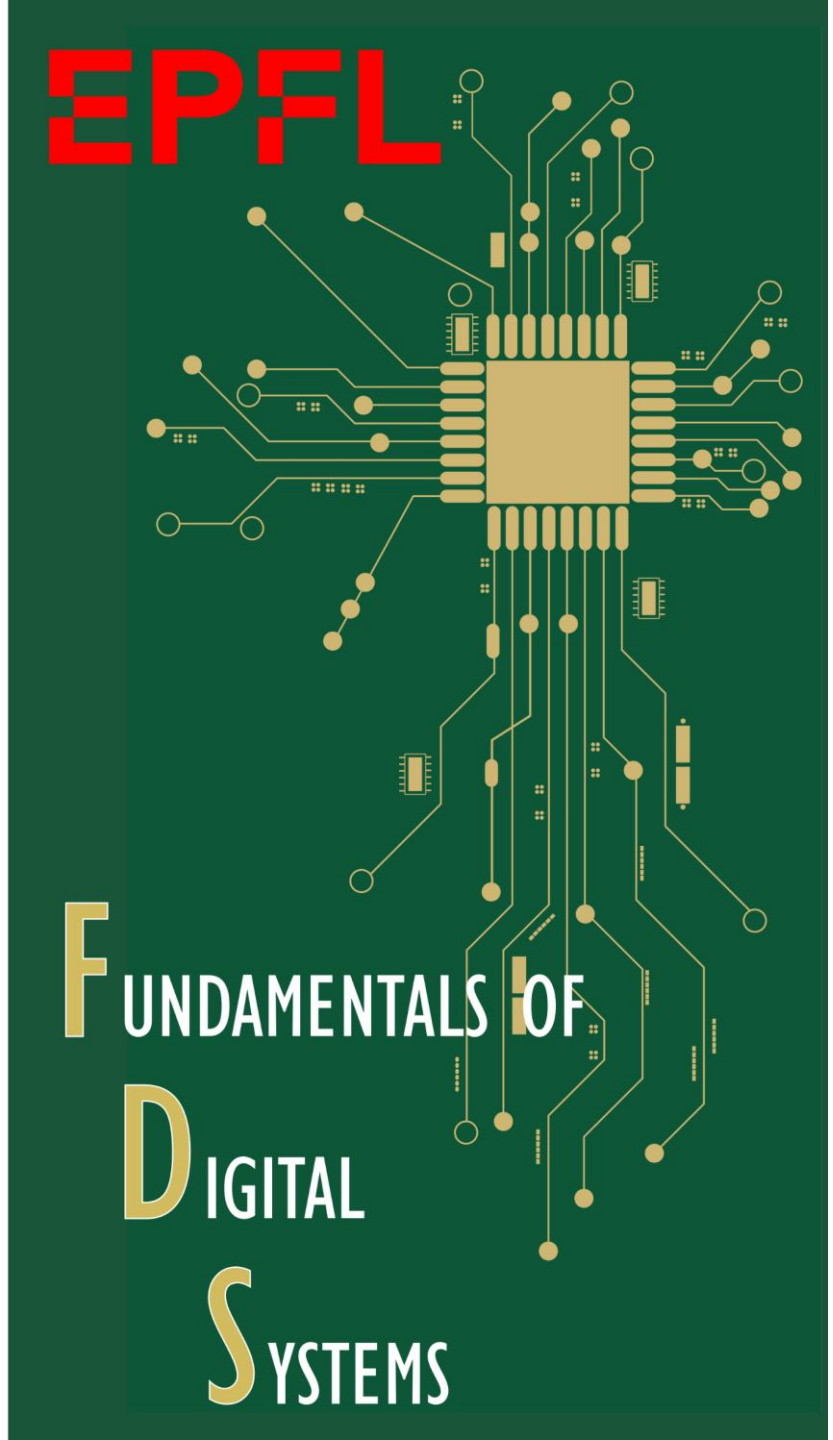


Digital Logic Circuits

Memories

CS-173 Fundamentals of Digital Systems

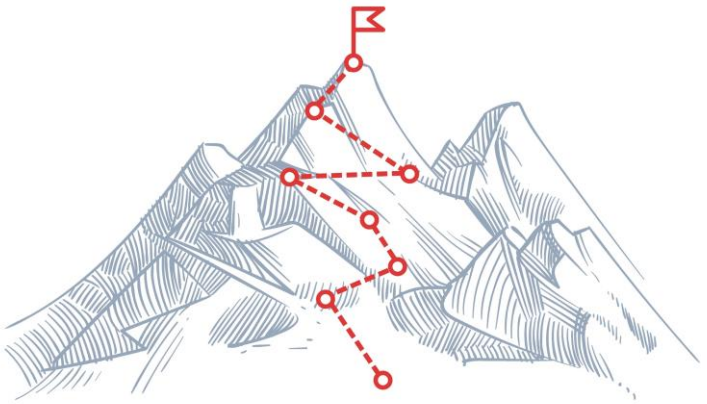
Mirjana Stojilović

Spring 2025

Previously on FDS

Timing Analysis of Synchronous Circuits

Previously



- Discovered FF timing constraints and parameters
  - Setup time and hold time
  - Clock-to-Q
- Defined expressions and algorithms for verifying if timing constraints are met
- Learned about metastability
- Clock skew: expanded our understanding of timing challenges in synchronous circuits in presence of clock delays

Quick Outline

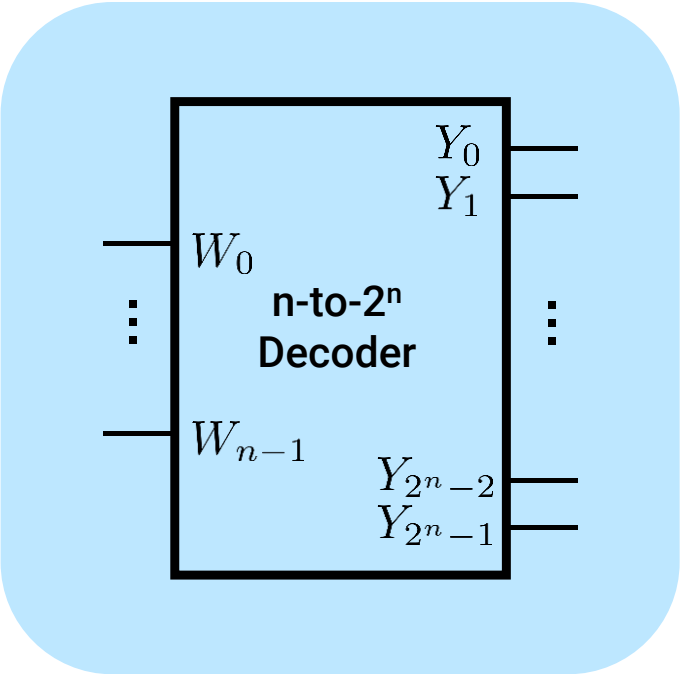
- n-to-2^n Binary decoders
- Memory
  - Data word size, memory capacity
  - Abstract view
  - Access protocol: write, read
  - Memory as a two-dimensional array of DFFs
  - Verilog model
- Verilog
  - Parameterized modules
  - Conditional operator

n-to-2^n Binary Decoders

n-to-2^n Binary Decoder

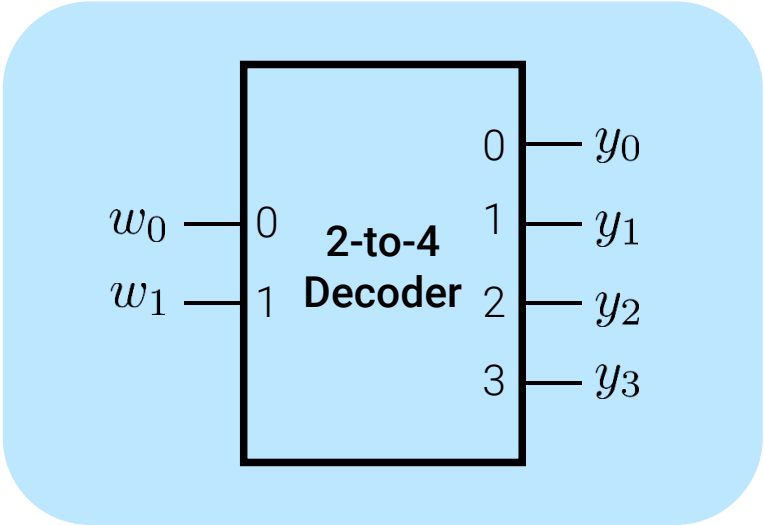
- **Decoder (in short)**, is a logic circuit that receives an **n-bit** binary vector and produces an output vector of **2^n bits** , in which all but one bit are logic zero
- **Decoding the input:** If **m** is the unsigned decimal equivalent of the **n**-bit binary number at the input of the decoder, the output bit at **index m** will be set to logic 1; all other output bits will be logic 0

- The outputs of a decoder are **one-hot encoded**, meaning that the single bit that is set to 1 is “hot”



2-to-4 Binary Decoder

- A (binary) decoder with $n = 2$ inputs and $2^n = 4$ outputs
- The input ($w_1 w_0$) corresponds to binary numbers
  - $(00)_2 = (0)_{10}$; $(01)_2 = (1)_{10}$; $(10)_2 = (2)_{10}$; $(11)_2 = (3)_{10}$
- Only one bit in the output vector (y_3, y_2, y_1, y_0) is set
  - $(w_1 w_0) = (00)_2 = (0)_{10} \rightarrow y_0 = 1$
  - $(w_1 w_0) = (01)_2 = (1)_{10} \rightarrow y_1 = 1$
  - $(w_1 w_0) = (10)_2 = (2)_{10} \rightarrow y_2 = 1$
  - $(w_1 w_0) = (11)_2 = (3)_{10} \rightarrow y_3 = 1$



2-to-4 Binary Decoder

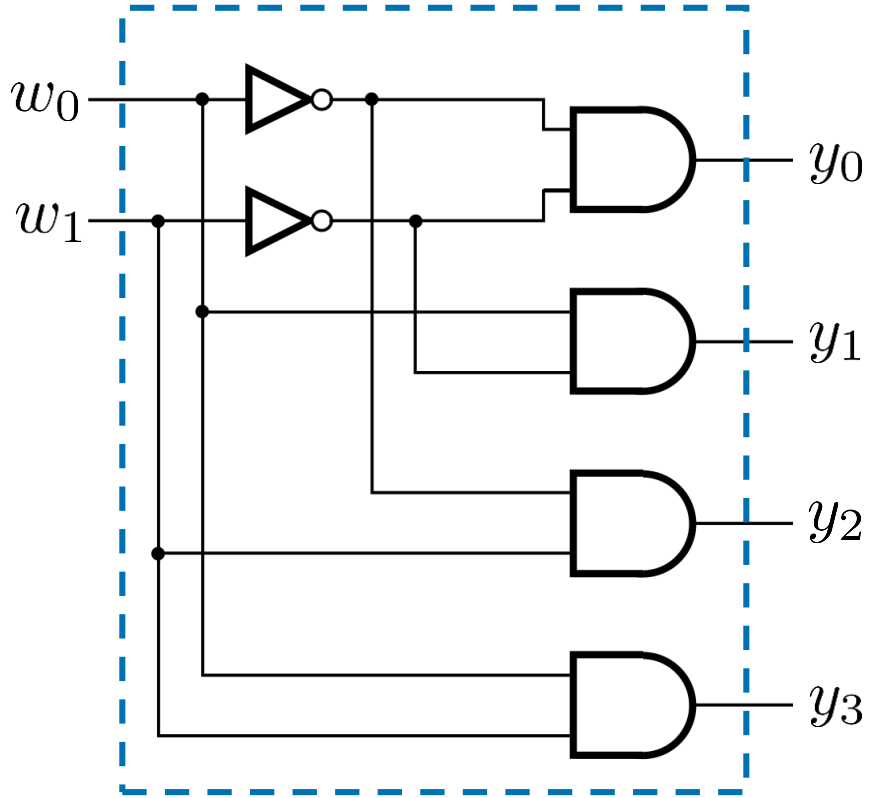
- A (binary) decoder with $n = 2$ inputs and $2^n = 4$ outputs
- Truth table

w_1	w_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- Logic expressions:

$$y_0 = \overline{w_1}\ \overline{w_0} \quad y_1 = \overline{w_1}\ w_0$$
$$y_2 = w_1\ \overline{w_0} \quad y_3 = w_1\ w_0$$

- Logic circuit

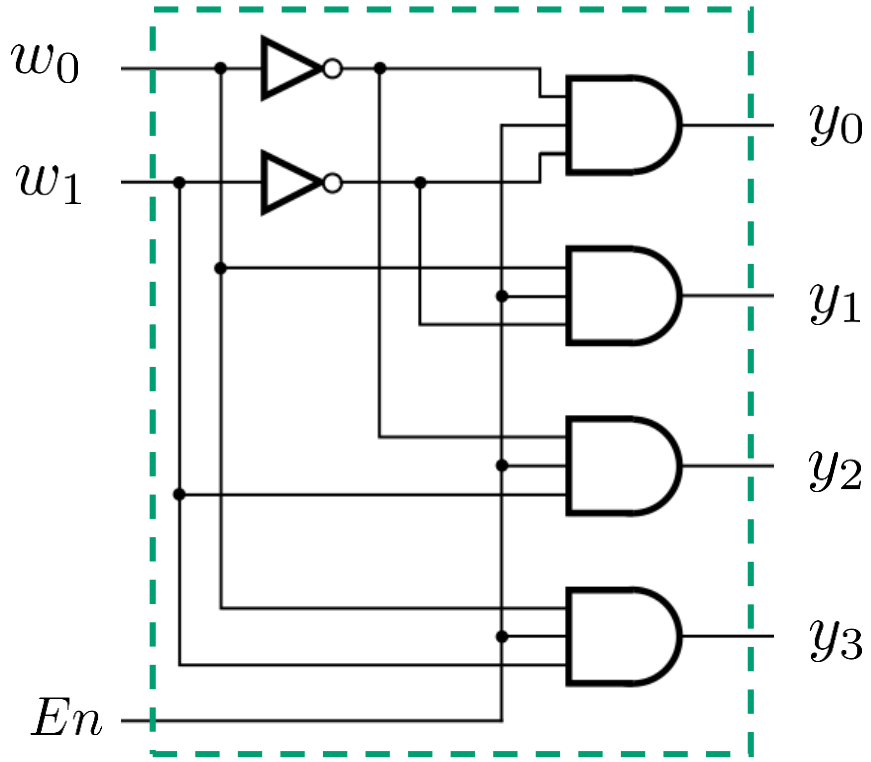


2-to-4 Binary Decoder with an Enable

- If the decoder is disabled, $En = 0$, then no output will be set
- Truth table:

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	X	X	0	0	0	0

- Logic circuit

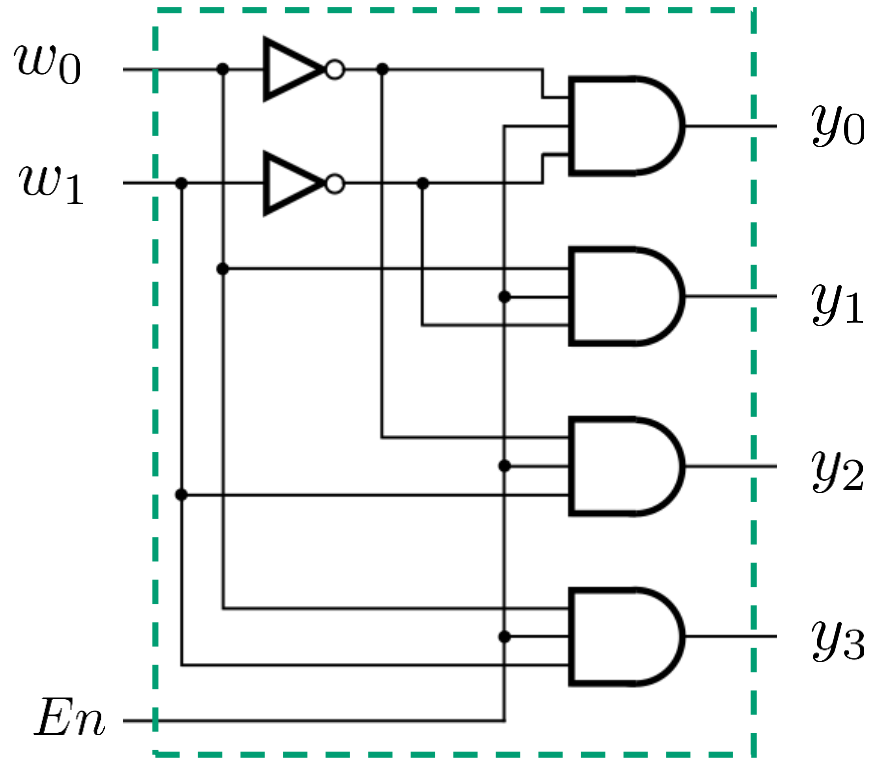


2-to-4 Binary Decoder with an Enable

In Verilog

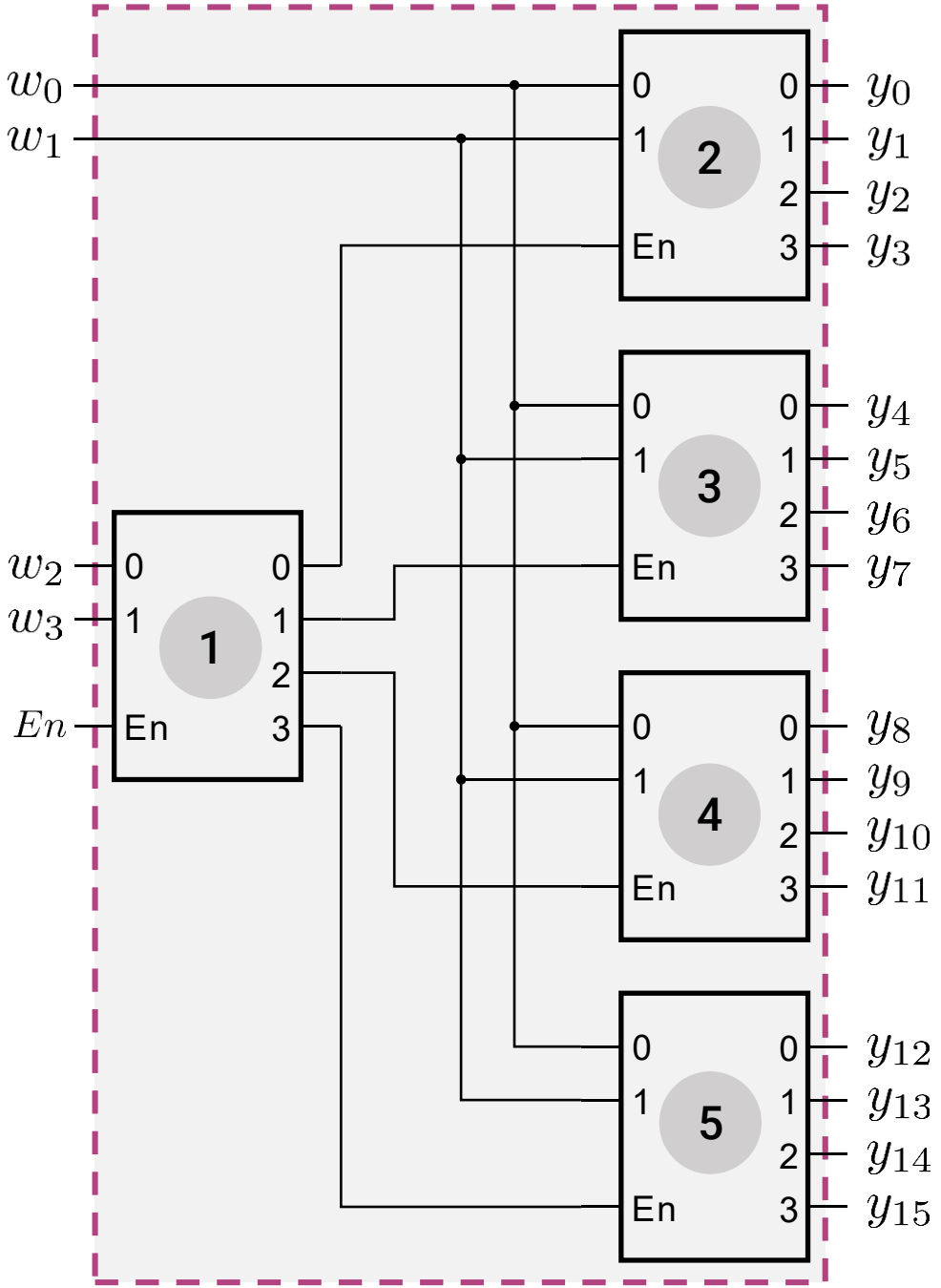
- Using Verilog's conditional operator `?:`
  The operator is explained on the following slide: link

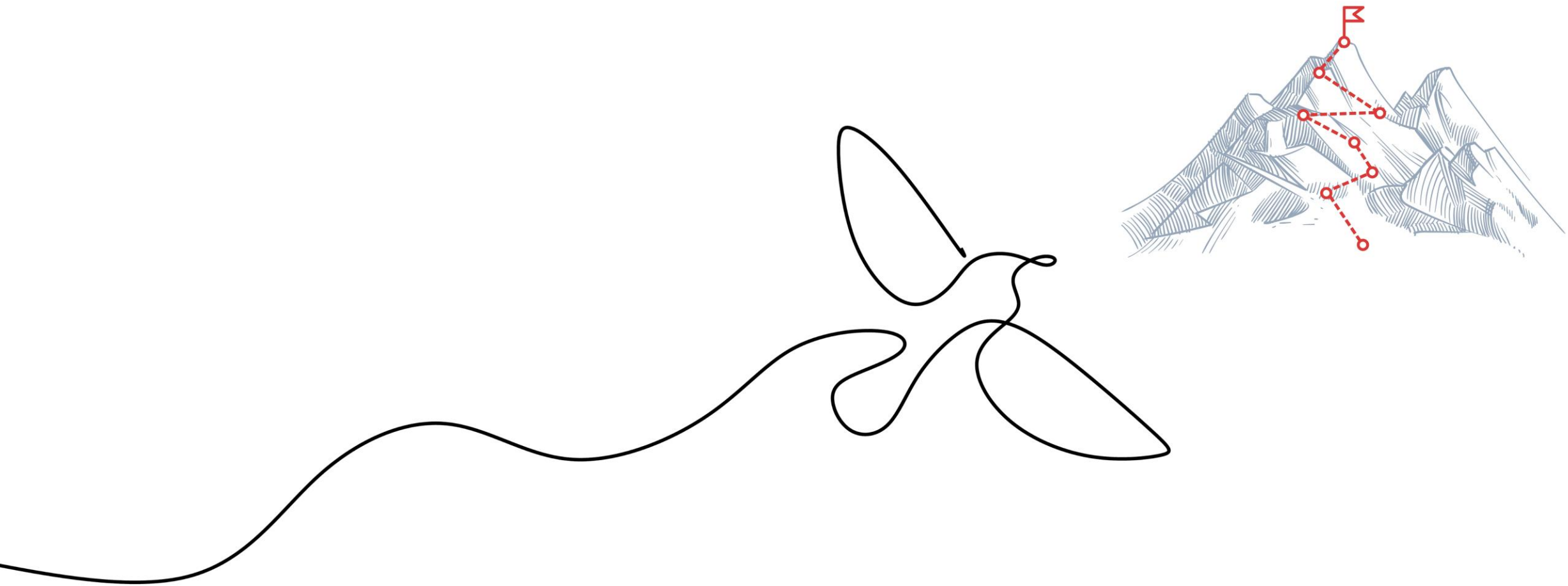
```
module two_to_four_dec(W, En, Y);

  input  [1:0] W;
  input  En;
  output [3:0] Y;

  assign Y[0] = En ? (W == 2'b00) : 0;
  assign Y[1] = En ? (W == 2'b01) : 0;
  assign Y[2] = En ? (W == 2'b10) : 0;
  assign Y[3] = En ? (W == 2'b11) : 0;

endmodule
```



4-to-16 Decoder

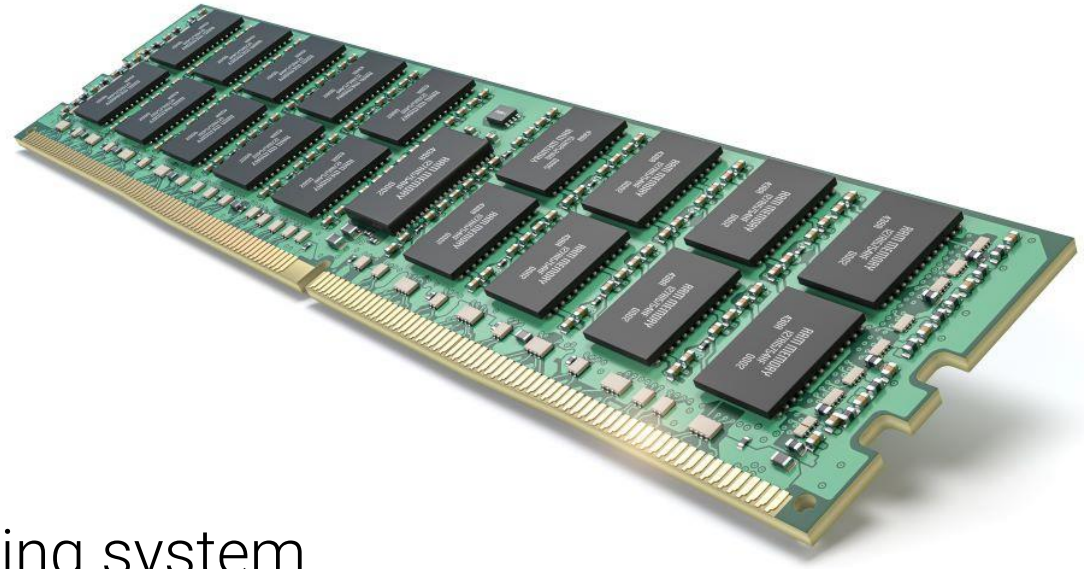
- Larger decoders can be constructed from smaller ones
- Figure: 4-to-16 decoder built using a decoder tree
  - One decoder in the “root”
  - Four in the “branches”
- Example:
  - ($w_3w_2w_1w_0$, En) = (1011, 1): $OUT_{DEC1}[2] = 1$, $En_{DEC4} = 1$, and $OUT_{DEC4}[3] = y_{11} = 1$





Memory

Memory

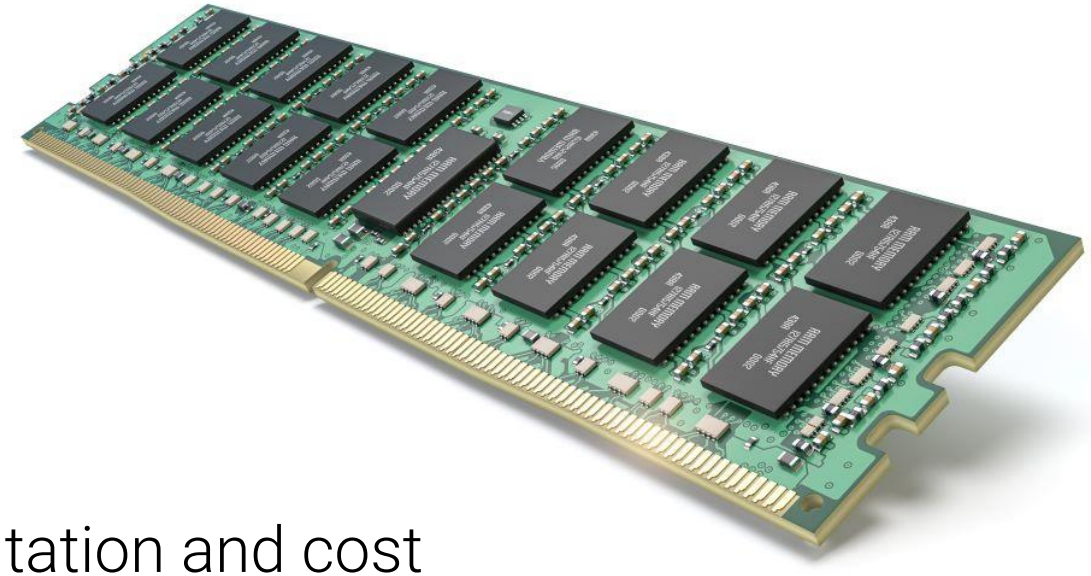


© Maksym Yemelyanov / Adobe Stock

- Is an essential component of any computing system
- A memory is a circuit capable of storing information temporarily or permanently
- Many types of memories exist
  - SRAM, DRAM, EPROM, Flash, magnetic, optical, …

Note: The implementation details of the above mentioned memories are out of scope of this course

Memory

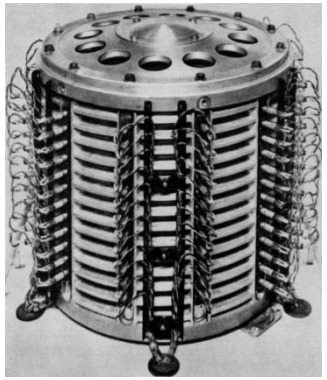


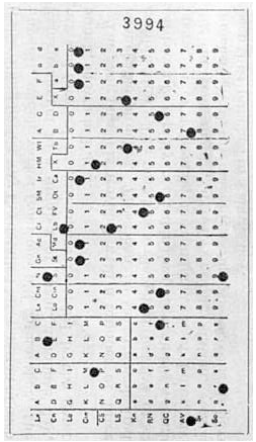
© Maksym Yemelyanov / Adobe Stock

- Memories differ in the physical implementation and cost
- As a consequence, their capabilities vary
  - **Capacity**: how many bytes can be stored
  - **Density**: how many bits per unit area (silicon)
  - **Speed**: how much time it takes to read from it or write to it
  - **Writable** or **read-only**
  - **Volatile** or **not**: loses contents once the power supply is removed or not

Evolution of Memory

Non-Volatile
Slow





(1930s) Punch Cards,
Magnetic Tapes / Drums
Big, slow, store KiBs

[Mechanical]
(Magnetic) Hard Disk, Floppy; (Optical) CD/DVD
Slow, store MiBs,GiBs,TiBs

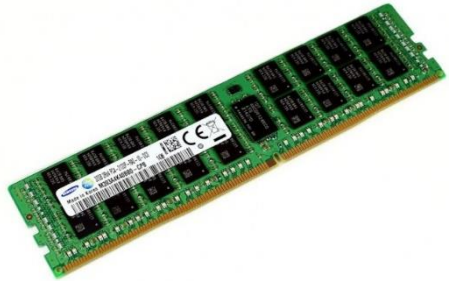
[Electronic] SSD
Fast, store GiBs

Non-Volatile, Slow → Data Storage

Volatile, Fast → Computational memory

(1970s) Semiconductors
Small, slow, store KiBs

SRAM
Fast, store MiBs

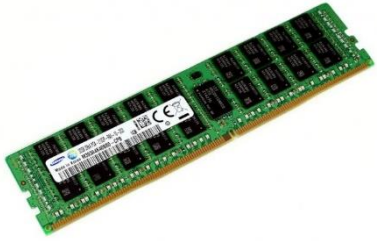


DRAM
Slower, store GiBs

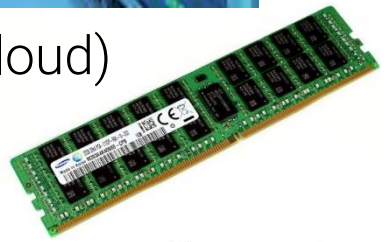
What are We Using?

Smartphone

Laptop / Workstation

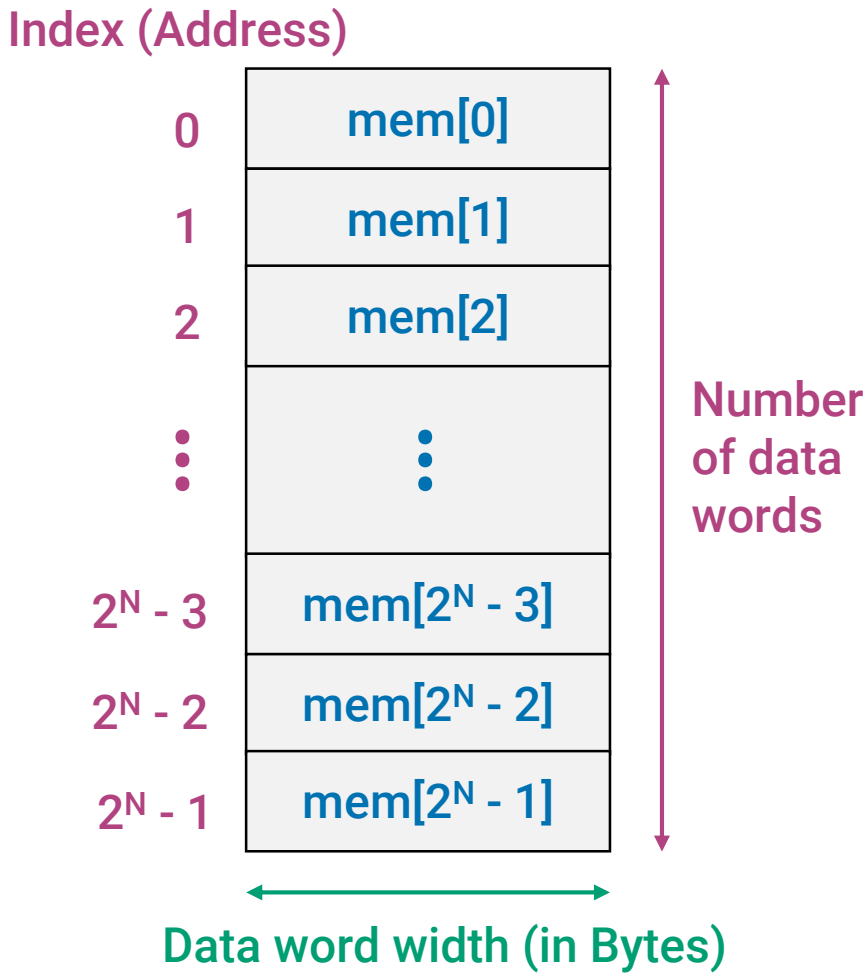


Datacenter (Cloud)



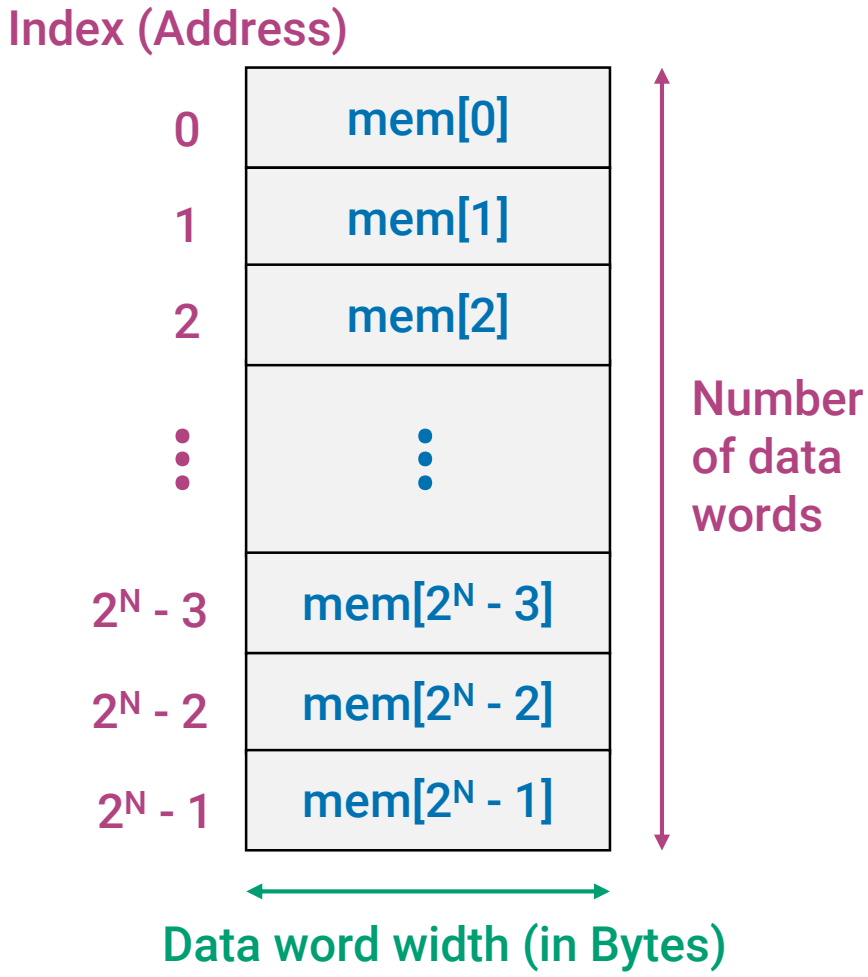
Abstract View of Memory

- A two-dimensional array of 1-bit memory elements (e.g., DFFs, latches, capacitors)
- One memory element **mem[i]**, where **i** is the index of the **row** in the two-dimensional array of bits, includes **all** bits in that **row** of the memory array and is typically referred to as one data **word**
- In memory terminology, instead of saying the index, we say **the address** of the data word



Word Sizes

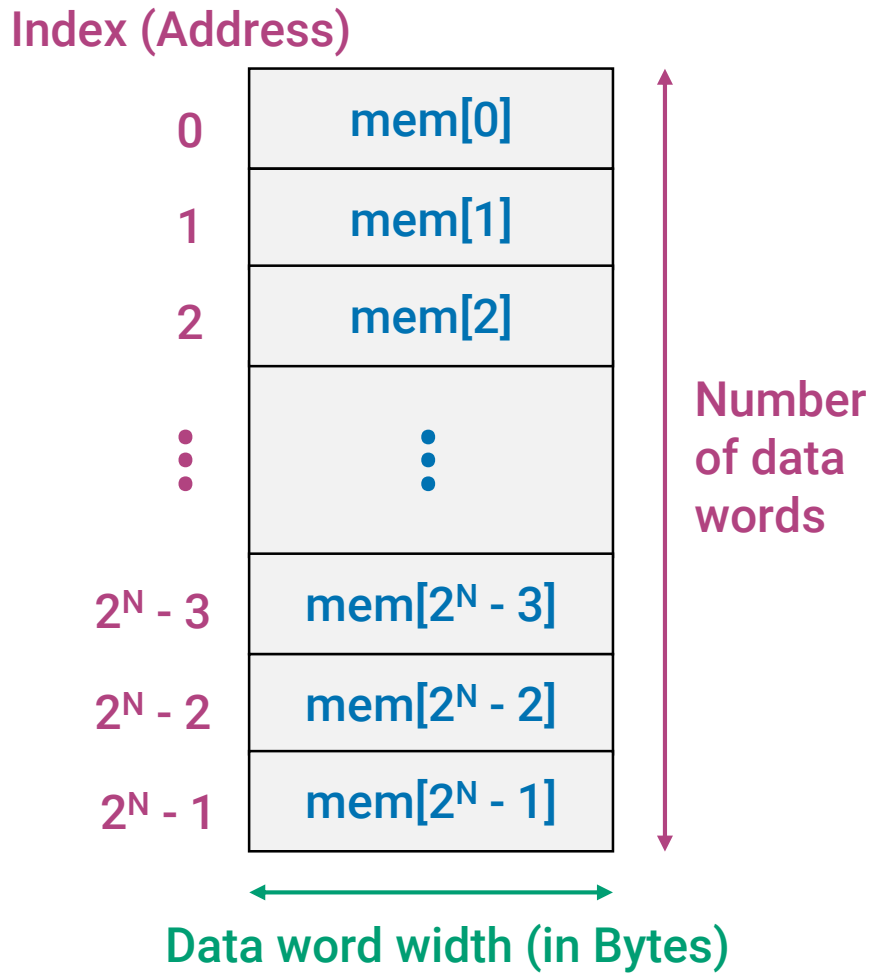
- Data word sizes and common terminology:
  - 8 bits = one byte (**Byte, B**)
  - 16 bits (also called half-word)
  - 32 bits, 64 bits
- Memory **capacity** is the total number of data bytes it can store
  - **(Number of words) × (word width in bytes)**
- Number of words = 2^N, where N is the number of bits of the address

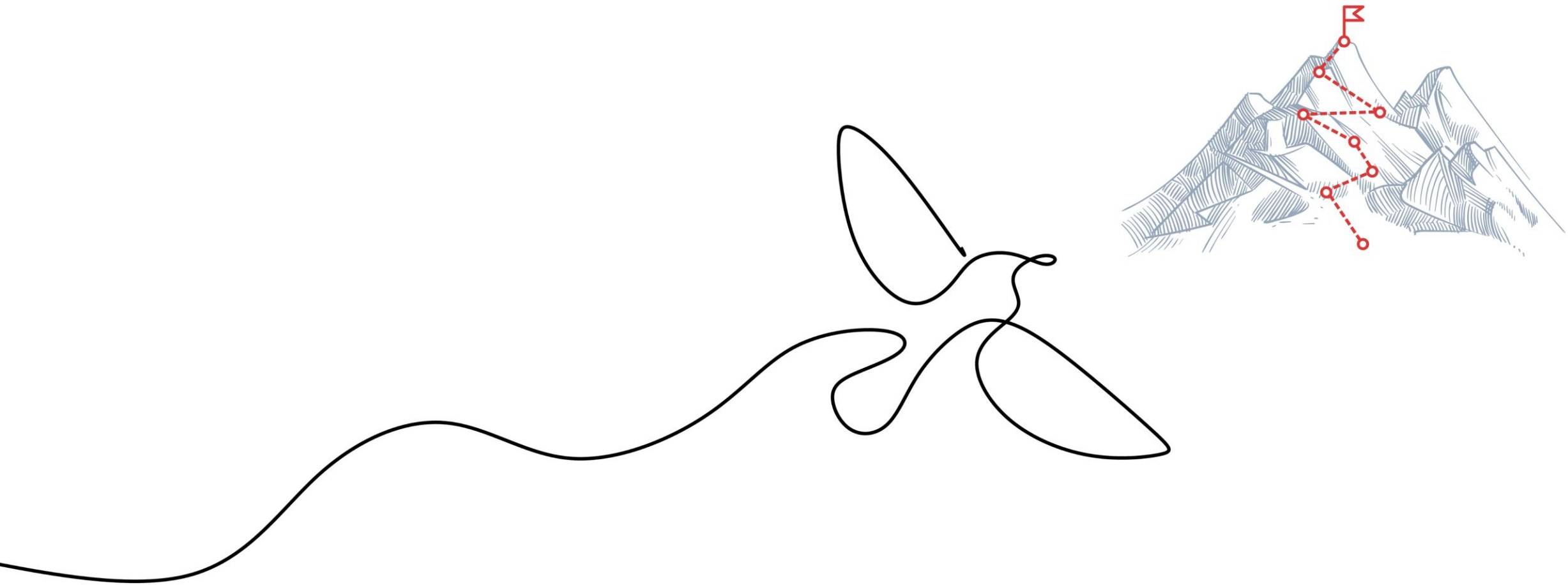


Memory Capacity

- *Recall:* memory **capacity** is the total number of data bytes it can store

- Memory units (Wiki link)
  - Kilobyte (**KiB**) = 2^{10} B
  - Megabyte (**MiB**) = 2^{20} B
  - Gigabyte (**GiB**) = 2^{30} B
  - Terabyte (**TiB**) = 2^{40} B



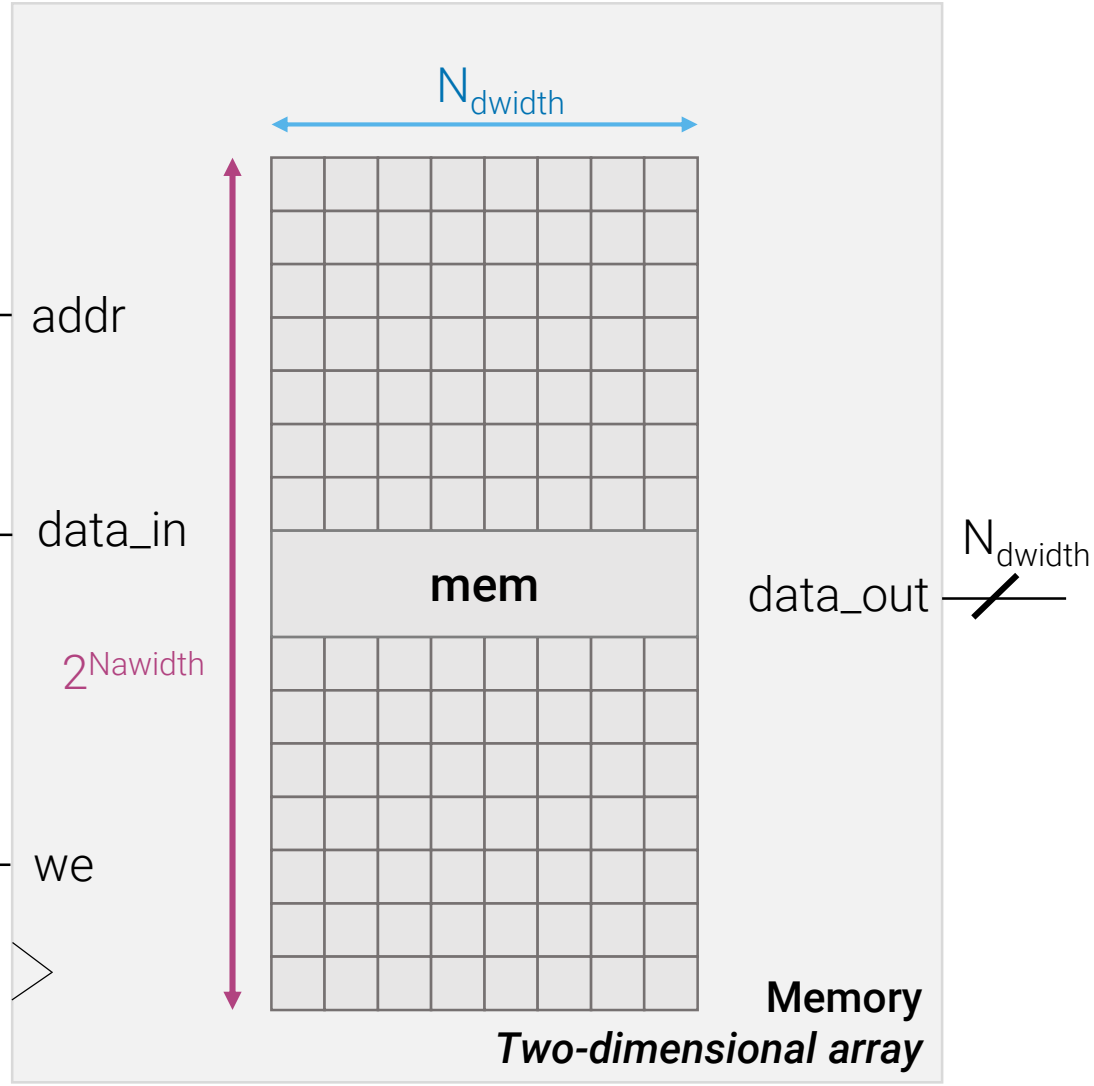


Memory
Access Protocol

- Many variants exist, differing in control signals, their polarity, number of inputs and outputs, width of inputs and outputs, etc.

In this lecture, we consider the following simple memory access protocol:

- **Synchronous write**: on the rising clock edge, if write enable (we) is active, memory **write** takes place: **mem[addr] = data_in**
- **Asynchronous read:** at all times, memory **read** takes place: **data_out = mem[addr]**



Memory

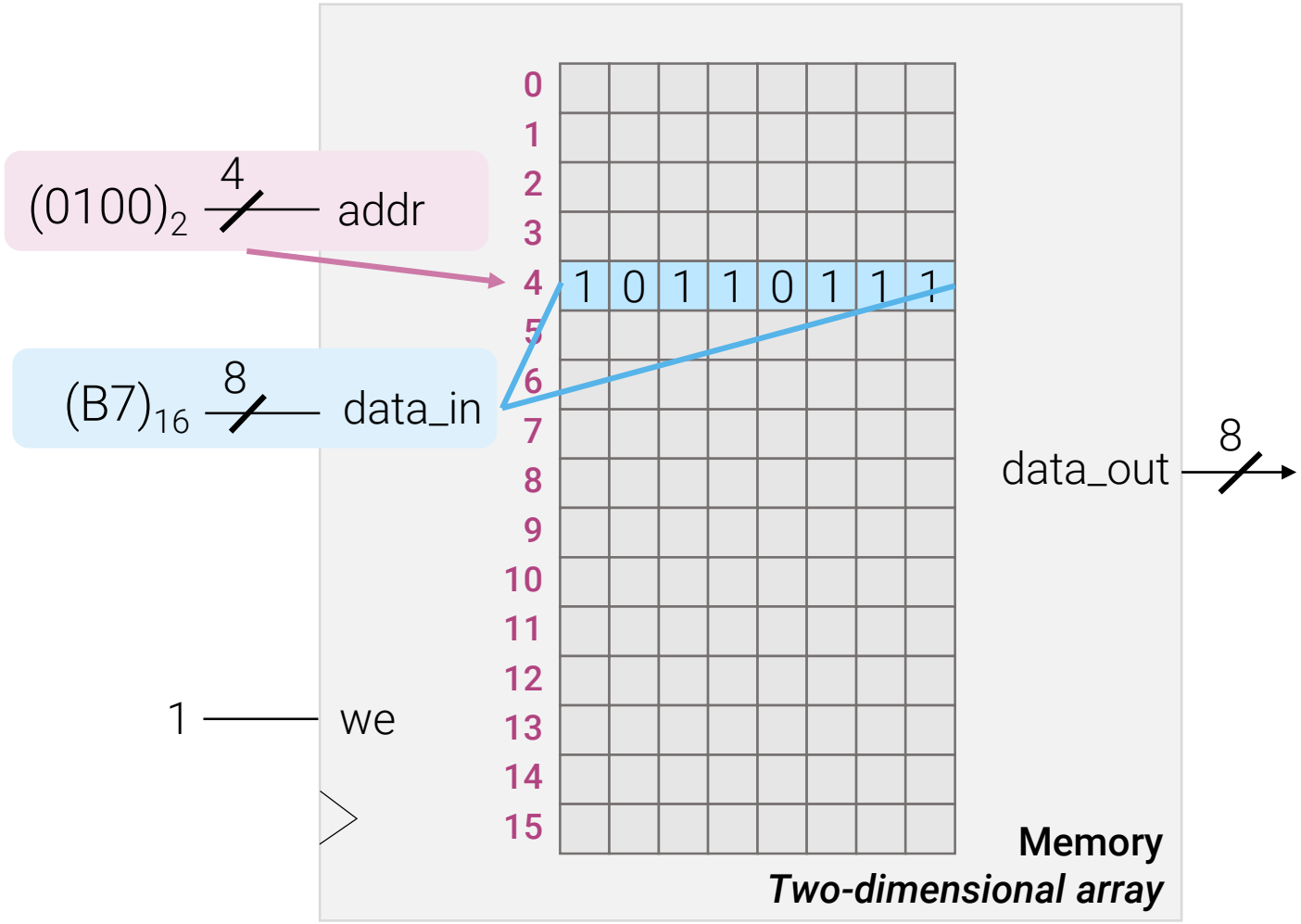
Write

Example:

- $N_{awidth} = 4$
- $N_{dwidth} = 8$
- addr = $(0100)_2$ = $(4)_{10}$
- data_in = $(B7)_{16}$
- we = 1

Memory read takes place and
old mem[4] appears at data_out port

Memory write takes place and
data_in overwrites mem[4];
new value of mem[4] = $(B7)_{16}$



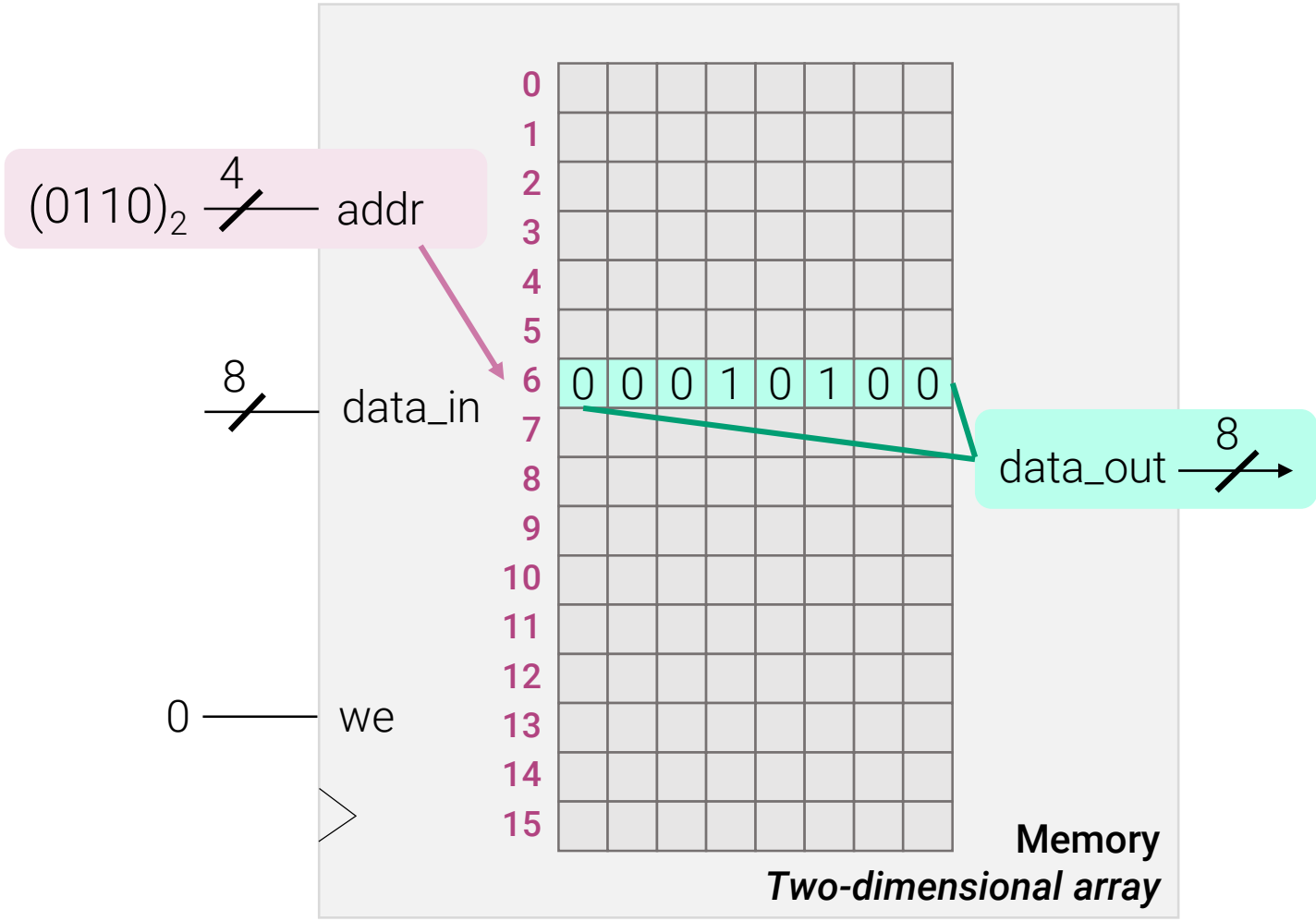
Memory
Read

Example:

- $N_{awidth} = 4$
- $N_{dwidth} = 8$
- addr = $(0110)_2$ = $(6)_{10}$
- we = 0

Memory read takes place and mem[6] appears at data_out port:
data_out = mem[6] = $(14)_{16}$

Memory write is not taking place



Constructing a 4×4 DFF Array

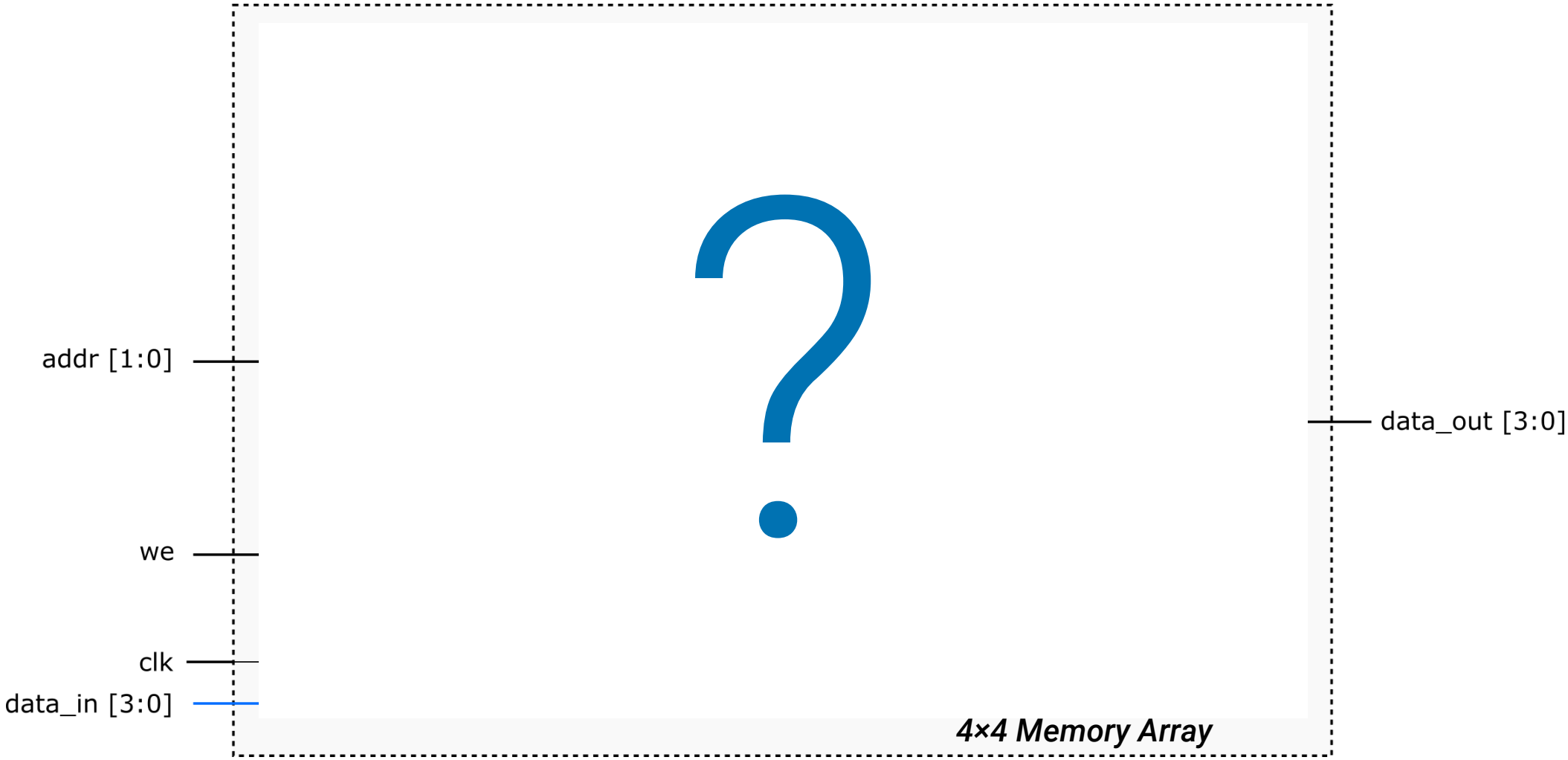
- 4×4 stands for 4 memory rows, each row having 4 data bits

- Q: N_{dwidth} = ?
- A: 4

- Q: N_{awidth} = ?
- A: log2(4) = 2

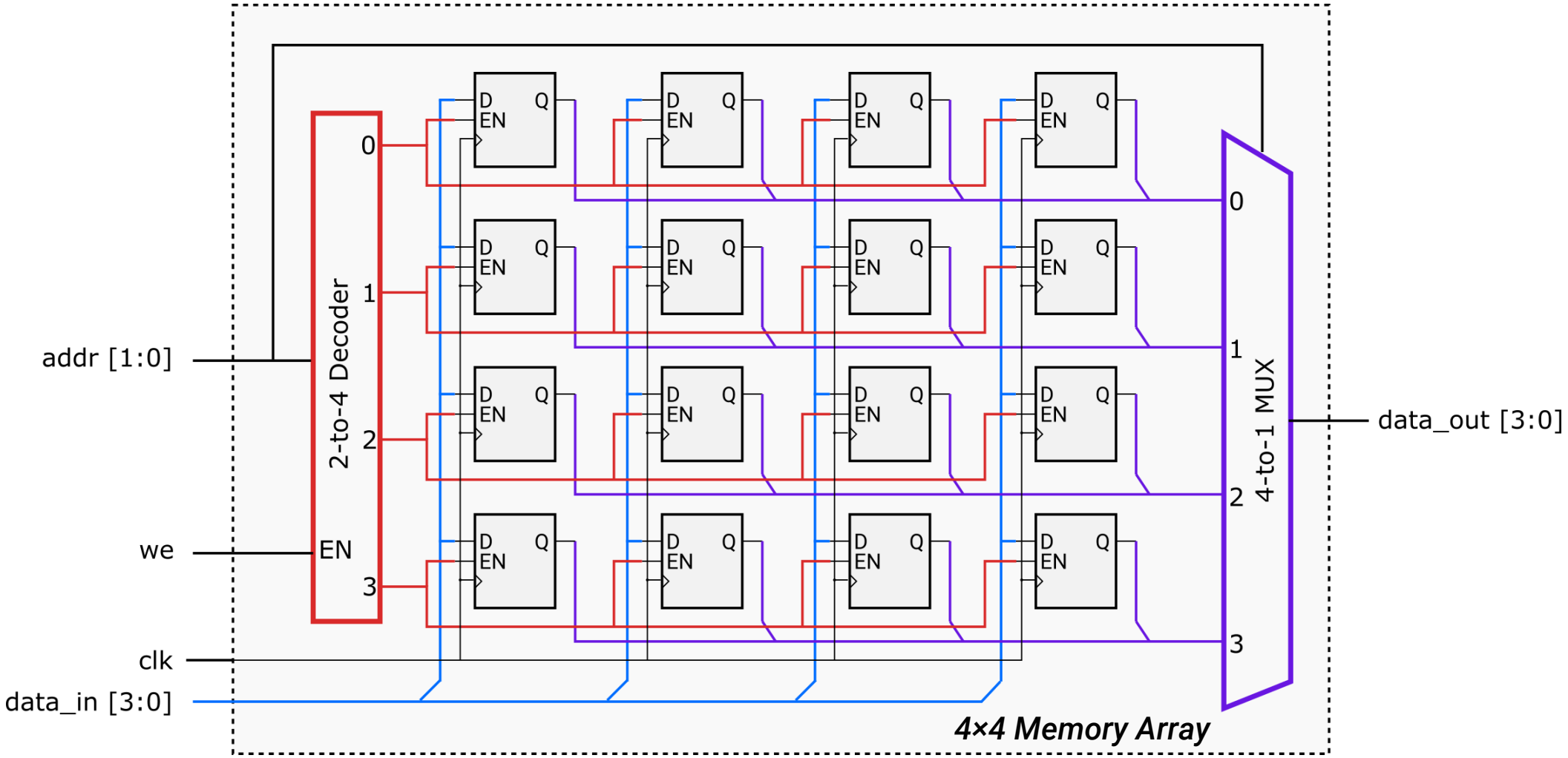
Memory as an Array of DFFs

Internals of a 4×4 Memory Array



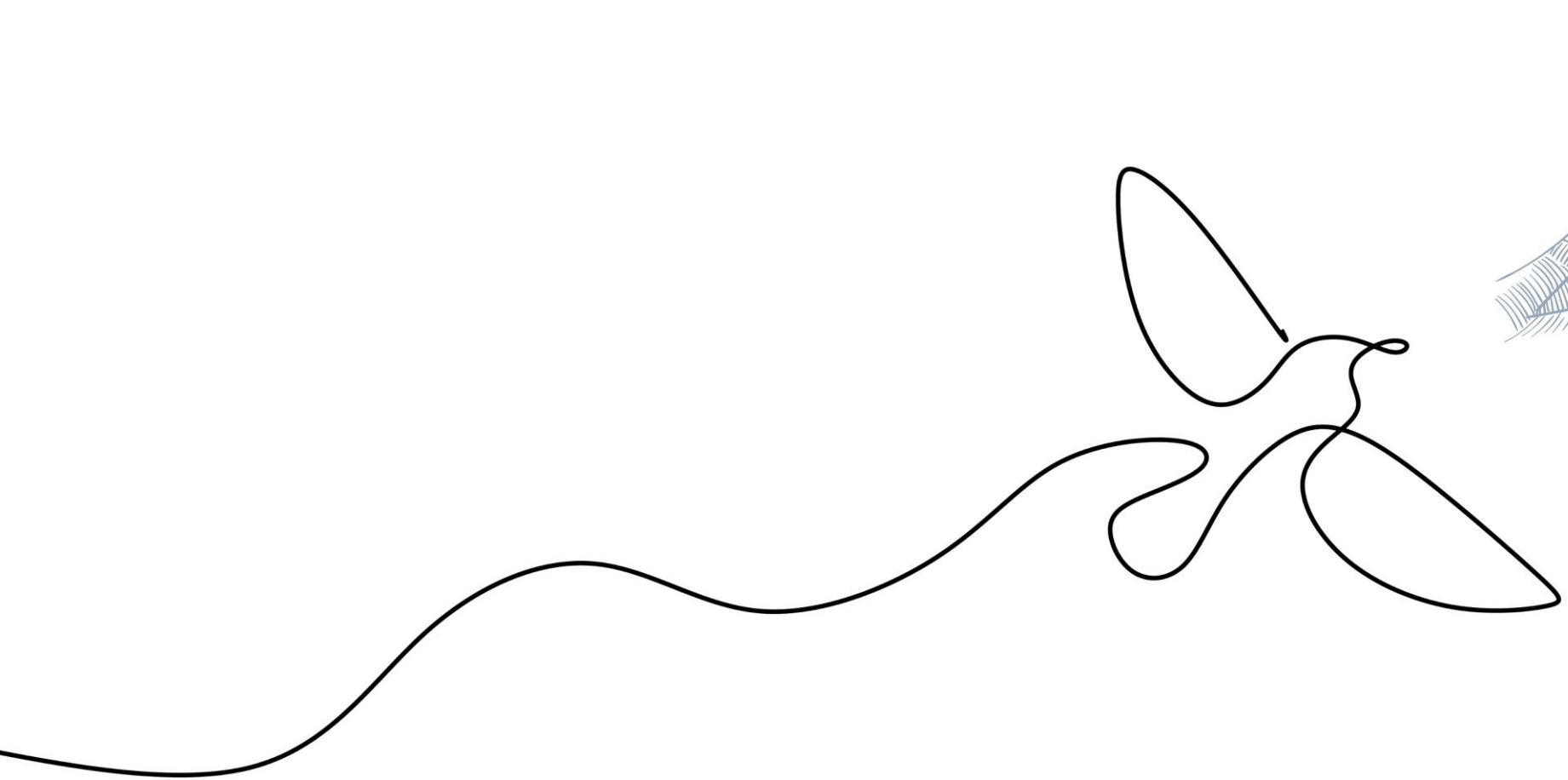
Memory as an Array of DFFs

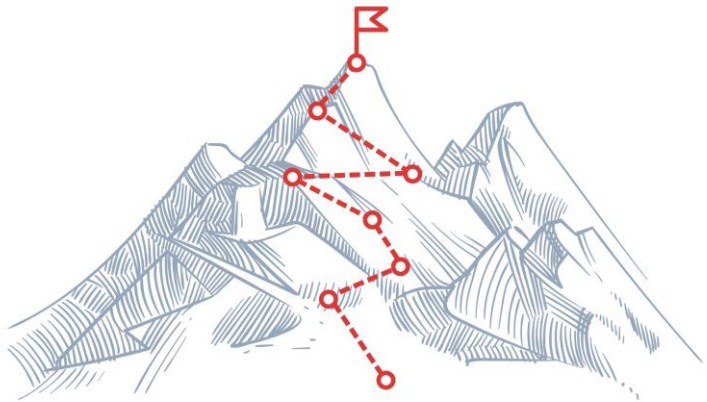
Internals of a 4×4 Memory Array



Memory Terminology, Contd.

- The outputs of the address decoder, which enable one entire row of the memory array, are called **word lines**
- The wires that carry data (input, output, or sometimes a shared in/out bus) are called **bit lines**





Arrays

In Verilog

Two-Dimensional Arrays
In Verilog

- *Recall:* A memory is a two-dimensional array of bits
- In Verilog, we can declare two-dimensional arrays
- An array **`mem`** of **`Ndw`** data words, where each word has **`Ndb`** bits, is declared as

```
reg [Ndb-1:0] mem [Ndw-1:0];
```

Two-Dimensional Arrays

In Verilog

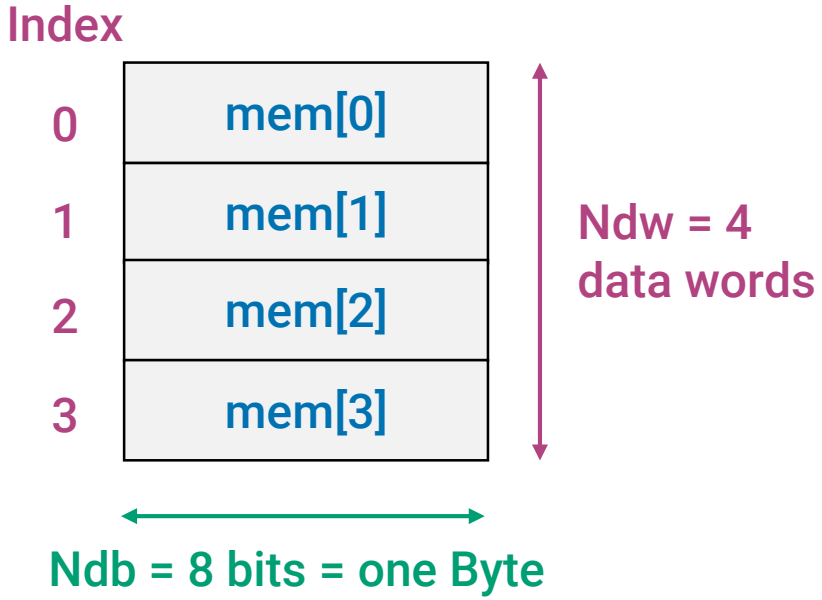
- *Recall:*

```
reg [Ndb-1:0] mem [Ndw-1:0];
```

- For example:

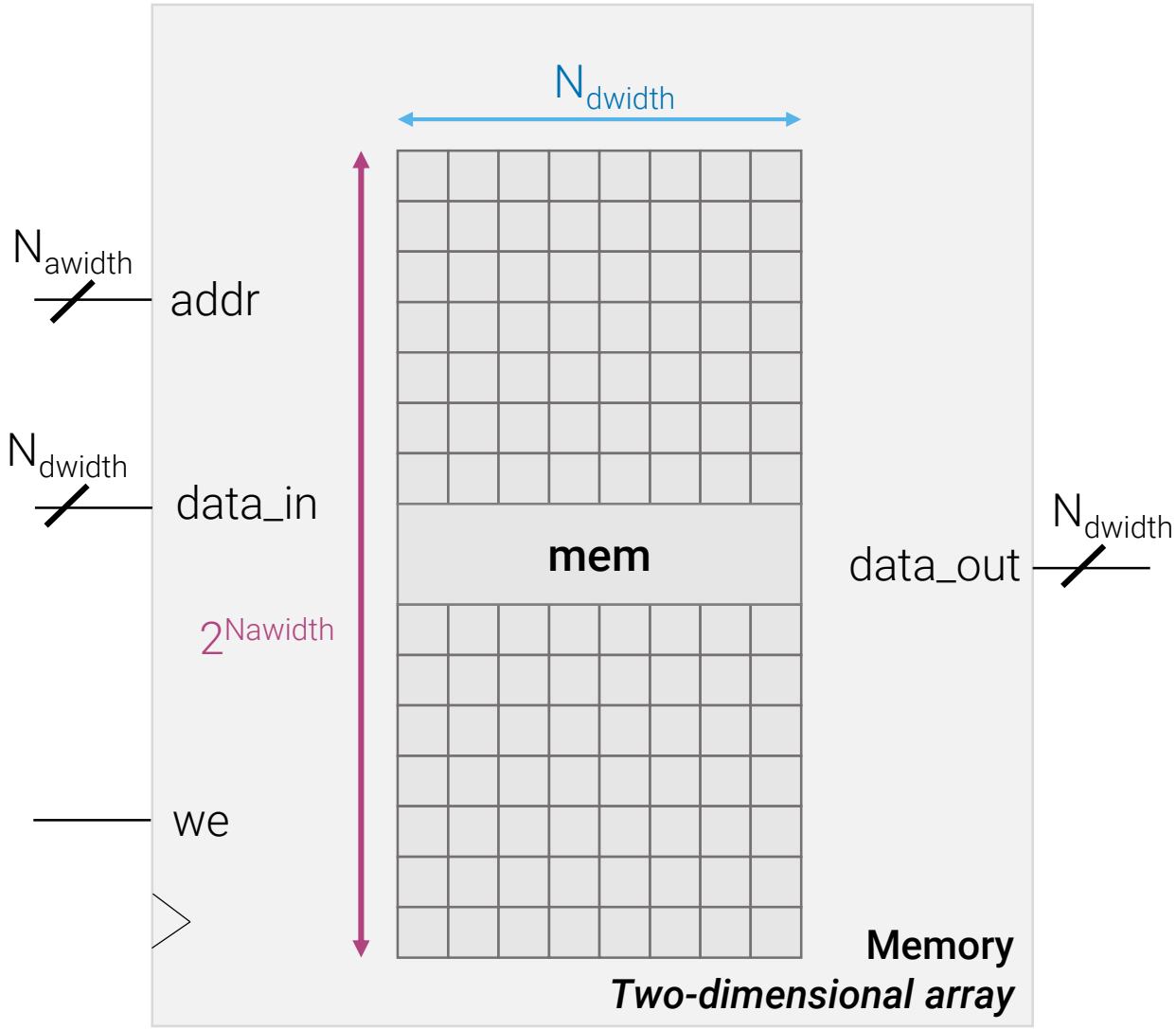
```
reg [7:0] mem [3:0];
```

  - `mem` is an array of four bytes `mem[3]`, `mem[2]`, `mem[1]`, and `mem[0]`
  - Two-level indexing can be used, e.g., `mem[3][7]`



Memories
In Verilog

- *Recall:*



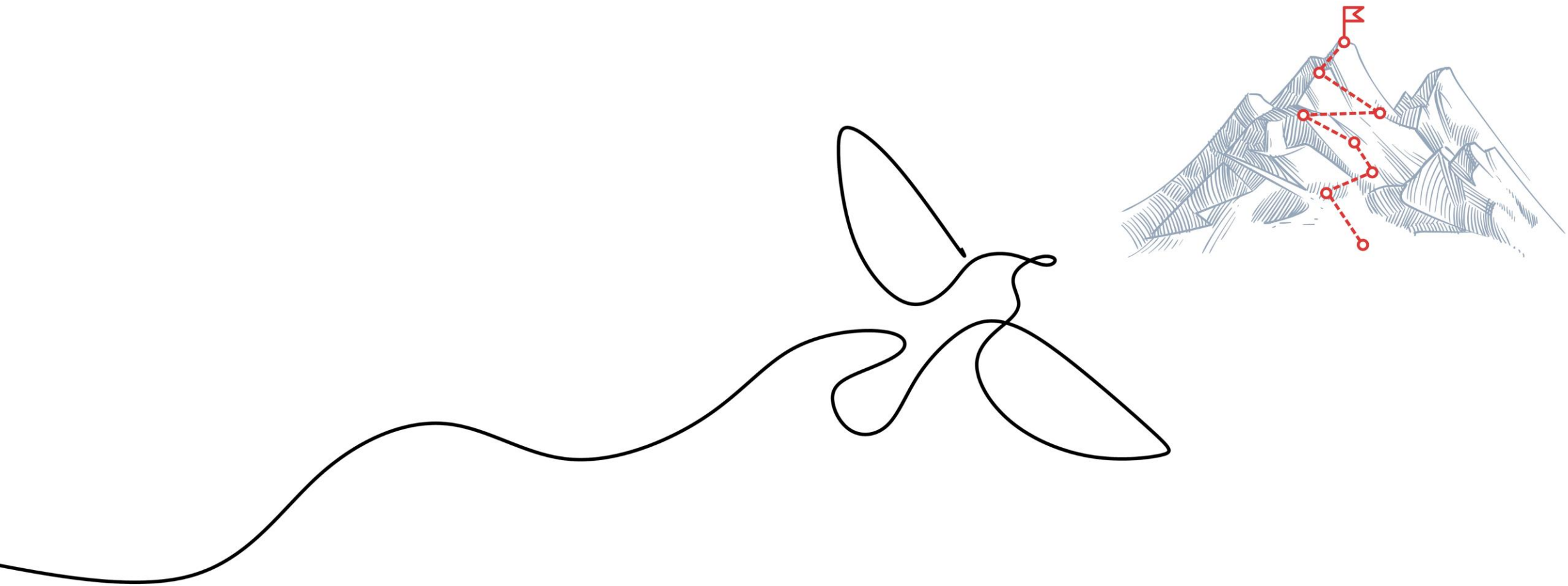
Memories

In Verilog

```
module mem (addr, data_in, we, clk, data_out);
  parameter  Nawidth = 2;  // arbitrary default
  parameter  Ndwidth = 4;  // arbitrary default
  input  we, clk;  // write enable and clock
  input  [Nawidth-1:0] addr;
  input  [Ndwidth-1:0] data_in;
  output [Ndwidth-1:0] data_out;
  // memory array:
  reg [Ndwidth-1:0] mem [2**Nawidth-1:0];

  always @(posedge clk) begin
    if (we) begin
      mem[addr] <= data_in;
    end
  end

  assign data_out = mem[addr];
endmodule
```

***Note 1**: Power operator **; for example: a ** b raises a to the power b*

***Note 2**: How to instantiate memory modules with different values for Naddr and Ndata: link*



Verilog

- Parameterized modules
- Conditional operator

Parameterized Verilog Modules

Parameterized Verilog Modules

- Verilog parameters can be put to good use, to allow instantiated modules to accept inputs and outputs of arbitrary width

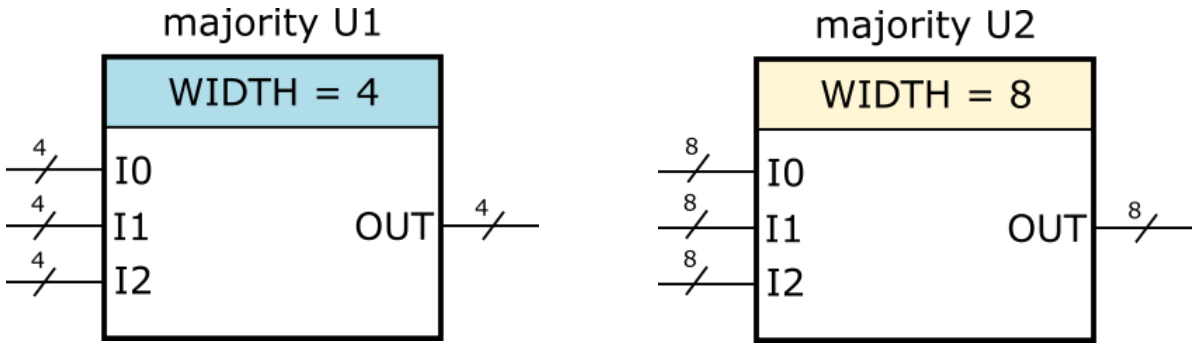
Parameterized Verilog Modules

Example: Majority Function

- Example: Consider a 3-input majority function, which produces a logic 1 if at least two of its inputs (i.e., the majority of its inputs) are logic 1

```
OUT = (I0 & I1) | (I0 & I2) | (I1 & I2);
```

How can we write one **`majority`** module description but then create two instances, **`U1`** and **`U2`**, each accepting different **`WIDTH`** of inputs and outputs?

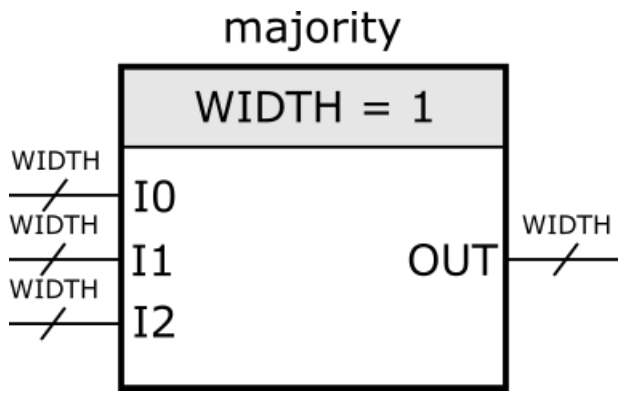


Parameterized Verilog Module

Example: Majority Function, Contd.

- Consider a 3-input majority function, which produces a logic 1 if at least two of its inputs (i.e., the majority of its inputs) are logic 1

```
module majority(I0, I1, I2, OUT);
  parameter  WIDTH = 1;  // default parameter value
  input     [WIDTH-1:0] I0, I1, I2;
  output    [WIDTH-1:0] OUT;

  assign OUT = (I0 & I1) | (I0 & I2) | (I1 & I2);
endmodule
```



- If we instantiate the module without modifying the value of the parameter `WIDTH`, the default value will be considered
  - In our example, the default is `WIDTH = 1`

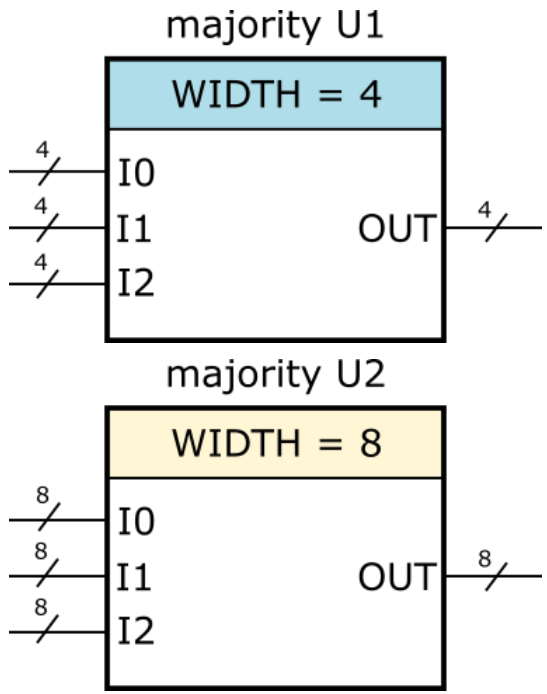
Parameterized Verilog Module

Example: Majority Function, Contd.

- Verilog allows us to override the default parameter value
  - Example, majority modules with 4-bit and 8-bit inputs and outputs:

```
// Instantiating majority module;
// Overriding the default value of WIDTH = 1;
// New value: WIDTH = 4;

majority #(.WIDTH(4)) U1 (.I0(A), .I1(B), .I2(C), .OUT(D));
```

```
// Instantiating majority module;
// Overriding the default value of WIDTH = 1;
// New value: WIDTH = 8;

majority #(.WIDTH(8)) U2 (.I0(X), .I1(Y), .I2(Z), .OUT(W));
```



Conditional Operator

In Verilog

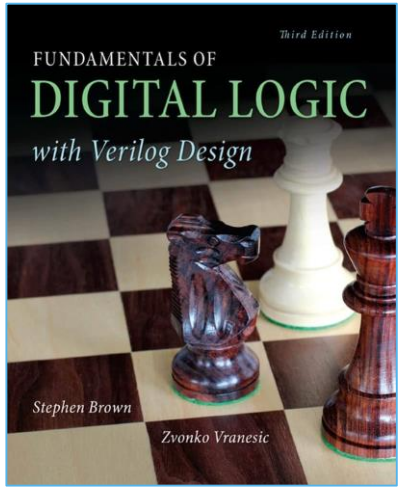
Verilog Conditional Operator

```
A ? B : C
```

- Conditional operator `?:` select one of the two alternate expressions (B, C) depending on the value of a logical expression (A)
  - If the logical expression (A) is true, it returns the first alternative (B)
  - Otherwise, it returns the second alternative (C)

Return to the Verilog description of 2-to-4 decoder: link

Literature



- Chapter 4: Combinational-Circuit Building Blocks
  - 4.2 Decoders
- Appendix A: Verilog Reference
  - A.6.3 Memories