



## Lecture 3

# Memory

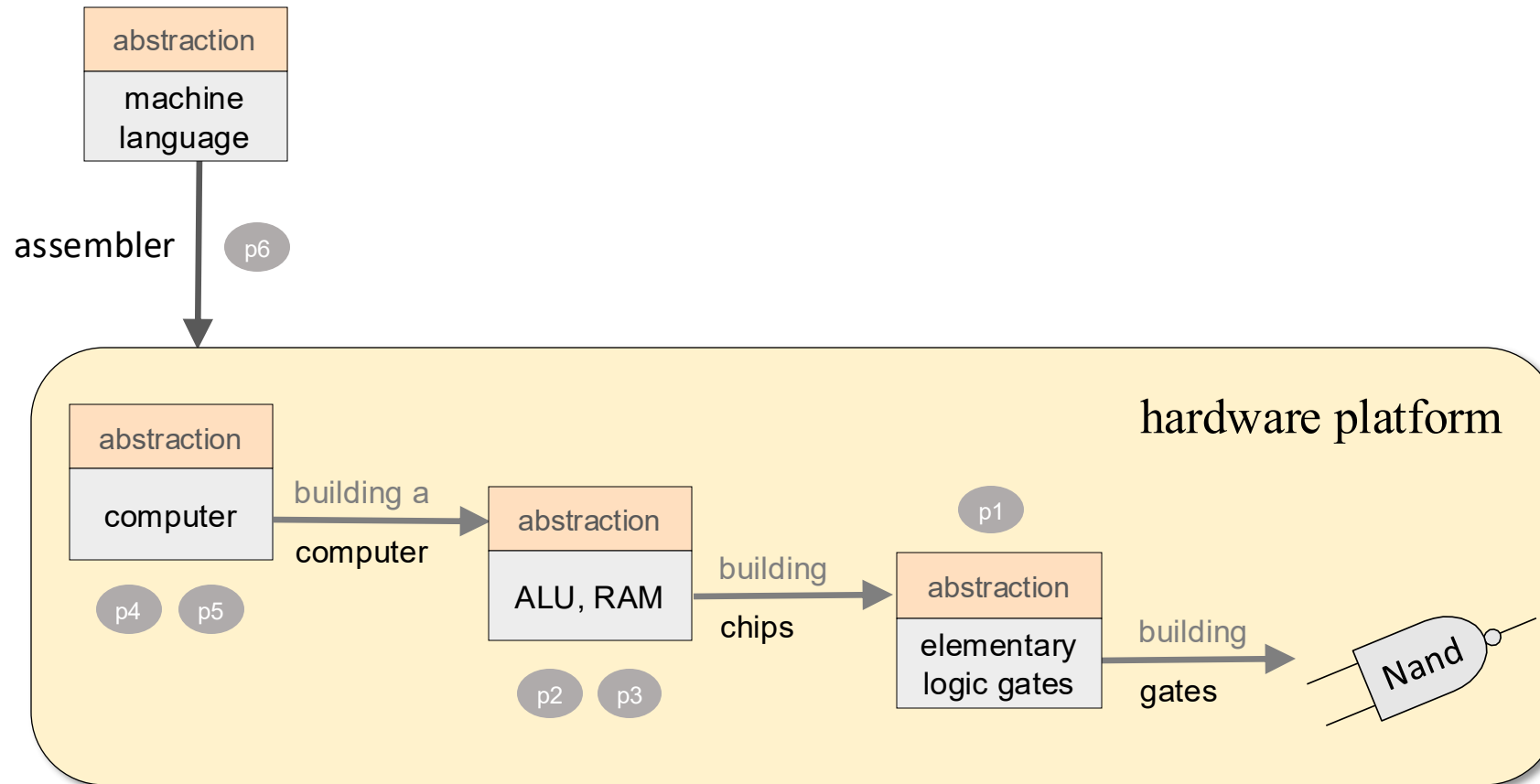
Slide deck for Chapter 3 of the book

*The Elements of Computing Systems* (2<sup>nd</sup> edition)

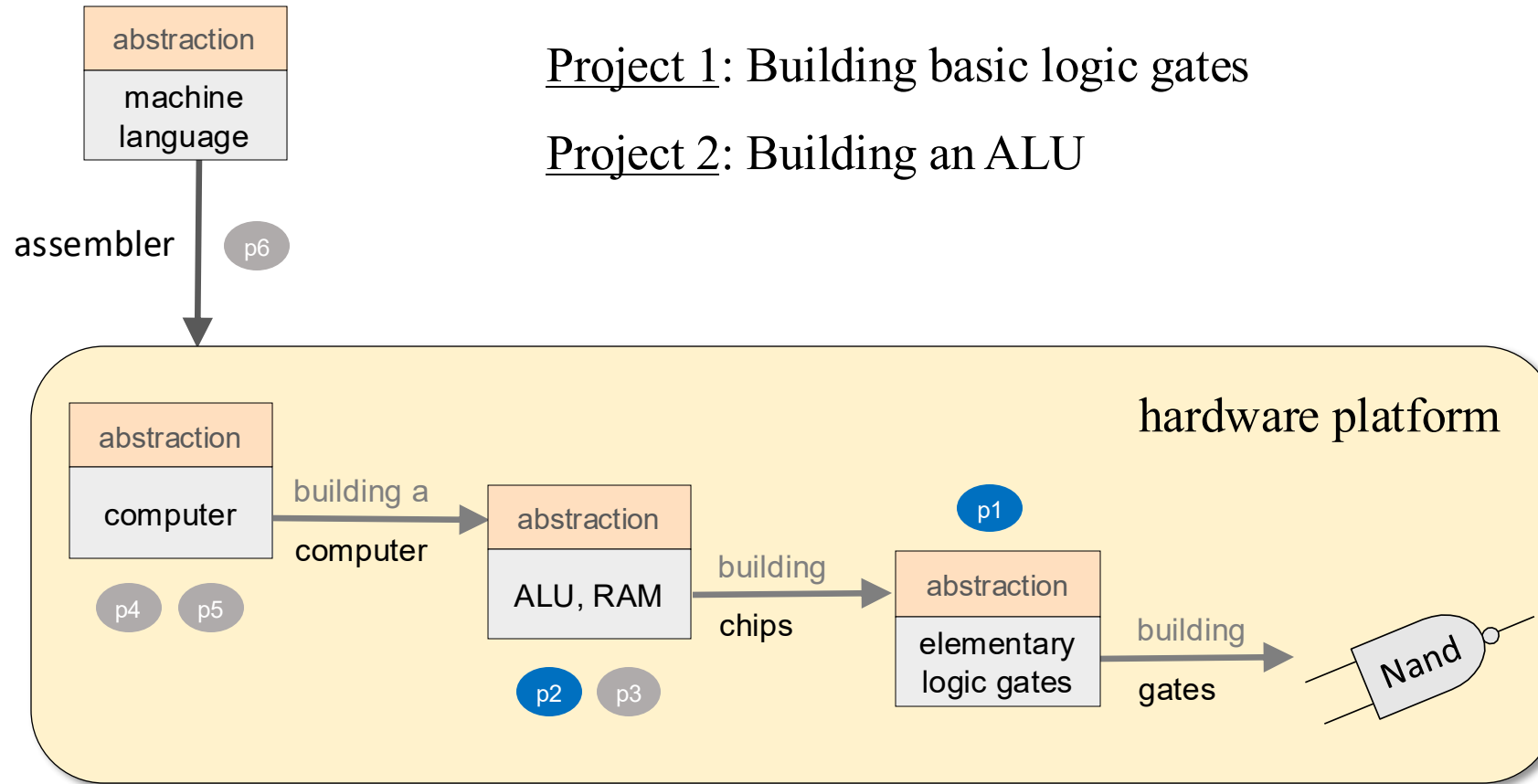
By Noam Nisan and Shimon Schocken

MIT Press

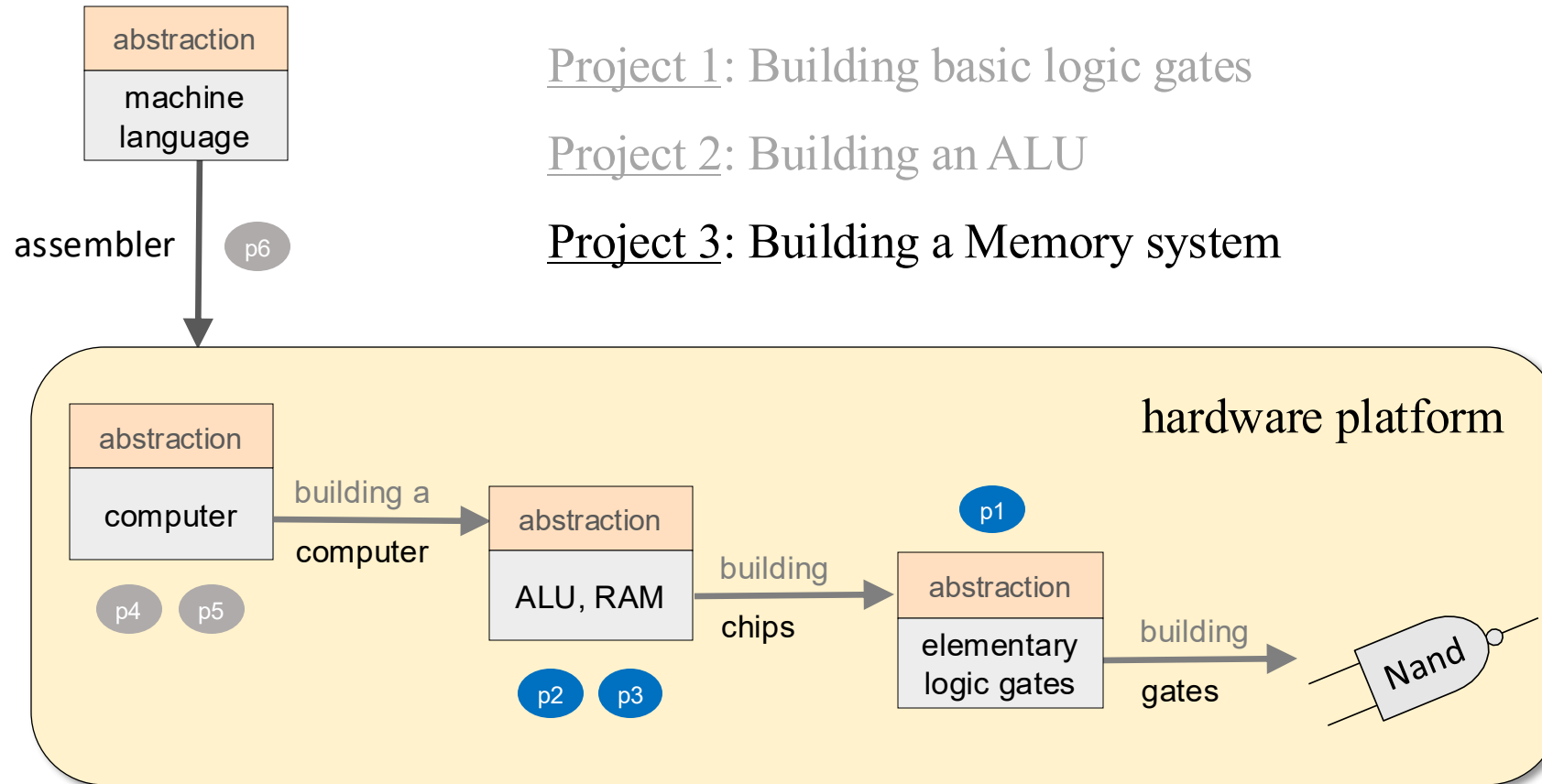
# Nand to Tetris Roadmap: Hardware



# Nand to Tetris Roadmap: Hardware



# Nand to Tetris Roadmap: Hardware

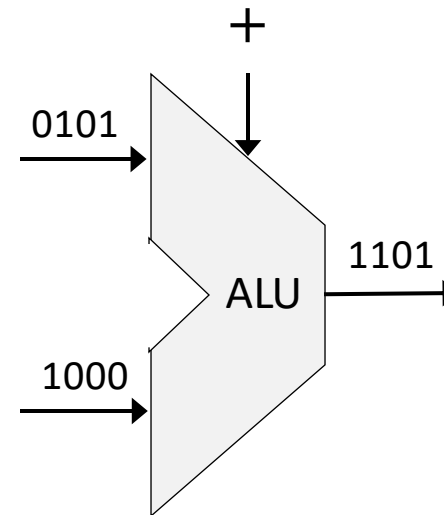


# Combinational logic

---

So far, when we built chips, we assumed that:

- The chip's current inputs are fixed and unchanging
- The chip's output is a function (“combination”) of the current inputs, and the current inputs only



# Combinational logic

---

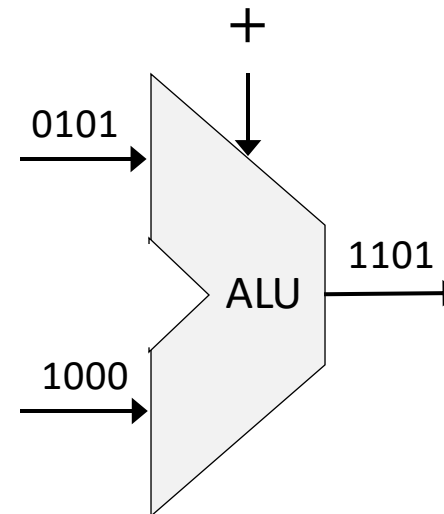
So far, when we built chips, we assumed that:

- The chip's current inputs are fixed and unchanging
- The chip's output is a function (“combination”) of the current inputs, and the current inputs only

This type of digital logic is called:

- *Combinational logic*
- *Time-independent logic*

ALU: The “topmost”  
combinational chip



# Hello, time

---

# Hello, time

---

## Software needs

- The hardware must be able to remember things, *over time*:
- The hardware must be able to do things, *one at a time* (sequentially):

Example (variables)

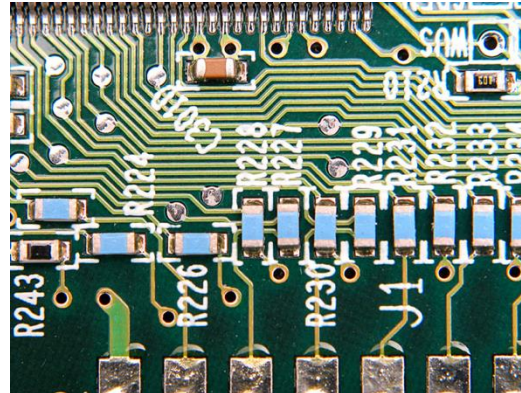
```
x = 17
```

Example (iteration)

```
for i in range(0, 10):  
    print(i)
```

## Hardware needs

- The hardware must handle the *physical time delays* associated with *moving* and *computing* data





# Hello, time

---

## Software needs

- The hardware must be able to remember things, *over time*:
- The hardware must be able to do things, *one at a time* (sequentially):

Example (variables):

```
x = 17
```

Example (iteration)

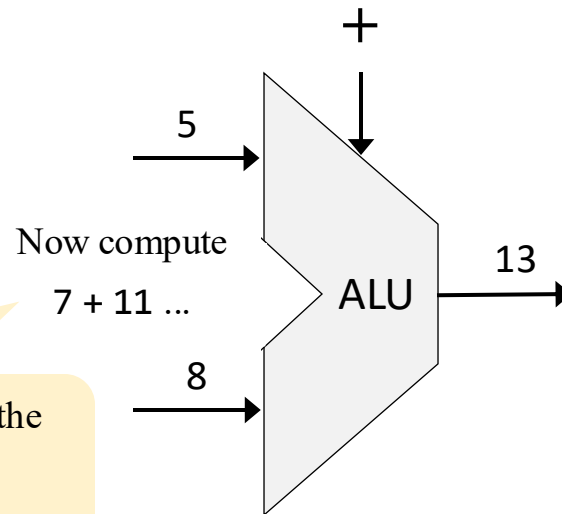
```
for i in range(0, 10):  
    print(i)
```

## Hardware needs

- The hardware must handle the *physical time delays* associated with *moving* and *computing* data

It will take time before 7 and 11 will settle down in the input pins, and before the sum 7 + 11 will stabilize.

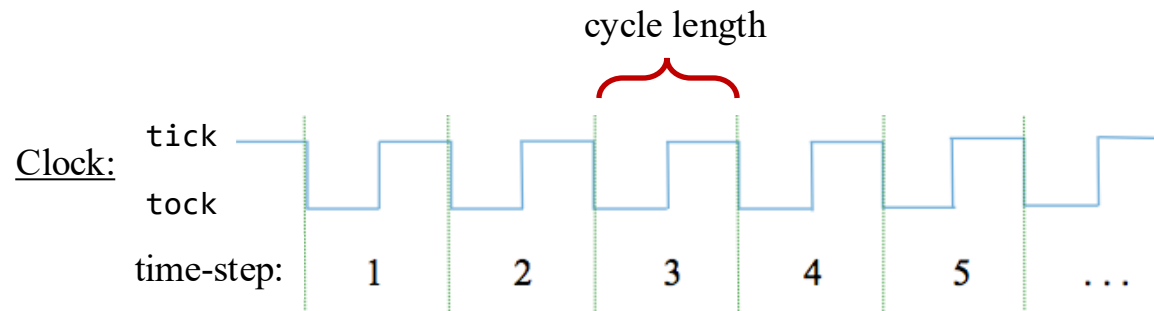
*Till then, the ALU will output nonsense.*



# Hello, time

---

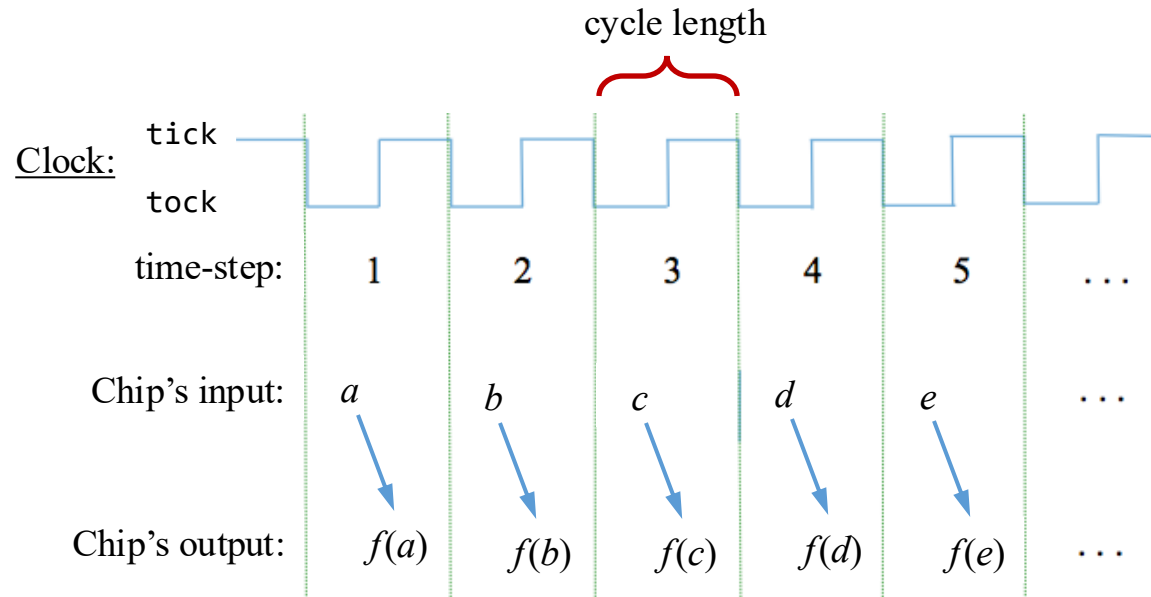
Solution: Neutralize the time delays by using *discrete time*



# Hello, time

---

Solution: Neutralize the time delays by using *discrete time*



## Hardware design decisions

- Set the *cycle length* to be slightly  $>$  than the maximum time delay
- Observe / use the chips' outputs only at the end of cycles (time-steps), ignoring what happens within cycles

Details later.

# Memory

---

**Memory** (Wikipedia): The faculty of the brain by which data or information is encoded, stored, and retrieved when needed. It is the retention of information *over time* for the purpose of influencing future action

Note: Memory is time-based

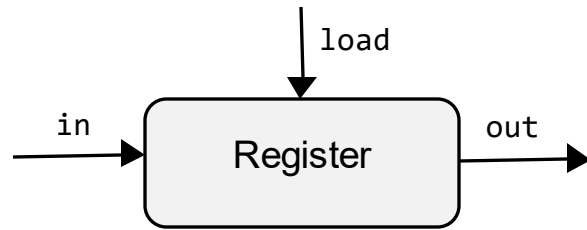
We remember *now* what was committed to memory *earlier*.



*It's a poor sort of memory  
that only works backwards.*

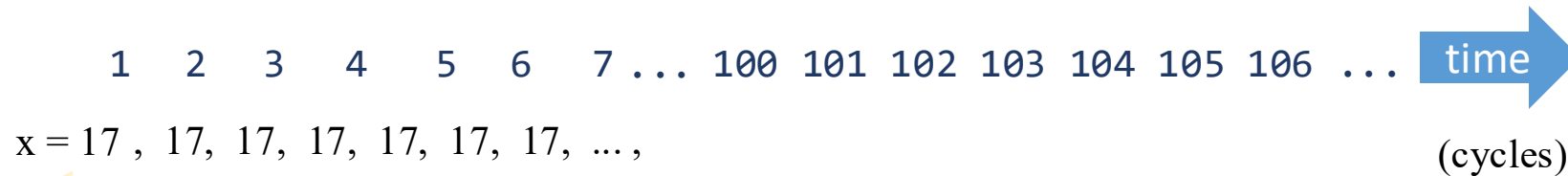
*-Lewis Carroll, through the White Queen*

# Memory



## Basic operations

- “Loading” a value
- “Storing” a value



loading

storing

x = 21 , 21, 21, 21, 21, 21, 21, ...

loading

storing

The challenge: Building chips that facilitate this functionality,  
i.e. chips designed to *maintain state* and *change state*.

# Chapter 3: Memory

---

## Abstraction

- Representing time
- Clock
- Registers
- RAM
- Counter

## Implementation

- Data Flip Flop
- Registers
- RAM
- PC

# Chapter 3: Memory

---

## Abstraction

 Representing time

- Clock
- Registers
- RAM
- Counter

## Implementation

- Data Flip Flop
- Registers
- RAM
- PC

# Representing time

---

physical  
time:



Arrow of time:

Continuous



# Representing time

---

physical  
time:



Arrow of time:

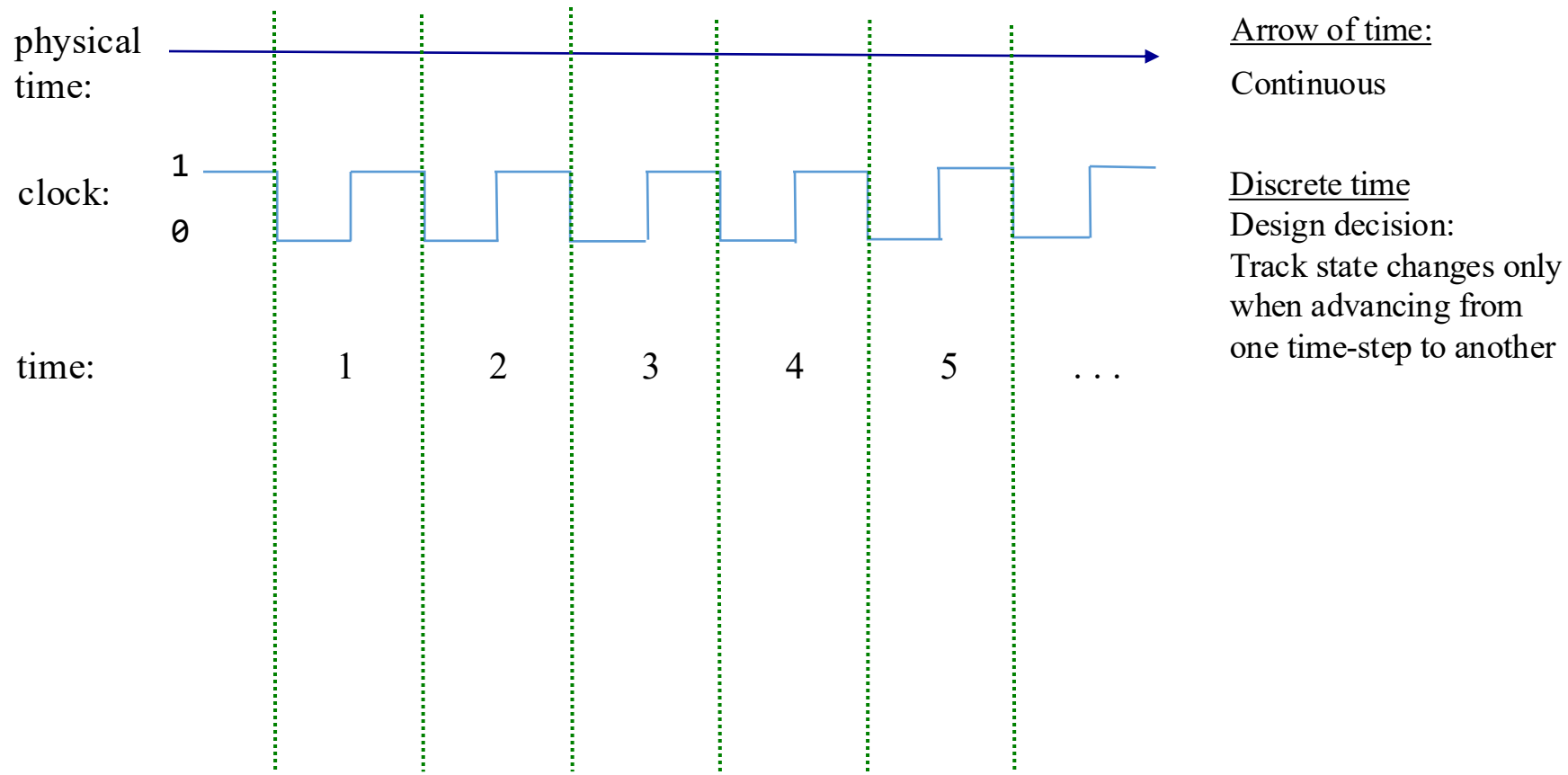
Continuous

clock:

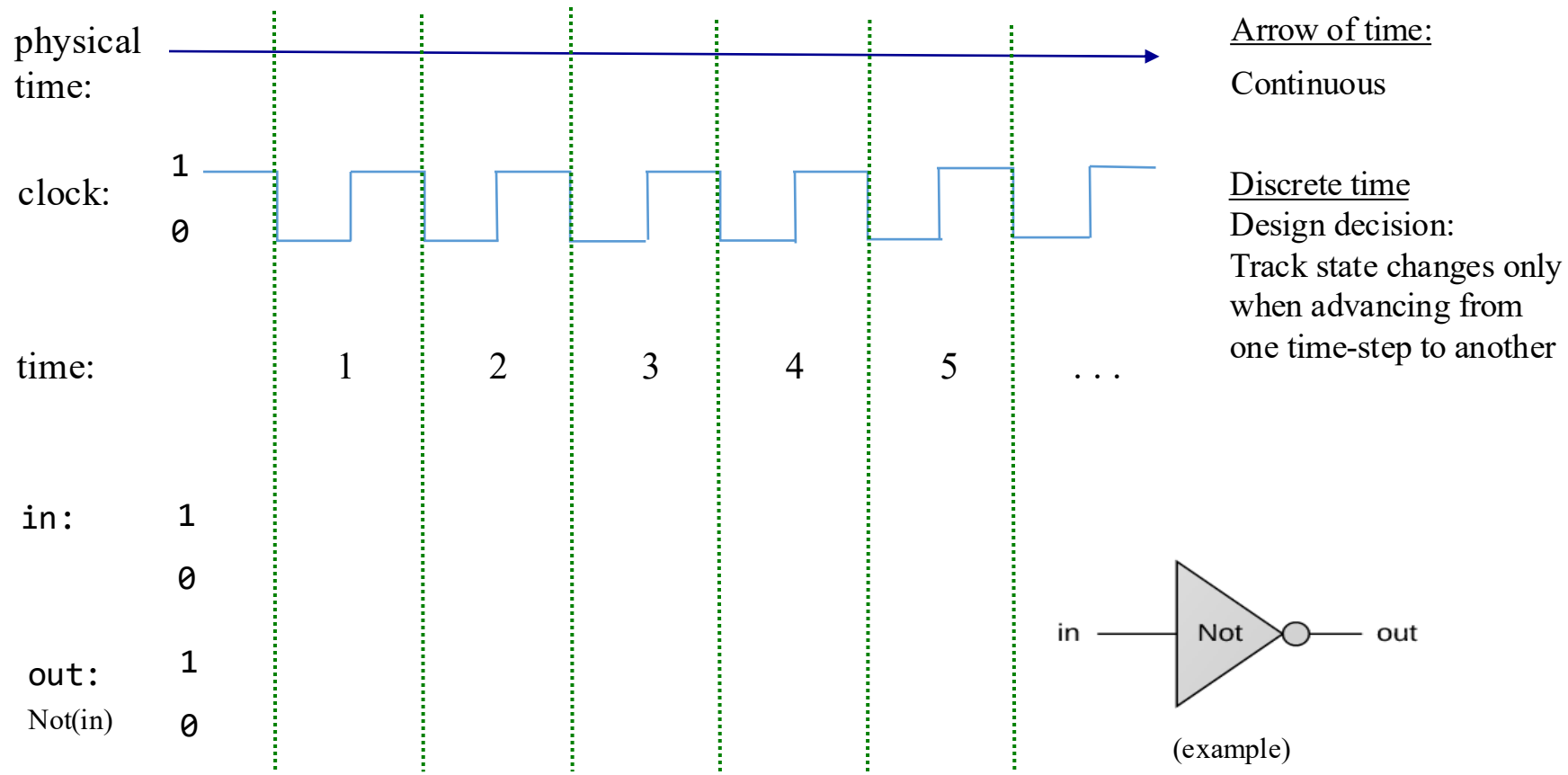


# Representing time

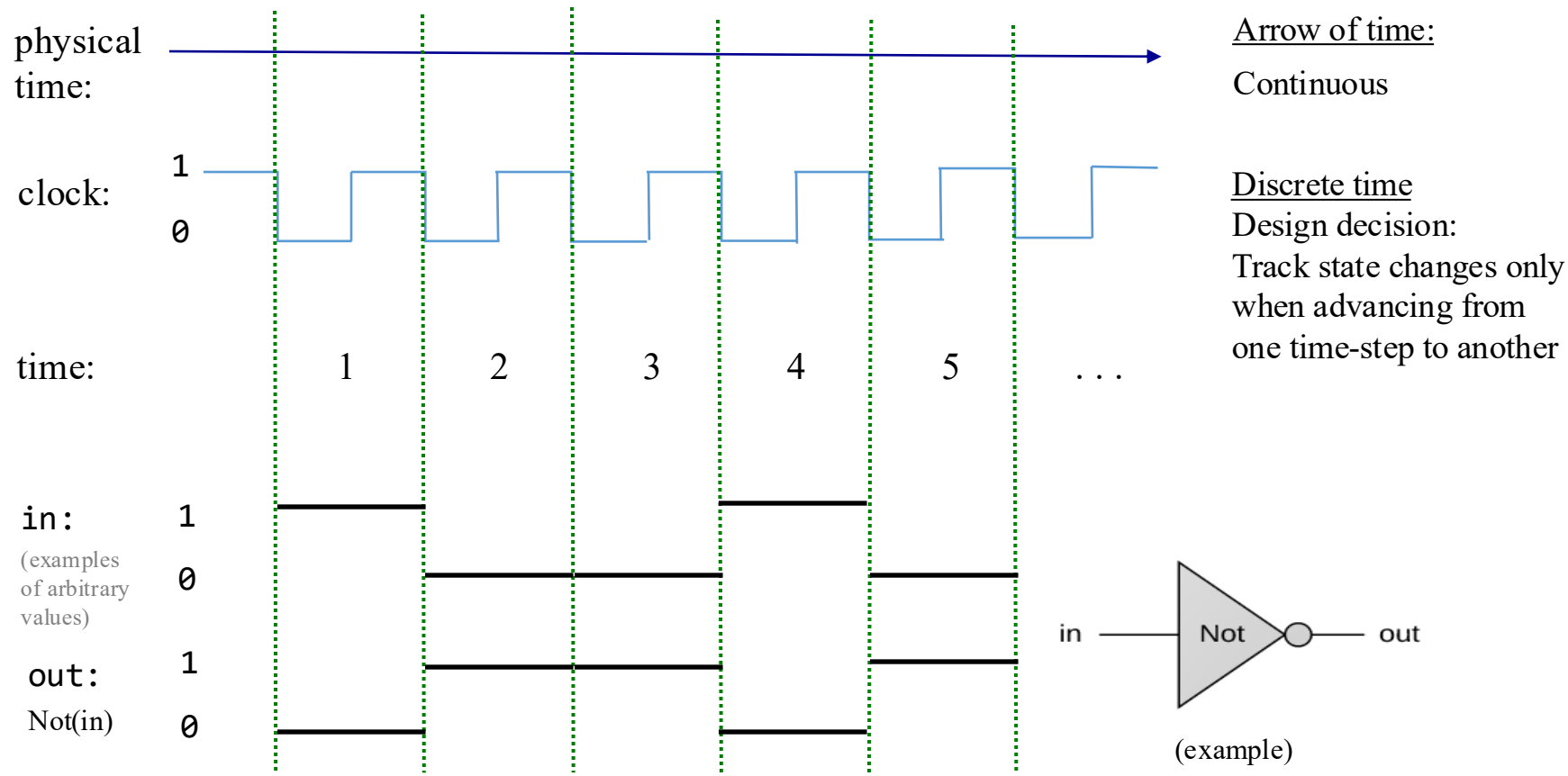
---



# Chip behavior over time (example: Not gate)



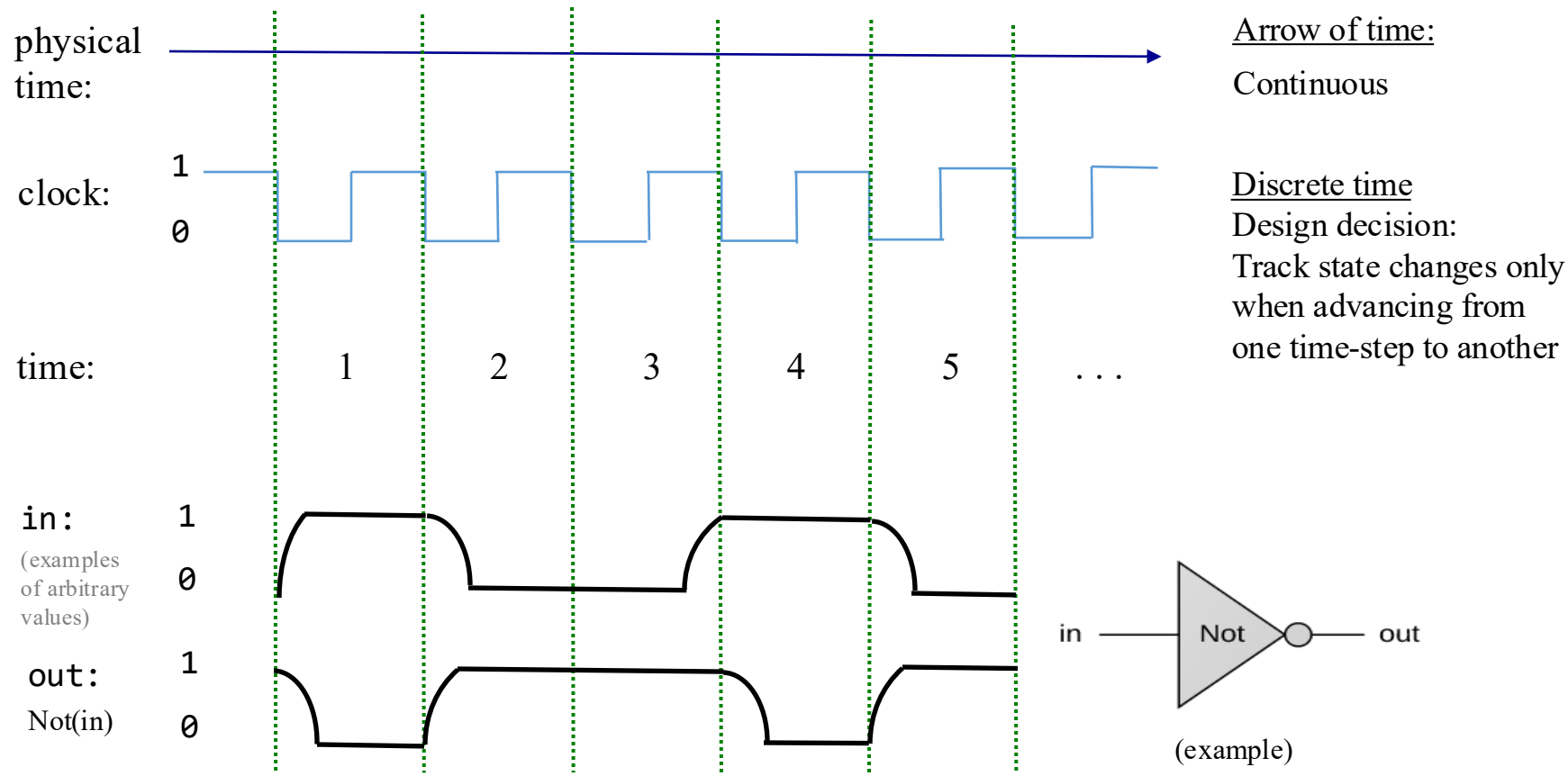
# Chip behavior over time (example: Not gate)



Desired / idealized behavior of the in and out signals:

That's how we *want* the hardware to behave

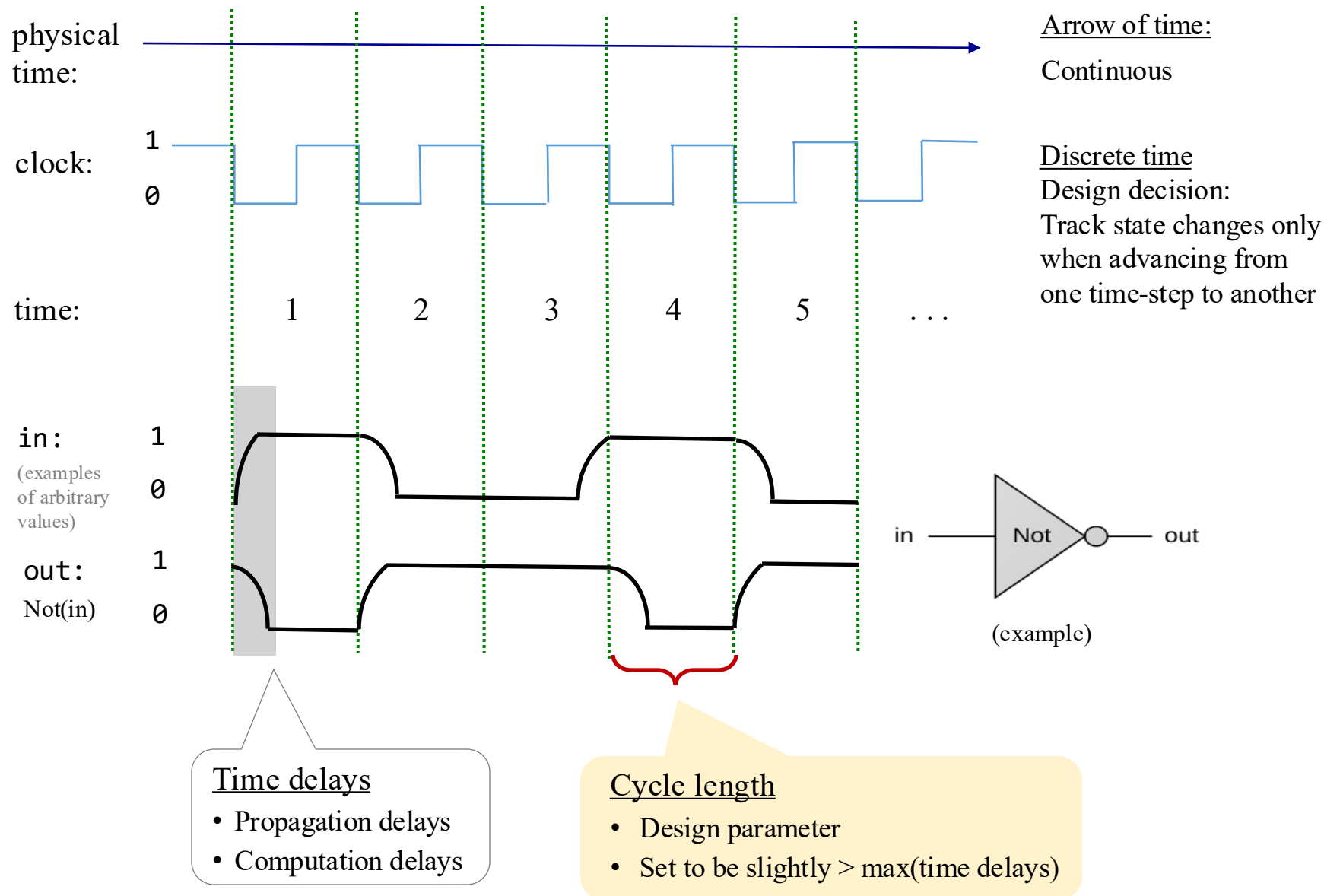
# Chip behavior over time (example: Not gate)



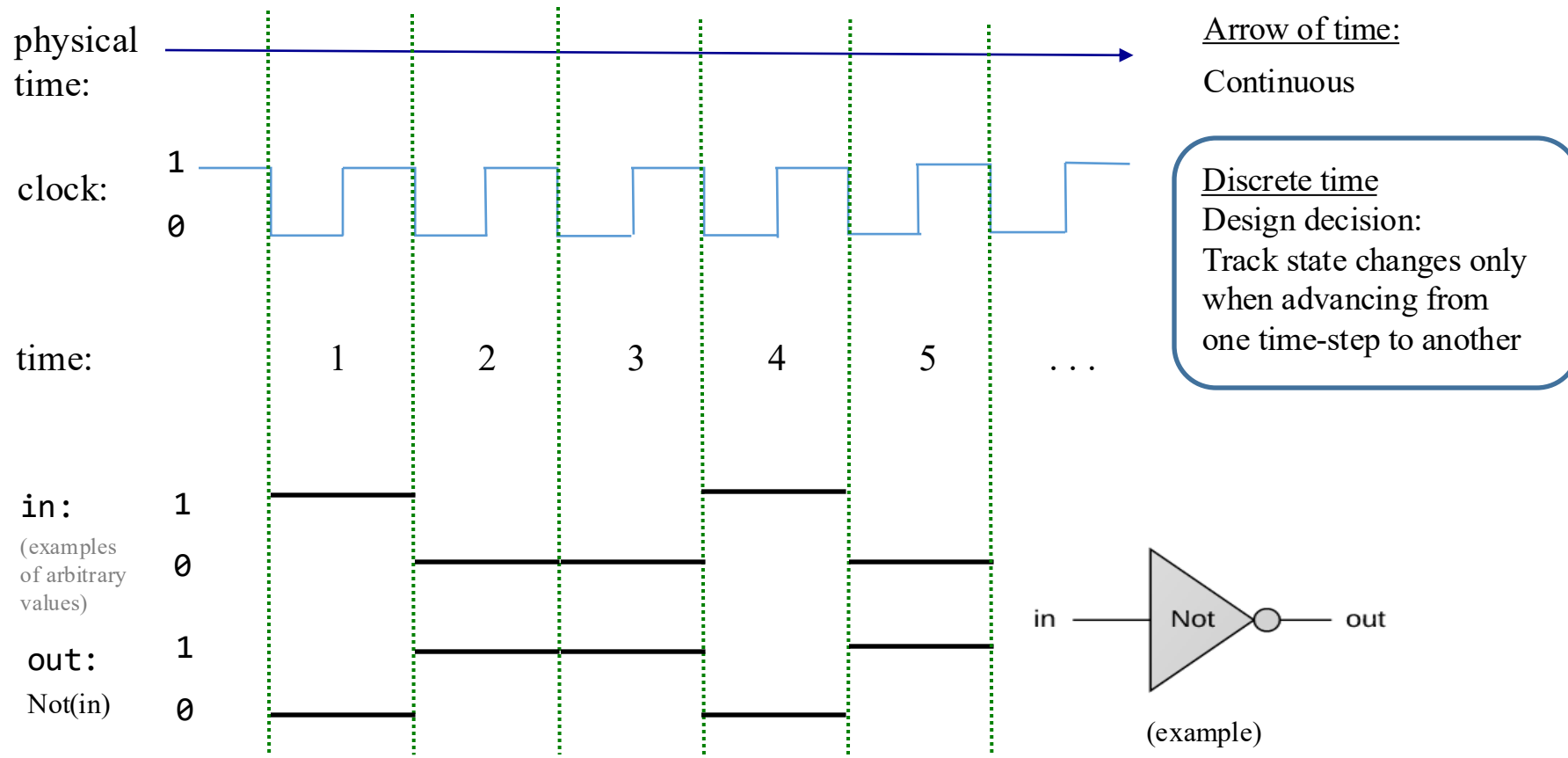
Actual behavior of the in and out signals:

Influenced by physical time delays

# Chip behavior over time (example: Not gate)



# Chip behavior over time (example: Not gate)



## Resulting effect

Since we track state changes only at cycle ends,

Combinational chips (like `Not`) react “immediately” to their inputs.

# Chapter 3: Memory

---

## Abstraction

✓ Representing time

➡ Clock

- Registers
- RAM
- Counter

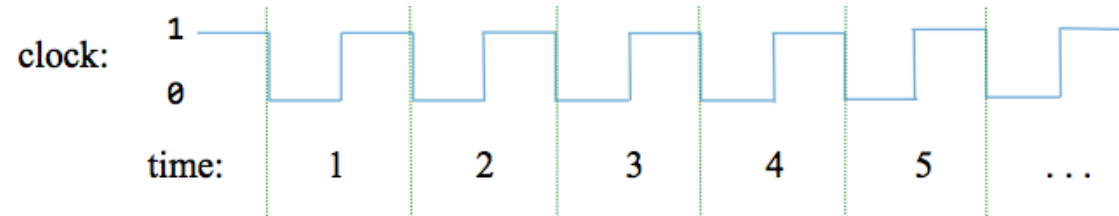
## Implementation

- Data Flip Flop
- Registers
- RAM
- PC



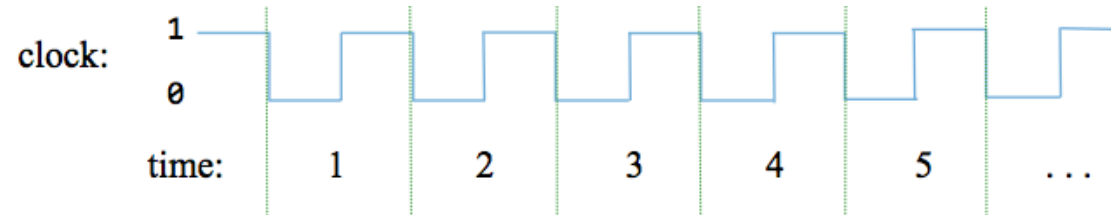
# Clock

---



# Clock: Simulated implementation

---



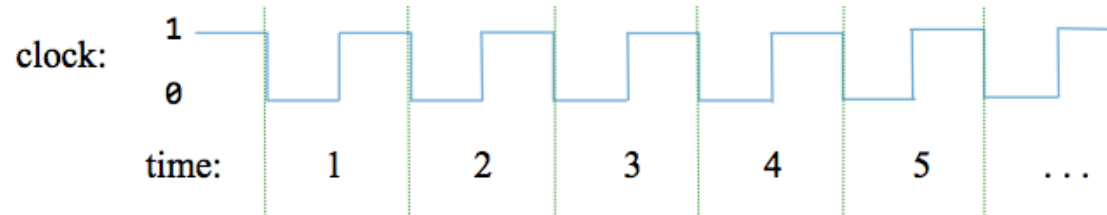
## Interactive simulation

A clock icon, used to generate a sequence of tick-tock signals:

0, 0+, 1, 1+, 2, 2+, 3, 3+, ...



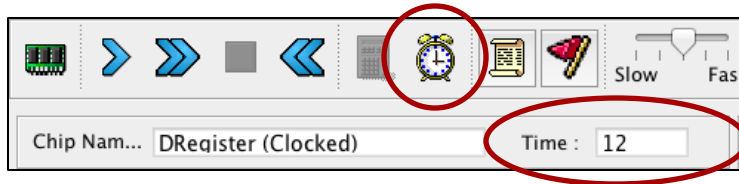
# Clock: Simulated implementation



## Interactive simulation

A clock icon, used to generate a sequence of tick-tock signals:

0, 0+, 1, 1+, 2, 2+, 3, 3+, ...

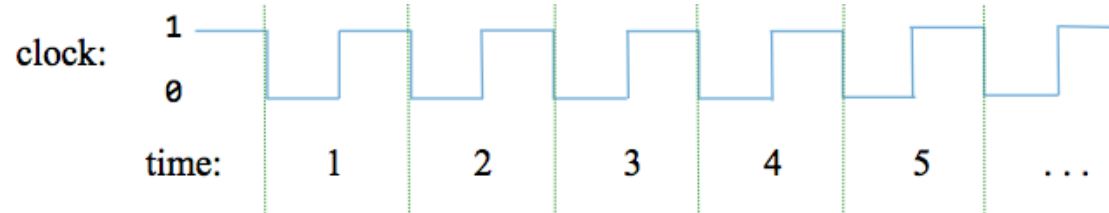


## Script-based simulation

“tick” and “tock” commands,  
used to advance the clock:

```
...  
// Sets inputs, advances the clock, and  
// writes output values as it goes along.  
set in 19,  
set load 1,  
tick,  
output,  
tock,  
output,  
tick, tock,  
output,  
...
```

# Clock: Physical implementation

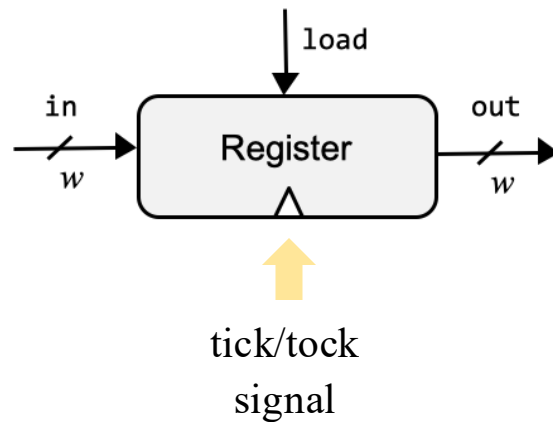


## Physical clock

An *oscillator* is used to deliver an ongoing train of “tick/tock” signals

“1 MHz electronic oscillator circuit uses the resonant properties of an internal quartz crystal to control the frequency. Provides the clock signal for digital devices such as computers.” (Wikipedia)

## Example (how the clock signal comes to play)



(Chip diagram convention:  
A triangle icon represents a clock signal input)

# Chapter 3: Memory

---

## Abstraction

✓ Representing time

✓ Clock

➡ Registers

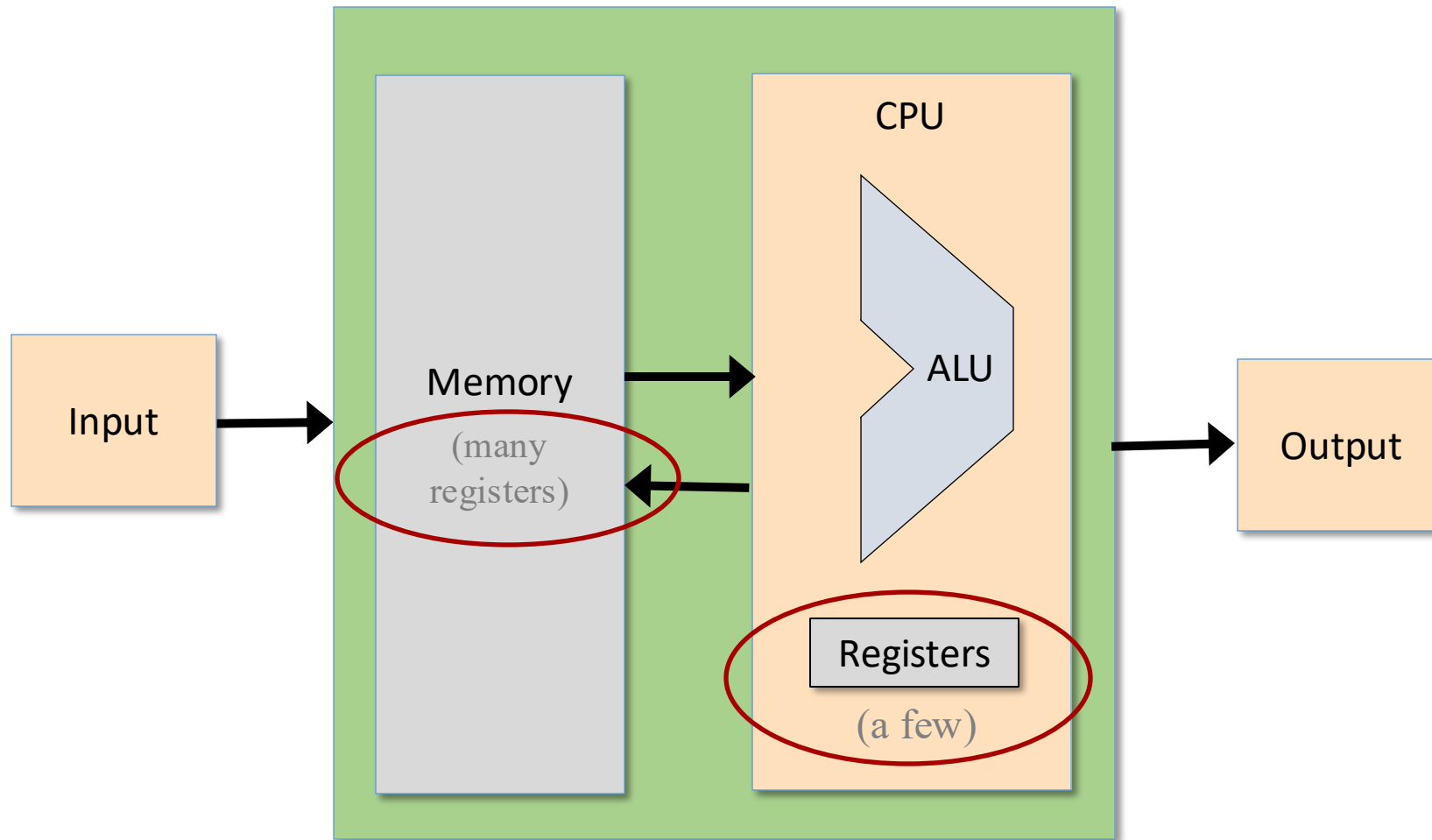
- RAM
- Counter

## Implementation

- Data Flip Flop
- Registers
- RAM
- PC

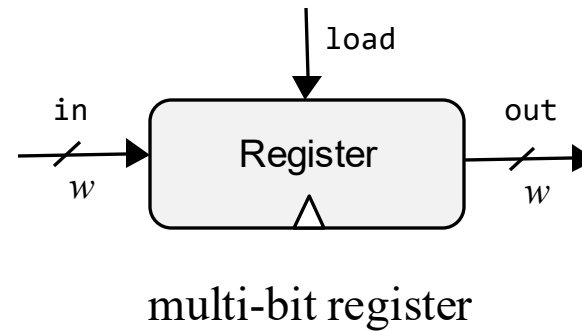
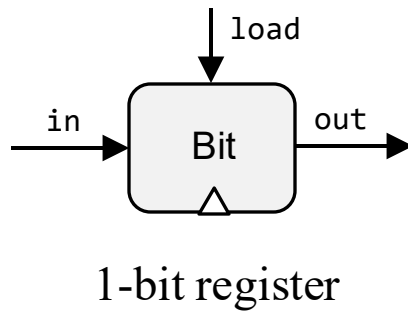
# Registers

---



Computer Architecture

# Registers



Designed to:

- “Store” a value , until...
- “Loaded” with a new value

time:

$x = 17, 17, 17, 17, 17, 17, 17, 17, \dots, 17$

loading

storing

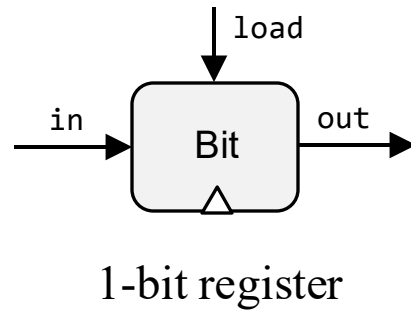
$x = 21, 21, 21, 21, 21, 21, \dots, 21$

loading

storing

# 1-Bit register

---

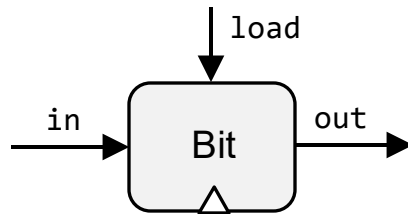


```
if load( $t$ ):  
    out( $t + 1$ ) = in( $t$ )  
else  
    out( $t + 1$ ) = out( $t$ )
```



# 1-Bit register

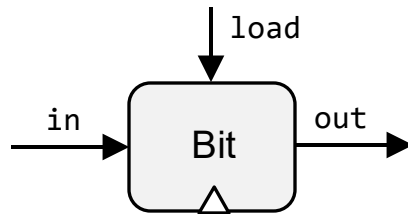
---



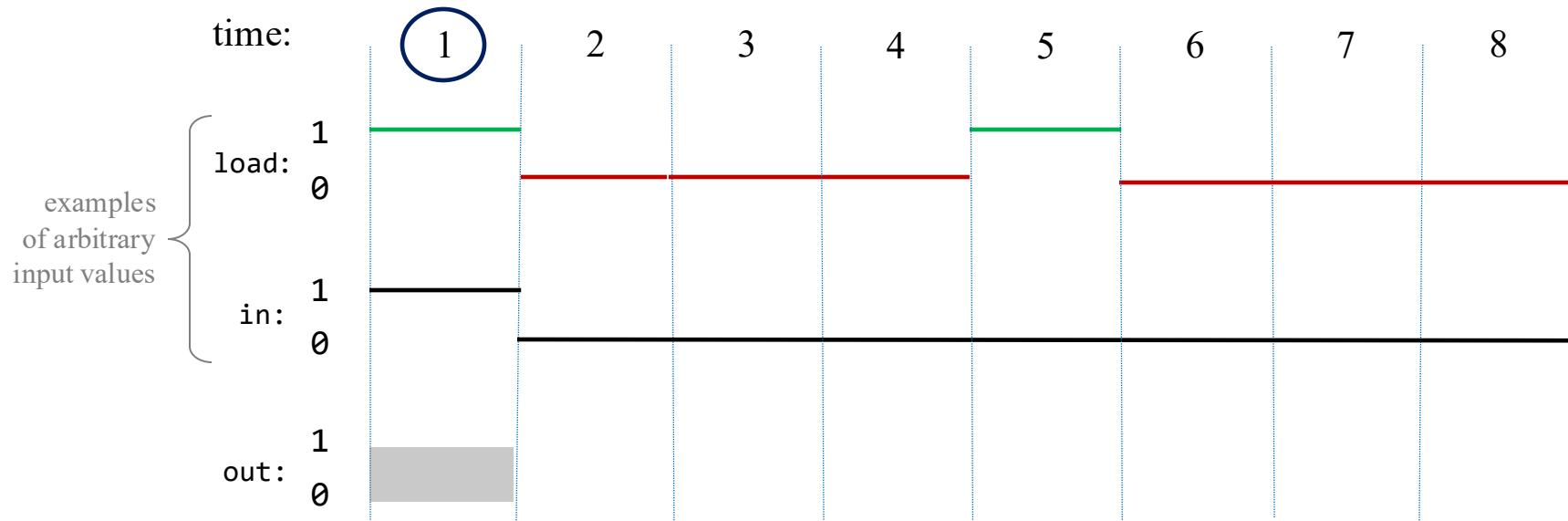
```
if load(t):  
    out(t + 1) = in(t)  
else:  
    out(t + 1) = out(t)
```

time:	1	2	3	4	5	6	7	8
load:	1							
	0							
in:	1							
	0							
out:	1							
	0							

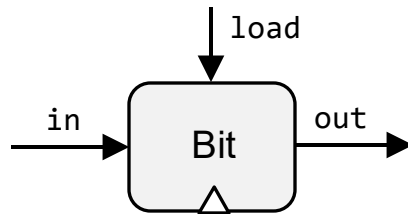
# 1-Bit register



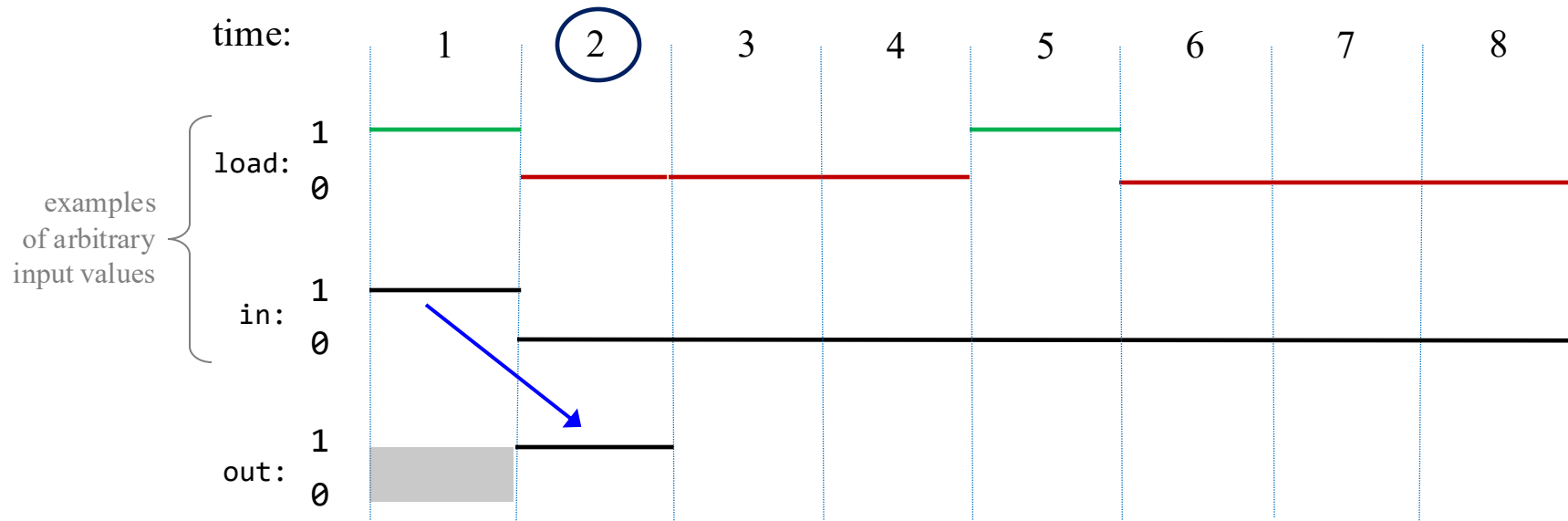
```
if load(t):  
    out(t + 1) = in(t)  
else  
    out(t + 1) = out(t)
```



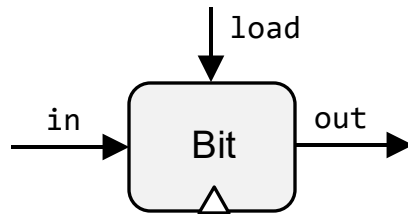
# 1-Bit register



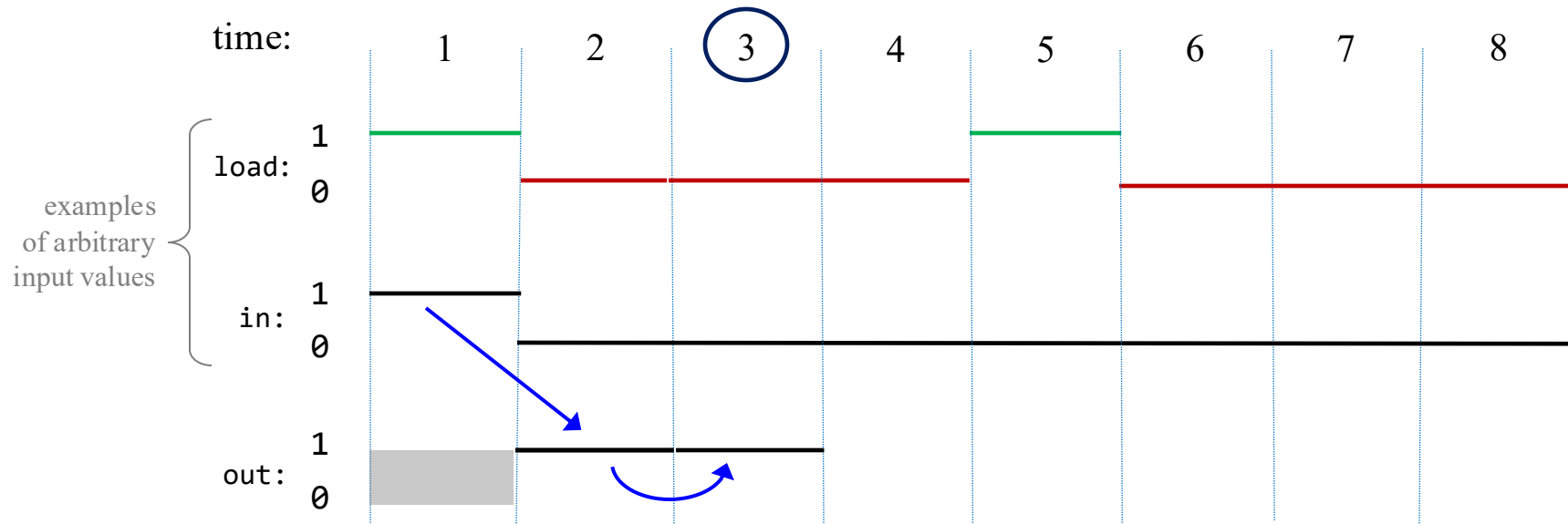
```
if load(t):  
    out(t + 1) = in(t)  
else  
    out(t + 1) = out(t)
```



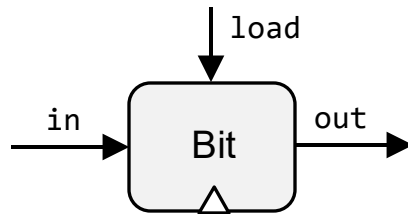
# 1-Bit register



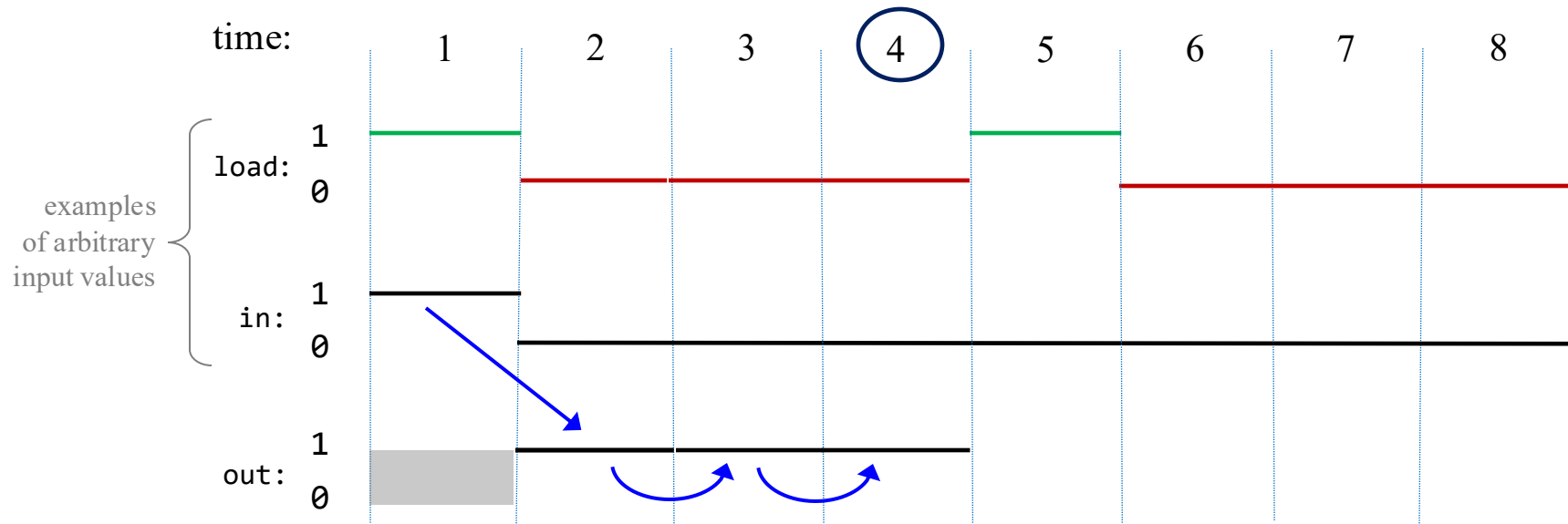
```
if load(t):  
    out(t + 1) = in(t)  
else  
    out(t + 1) = out(t)
```



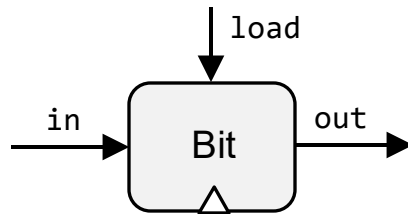
# 1-Bit register



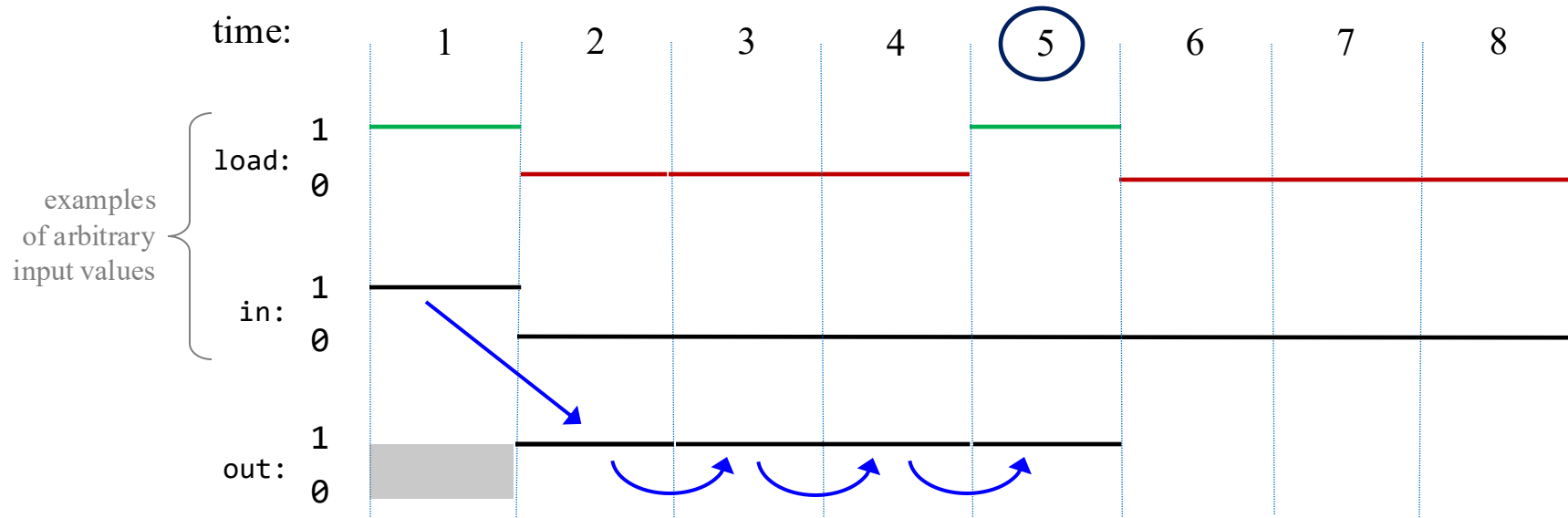
```
if load(t):  
    out(t + 1) = in(t)  
else  
    out(t + 1) = out(t)
```



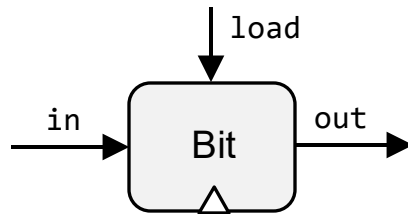
# 1-Bit register



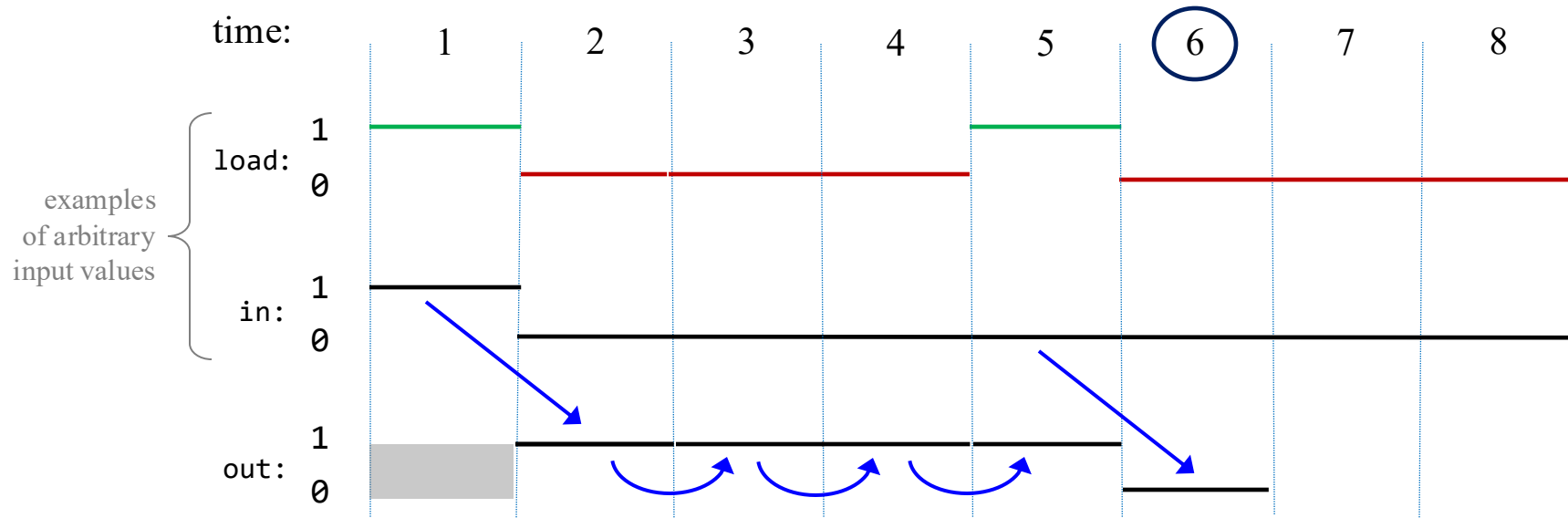
```
if load(t):  
    out(t + 1) = in(t)  
else  
    out(t + 1) = out(t)
```



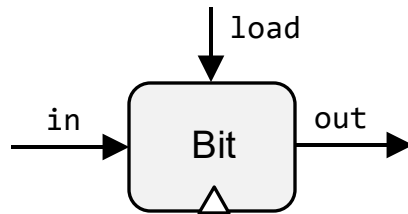
# 1-Bit register



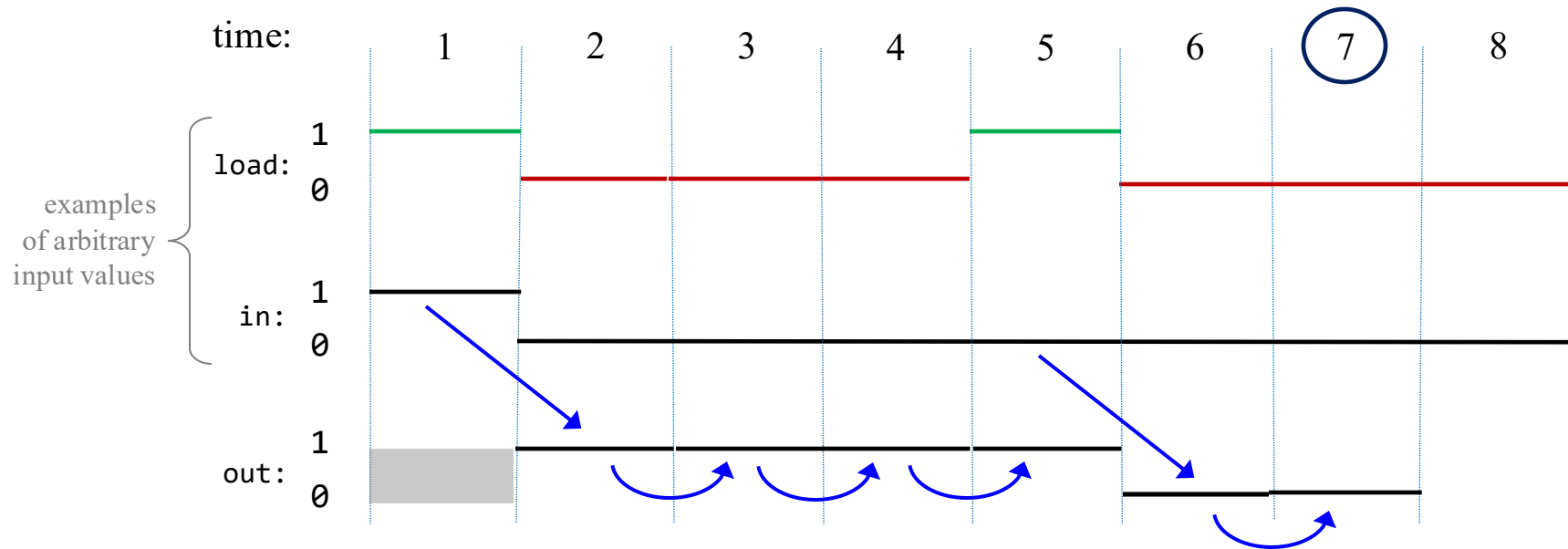
```
if load(t):  
    out(t + 1) = in(t)  
else  
    out(t + 1) = out(t)
```



# 1-Bit register

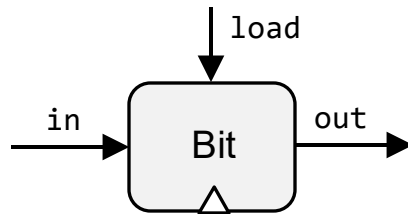


```
if load(t):  
    out(t + 1) = in(t)  
else  
    out(t + 1) = out(t)
```

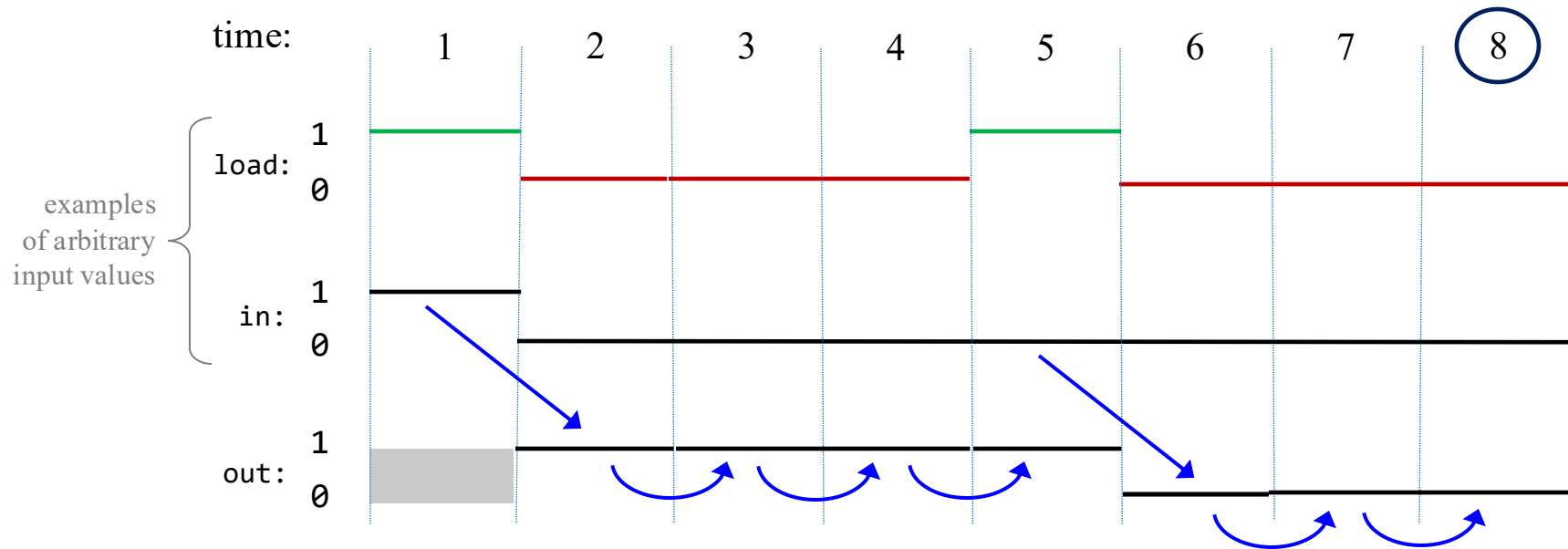




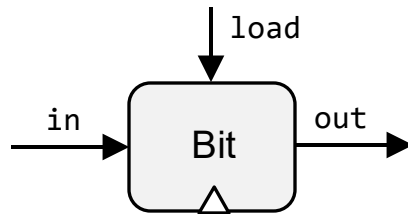
# 1-Bit register



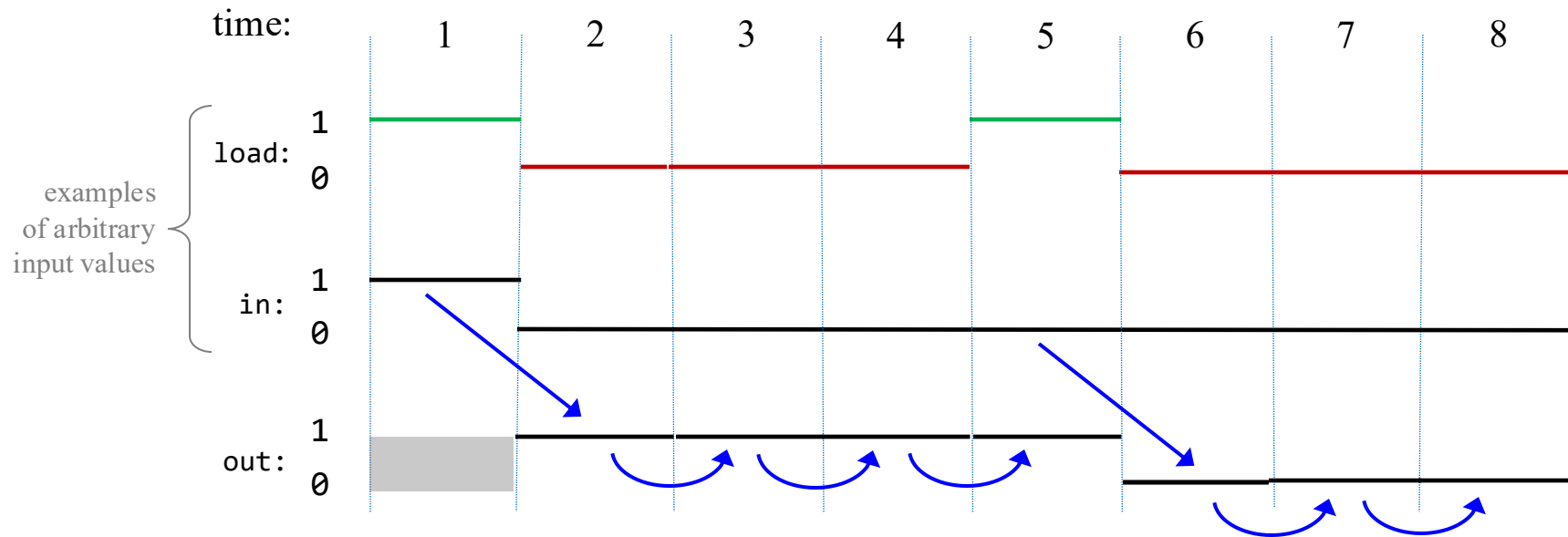
```
if load(t):  
    out(t + 1) = in(t)  
else  
    out(t + 1) = out(t)
```



# 1-Bit register



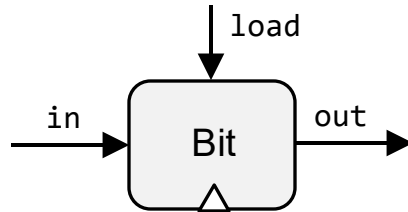
```
if load(t):  
    out(t + 1) = in(t)  
else  
    out(t + 1) = out(t)
```



Resulting behavior: “loading” “storing” “loading” “storing”

# 1-Bit register

---



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

## Usage:

### **To read:**

probe out      (out always emits the register's state)

### **To write:**

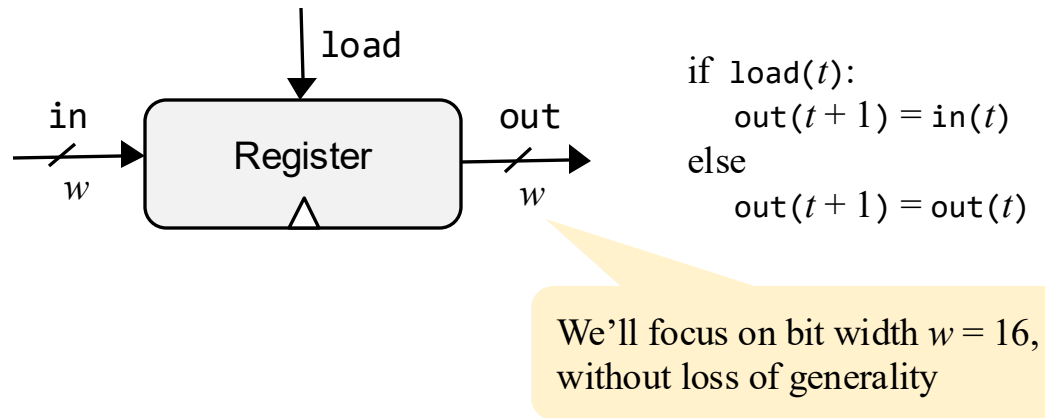
set in =  $v$   
set load = 1

Result: The register's state becomes  $v$ ;  
From the next time-step onward, out will emit  $v$ ,  
until the next load.

Best practice: After writing, set load to 0

# Multi-bit register

---



Load / store behavior: Exactly the same as a 1-bit register

Read / write usage: Exactly the same as a 1-bit register



# Chapter 3: Memory

---

## Abstraction

✓ Representing time

✓ Clock

✓ Registers

➡ RAM

- Counter

## Implementation

- Data Flip Flop

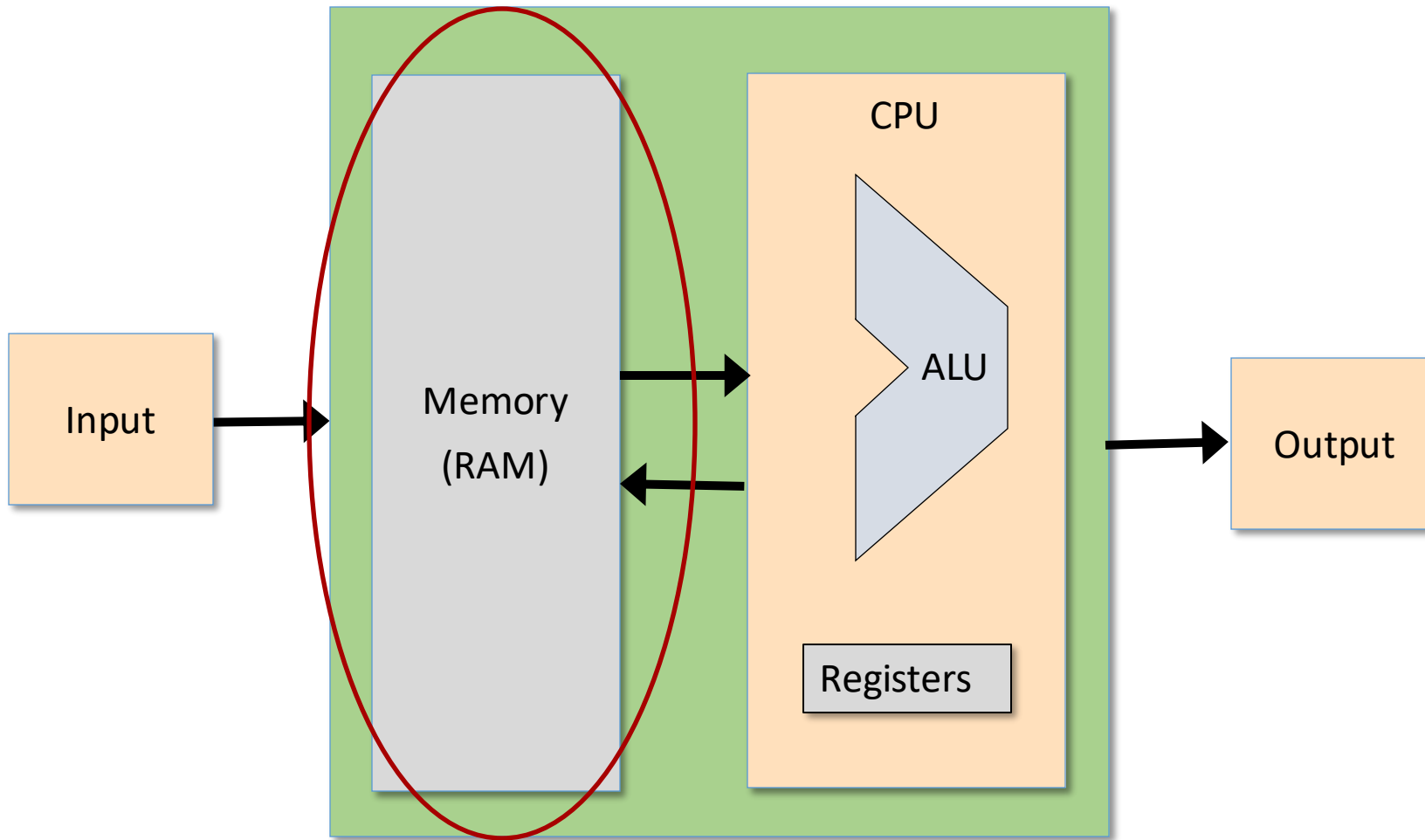
- Registers

- RAM

- PC

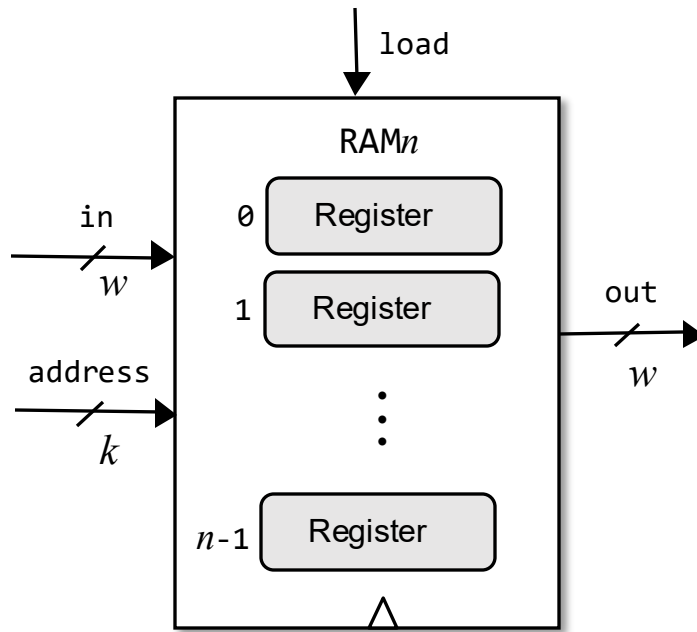
# Computer architecture

---



# RAM

---



Suppose that the RAM size is  $n$   
(for example,  $n = 8$  registers);  
What should be the value of  $k$ ?

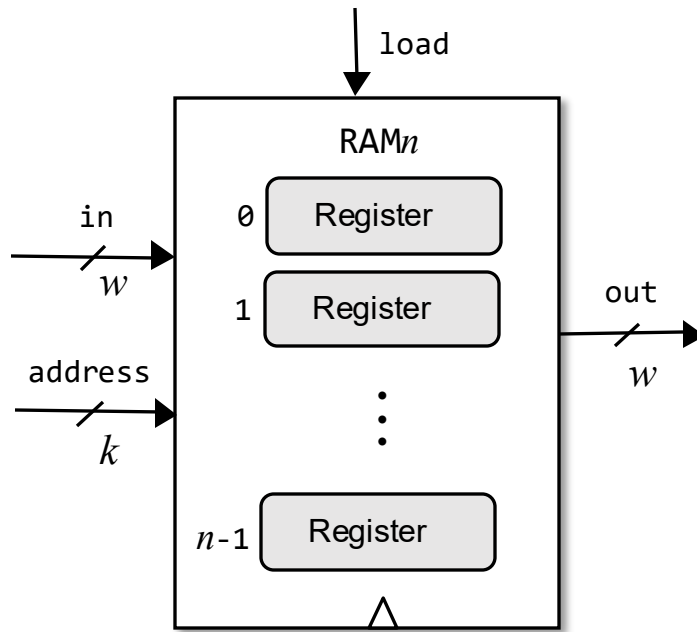
$$k = \log_2 n$$

Abstraction: A sequence of  $n$  addressable,  $w$ -bit registers

Word width: Typically  $w=16, 32, 64, 128$  bits (Hack computer:  $w=16$ )

# RAM

---



## Behavior

If  $load == 0$ , the RAM maintains its state

If  $load == 1$ ,  $RAM[address]$  is set to the value of  $in$

The loaded value will be emitted by  $out$  from the next time-step (cycle) onward

(Only one RAM register is selected;  
All the other registers are not affected)

## Usage

**To read register  $i$  :**

set  $address = i$ ,  
probe  $out$

( $out$  always emits  
the value of  $RAM[i]$ )

**To write  $v$  in register  $i$  :**

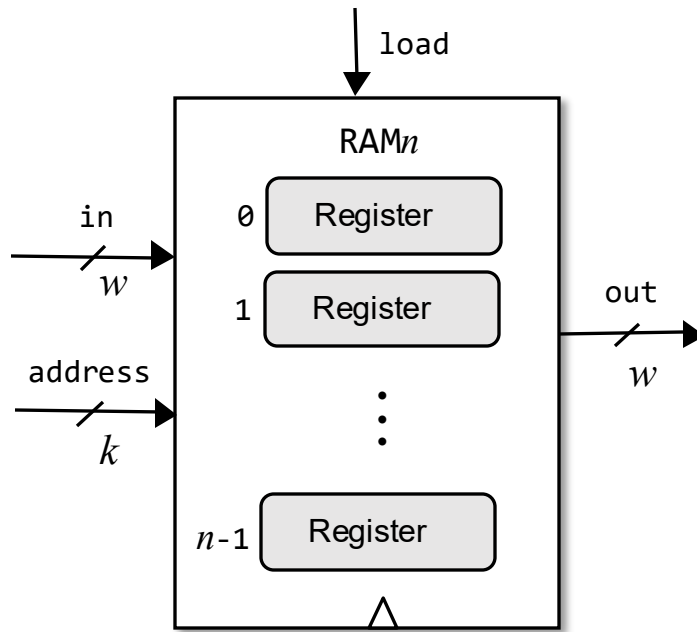
set  $address = i$ ,  
set  $in = v$ ,  
set  $load = 1$

Result:  $RAM[i] \leftarrow v$   
From the next time-step onward  $out$  will emit  $v$



# RAM

---



## Why “Random Access Memory”?

Irrespective of the RAM size ( $n$ ), any randomly selected register can be accessed “instantaneously”, in one time cycle.



# Chapter 3: Memory

---

## Abstraction

✓ Representing time

✓ Clock

✓ Registers

✓ RAM

➡ Counter

## Implementation

- Data Flip Flop

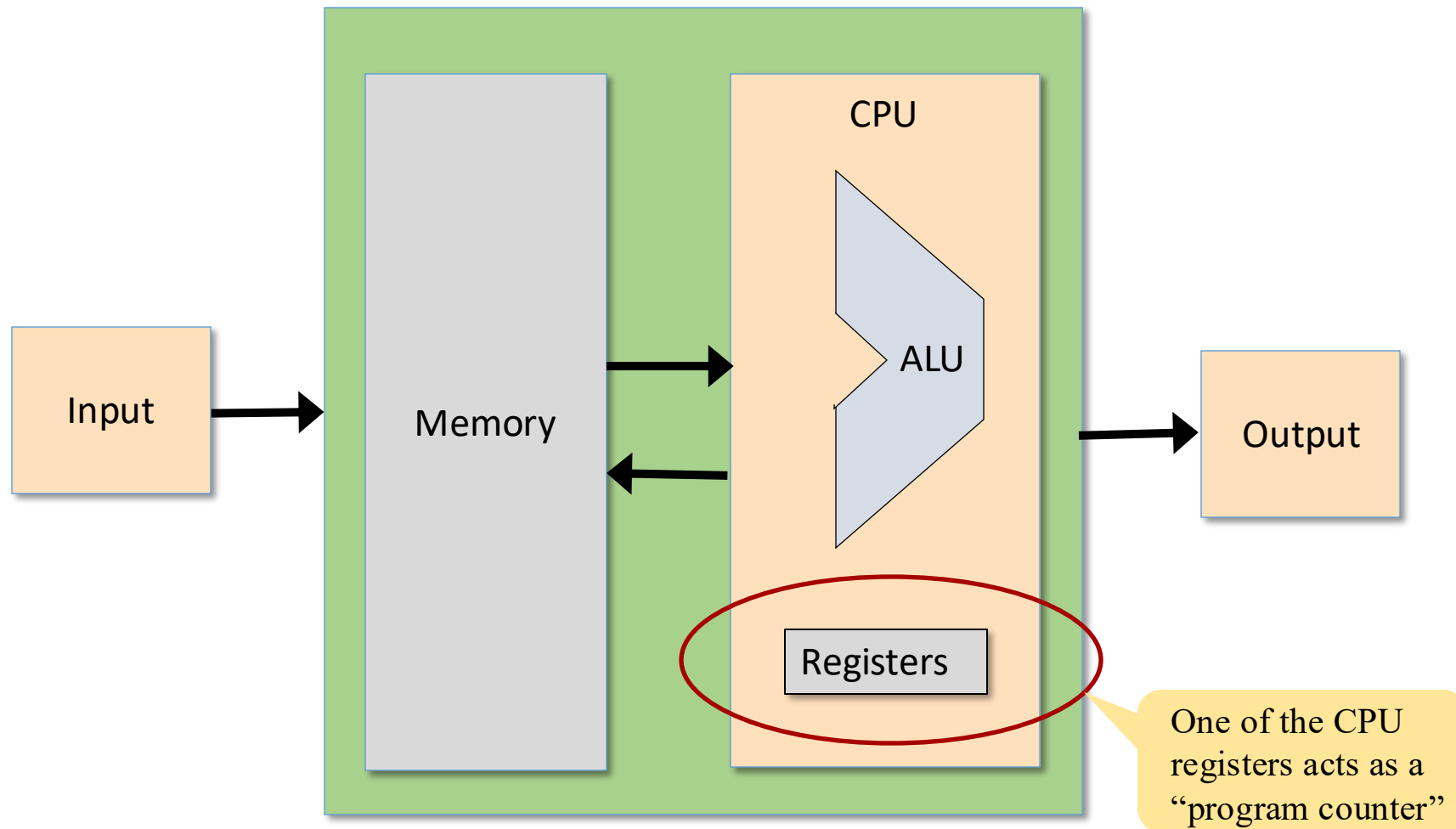
- Registers

- RAM

- PC

# Computer architecture

---



# Counter

---

- The computer needs to keep track of which instruction should be fetched and executed next
- This is done using a register typically called Program Counter
- The PC is used to store the address of the instruction that should be fetched and executed next

# Counter

---

- The computer needs to keep track of which instruction should be fetched and executed next
- This is done using a register typically called Program Counter
- The PC is used to store the address of the instruction that should be fetched and executed next

## Basic PC operations

increment:

PC++

facilitates fetching the next instruction

load:

PC =  $n$

facilitates “jumping to”, and fetching, some instruction

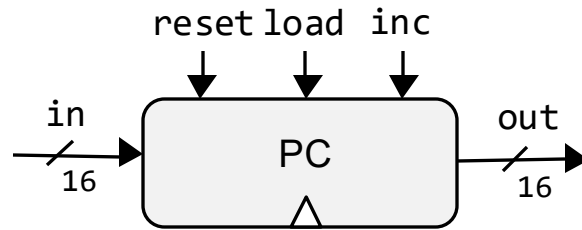
reset:

PC = 0

facilitates fetching the first instruction

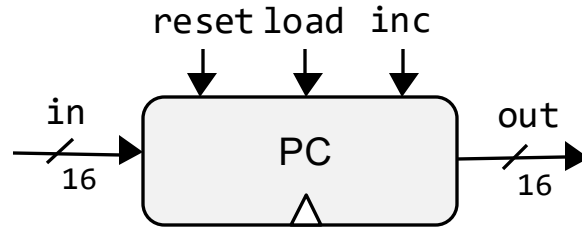
# Counter

---



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

# Counter



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

Usage:

**To read:**  
probe out

**To increment:** assert inc

**To load:** set in to  $v$ , assert load

**To reset:** assert reset



Best practice: When asserting one of the three control bits, set the other two bits to 0

# Recap: Combinational logic / Sequential logic

---



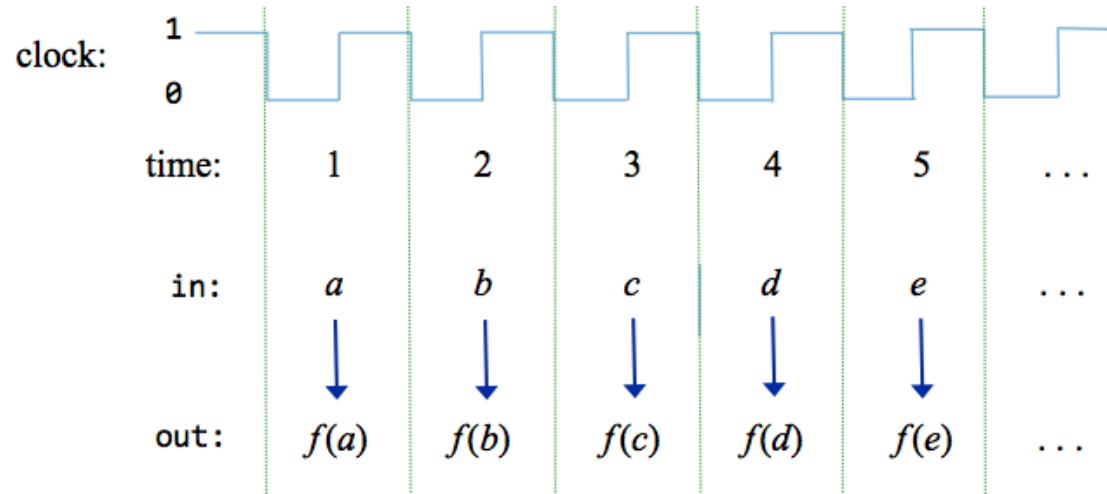
# Recap: Combinational logic / Sequential logic

## Combinational logic

The output depends  
on current inputs only

The clock is used to  
stabilize outputs

Used for building chips  
that do calculations



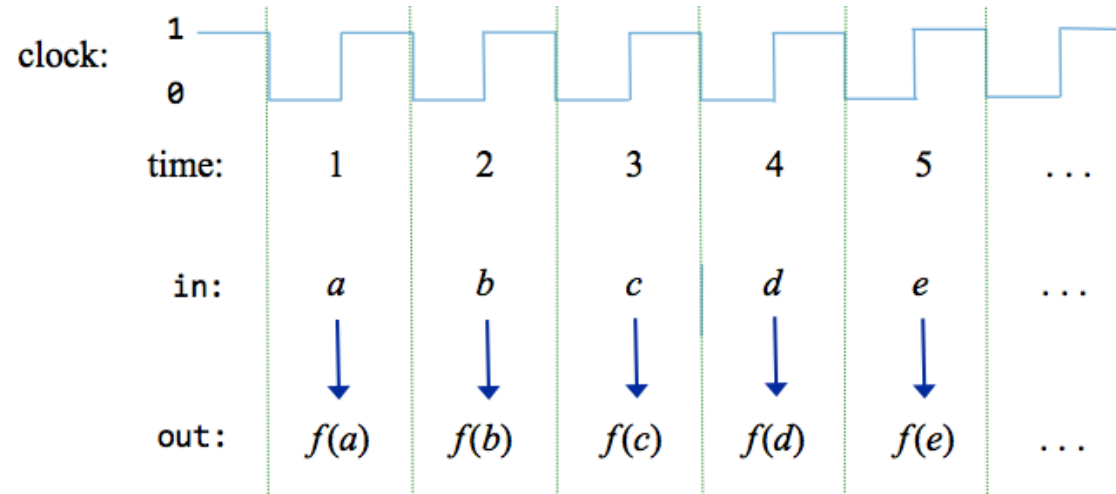
# Recap: Combinational logic / Sequential logic

## Combinational logic

The output depends on current inputs only

The clock is used to stabilize outputs

Used for building chips that do calculations

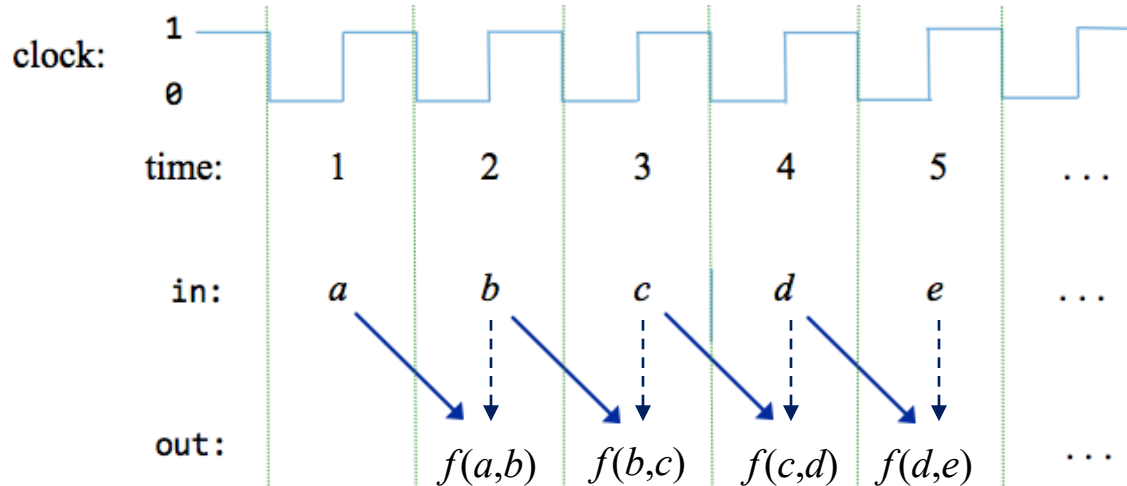


## Sequential logic

The output depends on:

- Previous inputs
- And, optionally, also on current inputs

Used for building memory chips (registers, RAM)



# Chapter 3: Memory

---

## Abstraction

- Representing time
- Clock
- Registers
- RAM
- Counter

## Implementation

- Data Flip Flop
- Registers
- RAM
- PC



# Chapter 3: Memory

---

## Abstraction

- Representing time
- Clock
- Registers
- RAM
- Counter

## Implementation



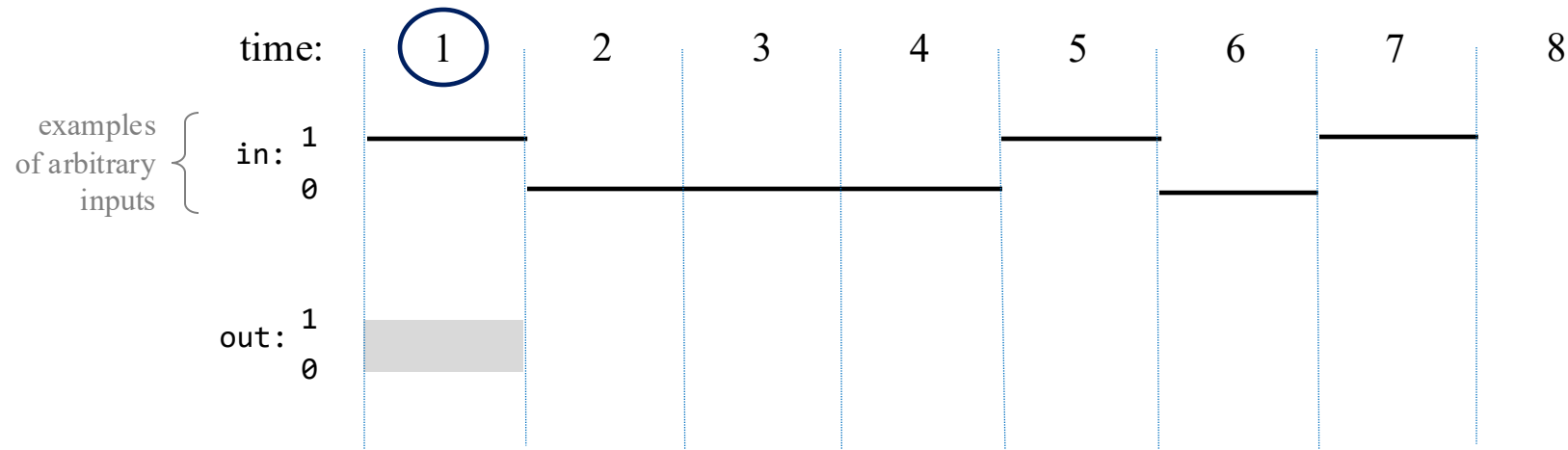
Data Flip Flop

- Registers
- RAM
- PC

# DFF

## Data Flip Flop (aka *latch*)

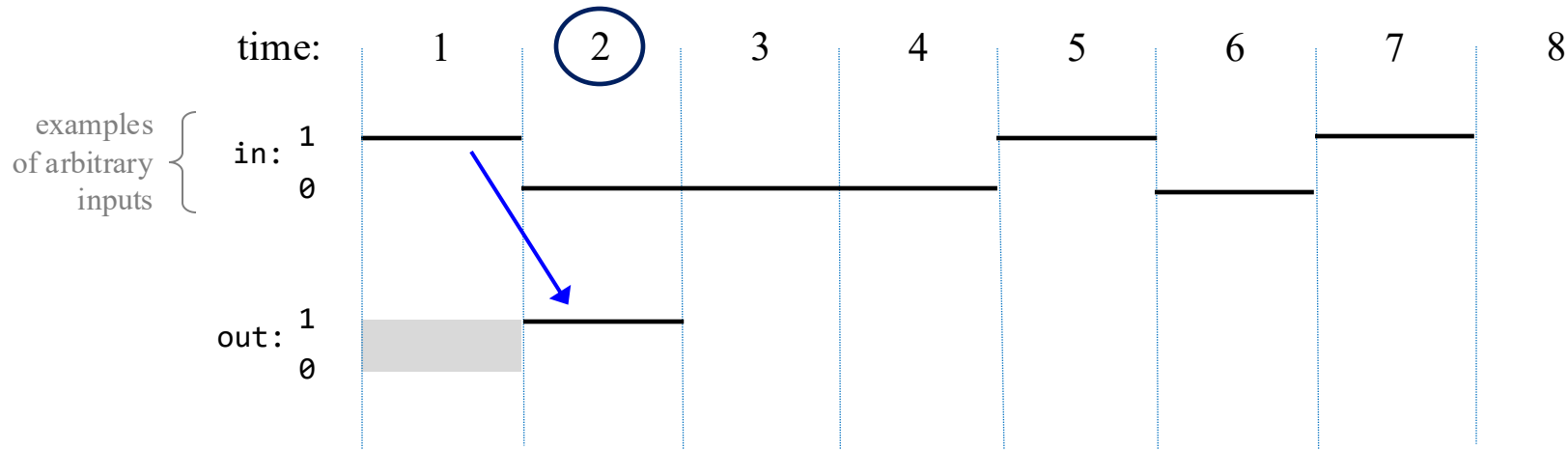
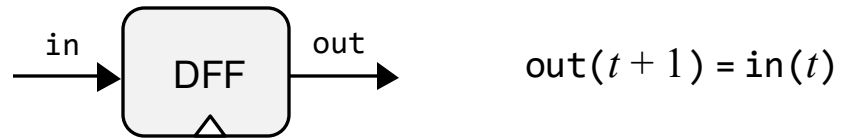
The most elementary sequential gate: Outputs the input in the previous time-step



# DFF

## Data Flip Flop (aka *latch*)

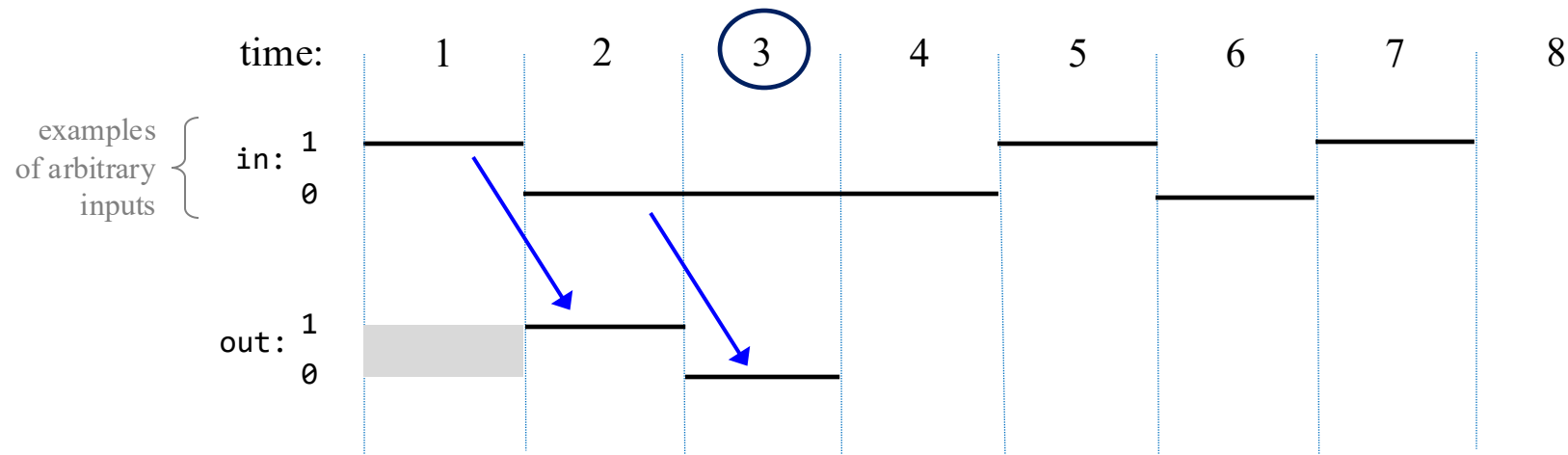
The most elementary sequential gate: Outputs the input in the previous time-step



# DFF

## Data Flip Flop (aka *latch*)

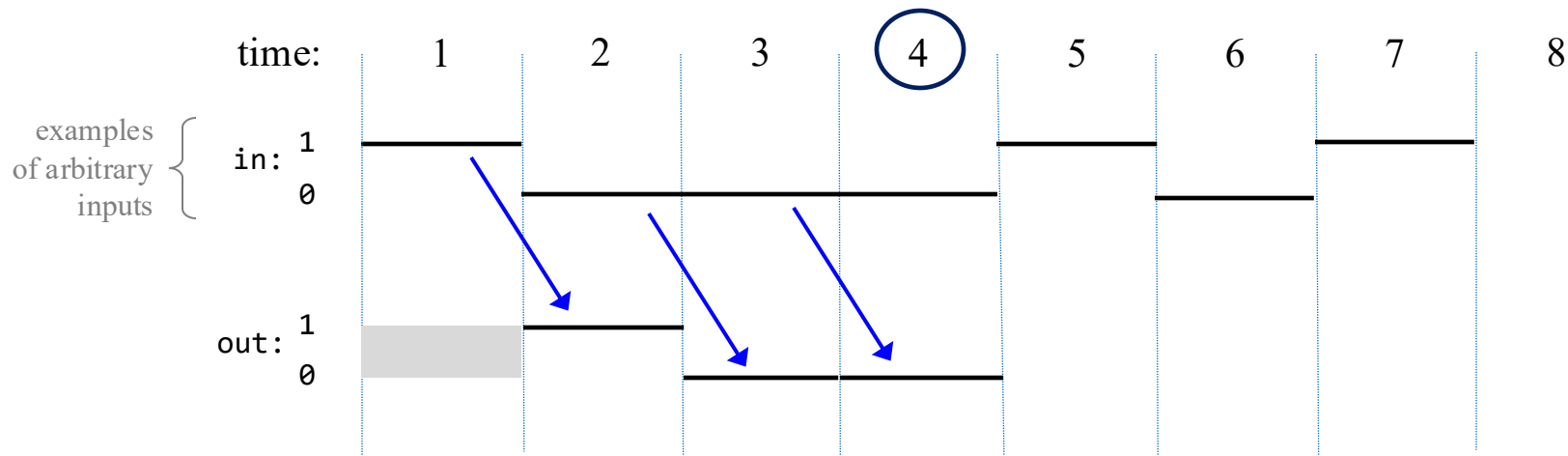
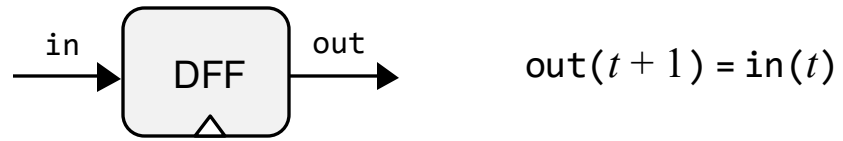
The most elementary sequential gate: Outputs the input in the previous time-step



# DFF

## Data Flip Flop (aka *latch*)

The most elementary sequential gate: Outputs the input in the previous time-step

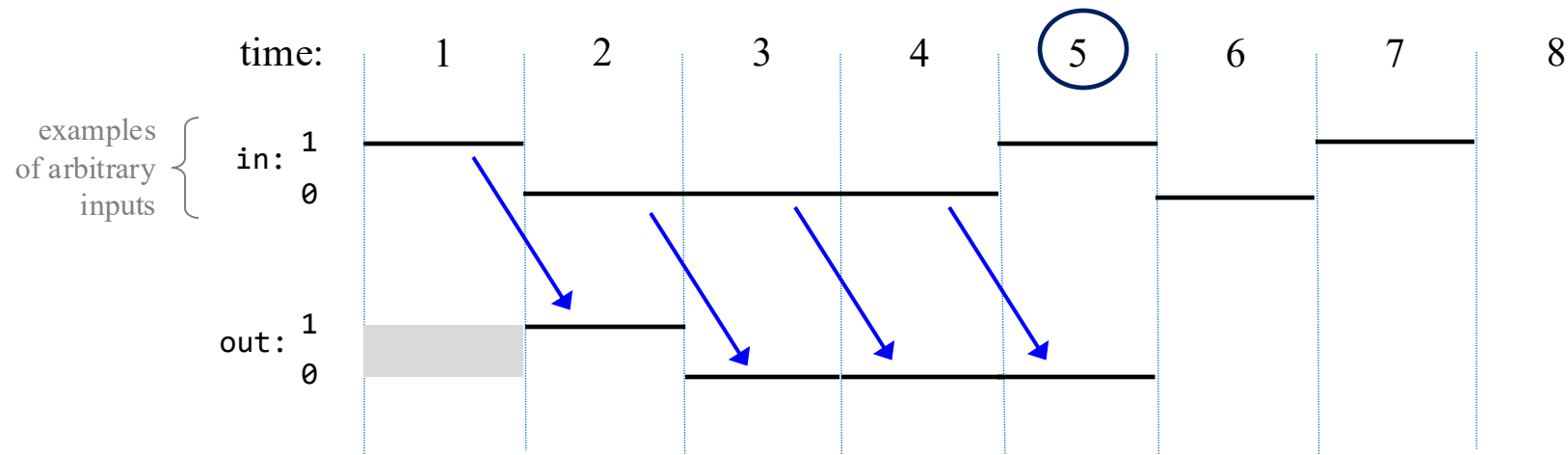
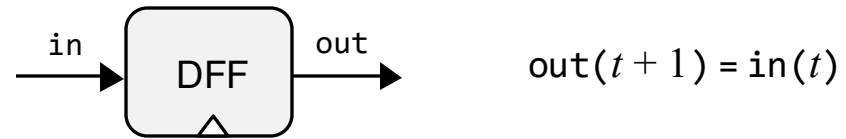




# DFF

## Data Flip Flop (aka *latch*)

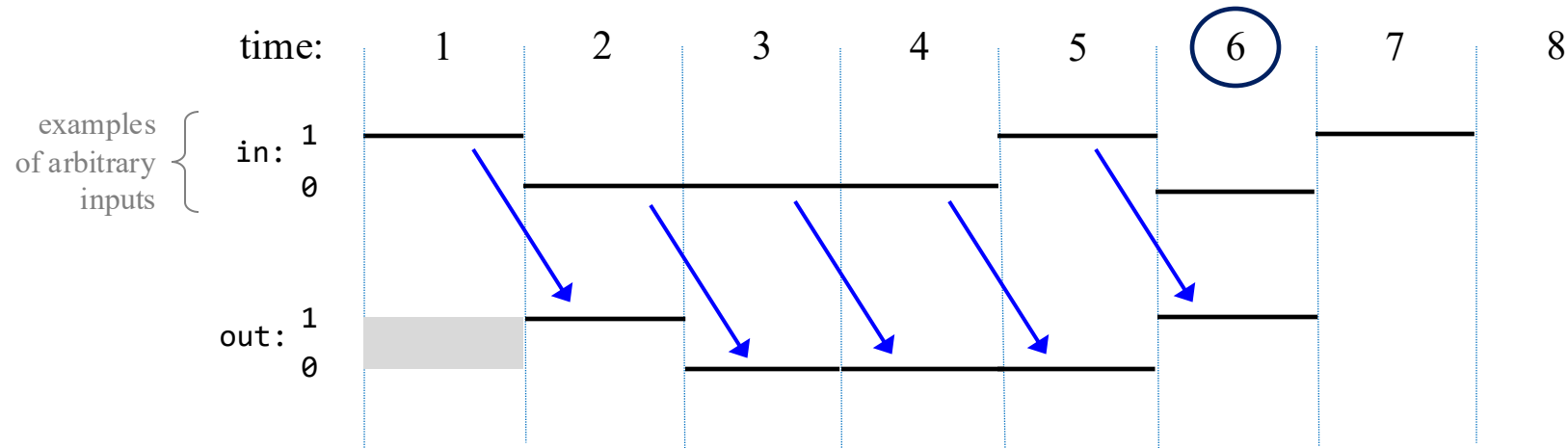
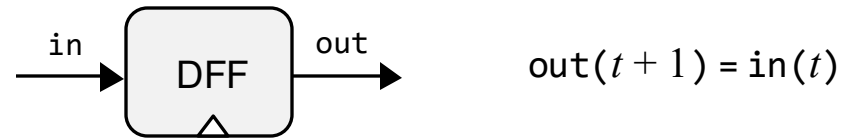
The most elementary sequential gate: Outputs the input in the previous time-step



# DFF

## Data Flip Flop (aka *latch*)

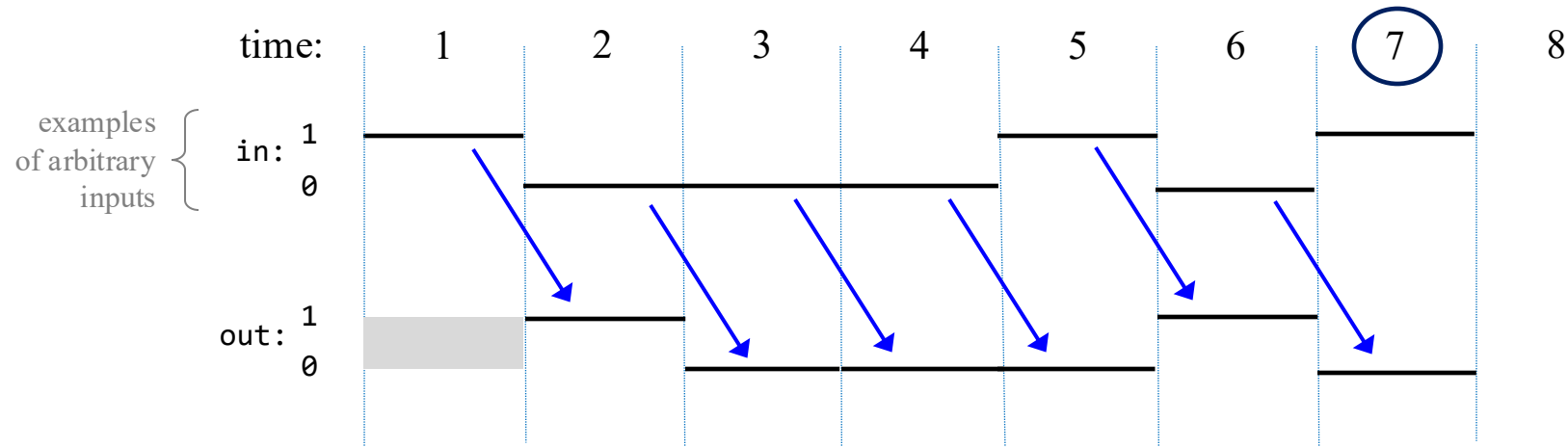
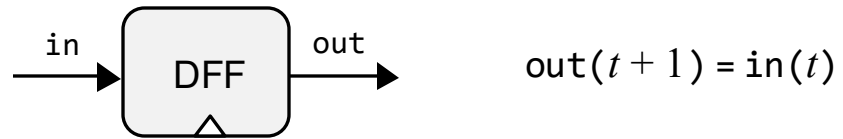
The most elementary sequential gate: Outputs the input in the previous time-step



# DFF

## Data Flip Flop (aka *latch*)

The most elementary sequential gate: Outputs the input in the previous time-step



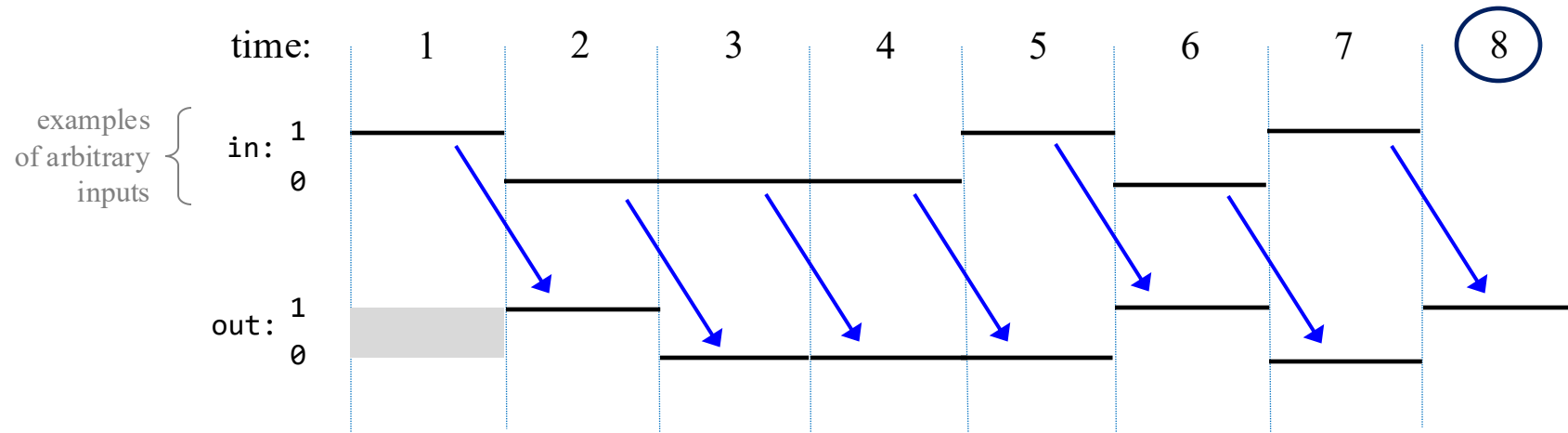
# DFF

## Data Flip Flop (aka *latch*)

The most elementary sequential gate: Outputs the input in the previous time-step



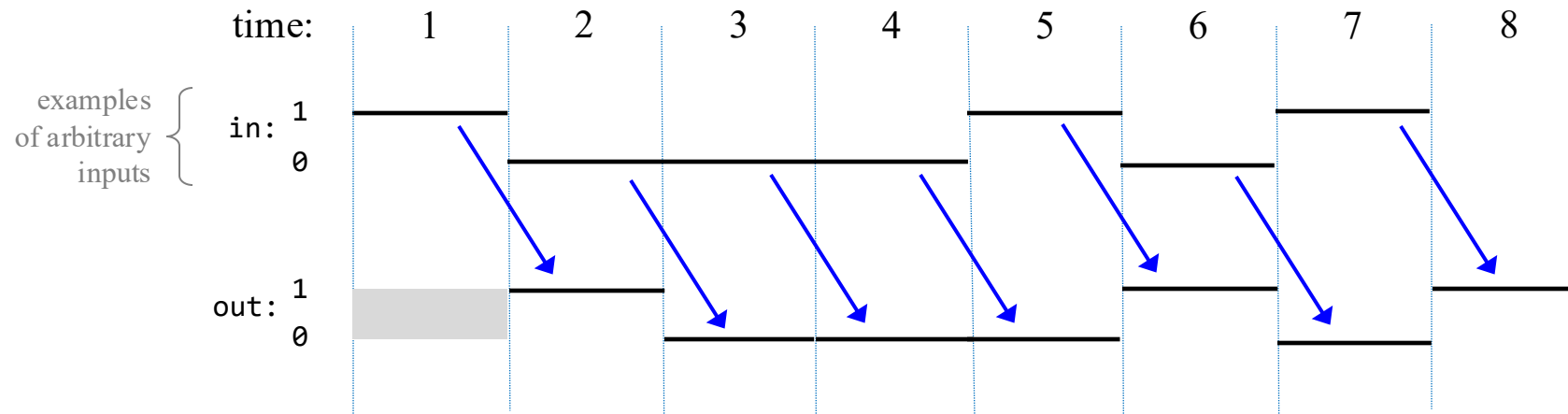
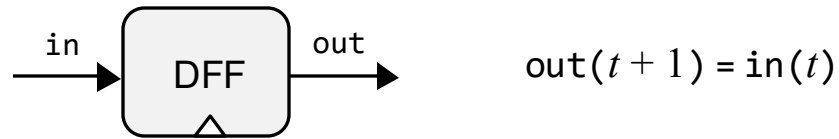
$$\text{out}(t + 1) = \text{in}(t)$$



# DFF

## Data Flip Flop (aka *latch*)

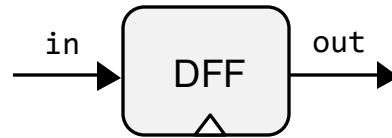
The most elementary sequential gate: Outputs the input in the previous time-step



How can we “load” and then “maintain” a value (0 or 1) over time, without having to re-feed the value in every cycle?

# From DFF to a 1-Bit register

---



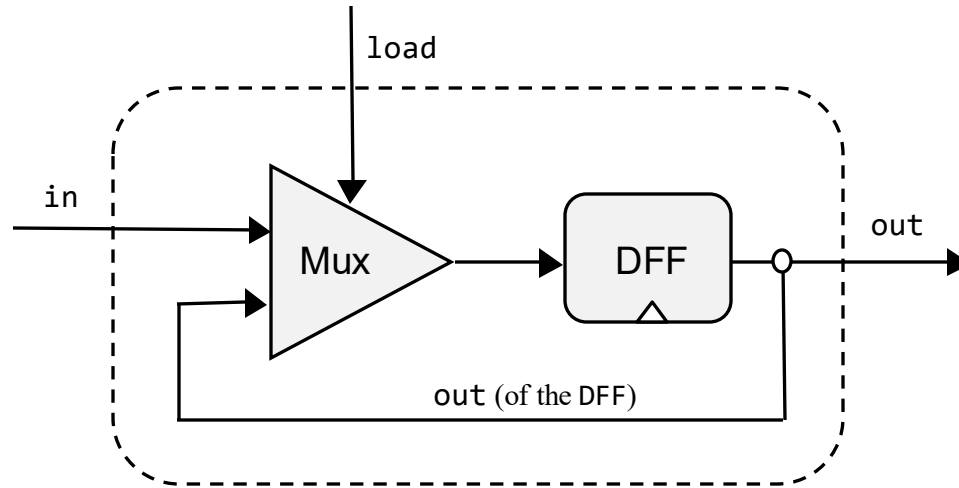
We'll realize a “loading” behavior and a ”storing” behavior, and use a control bit to select between these two states

# From DFF to a 1-Bit register

---

## 1-bit register

Stores one bit  
over time



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

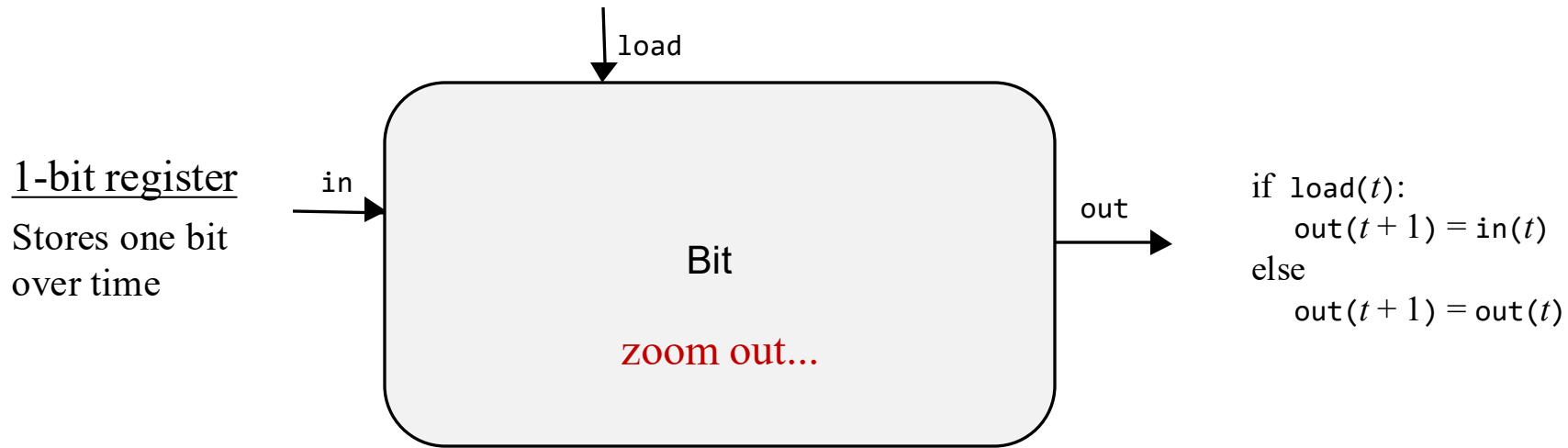
We'll realize a “loading” behavior and a ”storing” behavior,  
and use a control bit to select between these two states

## Behavior

```
if load = 1  the register's value becomes in
else         the register maintains its current value
```

# Register

---



## Behavior

if  $\text{load} = 1$  the register's value becomes in  
else the register maintains its current value

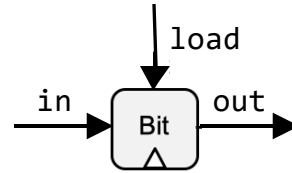


# Register

---

## 1-bit register

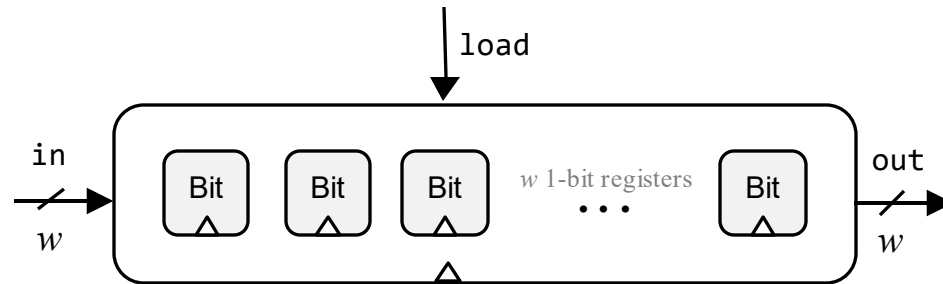
Stores one bit  
over time



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

## $w$ -bit register

Stores  $w$  bits  
over time



Exactly the  
same behavior

## Behavior

if load = 1    the register's value becomes in  
else            the register maintains its current value

# Chapter 3: Memory

---

## Abstraction

- Representing time
- Clock
- Registers
- RAM
- Counter

## Implementation

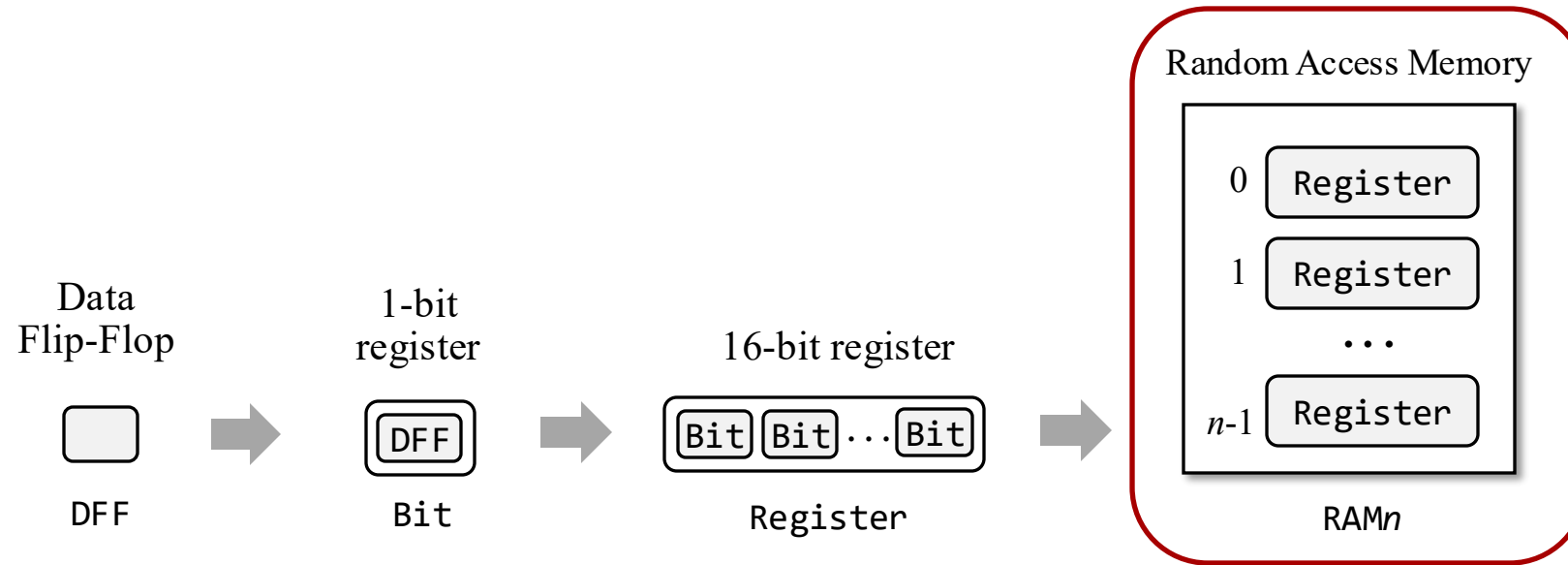
✓ Data Flip Flop

✓ Registers

➡ RAM

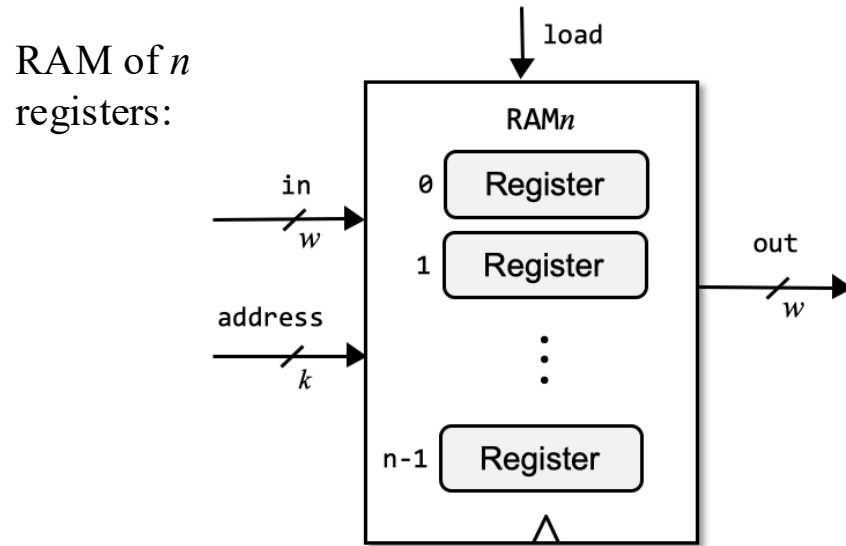
- PC

# From DFF to RAM



# RAM: Abstraction

---

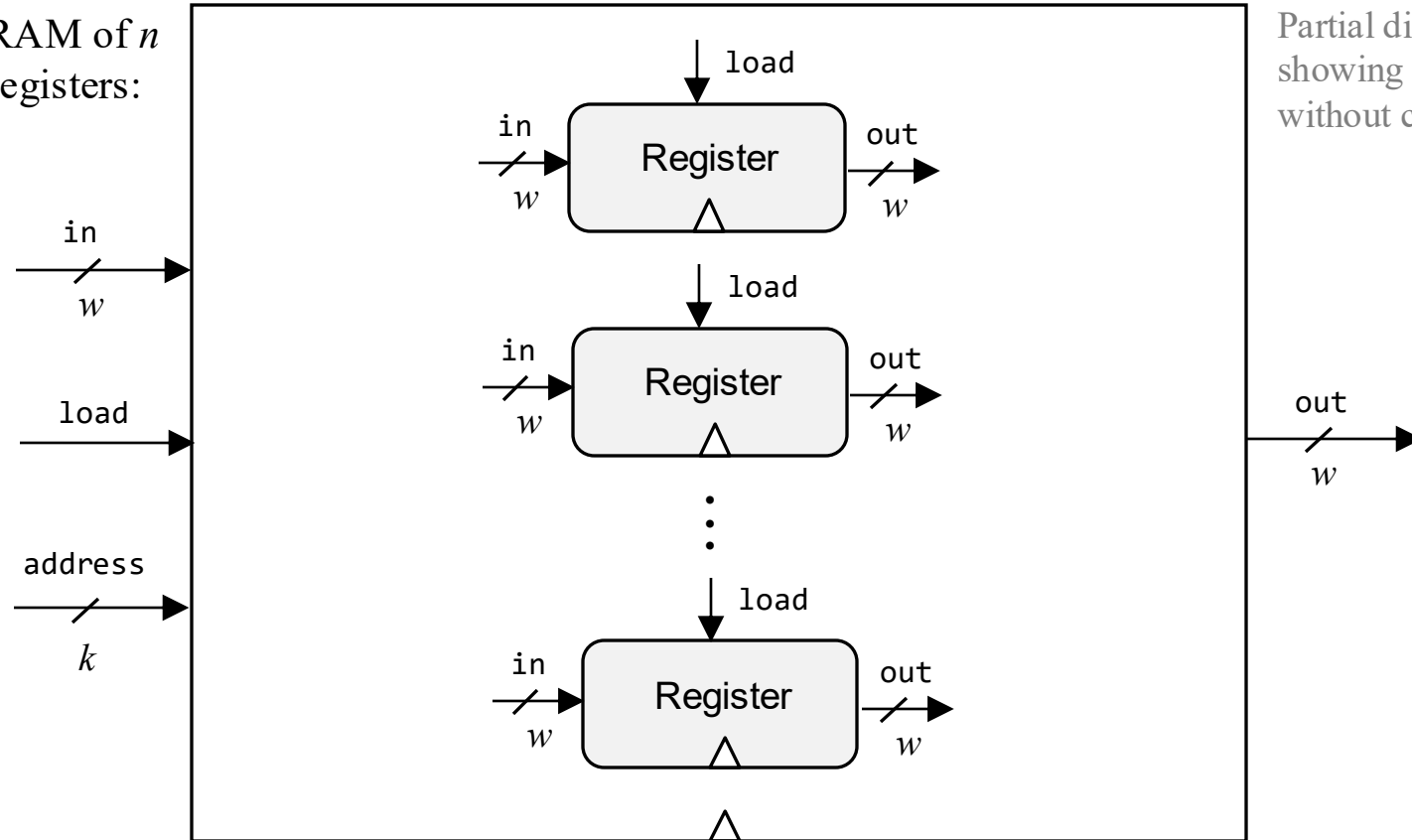


Usage:    **To read register  $i$  :**  
              set address =  $i$ ,  
              probe out (out always emits the state of  $RAM[i]$ )

**To write  $v$  in register  $i$  :**  
              set address =  $i$ ,  
              set in =  $v$ ,                    Result:  $RAM[i] \leftarrow v$   
              set load = 1                    From the next time-step onward, out emits  $v$

# RAM: Implementation

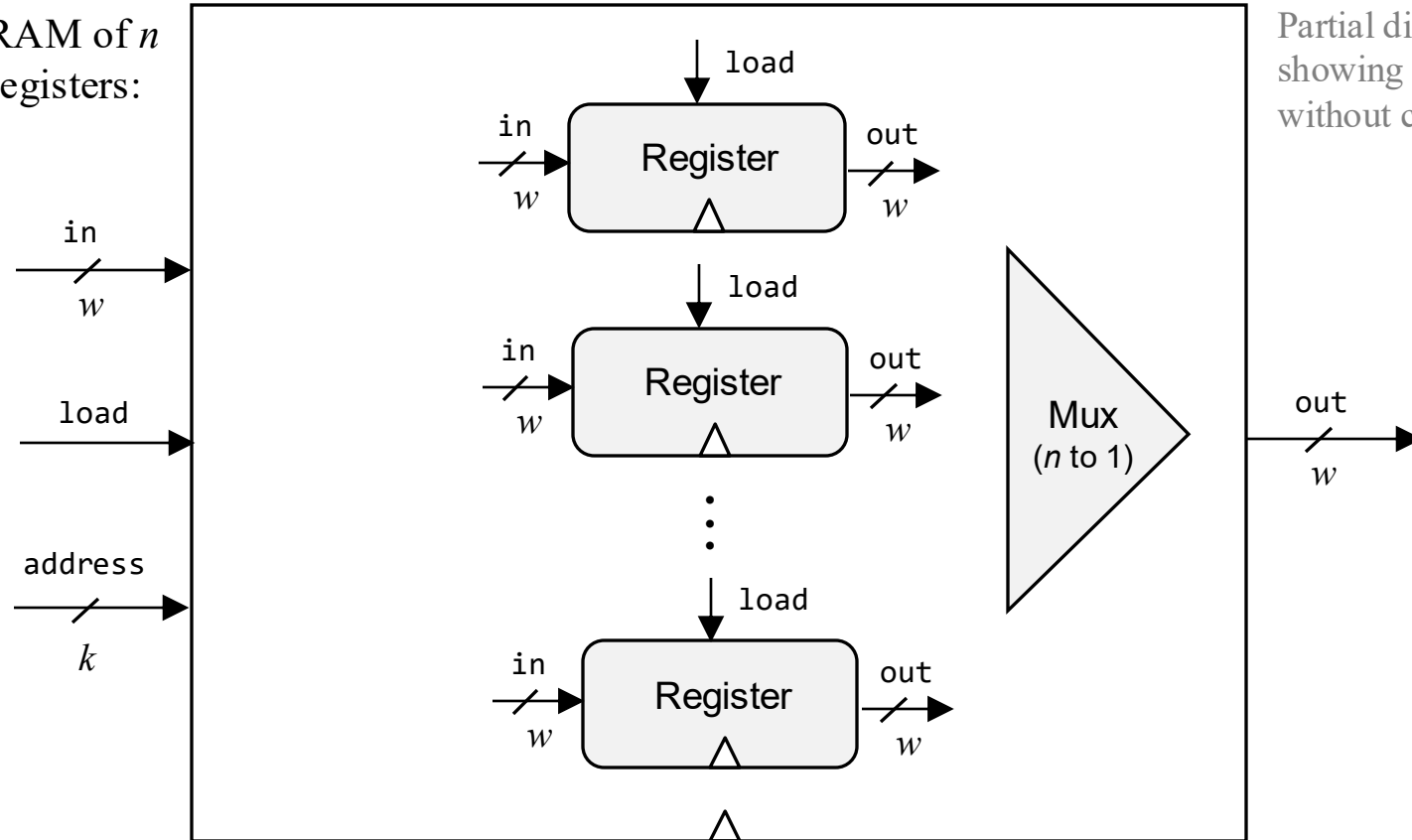
RAM of  $n$  registers:



Partial diagram,  
showing chip-parts  
without connections

# RAM: Implementation

RAM of  $n$  registers:

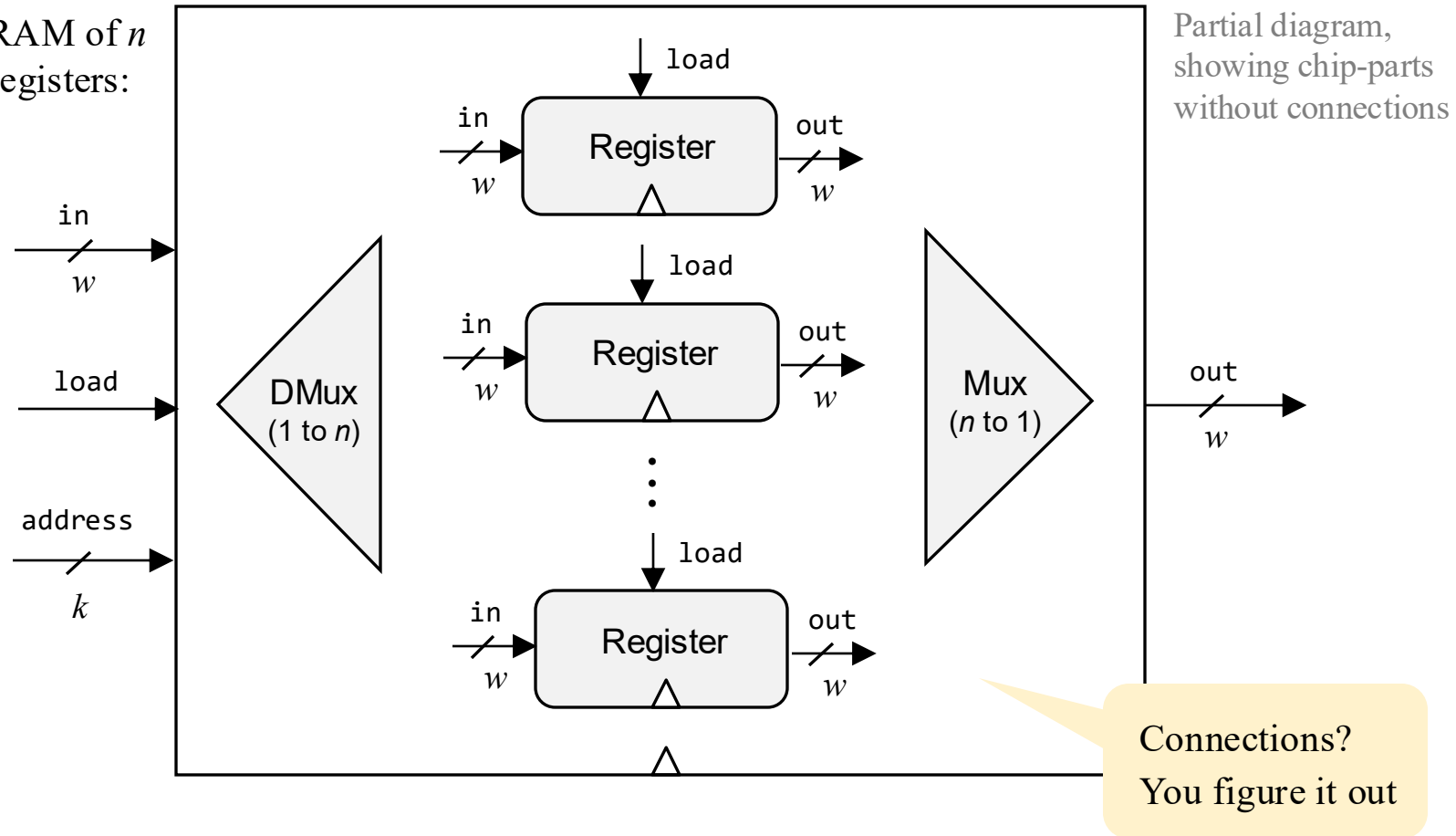


Partial diagram,  
showing chip-parts  
without connections

Reading: Can be realized using a Mux

# RAM: Implementation

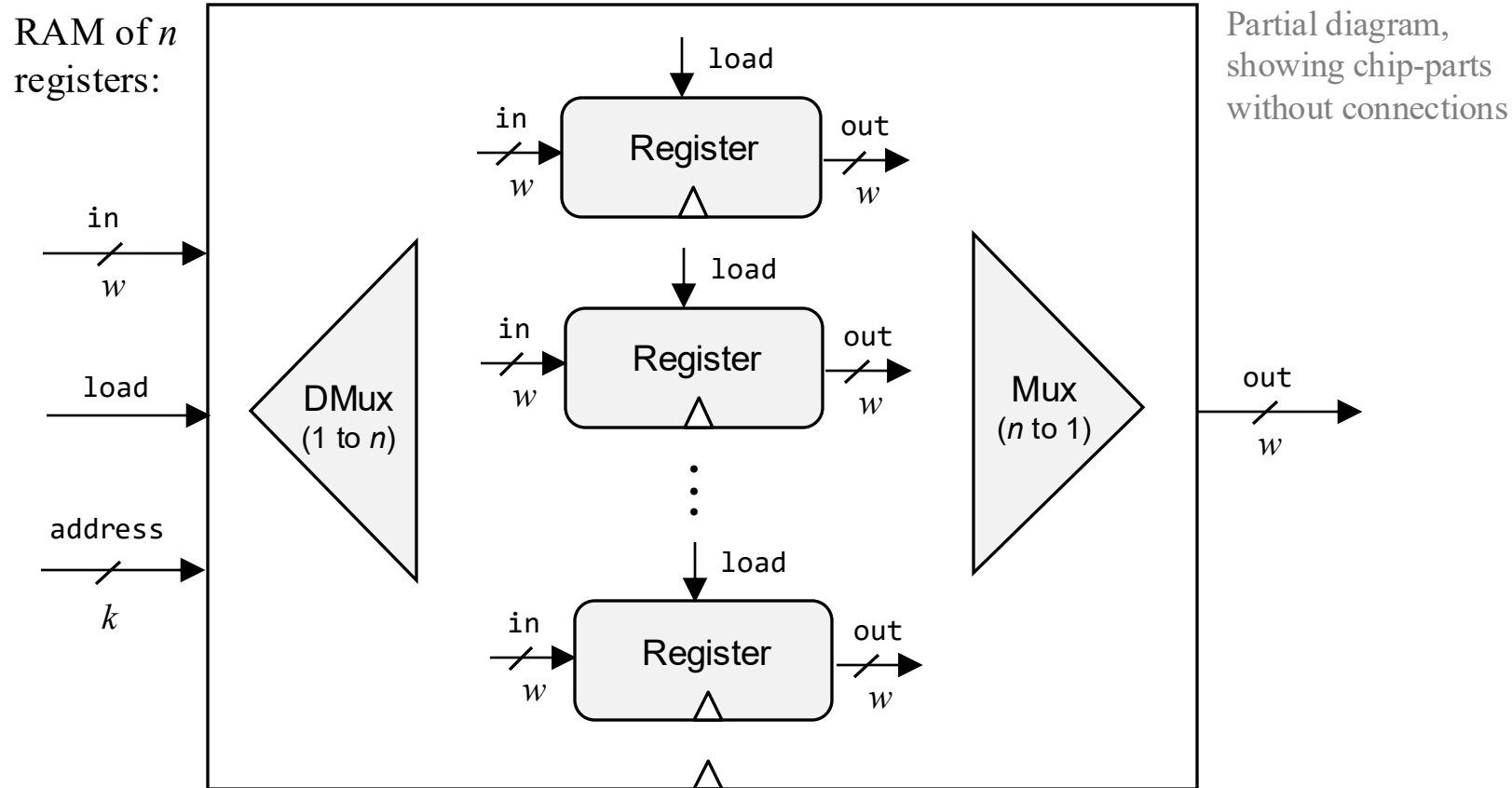
RAM of  $n$  registers:



Reading: Can be realized using a Mux

Writing: Can be realized using a DMux

# RAM: Implementation



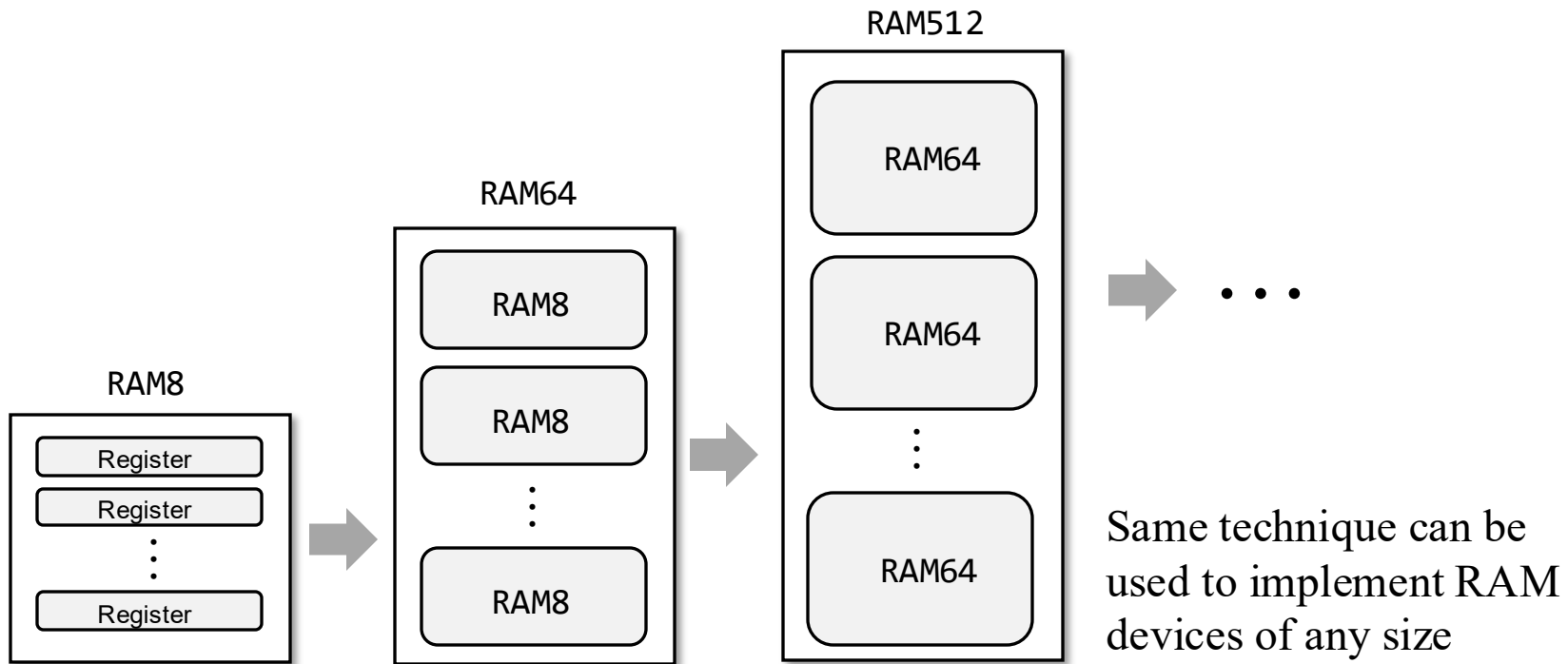
## How the magic of *direct access* works

- The storage behavior is based on *sequential* logic
- But, the *addressing* logic is realized by the Mux / DMux chips, which are *combinational*.

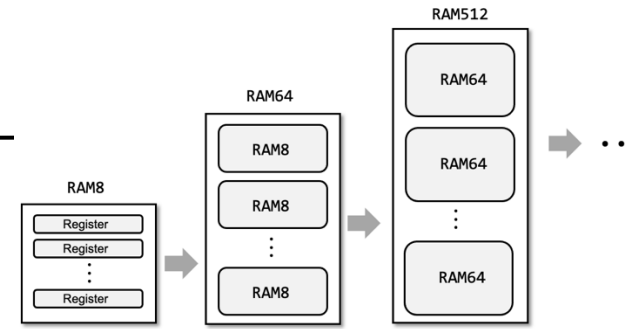
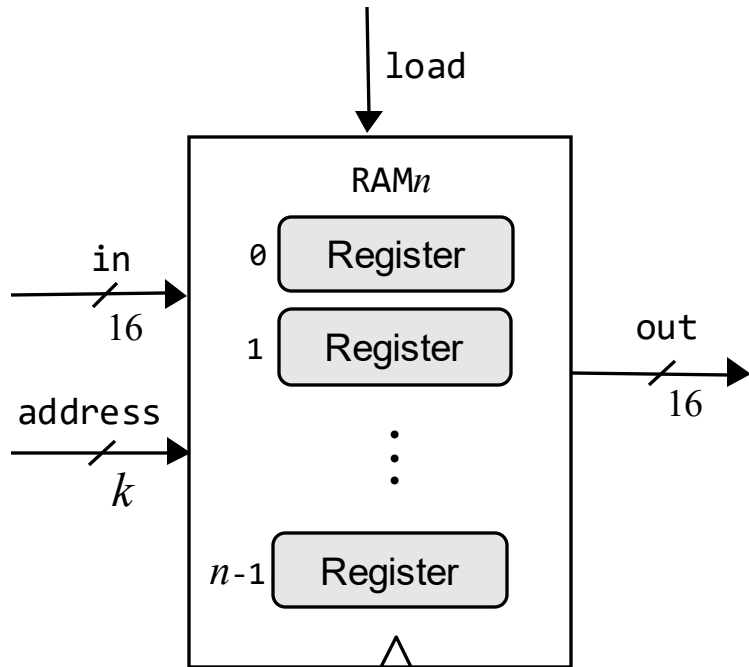


# RAM: Implementation

---



# RAM: Implementation



A family of 16-bit RAM chips:

chip name	$n$	$k$
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

Why stop at RAM16K?

Because that's what we need for building the Hack computer.

# Chapter 3: Memory

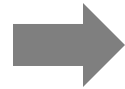
---

## Abstraction

- Representing time
- Clock
- Registers
- RAM
- Counter

## Implementation

- Data Flip Flop
- Registers
- RAM
- PC



Project 3

# Project 3

---

## Given:

- The chips built in projects 1 and 2



Data Flip-Flop (built-in DFF gate)

## Build

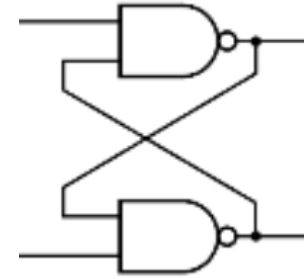
- Bit
- Register
- PC
- RAM8
- RAM64
- RAM512
- RAM4K
- RAM16K

# DFF implementation

---

- A DFF gate can be built by connecting Nand gates using feedback loops
- The resulting implementation is elegant, yet intricate
- Instead, we'll use a built-in DFF implementation:

```
/** Data Flip-flop: out(t + 1) = in(t)
 *  where t is the current time unit. */
CHIP DFF {
    IN in;
    OUT out;
    BUILTIN DFF;
    CLOCKED in;
}
```



Part of a possible  
DFF implementation

# Project 3

---

## Given:

- The chips built in projects 1 and 2
- Data Flip-Flop (built-in DFF gate)

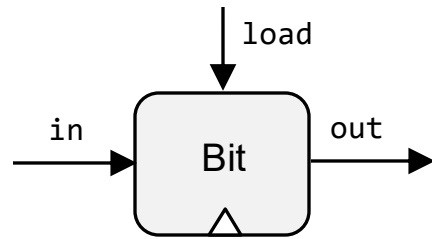
## Build:



Bit

- Register
- PC
- RAM8
- RAM64
- RAM512
- RAM4K
- RAM16K

# 1-Bit register

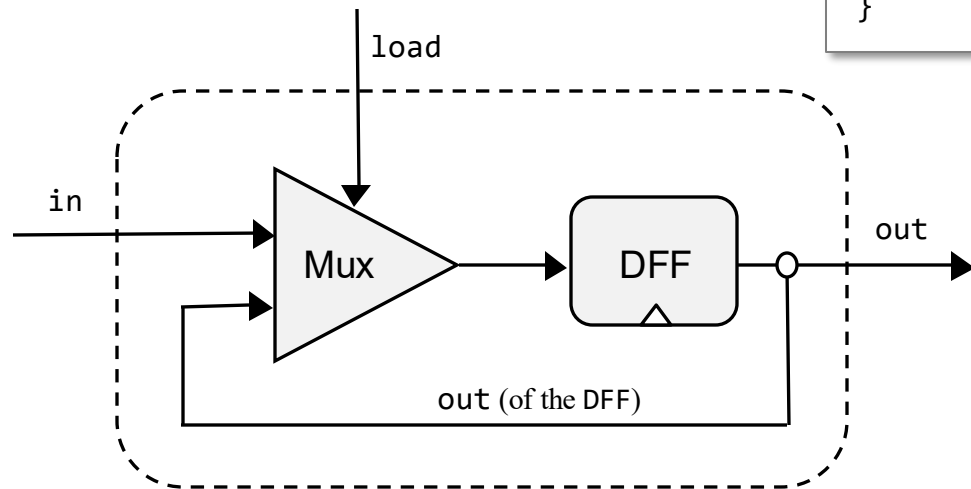


Bit abstraction

Bit.hdl

```
/** 1-bit register:
    if load(t): out(t + 1) = in(t)
    else       out(t + 1) = out(t)
*/
CHIP Bit {
    IN in, load;
    OUT out;

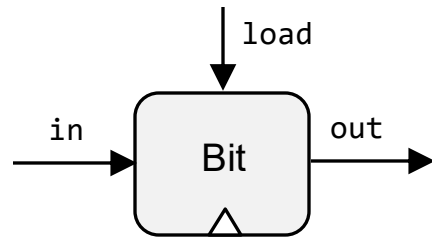
    PARTS:
        //// Replace with your code
}
```



Bit implementation

Implementation tip  
Realize the diagram

# 1-Bit register

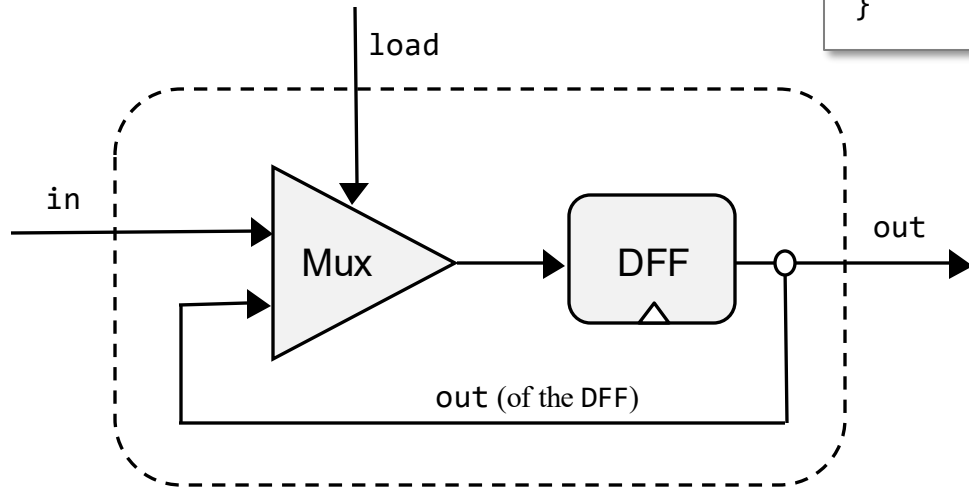


Bit abstraction

Bit.hdl

```
/** 1-bit register:
    if load(t): out(t+1) = in(t)
    else       out(t+1) = out(t)
*/
CHIP Bit {
    IN in, load;
    OUT out;

    PARTS:
        //// Replace with your code
}
```



Bit implementation

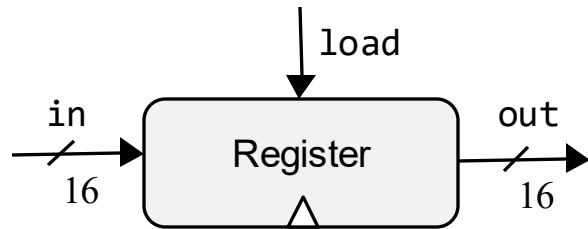
Note (about memory chips, in general)

The only memory chip that uses a DFF directly is Bit;

All the other memory chips are built on top of Bit.



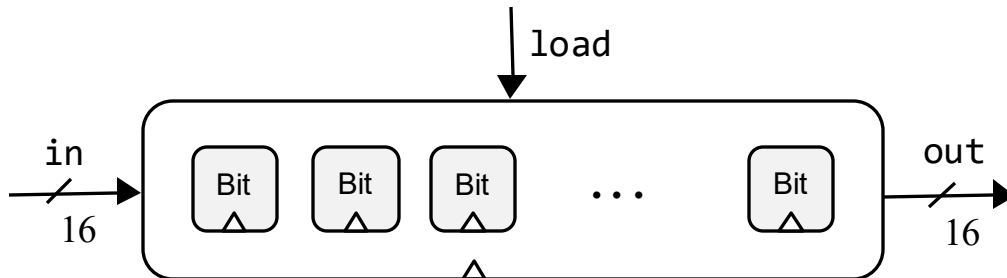
# 16-bit Register



Register.hdl

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

    PARTS:
        //// Replace with your code
}
```



Partial diagram, showing chip-parts without connections

Implementation tip  
Realize the diagram

# Project 3

---

## Given

- All the chips built in projects 1 and 2
- Data Flip-Flop (built-in DFF gate)

## Build the following chips

✓ Bit

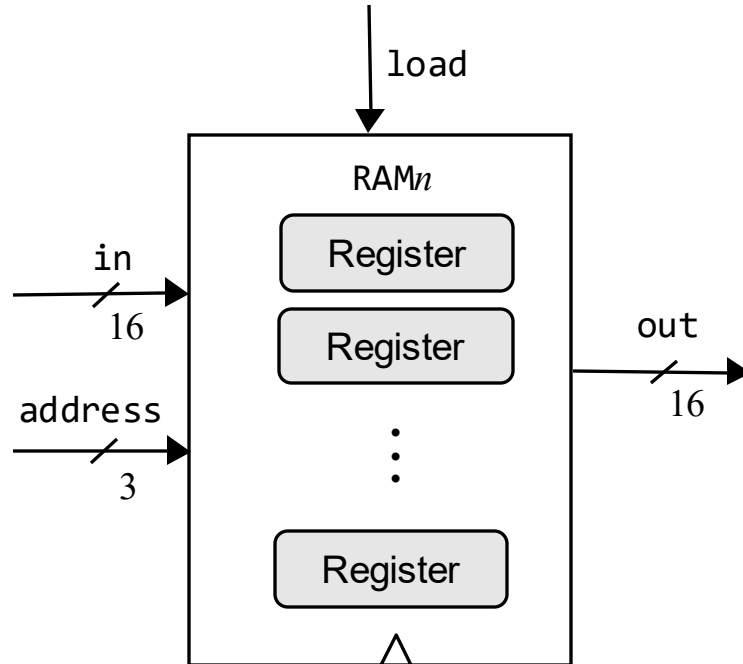
✓ Register

✓ PC

➡ RAM8

- RAM64
- RAM512
- RAM4K
- RAM16K

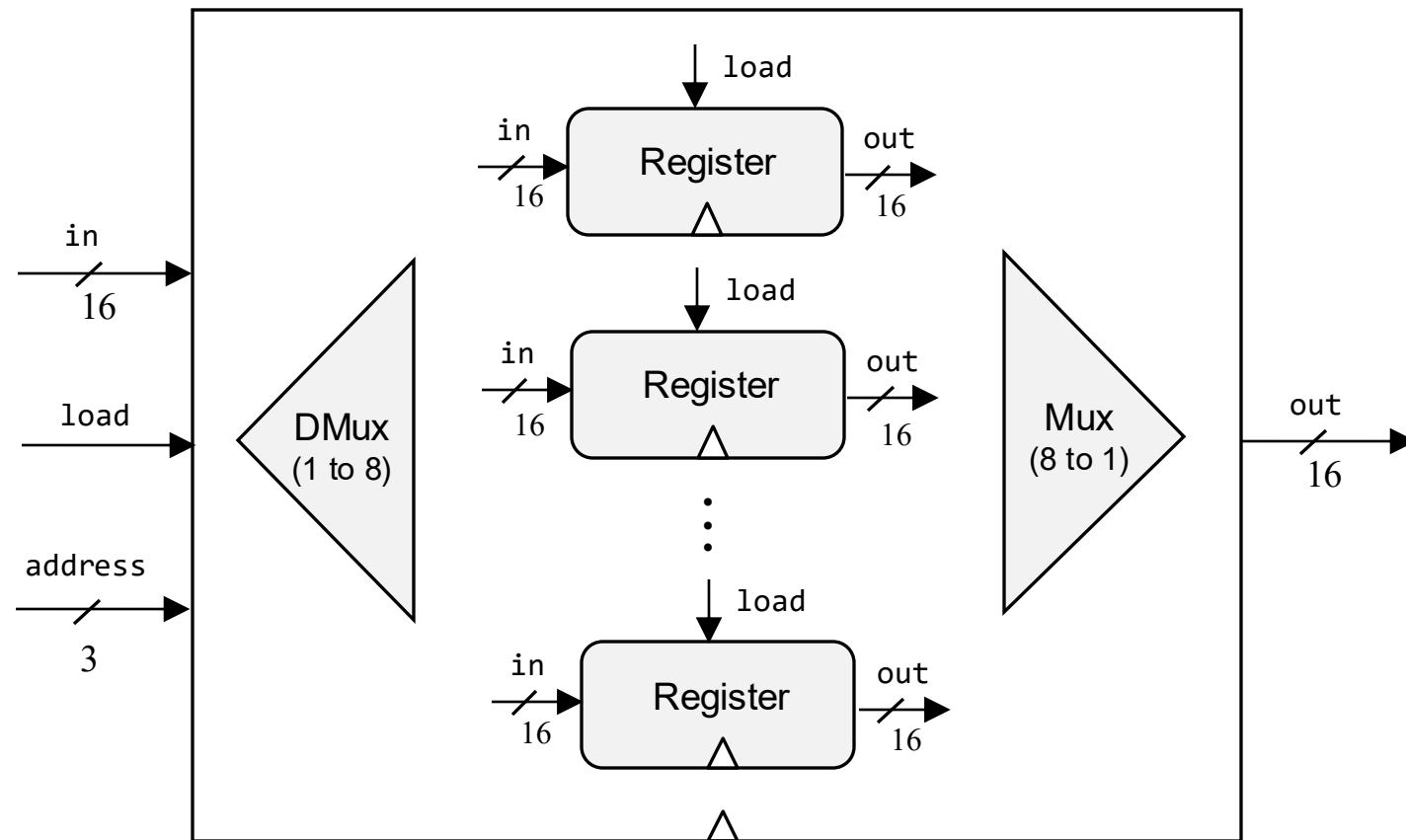
# 8-Register RAM



RAM8.hdl

```
/* Memory of 8 registers, each 16 bit-wide.
out holds the value stored at the memory location
specified by address. If load=1, then the in value
is loaded into the memory location specified by
address (the loaded value will appear in out from
the next time step onward).
*/
CHIP RAM8 {
    IN in[16], load, address[3];
    OUT out[16];
    PARTS:
        //// Replace with your code
}
```

# 8-Register RAM



Partial diagram, showing some of the chip-parts, without connections

## Implementation tip

Figure out the missing connections, and realize the diagram.

# Project 3

---

## Given

- All the chips built in projects 1 and 2
- Data Flip-Flop (built-in DFF gate)

## Build the following chips

✓ Bit

✓ Register

✓ PC

✓ RAM8

• RAM64

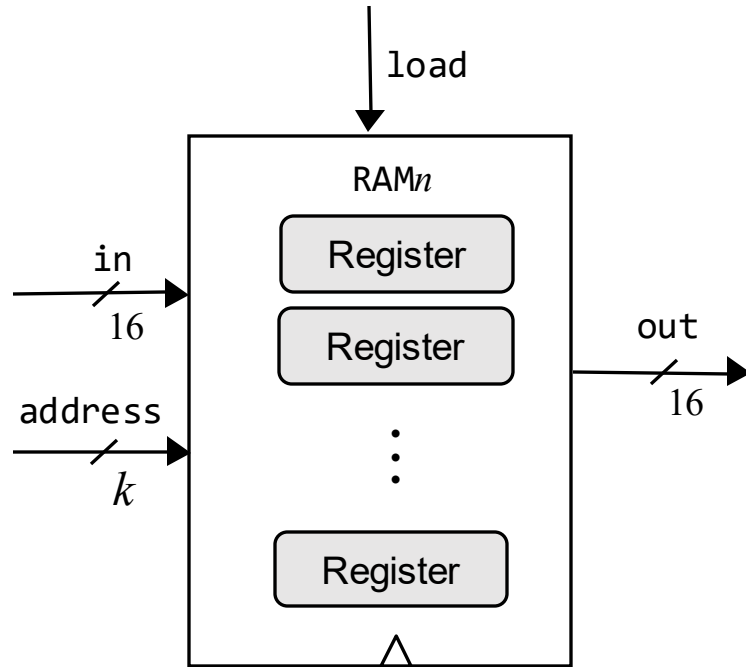
• RAM512

• RAM4K

• RAM16K

} A family of RAM chips

# $n$ -Register RAM

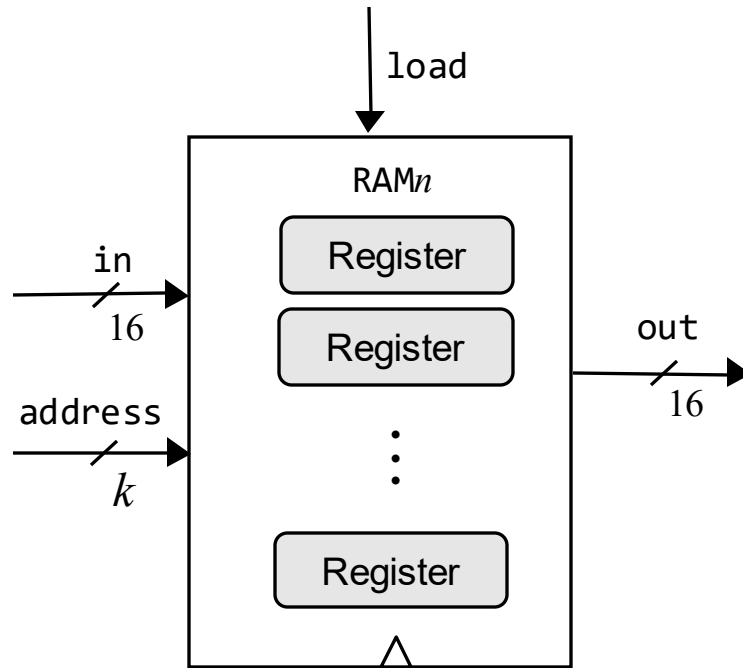


$RAM_n.hdl$

```
/* Memory of  $n$  registers, each 16 bit-wide.
out holds the value stored at the memory location
specified by address. If load=1, then the in value
is loaded into the memory location specified by
address (the loaded value will appear in out from
the next time step onward).
*/

CHIP  $RAM_n$  {
    IN in[16], load, address[k];
    OUT out[16];
    PARTS:
        //// Replace with your code
}
```

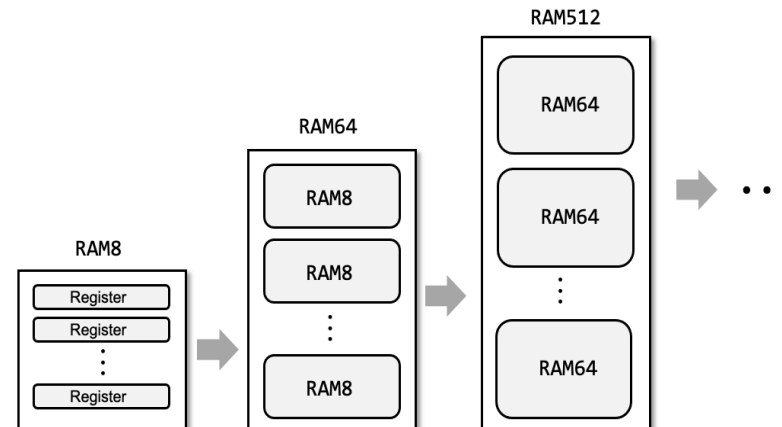
# $n$ -Register RAM



chip name	$n$	$k$
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

## Implementation tips

- Think about the address input as consisting of two fields:
  - One field selects a RAM-part;
  - The other field selects a register within that RAM-part
- Use logic gates to effect this addressing scheme



# Chapter 3: Memory

---

## Abstraction

- Representing time
- Clock
- Registers
- RAM
- Counter

## Implementation

✓ Data Flip Flop

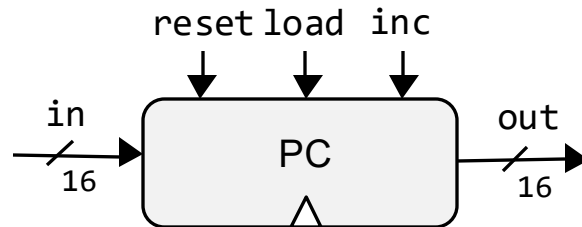
✓ Registers

✓ RAM

➡ PC



# 16-bit Counter



PC.hdl

```
/**
 16-bit counter:
  if      reset(t): out(t+1) = 0
  else if load(t):  out(t+1) = in(t)
  else if inc(t):   out(t+1) = out(t) + 1
  else              out(t+1) = out(t)
 */
CHIP PC {
  IN in[16], reset, load, inc;
  OUT out[16];

  PARTS:
    //// Replace with your code
}
```

## Implementation notes

- Can be built using Register, Inc16, and Mux16 chip-parts
- Tip: The output of a chip-part can simultaneously feed several destinations;  
Arbitrary example: And (a=a, b=b, out=d1, out=d2);
- Recommendation:  
Start by building a basic PC chip with one control bit: reset;  
Next, extend your design and HDL code to handle the load and inc bits.

# Chapter 3: Memory

---

## Abstraction

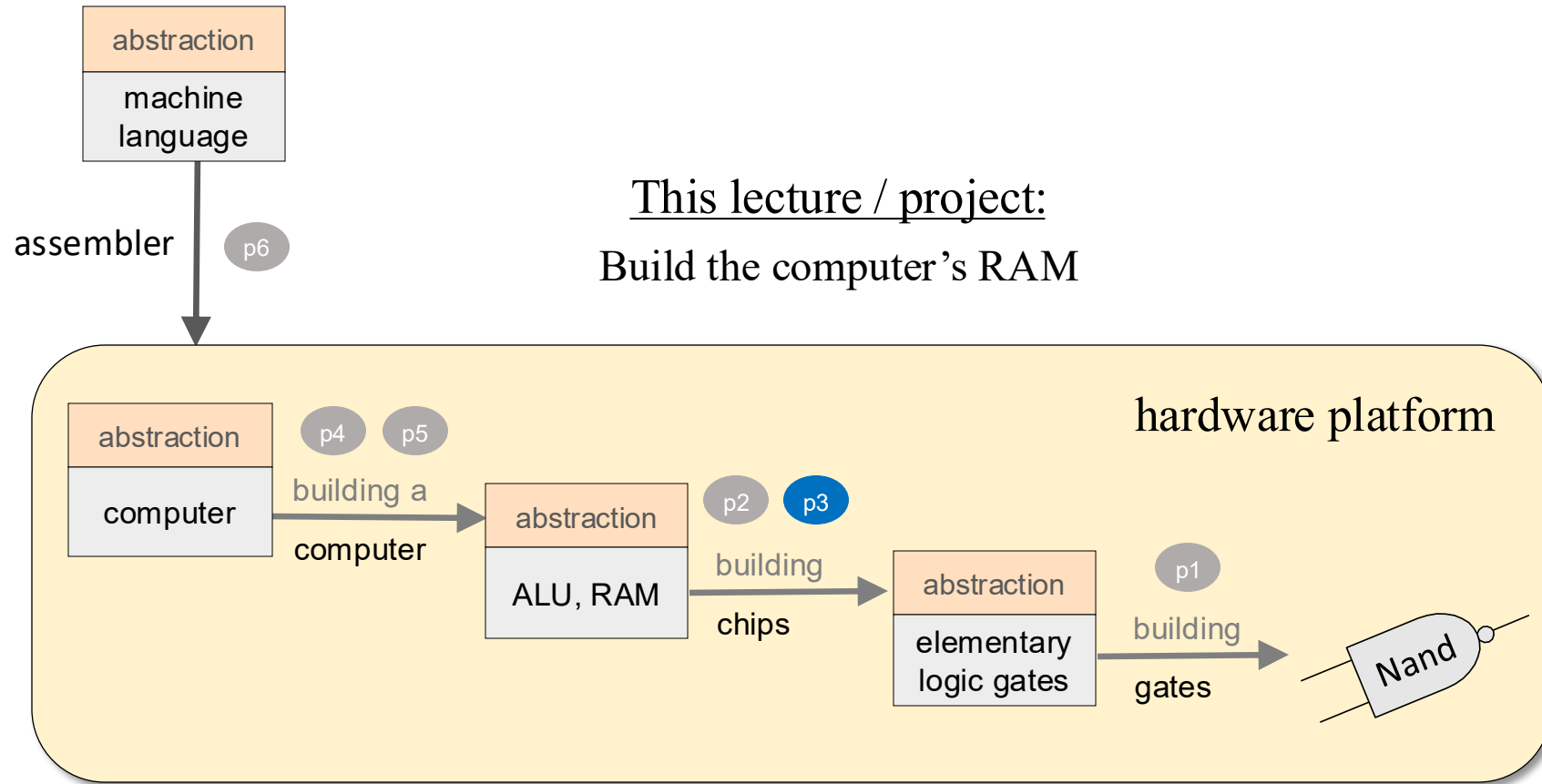
- Representing time
- Clock
- Registers
- RAM
- Counter

## Implementation

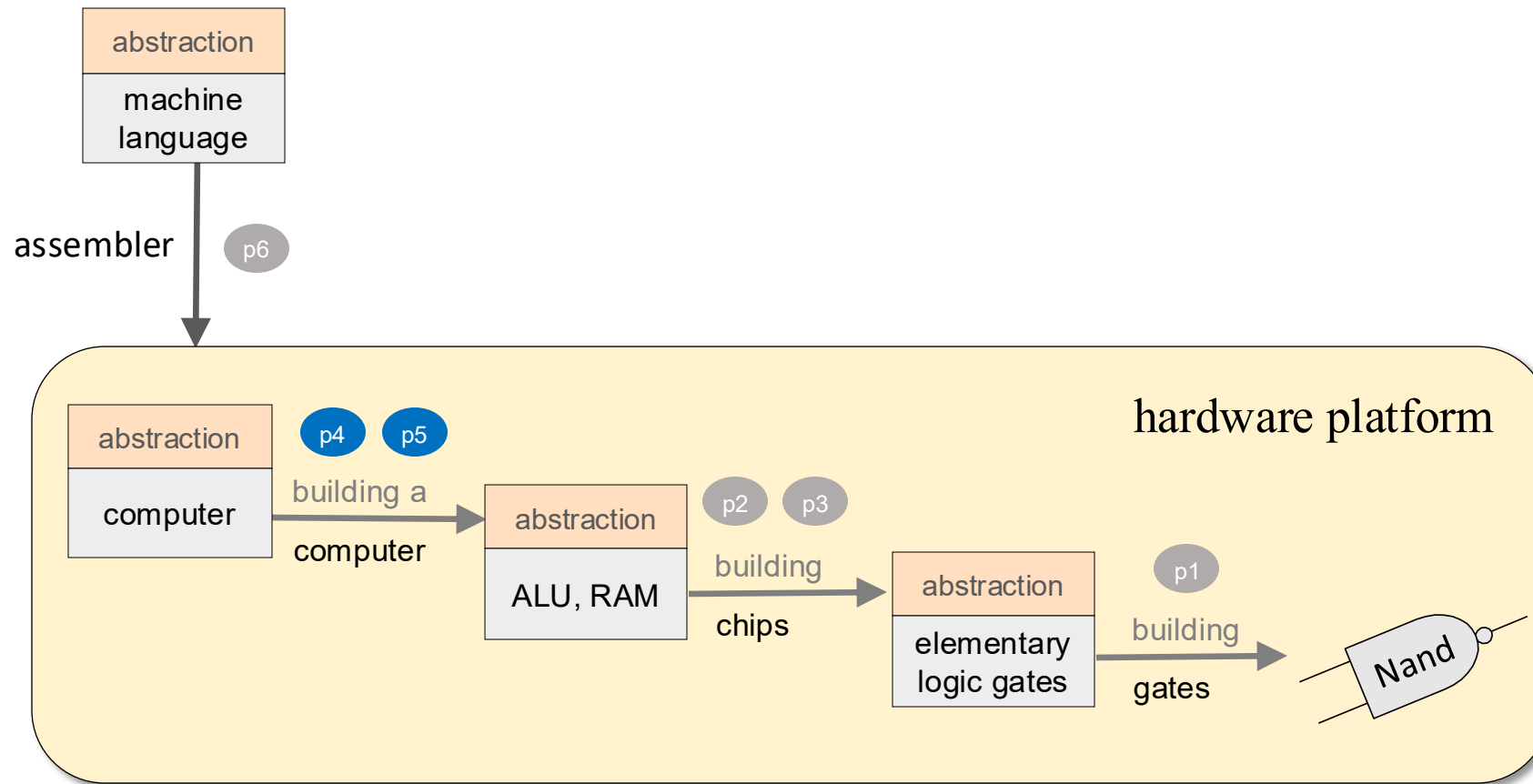
- ✓ Data Flip Flop
- ✓ Registers
- ✓ RAM
- ✓ PC

That's It!  
Go Do Project 3!

# Nand to Tetris Roadmap: Hardware



# Nand to Tetris Roadmap: Hardware



Next two lectures / projects:

We'll build the computer (p5)

But first, we'll get acquainted with the computer's machine language (p4).