Chapter 3

Memory

These slides support chapter 3 of the book

The Elements of Computing Systems

By Noam Nisan and Shimon Schocken

MIT Press

Usage Notice

You are welcome to use this presentation, or any part of it,
in Nand to Tetris courses, or in other courses, as you see fit.

Usage is free provided that it supports instruction in an educational,
non-profit setting.

Feel free to use the slides as-is, or modify them as needed.

We'll appreciate it if you will give credit somewhere to Nand to Tetris,
and put a reference to www.nand2tetris.org

You are welcome to remove this slide from the presentation. If you make
extensive changes to the slides, you can remove the copyright notice also.

Happy teaching!

Noam Nisam / Shimon Schocken

Chapter 3: Memory

→ Time matters

- Sequential logic

- Flip Flops

- Memory units

- Counters

- Project 3 overview

Time-independent logic

- So far we ignored the issue of *time*

Time-independent logic

- So far we ignored the issue of *time*

- The chip's inputs were just "sitting there" – fixed and unchanging

- The chip's output was a pure function of the current inputs, and did not depend on anything that happened previously

- The output was computed "instantaneously"

- This style of gate logic is sometimes called:

  - *time-independent logic*

  - *combinational logic.*

Hello, time

Abstraction issues:

- The hardware must support maintaining "state"

example:

```
x = 17
```

- The hardware must support computations over time

example:

```
for i = 0 … 99:

    sum = sum + a[i]
```

Implementation issues:

- The hardware must handle the *physical time delays* associated with calculating and moving data from one chip to another.
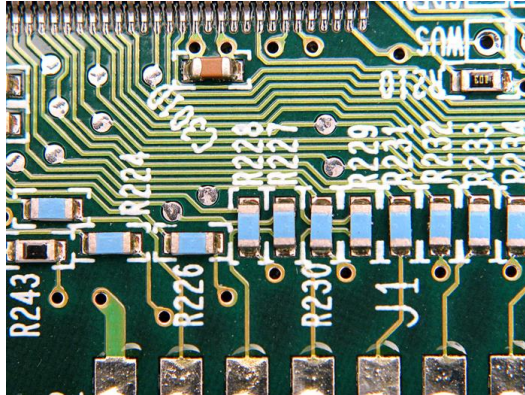
Physical time / clock time

physical
time:

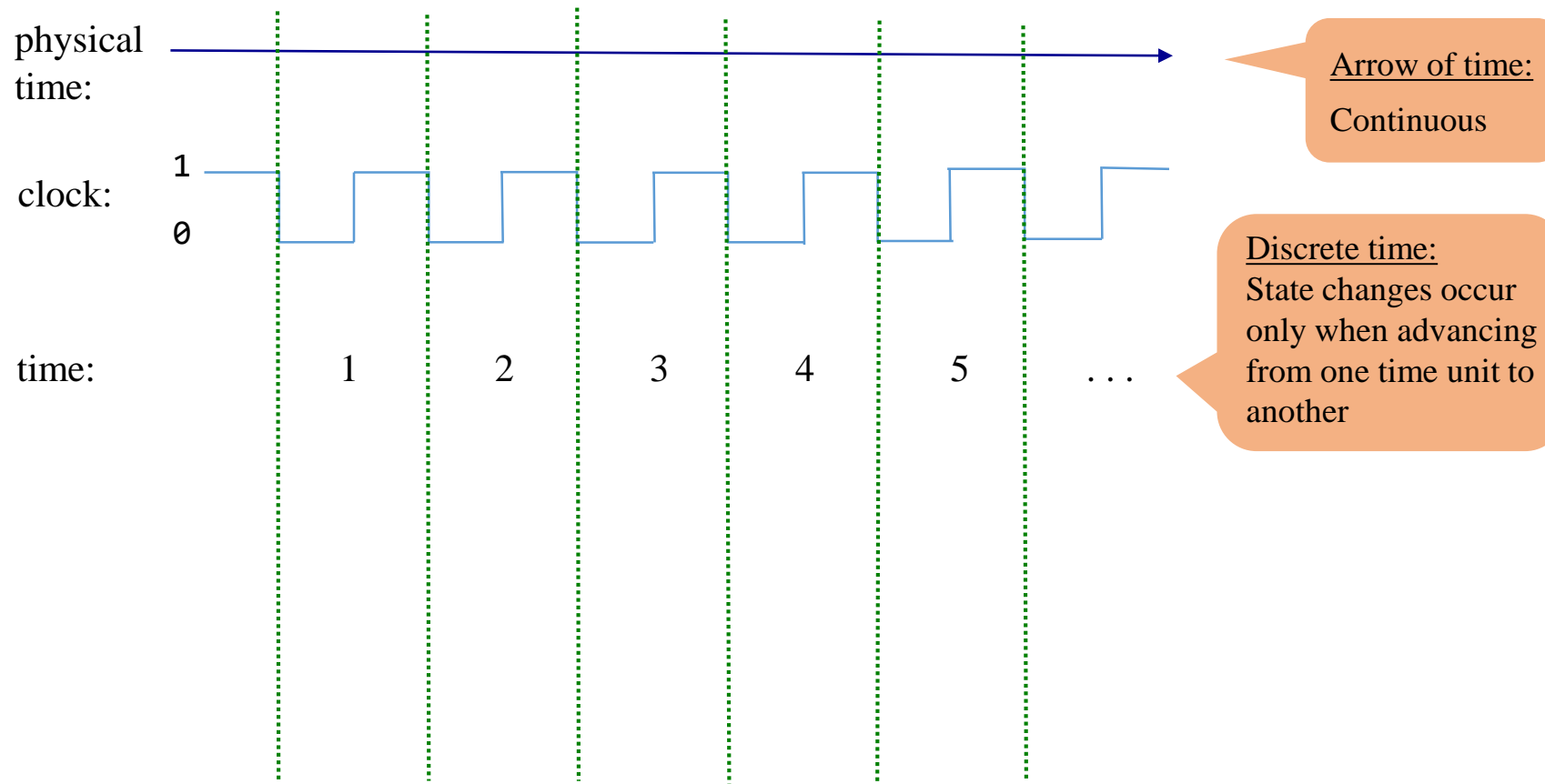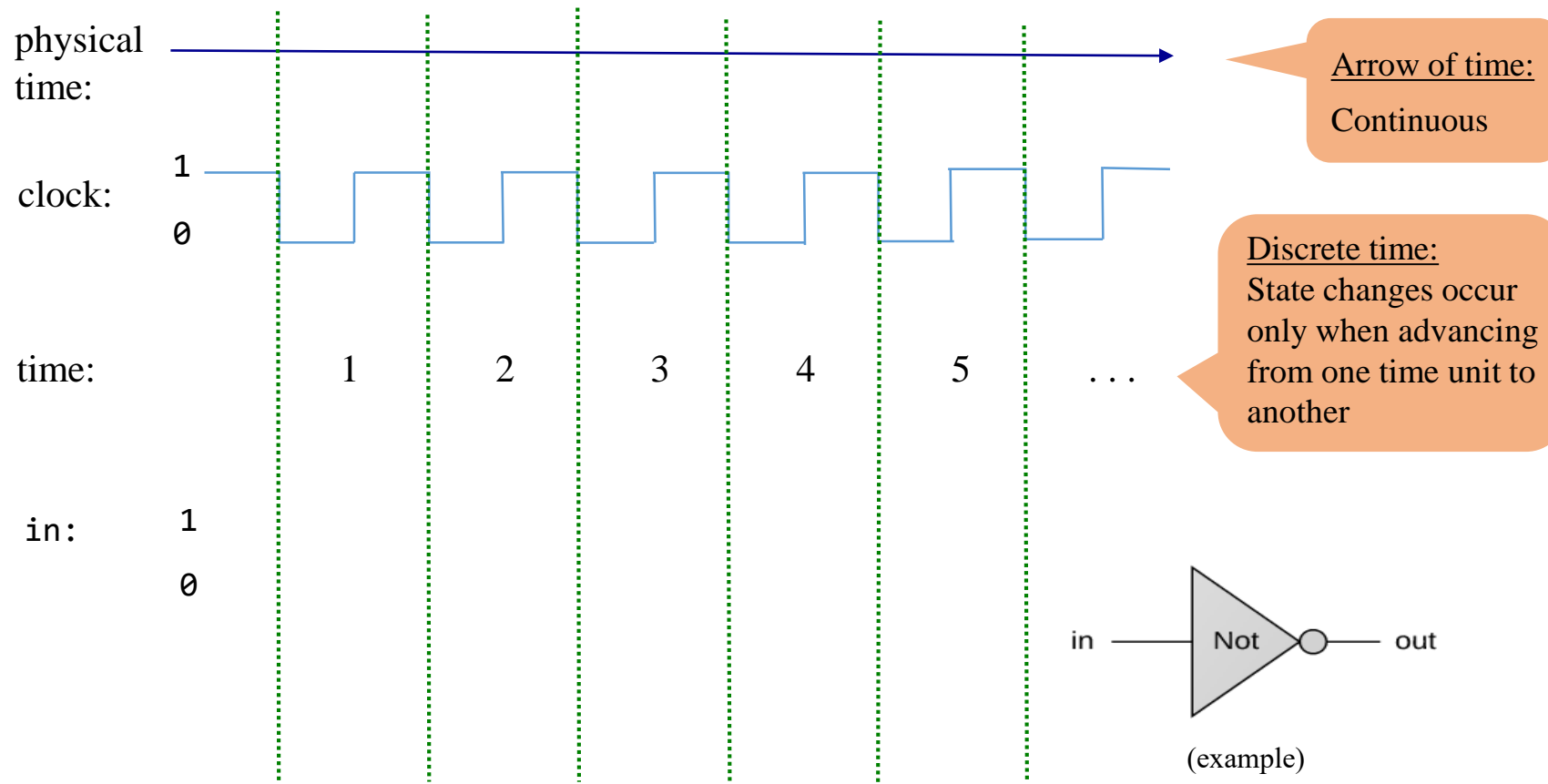Arrow of time:

Continuous
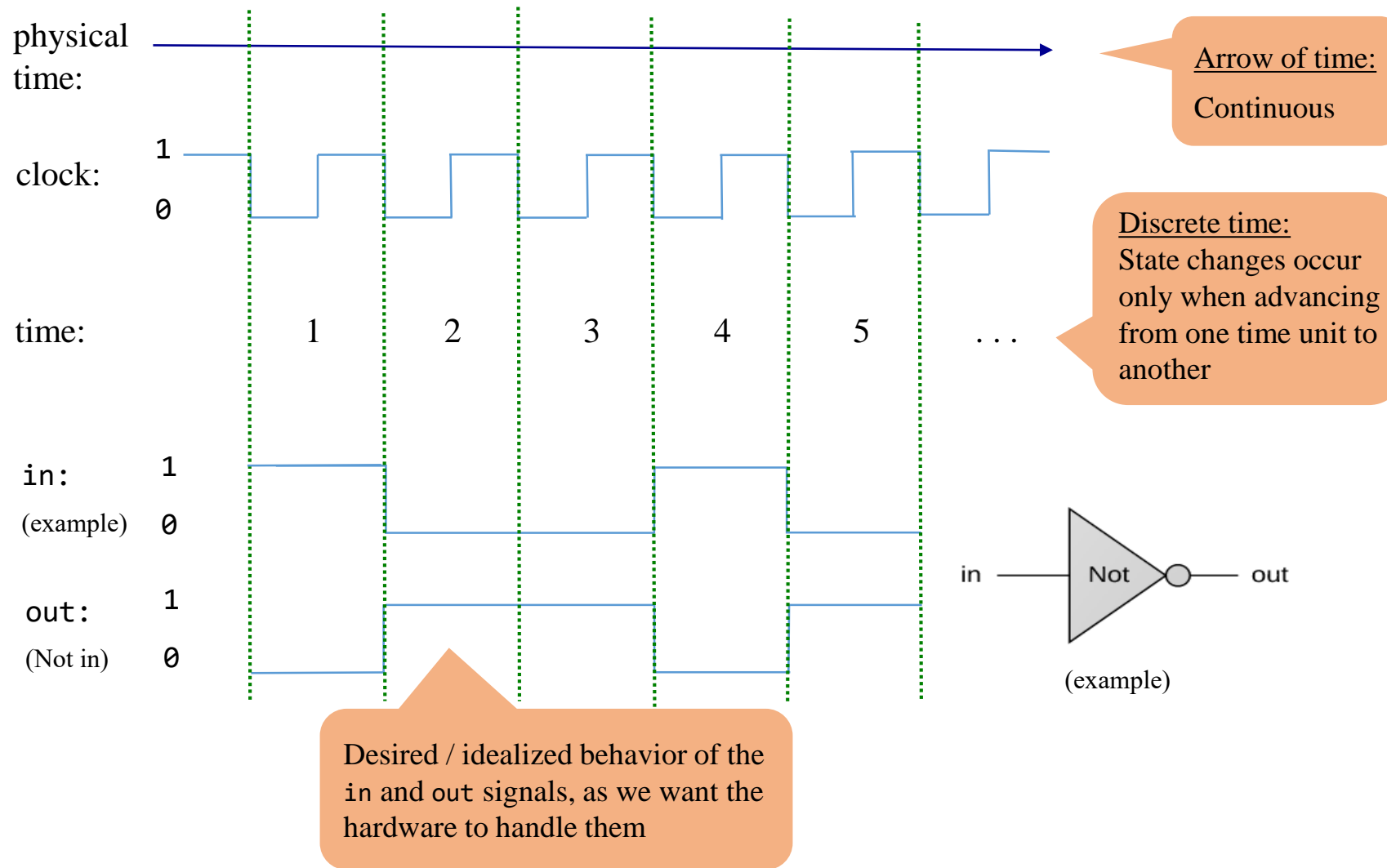
Physical time / clock time
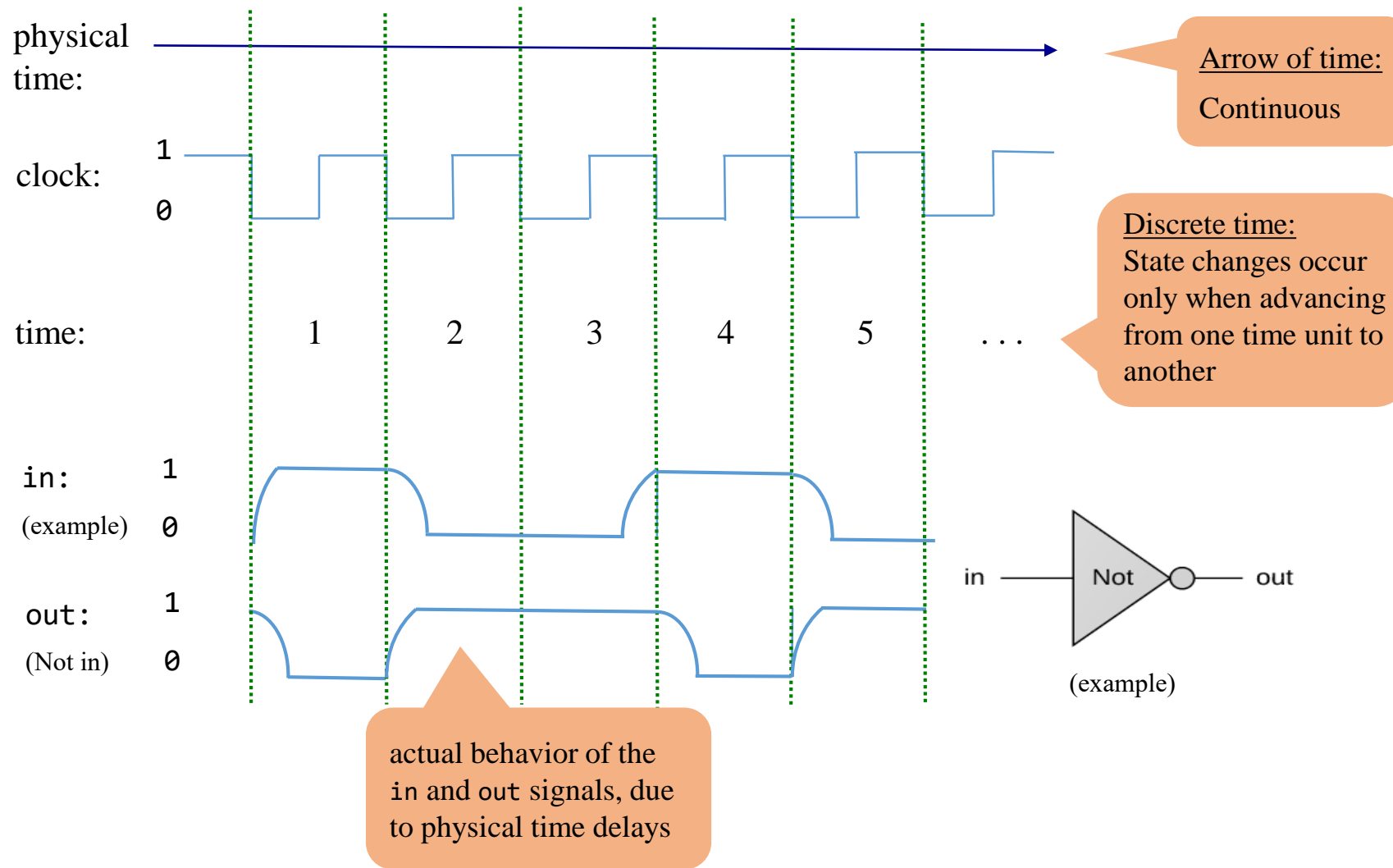
Physical time / clock time

Chip behavior over time (example)

Chip behavior over time (example)

Chip behavior over time (example)

physical time:

Arrow of time:
Continuous

clock: 1 0

Discrete time:
State changes occur only when advancing from one time unit to another

time: 1   2   3   4   5   . . .

in: 1
(example) 0

out: 1
(Not in) 0

in ——— Not ◯ ——— out

(example)

actual behavior of the in and out signals, due to physical time delays

Chip behavior over time (example)

physical time:

Arrow of time:

Continuous

clock:

1

0

time:

1   2   3   4   5   . . .

Discrete time:
State changes occur only when advancing from one time unit to another

in:
(example)

1

0

out:
(Not in)

1

0

in —— Not —o— out

(example)

Time delays
- propagation delays
- computation delays

Chip behavior over time (example)

physical time:

Arrow of time:
Continuous

clock:

1

0

Discrete time:
State changes occur only when advancing from one time unit to another

time:

1   2   3   4   5   . . .

in:
(example)

1

0

out:
(Not in)

1

0

in ——[Not >o]—— out

(example)

Time delays
- propagation delays
- computation delays

Clock cycle
- designed to neutralize the time delays
- cycle length: slightly longer than the time delays

Chip behavior over time (example)

physical time:

Arrow of time:
Continuous

clock: 1 0

Discrete time:
State changes occur only when advancing from one time unit to another

time: 1 2 3 4 5 . . .

in: 1 0
(example)

out: 1 0
(Not in)

in —— Not ○— out

(example)

Not: an example of a *combinational chip*:

- The gate reacts "immediately" to the inputs
- Well, not really, but the clock's behavior creates this effect.

Combinational logic / sequential logic

Combinational logic:

The output is a pure function of the present input only
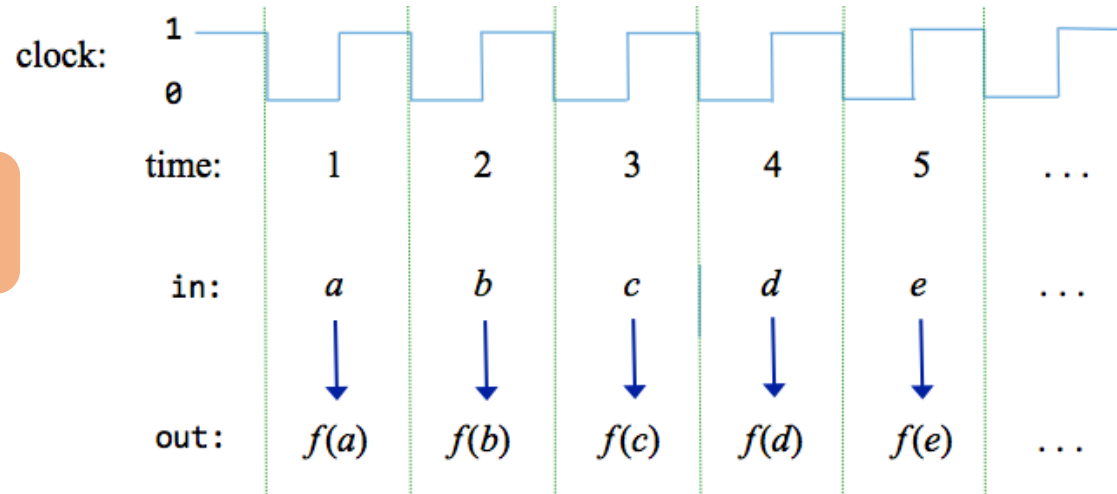
Sequential logic:

The output depends on:
- the present input (optionally)
- the history of the input (creates a memory effect).
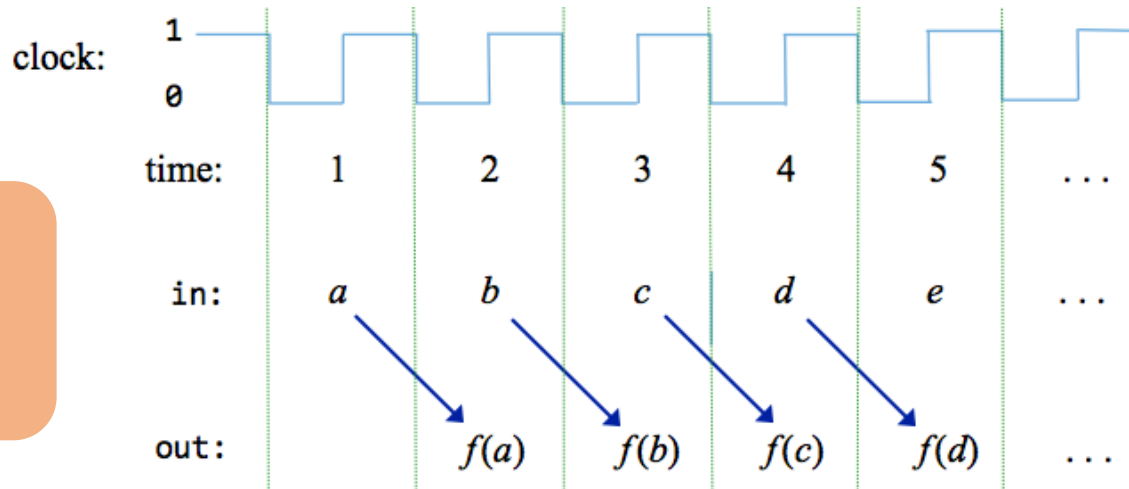
Flip-Flop
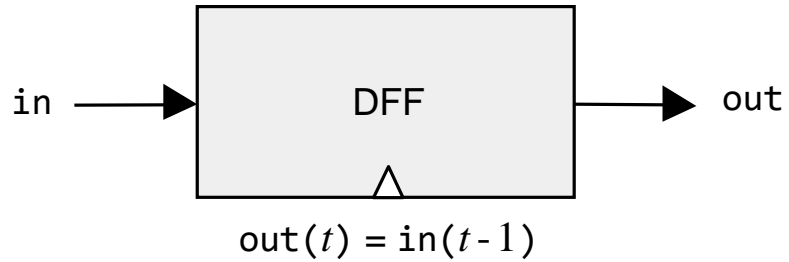
$$\text{out}(t) = \text{in}(t-1)$$

The simplest state-keeping gate:

- 1-bit input, 1-bit output

- The gate outputs its previous input: $\text{out}(t) = \text{in}(t-1)$

- Implementation: a gate that can flip between two stable states:
  "remembering 0", or "remembering 1"

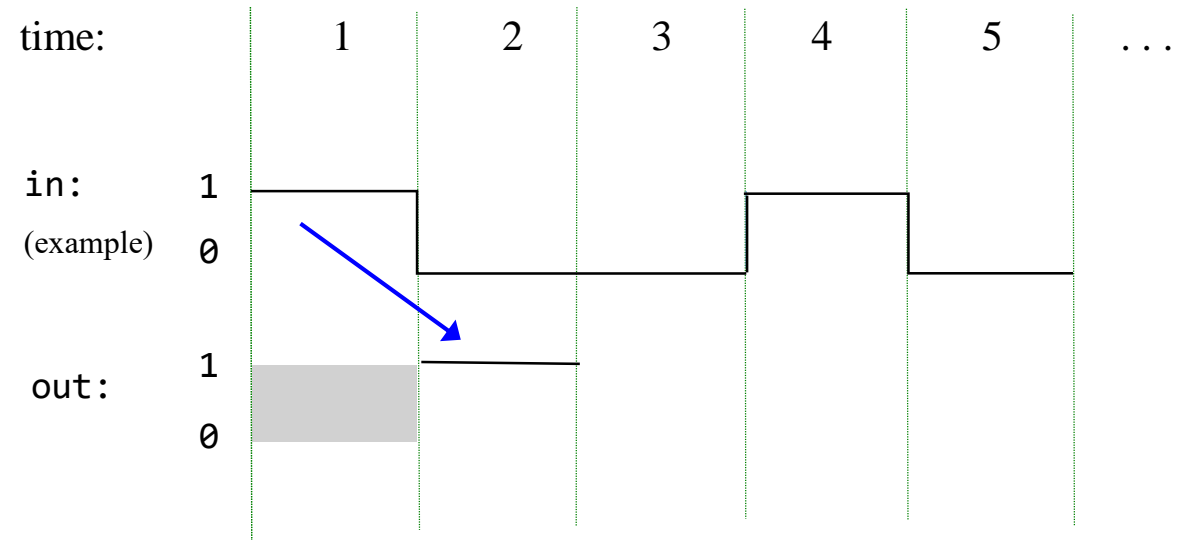- Gates that feature this behavior are called *data flip-flops*.
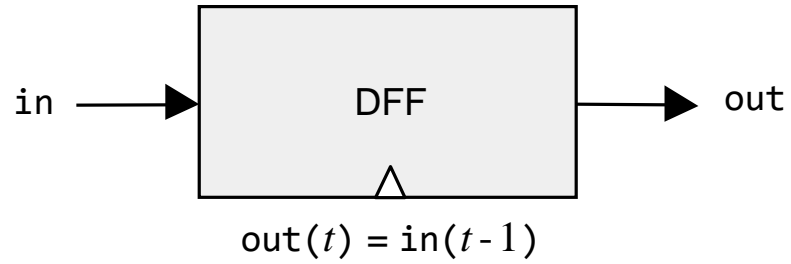
Flip-Flop

in → | DFF | → out

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

time: 1   2   3   4   5   . . .

in: 1

       0

Flip-Flop

in → DFF → out

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

time: 1 2 3 4 5 . . .

in: 1
(example) 0

out: 1
0

Flip-Flop

in → DFF → out

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

time: 1 2 3 4 5 . . .

in: (example) 1 0

out: 1 0

Flip-Flop

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

time: 1   2   3   4   5   . . .

in: 1

(example) 0

out: 1

        0

Flip-Flop

DFF

in → out

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

time: 1 2 3 4 5 . . .

in:
(example)

out:

Flip-Flop

in → DFF → out

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

time: 1 2 3 4 5 . . .
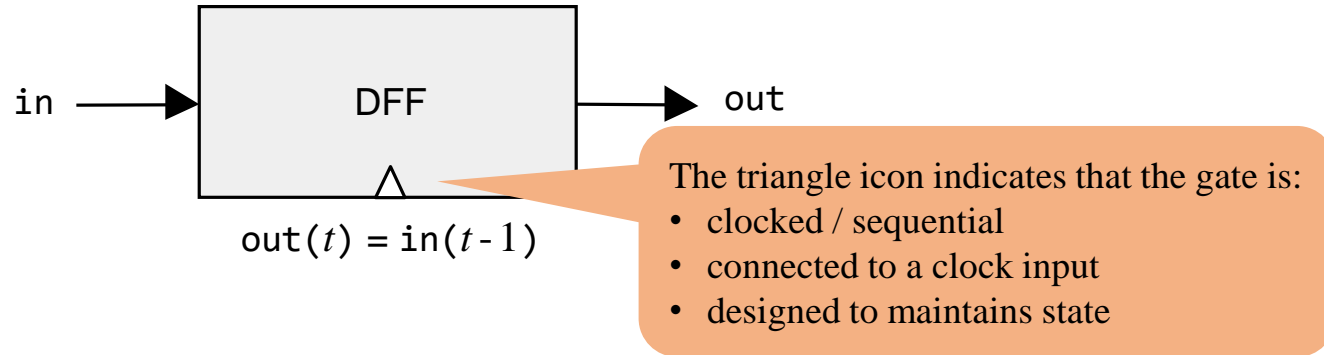
in: 1
(example) 0

out: 1
0

Flip-Flop

$$out(t) = in(t-1)$$
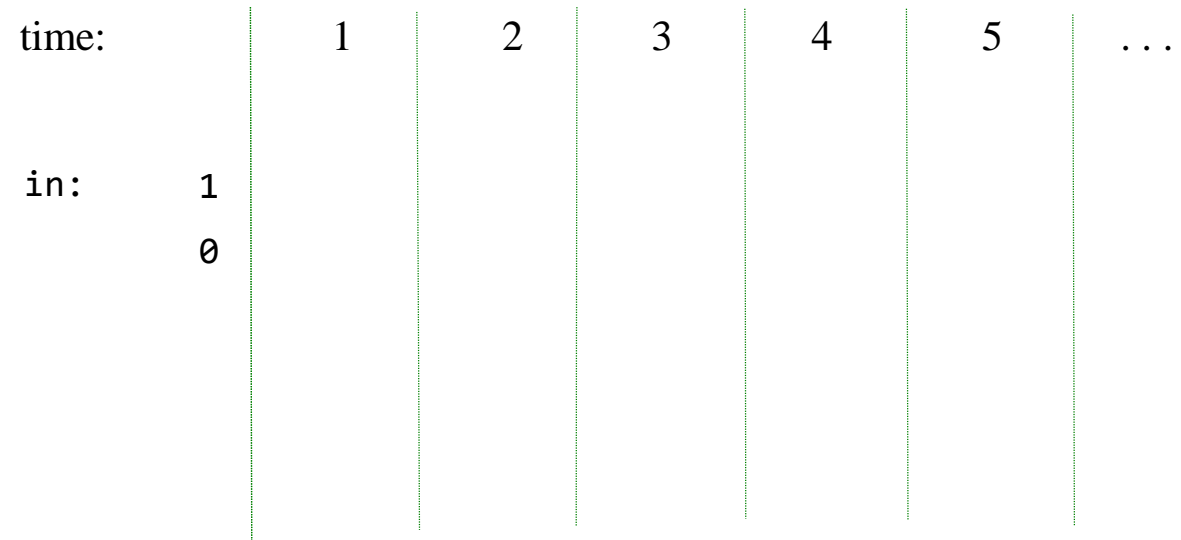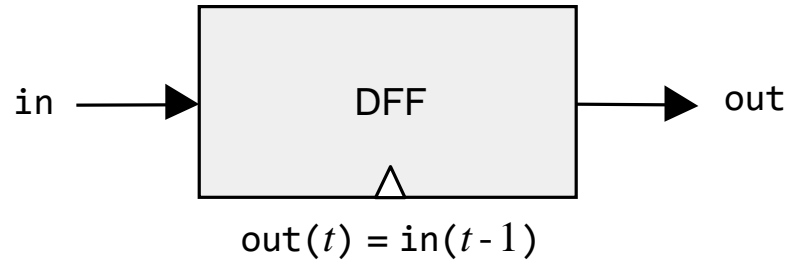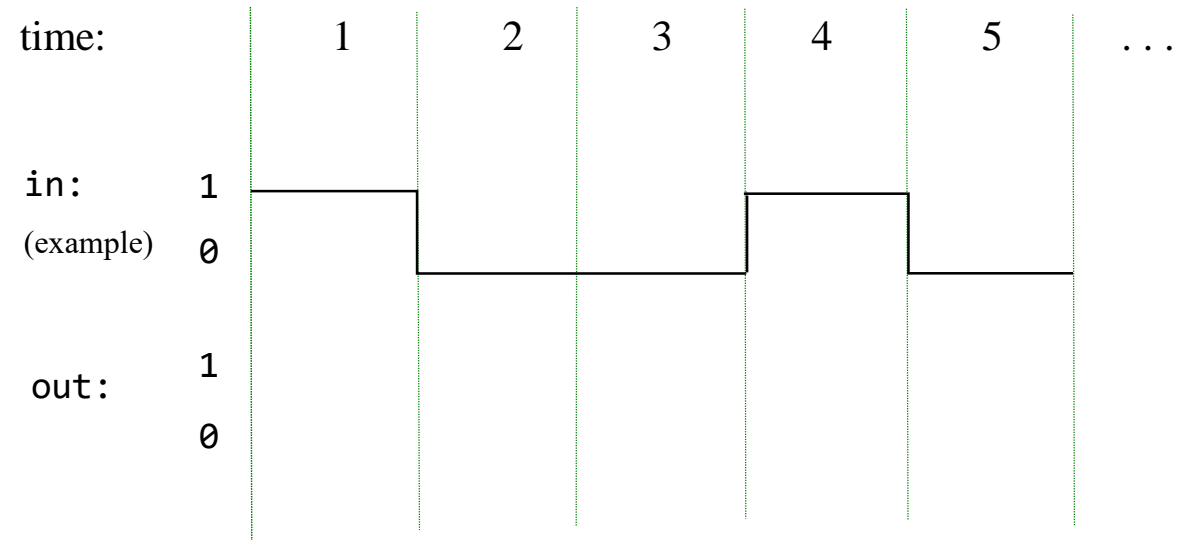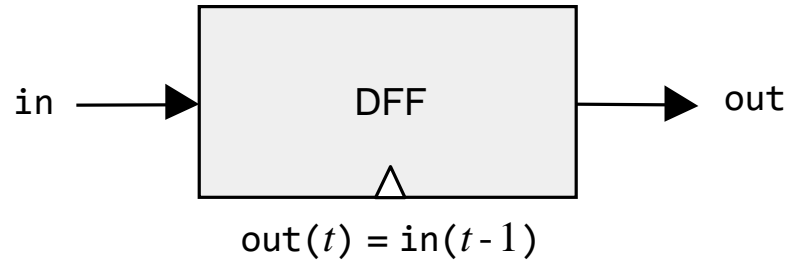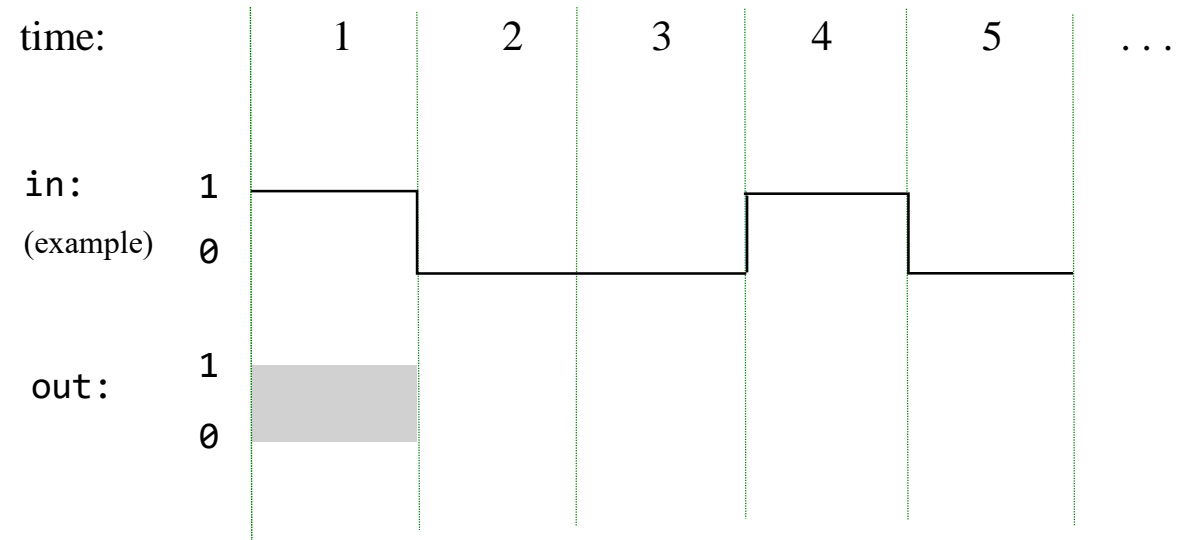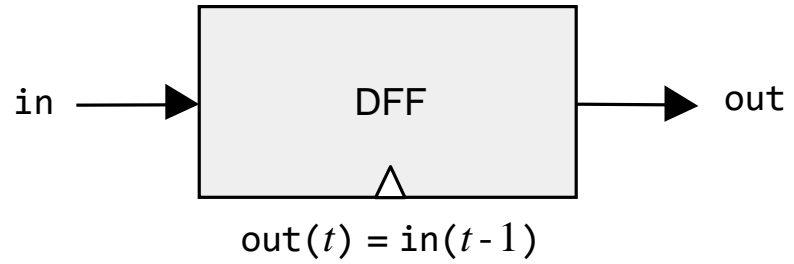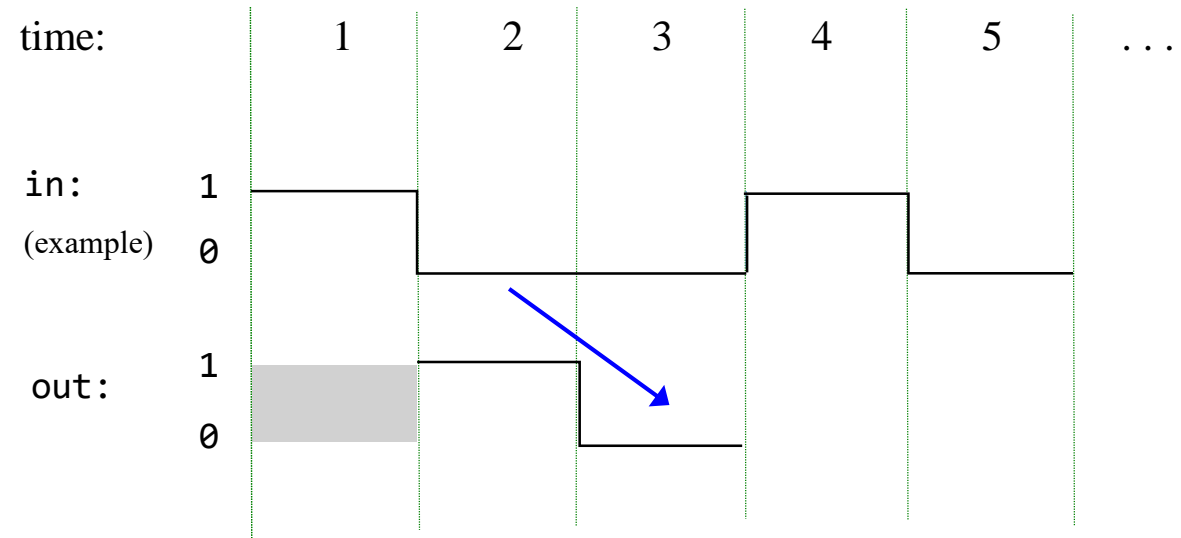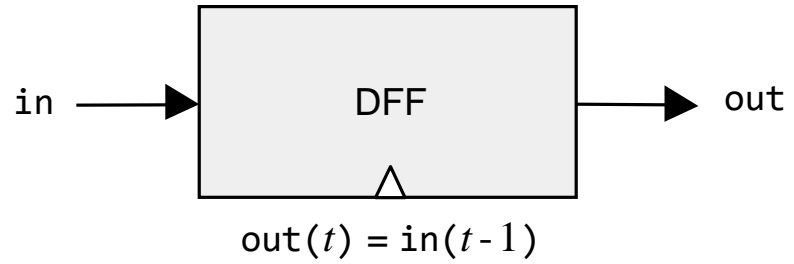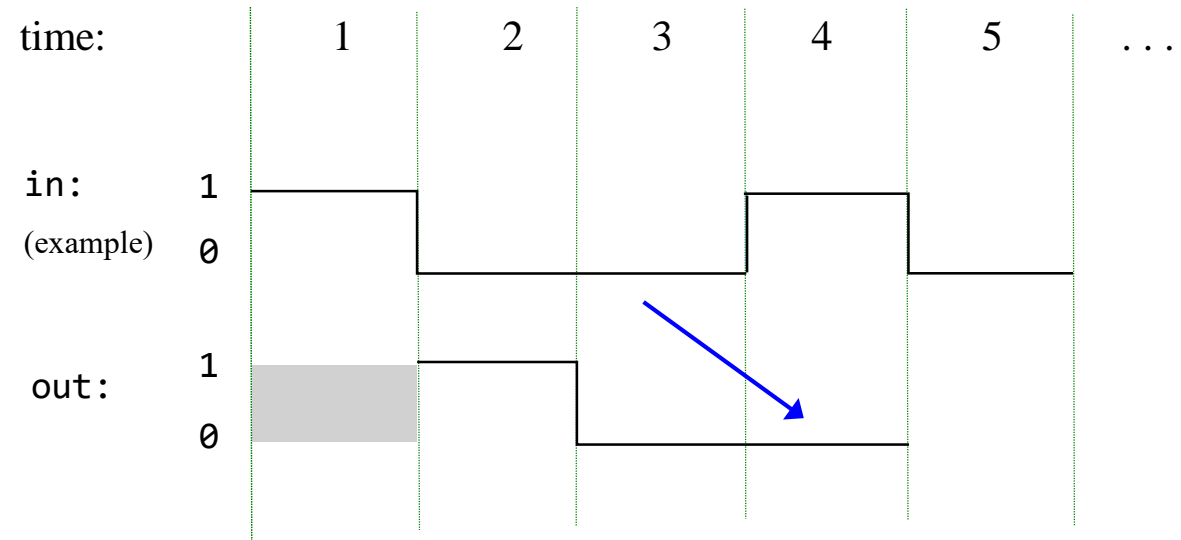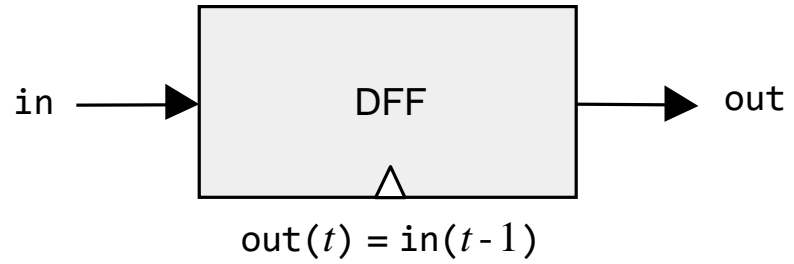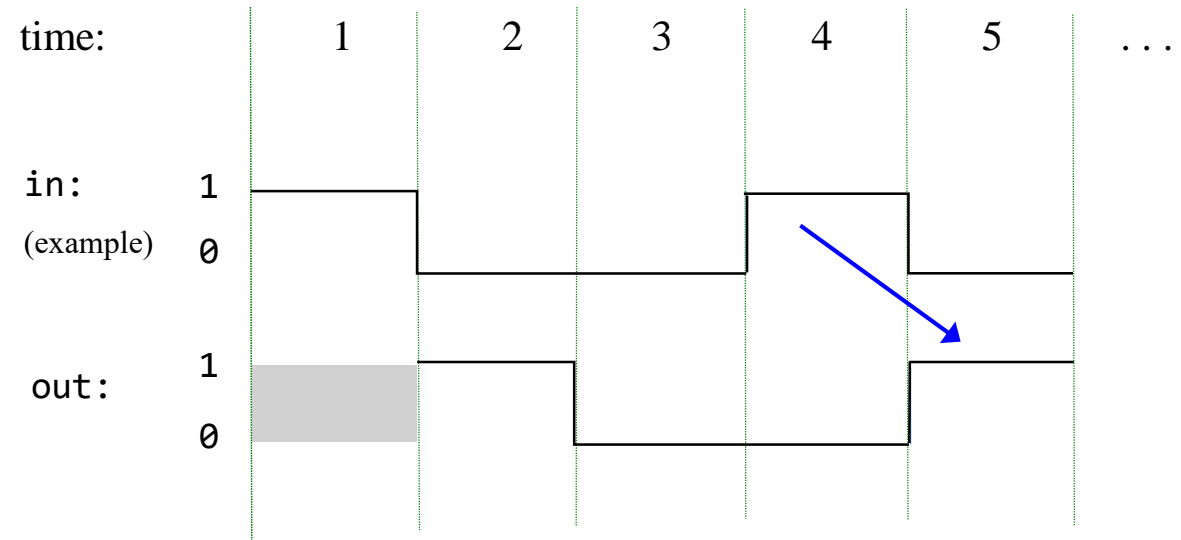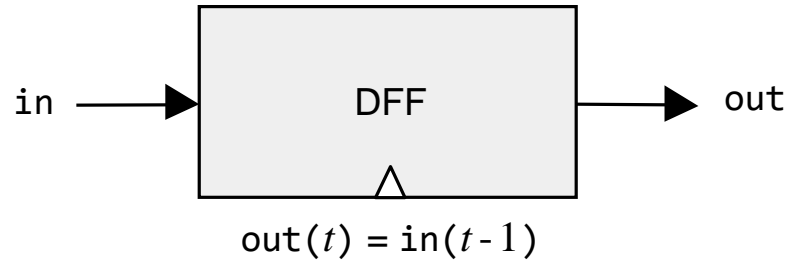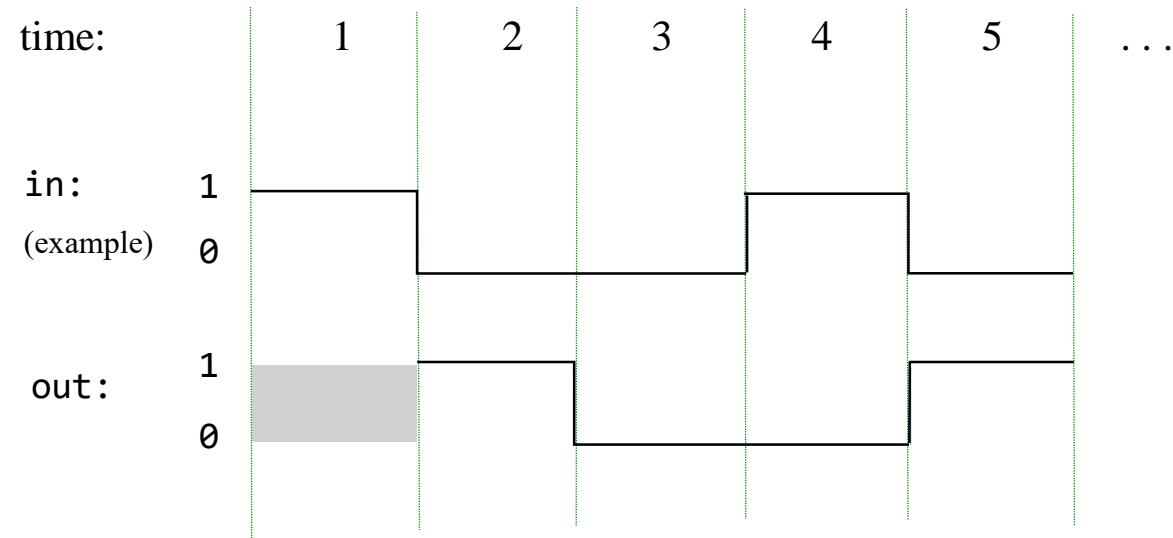
Flip-Flop

DFF implementation notes

A DFF bi-state architecture can be built from Nand gates:

- step 1: create an input-output loop,
        achieving a combinational (un-clocked) flip-flop

- step 2: isolate across time steps using a "master-slave" architecture

  The resulting implementation is elegant, but conceptually confusing

Technical note

The implementation described above is impossible in our hardware simulator, since:

- The supplied simulator does not permit combinational loops

- A cycle in hardware connections is allowed only if the cycle passes through a sequential ("clocked") gate

Implementing sequential chips

- The supplied simulator features a built-in DFF gate

- Sequential chips are implemented by using built-in DFF chip parts.

Sequential chips

Sequential chips are capable of:

- maintaining state, and, optionally,
- acting on the state, and on the current inputs

$$\text{state}(t) = f(\text{state}(t\text{-}1), \text{input}(t))$$

Example: DFF

- The DFF state: the value of the input from the previous time unit
- The simplest, most elementary sequential chip
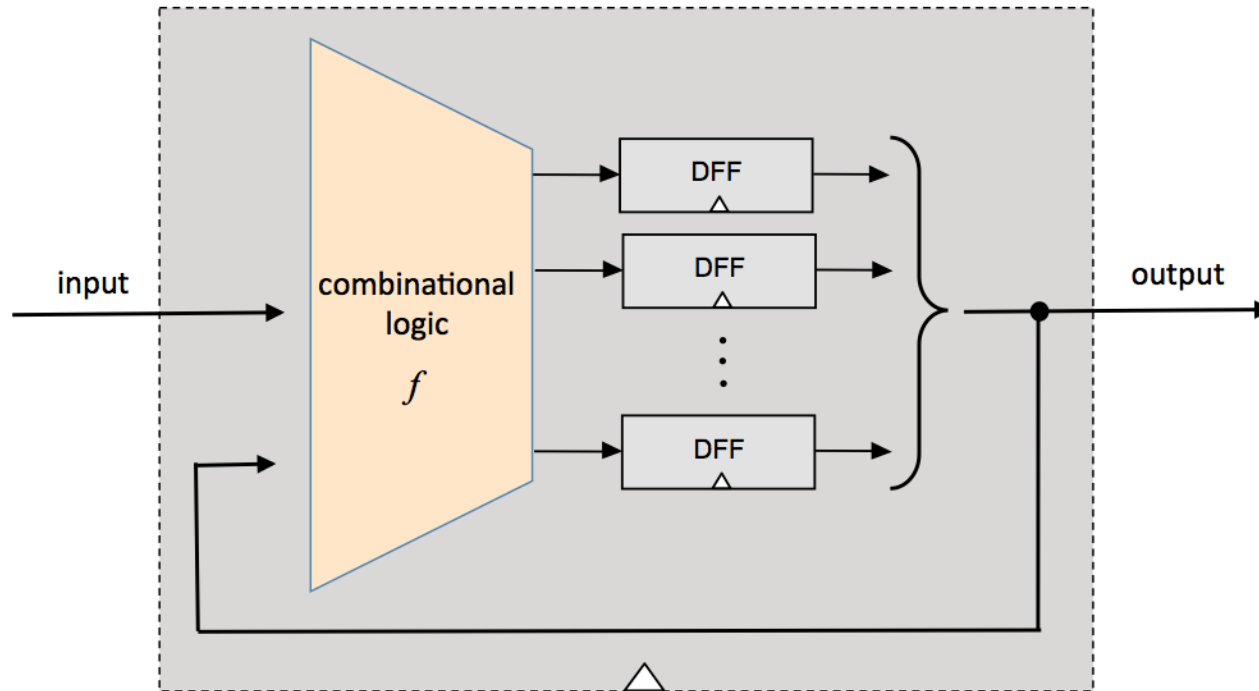
Example: RAM

- The RAM state: the current values of all its registers
- given some address (input), the RAM emits the value of the selected register

Implementation note

- All combinational chips are constructed from Nand gates
- All sequential chips are constructed from DFF gates, and combinational chips.
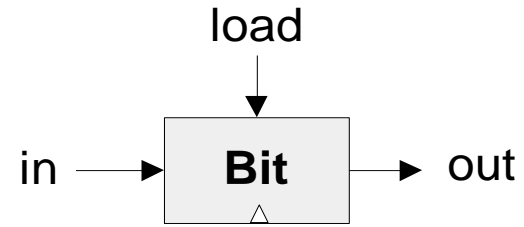
Sequential chips

$$\text{state}(t) = f(\text{state}(t-1), \text{input}(t))$$

Implementation note

- All combinational chips are constructed from Nand gates

- All sequential chips are constructed from DFF gates, and combinational chips.

Sequential chip: 1-bit register

if $\texttt{load}(t)$ then $\texttt{out}(t+1) = \texttt{in}(t)$
else $\qquad\qquad \texttt{out}(t+1) = \texttt{out}(t)$

- Designed to "remember", or "store", a single bit

- More accurately:

  - Stores a bit until...

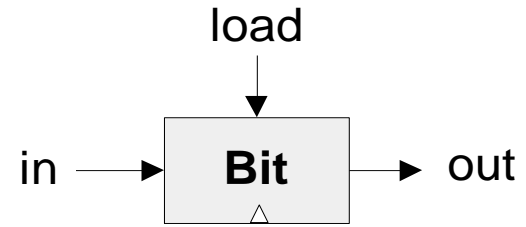 - Instructed to load, and store, another bit.

1-bit register
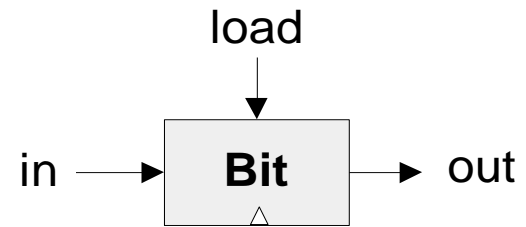
if $\text{load}(t)$ then $\text{out}(t+1) = \text{in}(t)$
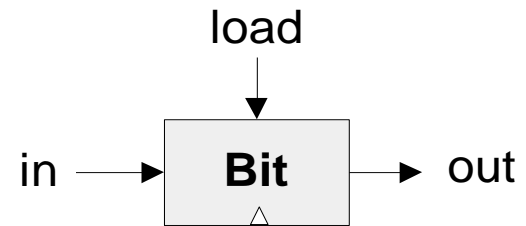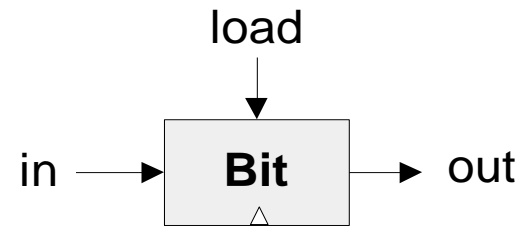else $\qquad\qquad \text{out}(t+1) = \text{out}(t)$

time:	1	2	3	4	5
load: 1 0					
in: 1 0					
out: 1 0					

1-bit register

if $\texttt{load}(t)$ then $\texttt{out}(t+1) = \texttt{in}(t)$
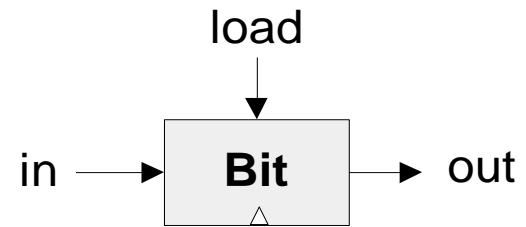else $\qquad\qquad \texttt{out}(t+1) = \texttt{out}(t)$

1-bit register

load

in → **Bit** → out

if $\text{load}(t)$ then $\text{out}(t+1) = \text{in}(t)$
else $\qquad\qquad\text{out}(t+1) = \text{out}(t)$

time: 1 2 3 4 5

load: 1
(example) 0

in: 1
(example) 0

out: 1
0

1-bit register

load

in → **Bit** → out

if load(t) then out$(t+1)$ = in(t)
else $\qquad$ out$(t+1)$ = out(t)

time: 1 2 3 4 5

load: 1
(example) 0

in: 1
(example) 0

out: 1
  0

1-bit register

if $\texttt{load}(t)$ then $\texttt{out}(t+1) = \texttt{in}(t)$
else $\qquad\qquad \texttt{out}(t+1) = \texttt{out}(t)$

1-bit register

$$\text{if } \texttt{load}(t) \text{ then } \texttt{out}(t+1) = \texttt{in}(t)$$
$$\text{else} \qquad\qquad \texttt{out}(t+1) = \texttt{out}(t)$$

1-bit register

$$\text{if } \texttt{load}(t) \text{ then } \texttt{out}(t+1) = \texttt{in}(t)$$
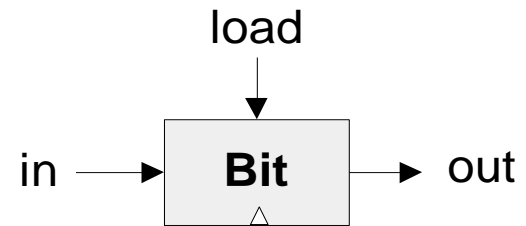$$\text{else} \qquad\qquad \texttt{out}(t+1) = \texttt{out}(t)$$

1-bit register

if $\text{load}(t)$ then $\text{out}(t+1) = \text{in}(t)$
else $\text{out}(t+1) = \text{out}(t)$

1-bit register

load

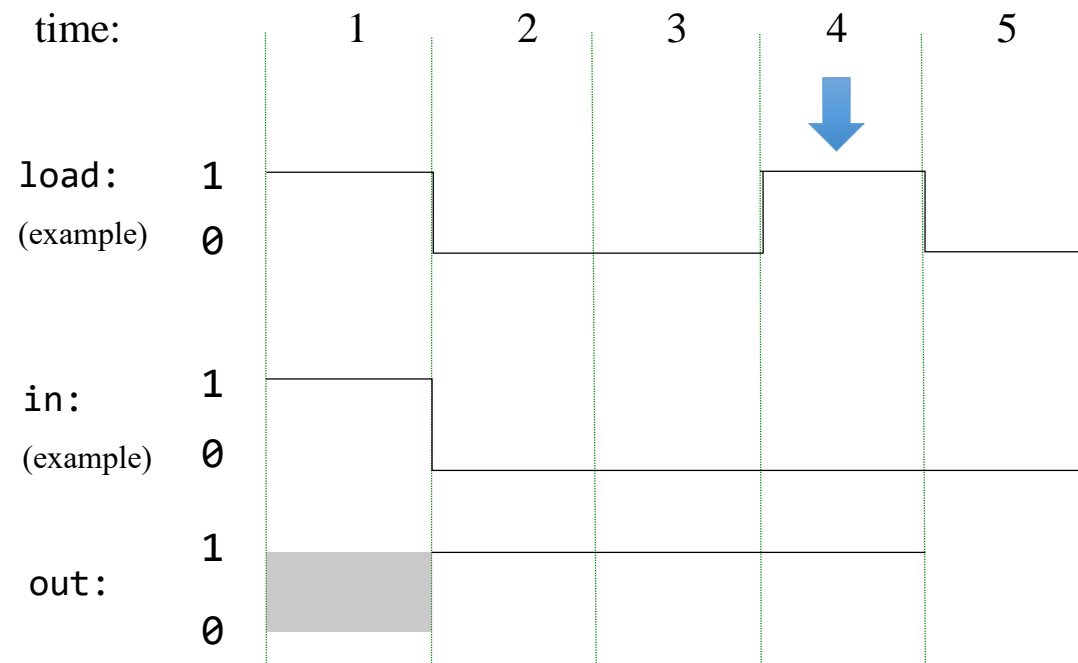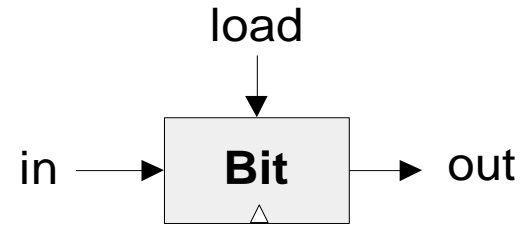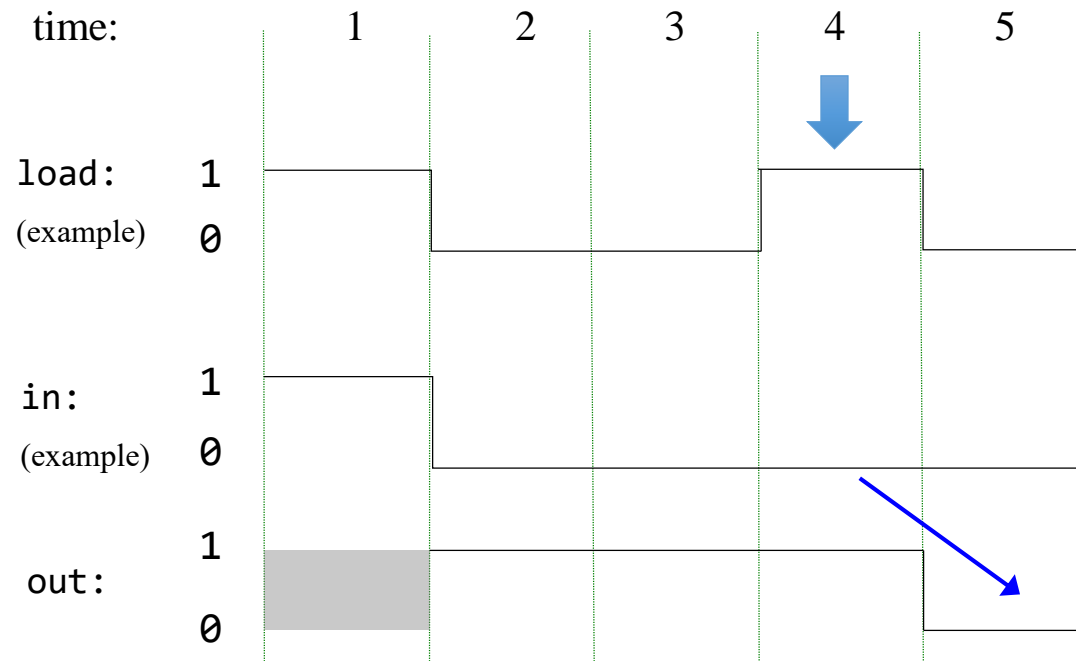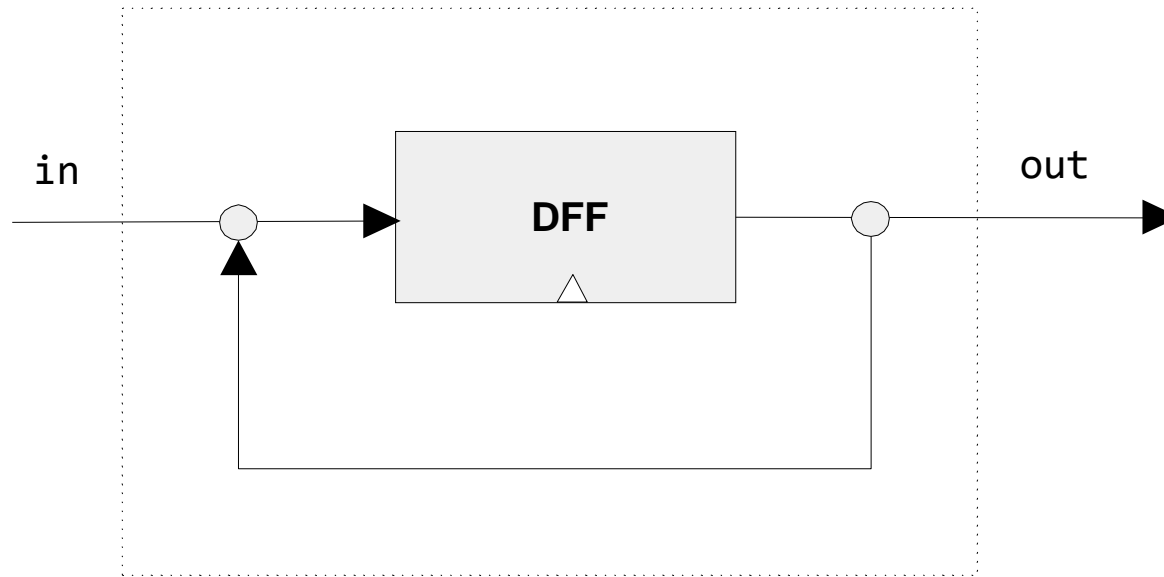in → **Bit** → out

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

time: 1   2   3   4   5

load: 1
(example) 0

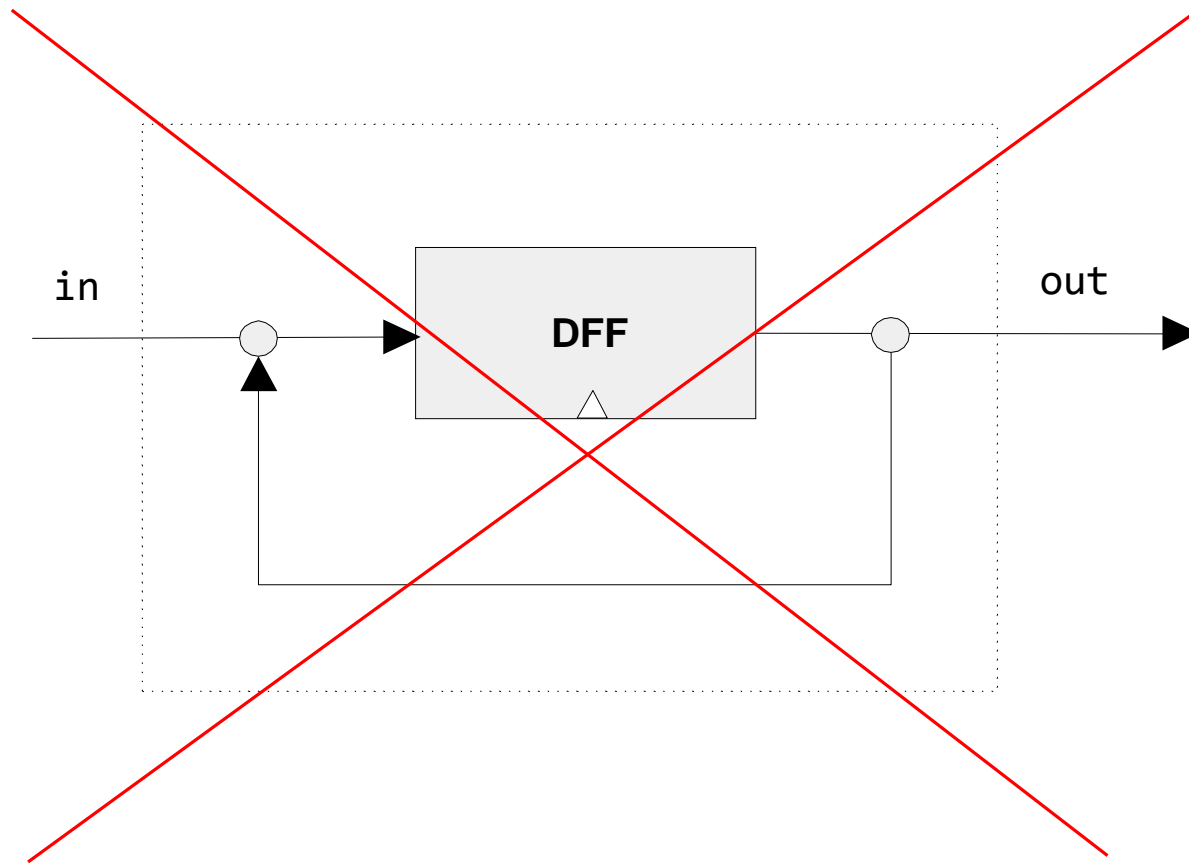in: 1
(example) 0

out: 1
         0

Resulting behavior:
Stores and emits a
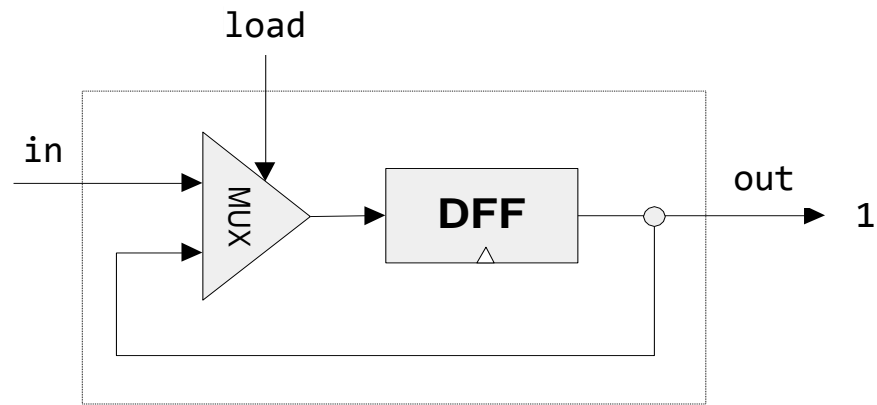value, until instructed
to load (and store) a
new value

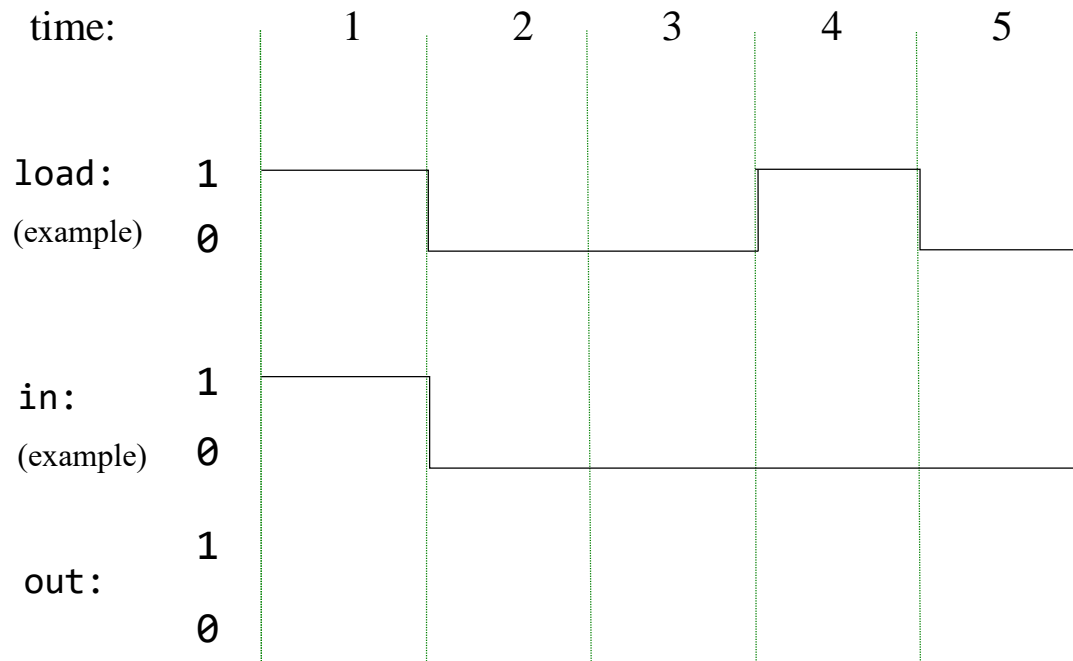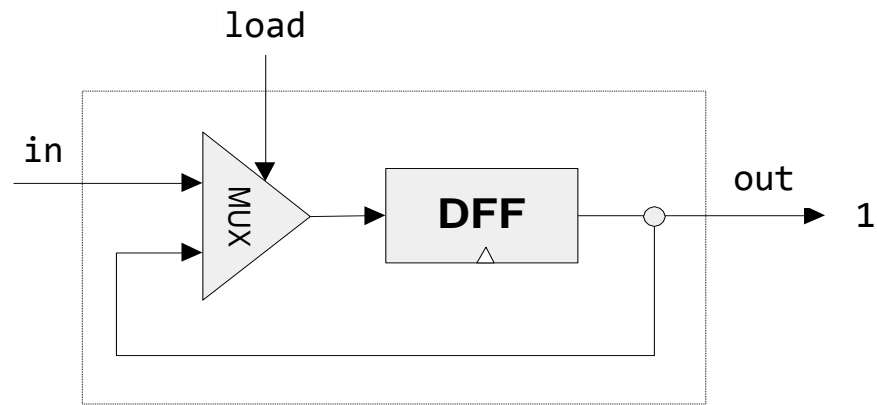# 1-bit register implementation – first attempt

1-bit register implementation – first attempt

1-bit register implementation

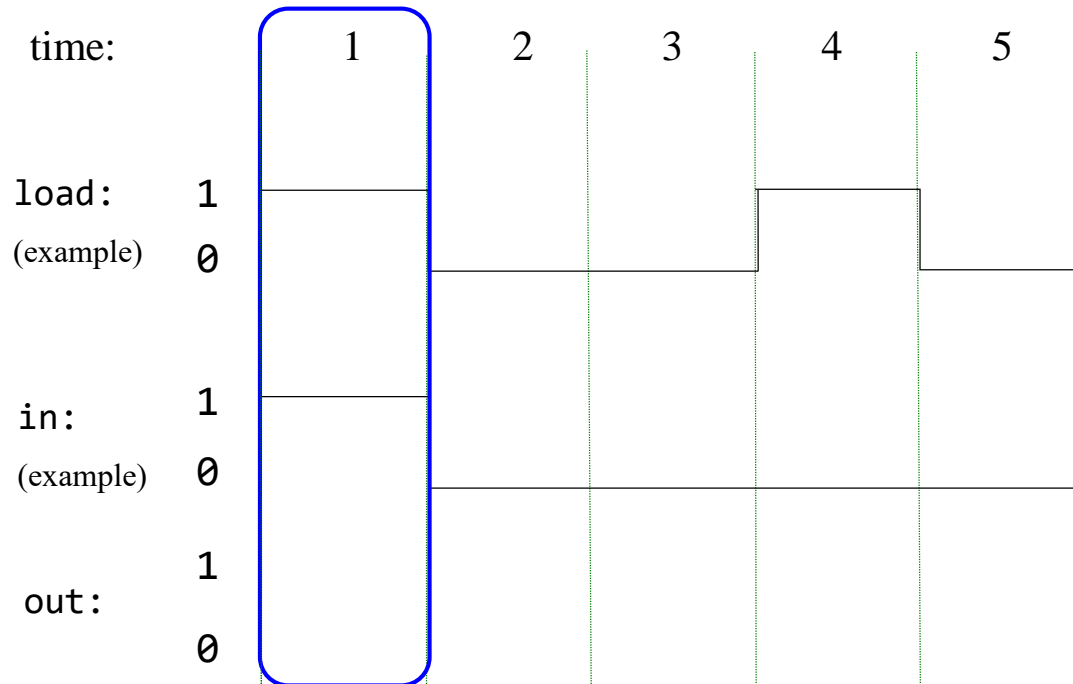# 1-bit register implementation

1-bit register implementation

1-bit register implementation

1-bit register implementation
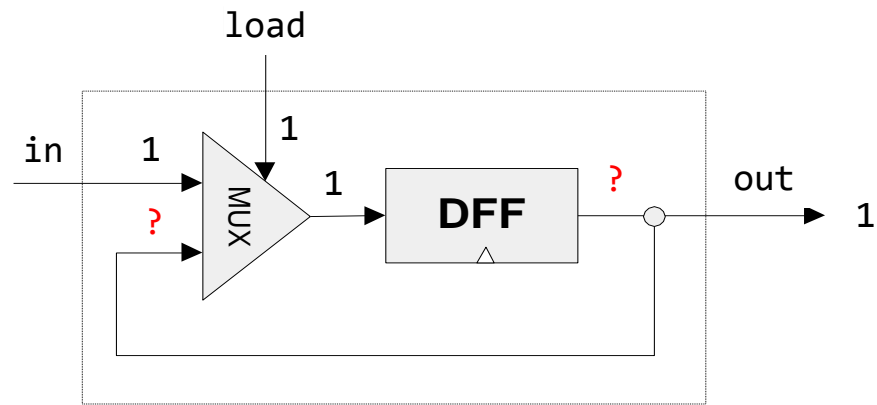
1-bit register implementation

1-bit register implementation

1-bit register implementation

1-bit register implementation

time:

load: (example)

in: (example)

out:
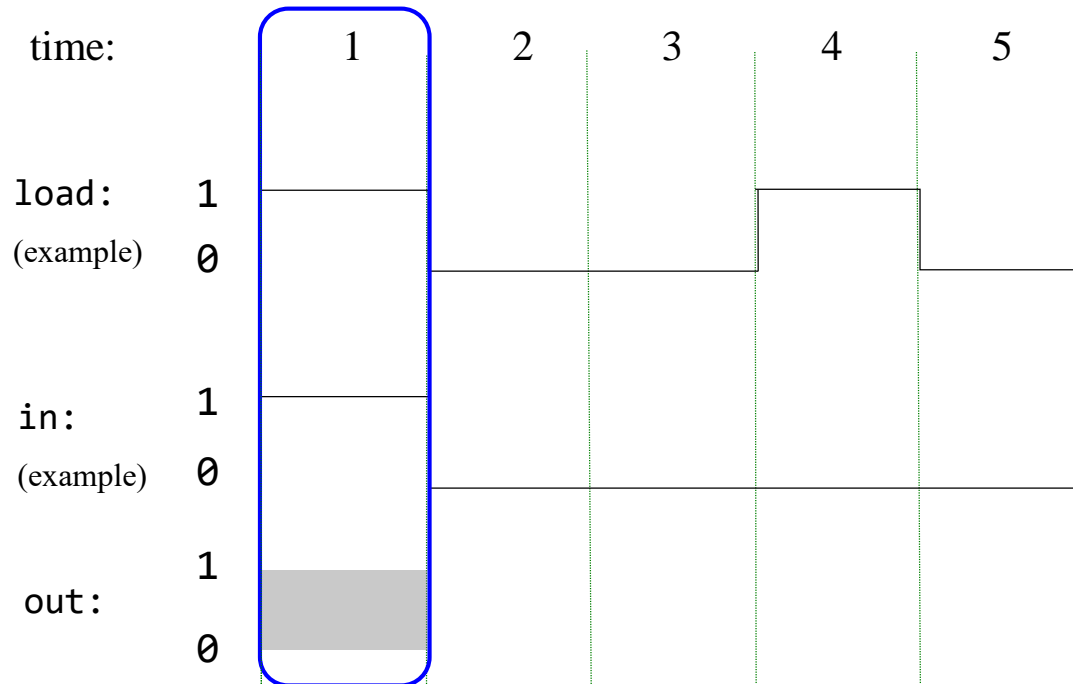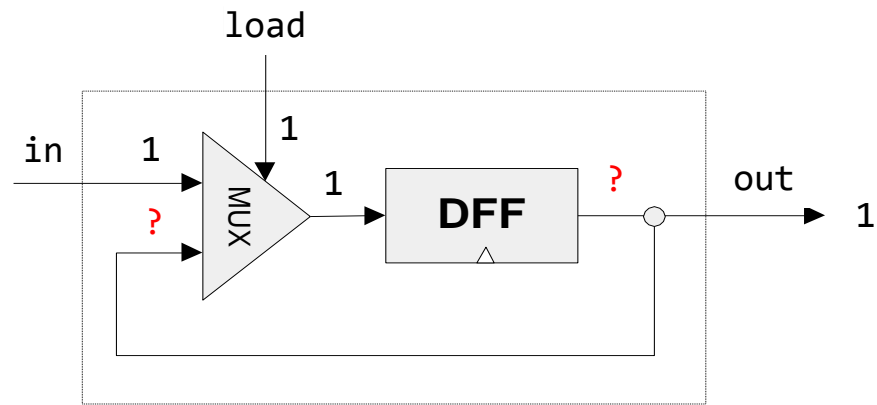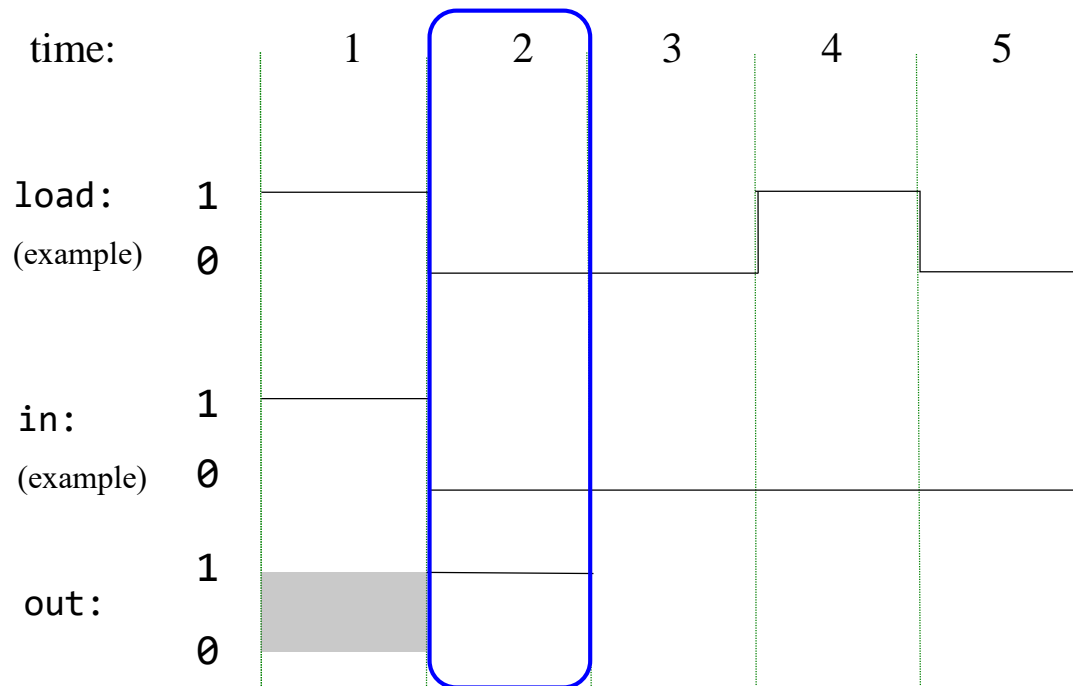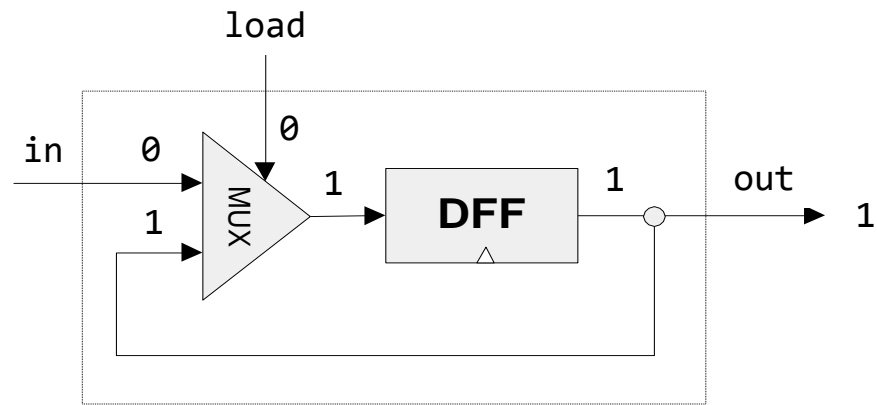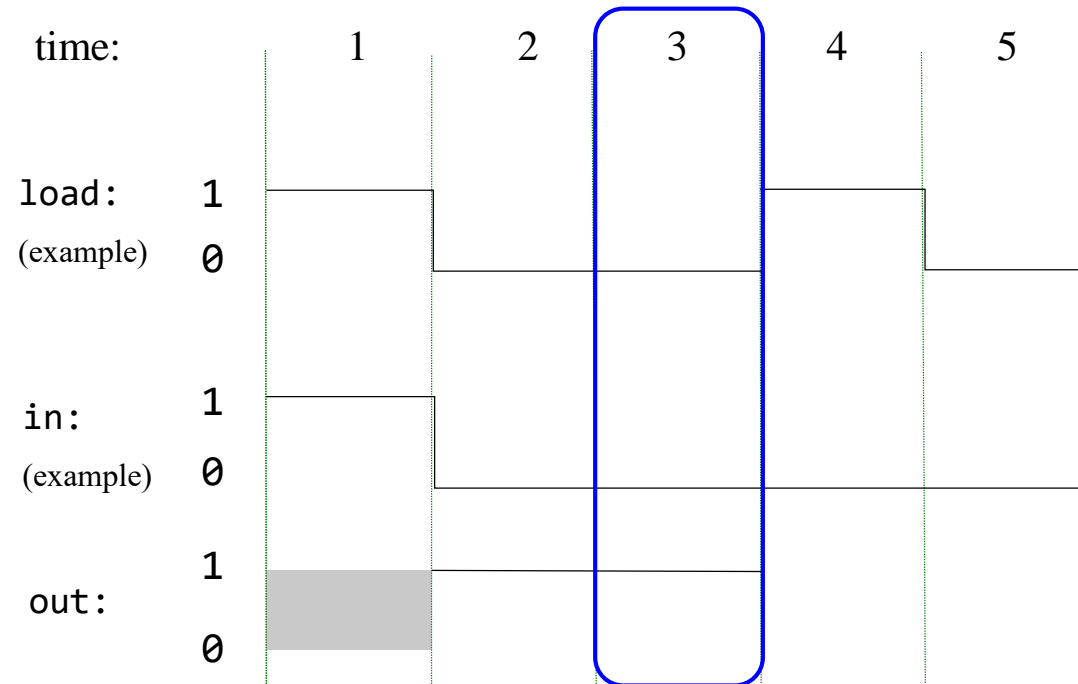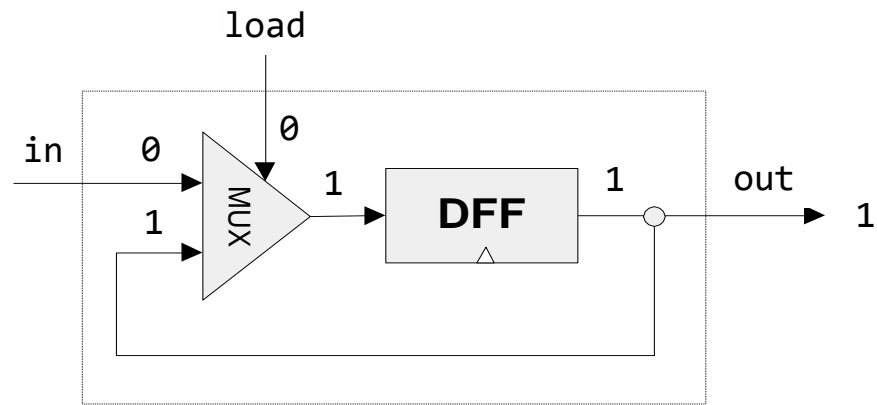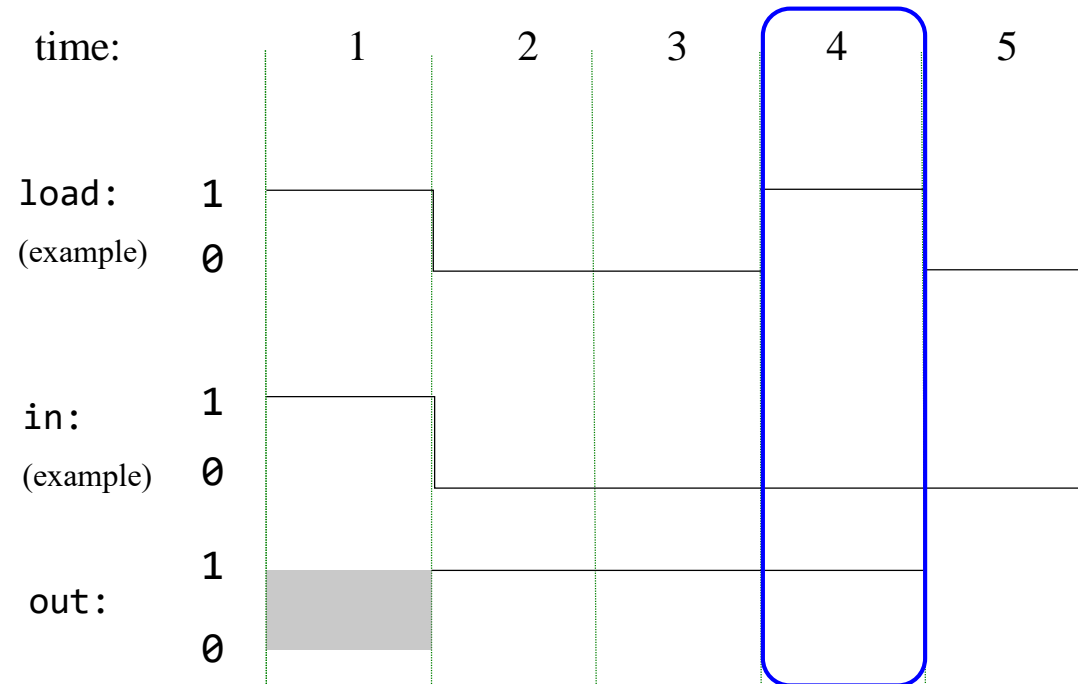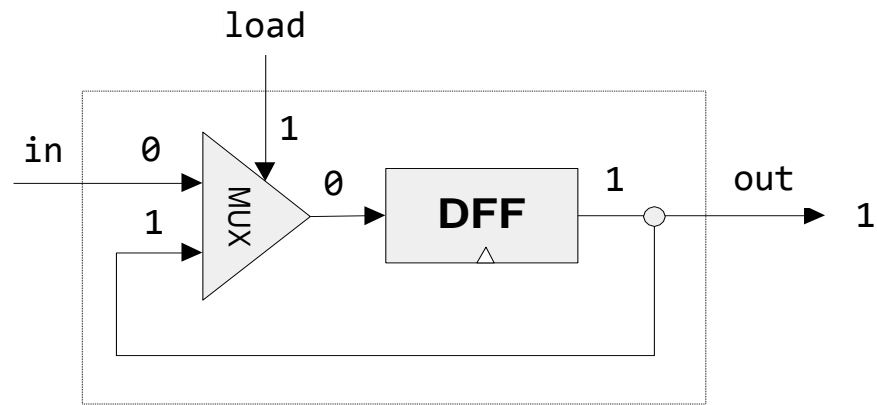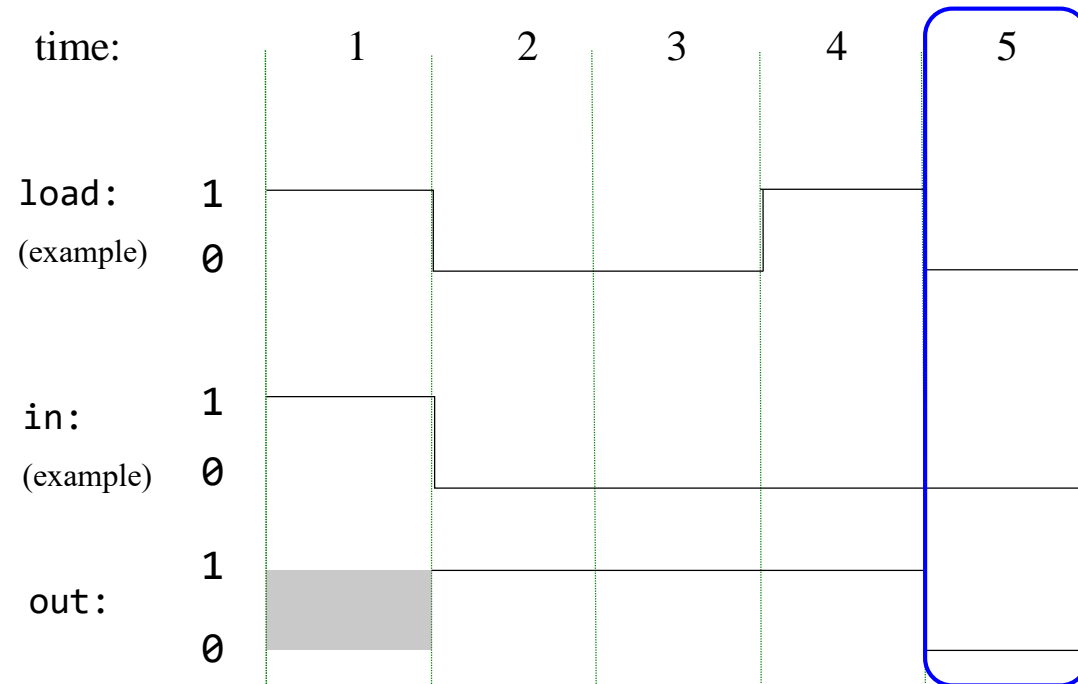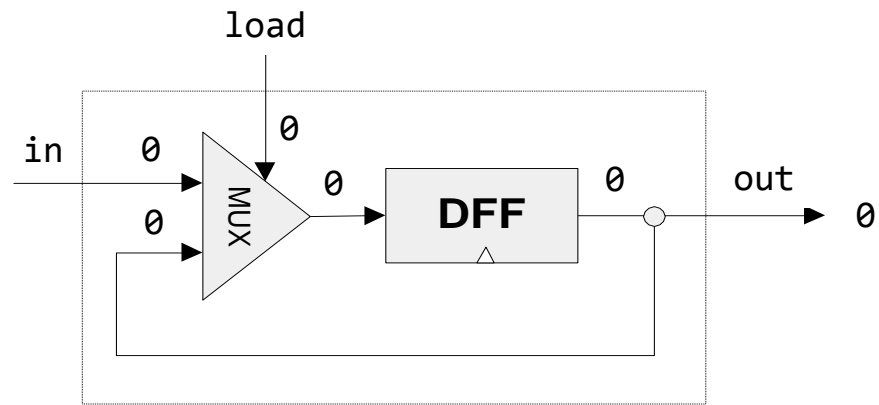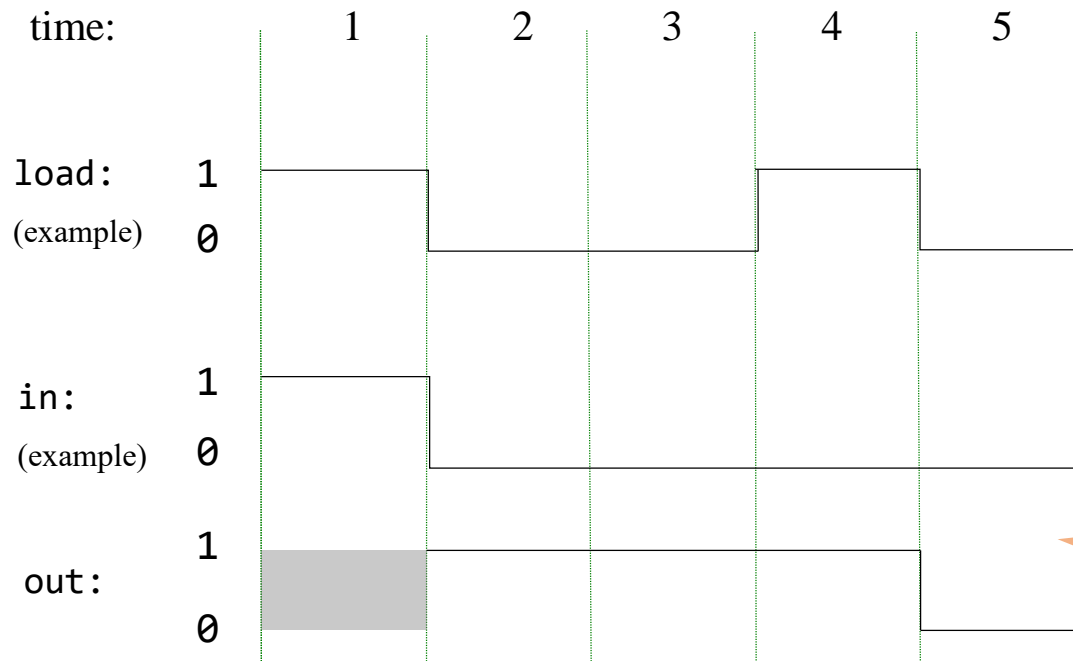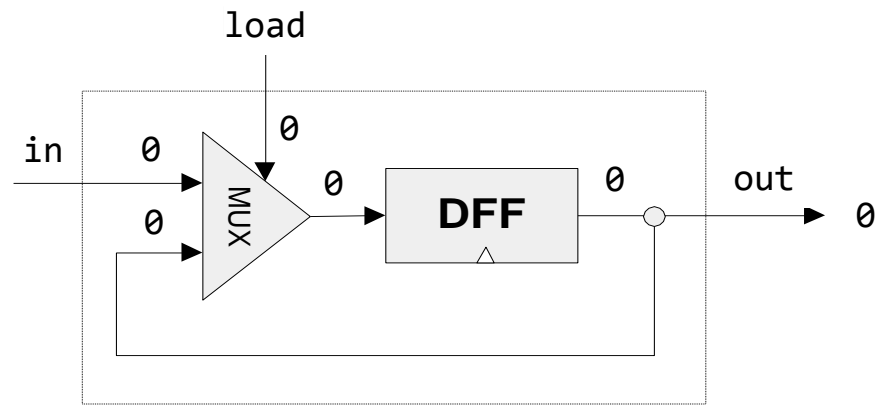
Resulting behavior: Stores and emits a value, until instructed to load (and store) a new value

Chapter 3: Memory

✓ Time matters

✓ Sequential logic

✓ Flip Flops

➡ Memory units

- Counters

- Project 3 overview

Memory units

We'll describe (and build) a progression of memory units:

✓ 1-bit register:
Designed to store a single bit

Memory units

We'll describe (and build) a progression of memory units:

✓ 1-bit register:
Designed to store a single bit

➡ Multi-bit register:
Designed to store, say, a 16-bit value

- Random Access Memory (RAM):
Designed to store n addressable 16-bit values,
each having a unique *index*, or *address*, ranging from 0 to n-1.

Multi-bit register (also known as "register")
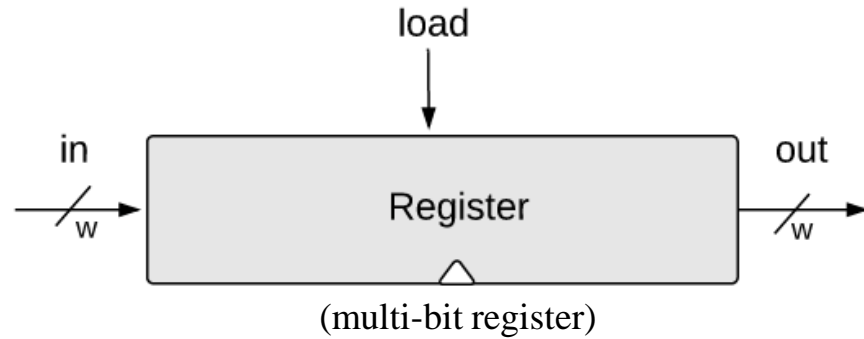
(multi-bit register)

Word width (*w*):

- 16-bit, 32-bit, 64-bit, …

- We will focus on 16-bit registers, without loss of generality.

Register: abstraction
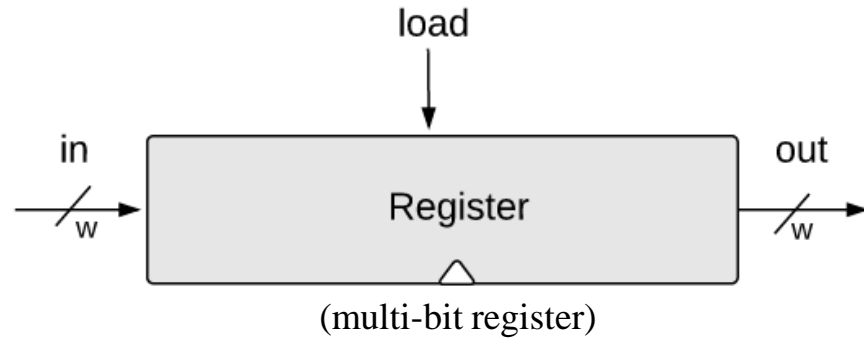
(multi-bit register)

To read a Register:

probe out

Result:
out emits the Register's state

To set Register = v

set in = v

set load = 1

Result:
- The Register's state becomes v;
- From the next cycle onward, out emits v

Register: implementation

(multi-bit register)

Register: implementation
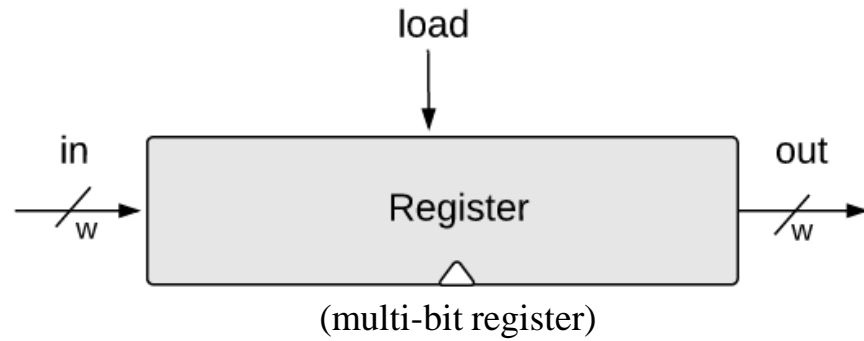
(multi-bit register)
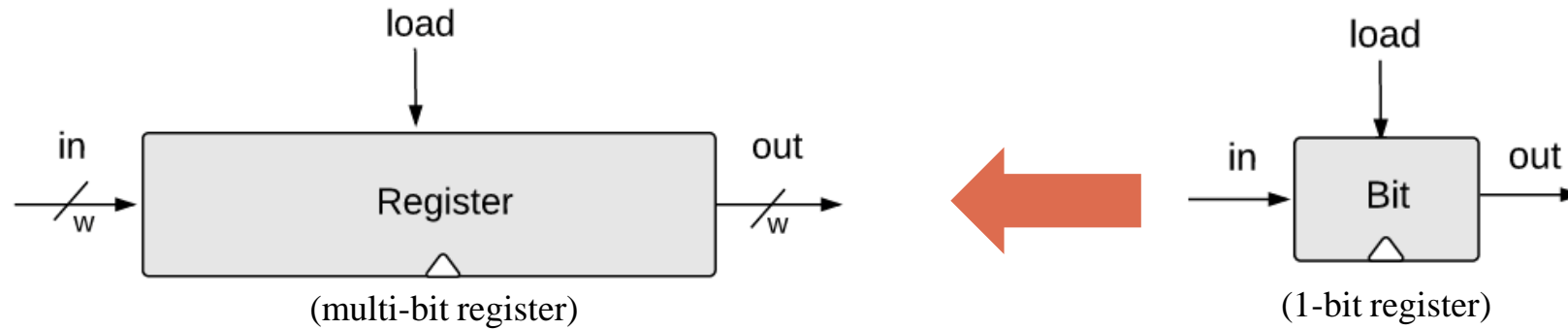
(1-bit register)

A *w*-bit register can be created from an array of *w* 1-bit registers.

Register chip in action

Register chip in action

Register chip in action

Run the clock

Set in to 17
Set load to 1

Inspect the register's output

Inspect the register's contents

D: 17

Register chip in action

Run the clock

Inspect the register's output

Inspect the register's contents

| Chip Nam... | DRegister (Clocked) | Time : | 13 |

Input pins

Name	Value
in[16]	17
load	1

Output pins

Name	Value
out[16]	17

D: 17

HDL

```
* This built-in chip implementa
* providing a GUI representatio
* called "D register" (typicall
*/

CHIP DRegister {

    IN  in[16], load;
    OUT out[16];

    BUILTIN DRegister;
    CLOCKED in, load;
}
```

Format: D...
View: Scr...

Program flow

Slow   Fast

Memory units

We'll describe (and build) a progression of memory units:

✓ 1-bit register:

✓ Multi-bit register:

➡ Random Access Memory (RAM)

RAM
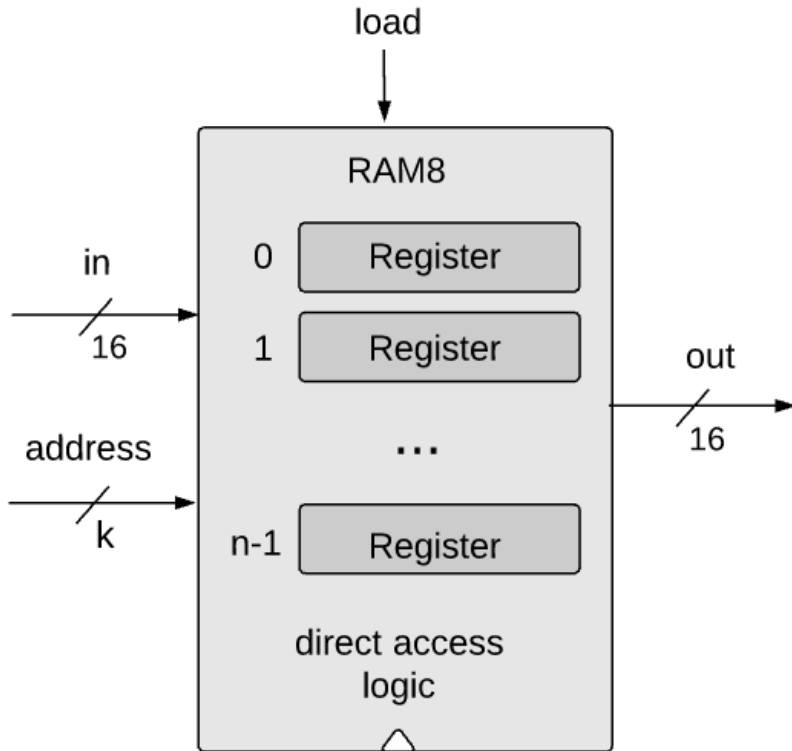
Architecture:

A sequence of n addressable registers, with addresses 0 to n-1

Address width:

$k = log_2n$

Word width:

No impact on the RAM logic

(Hack computer: $w = 16$]

RAM: abstraction
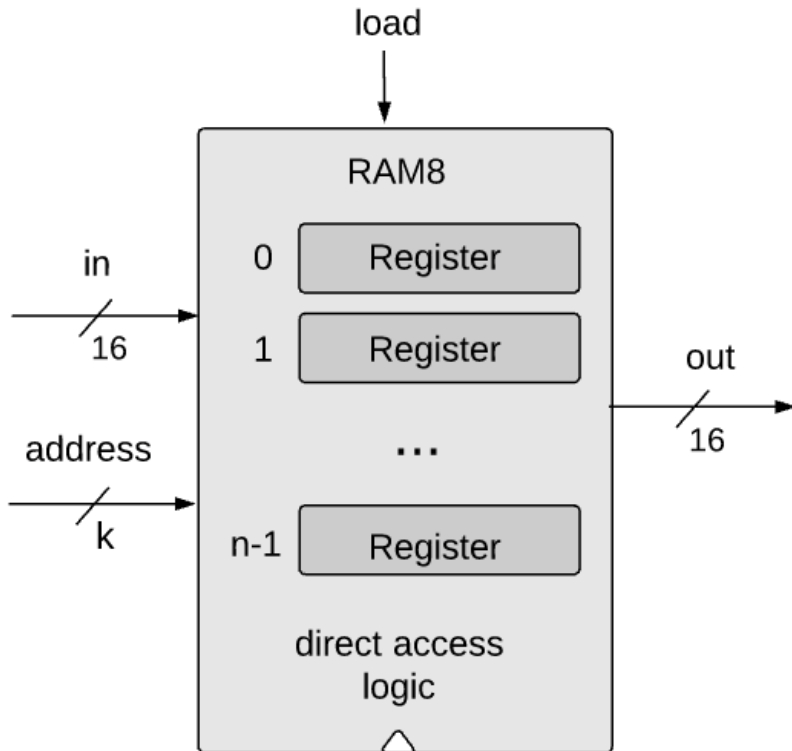
At any given point of time:

- *one* register in the RAM is selected

- all the other registers are irrelevant

To read `Register` *i* :

set `address` = *i*

probe `out`

Result:
`out` emits the value of `Register` *i*

RAM: abstraction
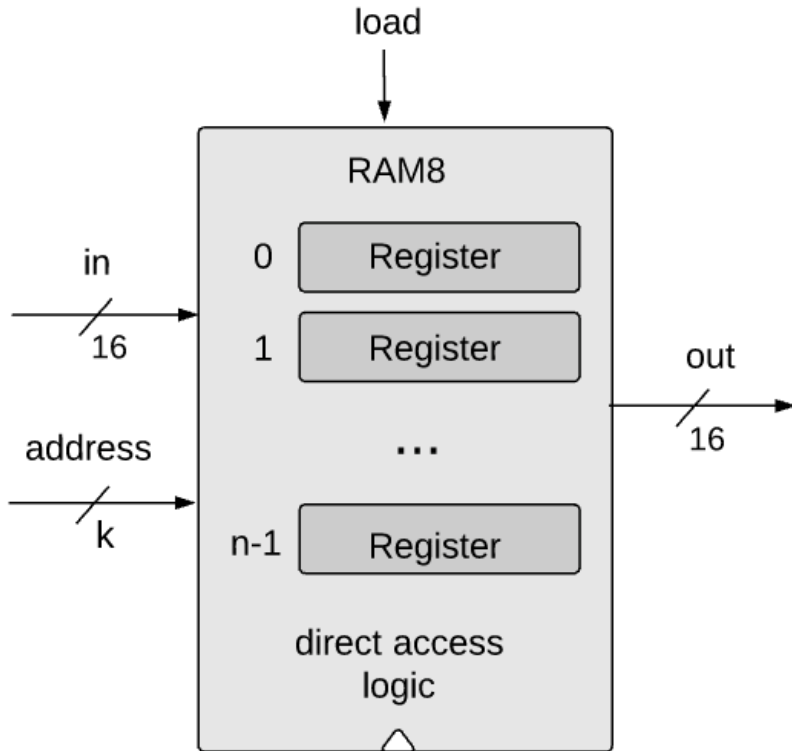
load

RAM8

0	Register
1	Register
	...
n-1	Register

in
/16

address
/k

out
/16

direct access logic

At any given point of time:

❑ *one* register in the RAM is selected

❑ all the other registers are irrelevant

To set Register *i* to *v* :

set address $= i$

set in $= v$

set load $= 1$

Result:
- The state of Register *i* becomes *v*
- From the next cycle onward, out emits *v*

RAM: abstraction
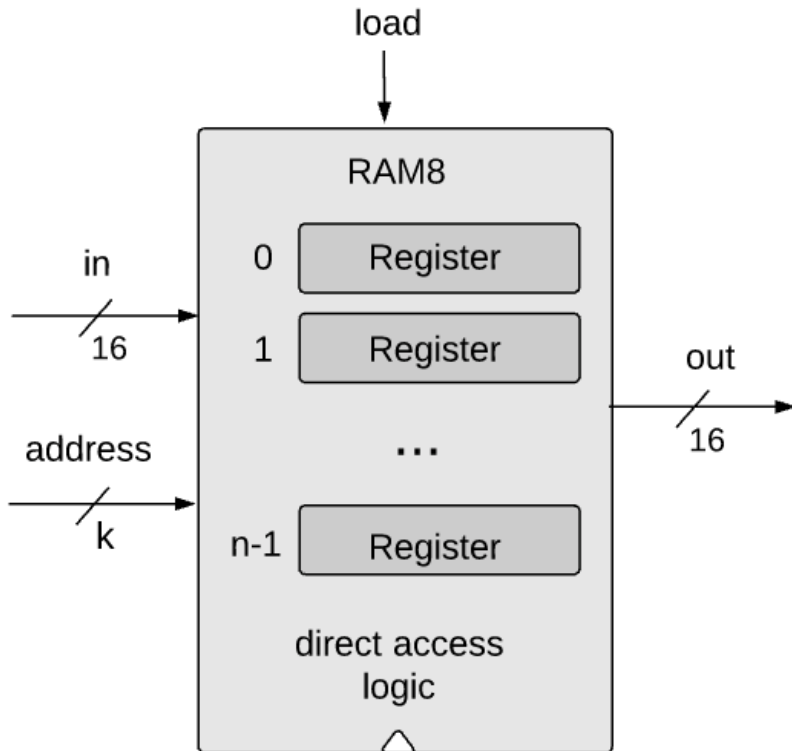
Why "Random Access Memory"?

Irrespective of the RAM size (*n*),
every randomly selected register can be accessed "instantaneously",
at more or less the same time.

A family of 16-bit RAM chips
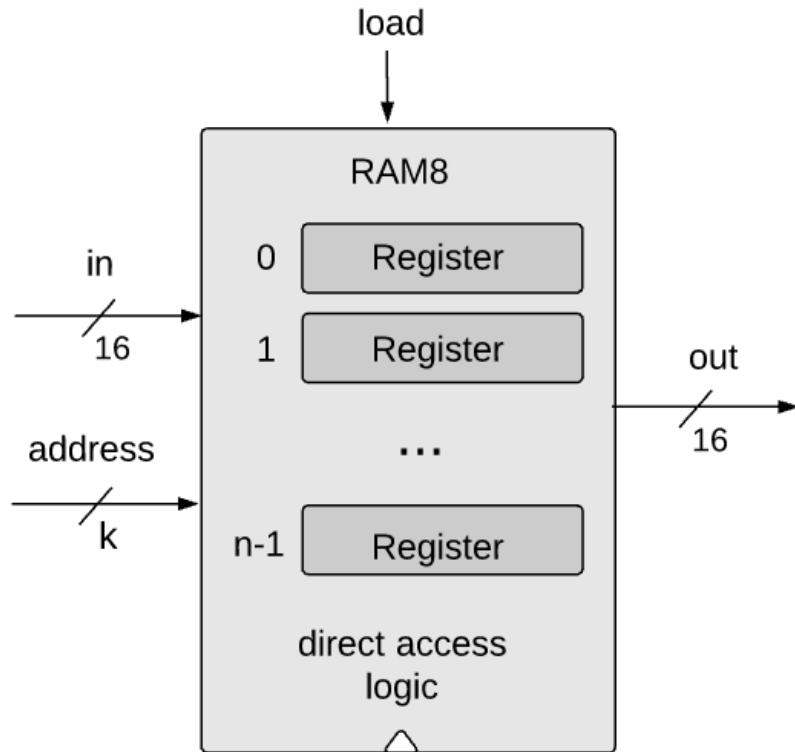
chip name	n	k
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

Why these particular RAM chips?

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

RAM chip in action

Chapter 3: Memory

✓ Time matters

✓ Sequential logic

✓ Flip Flops

✓ Memory units

➡ Counters

- Project 3 overview

Where counters come to play

- The computer must keep track of which instruction should be fetched and executed next

- This control mechanism can be realized by a register called `Program Counter`

- The `PC` contains the address of the instruction that will be fetched and executed next

- The PC is designed to support three possible control operations:

Reset: fetch the first instruction

```
PC = 0
```

Next: fetch the next instruction

```
PC++
```

Goto: fetch instruction n

```
PC = n
```

Program Counter
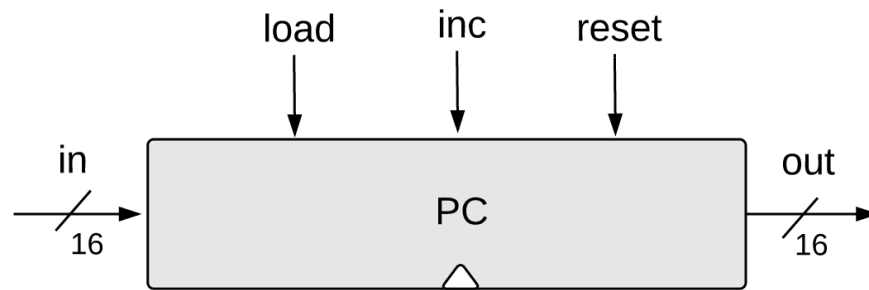

```
/**
 * A 16-bit counter with load and reset control bits.
```

Program Counter
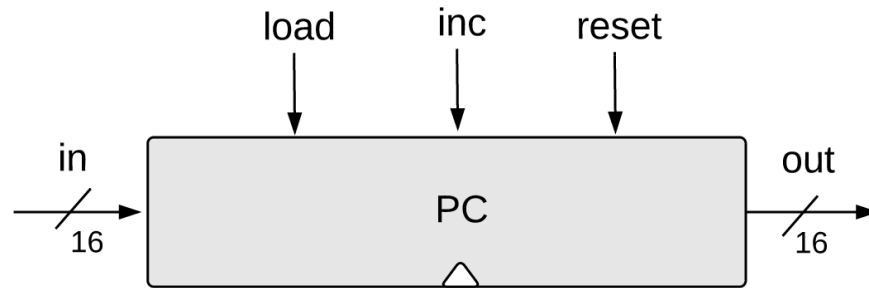

```
/**
 * A 16-bit counter with load and reset control bits.
 * if      reset(t) out(t+1) = 0
 * else if load(t)  out(t+1) = in(t)
 * else if inc(t)   out(t+1) = out(t) + 1  (integer addition)
 * else             out(t+1) = out(t)
 */

CHIP PC {
    IN in[16],load,inc,reset;
    OUT out[16];

    PARTS:
    // Put your code here:
}
```

Counter chip in action

HW Simulator

PC chip demo

Chapter 3: Memory

✓ Time matters

✓ Sequential logic

✓ Flip Flops

✓ Memory units

✓ Counters

➡ Project 3 overview

Project 3

Given:

- ❑ All the chips built in Projects 1 and 2
- ❑ Flip-Flop (built-in `DFF` gate)

Goal: Build the following chips:

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

A family of sequential chips, from a 1-bit register to a 16K RAM unit.

1-bit register
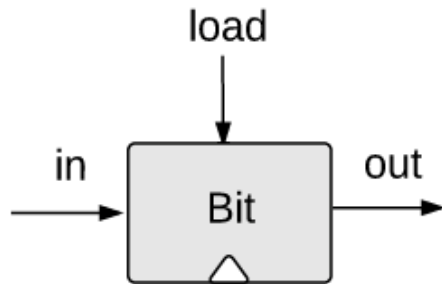
`Bit.hdl`

```
/**
 * 1-bit register:
 * If load(t) then out(t+1) = in(t)
 * else              out(t+1) = out(t))
 */

CHIP Bit {
    IN in, load;
    OUT out;

    PARTS:
    // Put your code here:
}
```

Implementation tip:

Can be built from a `DFF` and a multiplexor.

16-bit Register
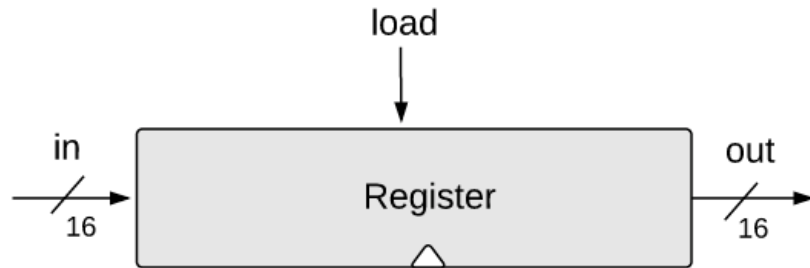
Register.hdl

```
/**
 * 16-bit register:
 * If load(t) then out(t+1) = in(t)
 * else            out(t+1) = out(t))
 */

CHIP Register {
    IN in[16], load;
    OUT out[16];

    PARTS:
    // Put your code here:
}
```

Implementation tip:

Can be built from an array of sixteen 1-bit registers.

8-Register RAM

load

RAM8

in → 16	0 Register
	1 Register → out 16
address → k	...
	7 Register

direct access logic
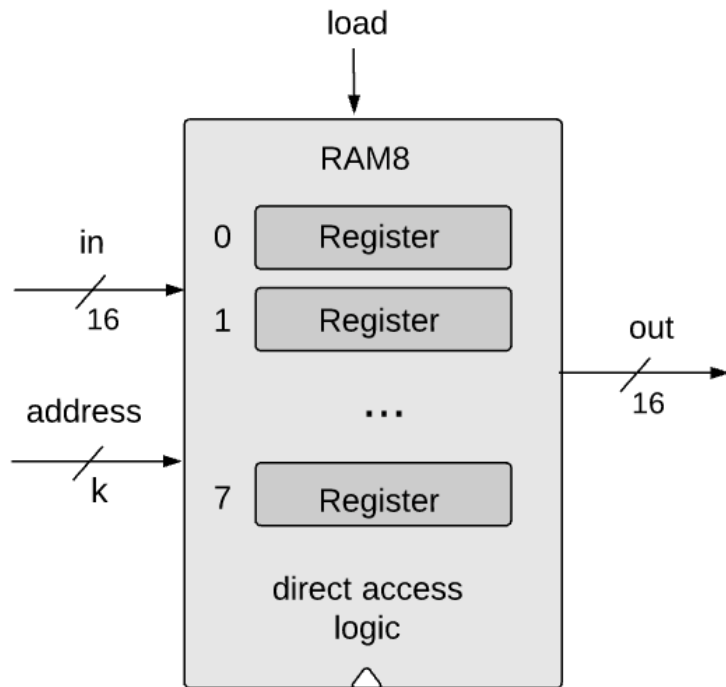
RAM8.hdl

```
/*
 * Let M stand for the state of the
 * register selected by address.
 * if load(t) then {M=in(t), out(t+1)=M}
 * else                      out(t+1)=M
 */

CHIP RAM8 {
    IN in[16], load, address[3];
    OUT out[16];

    PARTS:
    // Put your code here:
}
```

Implementation tips:

❑ Feed the `in` value to all the registers, simultaneously

❑ Use mux / demux chips to select the register specified by `address`.

Project 3

Given:

- ❑ All the chips built in Projects 1 and 2
- ❑ Flip-Flop (`DFF` gate)

Goal: Build the following chips:

✓ ❑ Bit
✓ ❑ Register
✓ ❑ RAM8
   ❑ RAM64 ⎫
   ❑ RAM512 ⎬ Our next task
   ❑ RAM4K
   ❑ RAM16K ⎭
   ❑ PC

RAM8, RAM64, … RAM16K

RAM512

RAM64

RAM8

Same technique can be used to implement RAM4K and RAM16K

Implementation tips

- A RAM unit can be built by grouping smaller RAM-parts together
- Think about the RAM's address input as consisting of two fields:
  - one field can be used to select a RAM-part;
  - the other field can be used to select a register within that RAM-part
- Use mux/demux logic to effect this addressing scheme.

Project 3

Given:

- ❑ All the chips built in Projects 1 and 2
- ❑ Flip-Flop (`DFF` gate)

Goal: Build the following chips:
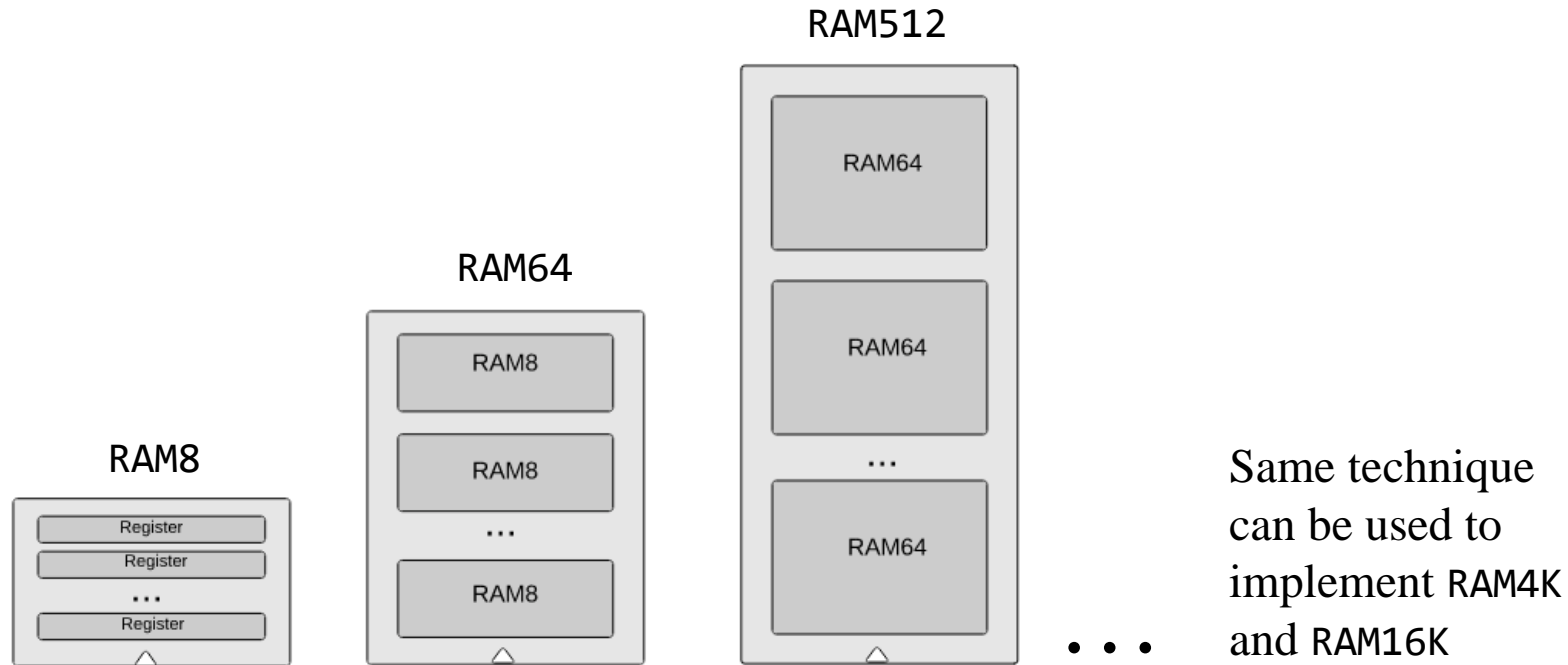
✓ ❑ Bit
✓ ❑ Register
✓ ❑ RAM8
✓ ❑ RAM64
✓ ❑ RAM512
✓ ❑ RAM4K
✓ ❑ RAM16K
❑ PC

Program Counter
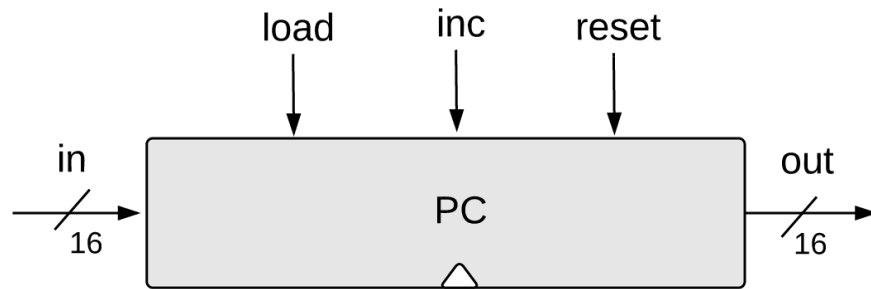
Implementation tip:

Can be built from a register, an incrementor, and some logic gates.

```
/**
 * A 16-bit counter with load, inc, and reset control bits.
 *
 * if reset(t)     out(t+1)=0        // resetting: counter = 0
 * else if load(t) out(t+1)=in(t)    // setting counter = value
 * else if inc(t)  out(t+1)=out(t)+1 // incrementing: counter++
 * else            out(t+1)=out(t)   // counter does not change
 */

CHIP PC {
    IN in[16], load, inc, reset;
    OUT out[16];

    PARTS:
    // Implementation comes here.
}
```

Project 3 resources

From NAND to Tetris
Building a Modern Computer From First Principles

www.nand2tetris.org

- Home
- Prerequisites
- Syllabus
- **Course**
- Book
- Software
- Terms
- Papers
- Talks
- Cool Stuff
- About
- Team
- Q&A

Project 3: Sequential Chips

Background

The computer's main memory, also called *Random Access Memory*, or *RAM*, is an addressable sequence of *n*-bit registers, each designed to hold an *n*-bit value. In this project you will gradually build a RAM unit. This involves two main issues: (i) how to use gate logic to store bits persistently, over time, and (ii) how to use gate logic to locate ("address") the memory register on which we wish to operate.

Objective

Build all the chips described in Chapter 3 (see list below), leading up to a *Random Access Memory* (RAM) unit. The only building blocks that you can use are primitive DFF gates, chips that you will build on top of them, and chips described in previous chapters.

Chips

Chip (HDL)	Description	Test script	Compare file
DFF	Data Flip-Flop (primitive)		
Bit	1-bit register	Bit.tst	Bit.cmp
Register	16-bit register	Register.tst	Register.cmp
RAM8	16-bit / 8-register memory	RAM8.tst	RAM8.cmp
RAM64	16-bit / 64-register memory	RAM64.tst	RAM64.cmp
RAM512	16-bit / 512-register memory	RAM512.tst	RAM512.cmp
RAM4K	16-bit / 4096-register memory	RAM4K.tst	RAM4K.cmp
RAM16K	16-bit / 16384-register memory	RAM16K.tst	RAM16K.cmp
PC	16-bit program counter	PC.tst	PC.cmp

All the necessary project 3 files are available in:
nand2tetris / projects / 03

More resources

- HDL Survival Guide

- Hardware Simulator Tutorial
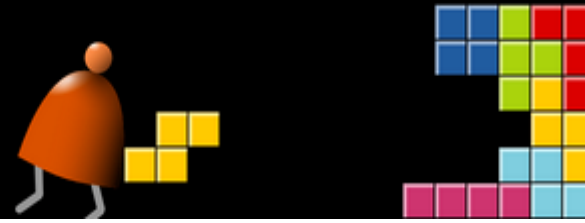
- nand2tetris Q&A forum

All available in: `www.nand2tetris.org`

Best practice advice

- Try to implement the chips in the given order

- If you don't implement some of the chips required in project 3, you can still use them as chip-parts in other chips. Just rename the given stub-files; this will cause the simulator to use the built-in versions of these chips

- You can invent new, "helper chips"; however, this is not required: you can build any chip using previously-built chips only

- Strive to use as few chip-parts as possible.

- You will have to use chips from Projects 1 and 2

- Best practice: use their built-in versions

- For technical reasons, the HDL files of this project are organized in two directories named a and b

- This directory structure should remain as is.

Chapter 3: Memory

✓ Time matters

✓ Sequential logic

✓ Flip Flops

✓ Memory units

✓ Counters

✓ Project 3 overview

Chapter 3

Memory

These slides support chapter 3 of the book

The Elements of Computing Systems

By Noam Nisan and Shimon Schocken

MIT Press