Chapter 3

Memory

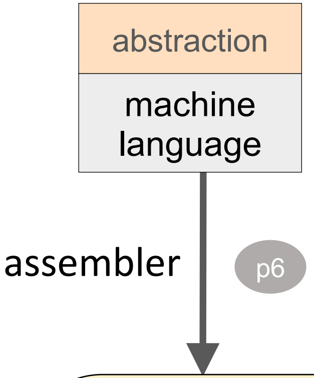
These slides support chapter 3 of the book

The Elements of Computing Systems

By Noam Nisan and Shimon Schocken

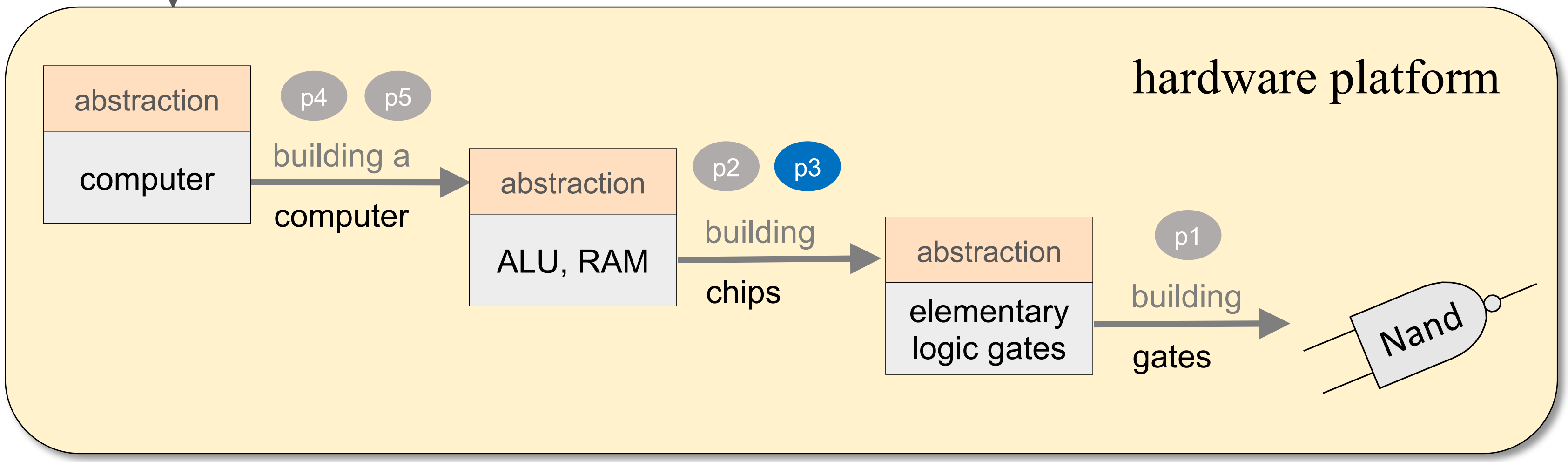
MIT Press, 2021

Nand to Tetris Roadmap: Hardware



<u>This lecture / chapter / project:</u>

Build the computer's memory system



<u>Project 1</u>: Build basic logic gates

<u>Project 2</u>: Build the ALU

A common theme in computer science

- We articulate a simple model (the simpler, the better)
- We explore the model's power:
  - What it can do
  - What it cannot do
- We then extend the model, making it more powerful

<u>Case in point:</u>

Logic gates.

Logic gates

Model: And, Or, Not, …

- Simple, and powerful:

  Logic gates can realize any Boolean function, and can be combined to form very useful artifacts, like an ALU

- But, as a *practical model of computation*, logic gates fall short

Some limitations

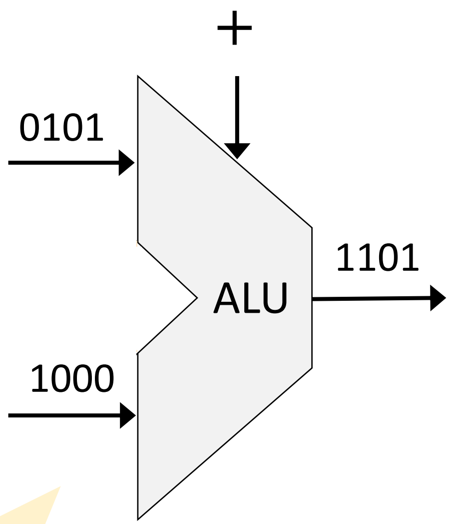
- Logic gates cannot store information (bits) over time
- Feedback loops are not allowed: A chip's output cannot serve as its input
- Logic gates can handle only inputs of a fixed size.
  For example, we can build an Or3 gate, and an Or4 gate, and so on, but we cannot build a single gate that computes Or for any given number of inputs

Extension

Allow logic gates to be sensitive to the progression of *time*.

Time-independent logic

- So far we ignored *time*
- The chip's inputs were just "sitting there" – fixed and unchanging
- The chip's output was a function ("combination") of the current inputs, and the current inputs only
- This style of gate logic is called:
  - *time-independent logic*
  - *combinational logic*
- All the chips that we discussed and developed so far were combinational



ALU: The "topmost" combinational chip

Hello, time

Software needs:

- The hardware must be able to remember things, over time:

Example (variables):

```
x = 17
```

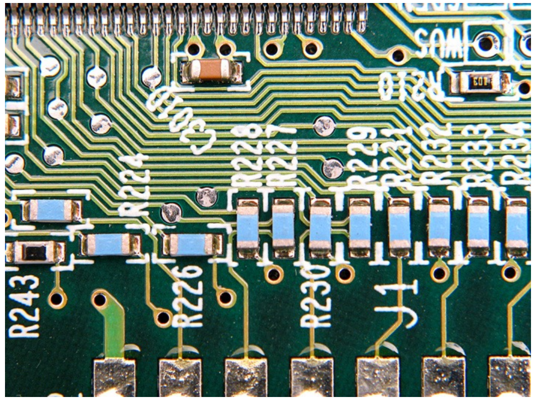
- The hardware must be able to do things, one at a time (sequentially):

Example (iteration):

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

Hardware needs:

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



Hello, time

Software needs:

- The hardware must be able to remember things, over time:

Example (variables):

```
x = 17
```

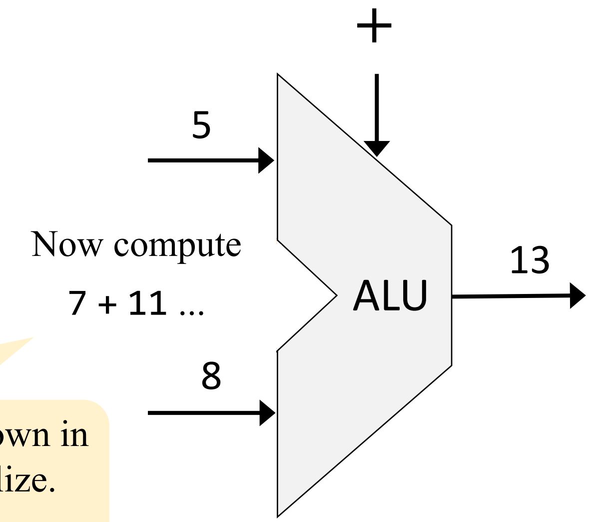
- The hardware must be able to do things, one at a time (sequentially):

Example (iteration):

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

Hardware needs:

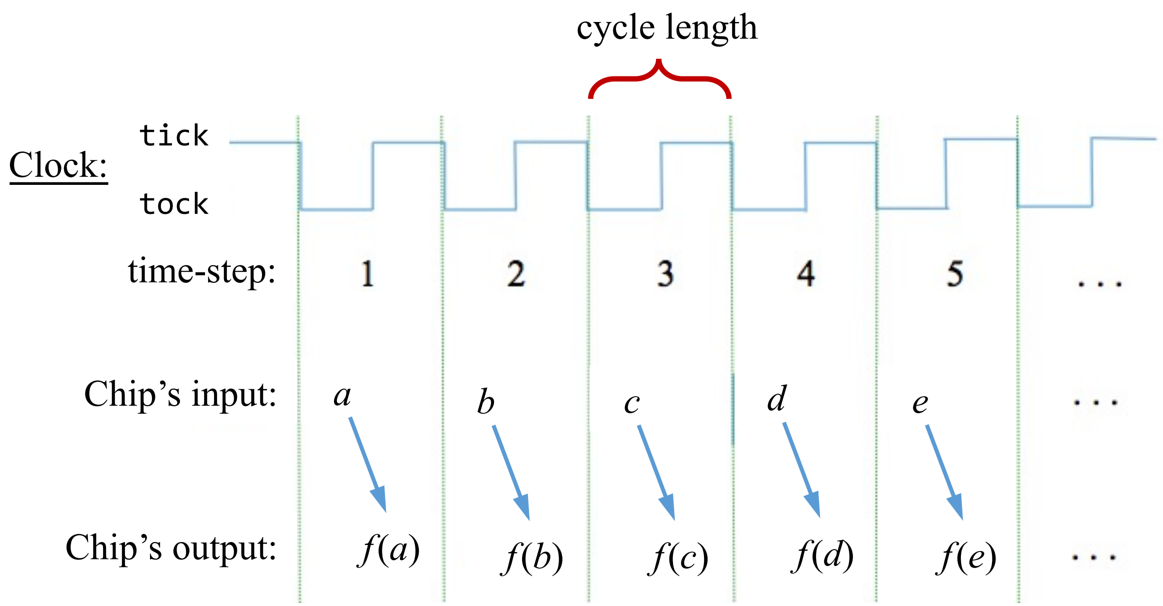
- The hardware must handle the *physical time delays* associated with *computing* and *moving* data from one chip to another.



It will take some 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

<u>Solution</u>: We can neutralize the time delays if we decide to use *discrete time*



The clock also enables the implementation of chips that behave like <u>memory devices</u>.

- Set the *cycle length* to be slightly > than the maximum time delay, and…
- Decide to use the chips' outputs only at the end of cycles (time-steps), ignoring what happens within cycles
- Details later.

Memory

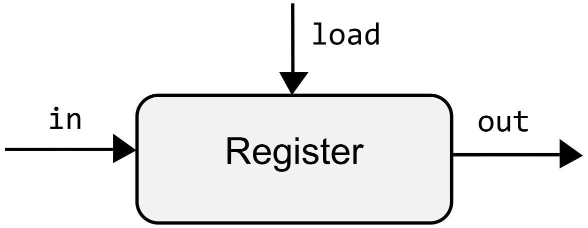
Memory: The faculty of the brain by which data or information is encoded, stored, and retrieved when needed.

It is the *retention of information over time* for the purpose of influencing future action (Wikipedia)

Memory is time-based:

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

Memory



Basic operations:

- "Loading" a value
- "Storing" a value

1 2 3 4 5 6 7 ... 100 101 102 103 104 105 106 ... time

(cycles)

x = 17 , 17, 17, 17, 17, 17, 17, ... ,

loading storing

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

loading storing

The challenge: Building chips that realize this functionality,

Chips that can *change and maintain state*.

Chapter 3: Memory

<u>Abstraction</u>

- Representing time
- Clock
- Registers
- RAM
- Counters

<u>Implementation</u>

- Data Flip Flop
- Registers
- RAM
- Project 3: Chips
- Project 3: Guidelines

Chapter 3: Memory

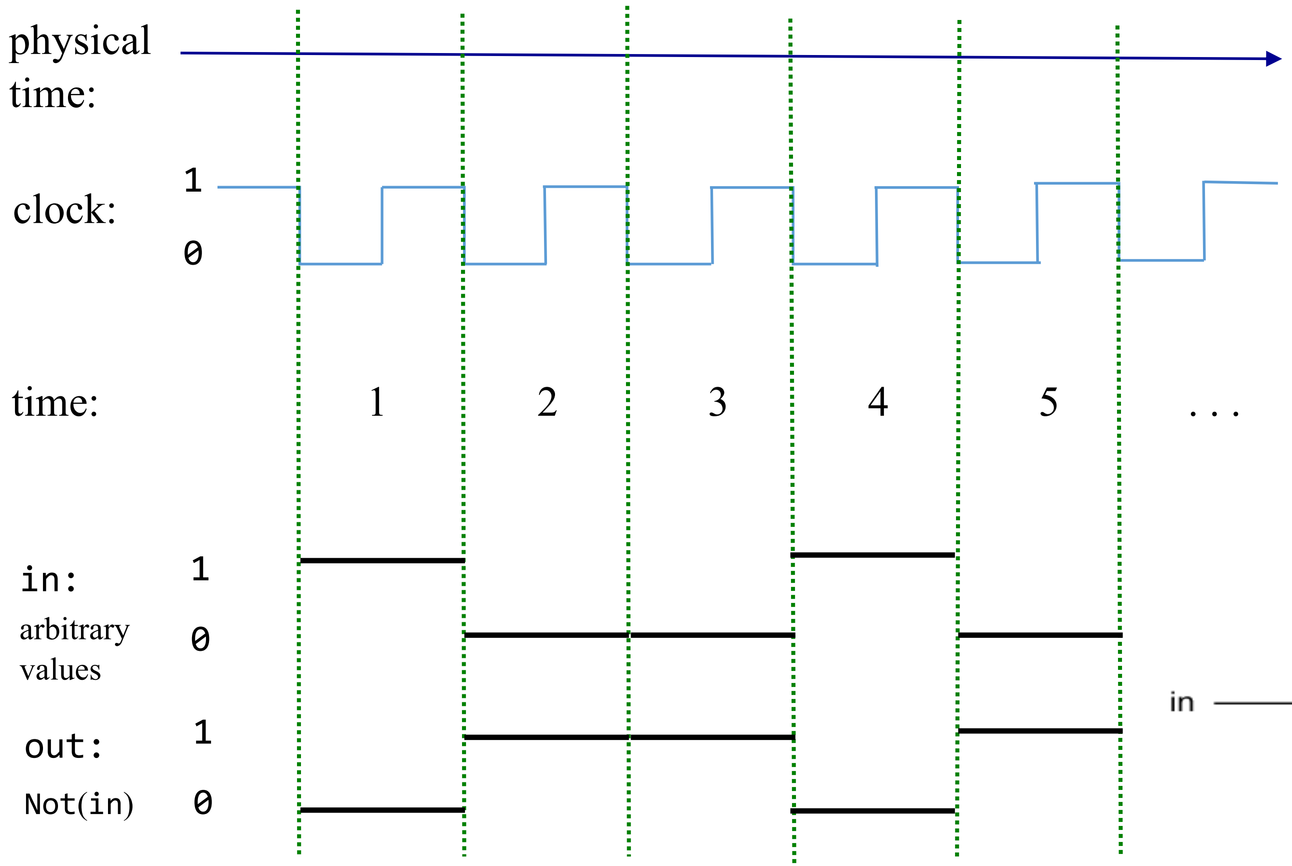
<u>Abstraction</u>

- Representing time
- Clock
- Registers
- RAM
- Counters

<u>Implementation</u>

- Data Flip Flop
- Registers
- RAM
- Project 3: Chips
- Project 3: Guidelines

Chip behavior over time (example: Not gate)



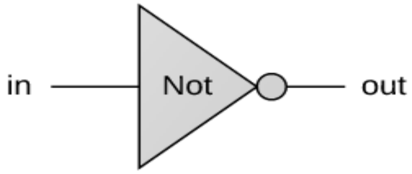
Arrow of time:

Continuous

Discrete time

Design decision:

Track state changes only when advancing from one time-step to another

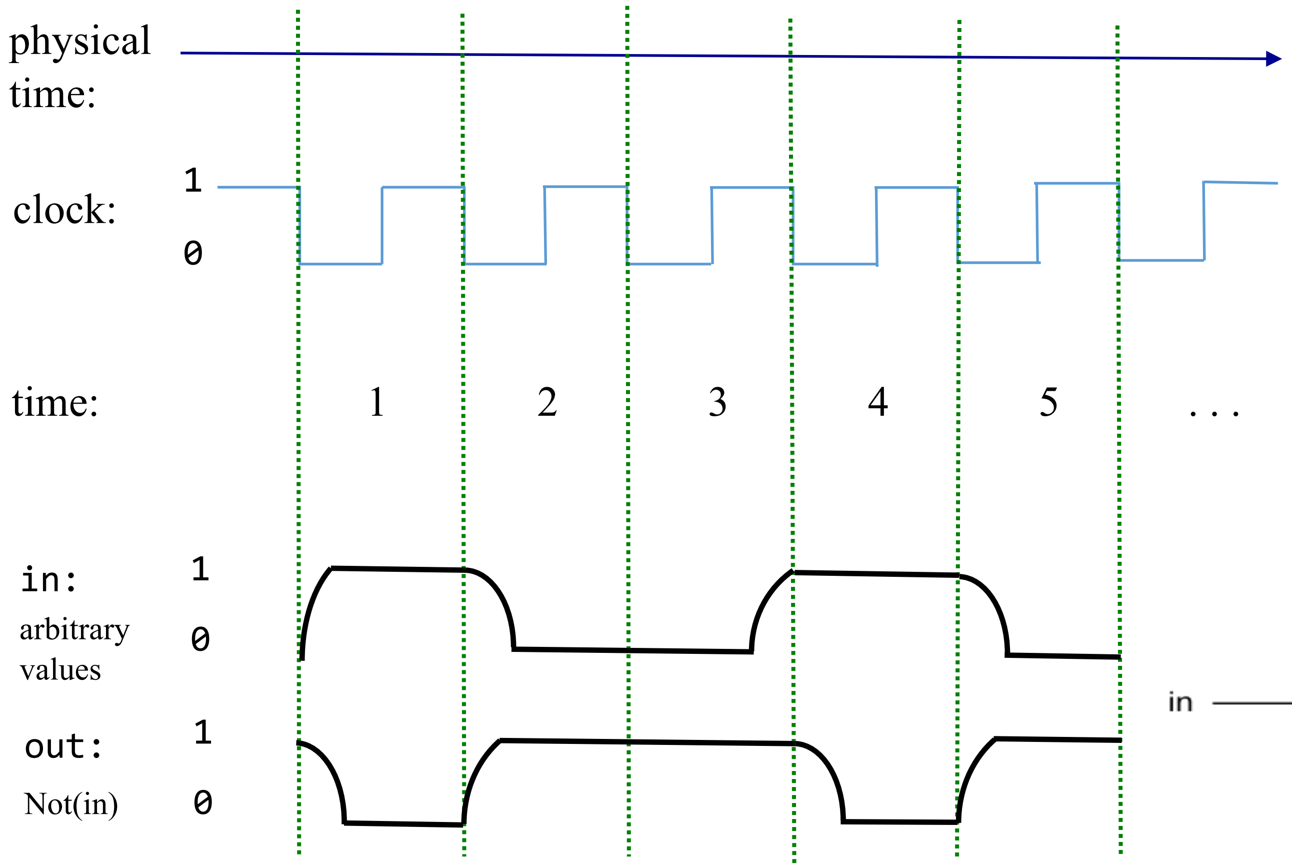


(example)

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)

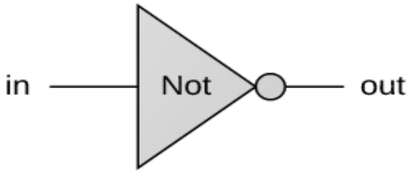


Arrow of time:

Continuous

Discrete time

Design decision:
Track state changes only when advancing from one time-step to another

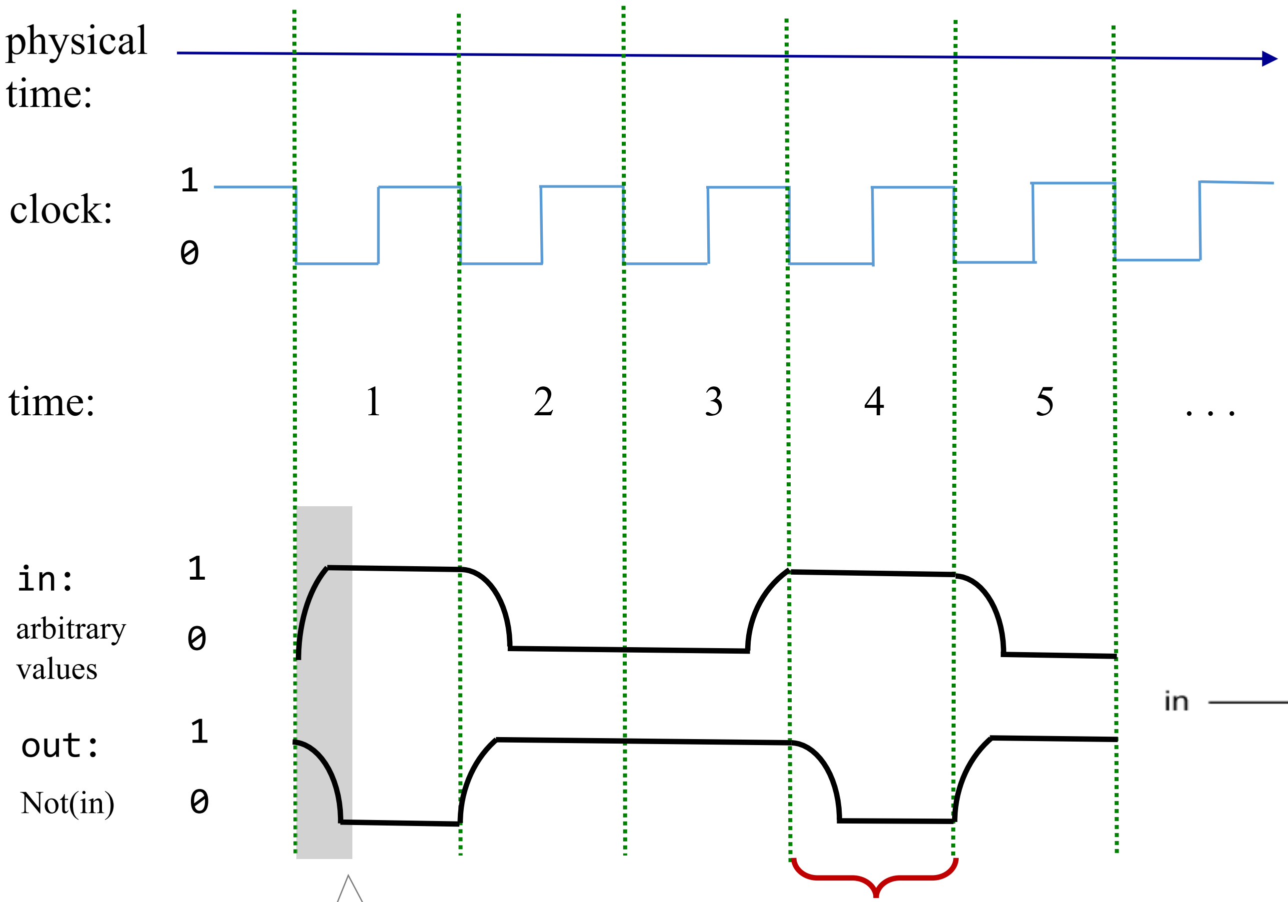


(example)

Actual behavior of the `in` and `out` signals:

Influenced by physical time delays

Chip behavior over time (example: Not gate)



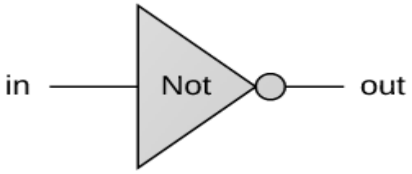
Arrow of time:

Continuous

Discrete time

Design decision:

Track state changes only when advancing from one time-step to another



(example)

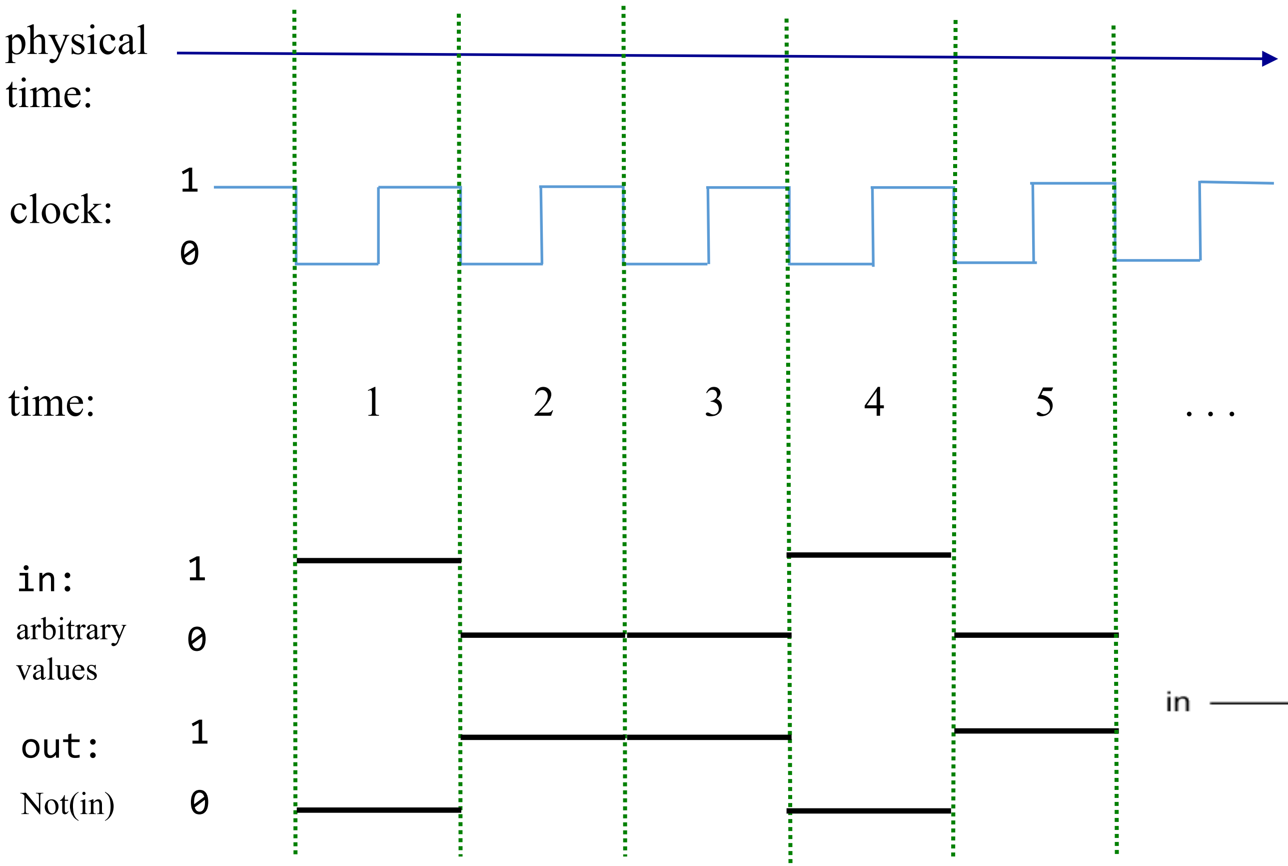
Time delays

- Propagation delays
- Computation delays

Cycle length

- Design parameter
- Set to be slightly > max(time delays)

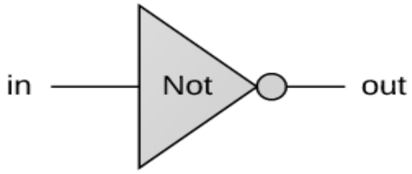
Chip behavior over time (example: Not gate)



Arrow of time:

Continuous

Discrete time
Design decision:
Track state changes only when advancing from one time-step to another



(example)

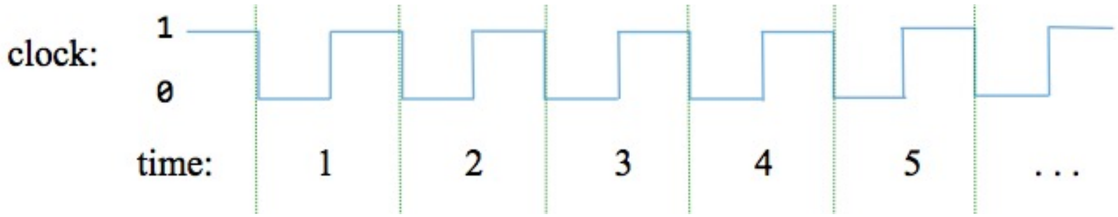
Resulting effect:

- Combinational chips react "immediately" to their inputs
- Facilitated by the decision to track changes only at cycle ends

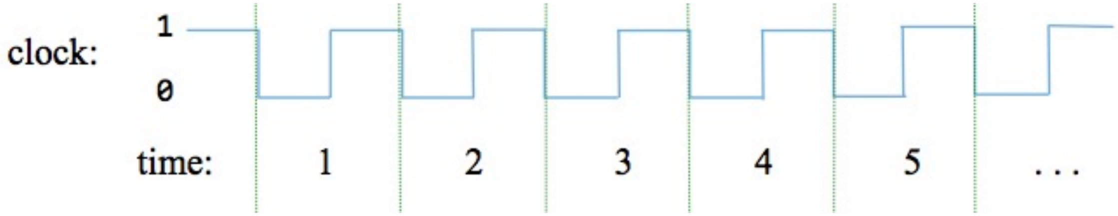
Chapter 3: Memory

Abstraction	Implementation
✓ Representing time	• Data Flip Flop
Clock	• Registers
• Registers	• RAM
• RAM	• Project 3: Chips
• Counters	• Project 3: Guidelines

Clock



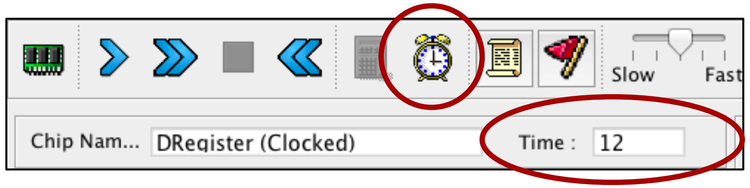
Clock: Simulated implementation



Interactive simulation

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

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

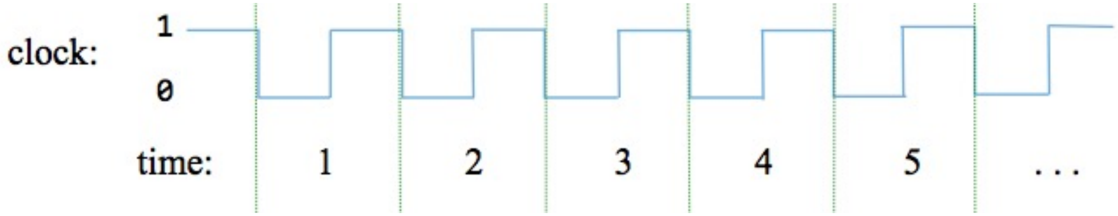


Script-based simulation

"`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,
...
```

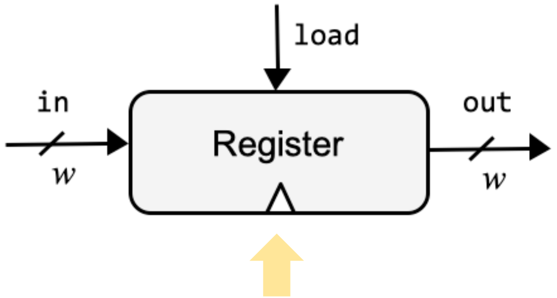
Clock: Physical implementation



Physical clock

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

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



Chip diagram convention:

A triangle icon represents a clock signal input

The oscillator’s output signal is fed to all the time-based (clocked) chips in the computer

Chapter 3: Memory

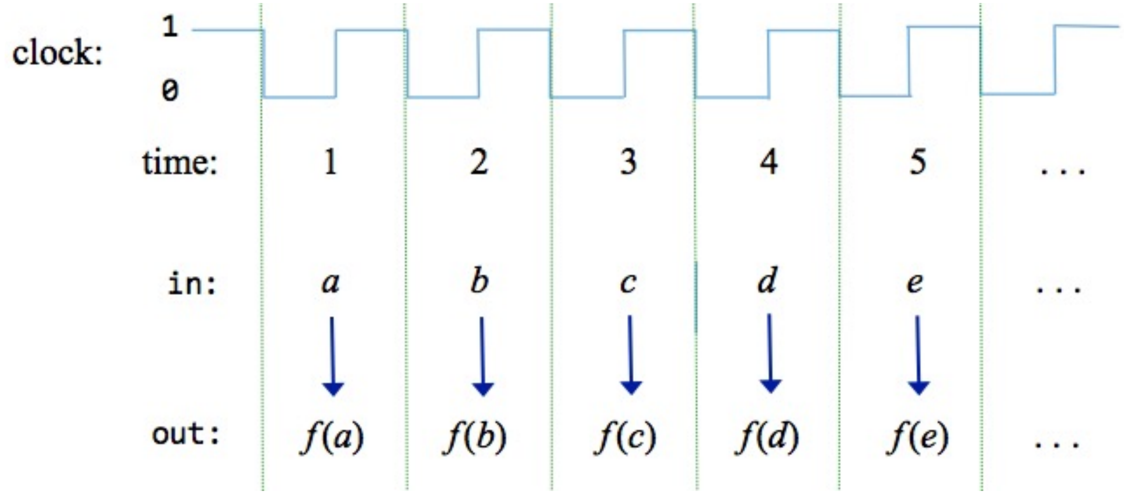
Abstraction	Implementation
✓ Representing time	• Data Flip Flop
✓ Clock	• Registers
➡ Registers	• RAM
• RAM	• Project 3: Chips
• Counters	• Project 3: Guidelines

Combinational logic / Sequential logic

Combinational logic:

The output depends on the current inputs

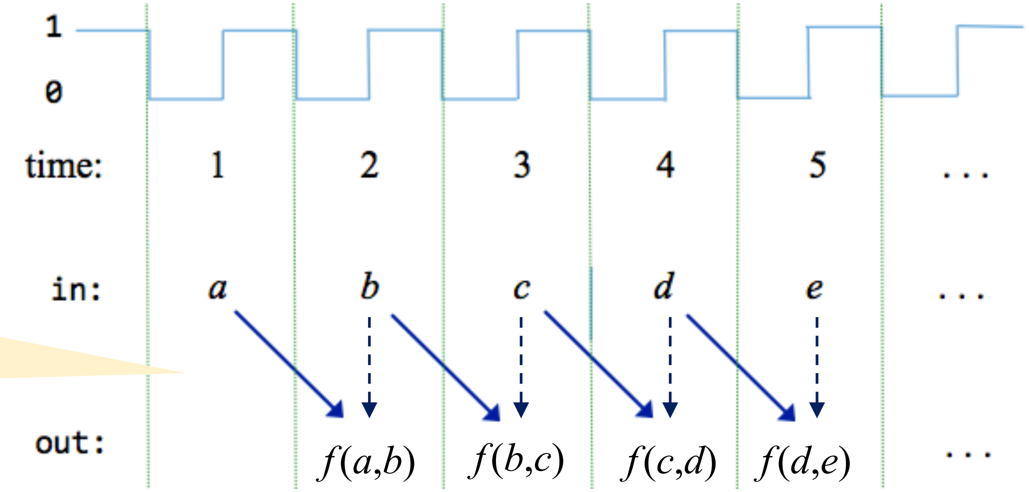
The clock is used to stabilize outputs



Sequential logic:

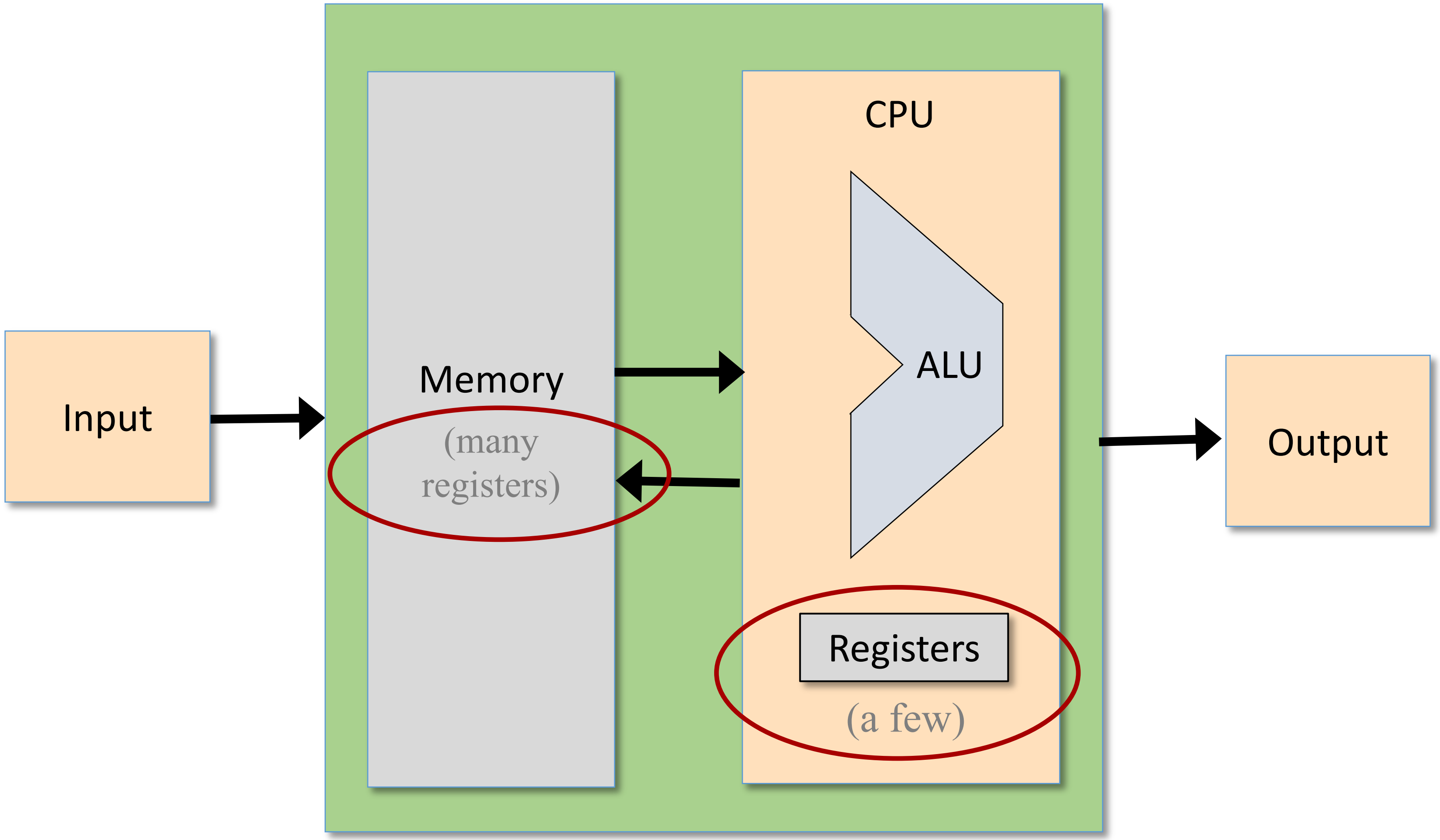
The output depends on:

- Previous inputs
- Current inputs (optionally)



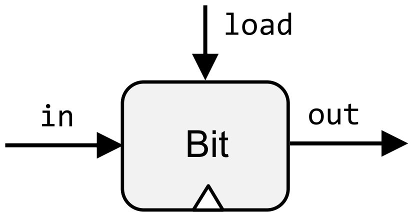
This behavior can be used to build chips designed to maintain state: **Registers**.

Registers

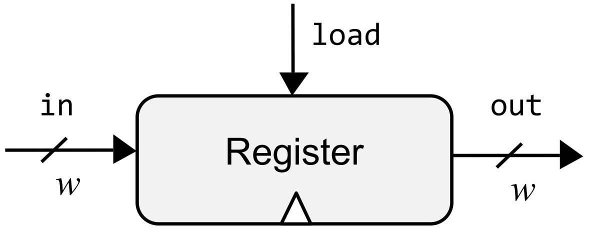


Computer Architecture

Registers



1-Bit register



multi-bit register

<u>Designed to:</u>

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

time:

x = 17, 17, 17, 17, 17, 17, 17, 17, ..., 17

loading

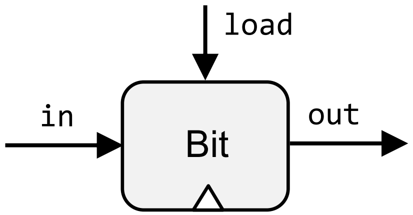
storing

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

loading

storing

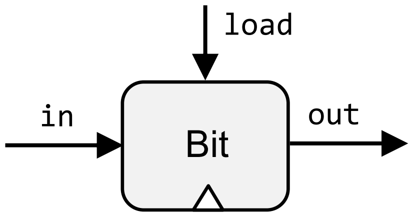
1-Bit register



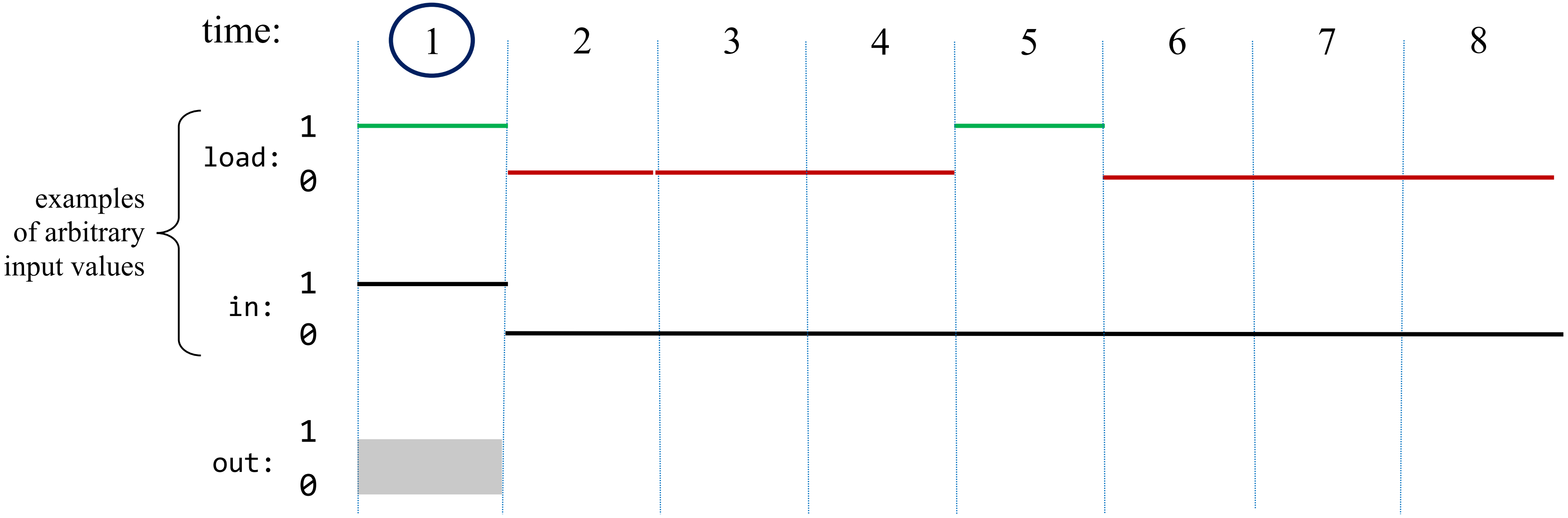
1-Bit register

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

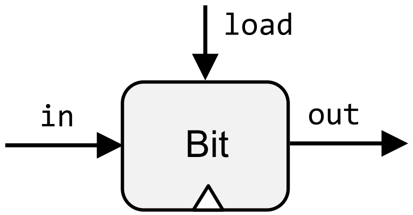
1-Bit register



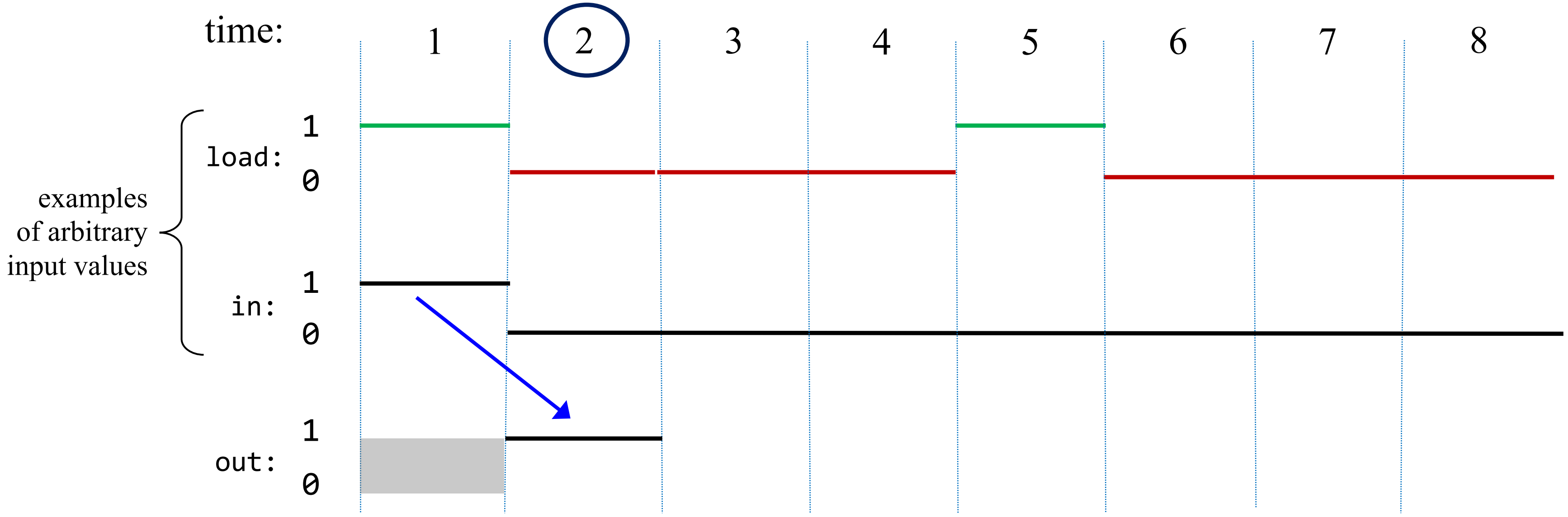
if $\texttt{load}(t)$ then
    $\texttt{out}(t+1) = \texttt{in}(t)$
else
    $\texttt{out}(t+1) = \texttt{out}(t)$



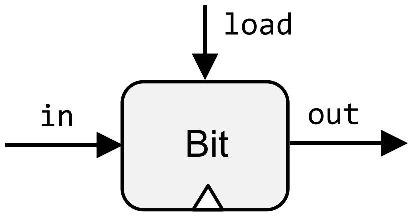
1-Bit register



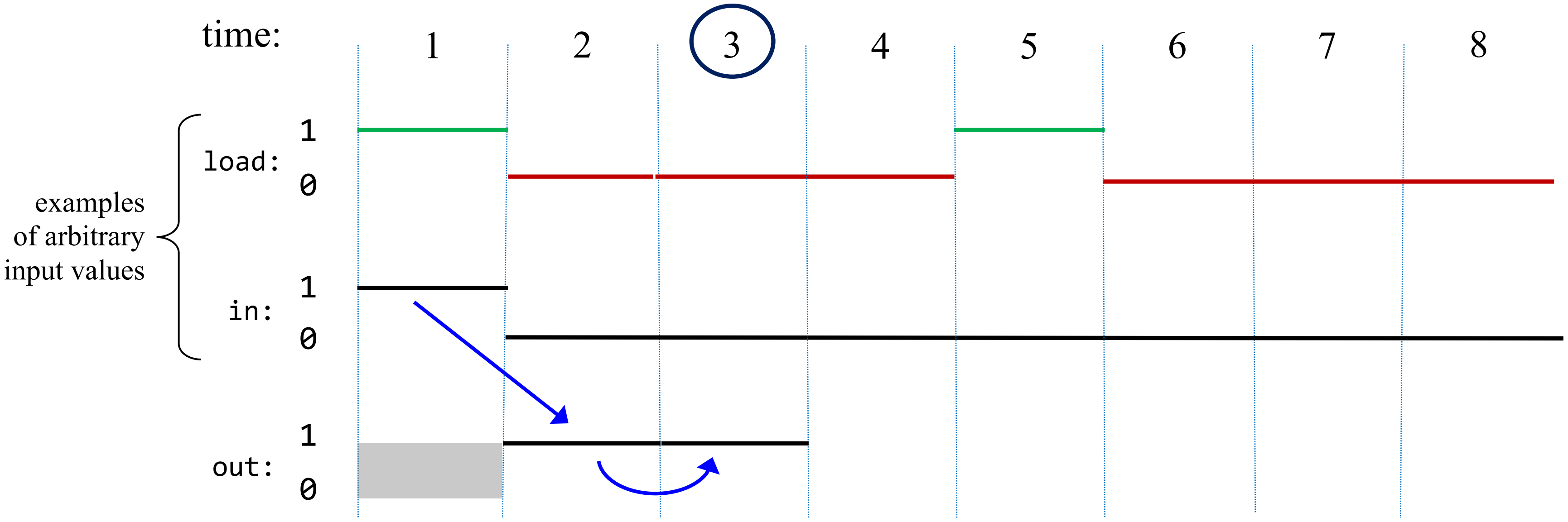
if load(t) then
  out($t+1$) = in(t)
else
  out($t+1$) = out(t)



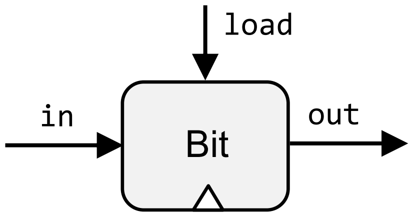
1-Bit register



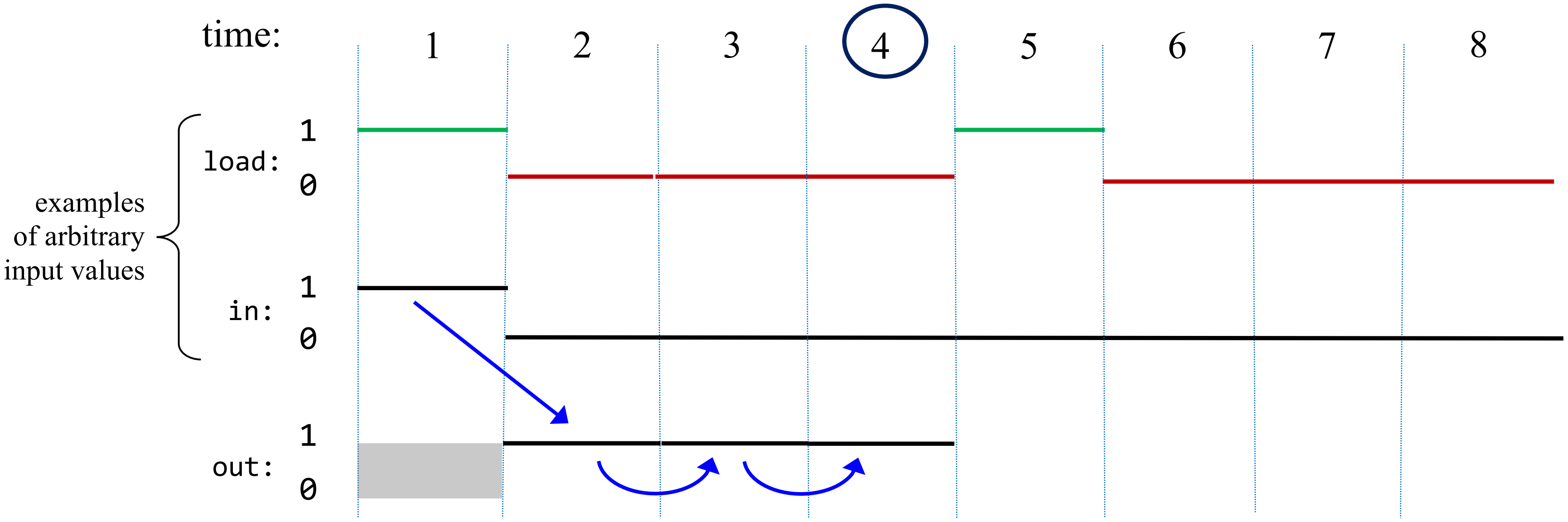
if `load(t)` then
    $\texttt{out}(t+1) = \texttt{in}(t)$
else
    $\texttt{out}(t+1) = \texttt{out}(t)$



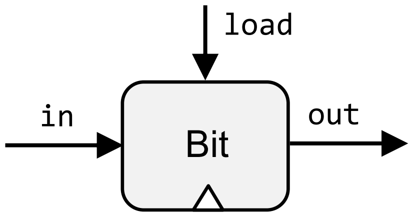
1-Bit register



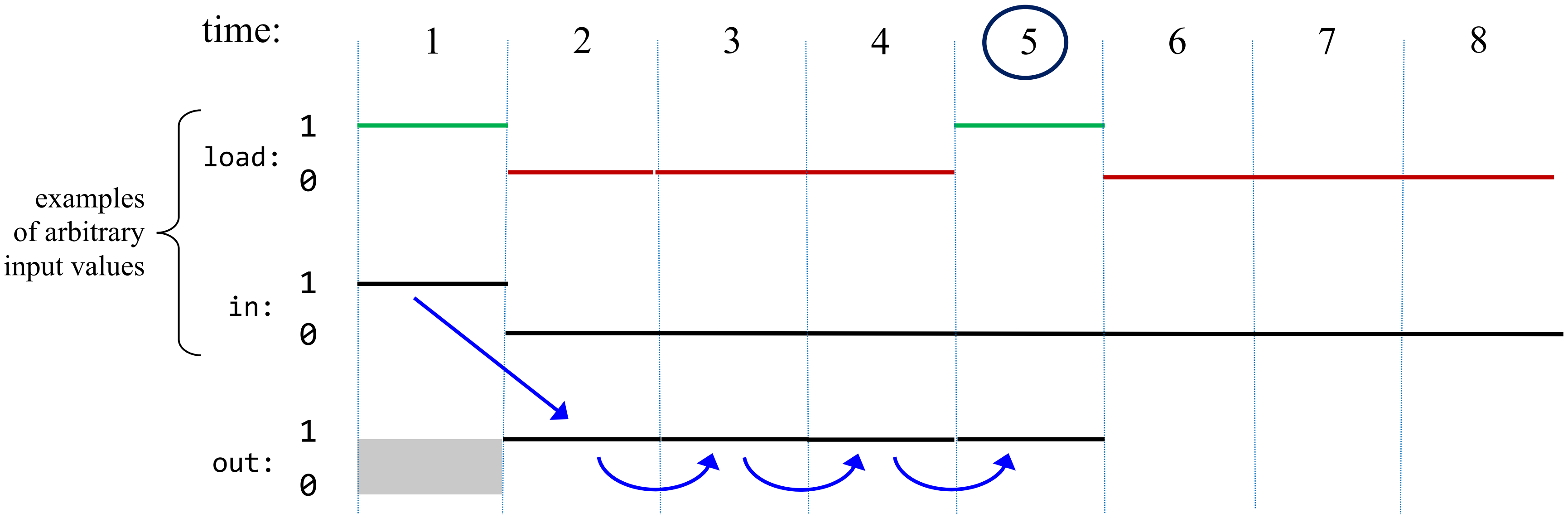
if load(t) then
  out($t+1$) = in(t)
else
  out($t+1$) = out(t)



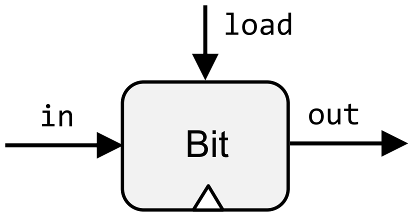
1-Bit register



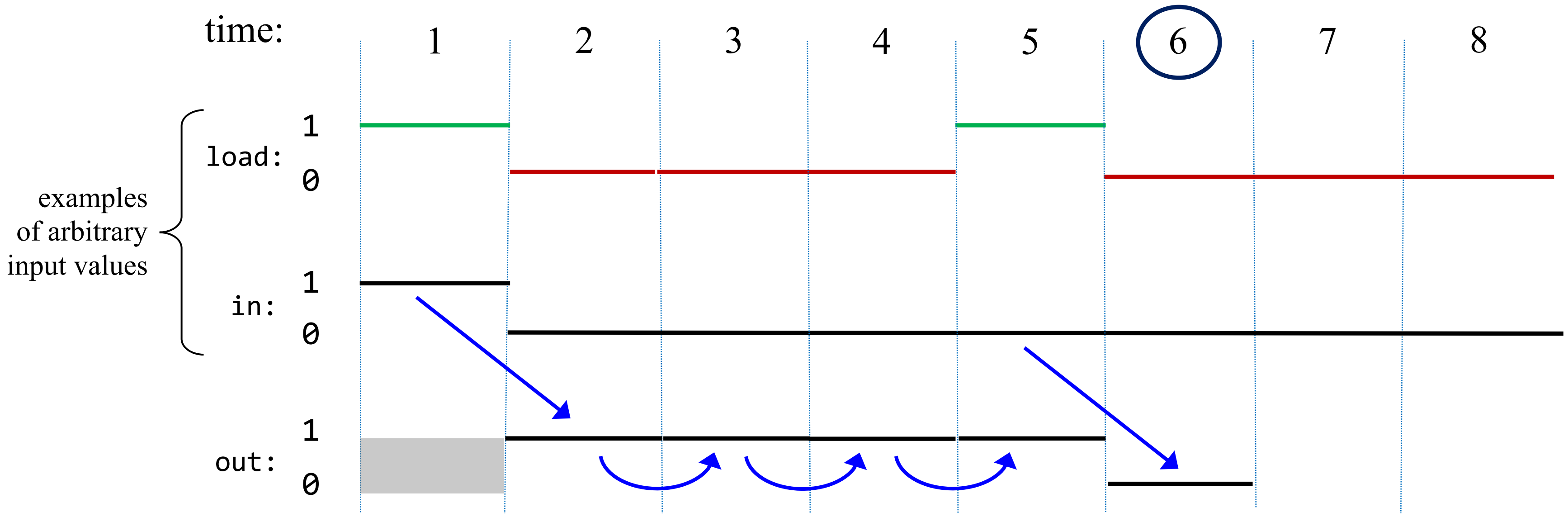
if `load`(t) then
  `out`$(t+1)$ = `in`(t)
else
  `out`$(t+1)$ = `out`(t)



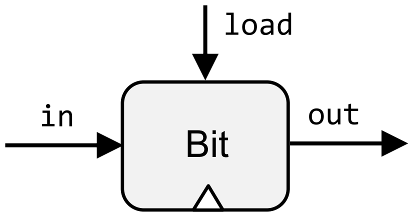
1-Bit register



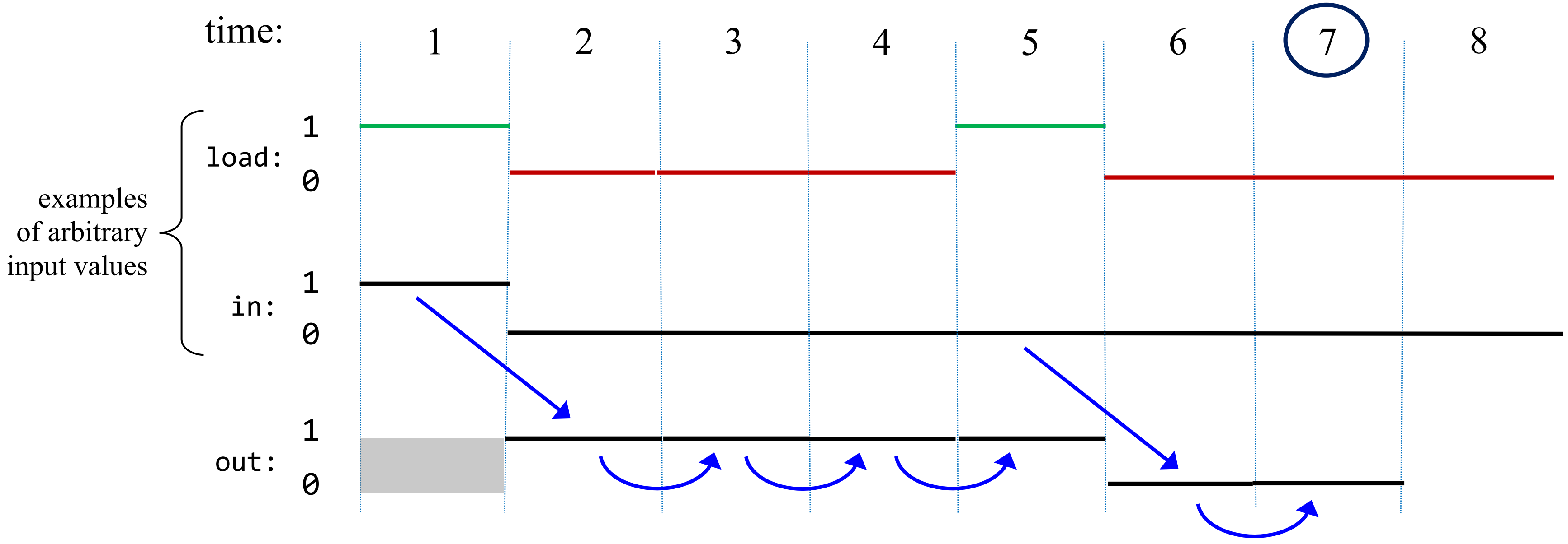
if `load`(t) then
    `out`($t+1$) = `in`(t)
else
    `out`($t+1$) = `out`(t)



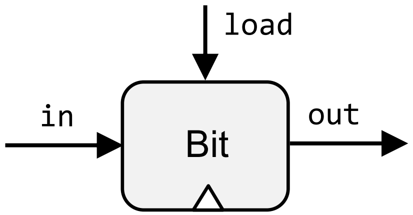
1-Bit register



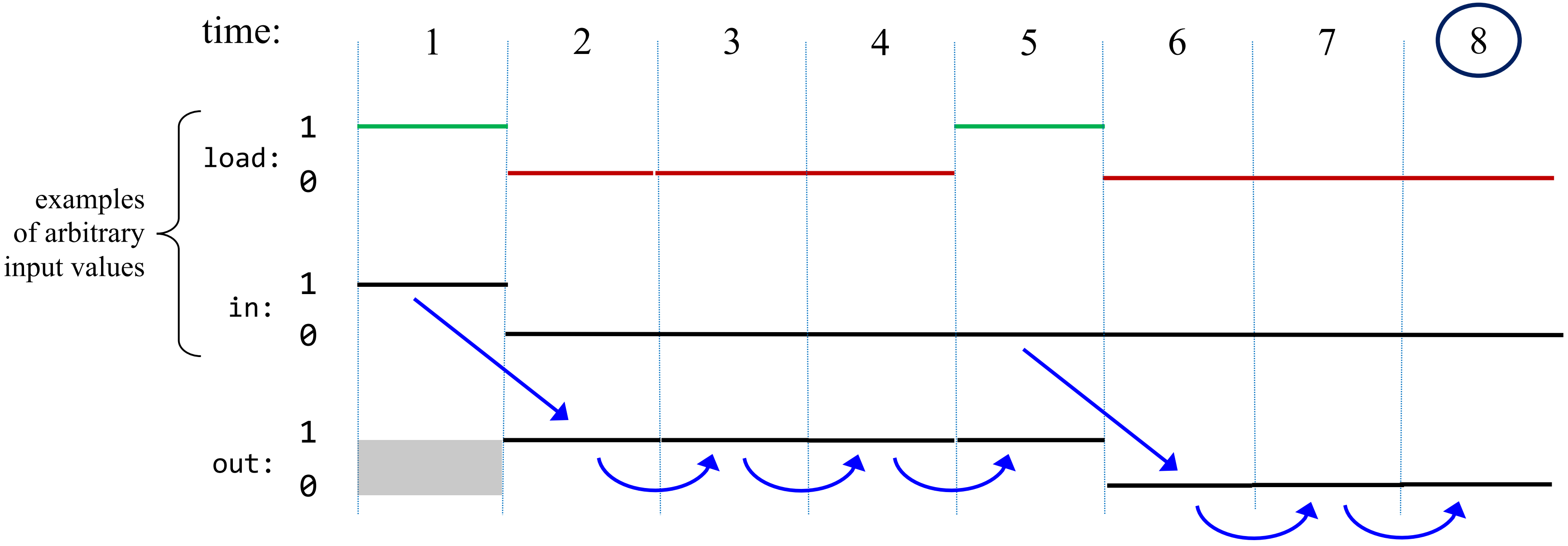
if load(t) then
    out($t+1$) = in(t)
else
    out($t+1$) = out(t)



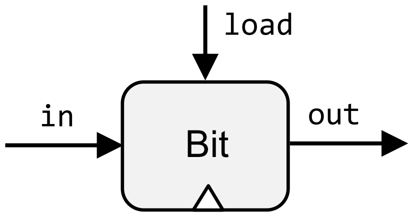
1-Bit register



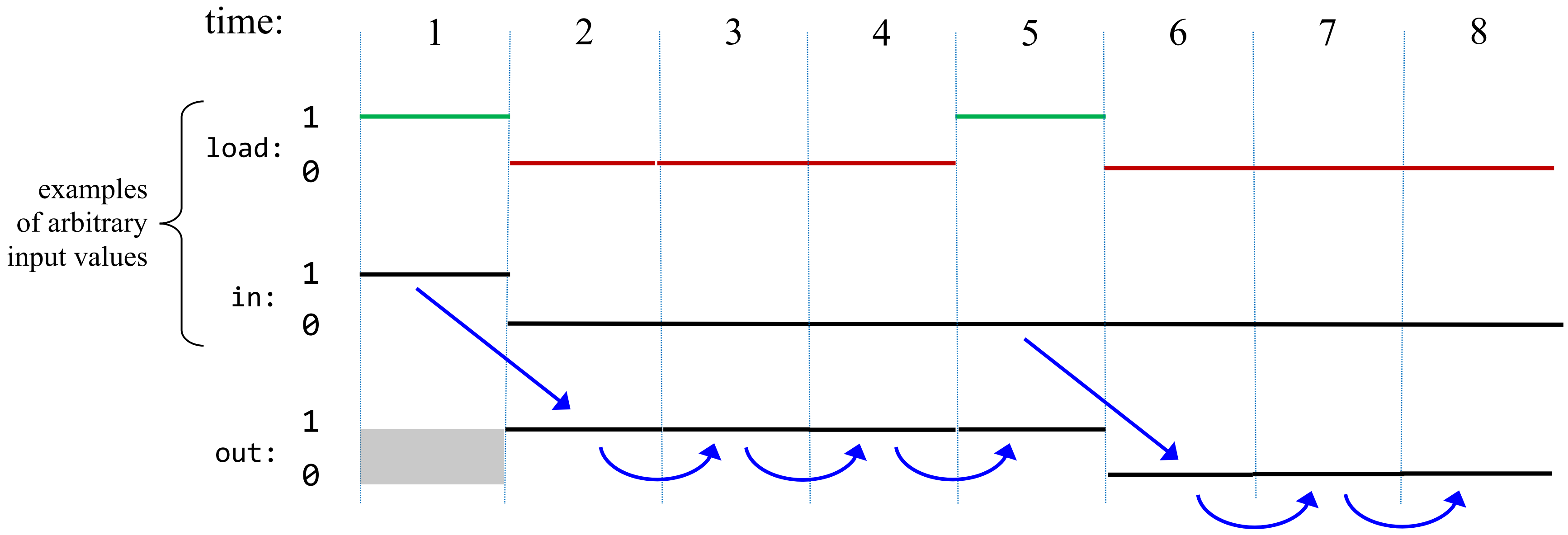
if `load`(t) then
  `out`$(t+1)$ = `in`(t)
else
  `out`$(t+1)$ = `out`(t)



1-Bit register

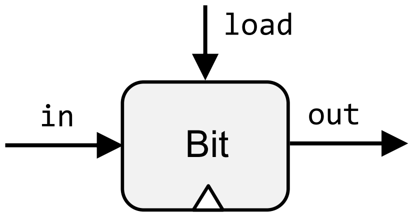


if `load(t)` then
    `out(t+1) = in(t)`
else
    `out(t+1) = out(t)`



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

1-Bit register



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

Usage:

To read:

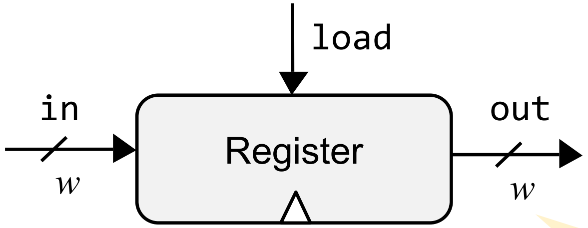
probe out   (out always emits the register's state)

To write:

set in = v Result: The register's state becomes v;
set load = 1   From the next time-step onward, out will emit v.

Best practice: Following writing, set load to 0

Multi-bit register



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

We'll focus on bit width $w = 16$, without loss of generality

<u>Load / store behavior:</u> Exactly the same as a 1-Bit register

<u>Read / write usage:</u> Exactly the same as a 1-Bit register

Chapter 3: Memory

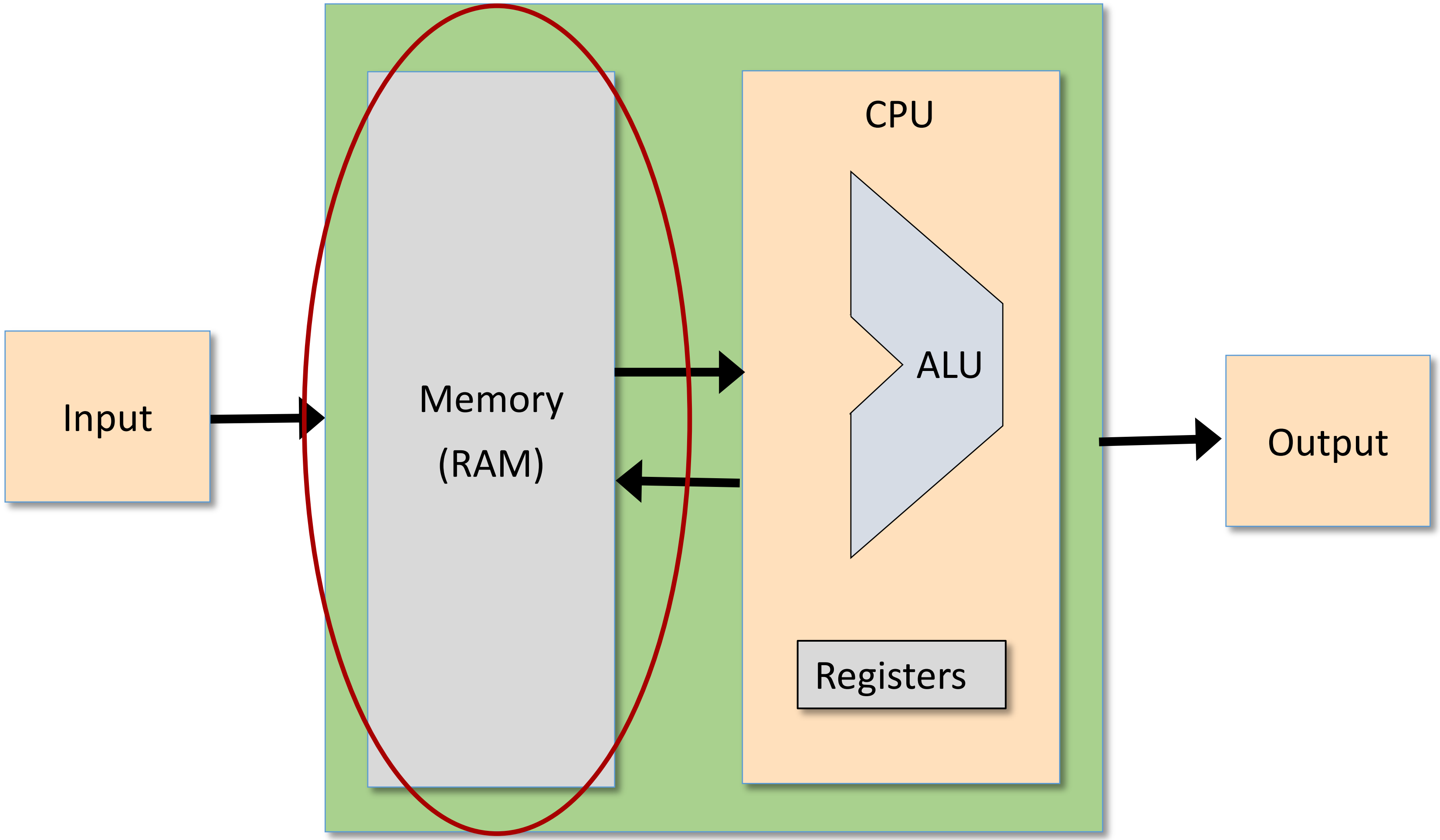
Abstraction

- ✓ Representing time
- ✓ Clock
- ✓ Registers
- RAM
- Counters

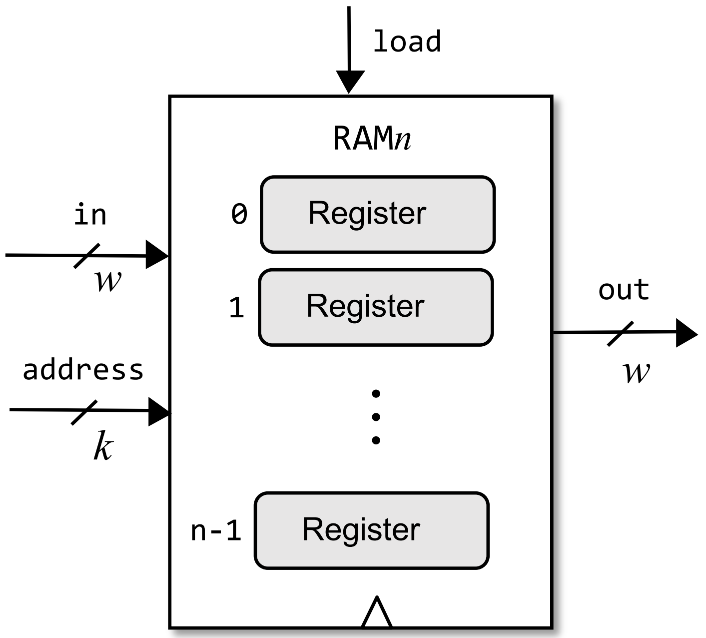
Implementation

- Data Flip Flop
- Registers
- RAM
- Project 3: Chips
- Project 3: Guidelines

Computer architecture



RAM



Practice question:

Suppose that the RAM size $n = 8$ registers.

What should be the value of k?

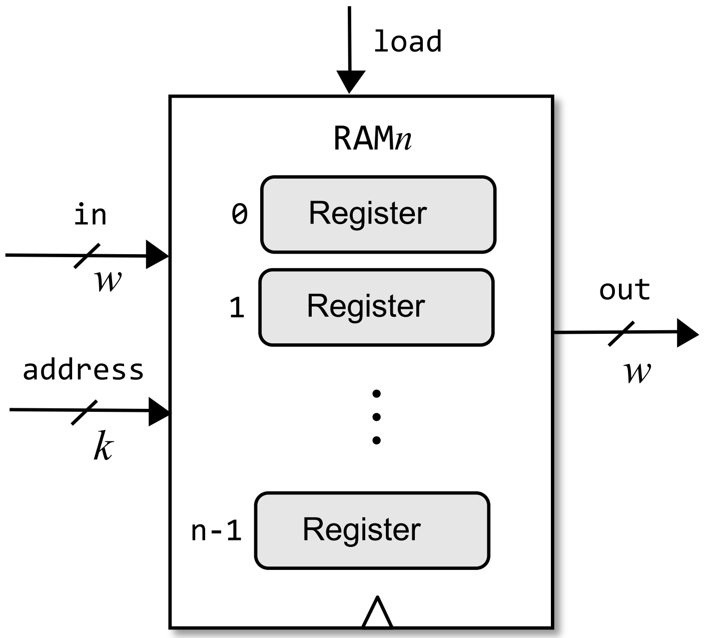
Answer:

$k = log_2 n$

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

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

RAM



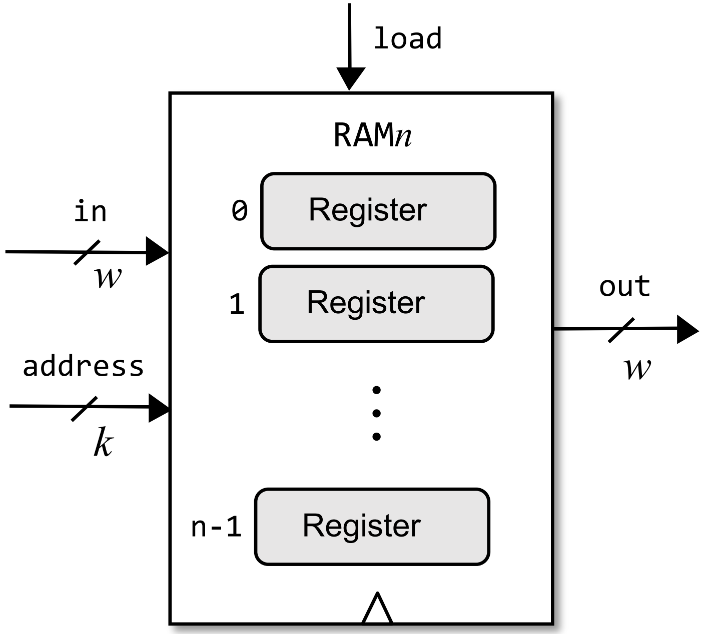
<u>Why "Random Access Memory"?</u>

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

Chapter 3: Memory

Abstraction	Implementation
✓ Representing time	• Data Flip Flop
✓ Clock	• Registers
✓ Registers	• RAM
✓ RAM	• Project 3: Chips
➡ Counters	• Project 3: Guidelines

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, until the next load

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

Usage:

To read register *i* :

set `address` = *i*,

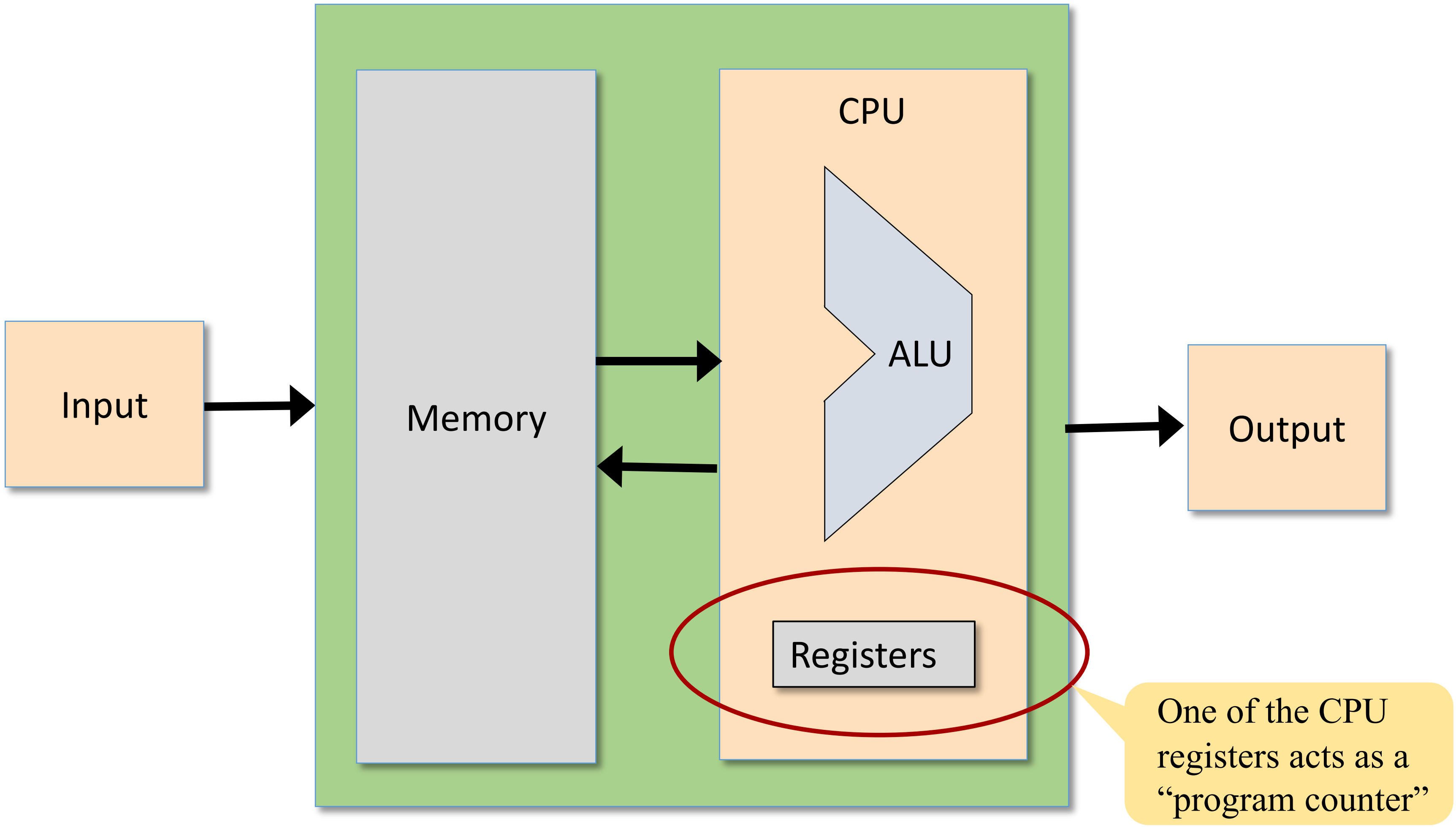
probe `out` (`out` always emits the value of `RAM[`*i*`]`)

To write *v* in register *i* :

set `address` = *i*,
set `in` = *v*,
set `load` = `1`

Result: `RAM[`*i*`]` ← *v*
From the next time-step onward
`out` will emit *v*

Computer architecture



Counter

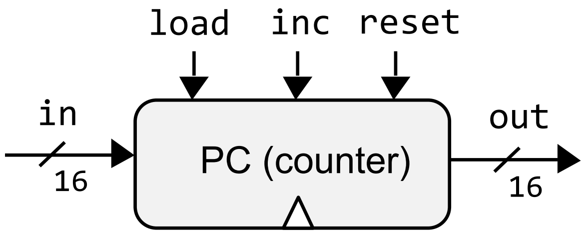
- Later in the course we will see that the computer keeps track of which instruction should be fetched and executed next
- This task is regulated by a register typically called `Program Counter`
- The `PC` stores the address of the instruction that should be fetched and executed next
- Three basic `PC` operations:

Reset: fetch the first instruction `PC = 0`

Next: fetch the next instruction `PC++`

Goto: fetch instruction *n* `PC =` *n*

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)

<u>Usage:</u>

To read:
probe `out`

To set:
set `in` to v,
assert `load`,
set the other control bits to `0`

To reset:
assert `reset`,
set the other control bits to `0`

To count:
assert `inc`,
set the other control bits to `0`

Chapter 3: Memory

<u>Abstraction</u>

- Representing time
- Clock
- Registers
- RAM
- Counters

<u>Implementation</u>

- Data Flip Flop
- Registers
- RAM
- Project 3: Chips
- Project 3: Guidelines

Chapter 3: Memory

Abstraction

- Representing time
- Clock
- Registers
- RAM
- Counters

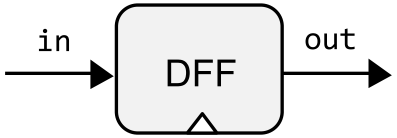
Implementation

- Data Flip Flop
- Registers
- RAM
- Project 3: Chips
- Project 3: Guidelines

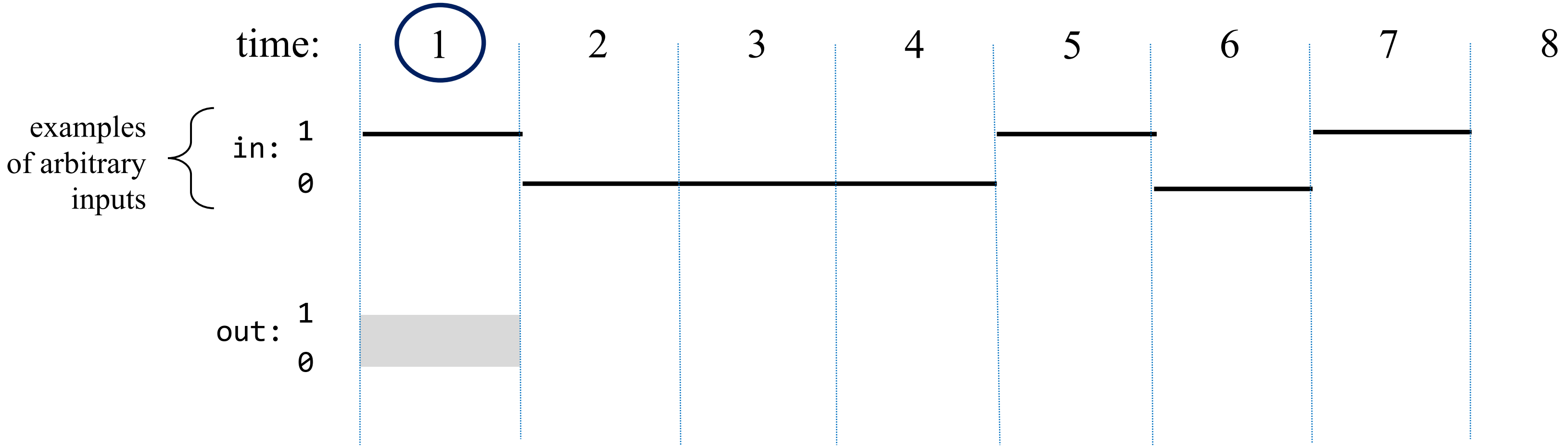
DFF

<u>Data Flip Flop</u> (aka *latch*)

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



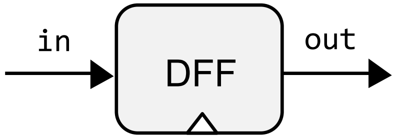
`out`$(t+1)$ `=` `in`(t)



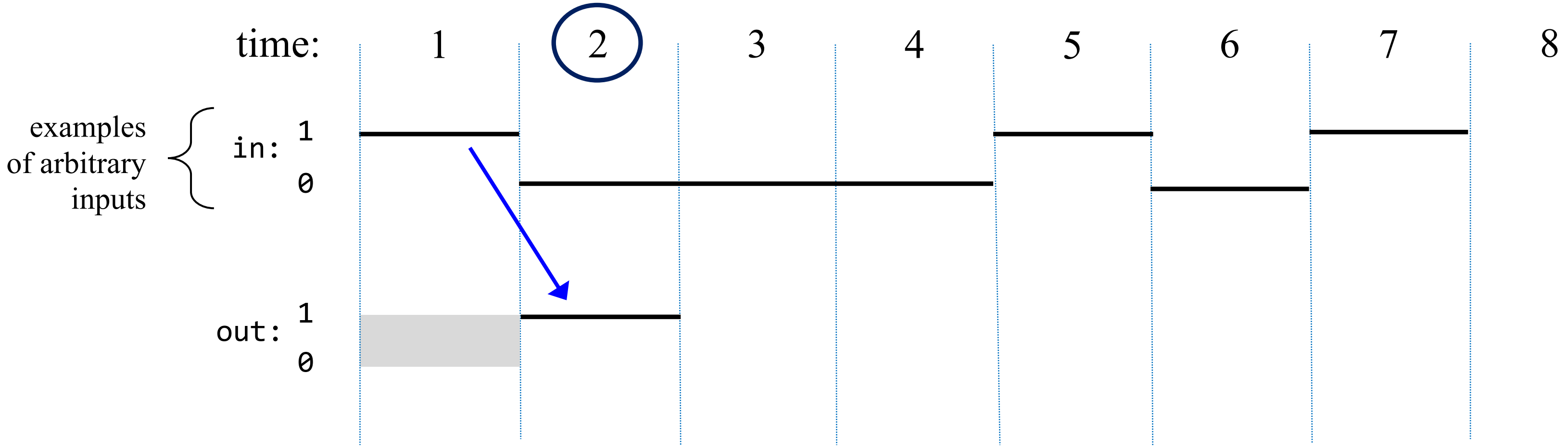
DFF

<u>Data Flip Flop</u> (aka *latch*)

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



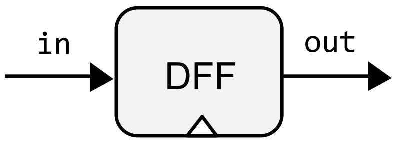
out$(t+1)$ = in(t)



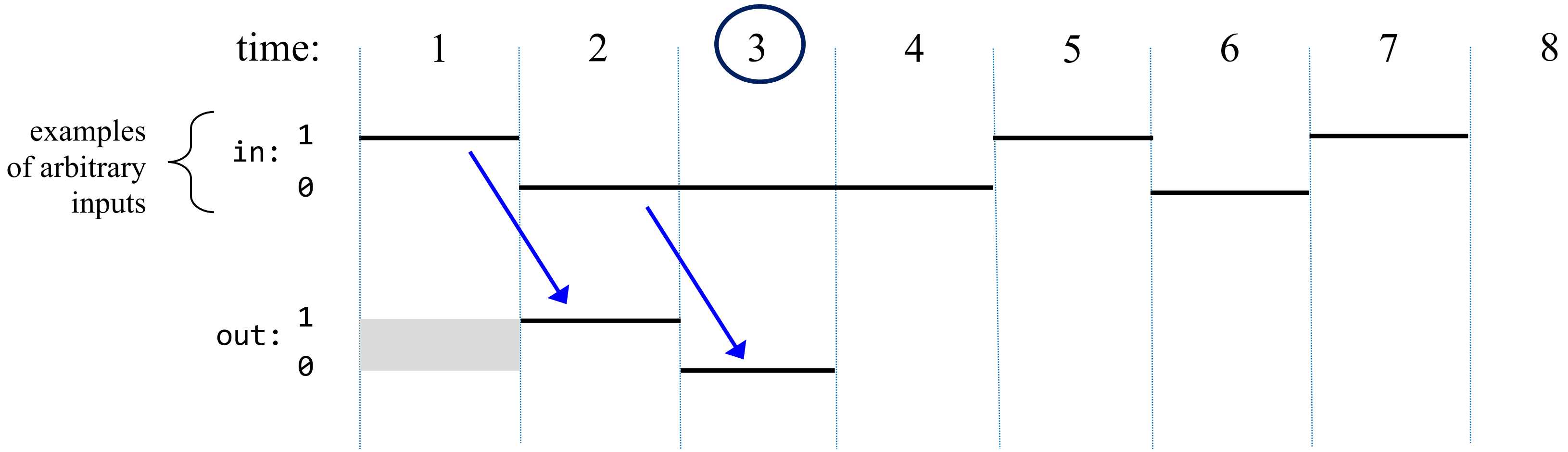
DFF

<u>Data Flip Flop</u> (aka *latch*)

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



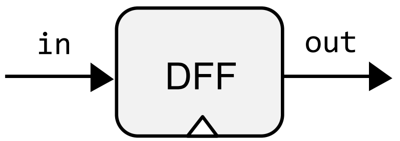
`out`$(t+1)$ = `in`(t)



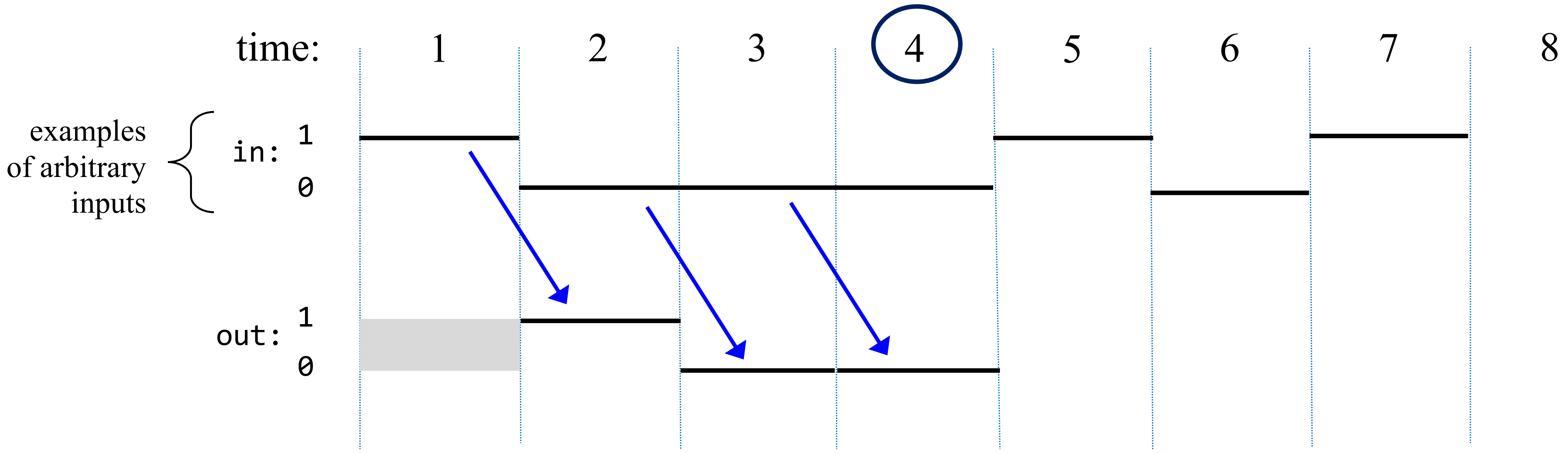
DFF

Data Flip Flop (aka *latch*)

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



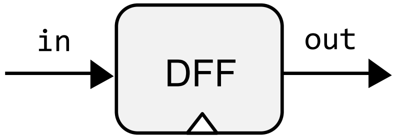
out($t+1$) = in(t)



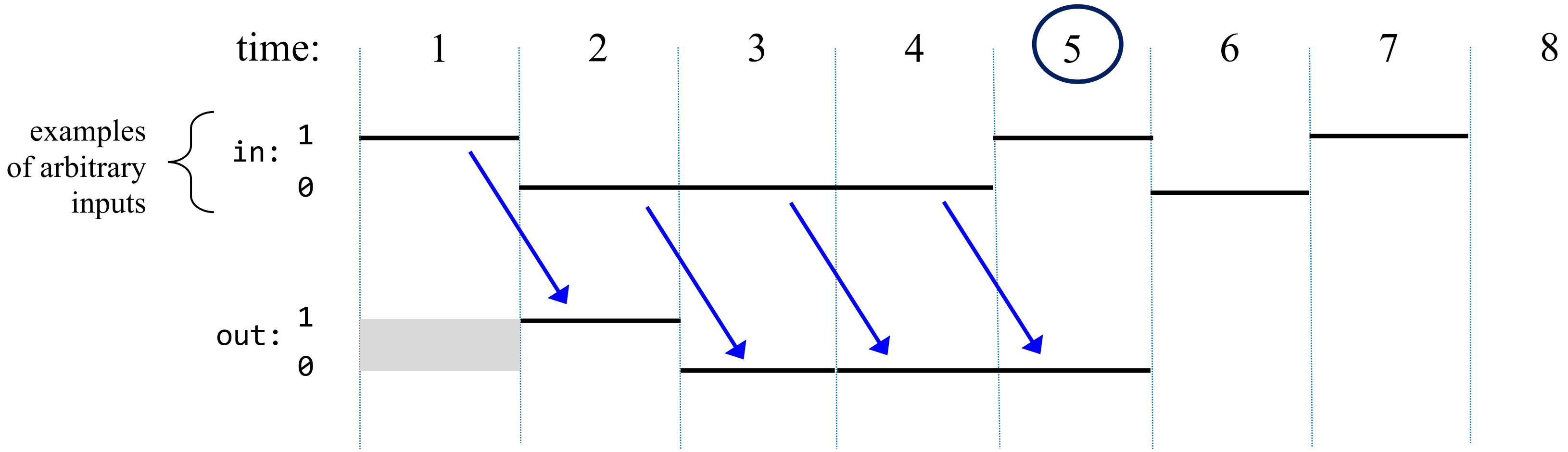
DFF

<u>Data Flip Flop</u> (aka *latch*)

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



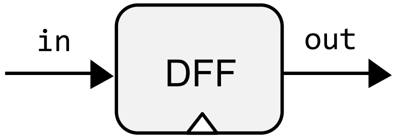
out$(t+1)$ = in(t)



DFF

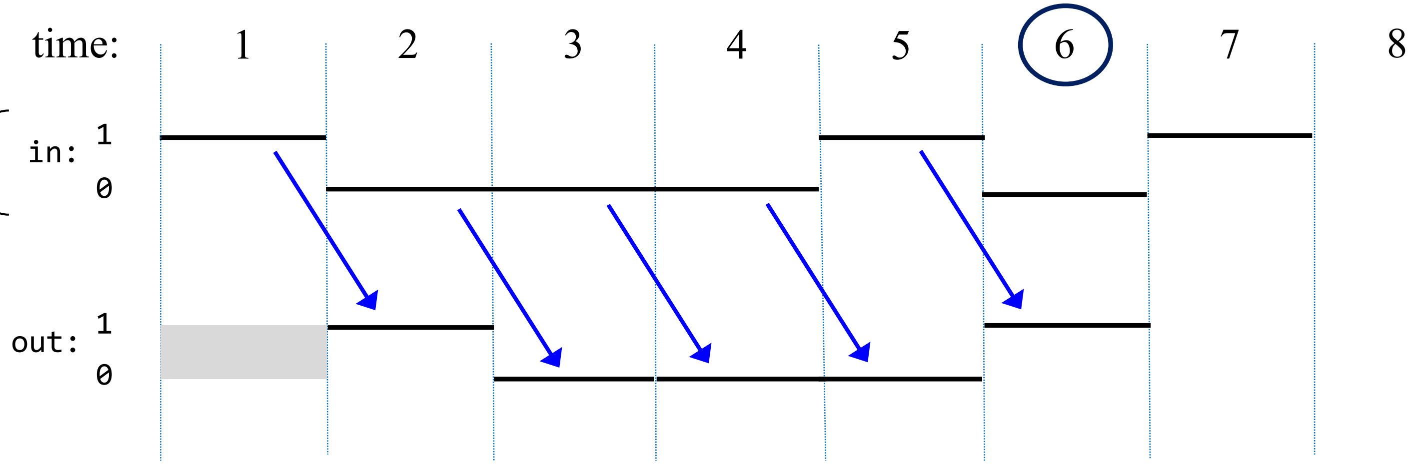
<u>Data Flip Flop</u> (aka *latch*)

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



`out`$(t+1)$ `=` `in`(t)

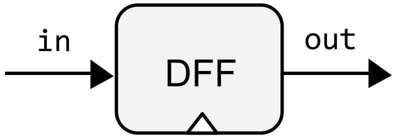
examples of arbitrary inputs



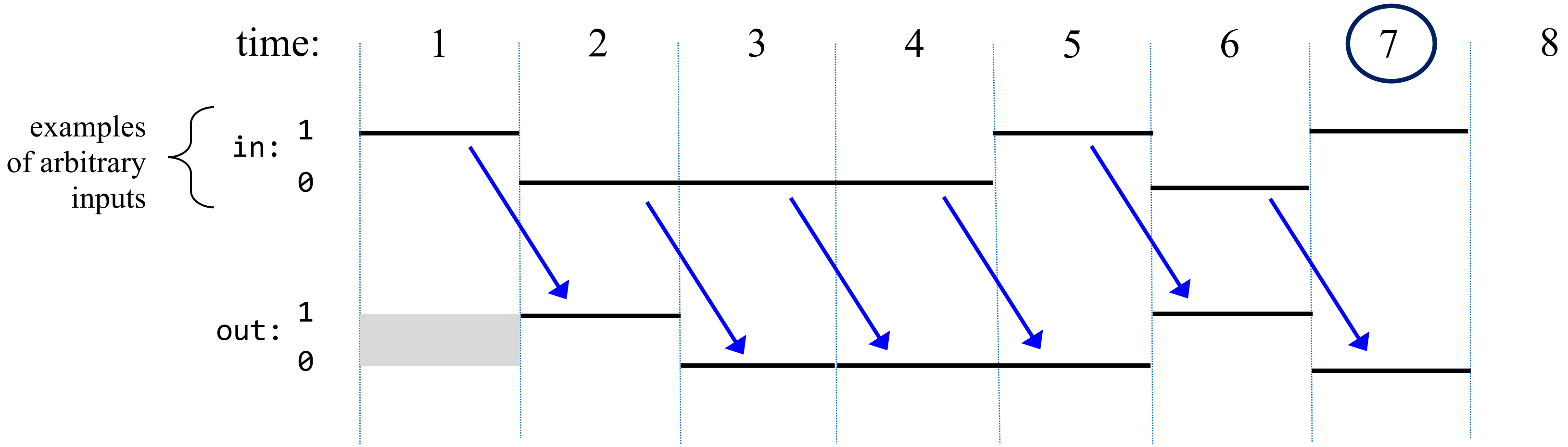
DFF

<u>Data Flip Flop</u> (aka *latch*)

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



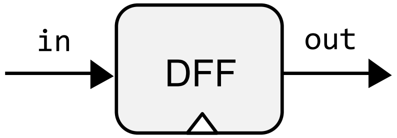
out($t+1$) = in(t)



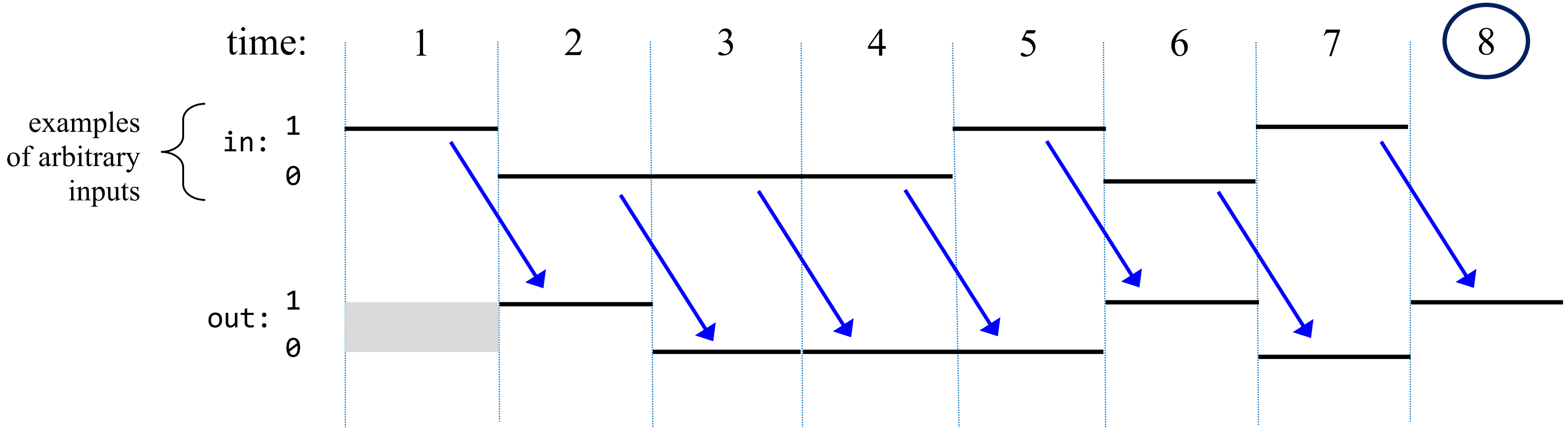
DFF

<u>Data Flip Flop</u> (aka *latch*)

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



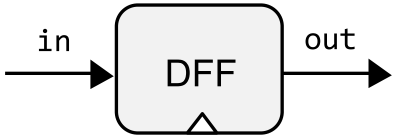
`out`$(t+1)$ = `in`(t)



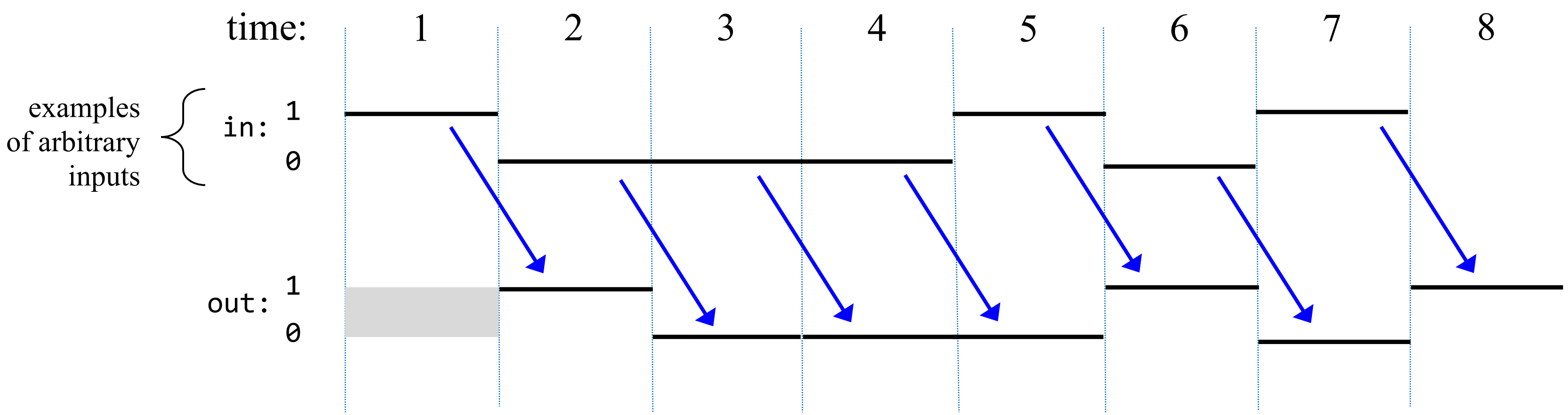
From DFF to a 1-Bit register

<u>Data Flip Flop</u> (aka *latch*)

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

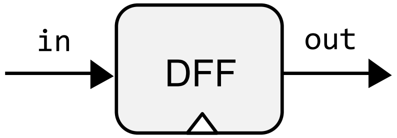


out$(t+1)$ = in(t)



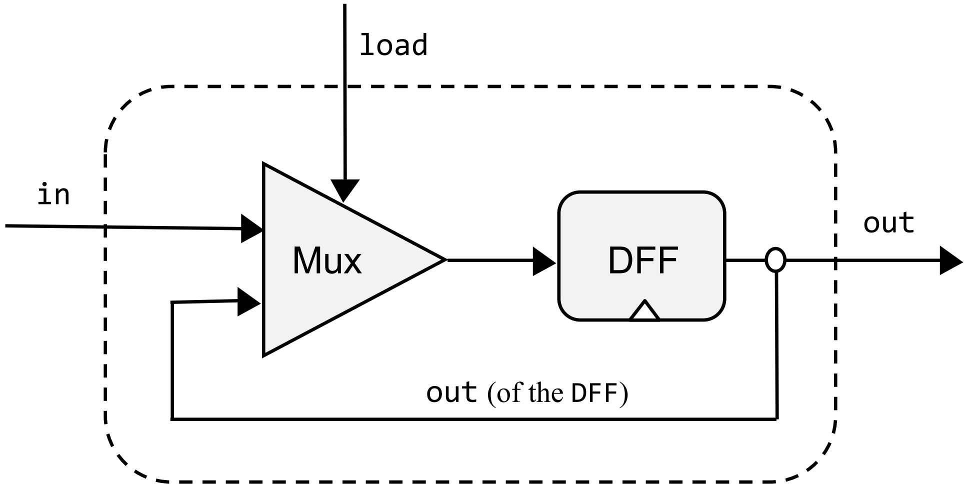
How can we “load” and then “maintain” a value (0 or 1) over time, without having to feed the value in every cycle?

From DFF to a 1-Bit register



We have to realize a “loading” behavior and a ”storing” behavior, and be able to select between these two states

From DFF to a 1-Bit register



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

We have to realize a "loading" behavior and a "storing" behavior, and be able to select between these two states

<u>Behavior</u>

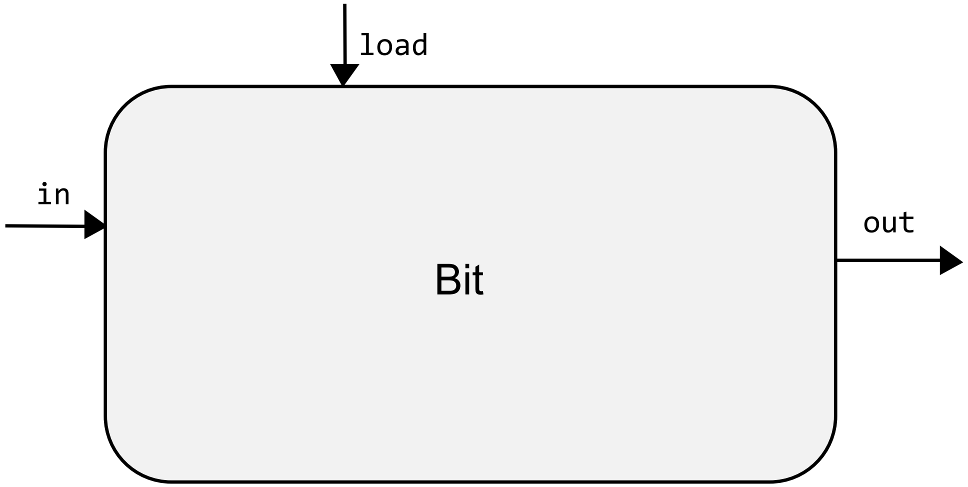
if `load == 1` the register's value becomes `in`

else the register maintains its current value

From DFF to a 1-Bit register

<u>1-Bit register</u>

Stores one bit over time



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

<u>Behavior</u>

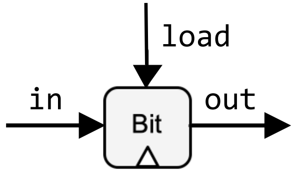
if `load == 1` the register's value becomes `in`

else the register maintains its current value

Register

<u>1-Bit register</u>

Stores one bit over time



zoom out...

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

<u>Behavior</u>

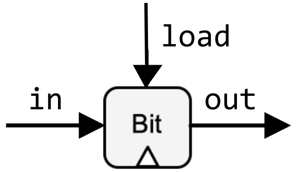
if `load == 1` the register's value becomes `in`

else the register maintains its current value

Register

<u>1-Bit register</u>

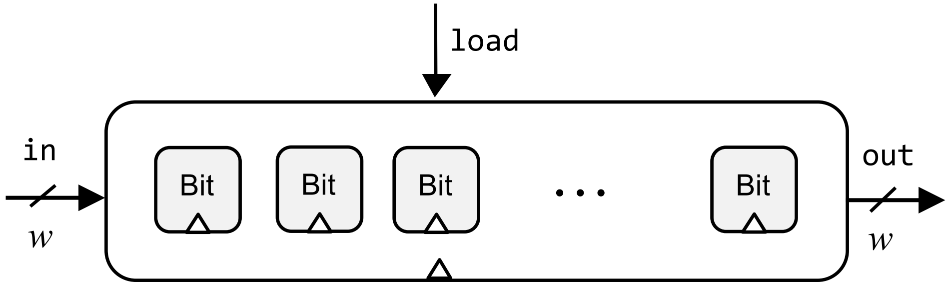
Stores one bit
over time



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

<u>*w*-bit Register</u>

Stores *w* bits
over time



<u>Behavior</u>

if `load == 1` the register's value becomes `in`

else the register maintains its current value

Chapter 3: Memory

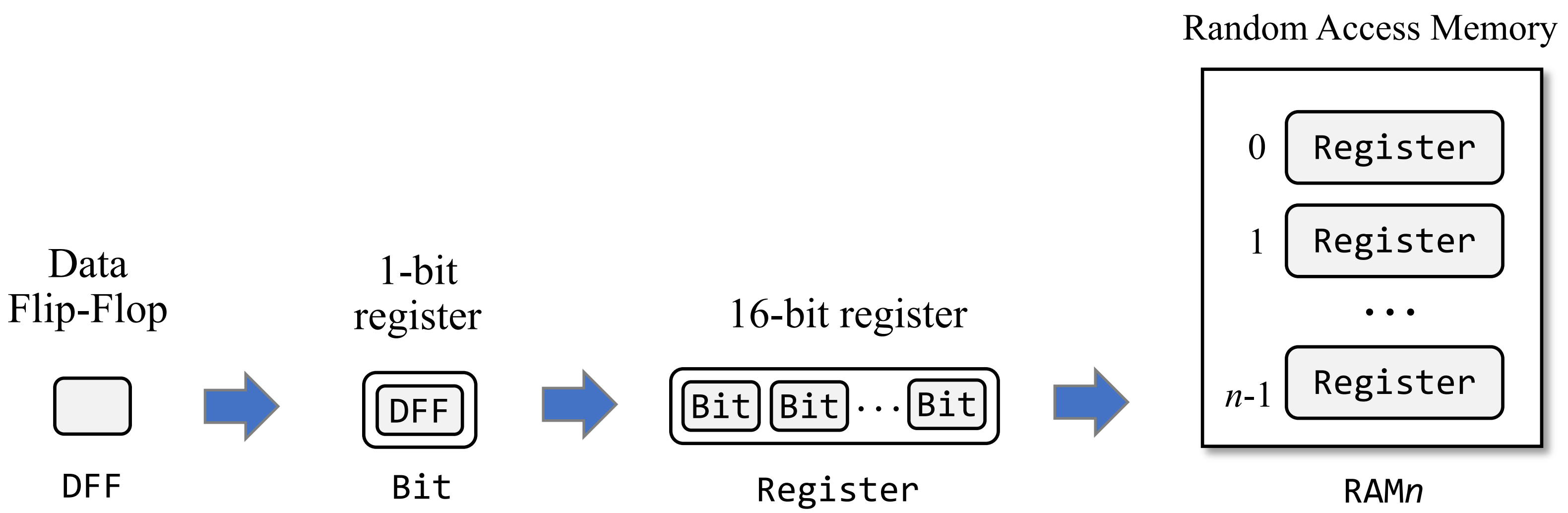
<u>Abstraction</u>

- Representing time
- Clock
- Registers
- RAM
- Counters

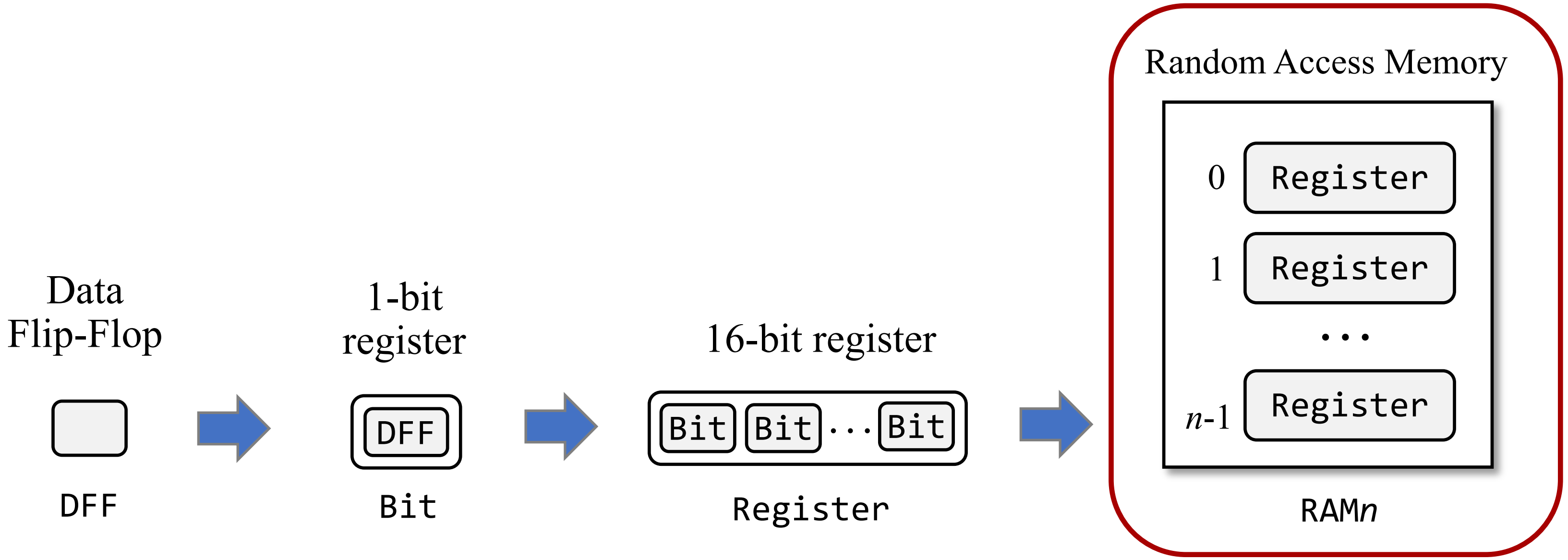
<u>Implementation</u>

- ✓ Data Flip Flop
- ✓ Registers
- → RAM
- Project 3: Chips
- Project 3: Guidelines

Memory hierarchy

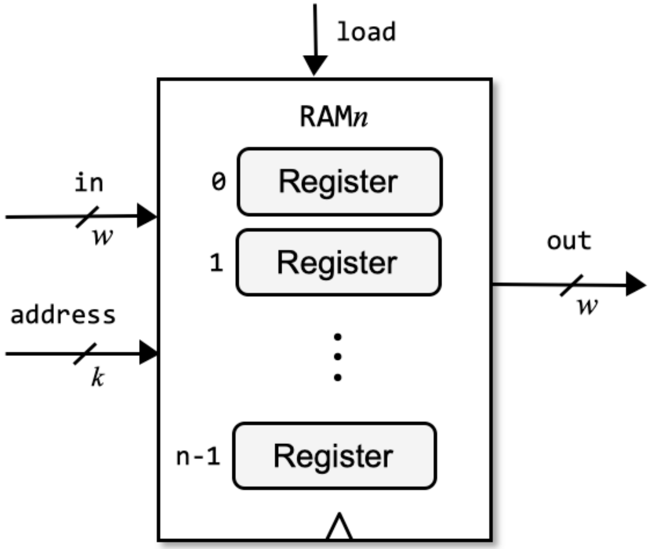


Memory hierarchy



RAM: Abstraction

RAM of n registers:



<u>Usage:</u>

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` $= v$,

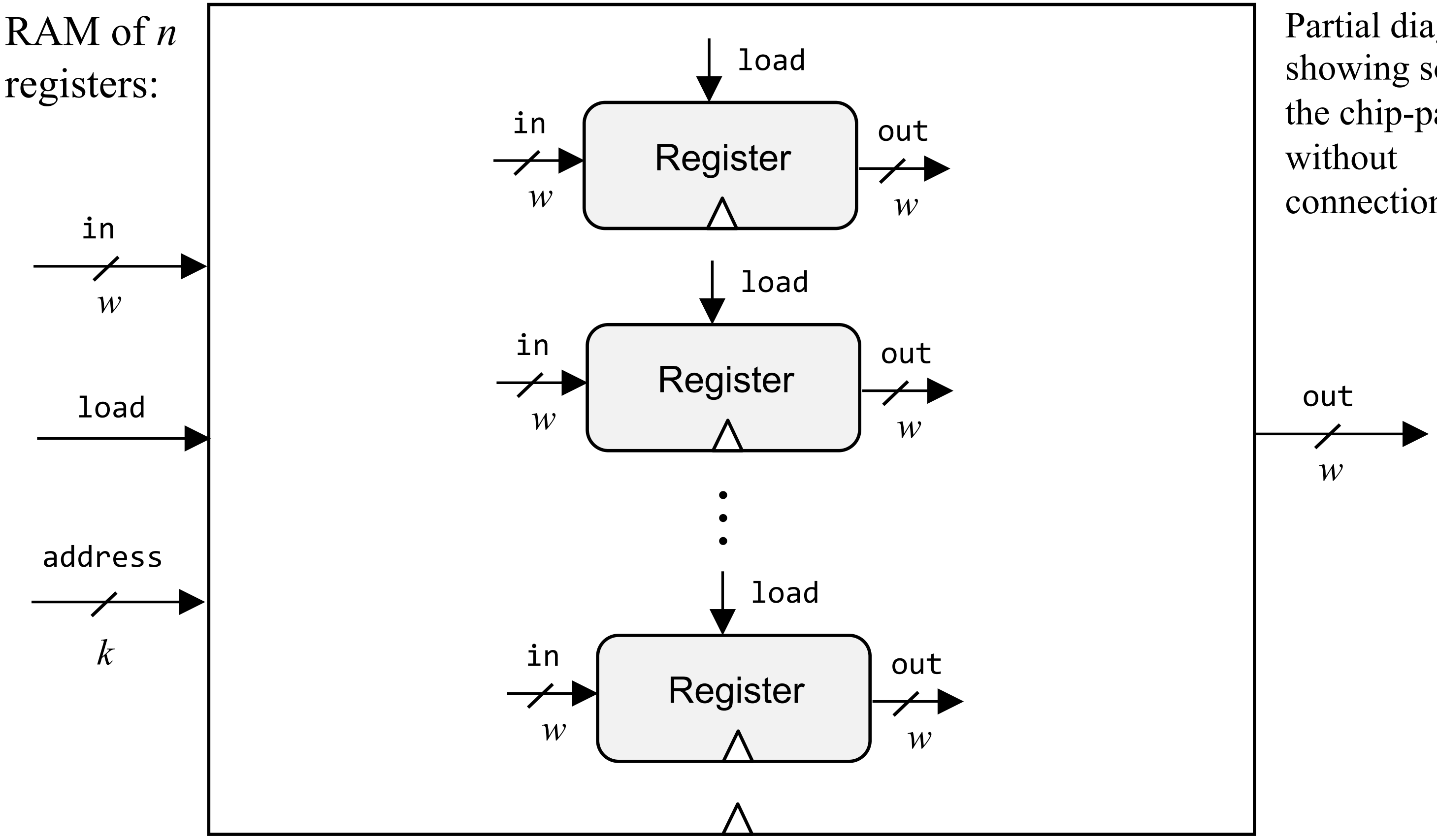
set `load` $=$ `1`

Result: `RAM[`i`]` $\leftarrow v$

From the next time-step onward, `out` emits v

RAM: Implementation

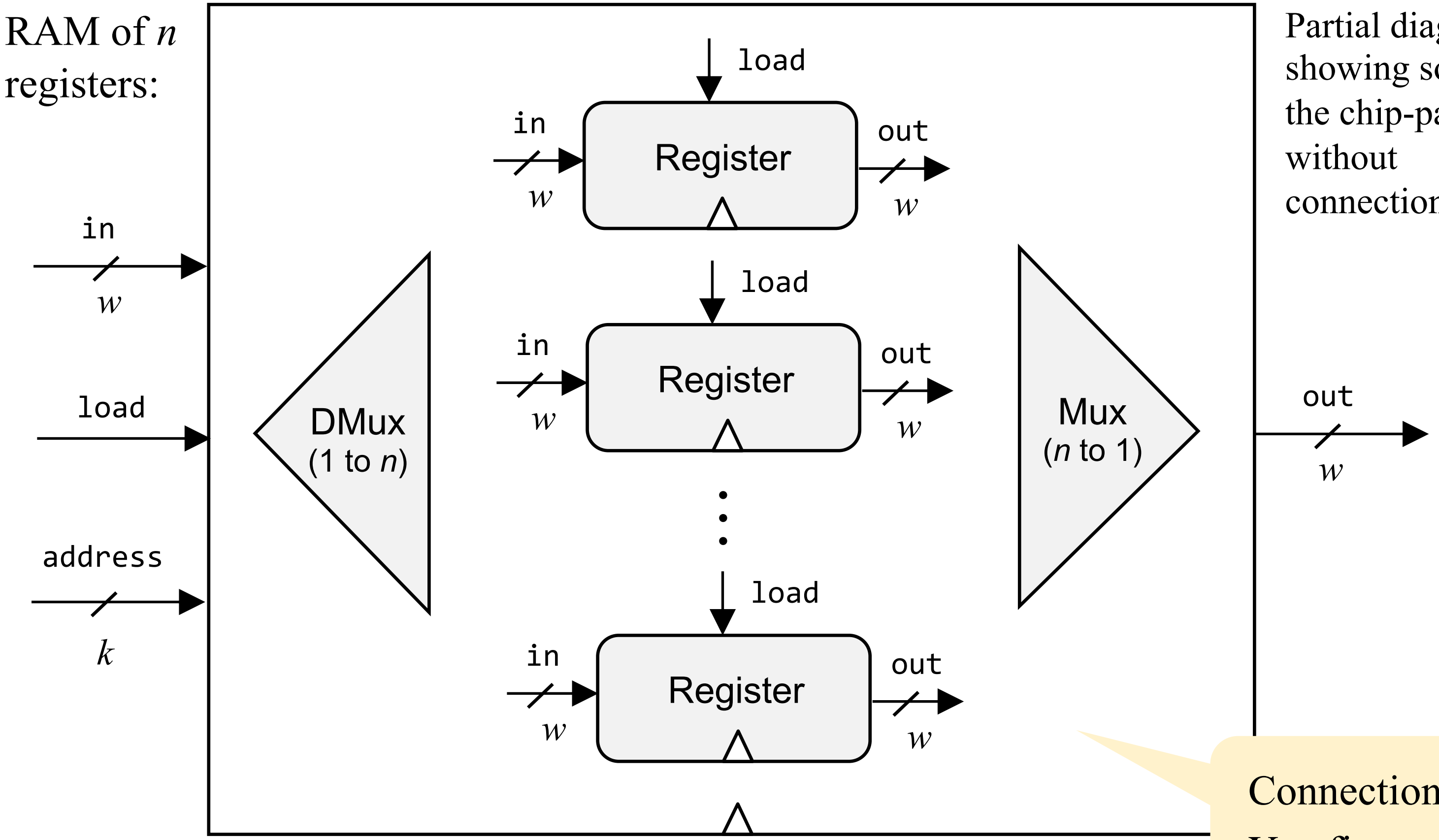
RAM of n registers:



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

RAM: Implementation

RAM of n registers:



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

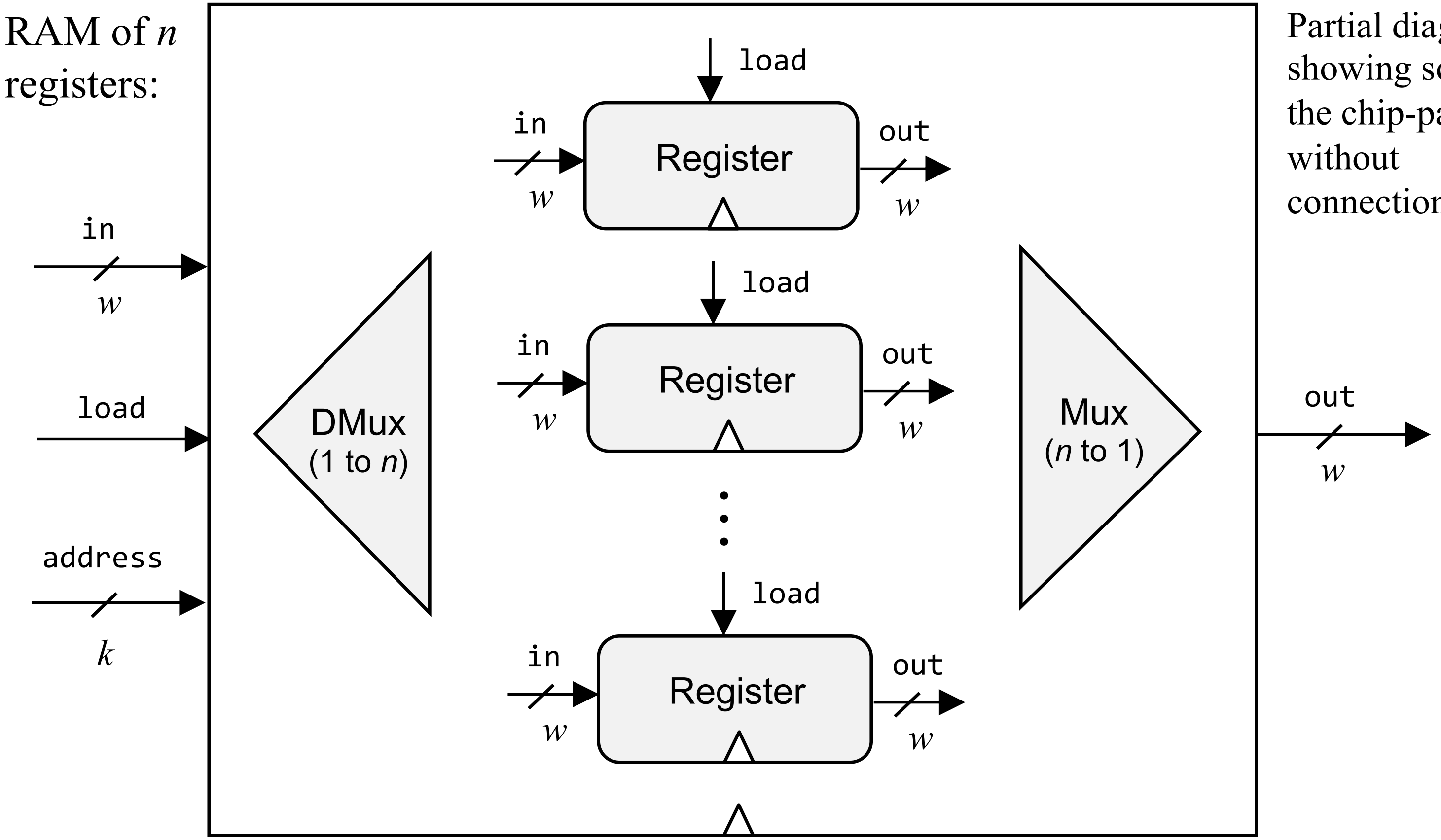
Connections?
You figure it out

<u>Reading:</u> Can be realized using a Mux

<u>Writing:</u> Can be realized using a DMux

RAM: Implementation

RAM of n registers:

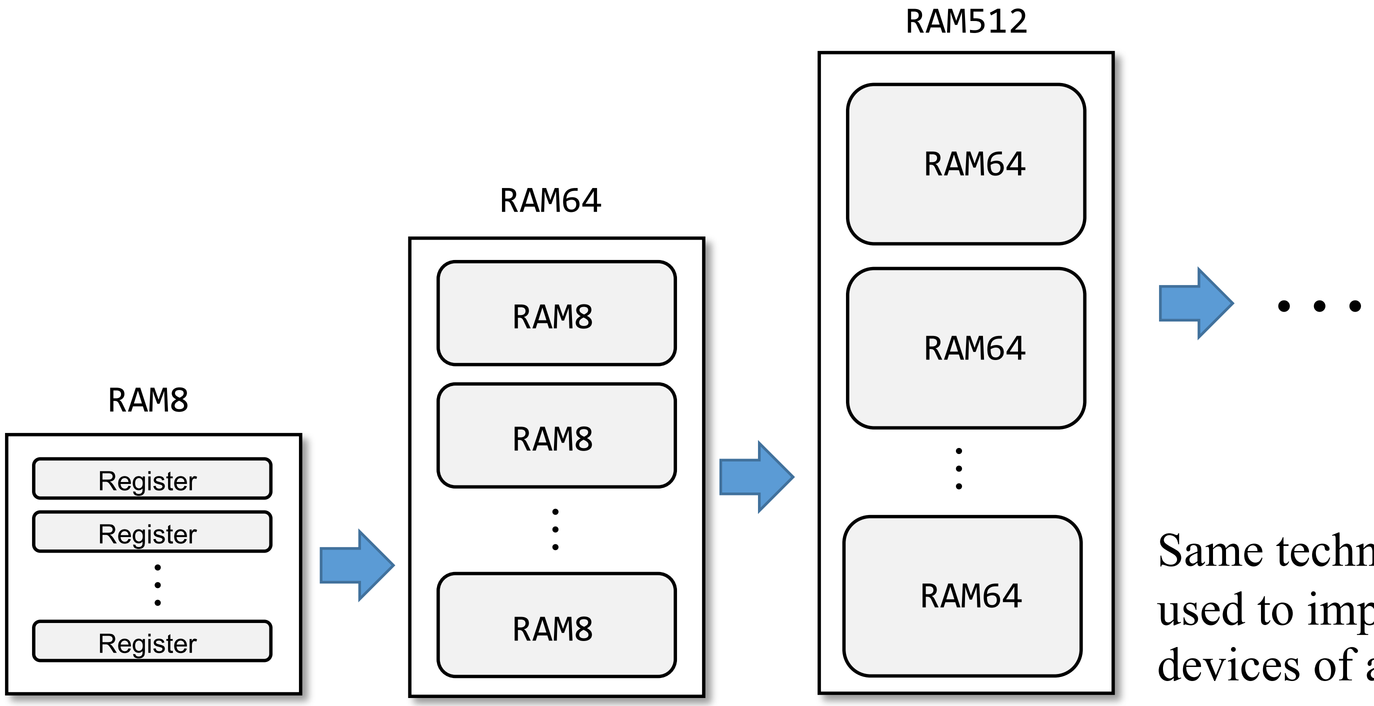


Partial diagram, showing some of the chip-parts, without connections

<u>The RAM is a sequential logic chip</u>:

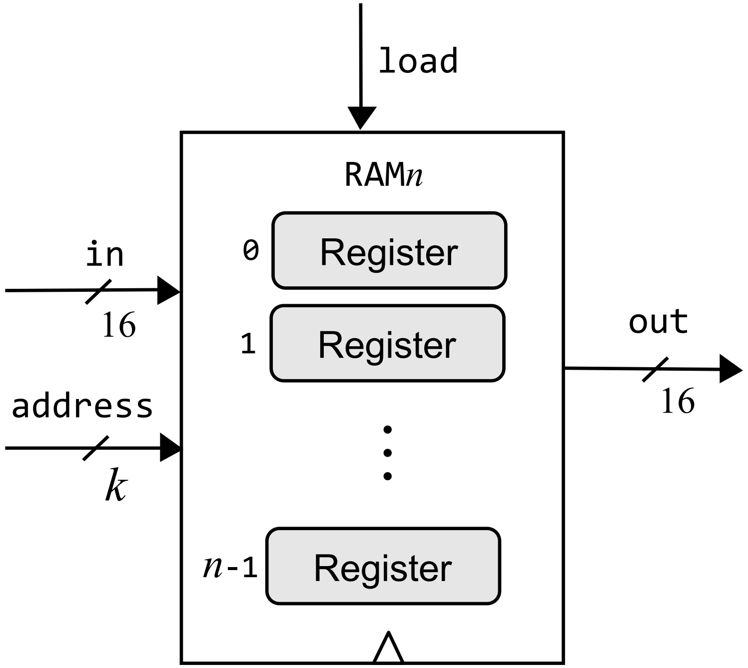
- The direct access behavior is realized by the Mux / Dmux, which are combinational
- The storage behavior is reazlied by the registers, which are based on sequential logic.

RAM: Implementation



Same technique can be used to implement RAM devices of any size

Hack RAM



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

<u>Why these particular RAM chips?</u>

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

Chapter 3: Memory

<u>Abstraction</u>

- Representing time
- Clock
- Registers
- RAM
- Counters

<u>Implementation</u>

- ✓ Data Flip Flop
- ✓ Registers
- ✓ RAM
- ➡ Project 3: Chips
- Project 3: Guidelines

Project 3

Given:

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

Why is the `DFF` built-in?

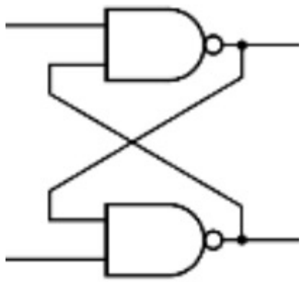
Build:

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

DFF implementation

- The DFF gate logic implementation requires, among other things, connecting Nand gates with feedback loops
- The resulting implementation is elegant, yet impossible to realize on the supplied hardware simulator

  The hardware simulator does not allow loops of combinational chips (like `Nand`)
- Instead, we use a built-in DFF implementation:



Part of a possible DFF implementation

```
/** Data Flip-flop: out(t) = in(t–1)
 *   where t is the current time unit. */

CHIP DFF {
    IN  in;
    OUT out;

    BUILTIN DFF;
    CLOCKED in;
}
```

Implementation notes:

We need the `DFF` as a chip-part in one chip only: `Bit`

But… All the other memory chips are built on top of it.

Project 3

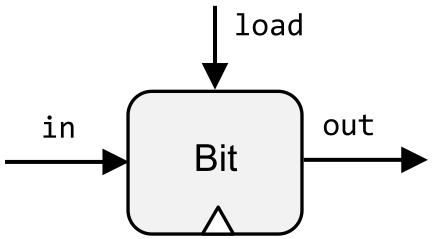
Given:

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

Build:

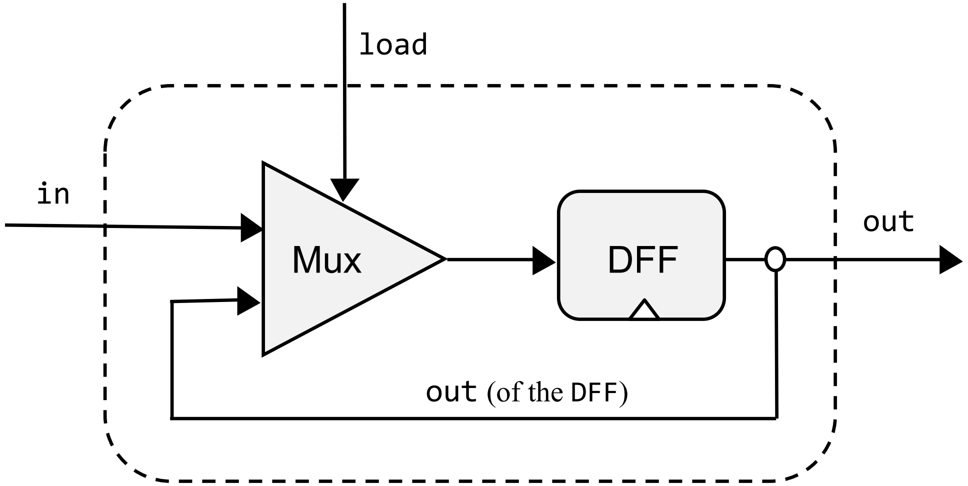
- `Bit`
- `Register`
- `PC`
- `RAM8`
- `RAM64`
- `RAM512`
- `RAM4K`
- `RAM16K`

1-Bit register



Bit.hdl

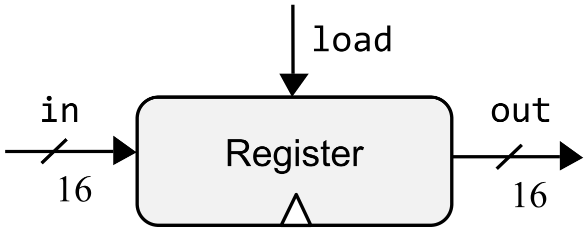
```
/** 1-Bit register:
      if  load(t) then out(t + 1) = in(t)
      else                 out(t + 1) = out(t)  */

CHIP Bit {
    IN in, load;
    OUT out;

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



Implementation tip:

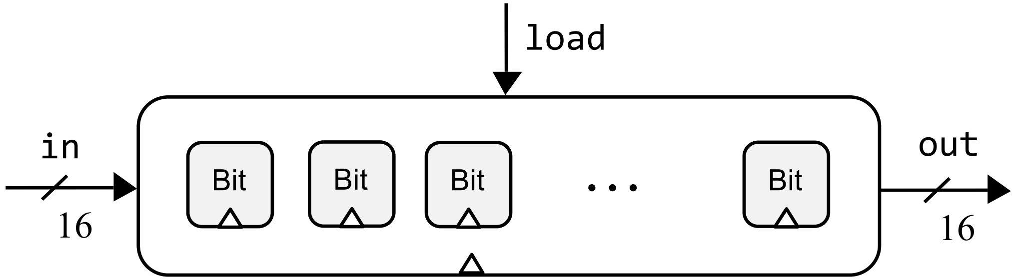
Follow the chip diagram

16-bit Register



Register.hdl

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

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

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

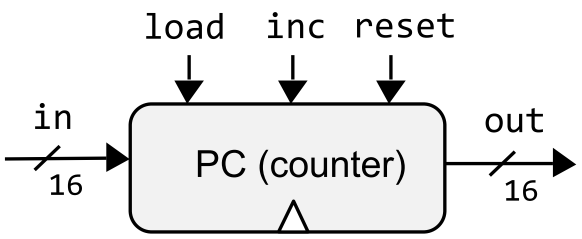


Partial diagram, showing some of
the chip-parts, without connections

<u>Implementation tip:</u>

Follow the chip diagram

16-bit Counter




```
/**
   A 16-bit counter.
   if         reset(t)   out(t + 1) = 0              // resetting
   else  if  load(t)   out(t + 1) = in(t)           // setting
   else  if  inc(t)     out(t + 1) = out(t) + 1   // incrementing
   else                      out(t + 1) = out(t)        // maintaining
 */

CHIP PC {
   IN in[16], load, inc, reset;
   OUT out[16];
   PARTS:
    // Put your code here:
}
```

<u>Implementation tip</u>:

Can be built from a Register, an Incrementor, and Mux's

Project 3

Given

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

Build the following chips

✓ `Bit`

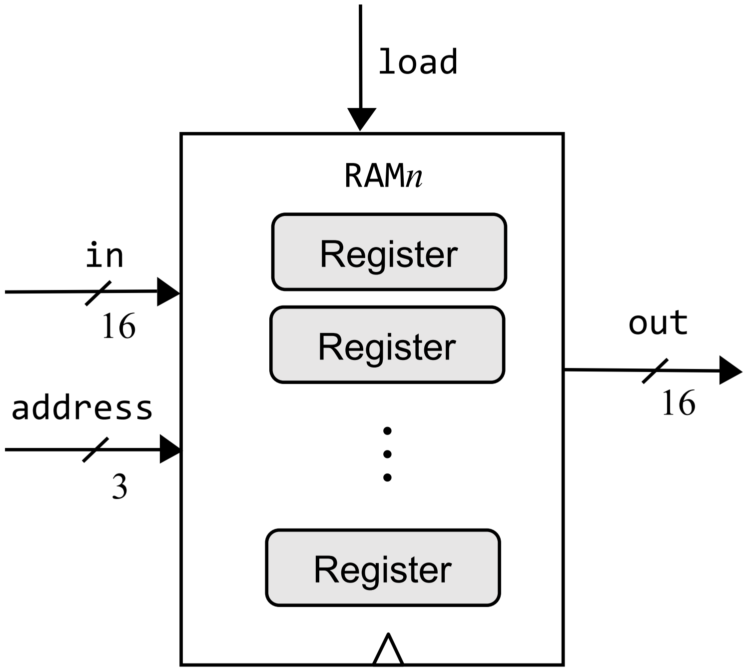
✓ `Register`

✓ `PC`

➡ `RAM8`

- `RAM64`
- `RAM512`
- `RAM4K`
- `RAM16K`

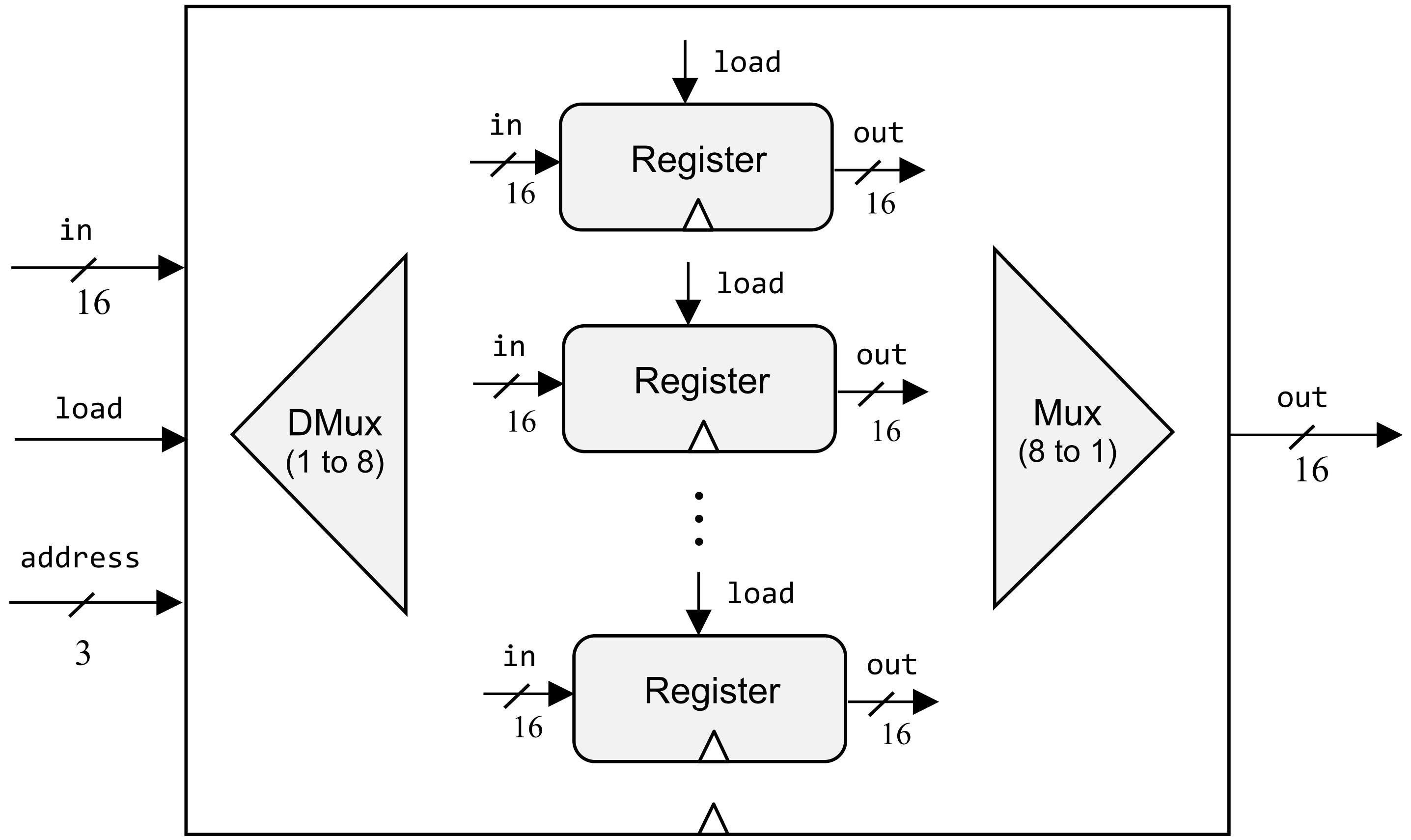
8-Register RAM



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

8-Register RAM



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

<u>Implementation tip:</u>

Follow the chip diagram, and figure out the missing connections.

Project 3

Given

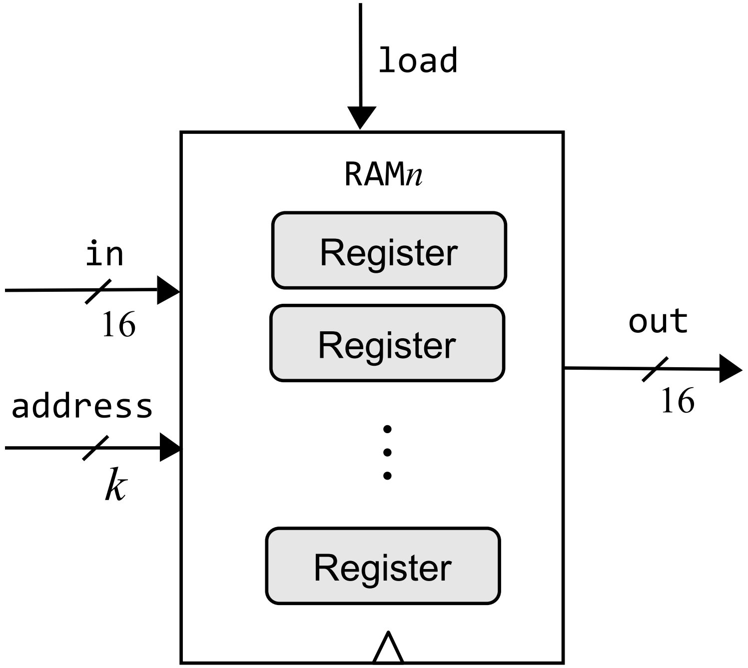
- All the chips built in projects 1 and 2
- Data Flip-Flop (built-in `DFF` gate)

Build the following chips

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

A family of RAM chips

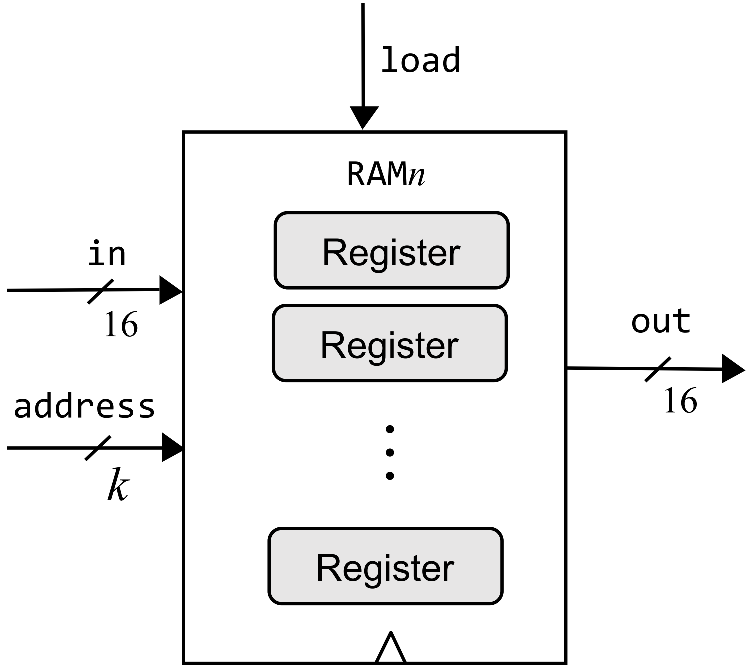
n-Register RAM



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 RAMn {
    IN in[16], load, address[k];
    OUT out[16];

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

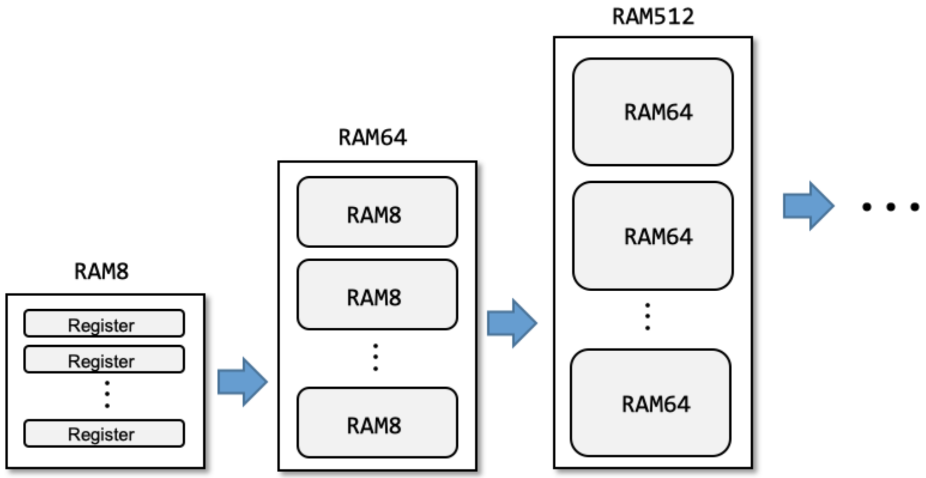
n-Register RAM



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

Implementation tips

- Think about the RAM's 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 logic.



Chapter 3: Memory

Abstraction	Implementation
• Representing time	✓ Data Flip Flop
• Clock	✓ Registers
• Registers	✓ RAM
• RAM	✓ Project 3: Chips
• Counters	➡ Project 3: Guidelines

Guidelines

- Implement the chips in the order in which they appear in the project guidelines
- If you don't implement some chips, you can still use their built-in implementations
- No need for "helper chips": Implement / use only the chips we specified
- In each chip definition, strive to use as few chip-parts as possible
- You will have to use chips implemented in previous projects;

  For efficiency and consistency's sake, use their built-in versions, rather than your own HDL implementations.

For technical reasons, the chips of project 3 are organized in two sub-folders named `projects/03/a` and `projects/03/b`

When writing and simulating the `.hdl` files, leave this folder structure as is.

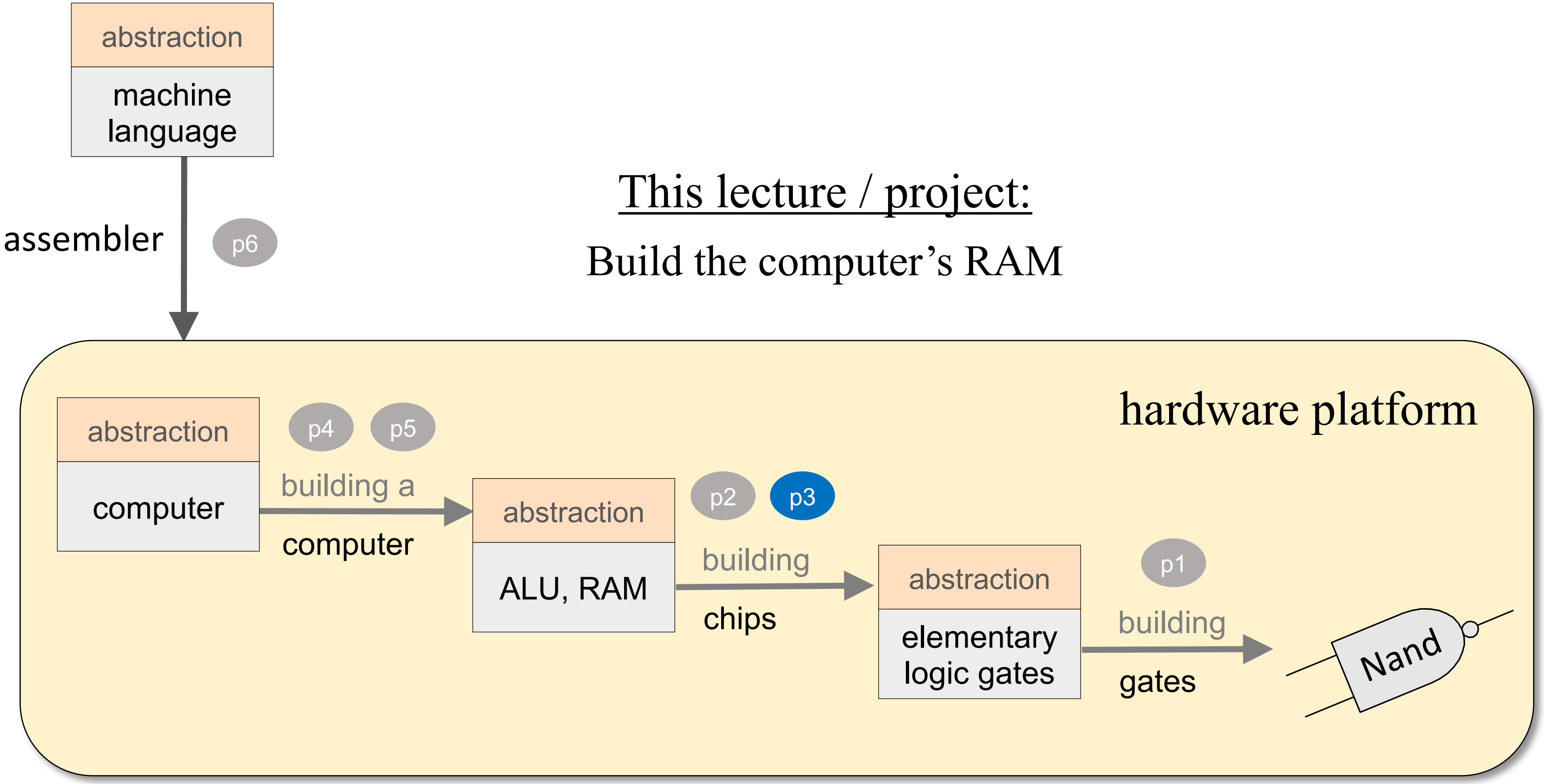
That's It!

Go Do Project 3!

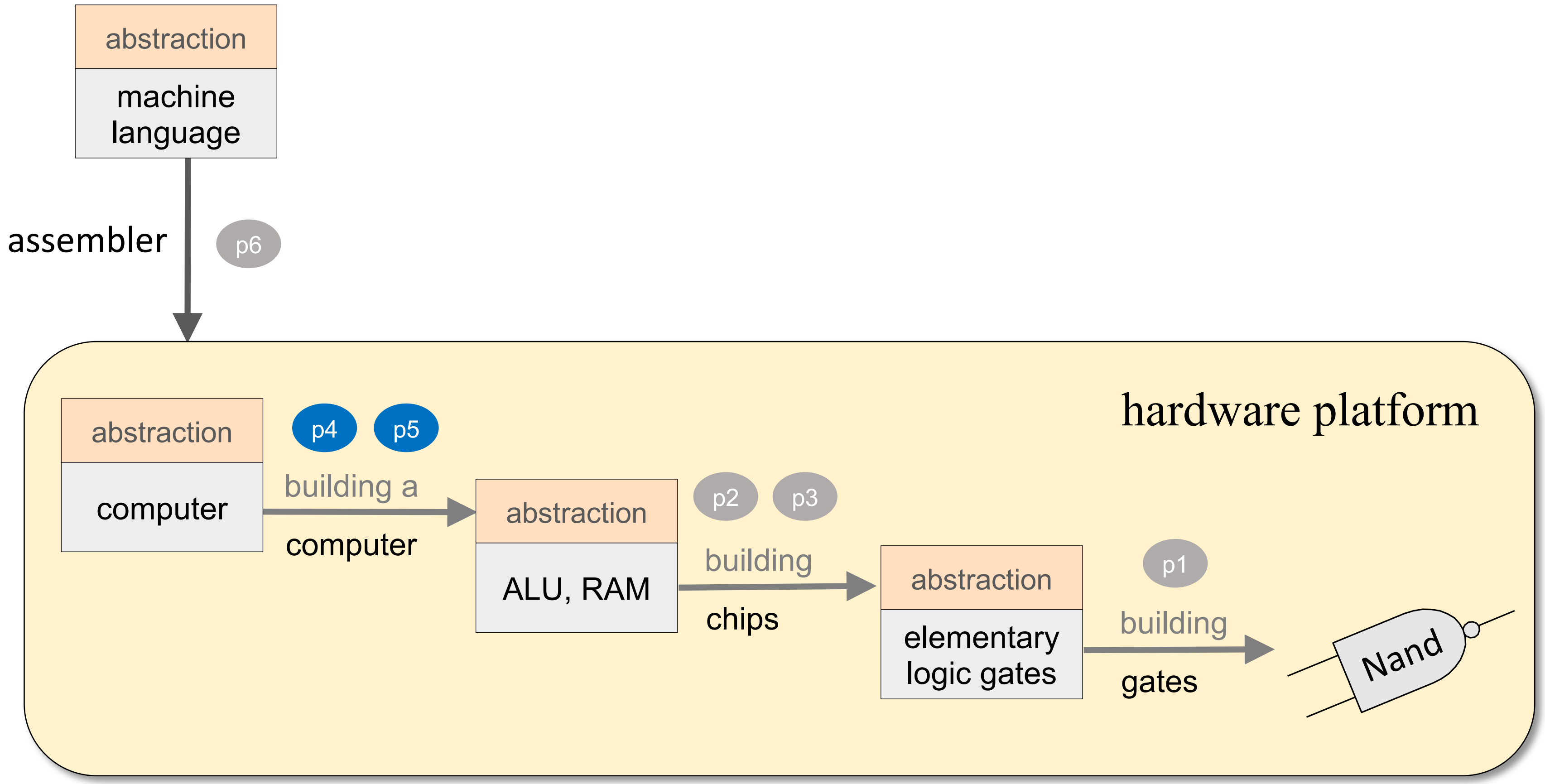
Nand to Tetris Roadmap: Hardware

This lecture / project:

Build the computer's RAM



Nand to Tetris Roadmap: Hardware



<u>Next two lectures / projects:</u>

- We'll build the computer (p5)
- But first, we'll get acquainted with its instruction set / machine language (p4).