Chapter 3

Memory

These slides support chapter 3 of the book

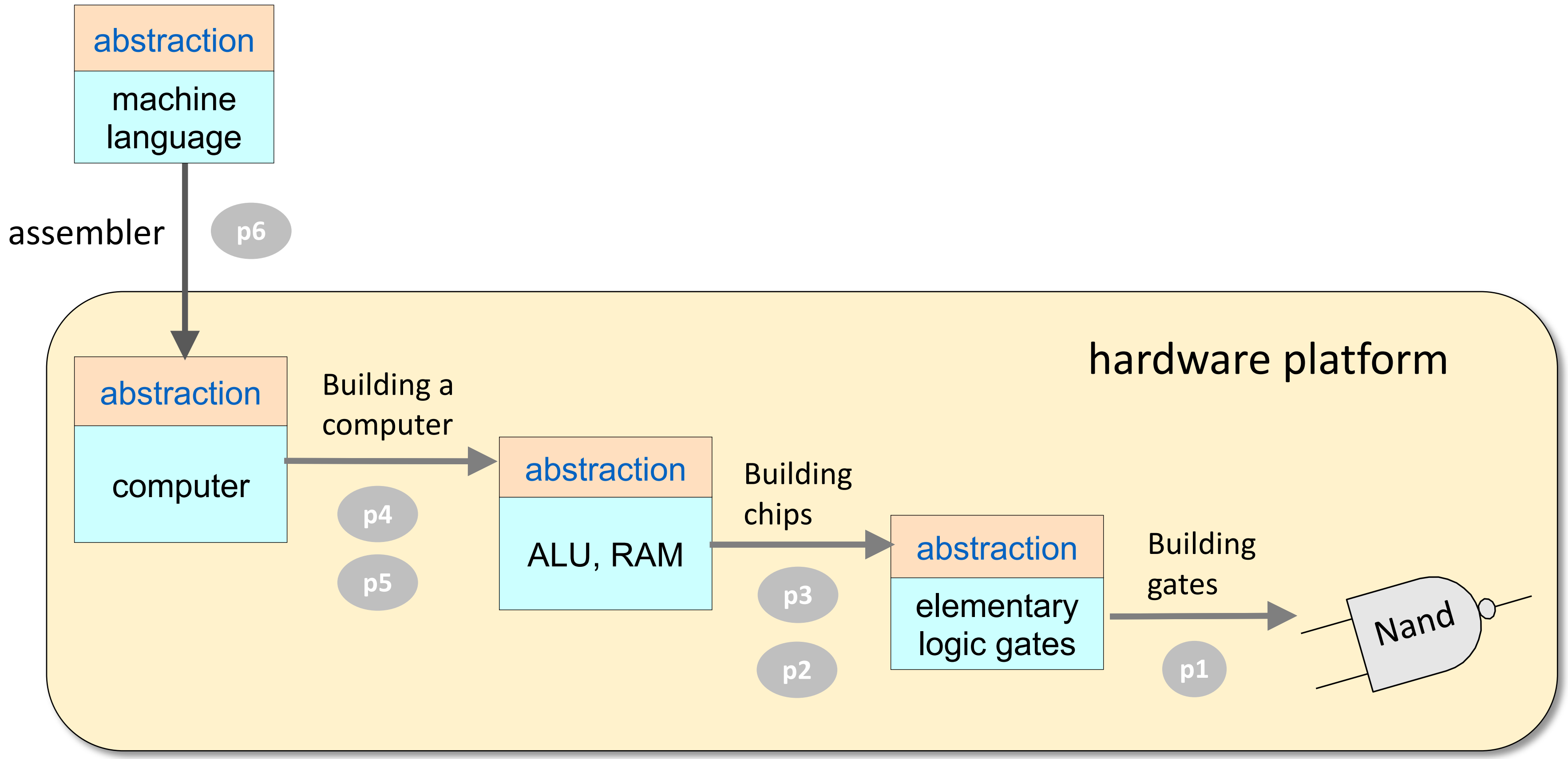
The Elements of Computing Systems

(1st and 2nd editions)

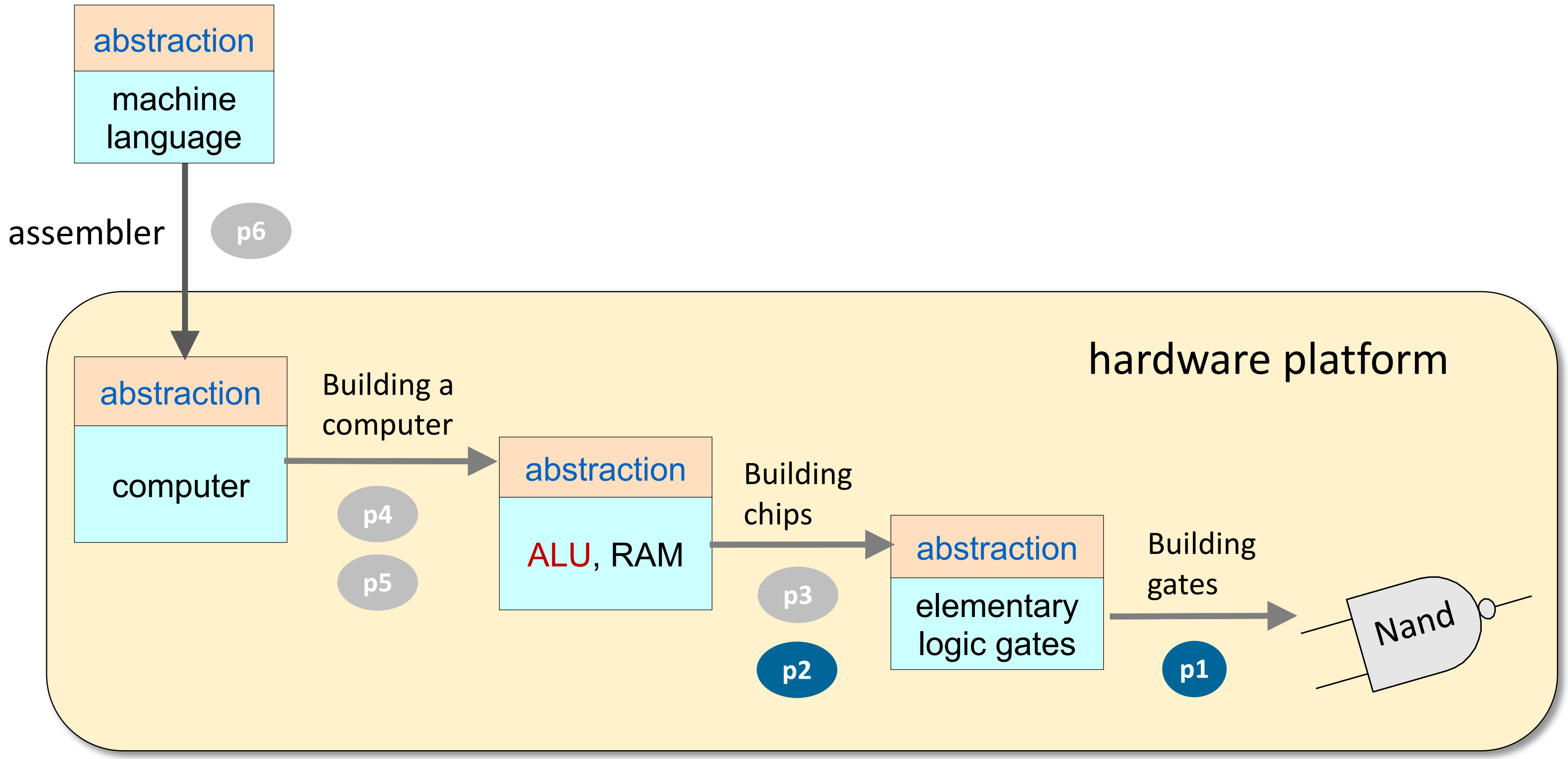
By Noam Nisan and Shimon Schocken

MIT Press

Nand to Tetris Roadmap: Hardware



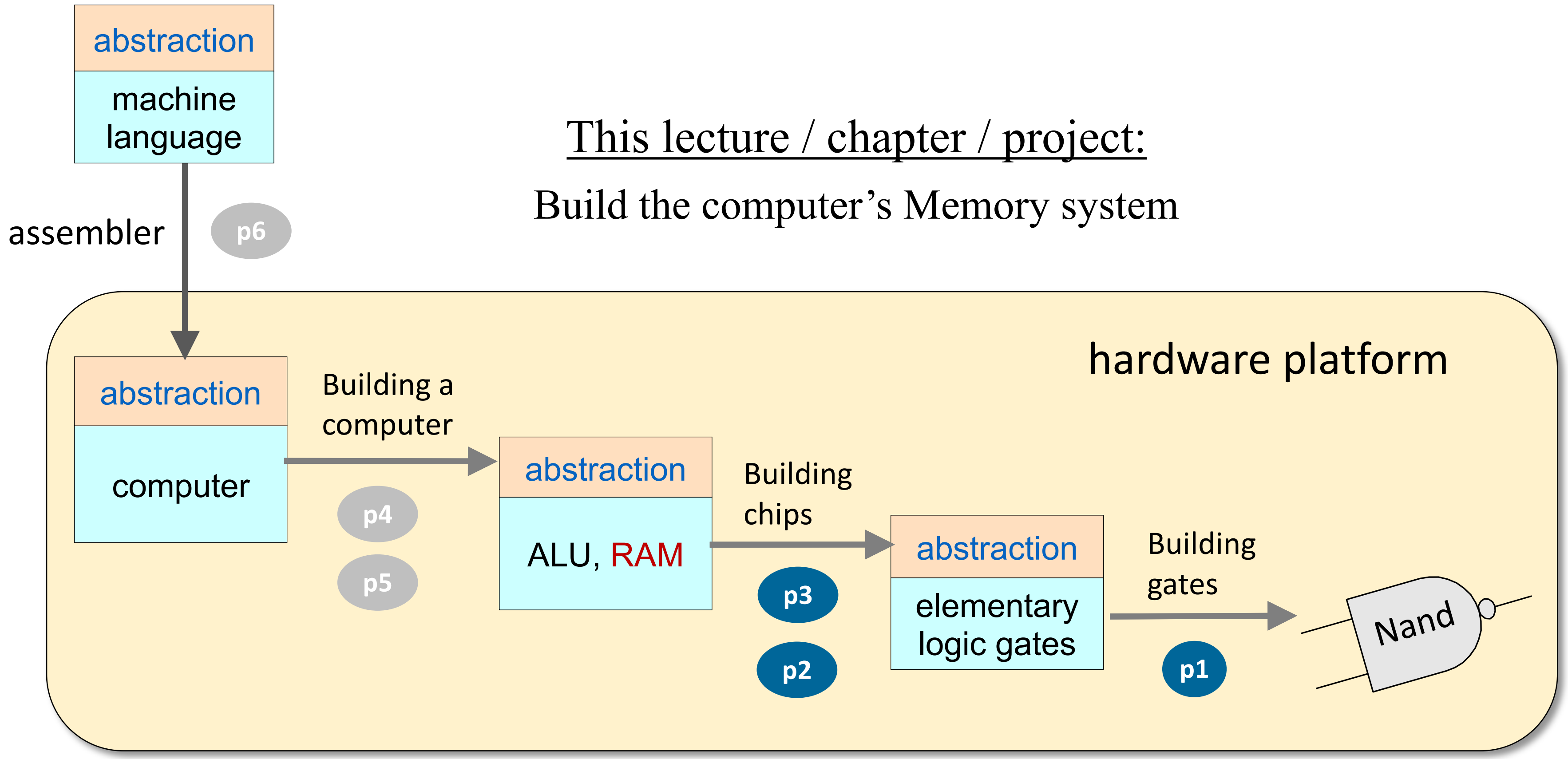
Nand to Tetris Roadmap: Hardware



Project 1: Build basic logic gates

Project 2: Build the ALU

Nand to Tetris Roadmap: Hardware



This lecture / chapter / project:

Build the computer's Memory system

Project 1: Build basic logic gates

Project 2: Build the ALU

A common theme in computer science

- We present a simple model (the simpler, the better)
- We explore the model's power:
  - What the model can do
  - What it cannot do
- We then extend the model, to make 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 powerful chips, like an ALU

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

Limitations

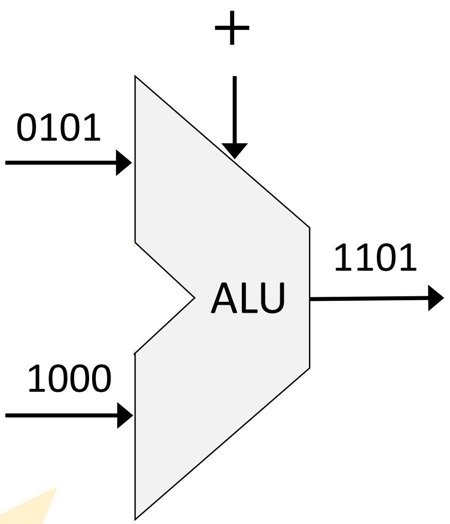
- Logic gates cannot store information (bits) over time
- Feedback loops are not allowed: A chip's output cannot serve as its input
- Logic gates can handle only inputs of a fixed size.
  For example, we can build an Or3 gate, and an Or4 gate, and so on, but we cannot build a single gate that computes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

<u>Software needs:</u>

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

Example (variables):

```
x = 17
```

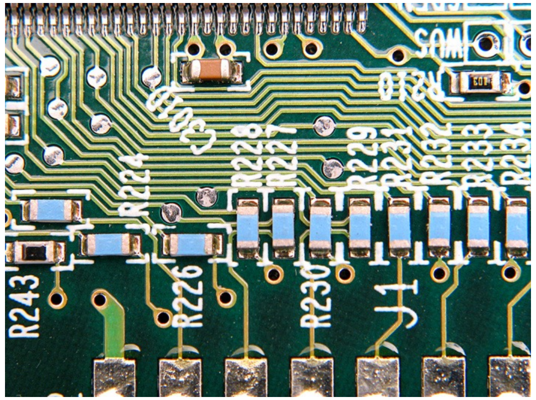
- The hardware must be able to do things, one at a time (sequentially):

Example (iteration):

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

<u>Hardware needs:</u>

- 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
```

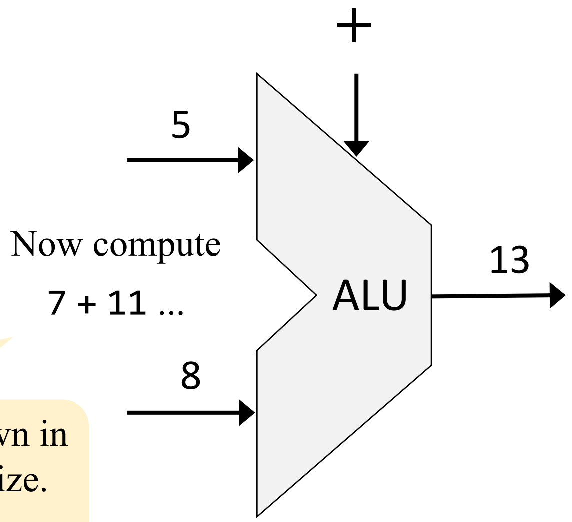
- The hardware must be able to do things, one at a time (sequentially):

Example (iteration):

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

Hardware needs:

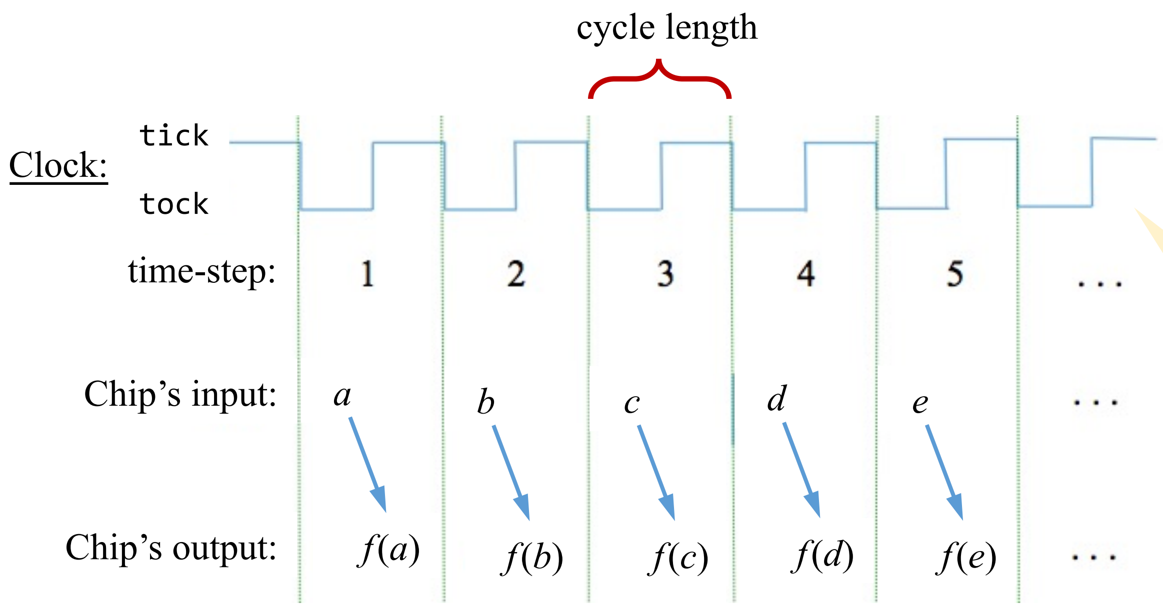
- The hardware must handle the *physical time delays* associated with *computing* and *moving* data from one chip to another.



It will take some time before 7 and 11 will settle down in the input ports, 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*



As it turns out, the clock can also be used to help implementing 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's outputs only at the end of cycles (time-steps), ignoring what happens within cycles
- Details later.

Memory

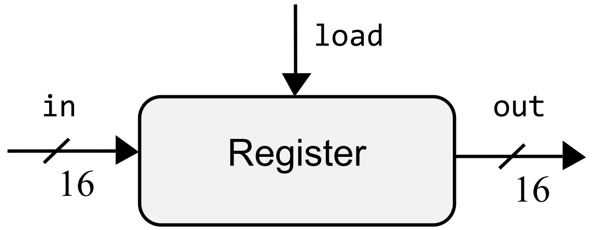
Memory: The faculty of the brain by which data or information is encoded, stored, and retrieved when needed.

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 abstractions:

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

1 2 3 4 5 6 7 ... 205 206 207 208 209 210 211 ... time

(cycles)

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

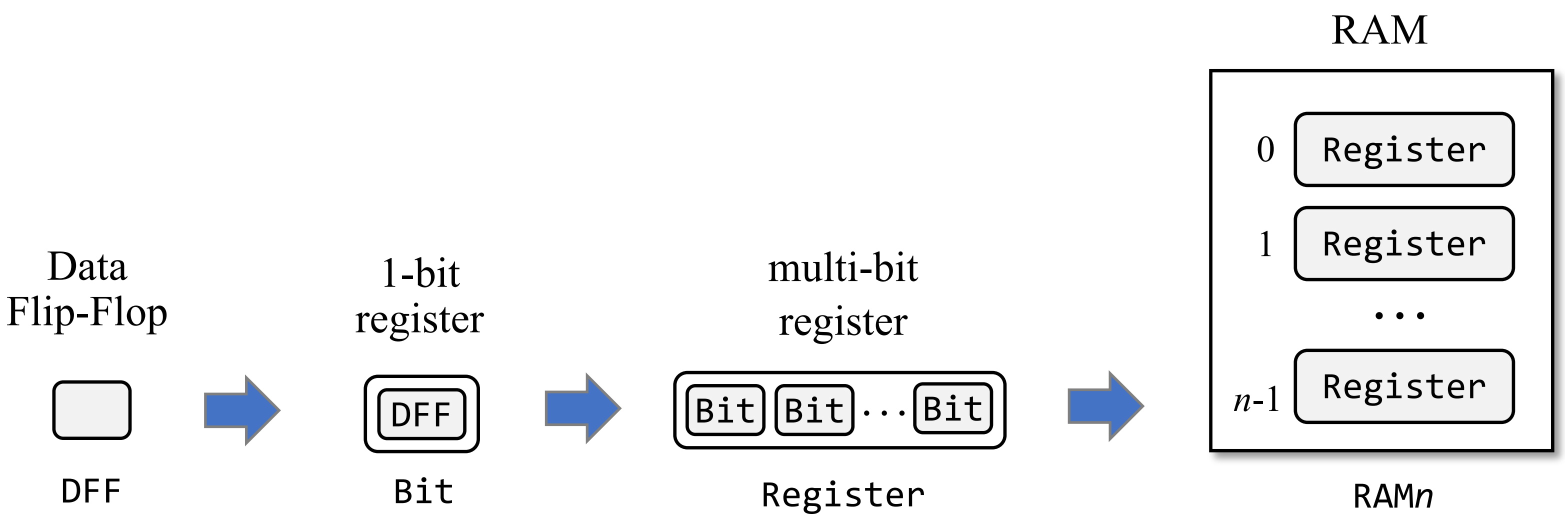
loading storing

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

loading storing

The challenge: Building chips that realize this functionality.

Memory



The challenge: Building chips that realize this functionality.

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

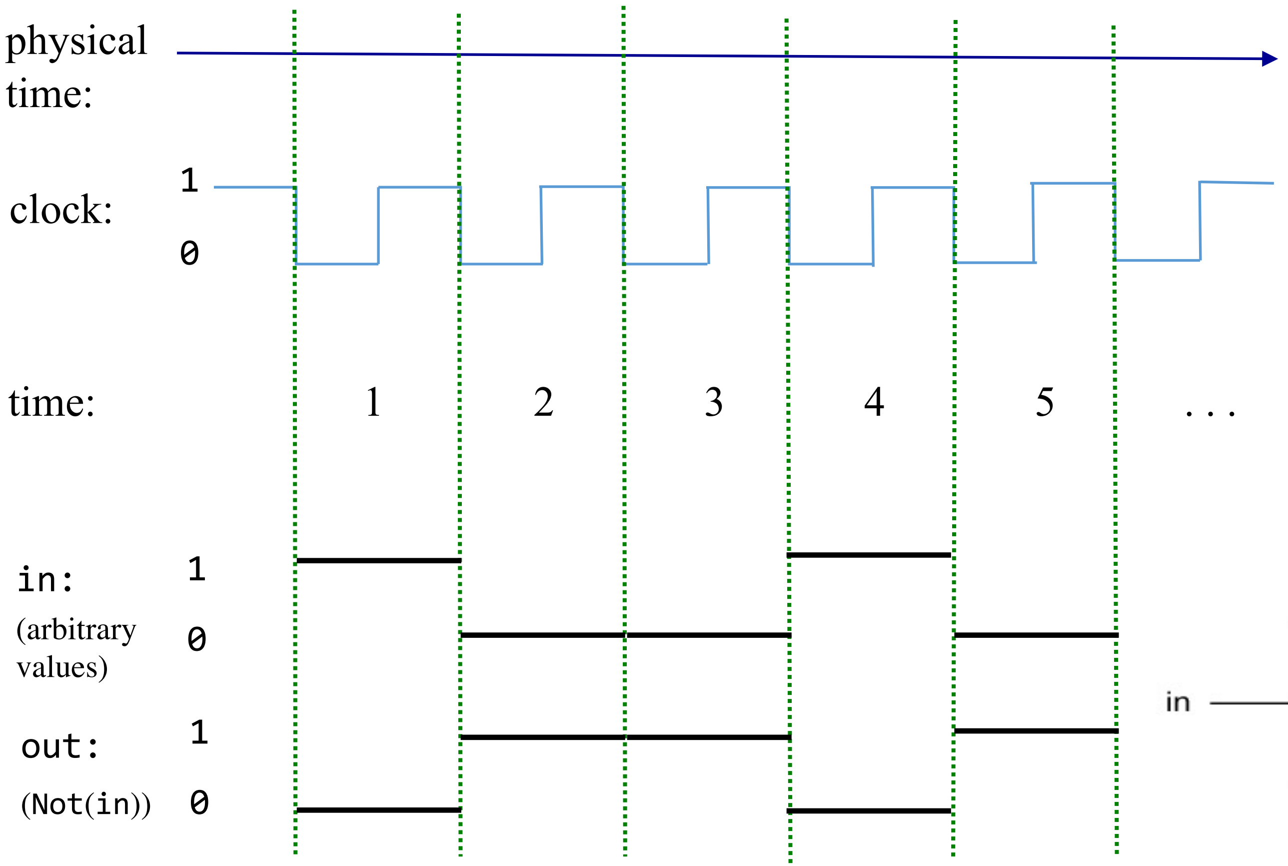
<u>Abstraction</u>

- Representing time
- Clock
- Registers
- RAM
- Counters

<u>Implementation</u>

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

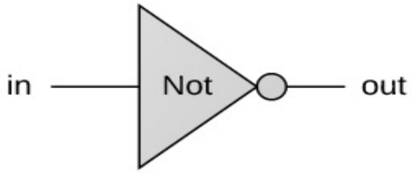
Chip behavior over time (example: Not gate)



<u>Arrow of time:</u>

Continuous

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

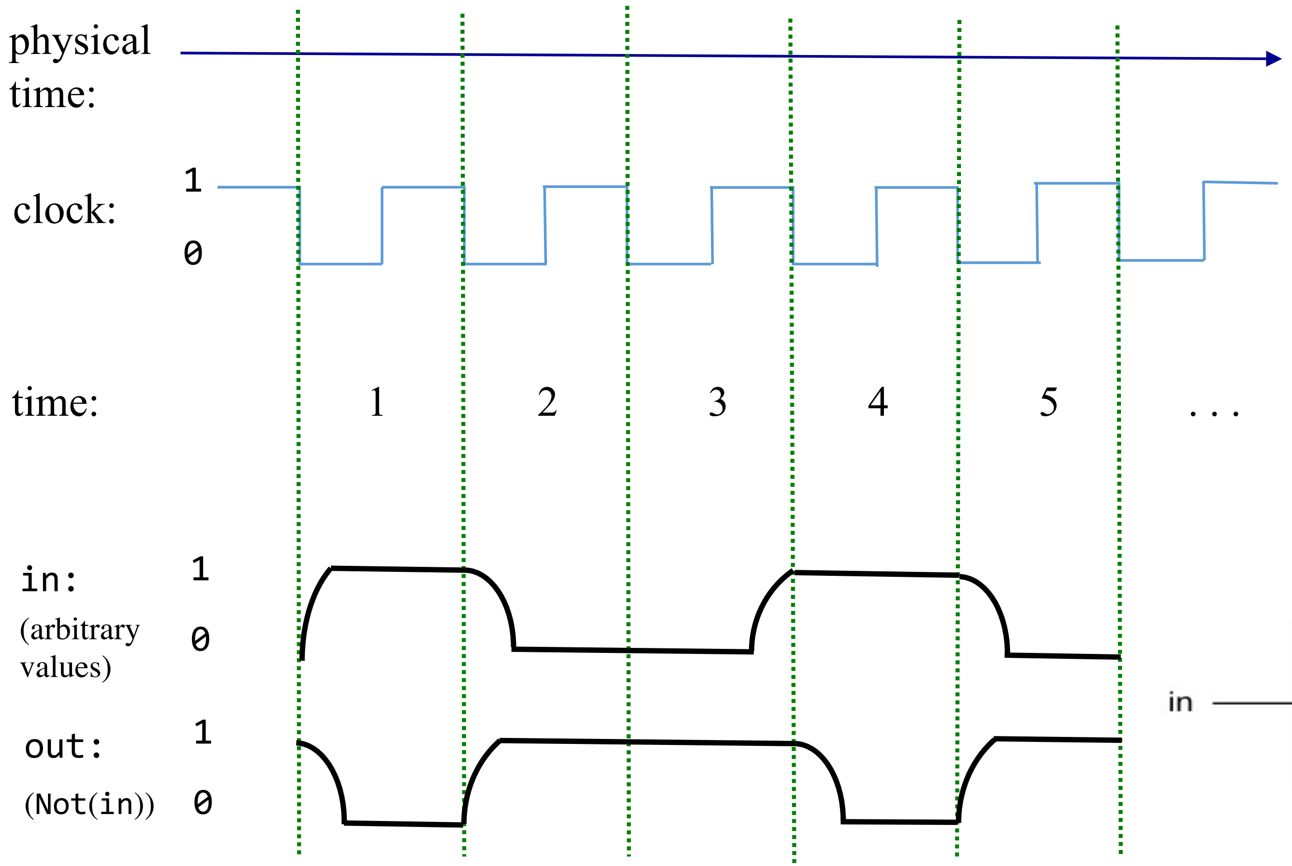


(example)

<u>Desired / idealized behavior of the `in` and `out` signals:</u>

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

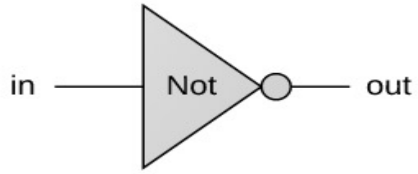
Chip behavior over time (example: Not gate)



Arrow of time:

Continuous

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

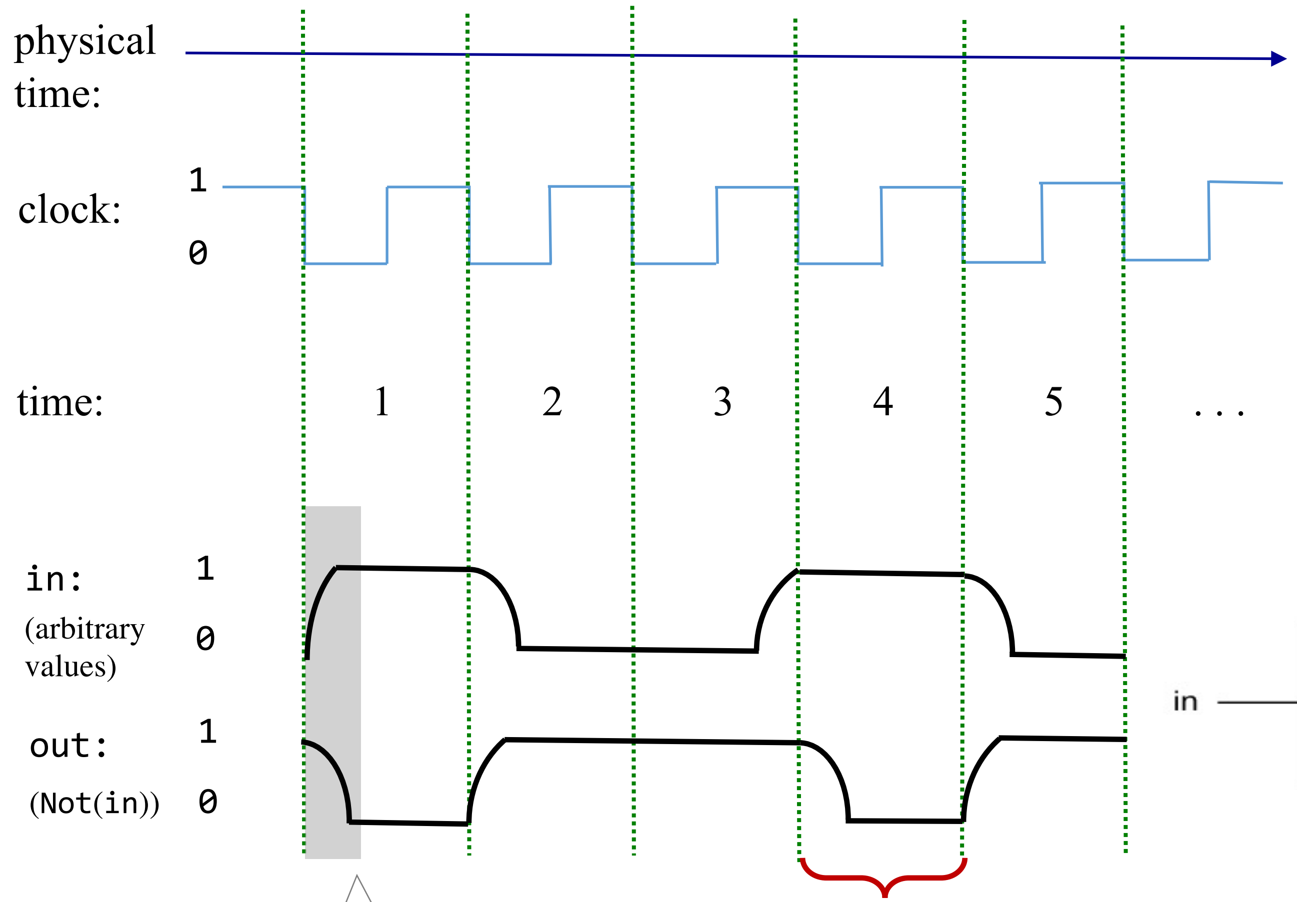


(example)

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

Influenced by physical time delays

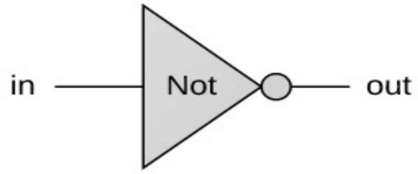
Chip behavior over time (example: Not gate)



Arrow of time:

Continuous

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

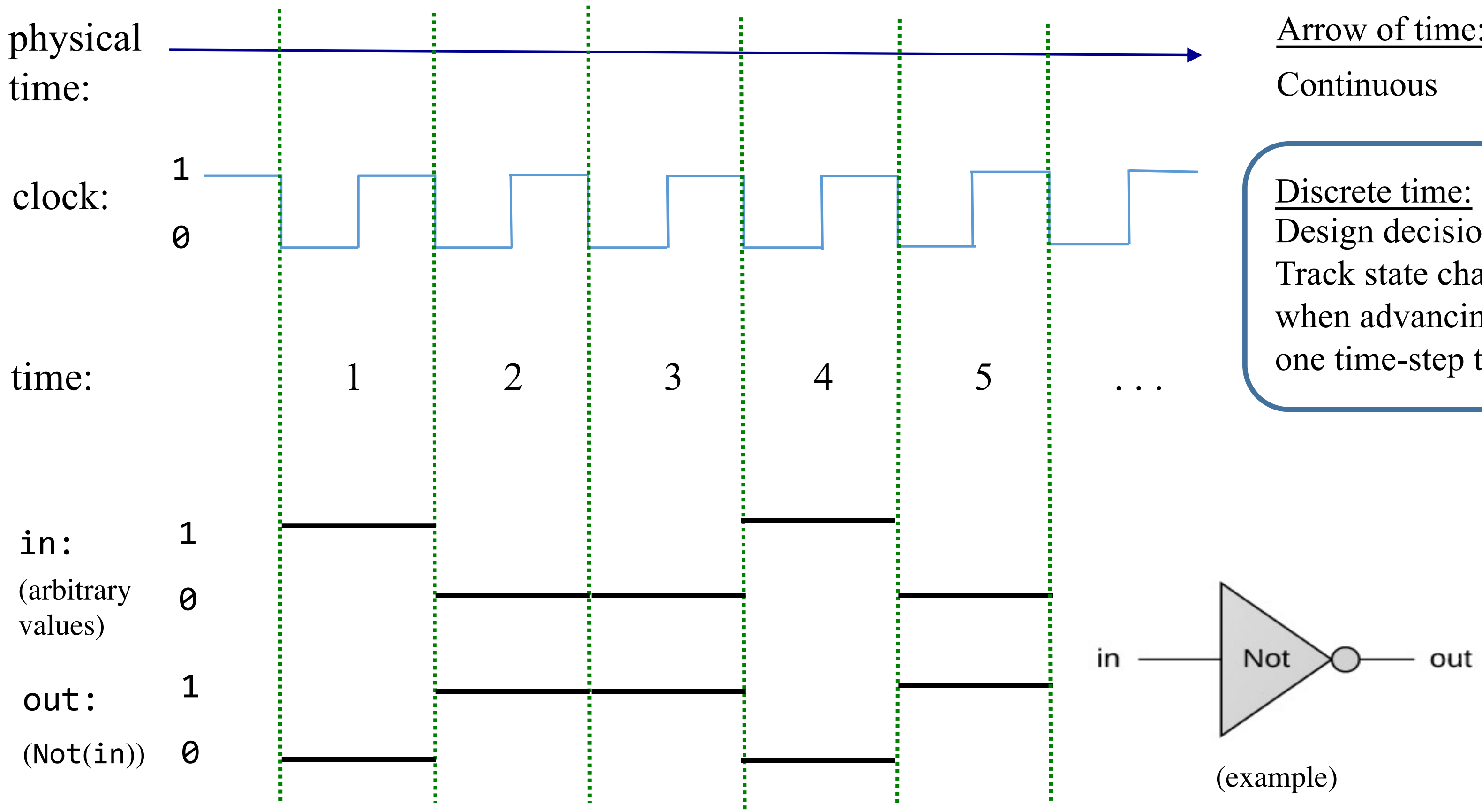


(example)

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

Resulting effect:

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

Chapter 3: Memory

Abstraction

✓ Representing time

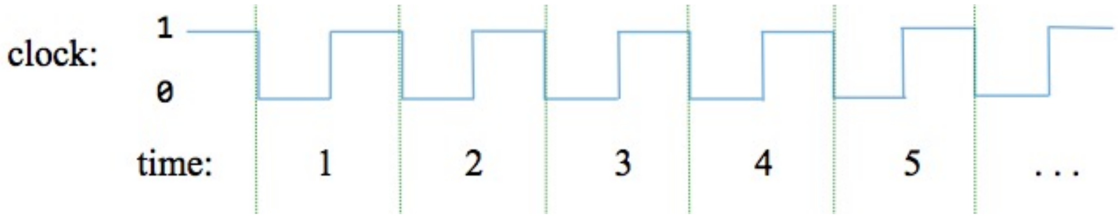
➡ Clock

- Registers
- RAM
- Counters

Implementation

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

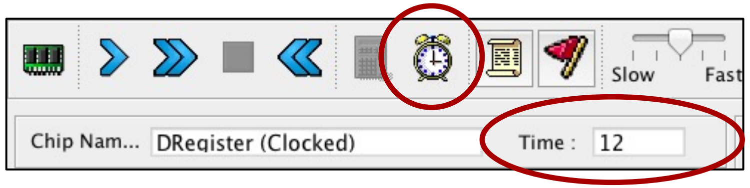
Clock: Simulated implementation



Interactive simulation

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

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

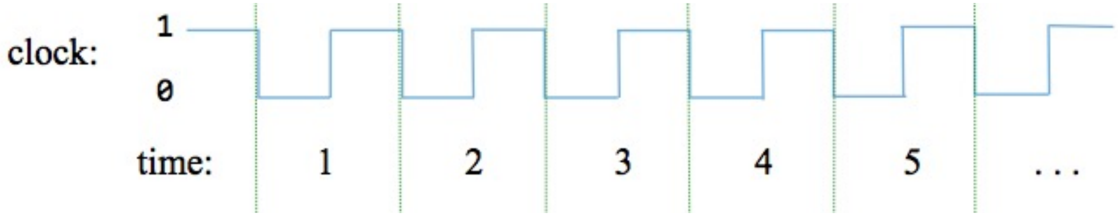


Script-based simulation

"`tick`" and "`tock`" commands can be 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

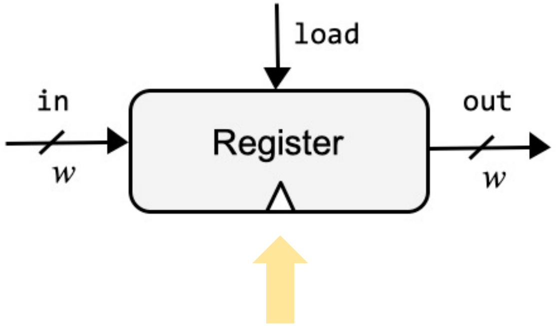


Physical clock

- 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)

- The oscillator's output is connected to all the time-based (clocked) chips in the computer



Chip diagram convention:

A triangle icon represents a clock signal input

Chapter 3: Memory

<u>Abstraction</u>

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

<u>Implementation</u>

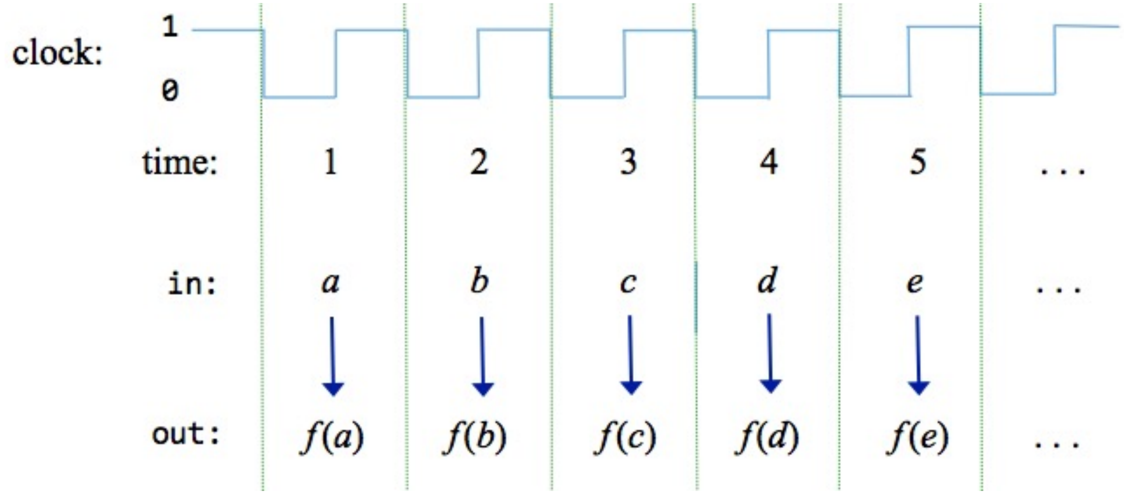
- Data Flip Flop
- Registers
- RAM
- Project 3: Chips
- Project 3: Guidelines

Combinational logic / Sequential logic

Combinational logic:

The output depends on the current inputs

The clock is used to stabilize outputs

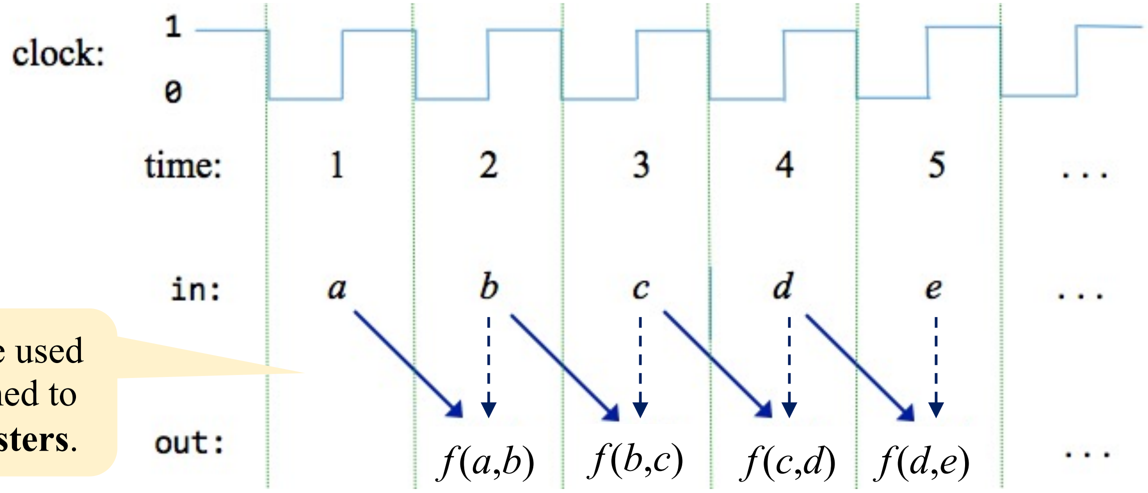


Sequential logic:

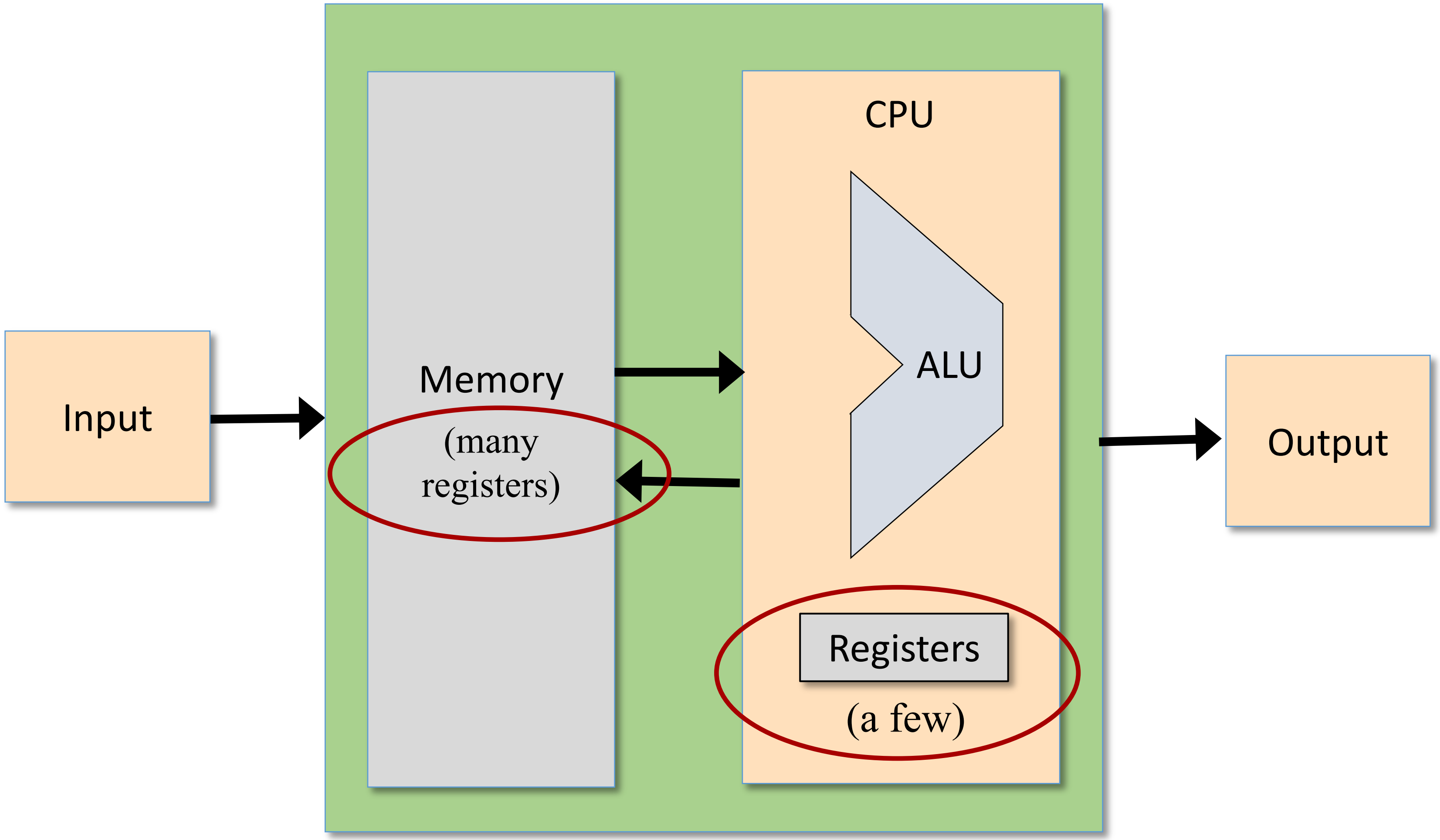
The output depends on:

- Previous inputs
- Current inputs (optionally)

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

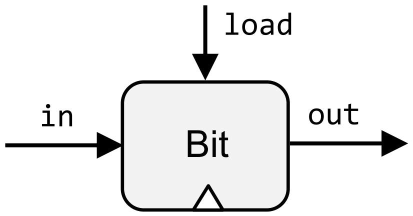


Registers

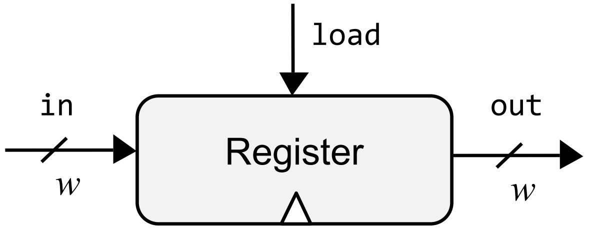


Computer Architecture

Registers



1-bit register



multi-bit register

<u>Designed to:</u>

- "Store" / "remember" / "maintain" / "persist" a value , until...
- "Instructed" to "load", and then "store", another value.

time:

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

loading

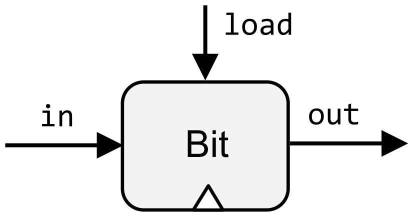
maintaining state

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

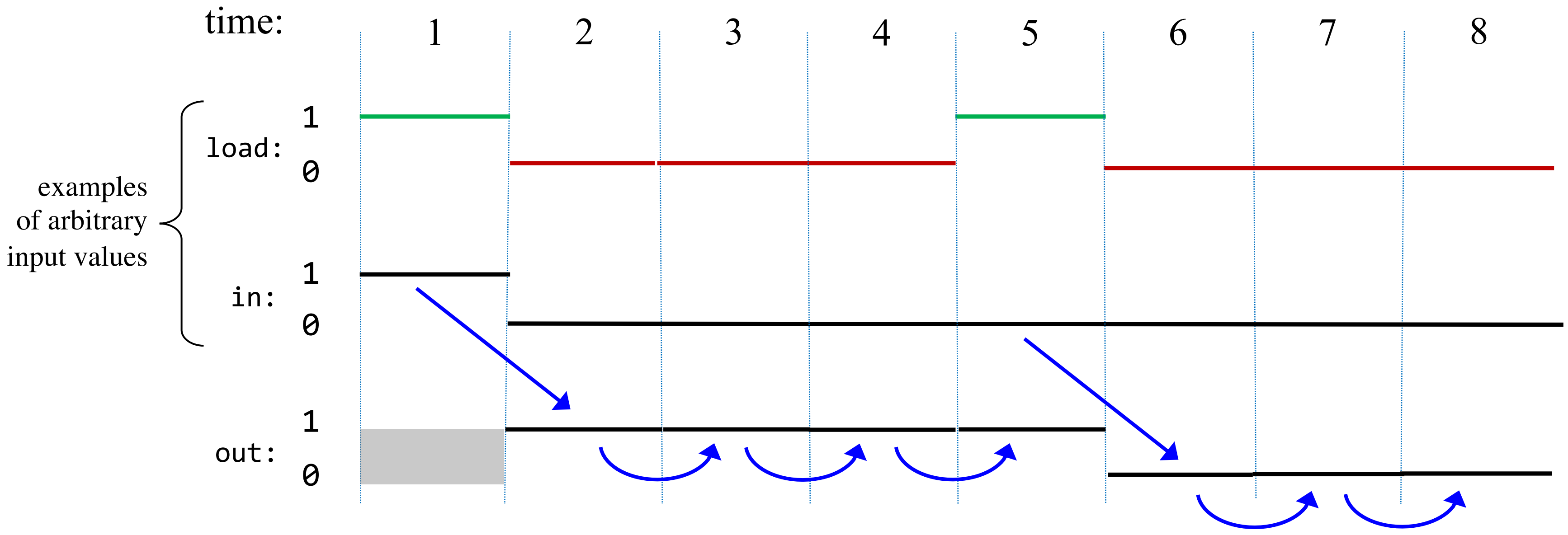
loading

maintaining state

1-Bit Register

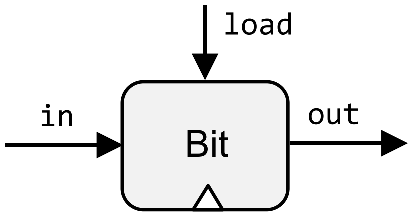


if load($t-1$) then out(t) = in($t-1$)
else out(t) = out($t-1$)



Resulting behavior: "loading" "storing" "loading" "storing"

1-Bit Register



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

<u>Usage:</u>

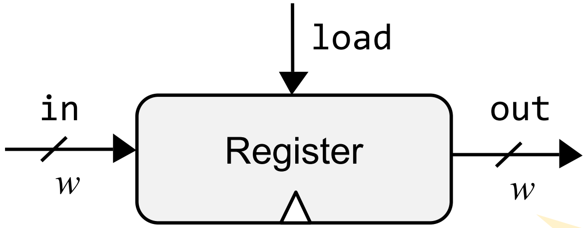
To read:

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

Multi-bit Register



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

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

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

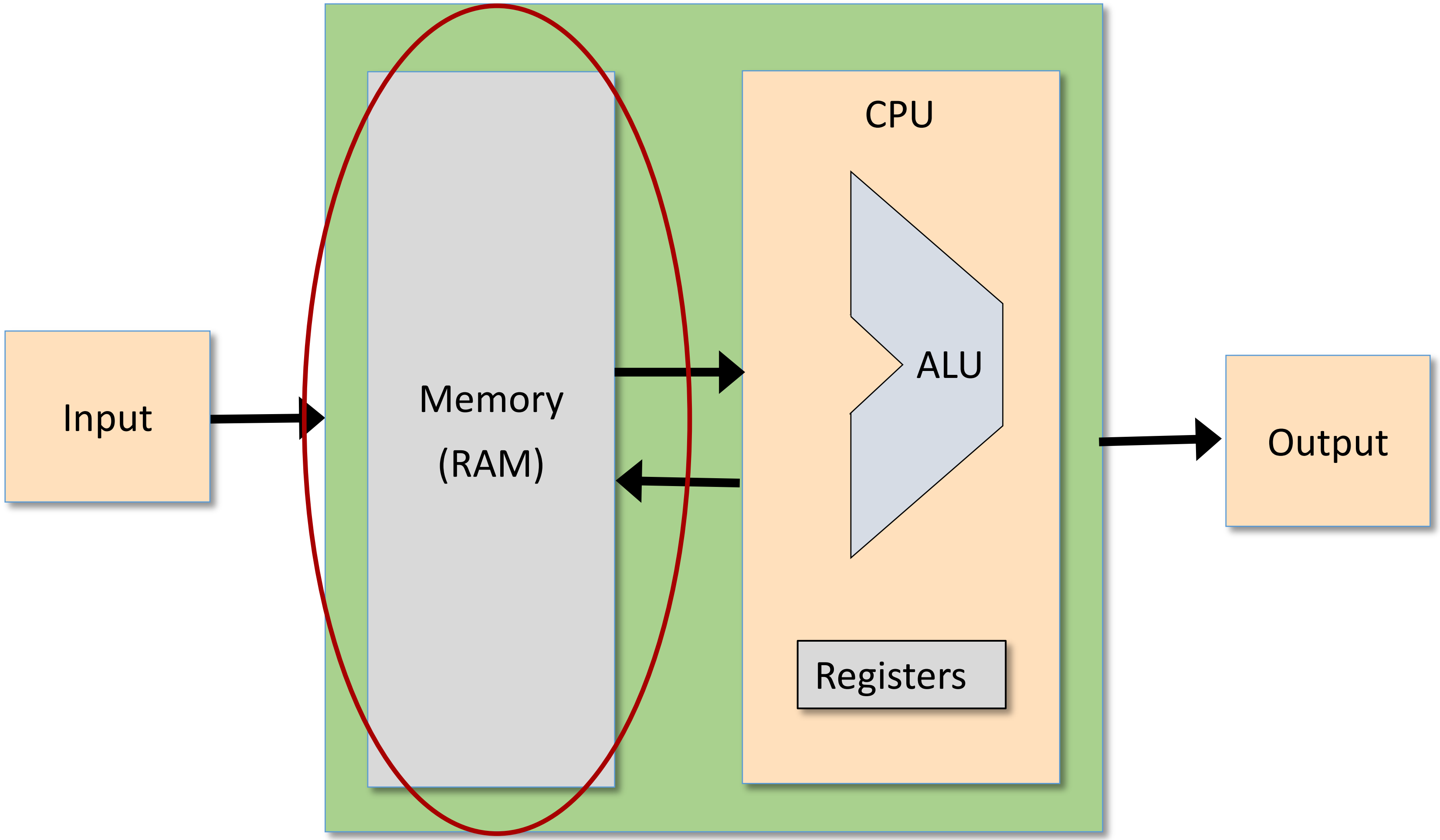
<u>Abstraction</u>

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

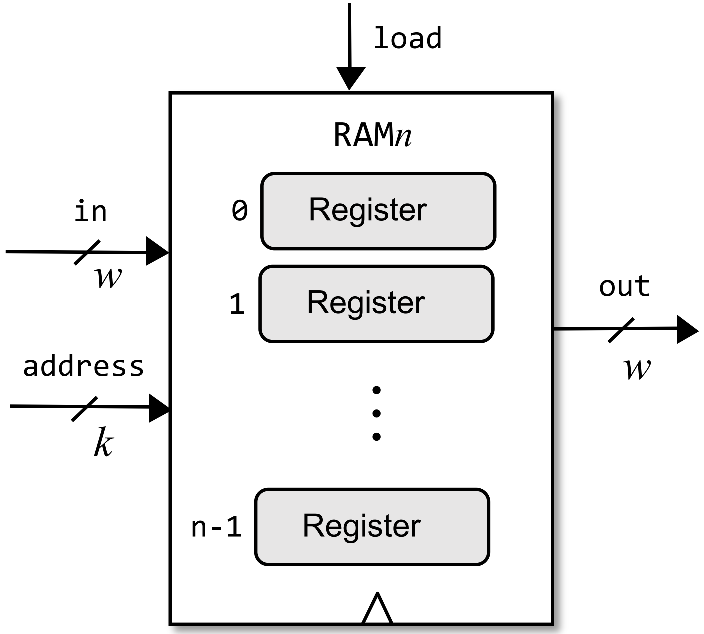
<u>Implementation</u>

- 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?

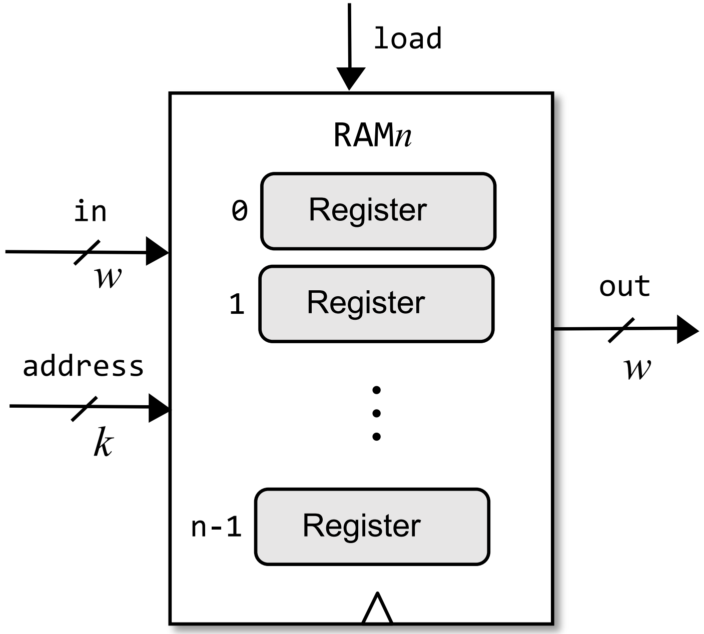
Answer:

$k = log_2 n$

Abstraction: A sequence of *n* addressable, *w*-bit registers, with addresses 0 to *n*-1

Word width: Typically 16, 32, 64 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, 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*,

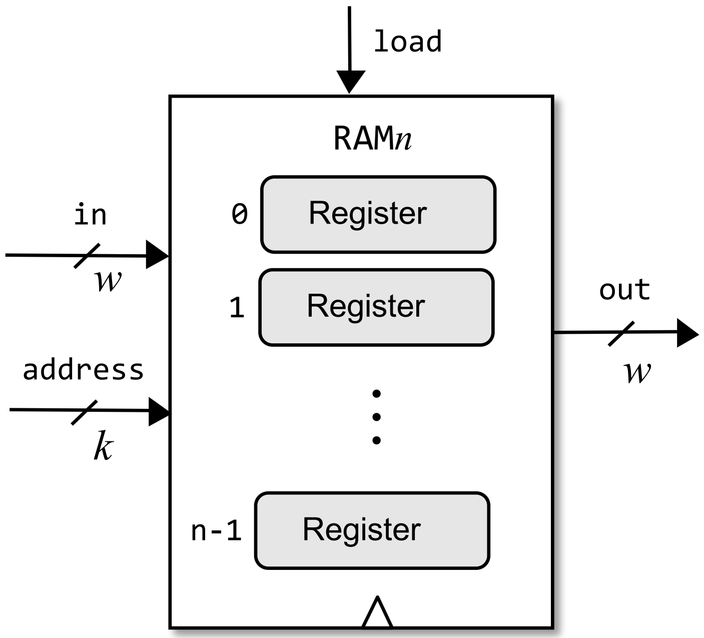
set `in` = *v*,

set `load` = `1`

Result: `RAM[`*i*`]` ← *v*

From the next time-step onward, `out` will emit *v*

RAM



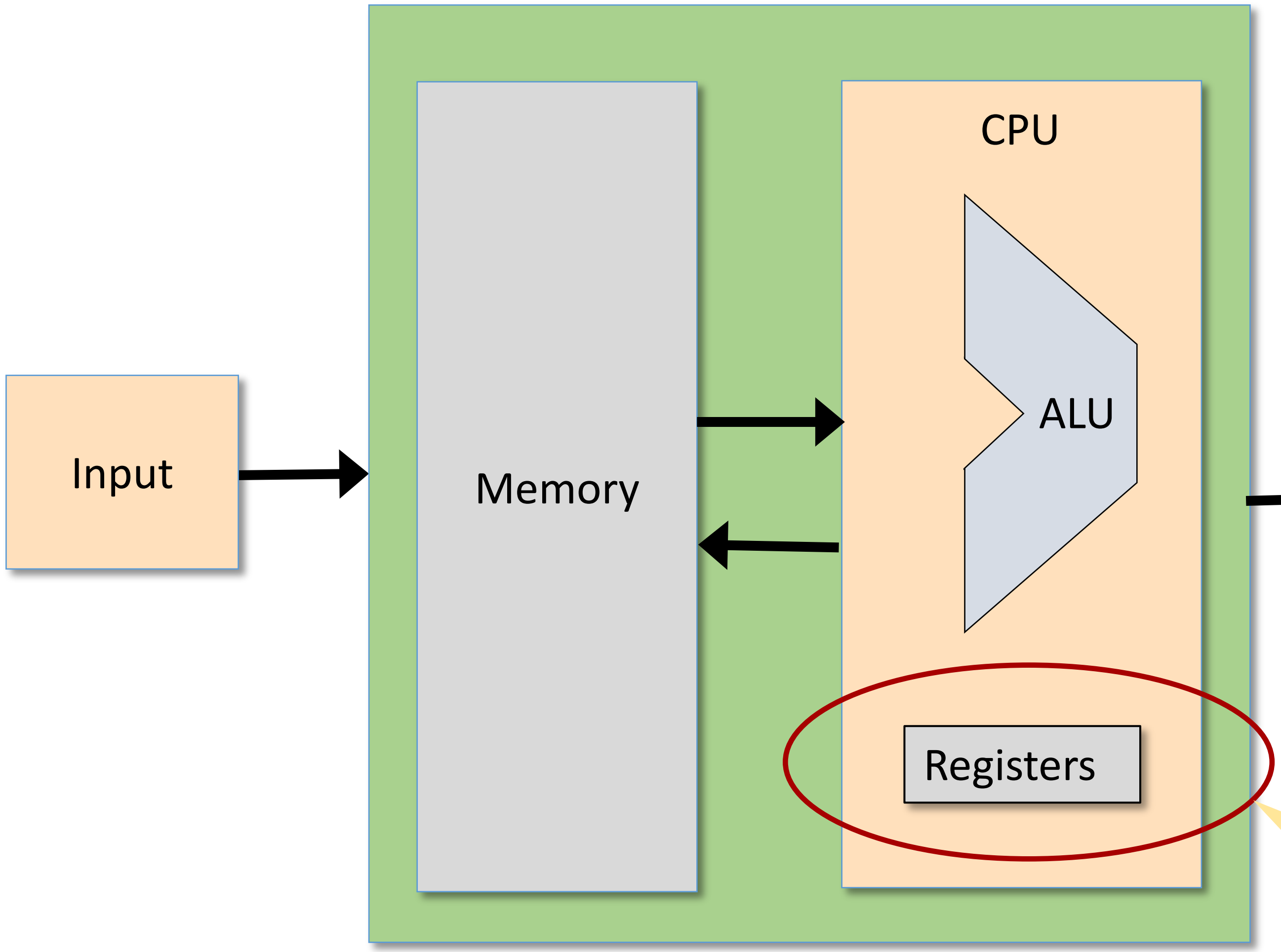
Why "Random Access Memory"?

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

Computer architecture



One of the CPU registers acts as a "program counter"

Counter

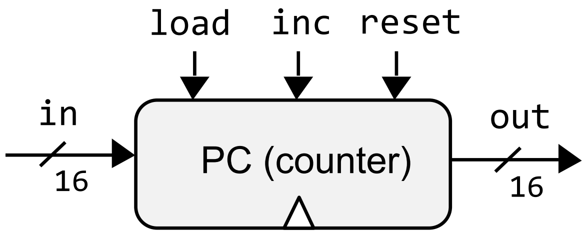
- Later in the course (chapter 5), we will see that the computer must keep track of which instruction should be fetched and executed next
- This task is regulated by a register typically called `Program Counter`
- We'll use the `PC` to store the address of the instruction that should be fetched and executed next
- The `PC` should support three abstractions:

<u>Reset:</u> fetch the first instruction `PC = 0`

<u>Next:</u> fetch the next instruction `PC++`

<u>Goto:</u> 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

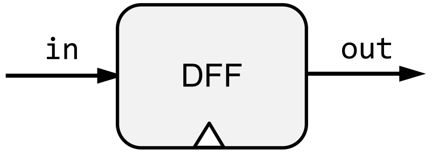
Implementation

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

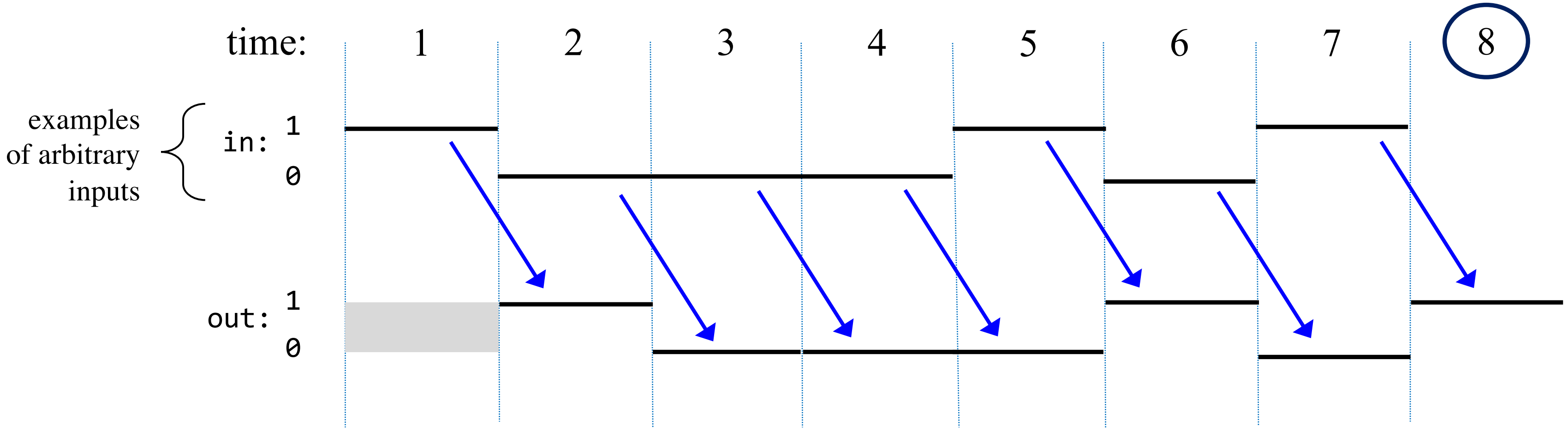
DFF

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

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



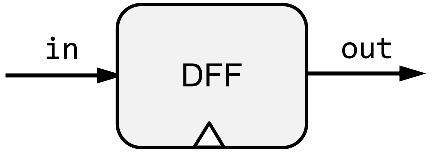
out(t) = in($t-1$)



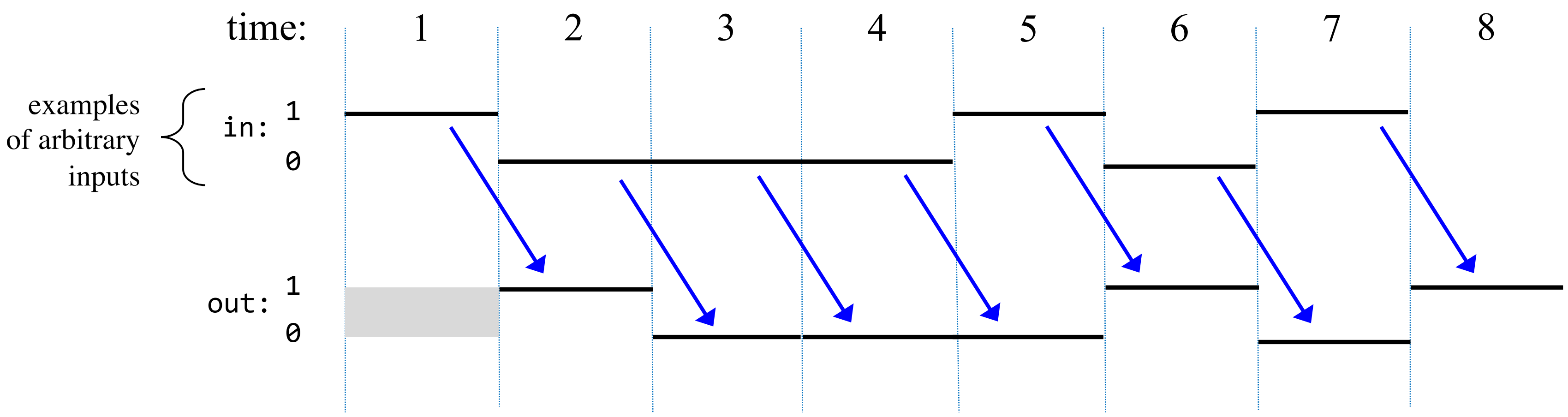
From DFF to a 1-bit register

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

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

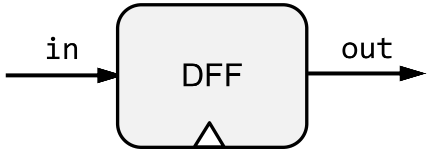


out(t) = in($t-1$)



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



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

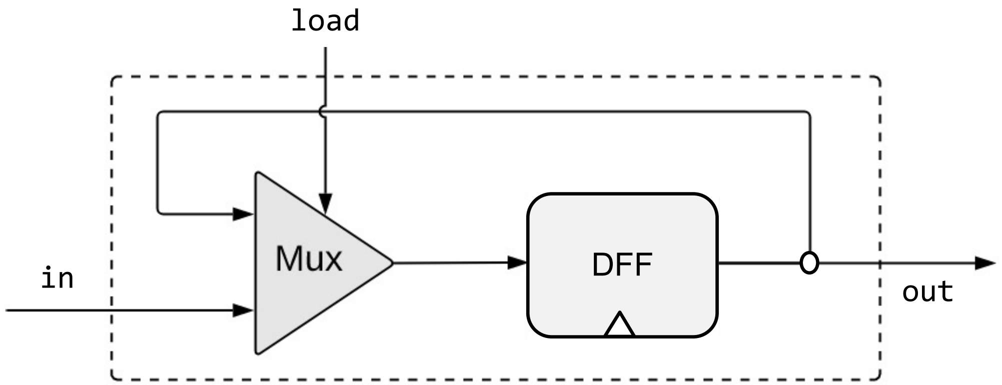
We have to realize a “loading” behavior and a ”storing” behavior, and be able to select between these two states

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

1-bit Register

Stores one bit over time



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

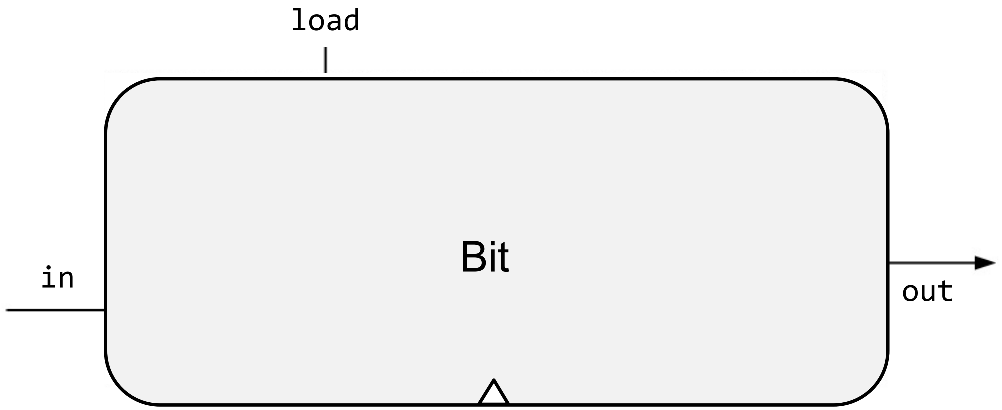
We have to realize a “loading” behavior and a ”storing” behavior, and be able to select between these two states

Behavior

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

else   the register maintains its current value

Register



1-bit Register

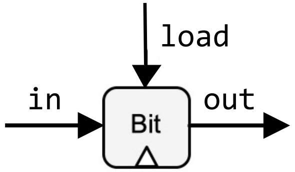
Stores one bit over time

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

Register

1-bit Register

Stores one bit over time



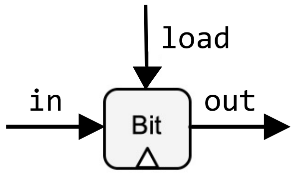
zoom out...

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

Register

<u>1-bit Register</u>

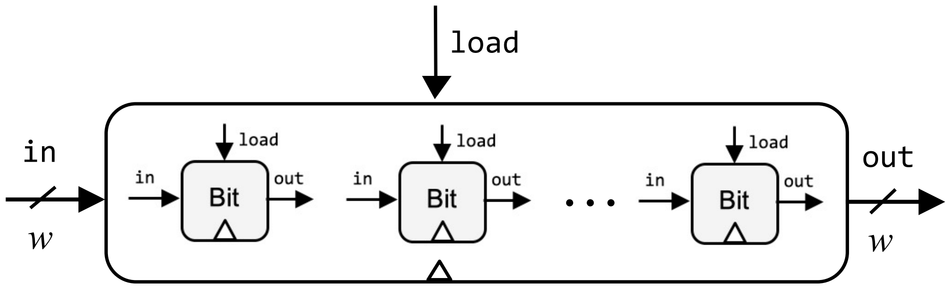
Stores one bit over time



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

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

Stores *w* bits over time



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

Chapter 3: Memory

Abstraction

- Representing time
- Clock
- Registers
- RAM
- Counters

Implementation

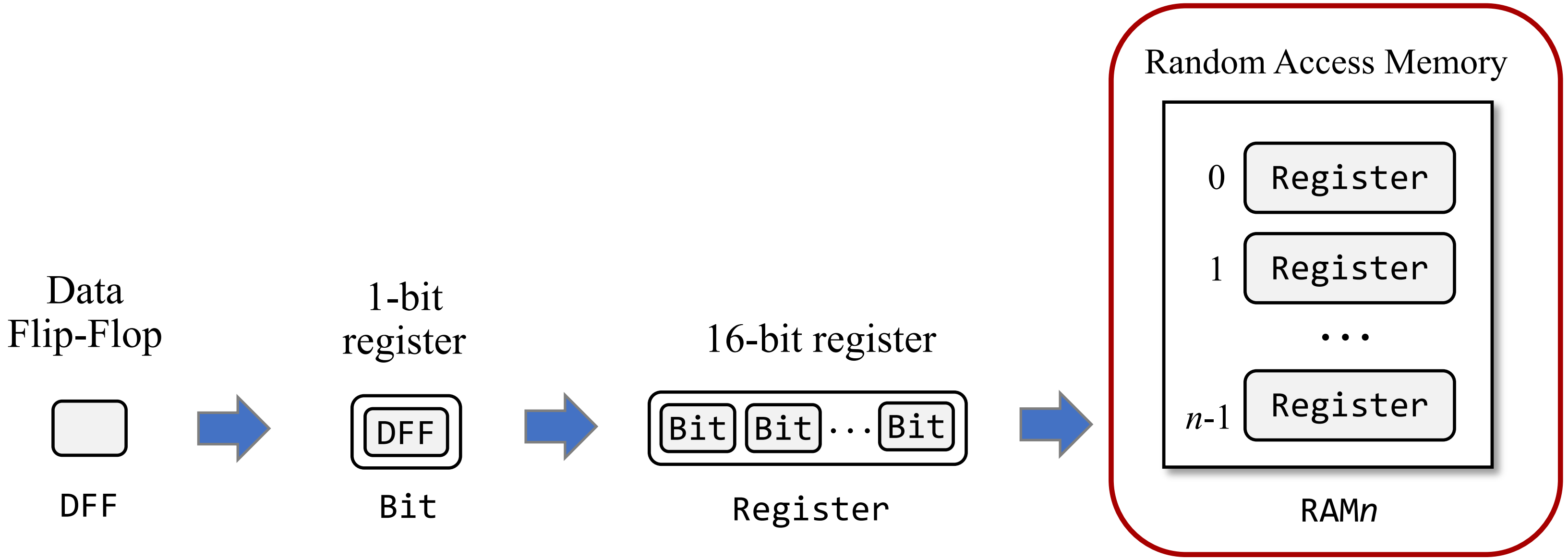
✓ Data Flip Flop

✓ Registers

➡ RAM

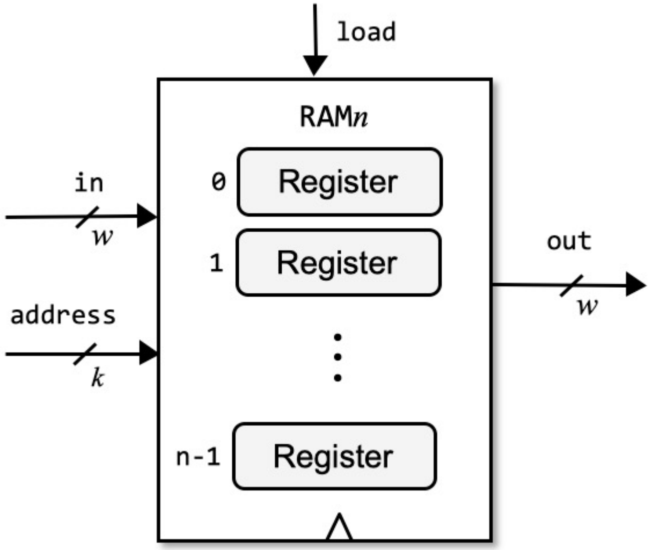
- Project 3: Chips
- Project 3: Guidelines

Memory hierarchy



RAM: Abstraction

RAM of n registers:



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` $= v$,

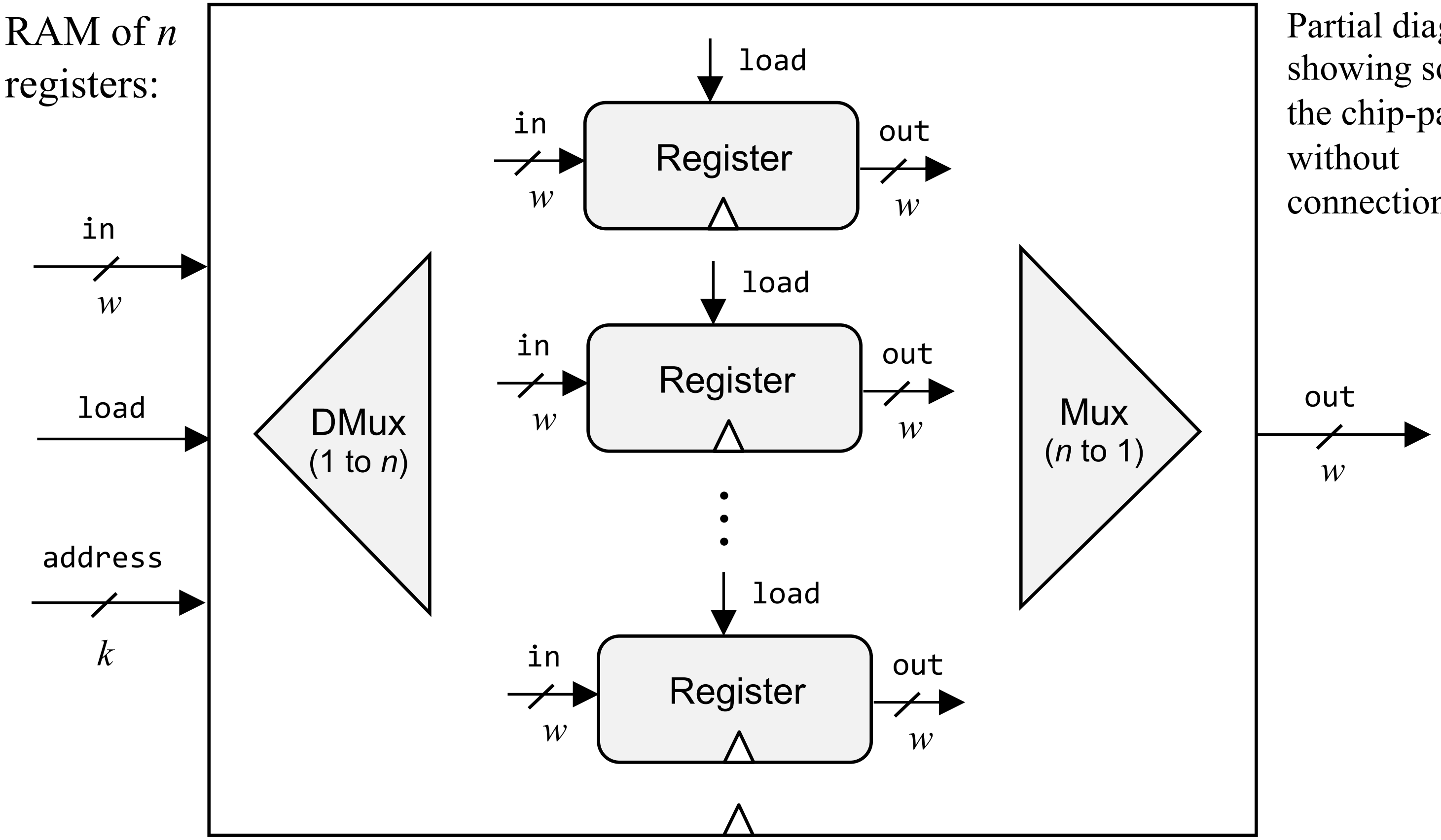
set `load` $=$ `1`

Result: `RAM[`i`]` ← 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

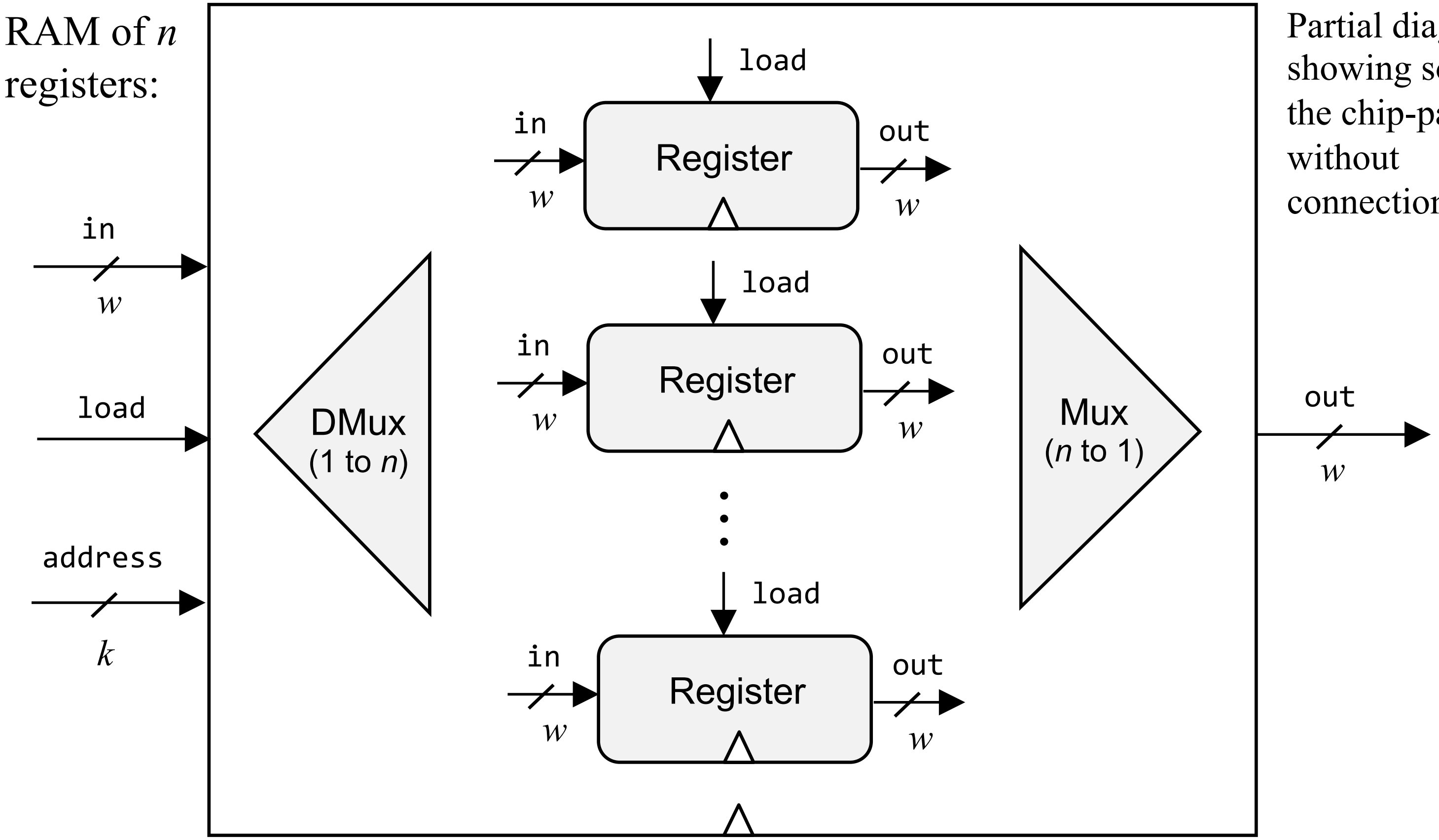
Reading: Can be realized using a Mux

Writing: Can be realized using a DMux

Connections?
You figure it out

RAM: Implementation

RAM of *n* registers:

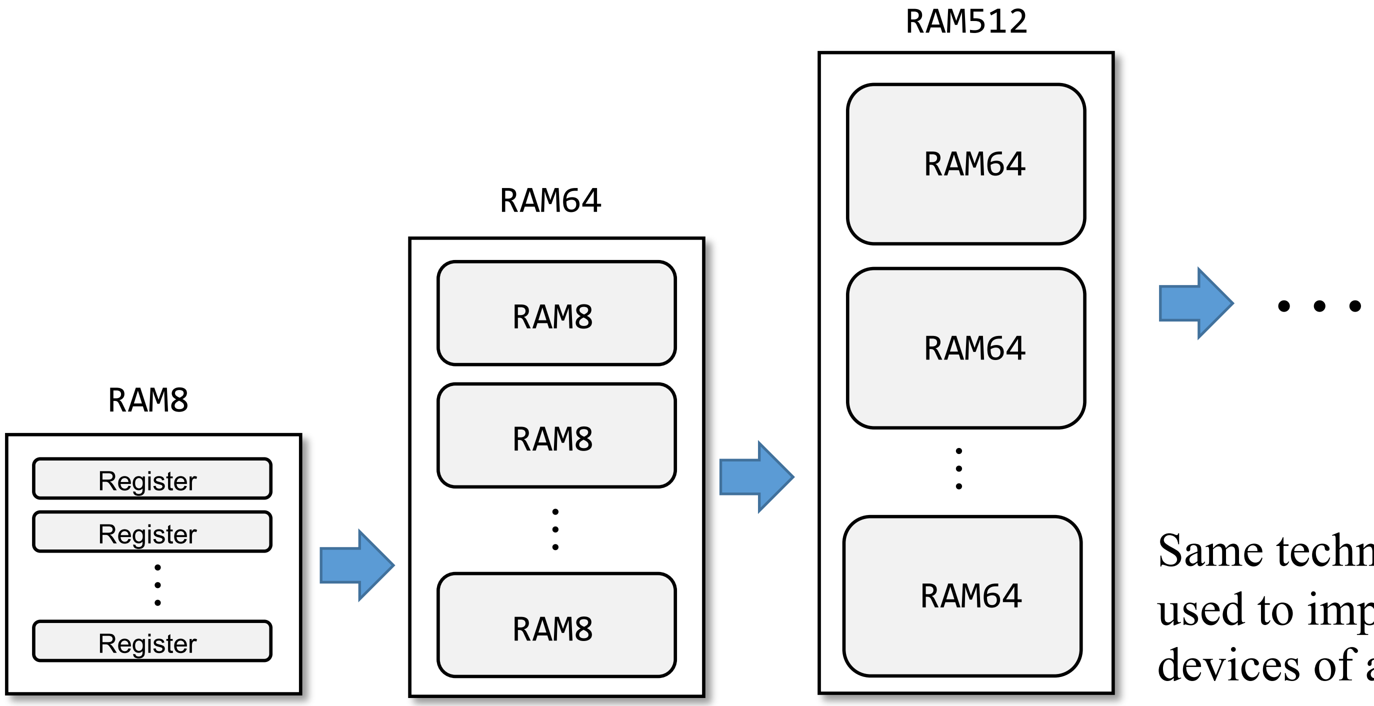


Partial diagram, showing some of the chip-parts, without connections

<u>Observations</u>

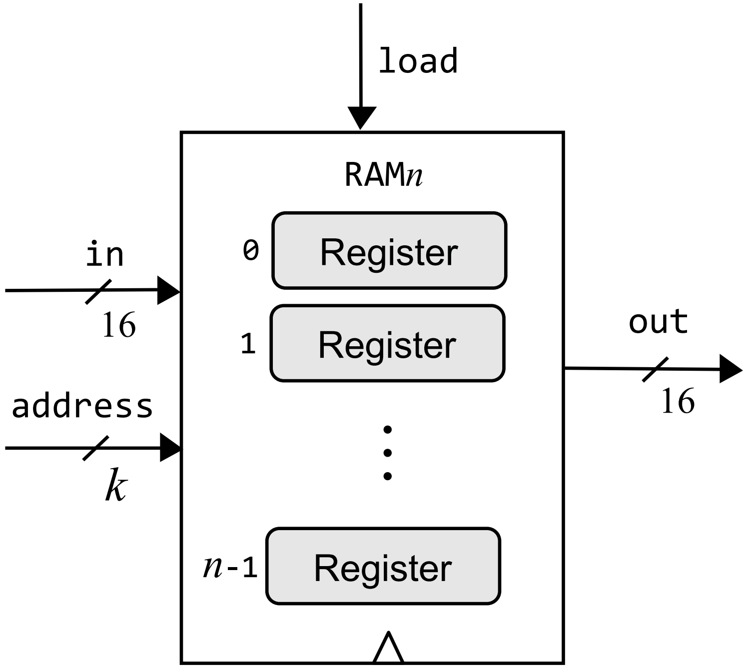
- The addressing/selection/reading logic is *combinational*
- The writing logic is (i) *sequential* (*clocked*)
  (ii) embedded in the Register logic.

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

Why these particular RAM chips?

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

Chapter 3: Memory

Abstraction

- Representing time
- Clock
- Registers
- RAM
- Counters

Implementation

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

Project 3

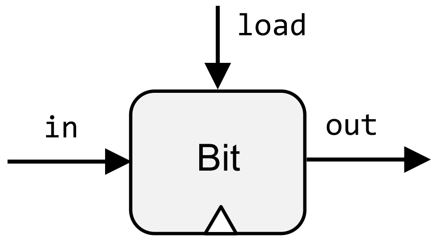
Given:

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

Build:

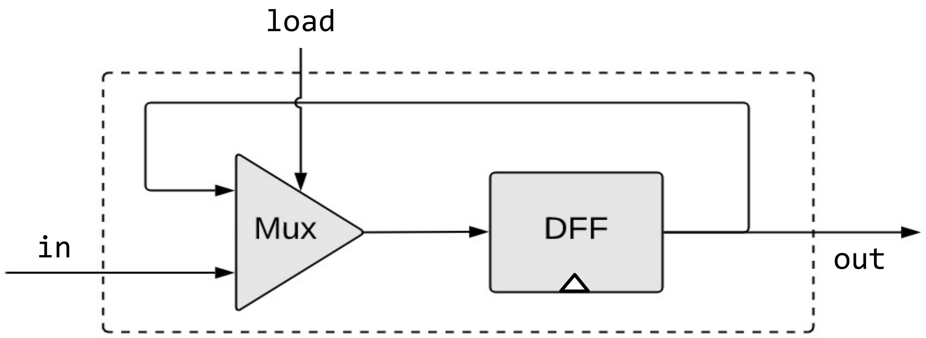
- `Bit`
- `Register`
- `PC`
- `RAM8`
- `RAM64`
- `RAM512`
- `RAM4K`
- `RAM16K`

1-bit Register



Bit.hdl

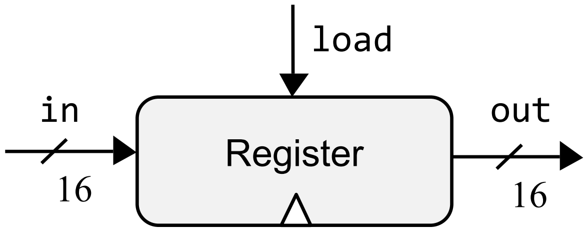
```
/** 1-bit register:
    if  load(t–1) then out(t) = in(t–1)
    else                out(t) = out(t–1))  */

CHIP Bit {
    IN in, load;
    OUT out;

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



Implementation tip:

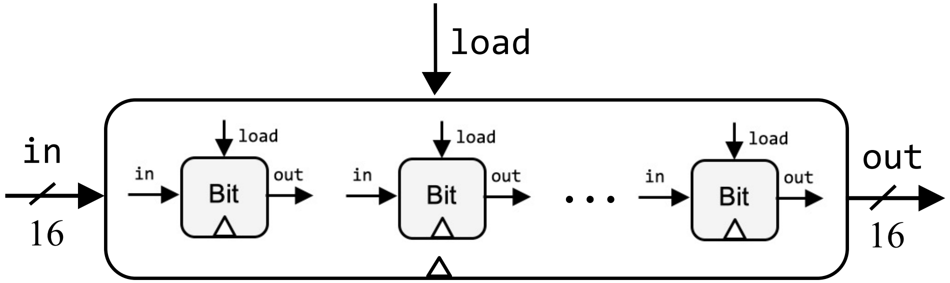
Follow the chip diagram

16-bit Register



Register.hdl

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

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

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

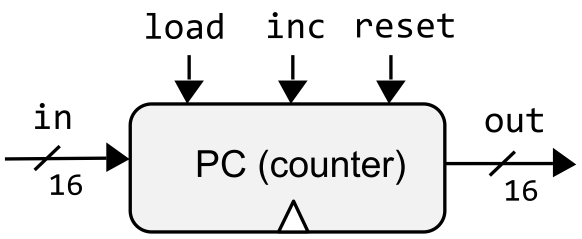


Partial diagram, showing some of
the chip-parts, without connections

<u>Implementation tip:</u>

Follow the chip diagram

16-bit Counter




```
/**
   A 16-bit counter with control bits.
   if         reset(t – 1)   out(t) = 0                  // resetting
   else  if  load(t – 1)    out(t) = in(t – 1)          // setting
   else  if  inc(t – 1)     out(t) = out(t – 1) + 1    // incrementing
   else                     out(t) = out(t – 1)         // 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 Incrementer, and Mux's

Project 3

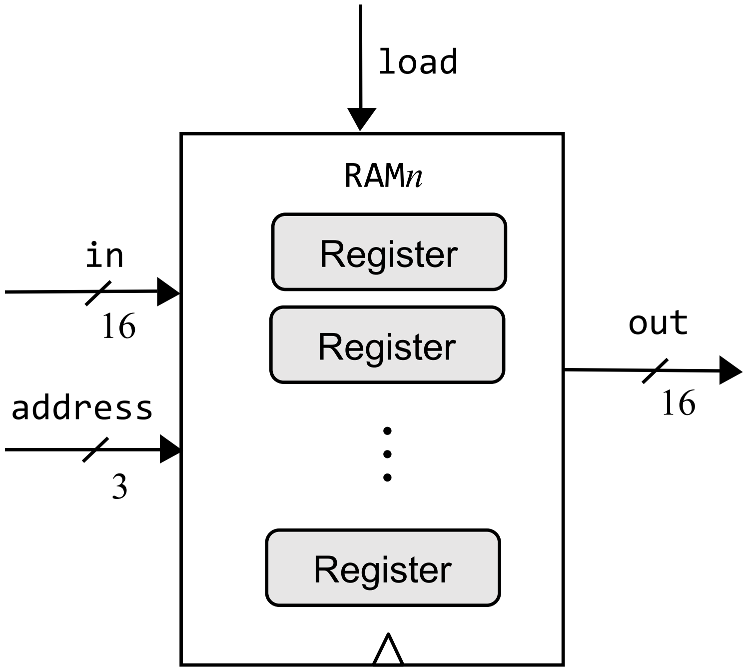
<u>Given</u>

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

<u>Build the following chips</u>

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

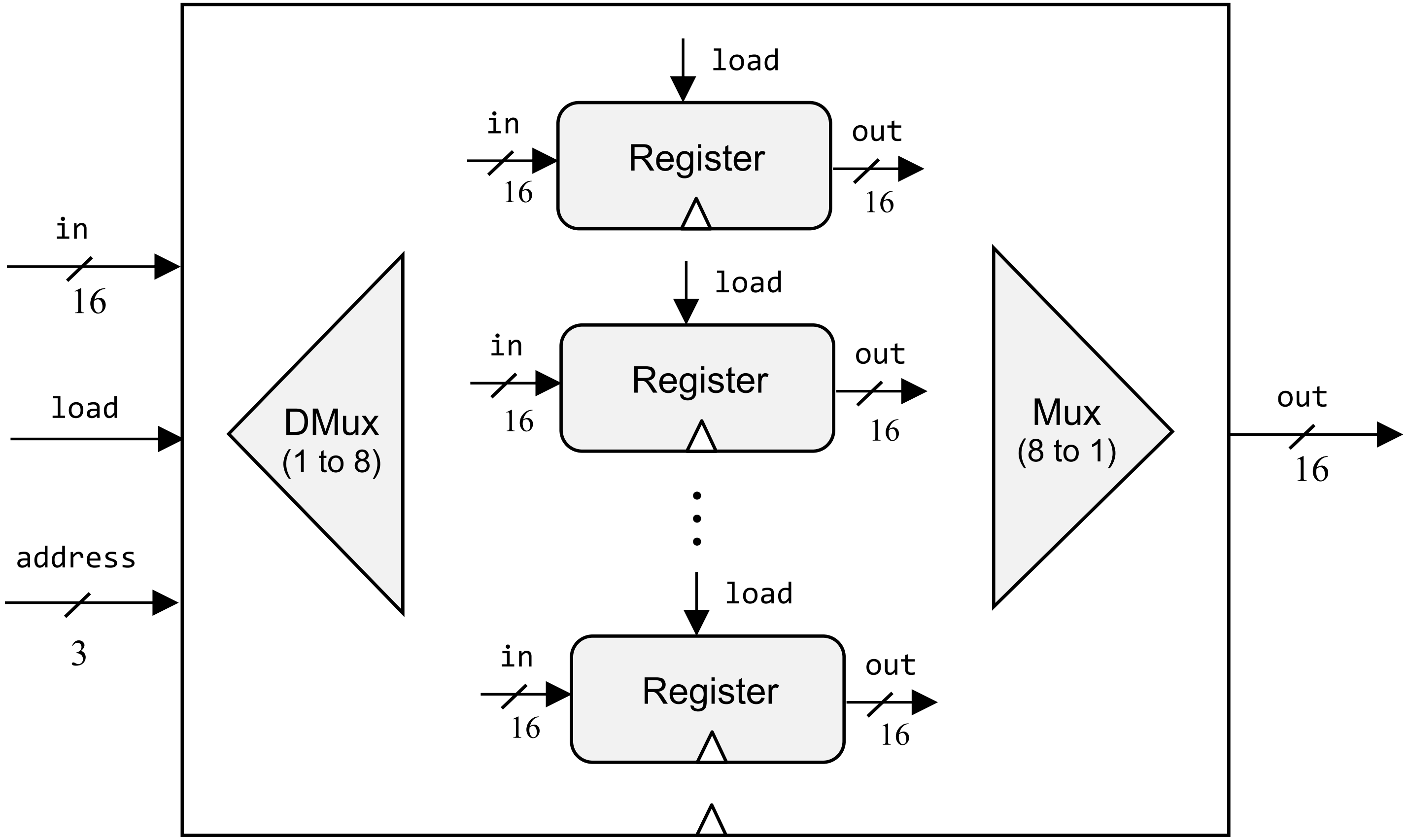
8-Register RAM: Abstraction



RAM8.hdl

```
/*
   Let M stand for the state of the register
   selected by address.
   if  load(t – 1) then  {M = in(t), out(t) = M}
   else                        out(t) = M
 */
CHIP RAM8 {
    IN in[16], load, address[3];
    OUT out[16];
    PARTS:
    // Put your code here:
}
```

8-Register RAM: Implementation



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

<u>Implementation tip:</u>

Follow the chip diagram

Project 3

Given

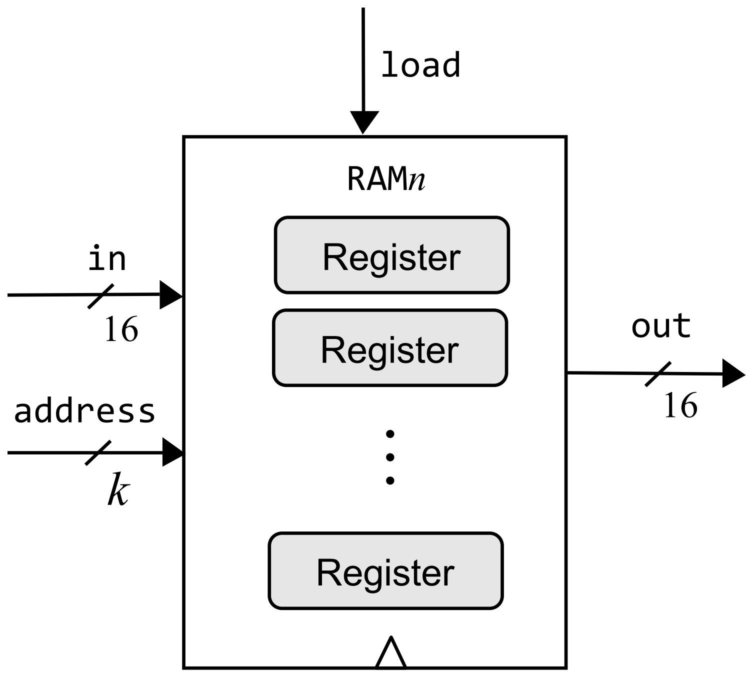
- All the chips built in projects 1 and 2
- Data Flip-Flop (built-in `DFF` gate)

Build the following chips

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

(RAM8 through RAM16K: A family of RAM chips)

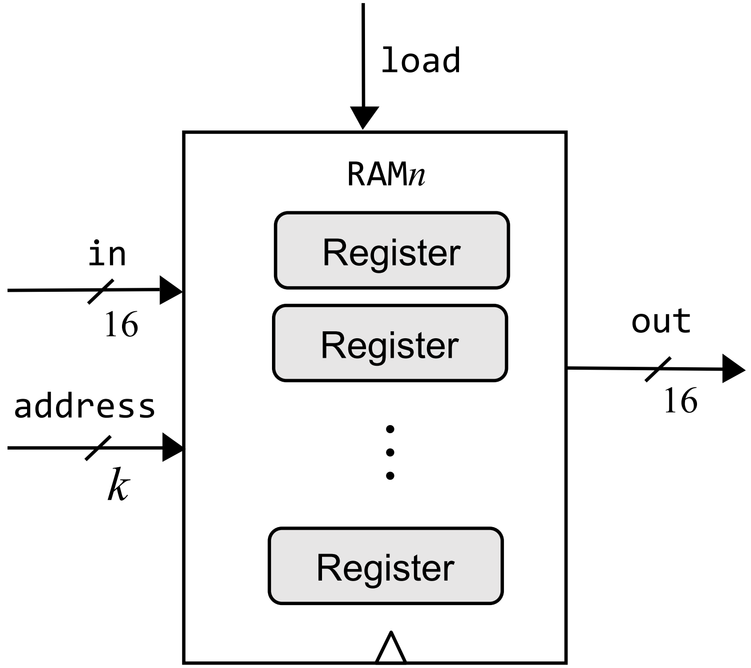
n-Register RAM



RAM*n*.hdl

```
/*
   Let M stand for the state of the register
   selected by address.
   if load(t – 1) then {M = in(t), out(t) = M}
   else                         out(t) = M
 */
CHIP RAMn {
    IN in[16], load, address[k];
    OUT out[16];
    PARTS:
    // Put your code here:
}
```

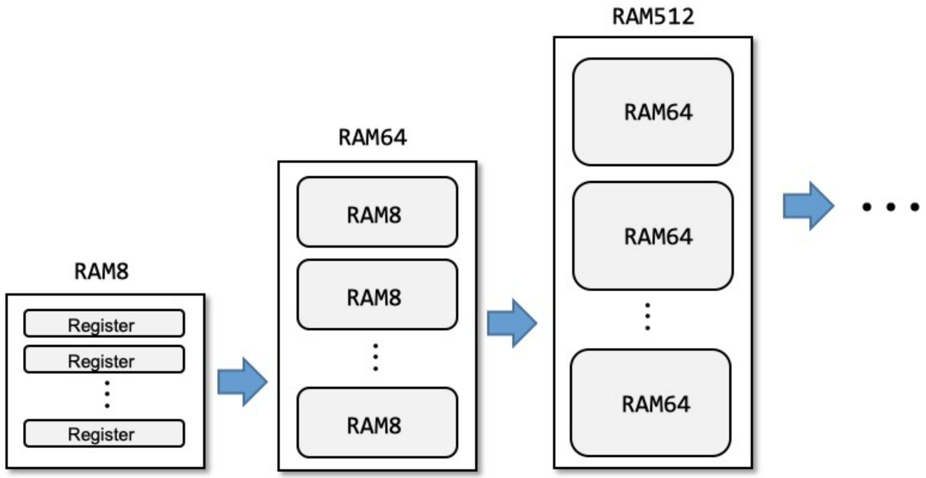
n-Register RAM



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 addressing scheme.



Chapter 3: Memory

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

Project 3

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

Resources

Project 3 folder (`.hdl`, `.tst`, `.cmp` files): `nand2tetris/projects/03`

Tools

- Text editor (for completing the given `.hdl` stub-files)
- Hardware simulator: `nand2tetris/tools`

Guides

- Hardware Simulator Tutorial
- HDL Guide
- Hack Chip Set API

Best practice advice

- 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!

Chapter 3: Memory

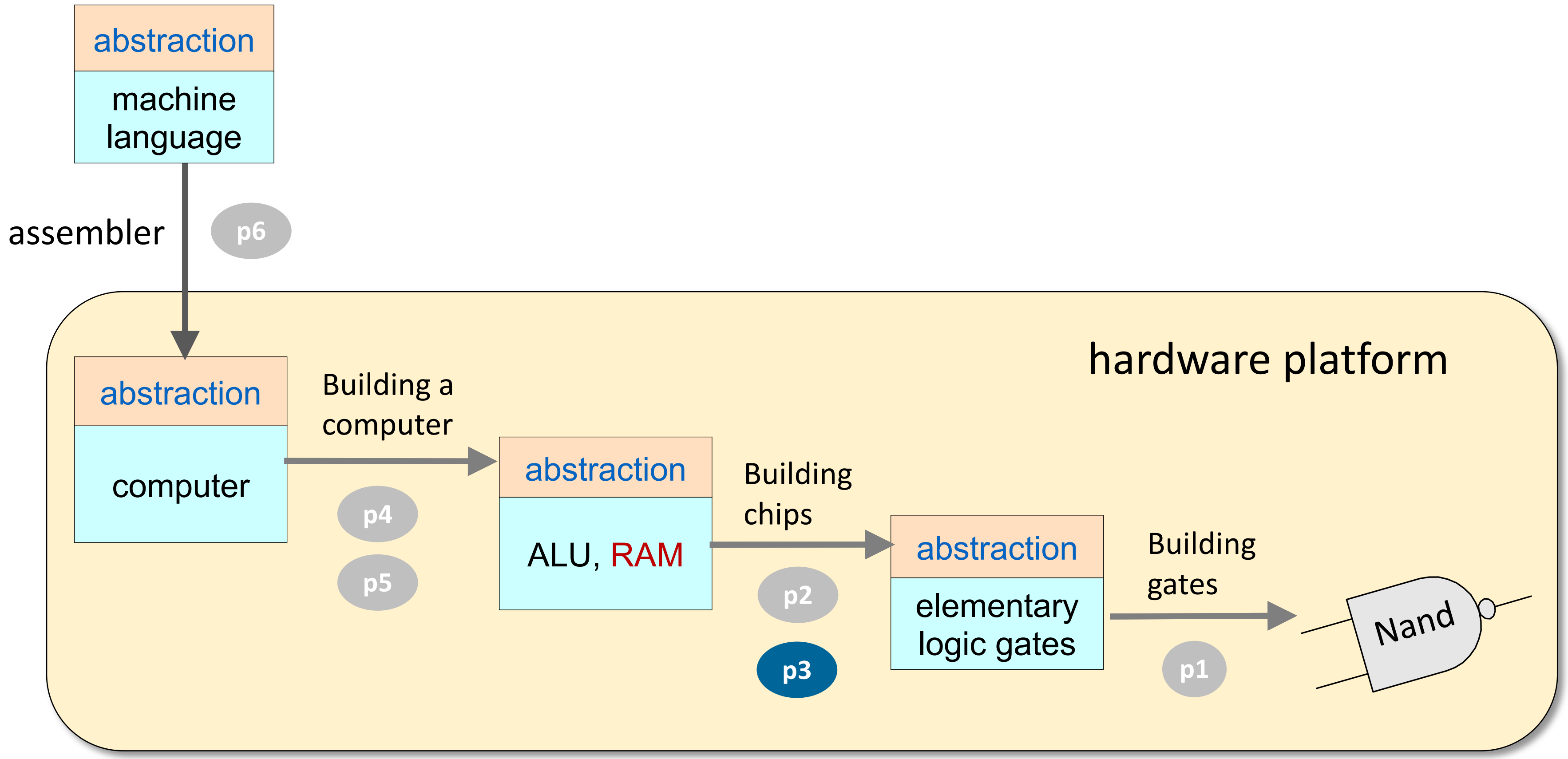
<u>Abstraction</u>

- Representing time
- Clock
- Registers
- RAM
- Counters

<u>Implementation</u>

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

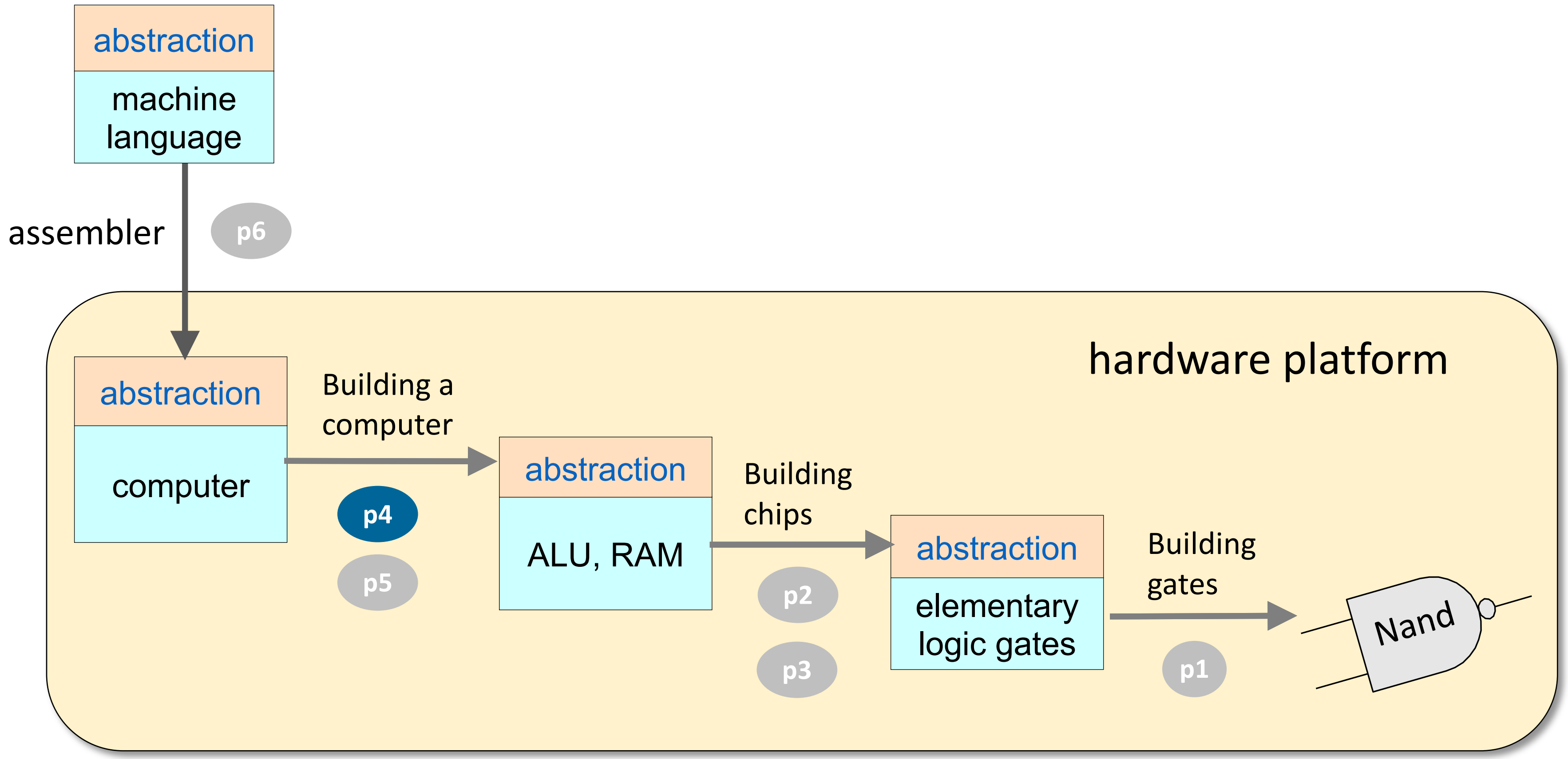
What's next?



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

Build the computer's RAM

What's next?



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

- Get acquainted with the computer architecture
- Write machine language programs