Sequential Logic

Introduction to Computer

Yung-Yu Chuang

with slides by Sedgewick & Wayne (introcs.cs.princeton.edu), Nisan & Schocken (www.nand2tetris.org) and Harris & Harris (DDCA)

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*

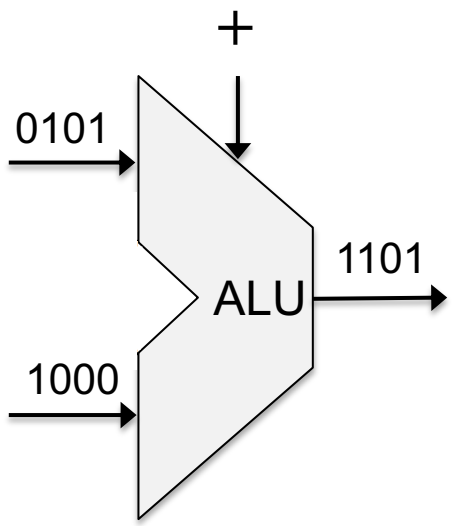
The chip's inputs were just "sitting there" – fixed and unchanging

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*

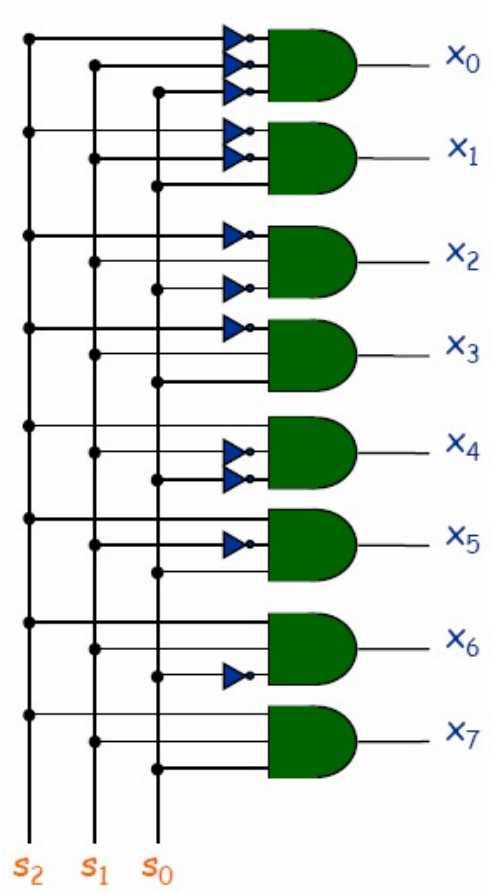
All the chips that we discussed and developed so far were combinational



Combinational vs. Sequential Circuits

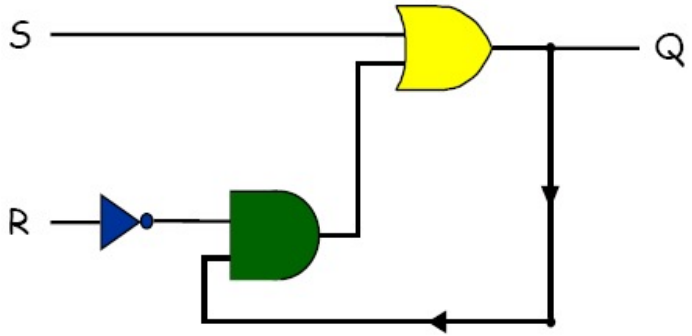
Combinational circuits.

- Output determined solely by inputs.
- Can draw with no loops.
- Ex: majority, adder, ALU.



Sequential circuits.

- Output determined by inputs and previous outputs.
- Ex: memory, program counter, CPU.

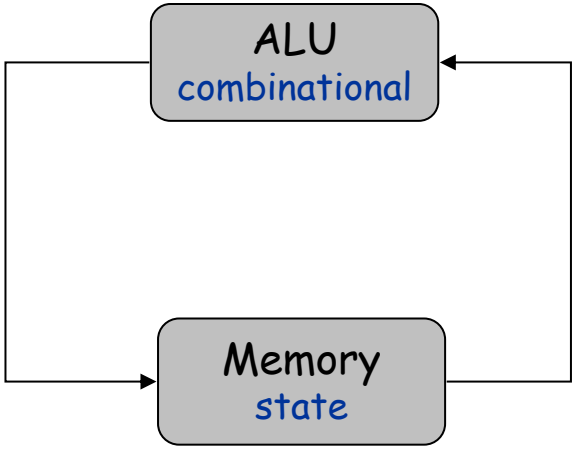


Combinational vs. Sequential Circuits

Combinational circuits.

- Basic abstraction = switch.
- In principle, can build TOY computer with a combinational circuit.
  - $255 \times 16 = 4{,}080$ inputs $\Rightarrow$ 2^{4080} rows in truth table!
  - no simple pattern
  - each circuit element used at most once

Sequential circuits. Reuse circuit elements by storing bits in "memory."



Time

Software needs:

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

Example (variables):

```
x = 17
```

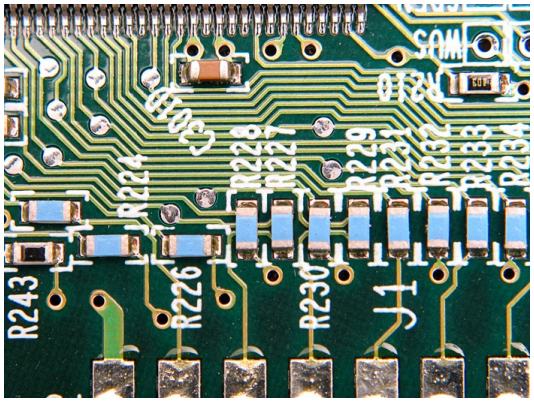
- The hardware must be able to do things, one at a time (sequentially):

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.



Representing time

physical time:

Arrow of time:

Continuous

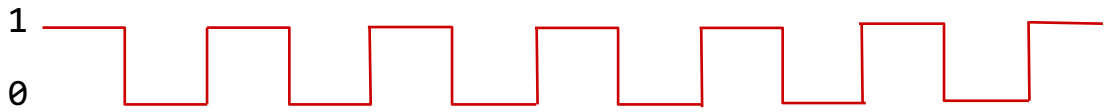
Representing time

physical time:

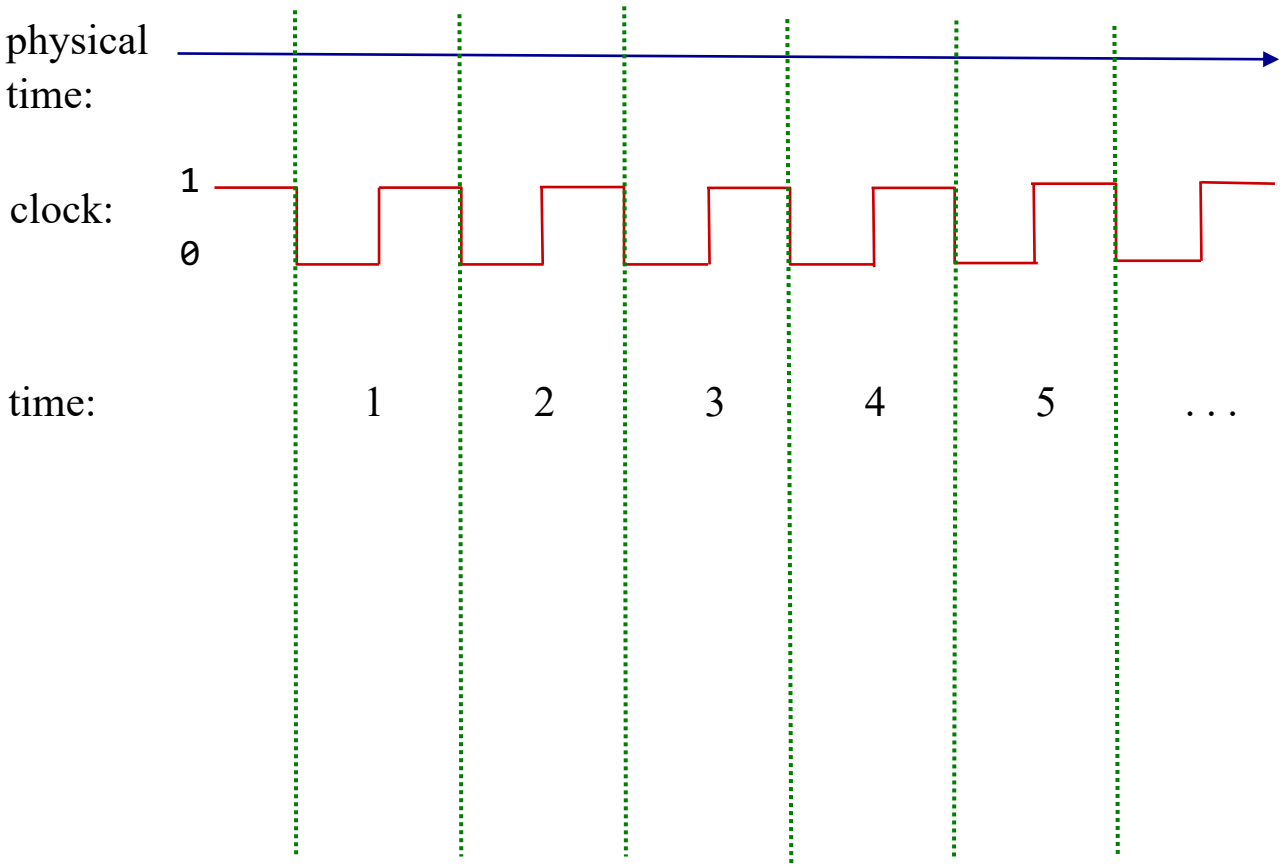
Arrow of time:

Continuous

clock:



Chip behavior over time (example: Not gate)



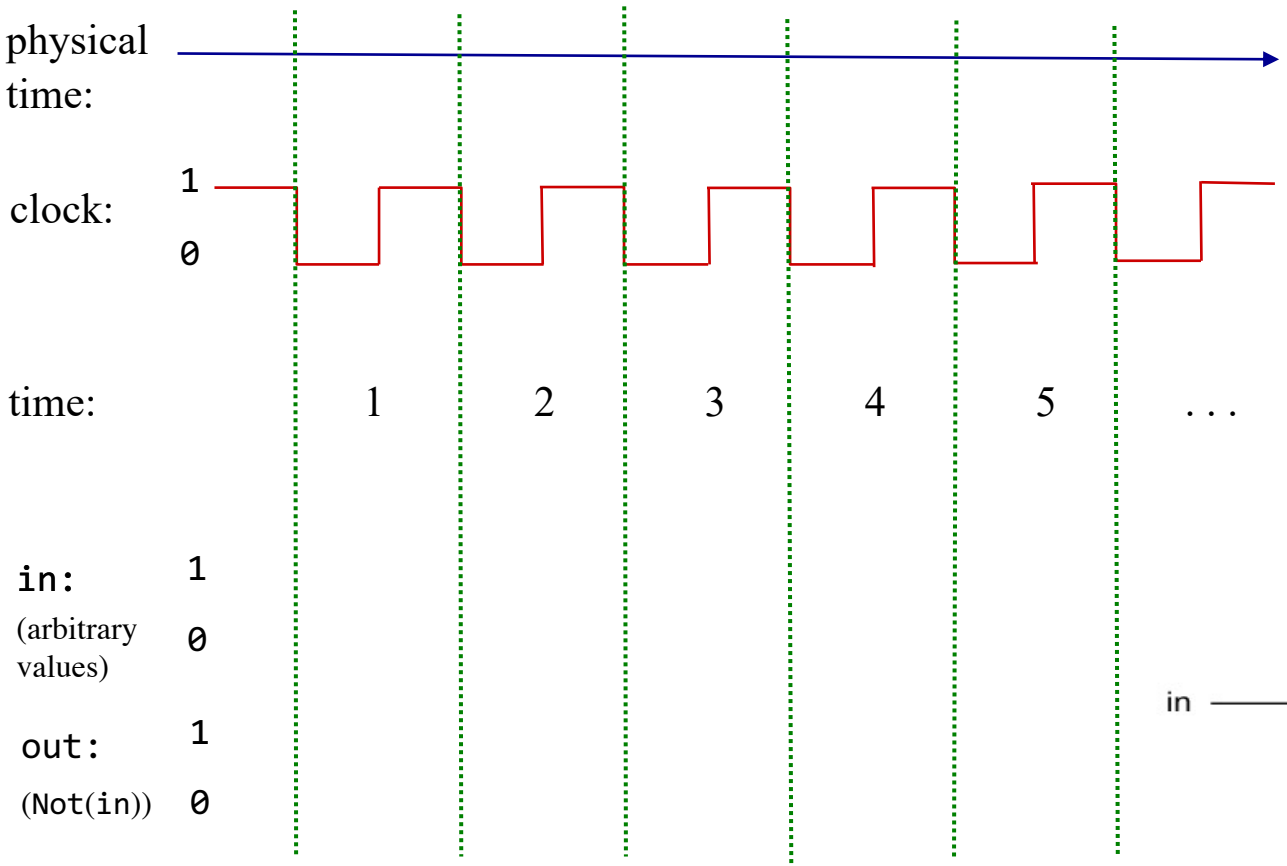
Arrow of time:

Continuous

Discrete time:

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

Chip behavior over time (example: Not gate)

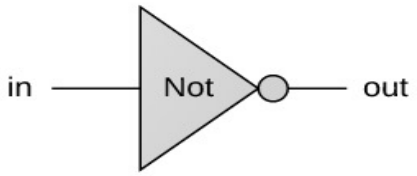


Arrow of time:

Continuous

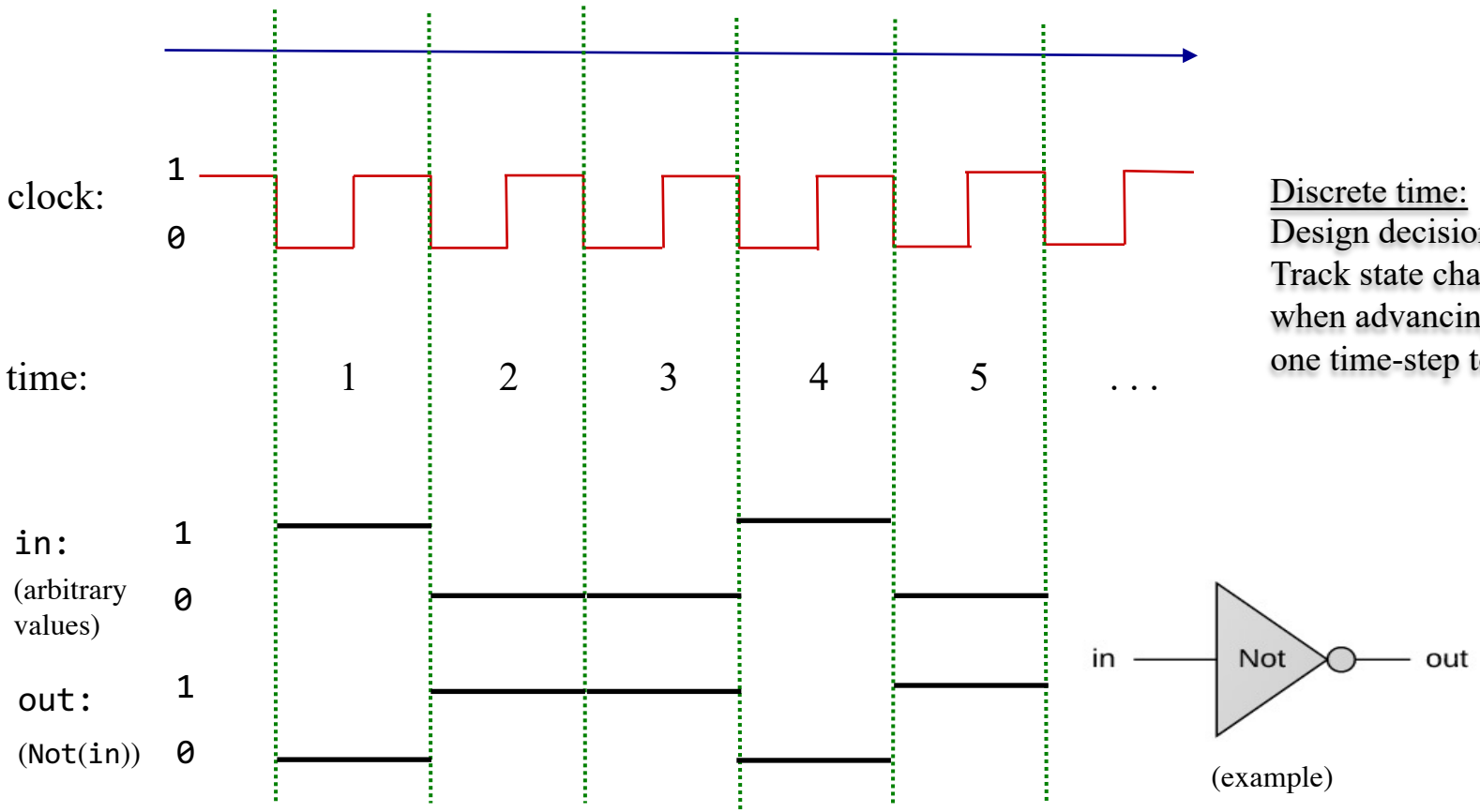
Discrete time:

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



(example)

Chip behavior over time (example: Not gate)

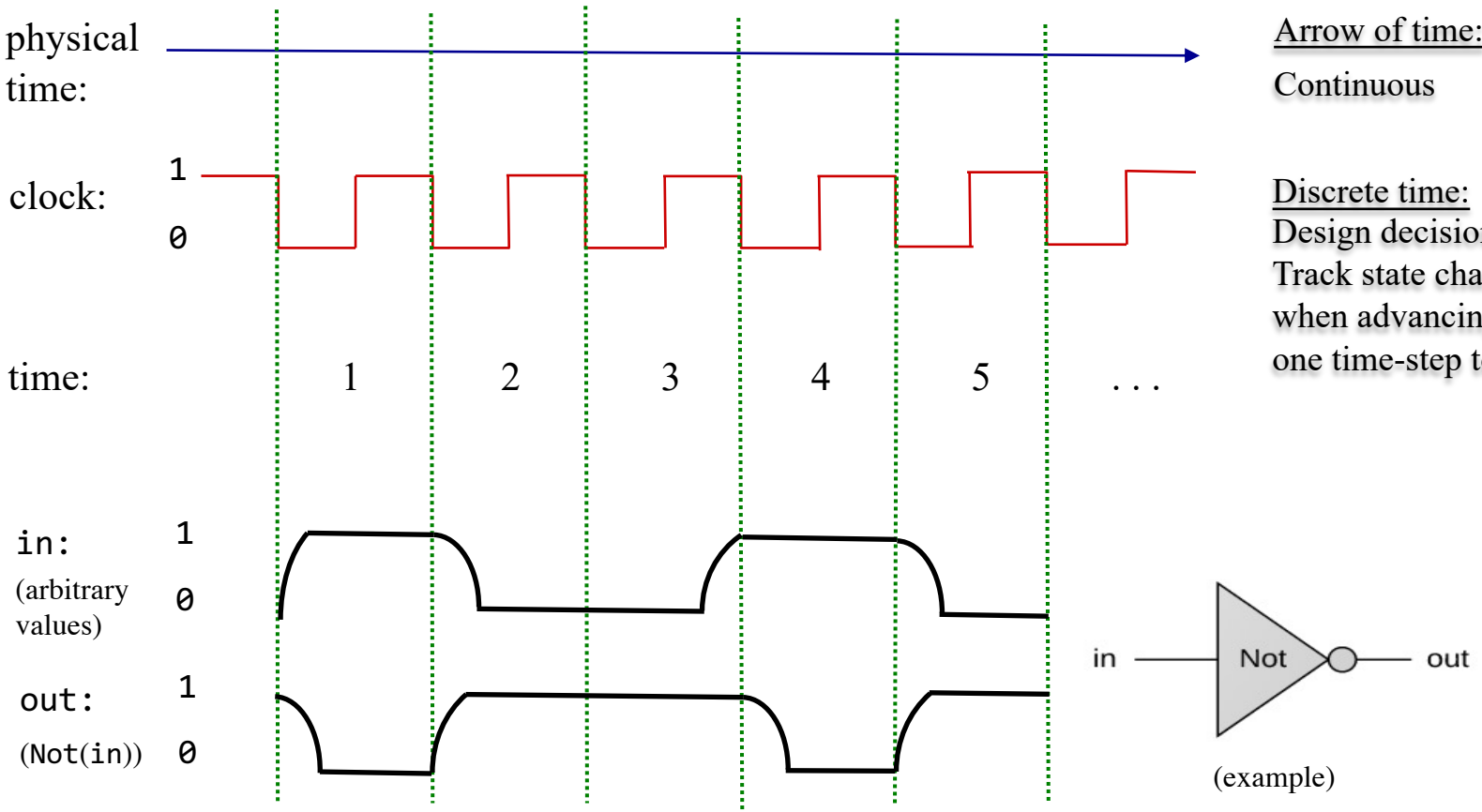


Discrete time:
Design decision:
Track state changes only when advancing from one time-step to another

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

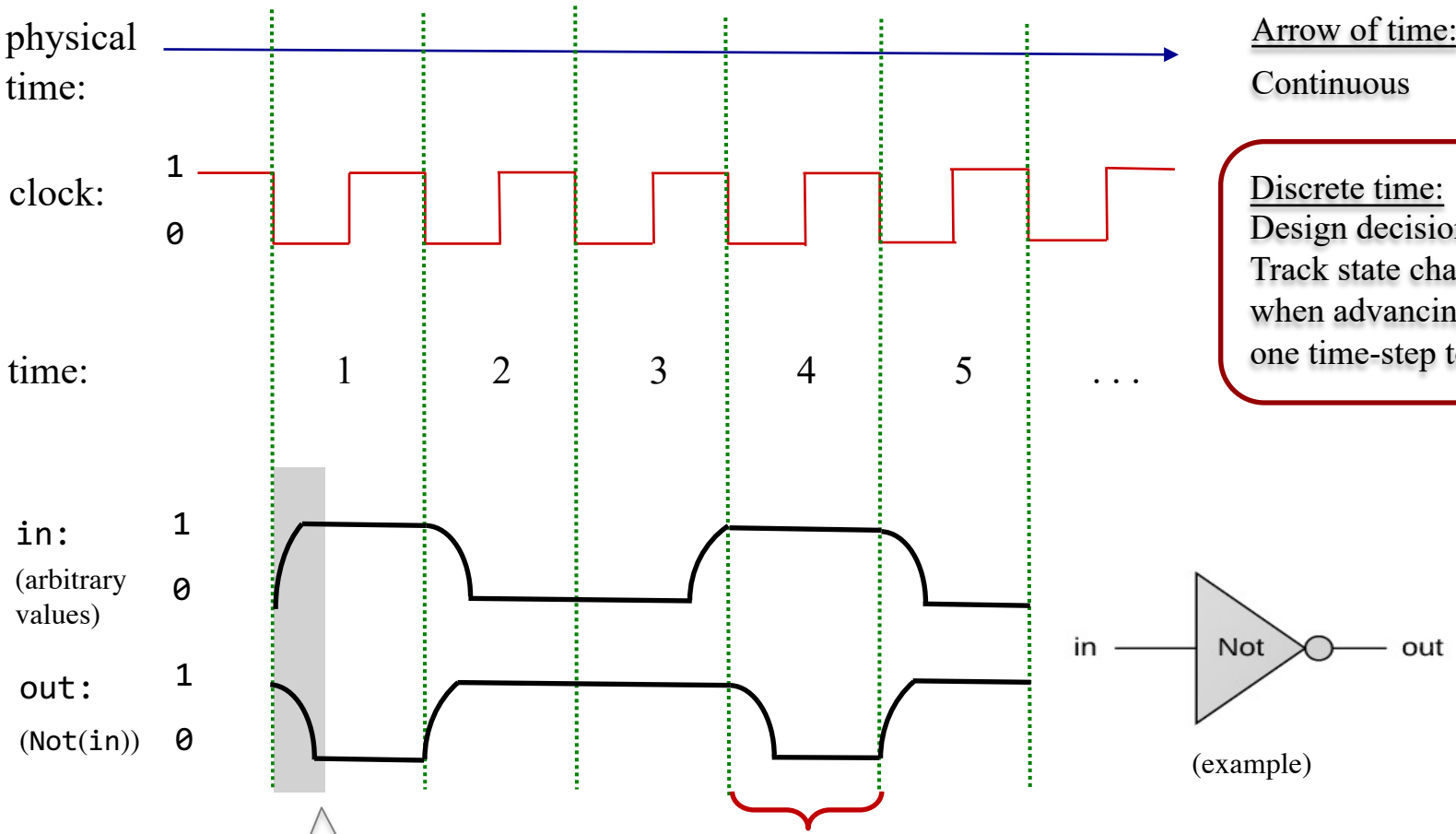
Discrete time:

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

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

Influenced by physical time delays

Chip behavior over time (example: Not gate)



Arrow of time:

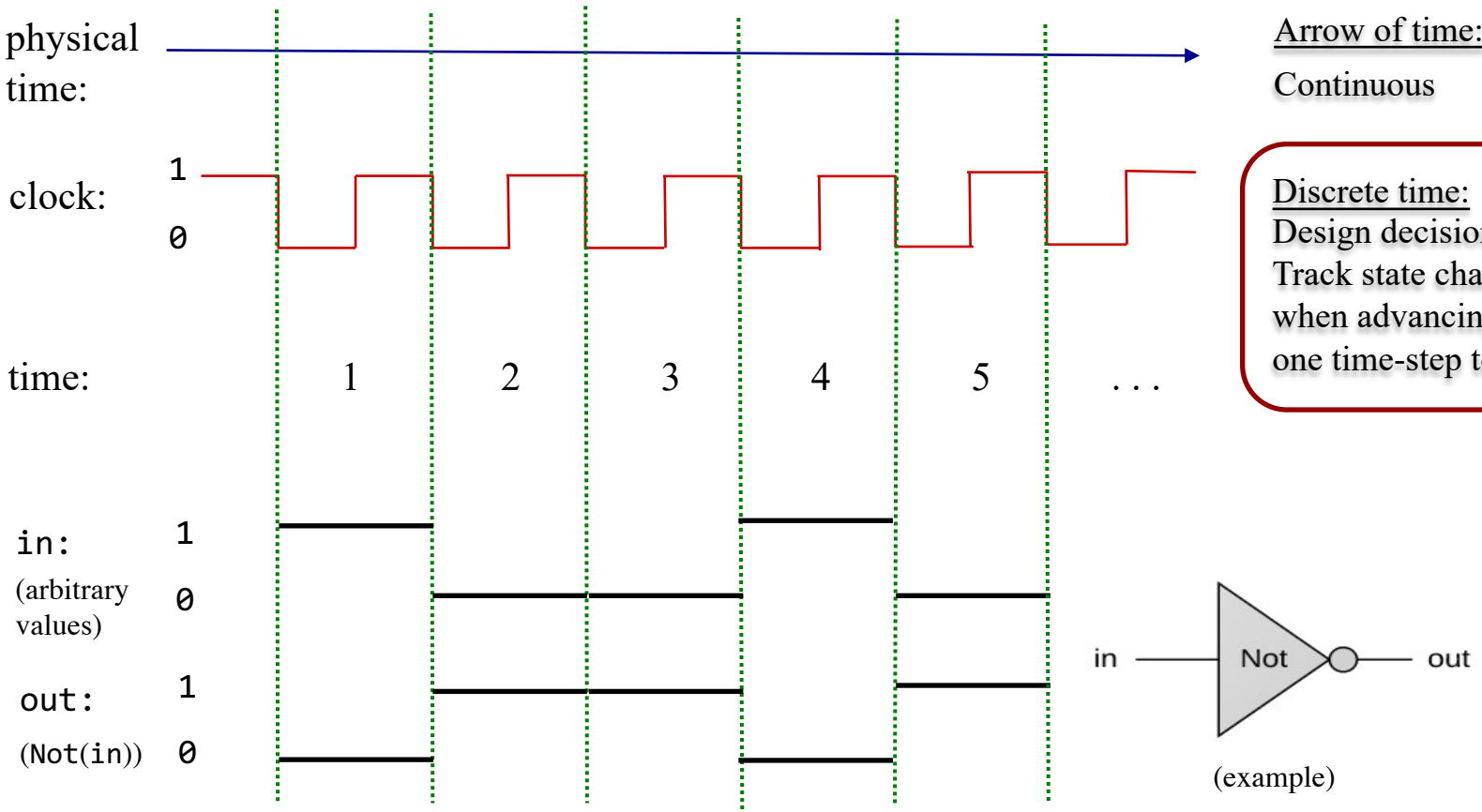
Continuous

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

Time delays
- Propagation delays
- Computation delays

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

Chip behavior over time (example: Not gate)



Resulting effect:

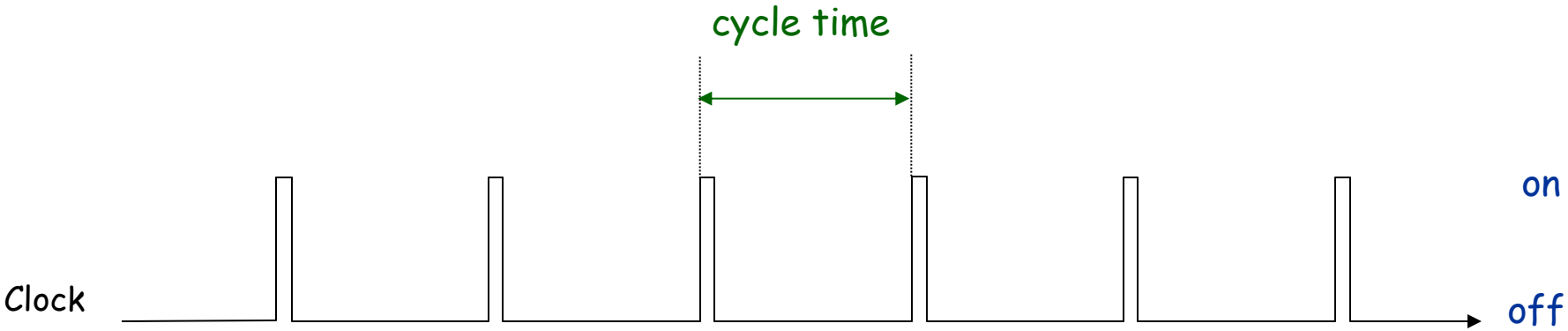
Combinational chips react "immediately" to their inputs

Facilitated by the decision to track changes only at cycle ends

Clock

Clock.

- Fundamental abstraction: regular on-off pulse.
  - on: fetch phase
  - off: execute phase
- External analog device.
- Synchronizes operations of different circuit elements.
- Requirement: clock cycle longer than max switching time.



How much does it Hert?

Frequency is inverse of cycle time.

- Frequency of 1 Hz (Hertz) means that there is 1 cycle per second.
  - 1 kilohertz (kHz) means 1000 cycles/sec.
  - 1 megahertz (MHz) means 1 million cycles/sec.
  - 1 gigahertz (GHz) means 1 billion cycles/sec.
  - 1 terahertz (THz) means 1 trillion cycles/sec.

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)

Heinrich Rudolf Hertz (1857-1894)

Flip-Flop

Flip-flop

- A small and useful sequential circuit
- Abstraction that remembers one bit
- Basis of important computer components for
  - register
  - memory
  - counter
- There are several flavors

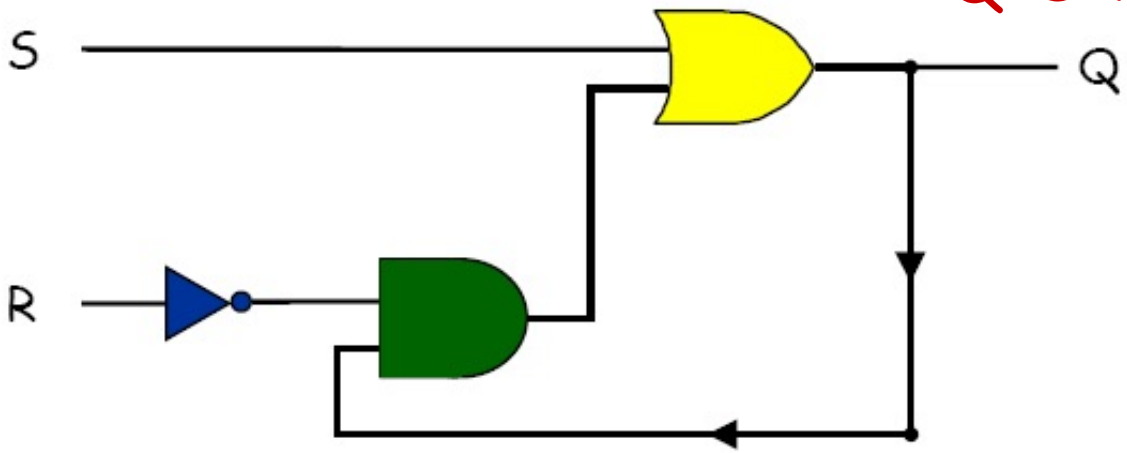
S-R flip flop

SR Flip-Flop.

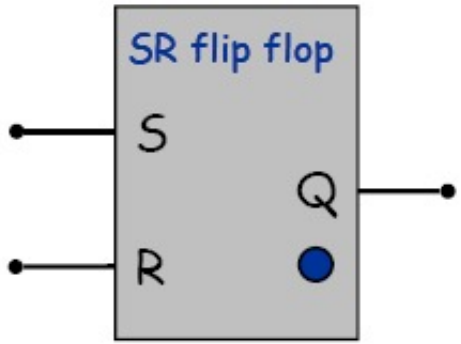
- S = 1, R = 0 (set) ⇒ Flips "bit" on.
- S = 0, R = 1 (reset) ⇒ Flops "bit" off.
- S = R = 0 ⇒ Status quo.
- S = R = 1 ⇒ Not allowed.

R	S	Q
0	0	
0	1	
1	0	
1	1	

$Q = S + \bar{R}Q$



Implementation



Interface

S-R flip flop

SR Flip-Flop.

- S = 1, R = 0 (set) ⇒ Flips "bit" on.
- S = 0, R = 1 (reset) ⇒ Flops "bit" off.
- S = R = 0 ⇒ Status quo.
- S = R = 1 ⇒ Not allowed.

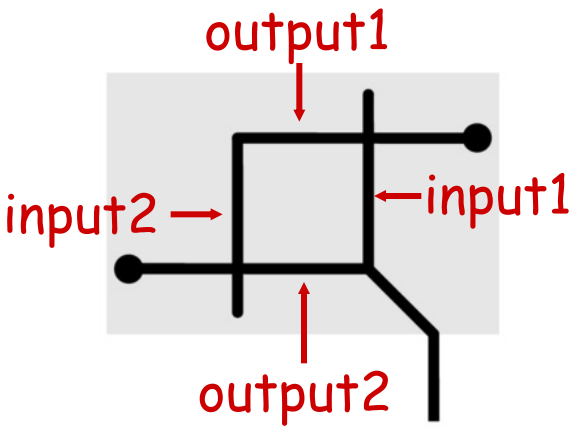
R	S	Q
0	0	
0	1	
1	0	
1	1	

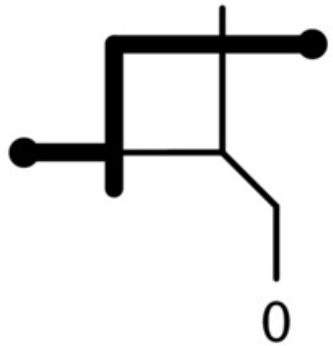
$$Q=S+\bar{R}Q$$

Relay-based flip-flop

Ex. Simplest feedback loop.

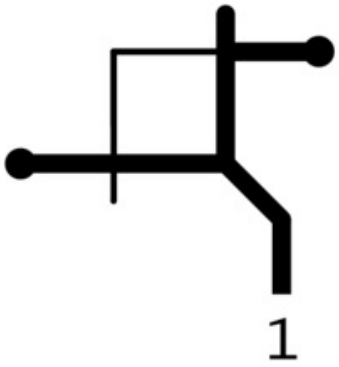
- Two relays A and B, both connected to power, each blocked by the other.
- State determined by whichever switches first. The state is latched.
- Stable.





0 state

0

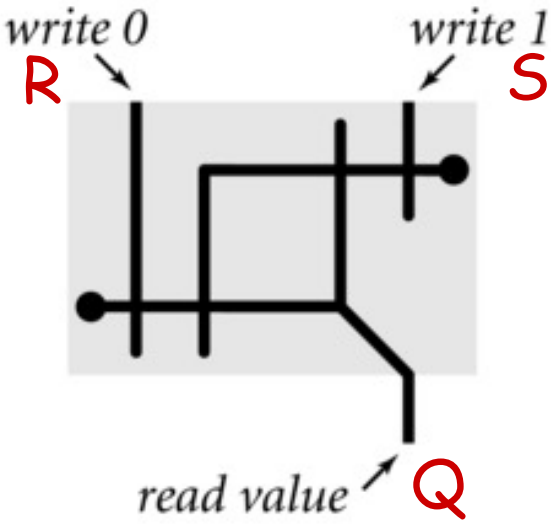


1 state

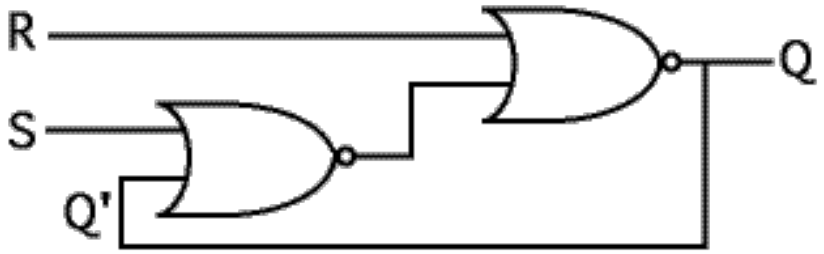
1

SR Flip Flop

SR flip flop. Two cross-coupled NOR gates.



$Q=\overline{R}(S+Q)$

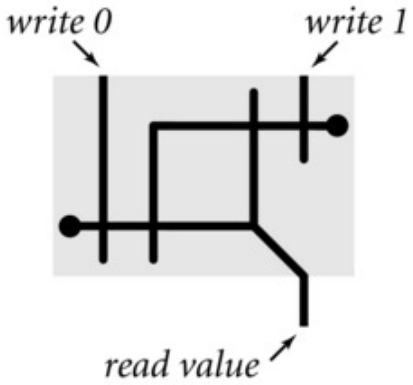


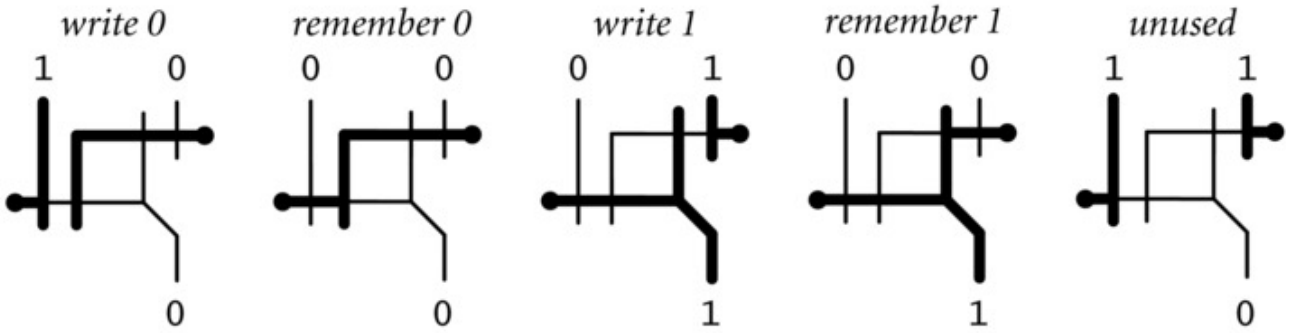
R	S	Q
0	0	
0	1	
1	0	
1	1	

Flip-Flop

Flip-flop.

- A way to control the feedback loop.
- Abstraction that "remembers" one bit.
- Basic building block for memory and registers.





Caveat. Need to deal with switching delay.

Truth Table and Timing Diagram

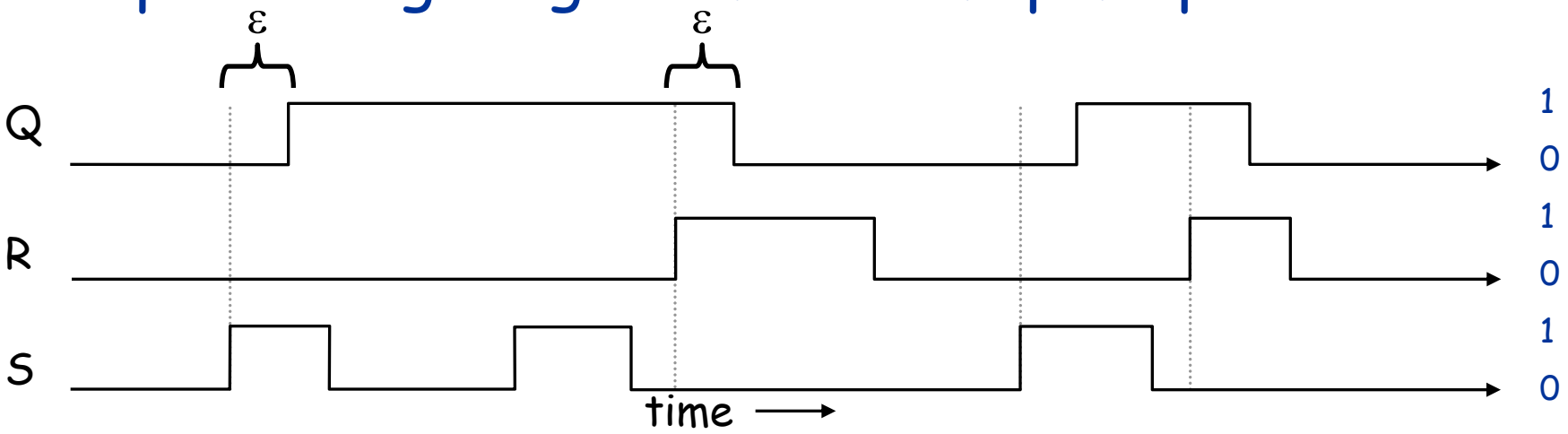
Truth table.

- Values vary over time.
- S(t), R(t), Q(t) denote value at time t.

SR Flip Flop Truth Table

S(t)	R(t)	Q(t)	Q(t+ε)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	
1	1	1	

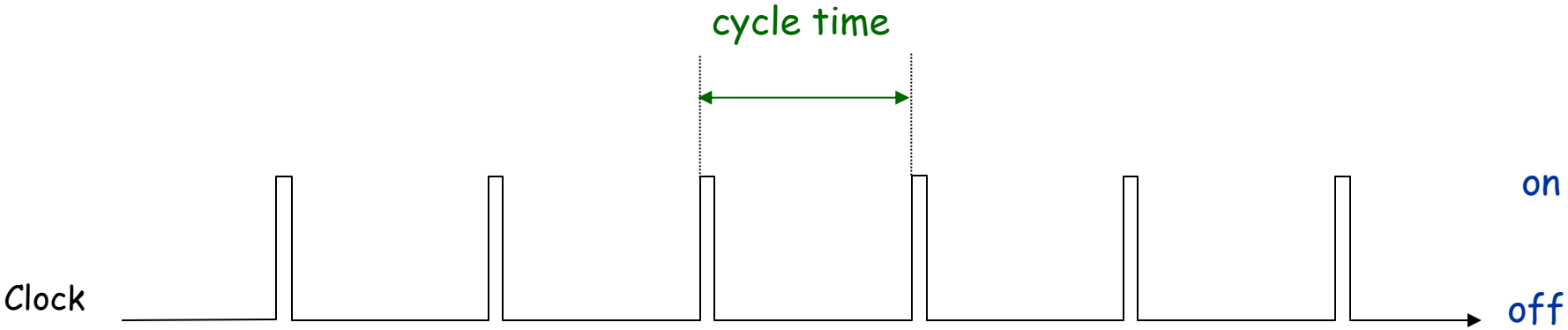
Sample timing diagram for SR flip-flop.



Clock

Clock.

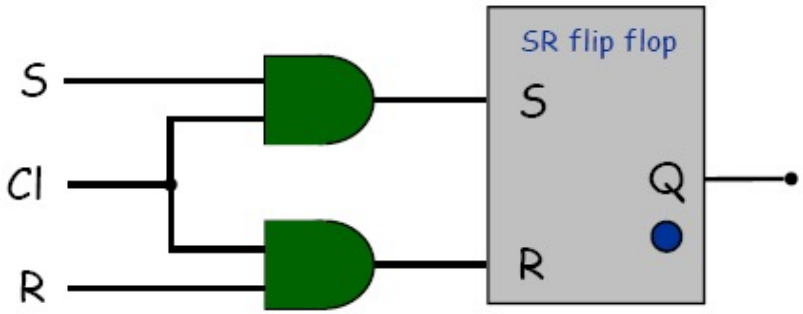
- Fundamental abstraction: regular on-off pulse.
  - on: fetch phase
  - off: execute phase
- External analog device.
- Synchronizes operations of different circuit elements.
- Requirement: clock cycle longer than max switching time.



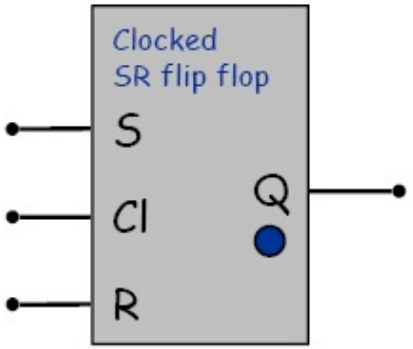
Clocked S-R flip-flop

Clocked SR Flip-Flop.

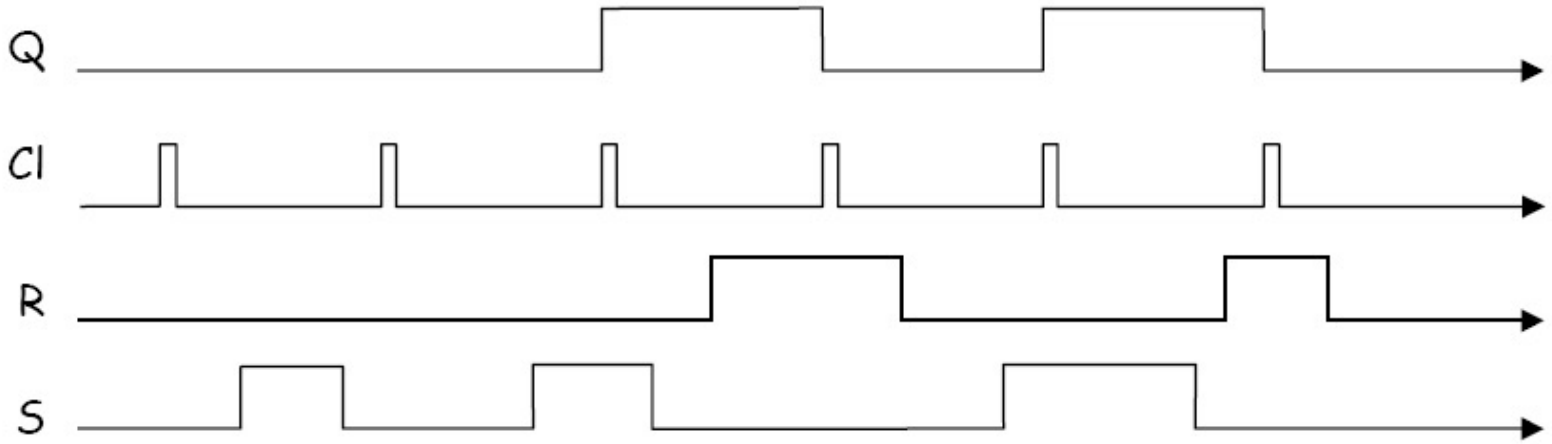
- Same as SR flip-flop except S and R only active when clock is 1.



Implementation



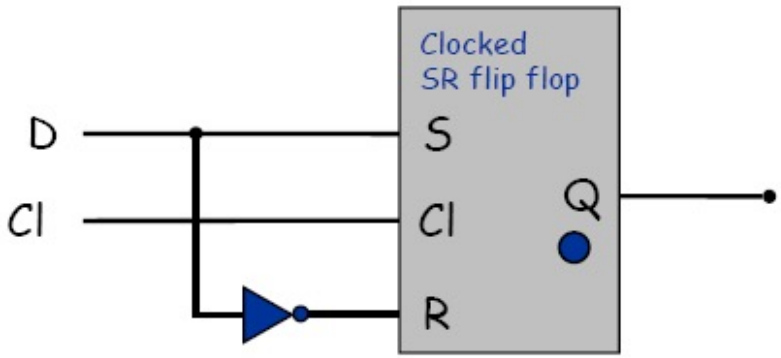
Interface



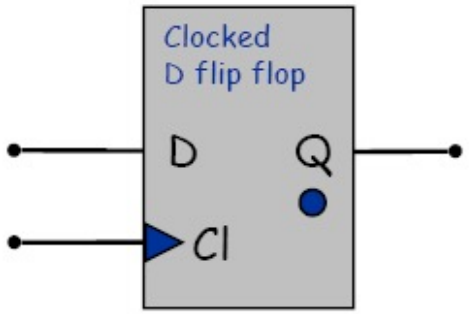
Clocked D flip-flop

Clocked D Flip-Flop.

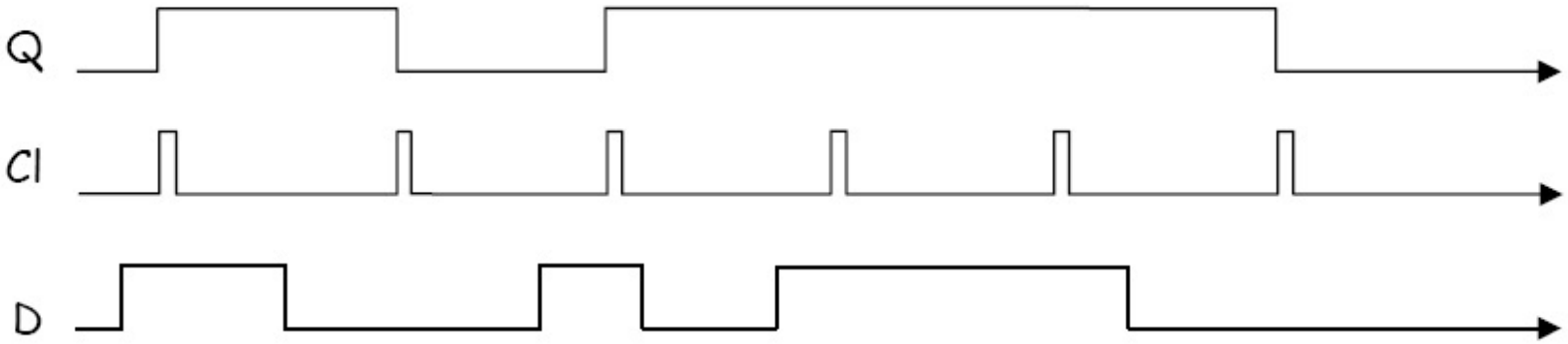
- Output follows D input while clock is 1.
- Output is remembered while clock is 0.



Implementation



Interface

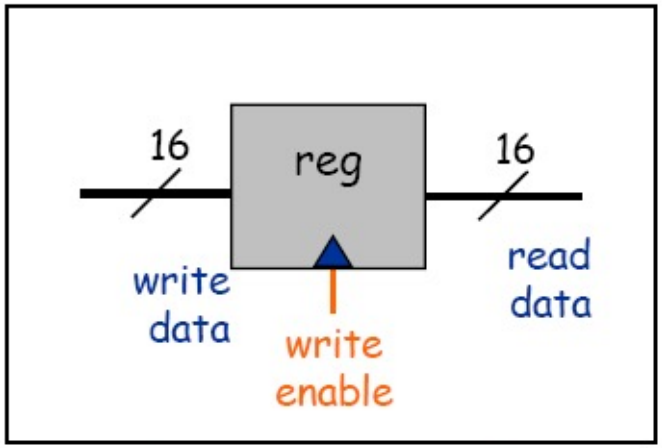


Stand-Alone Register

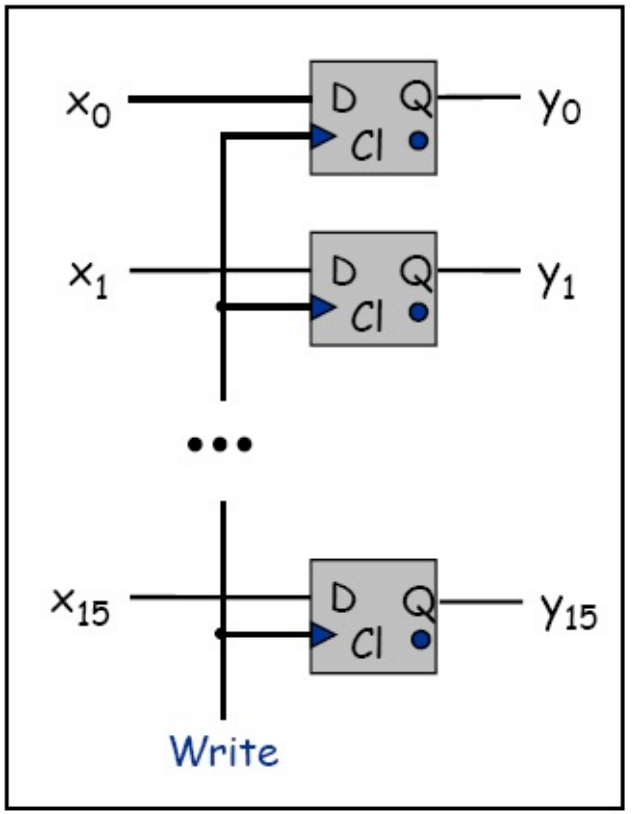
k-bit register.

- Stores k bits.
- Register contents always available on output.
- If write enable is asserted, k input bits get copied into register.

Ex: Program Counter, 16 TOY registers, 256 TOY memory locations.



16-bit Register Interface



16-bit Register Implementation

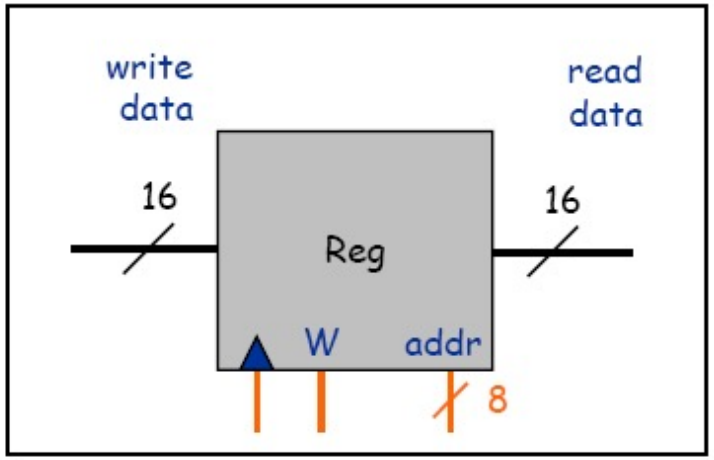
Register file interface

n x k register file.

- Bank of n registers; each stores k bits.
- Read and write information to *one* of n registers.
  - $\log_2 n$ address inputs specifies which one
- Addressed bits always appear on output.
- If write enable and clock are asserted, k input bits are copied into addressed register.

Examples.

- TOY registers: n = 16, k = 16.
- TOY main memory: n = 256, k = 16.
- Real computer: n = 256 million, k = 32.
  - 1 GB memory
  - 1 byte = 8 bits

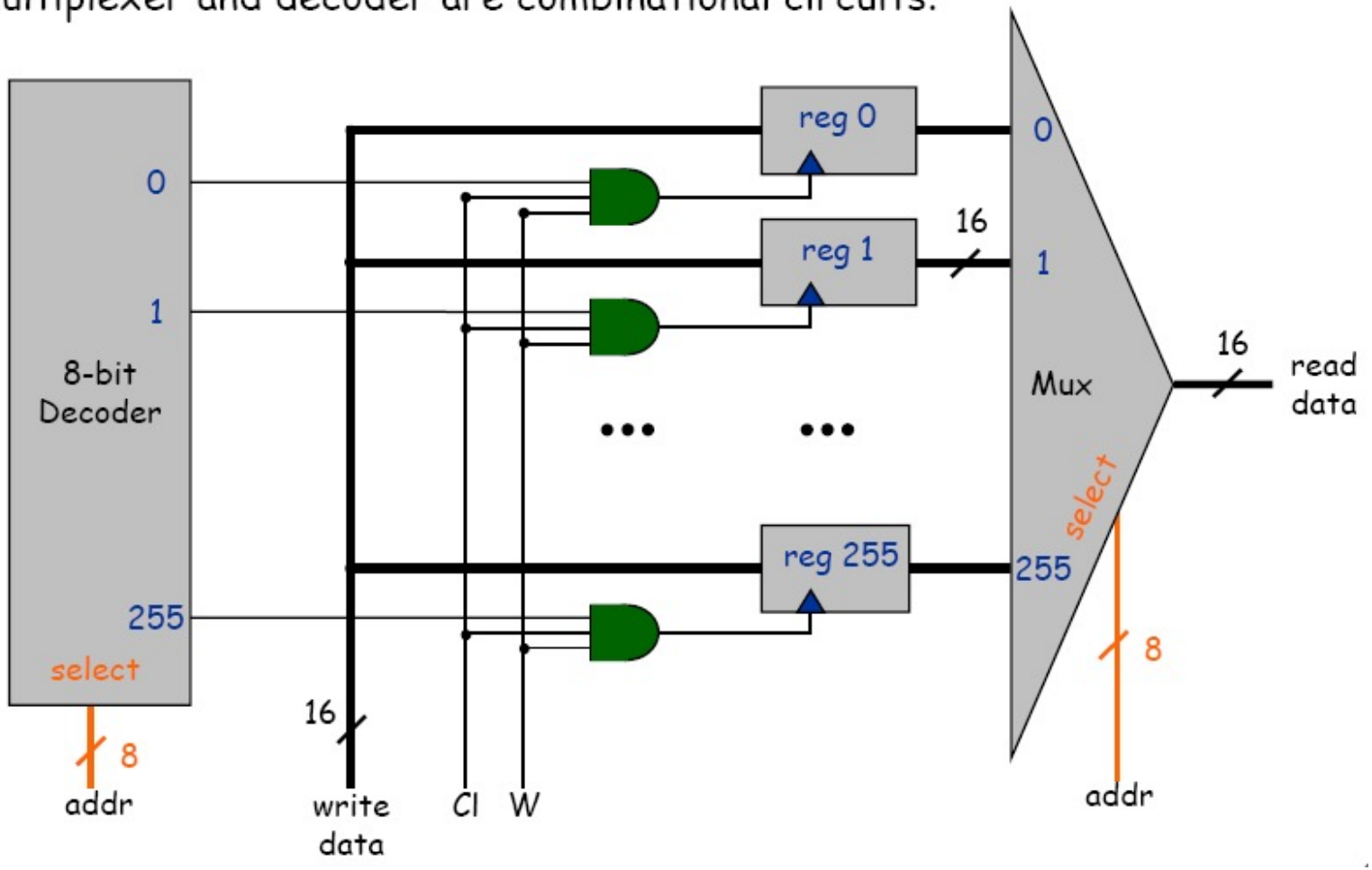


256 x 16 Register File Interface

Register file implementation

Implementation example: TOY main memory.

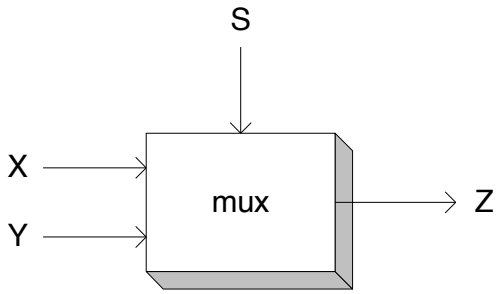
- Use 256 16-bit registers.
- Multiplexer and decoder are combinational circuits.



Multiplexer

When s=0, return x; otherwise, return y.

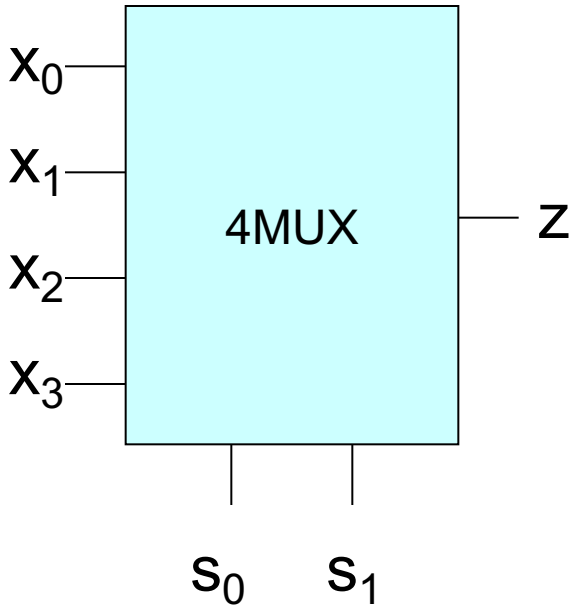
Example: $(Y \wedge S) \vee (X \wedge \neg S)$

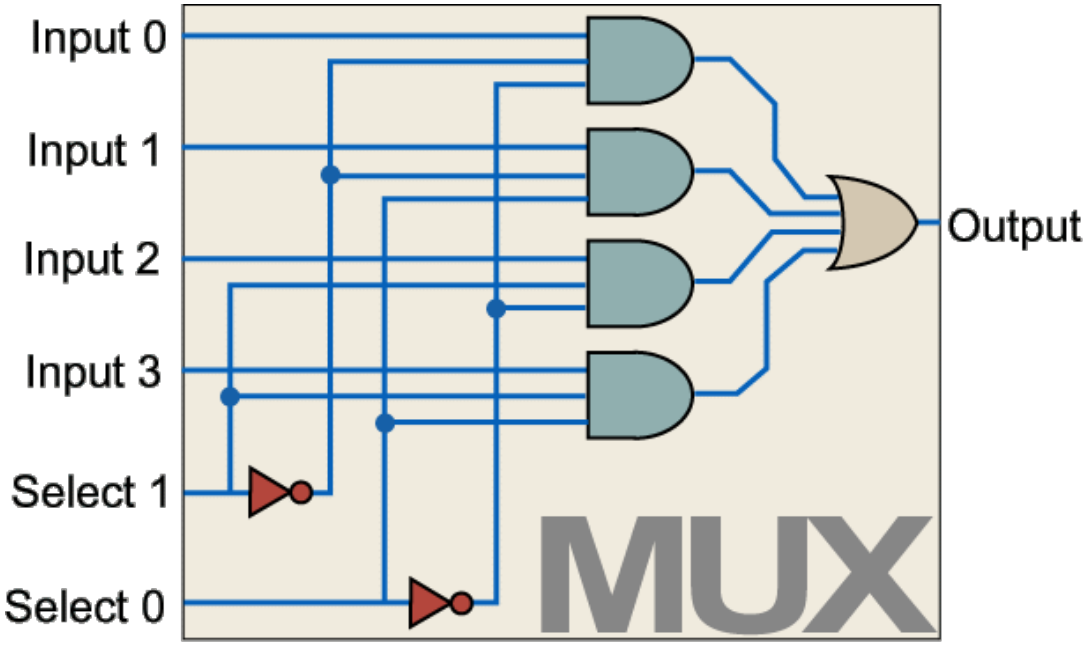


Two-input multiplexer

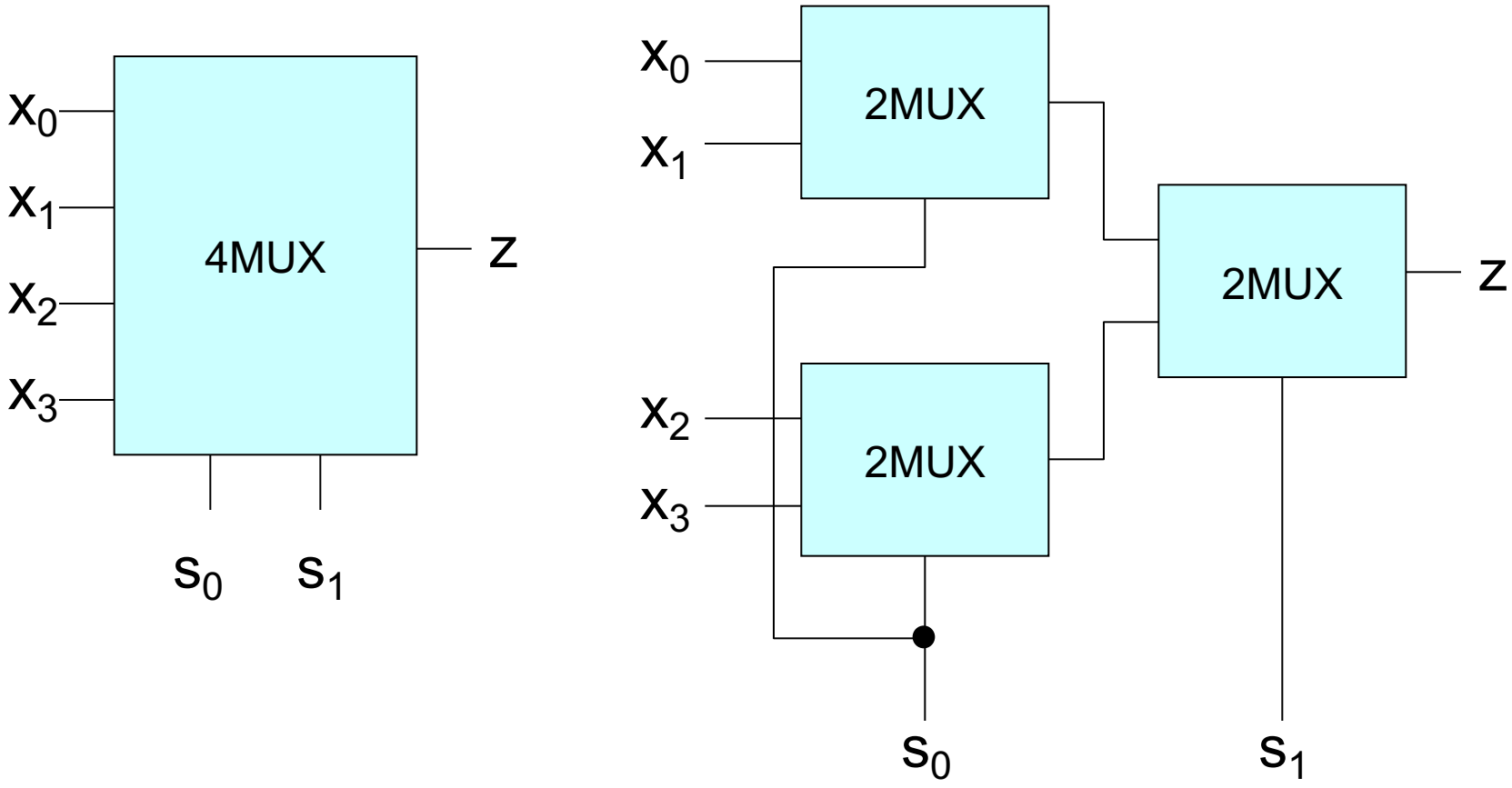
X	Y	S	$Y \wedge S$	$\neg S$	$X \wedge \neg S$	$(Y \wedge S) \vee (X \wedge \neg S)$
F	F	F	F	T	F	F
F	T	F	F	T	F	F
T	F	F	F	T	T	T
T	T	F	F	T	T	T
F	F	T	F	F	F	F
F	T	T	T	F	F	T
T	F	T	F	F	F	F
T	T	T	T	F	F	T

4-to-1 multiplexer





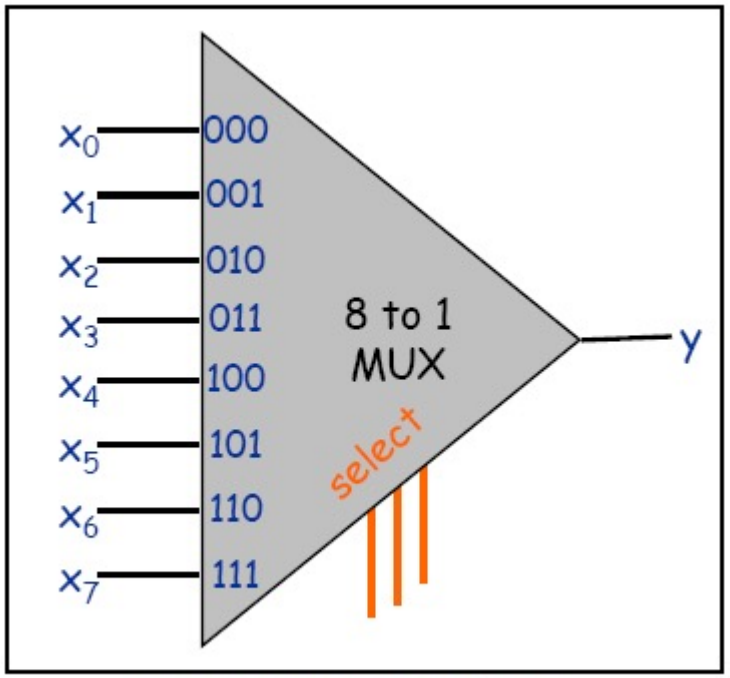
4-to-1 multiplexer



8-to-1 Multiplexer

2^N-to-1 multiplexer

- N select inputs, 2^N data inputs, 1 output
- Copies "selected" data input bit to output

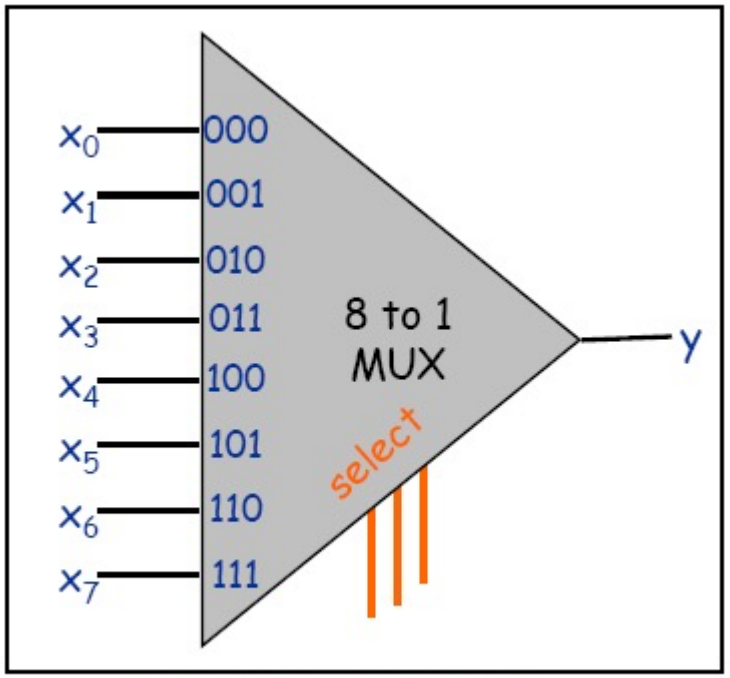


8-to-1 Mux Interface

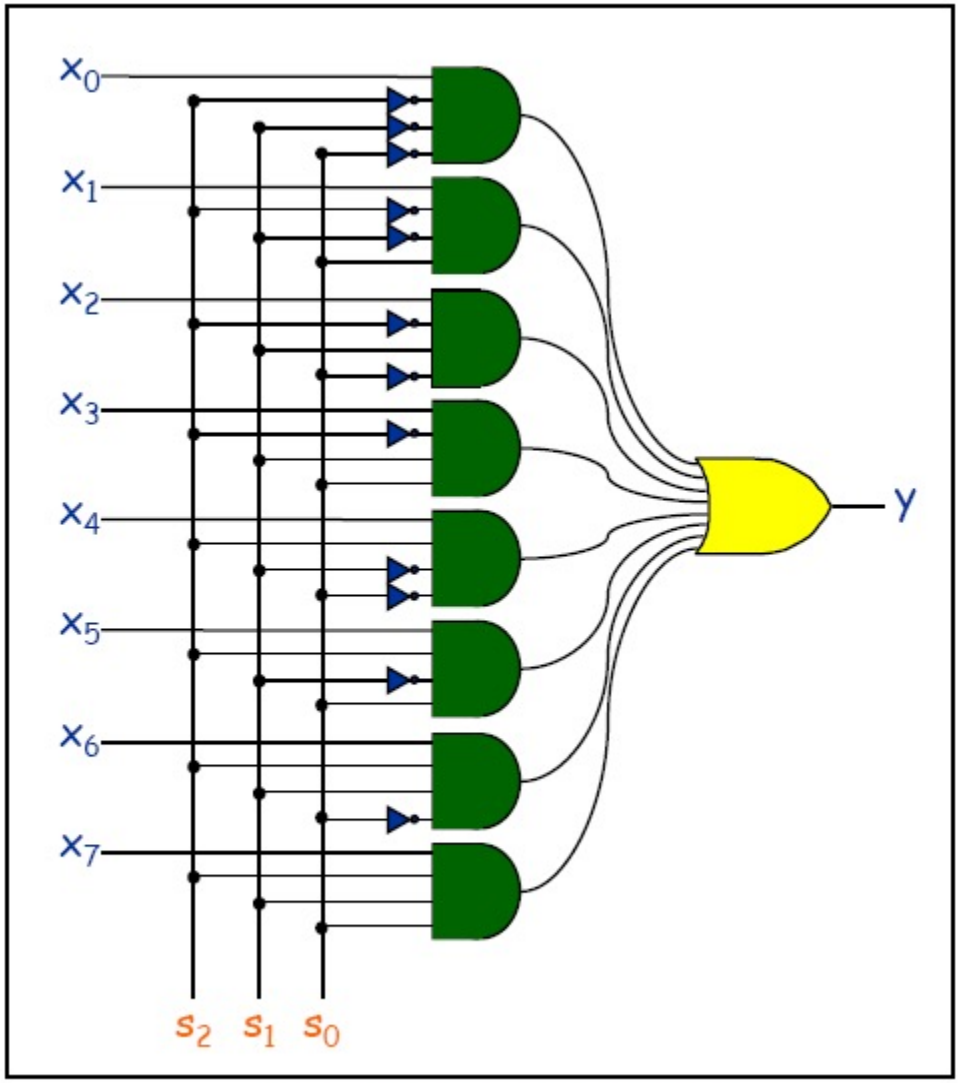
8-to-1 Multiplexer

2^N-to-1 multiplexer

- N select inputs, 2^N data inputs, 1 output
- Copies "selected" data input bit to output



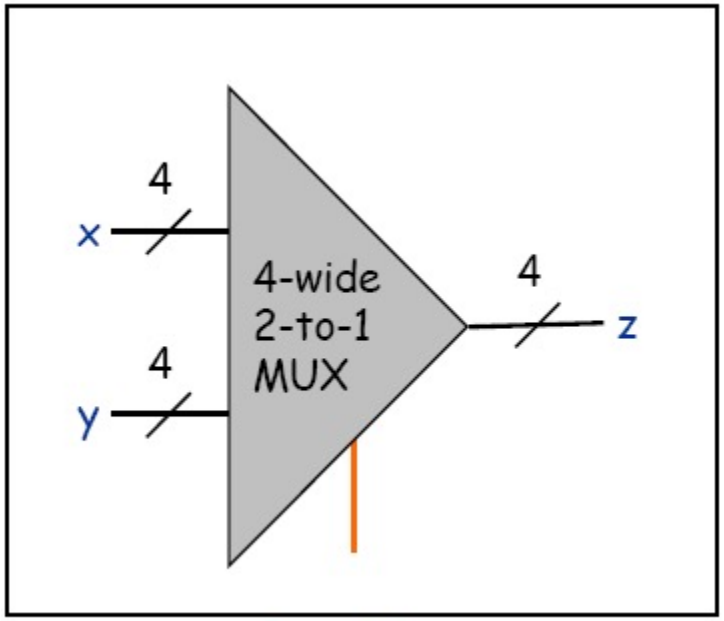
8-to-1 Mux Interface



8-to-1 Mux Implementation

4-Wide 2-to-1 Multiplexer

Goal: select from one of two 4-bit buses

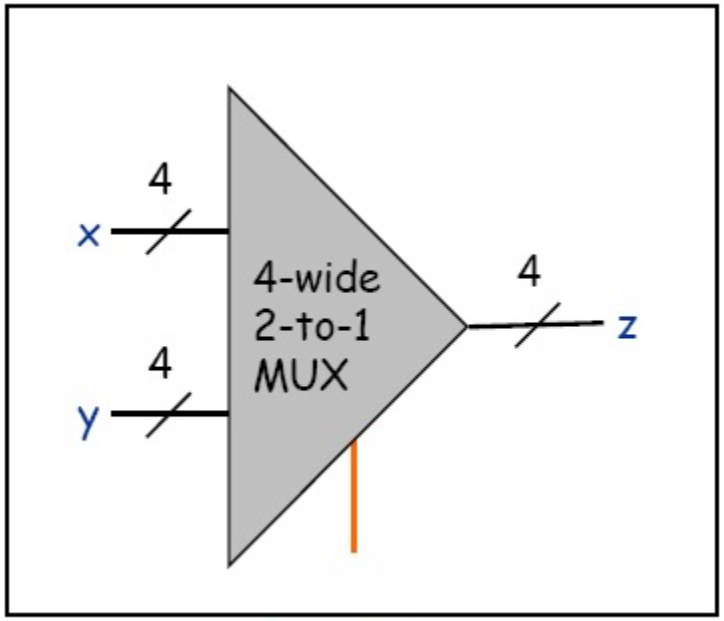


Interface

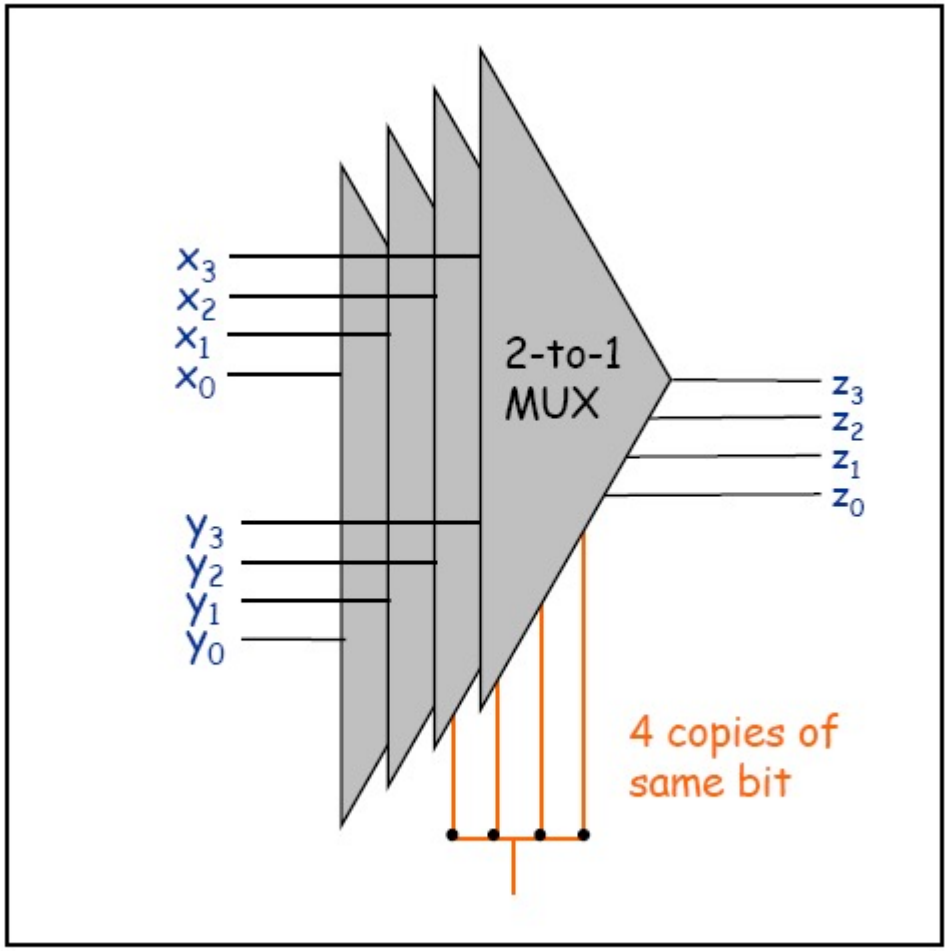
4-Wide 2-to-1 Multiplexer

Goal: select from one of two 4-bit buses

- Implemented by layering 4 2-to-1 multiplexer



Interface

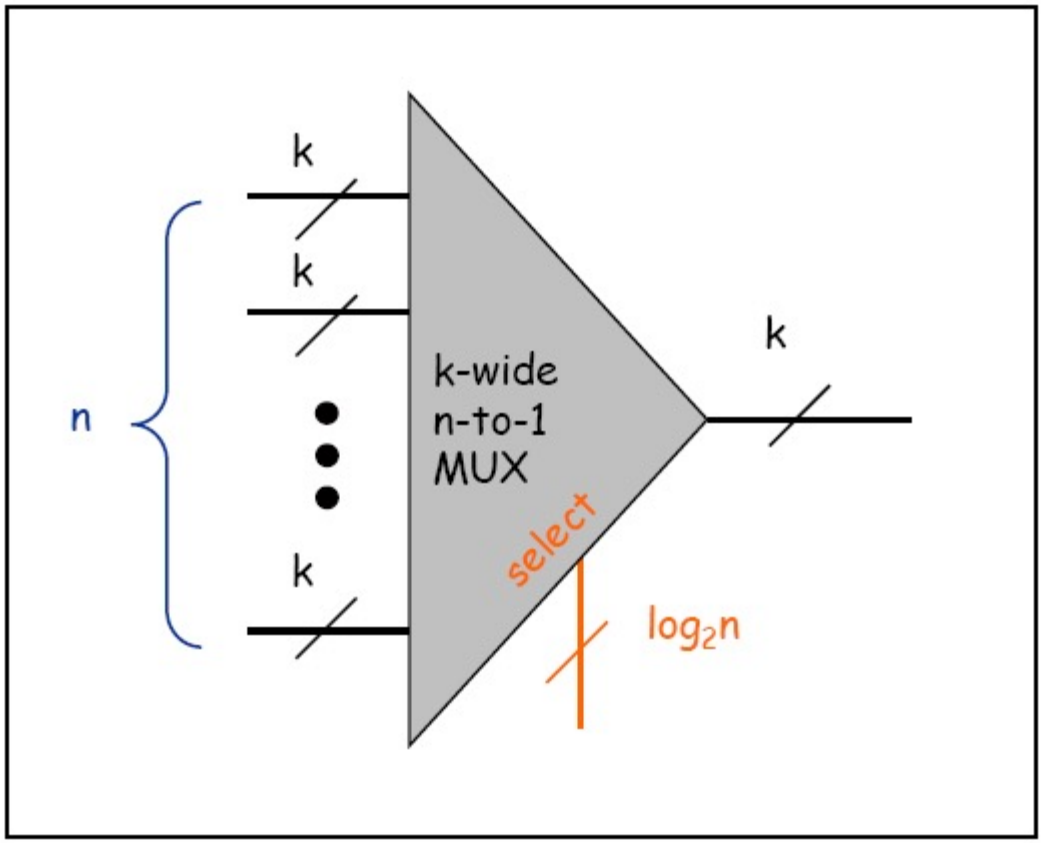


Implementation

k-Wide n-to-1 Multiplexer

Goal: select from one of n k-bit buses

- Implemented by layering k n-to-1 multiplexer

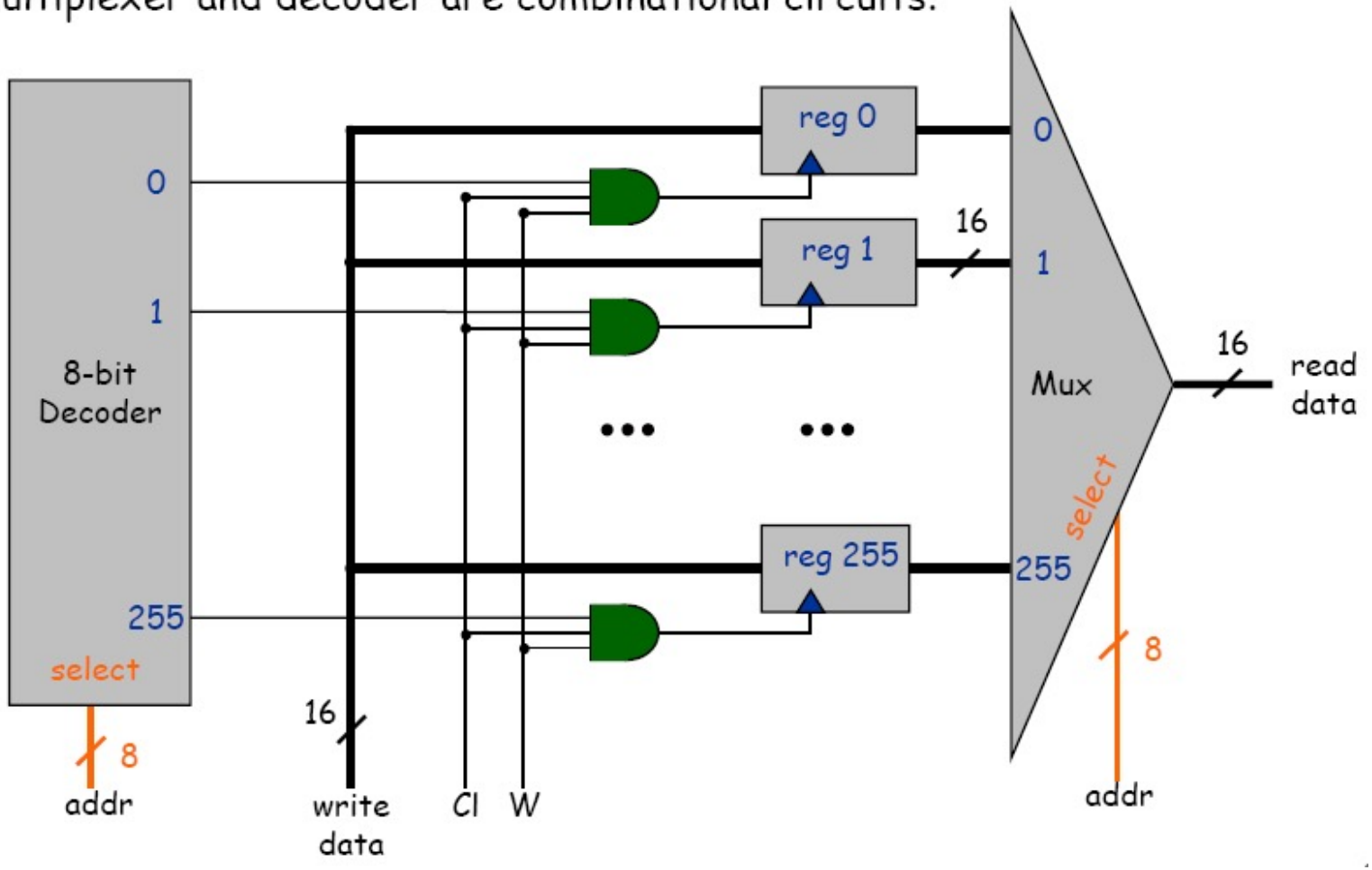


Interface

Register file implementation

Implementation example: TOY main memory.

- Use 256 16-bit registers.
- Multiplexer and decoder are combinational circuits.



Memory Overview

Computers and TOY have several memory components.

- Program counter.
- Registers.
- Main memory.

Implementation. Use one flip-flop for each bit of memory.

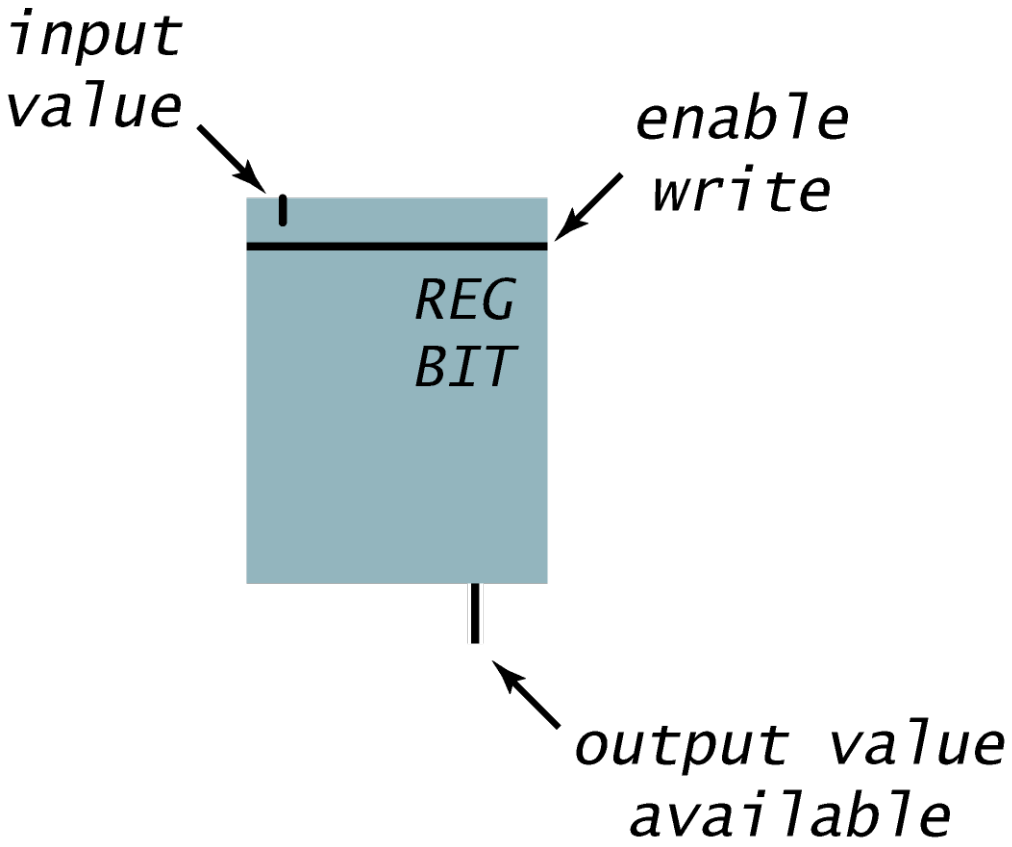
Access. Memory components have different access mechanisms.

TOY has 16 bit words,
8 bit memory addresses, and
4 bit register names.

Organization. Need mechanism to manipulate groups of related bits.

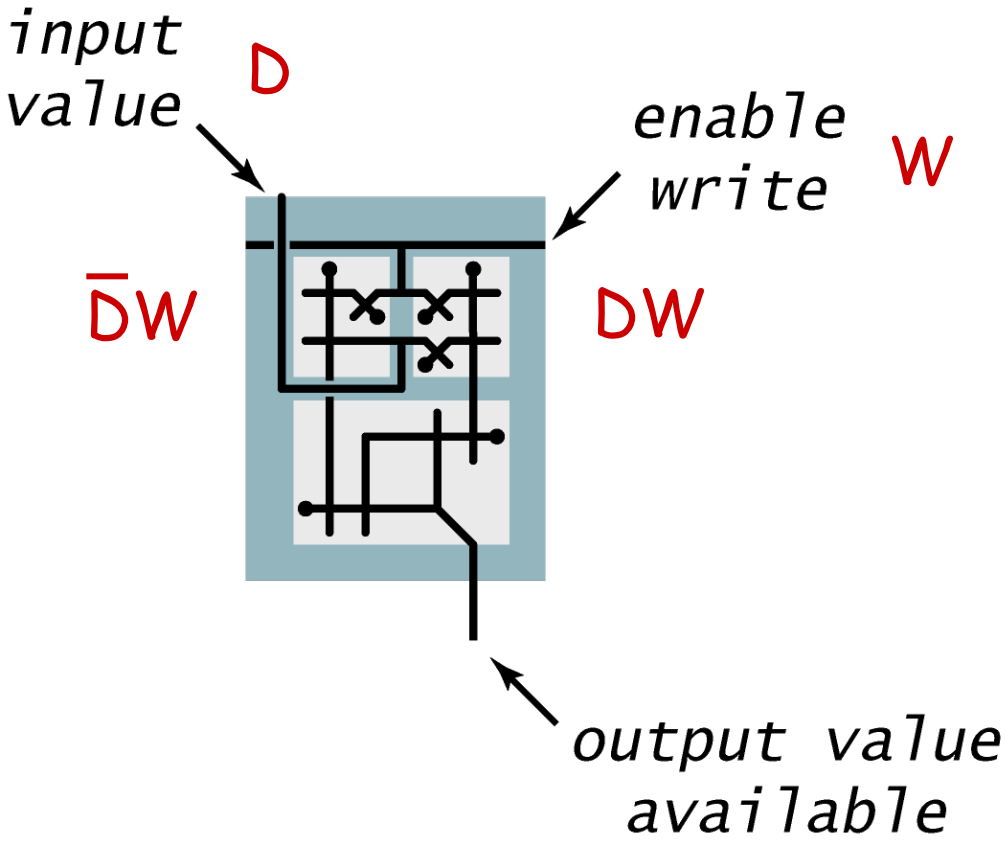
Register Bit

Register bit. Extend a flip-flop to allow easy access to values.



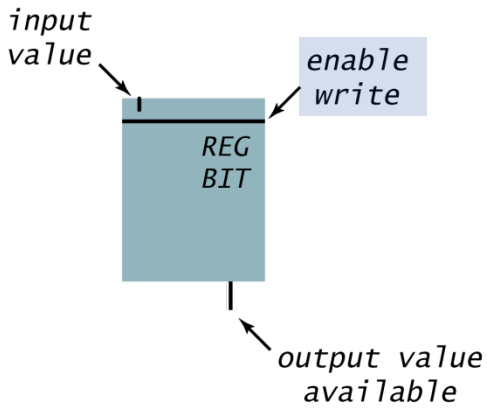
Register Bit

Register bit. Extend a flip-flop to allow easy access to values.

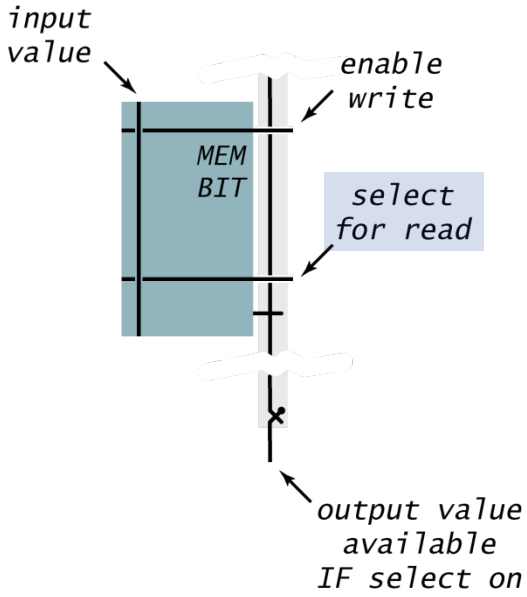


Memory Bit: Interface

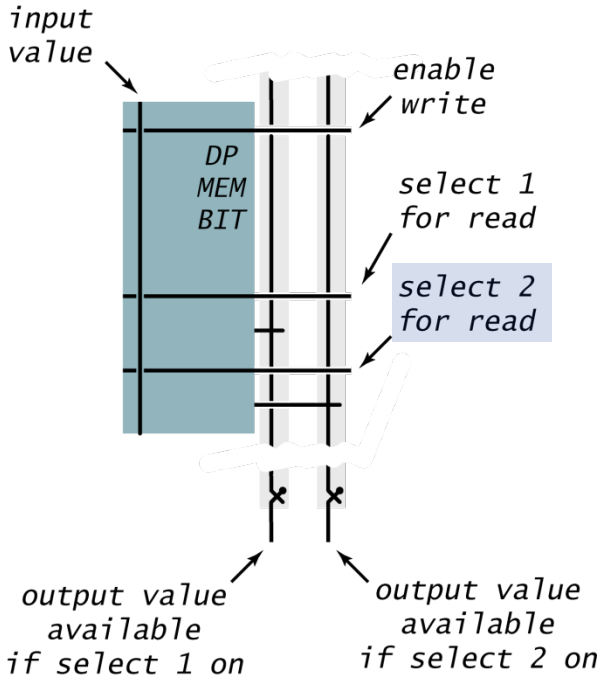
Memory bit. Extend a flip-flop to allow easy access to values.



REGISTER BIT



MEMORY-BANK BIT

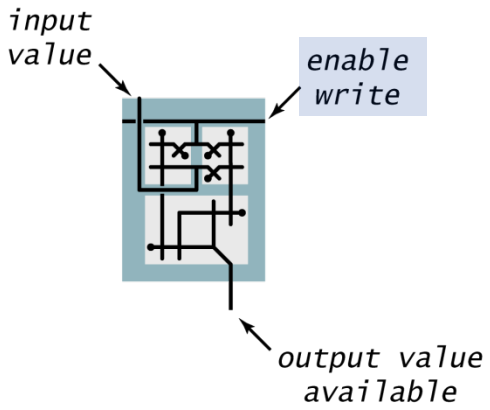


DUAL-PORT MEMORY-BANK BIT

[TOY PC, IR] [TOY main memory] [TOY registers]

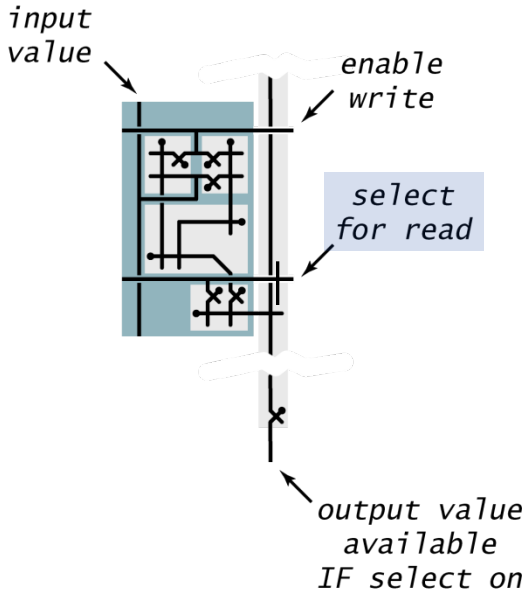
Memory Bit: Switch Level Implementation

Memory bit. Extend a flip-flop to allow easy access to values.



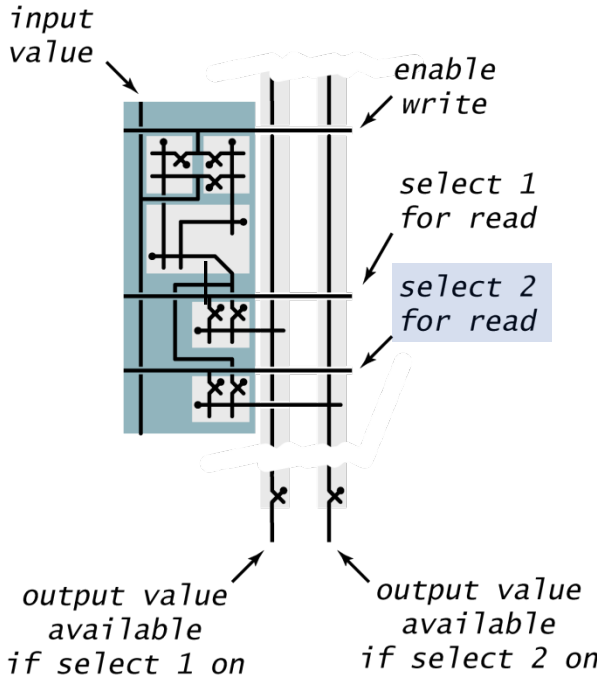
REGISTER BIT

[TOY PC, IR]



MEMORY-BANK BIT

[TOY main memory]



DUAL-PORT MEMORY-BANK BIT

[TOY registers]

Processor Register

Processor register.

- Stores k bits.
- Register contents always available on output bus.
- If enable write is asserted, k input bits get copied into register.

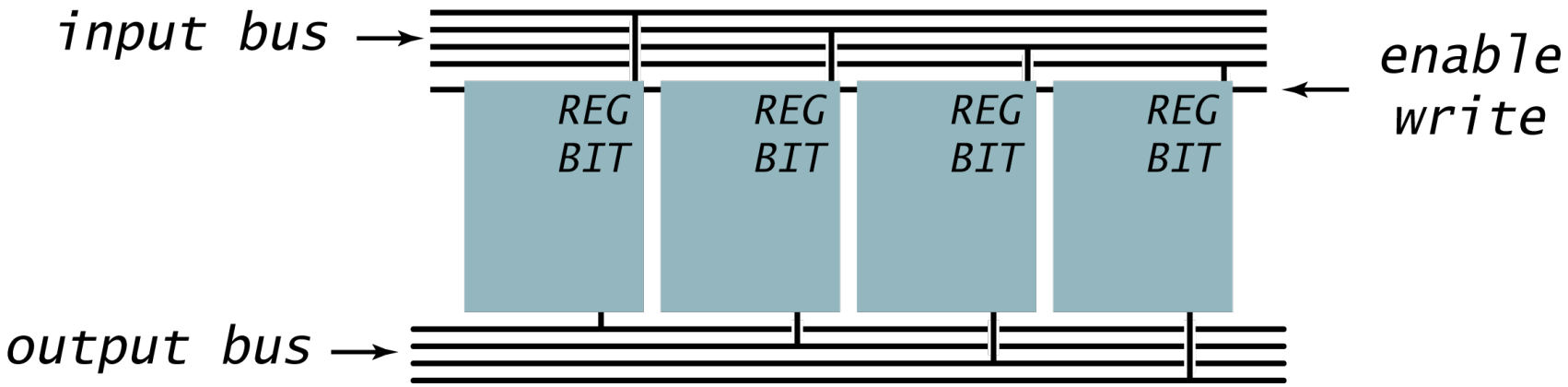
Ex 1. TOY program counter (PC) holds 8-bit address.
Ex 2. TOY instruction register (IR) holds 16-bit current instruction.

Processor Register

Processor register.

- Stores k bits.
- Register contents always available on output bus.
- If enable write is asserted, k input bits get copied into register.

Ex 1. TOY program counter (PC) holds 8-bit address.
Ex 2. TOY instruction register (IR) holds 16-bit current instruction.

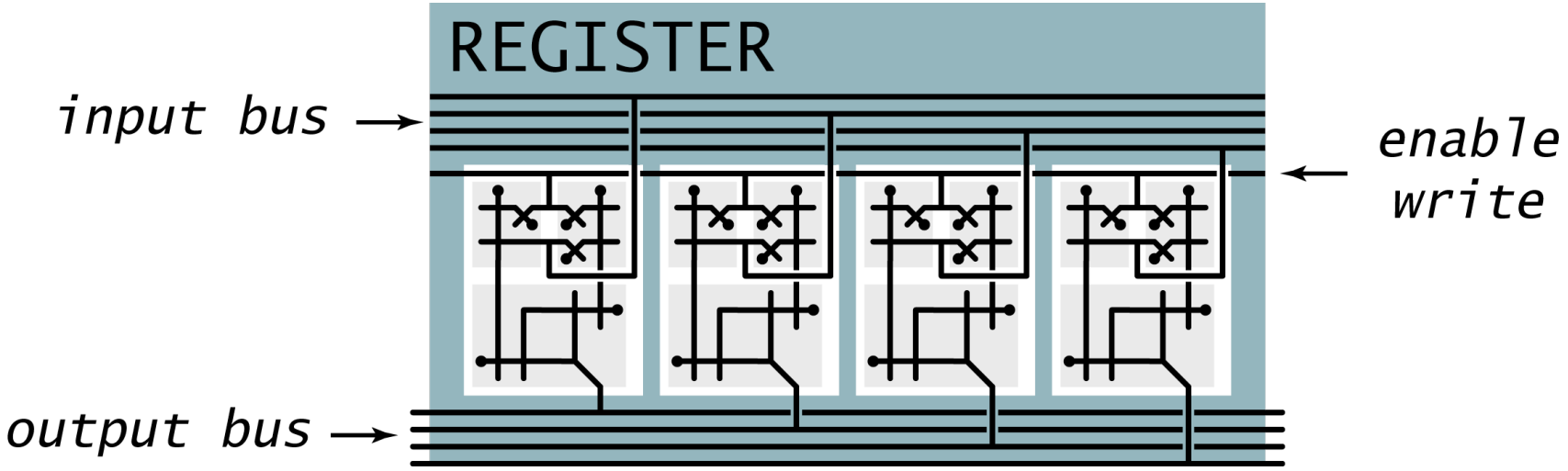


Processor Register

Processor register.

- Stores k bits.
- Register contents always available on output bus.
- If enable write is asserted, k input bits get copied into register.

Ex 1. TOY program counter (PC) holds 8-bit address.
Ex 2. TOY instruction register (IR) holds 16-bit current instruction.



Memory Bank

Memory bank.

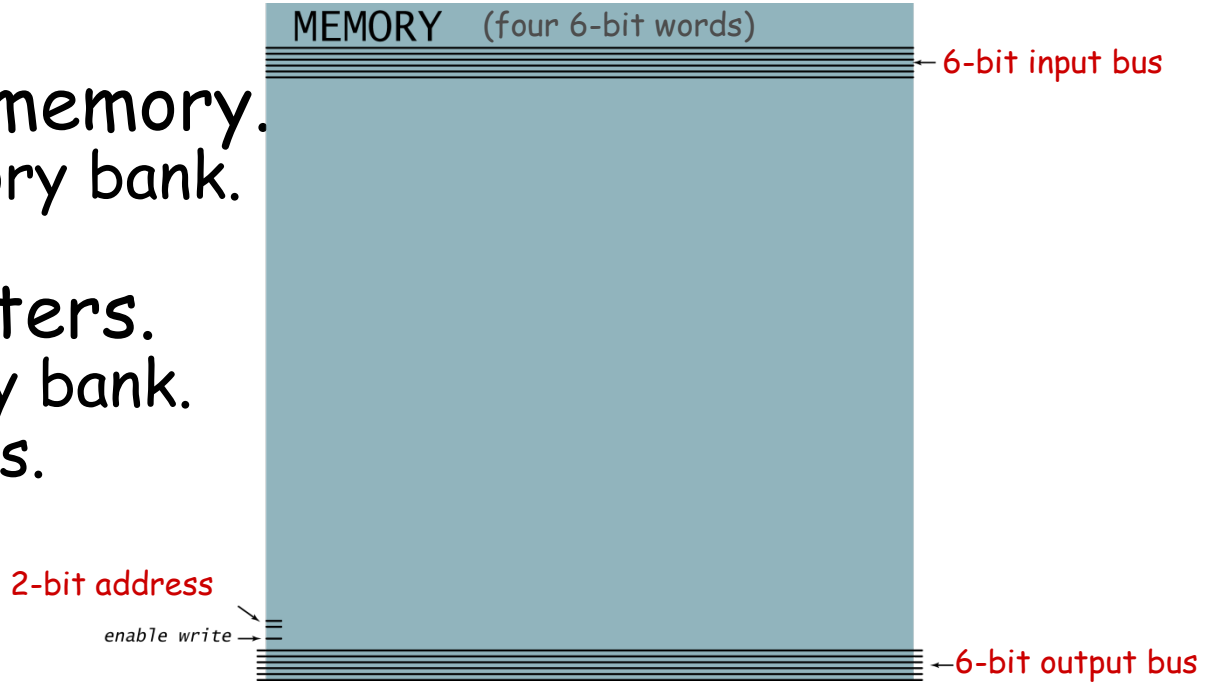
- Bank of n registers; each stores k bits.
- Read and write information to *one* of n registers.
- Address inputs specify which one. ← $\log_2 n$ address bits needed
- Addressed bits always appear on output.
- If write enabled, k input bits are copied into addressed register.

Ex 1. TOY main memory.

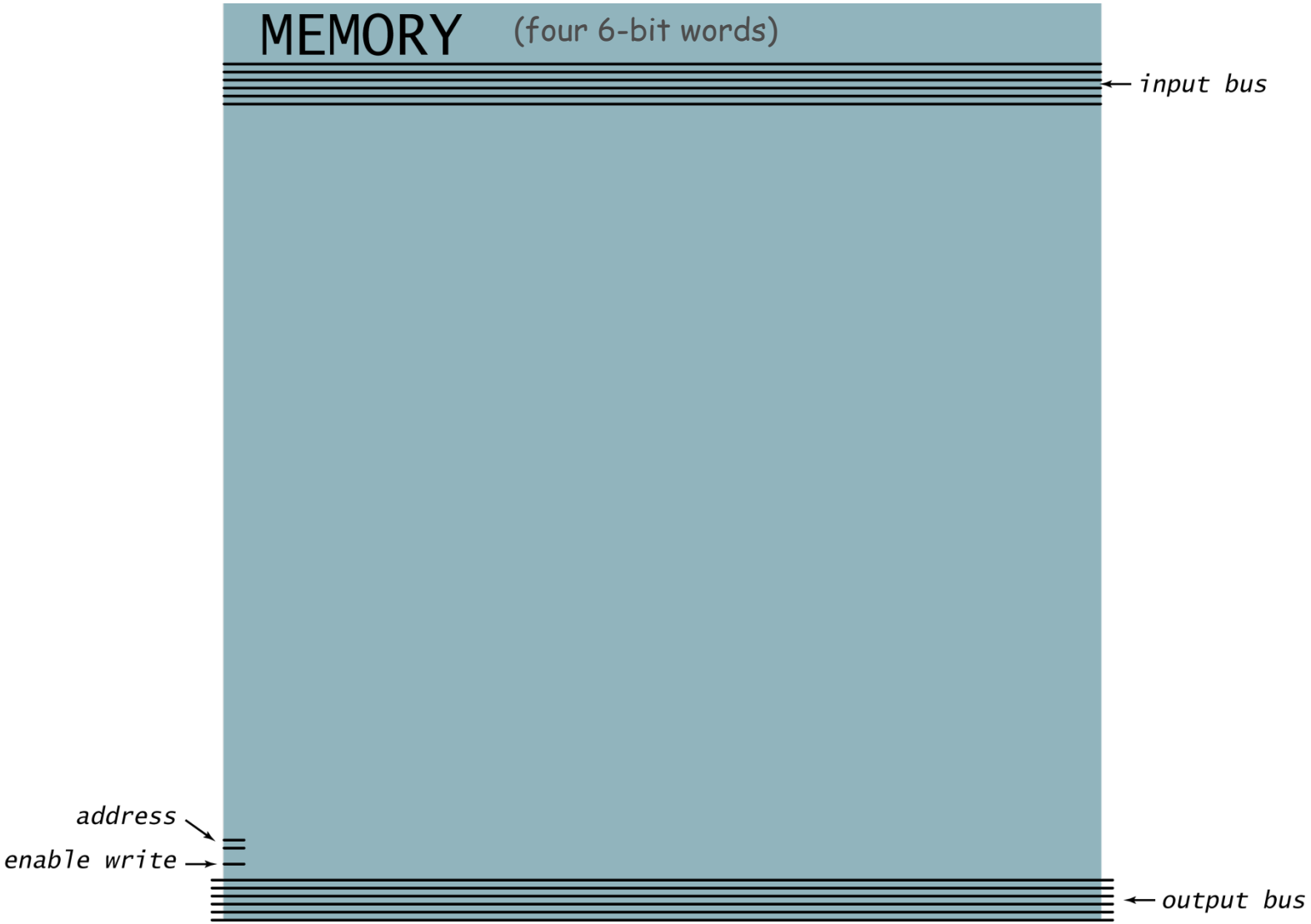
- 256-by-16 memory bank.

Ex 2. TOY registers.

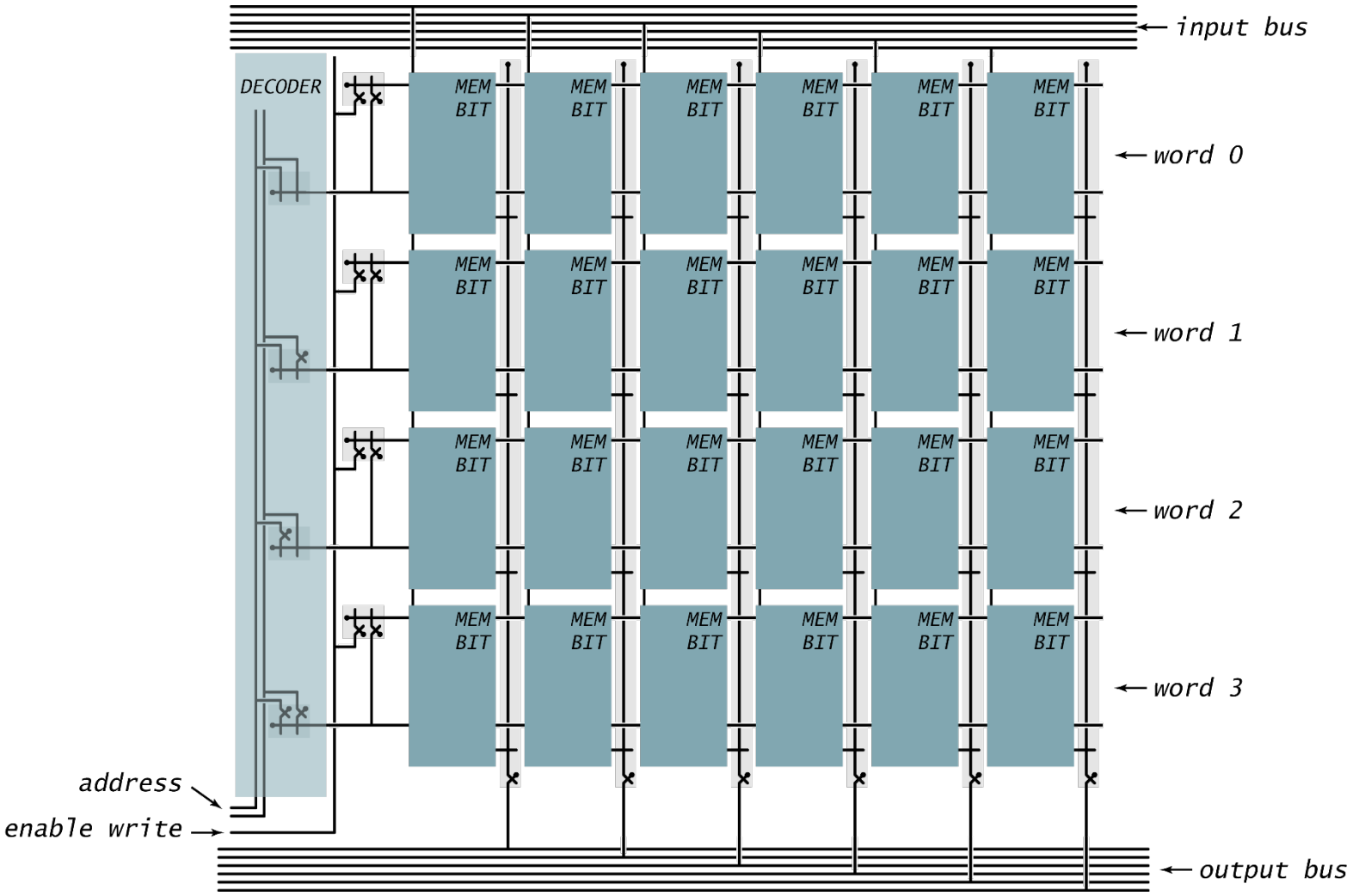
- 16-by-16 memory bank.
- Two output buses.



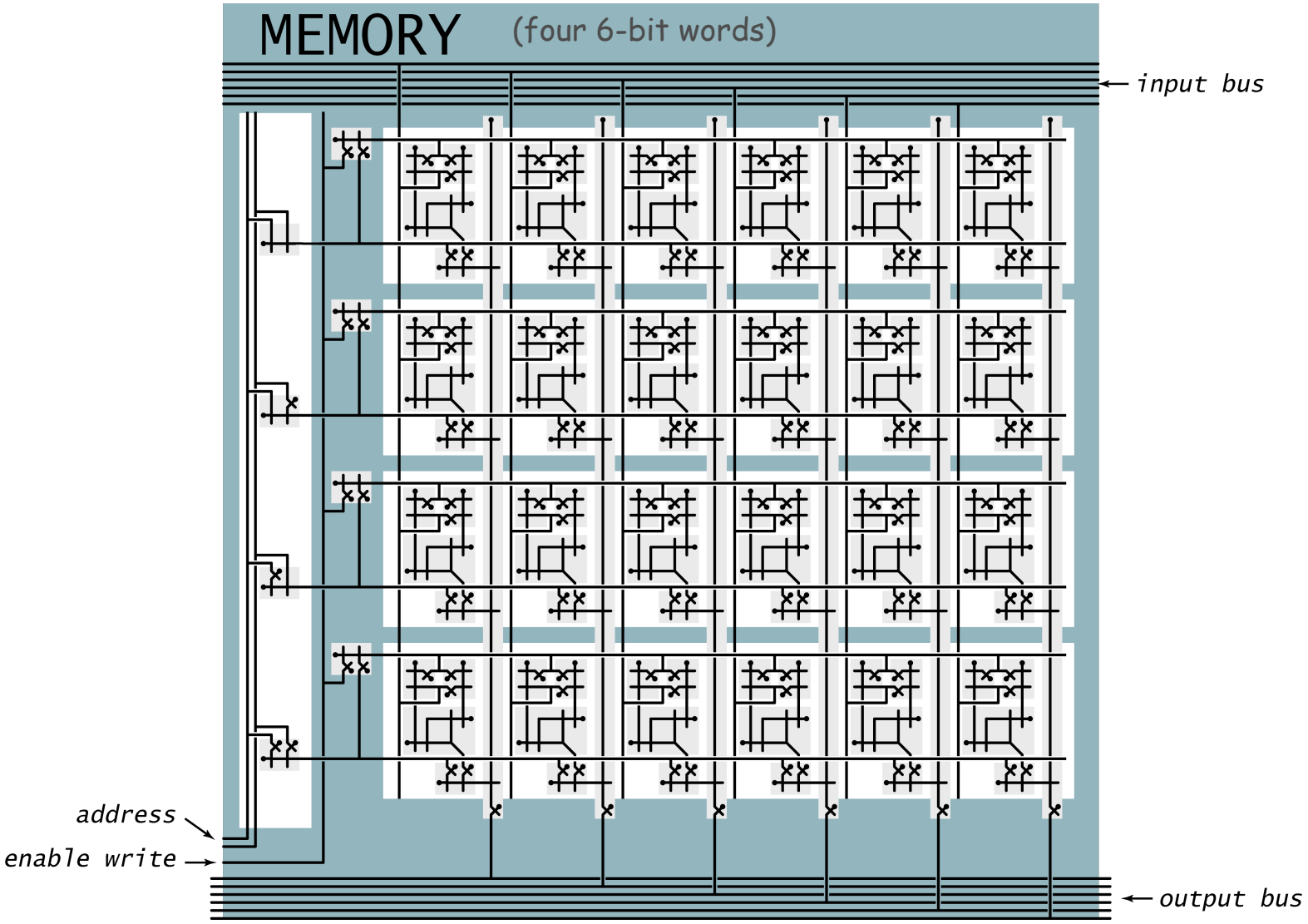
Memory: Interface



Memory: Component Level Implementation



Memory: Switch Level Implementation



TOY sequential circuits

Sequential circuits add "state" to digital hardware.

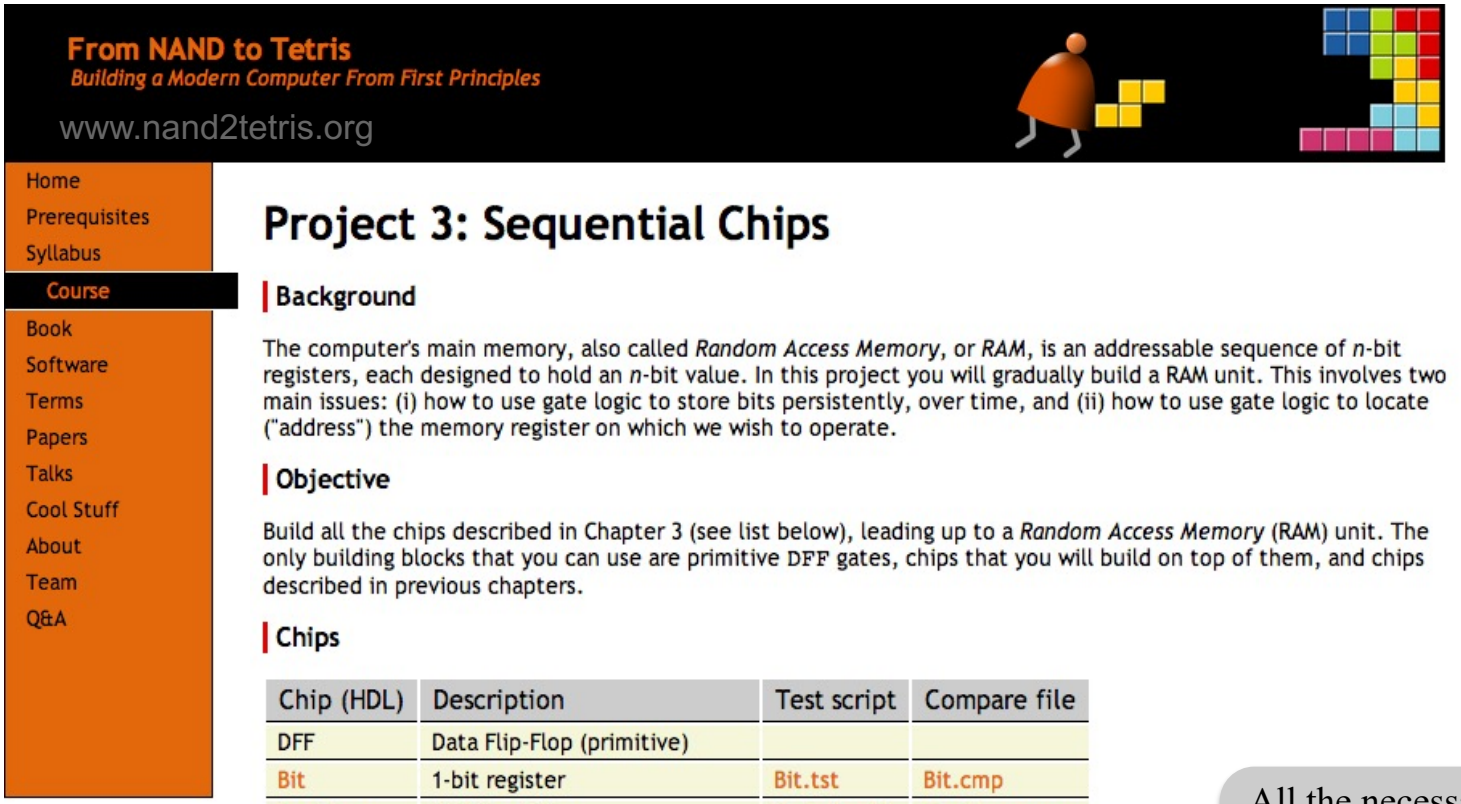
- Flip-flop. [represents 1 bit]
- TOY word. [16 flip-flops]
- TOY registers. [16 words]
- TOY main memory. [256 words]

Modern technologies for registers and main memory are different.

- Few registers, easily accessible, high cost per bit.
- Huge main memories, less accessible, low cost per bit.
- Drastic evolution of technology over time.

Next. Build a complete TOY computer.

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

Project 3

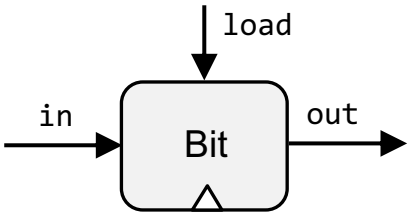
Given:

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

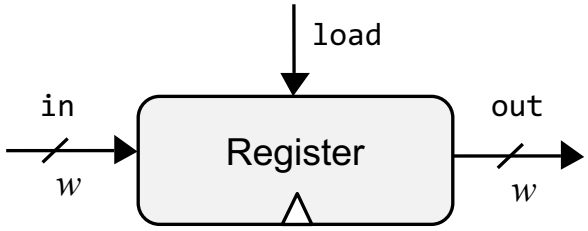
Build:

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

Registers



1-bit register



multi-bit register

Designed to:

"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

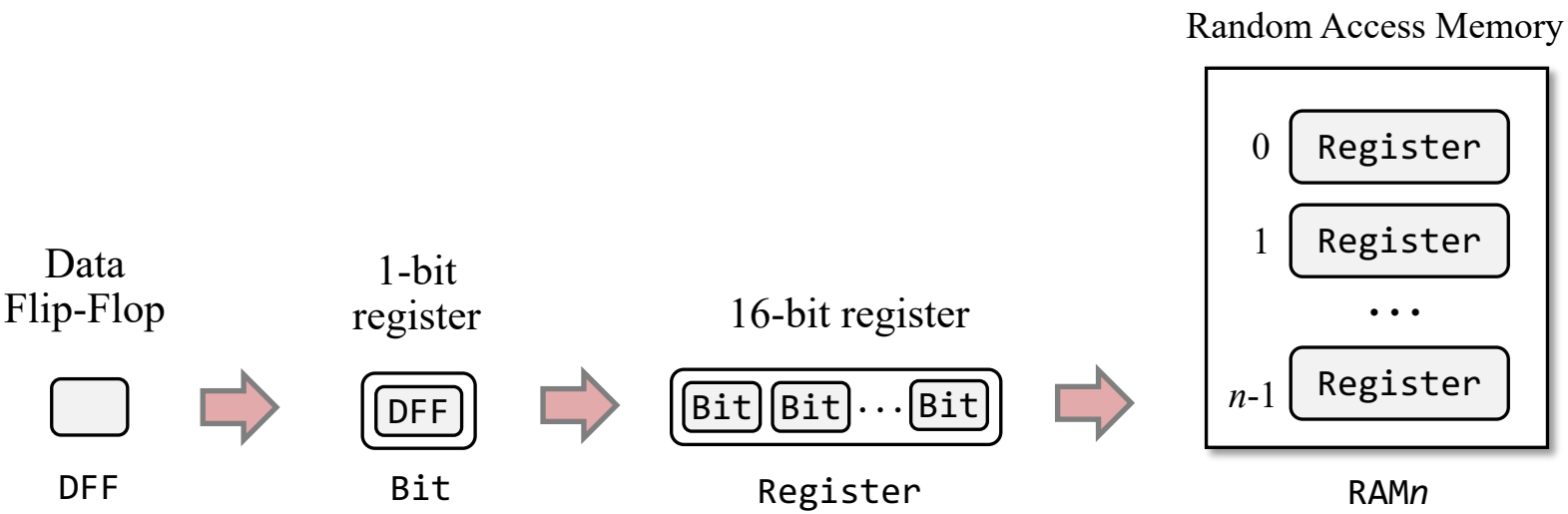
maintaining state

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

loading

maintaining state

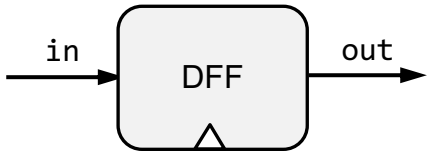
Memory hierarchy



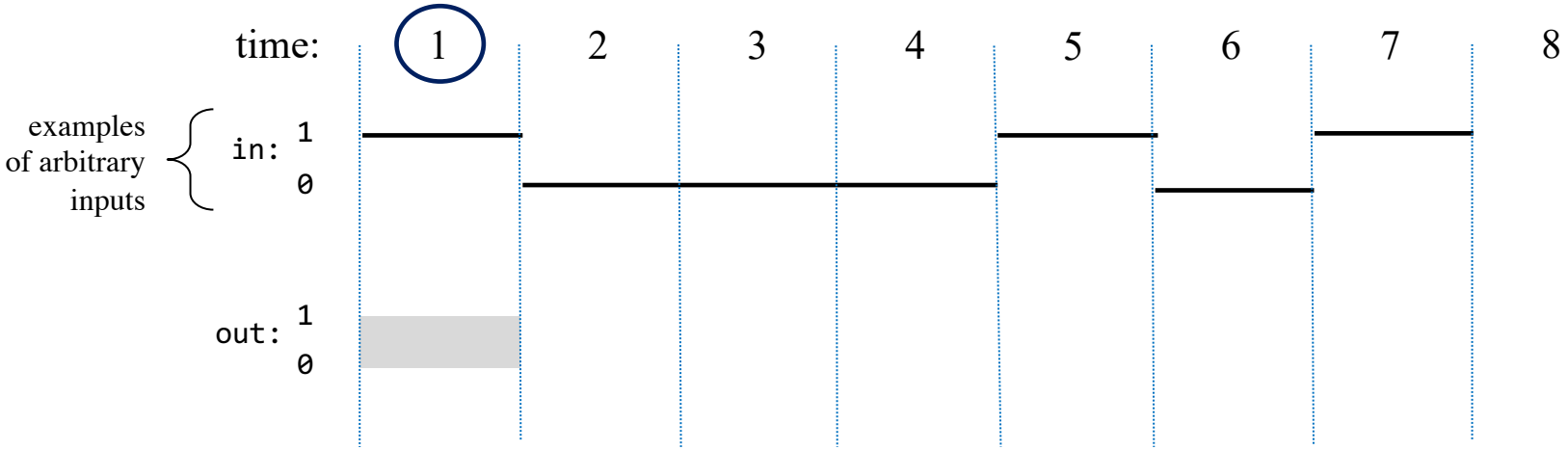
DFF

Data Flip Flop (aka *latch*)

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



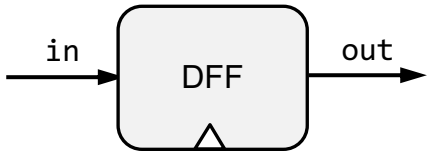
$\texttt{out}(t) = \texttt{in}(t-1)$



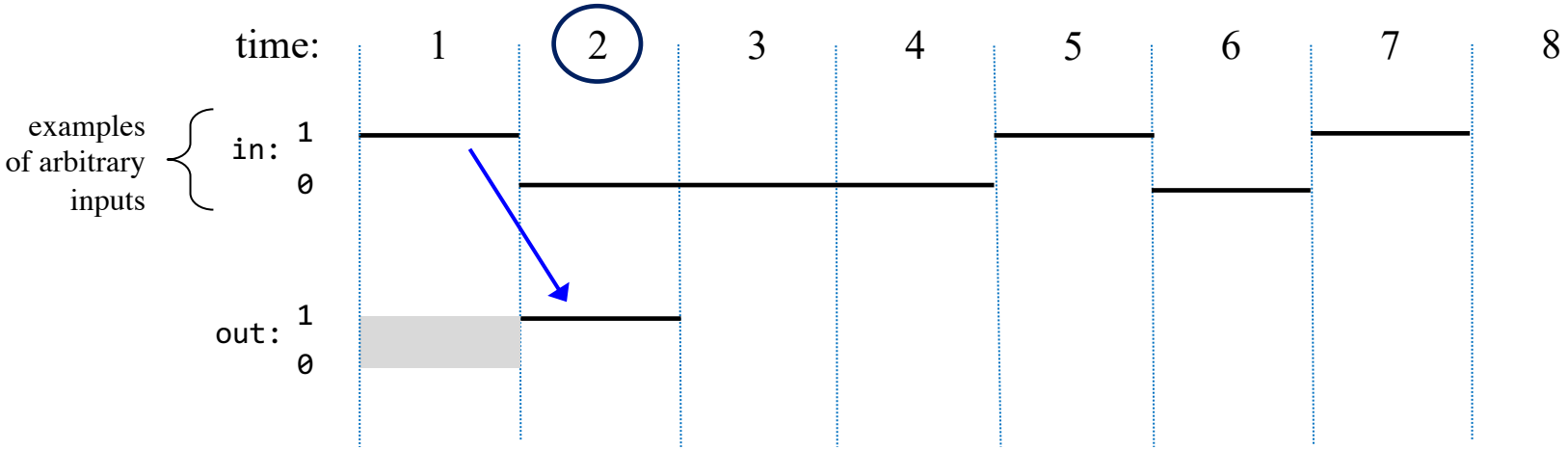
DFF

Data Flip Flop (aka *latch*)

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



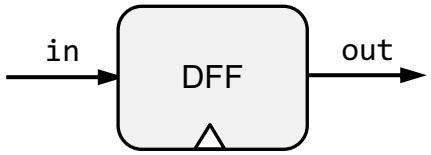
$\texttt{out}(t) = \texttt{in}(t-1)$



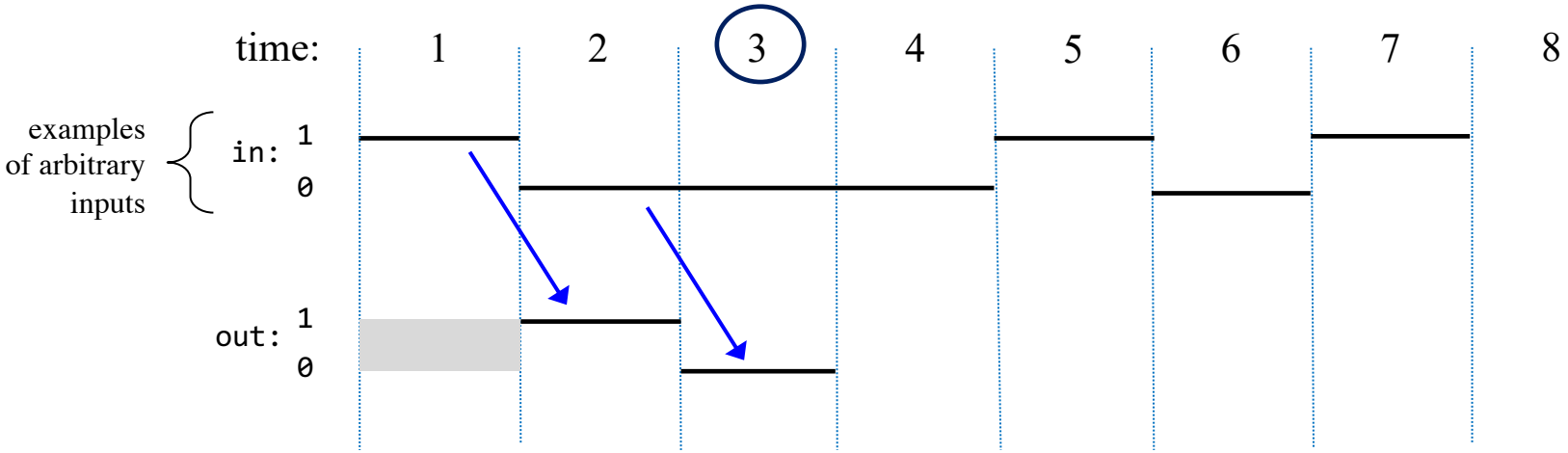
DFF

Data Flip Flop (aka *latch*)

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



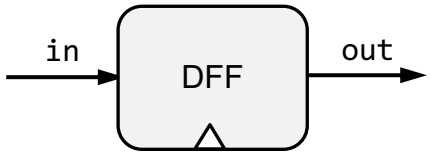
$\texttt{out}(t) = \texttt{in}(t-1)$



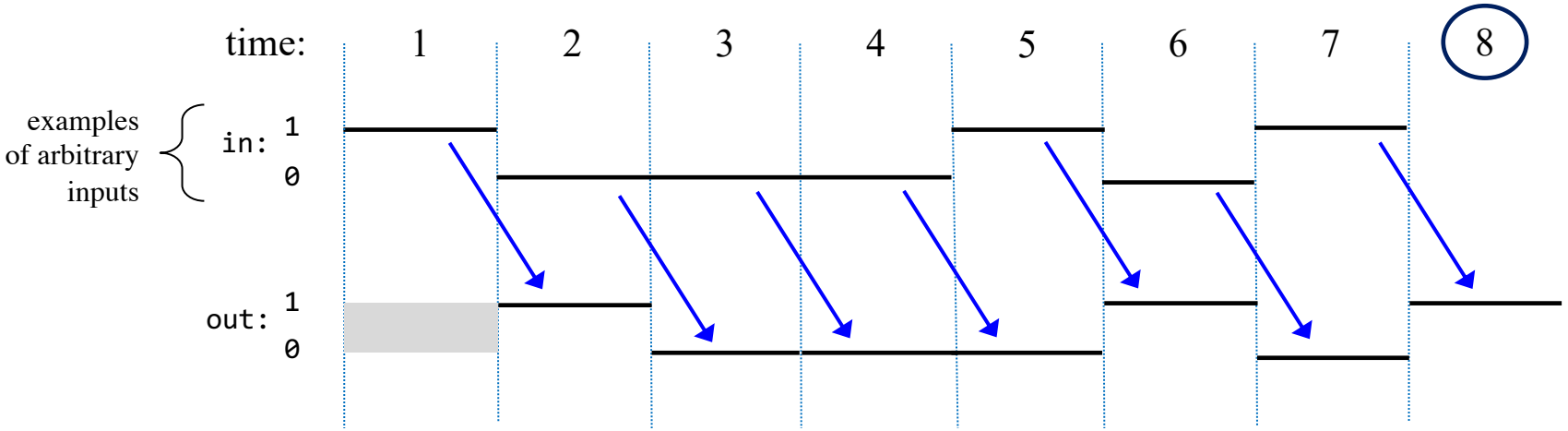
DFF

Data Flip Flop (aka *latch*)

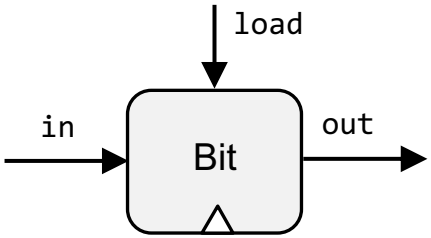
The most elementary sequential gate: Outputs the input in the previous time-step



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

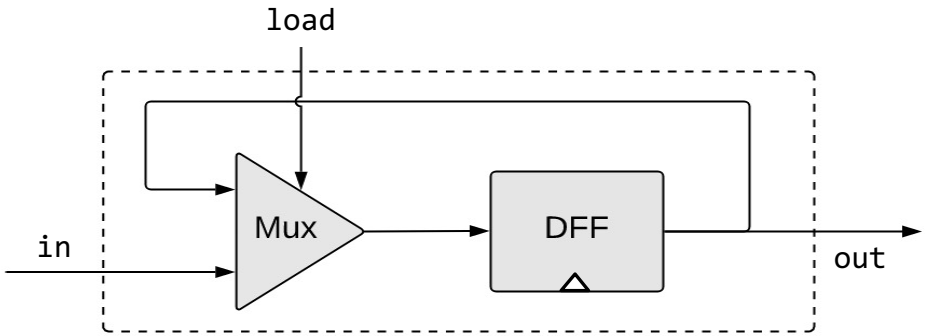


1-bit register



Bit.hdl

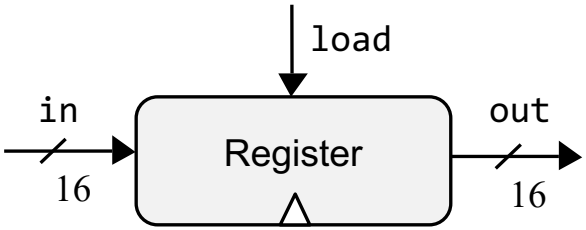
```
/** 1-bit register:
    if  load(t–1) then out(t) = in(t–1)
    else               out(t) = out(t–1))  */

CHIP Bit {
    IN in, load;
    OUT out;

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



Implementation tip:

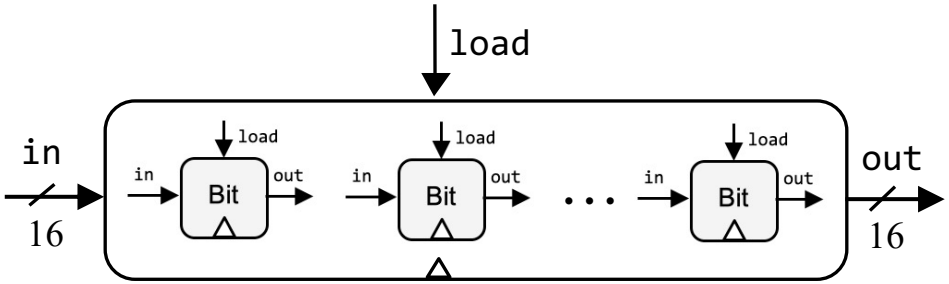
Follow the chip diagram

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

Implementation tip:

Follow the chip diagram

Counter

Later in the course, 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:

Reset: fetch the first instruction

```
PC = 0
```

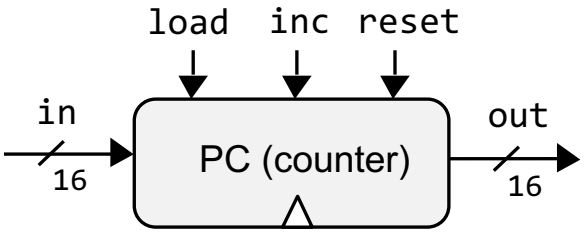
Next: fetch the next instruction

```
PC++
```

Goto: fetch instruction *n*

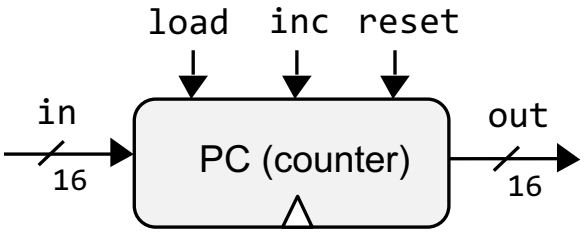
```
PC = n
```

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)

Counter



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

Usage:

To read:

probe `out`

To set:

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

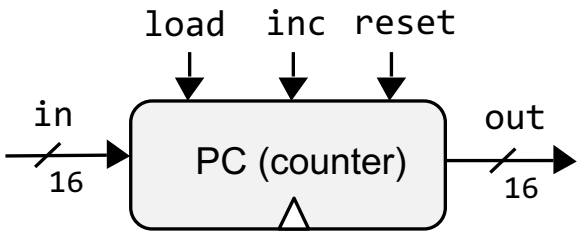
To reset:

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

To count:

assert `inc`,
set the other control bits to `0`

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:
}
```

Implementation tip: Can be built from a Register, an Incrementer, and Mux's

Project 3

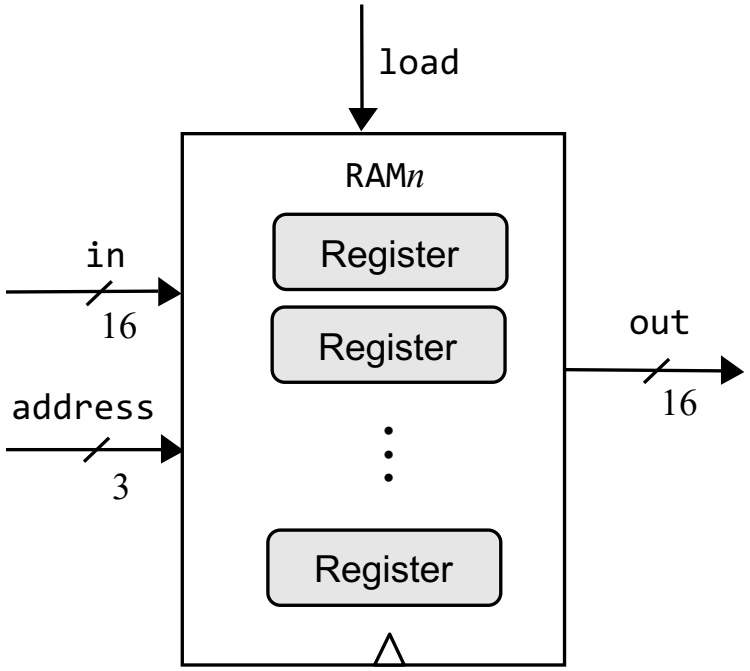
Given:

- 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`

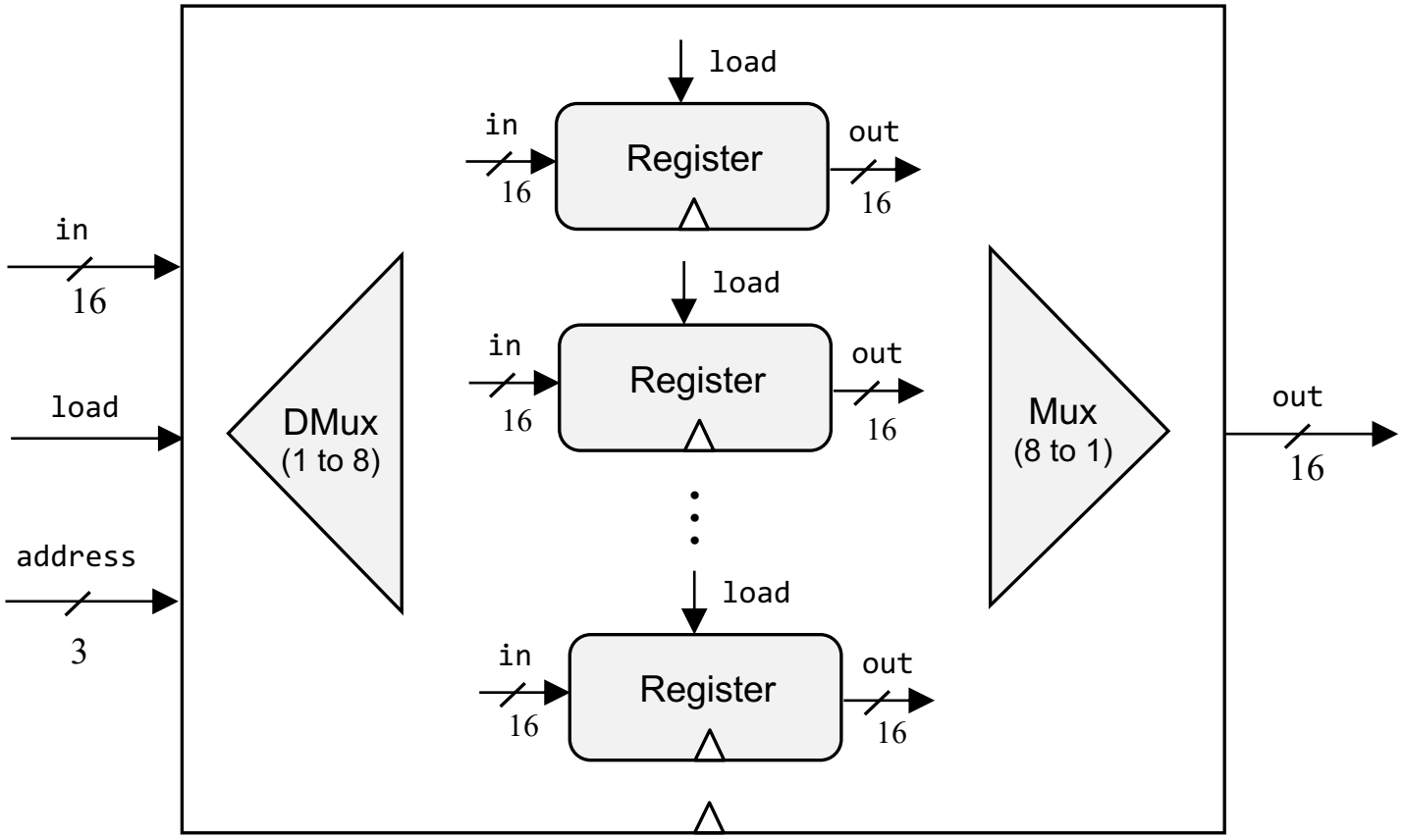
8-register RAM: abstraction



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

Implementation tip:

Follow the chip diagram

Project 3

Given:

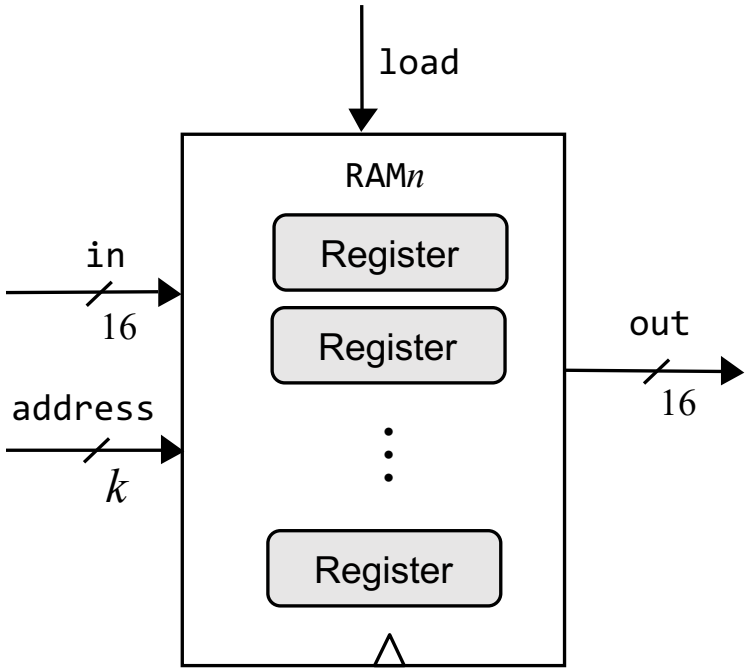
- All the chips built in projects 1 and 2
- Data Flip-Flop (built-in `DFF` gate)

Build the following chips

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

} A family of RAM chips

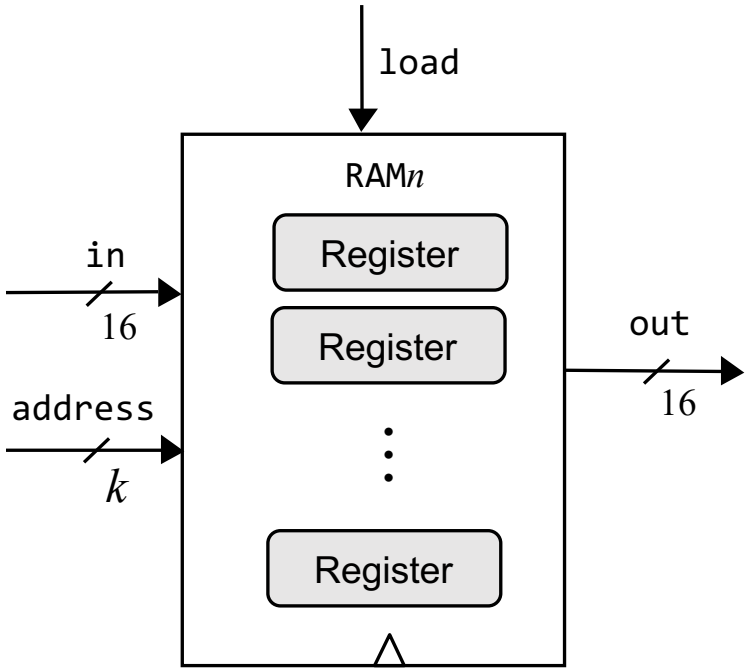
N-Register RAM



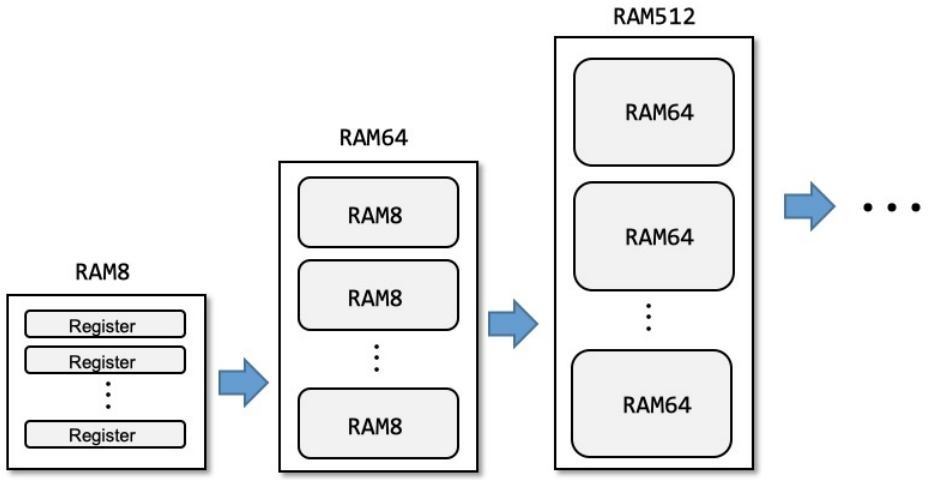
RAM*n*.hdl

```
/*
   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.