Agenda

Hack, the big picture: simulation from gates to computer

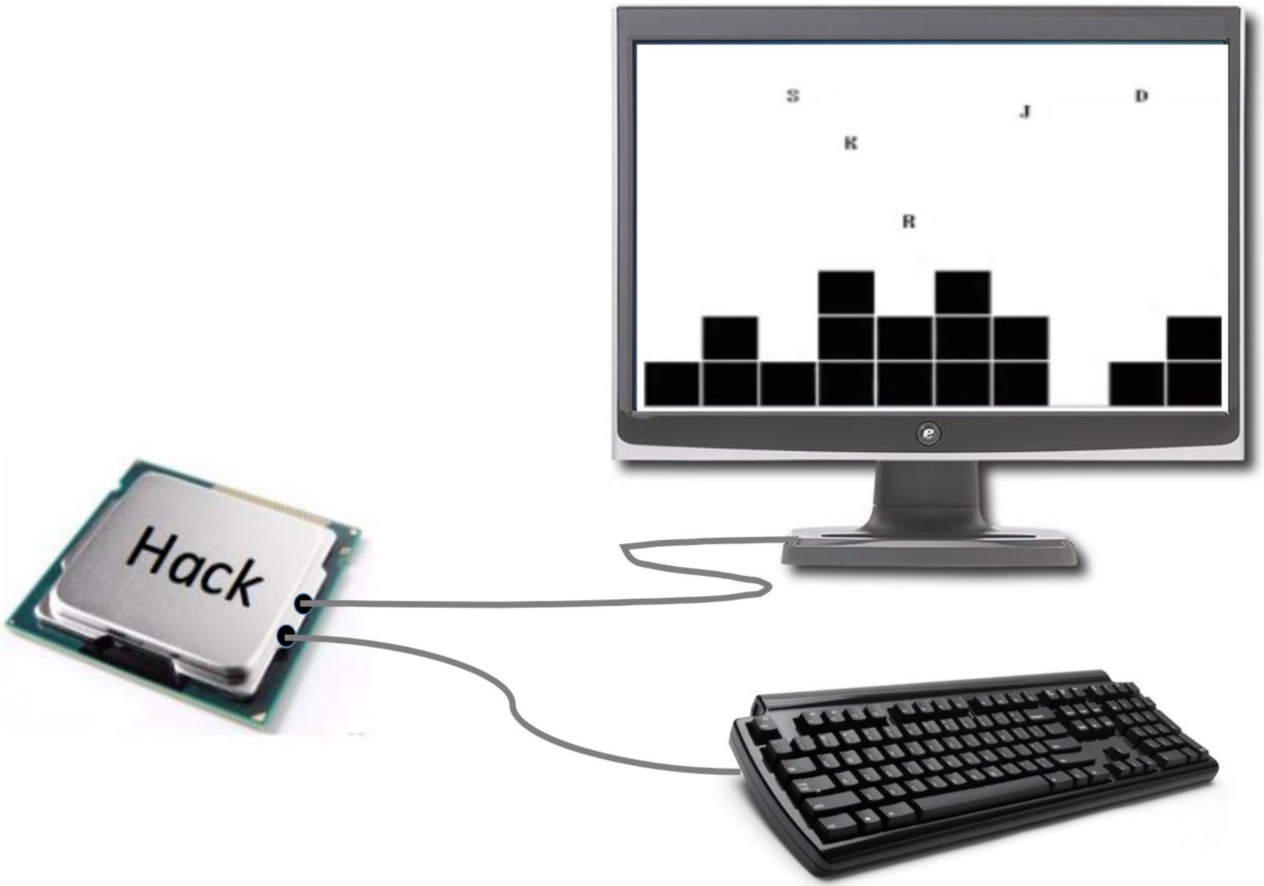
Hack Hardware Architecture

- Gates revisited: Theory
- ALU Implementation
- Storage Implementation

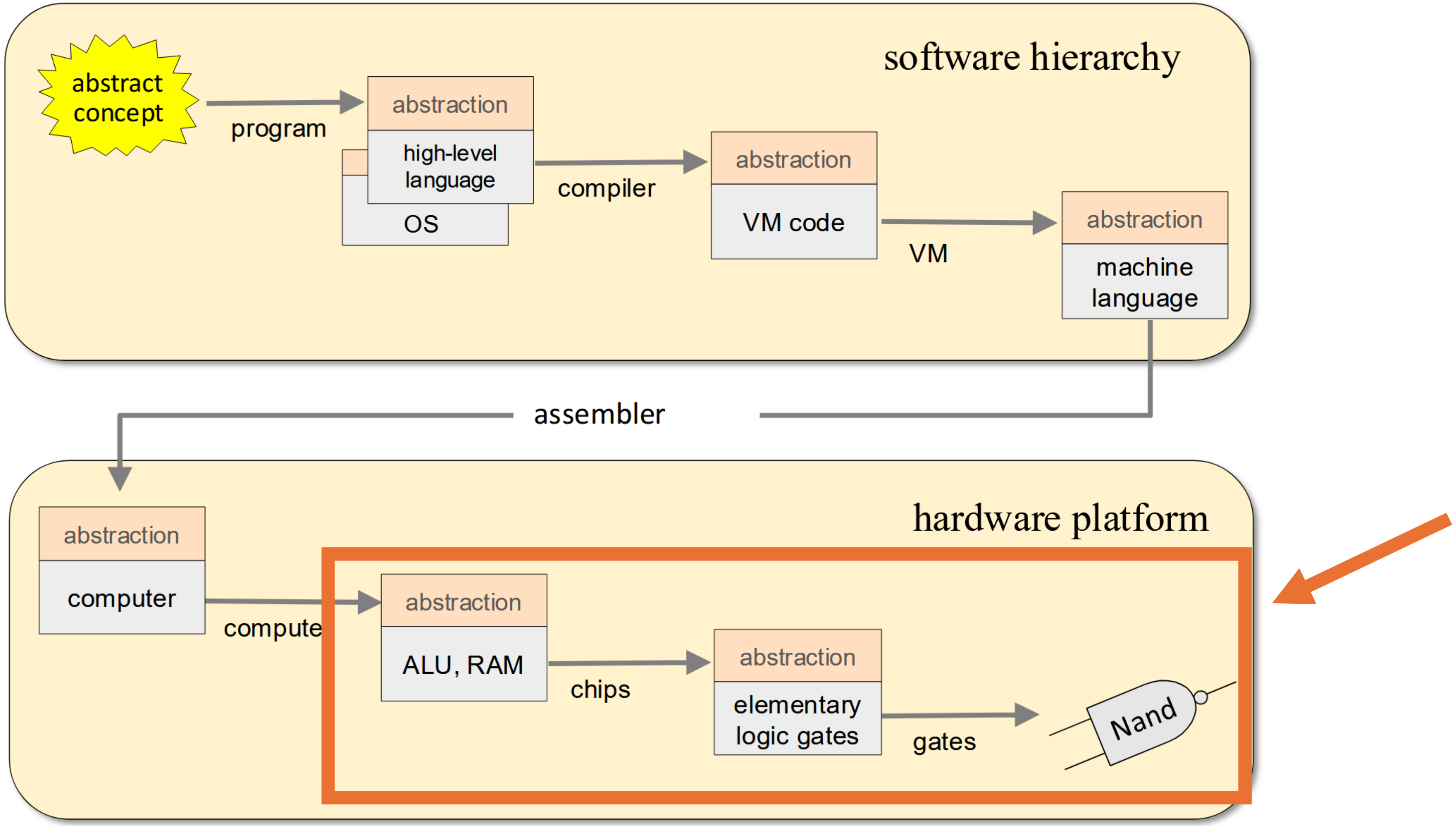
The big picture: Nand to Tetris

Semester Goal: Build a computer from first principles

Idea: Code is translated to hardware instructions through multiple layers of abstraction

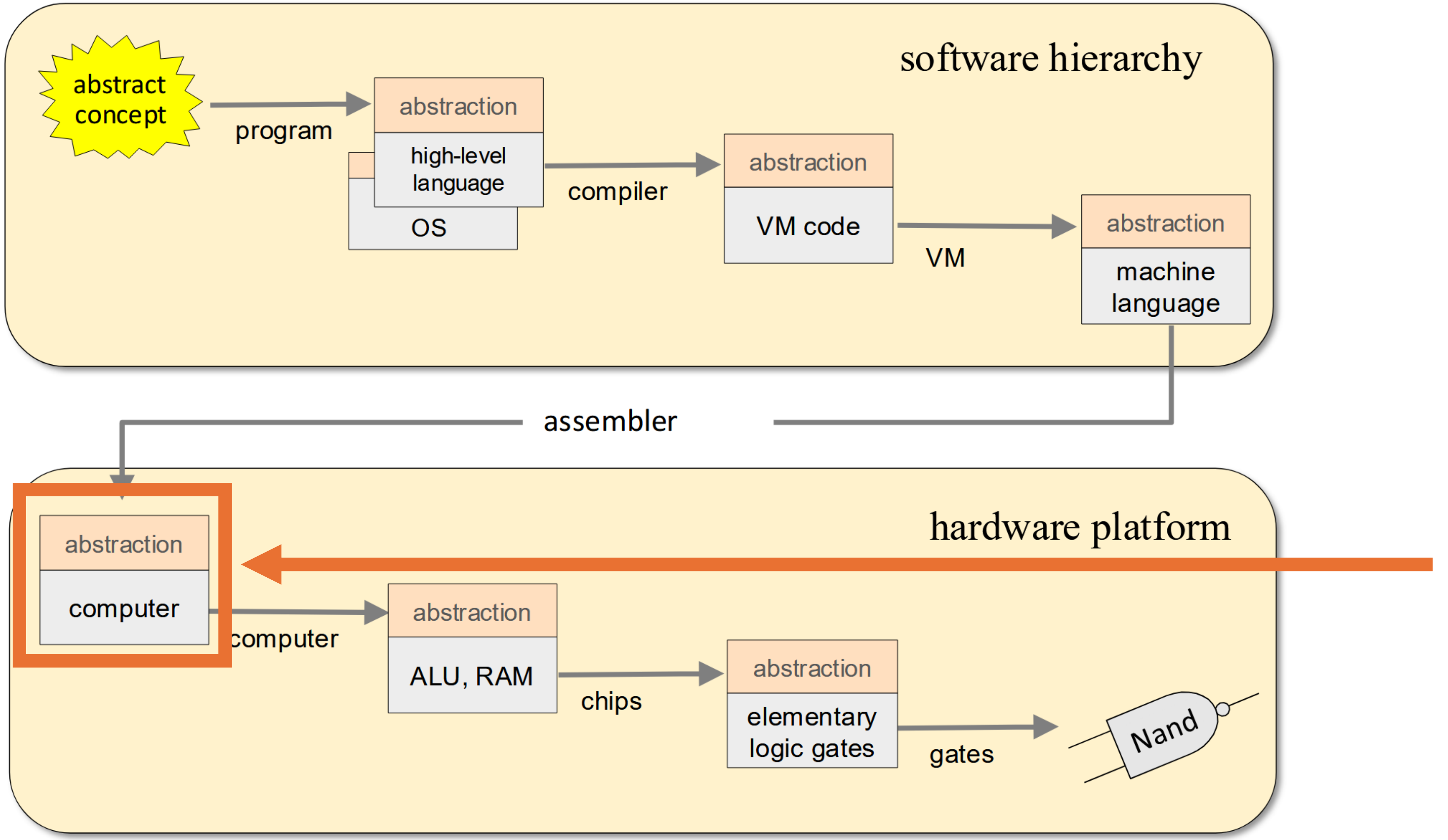


The big picture: Hack



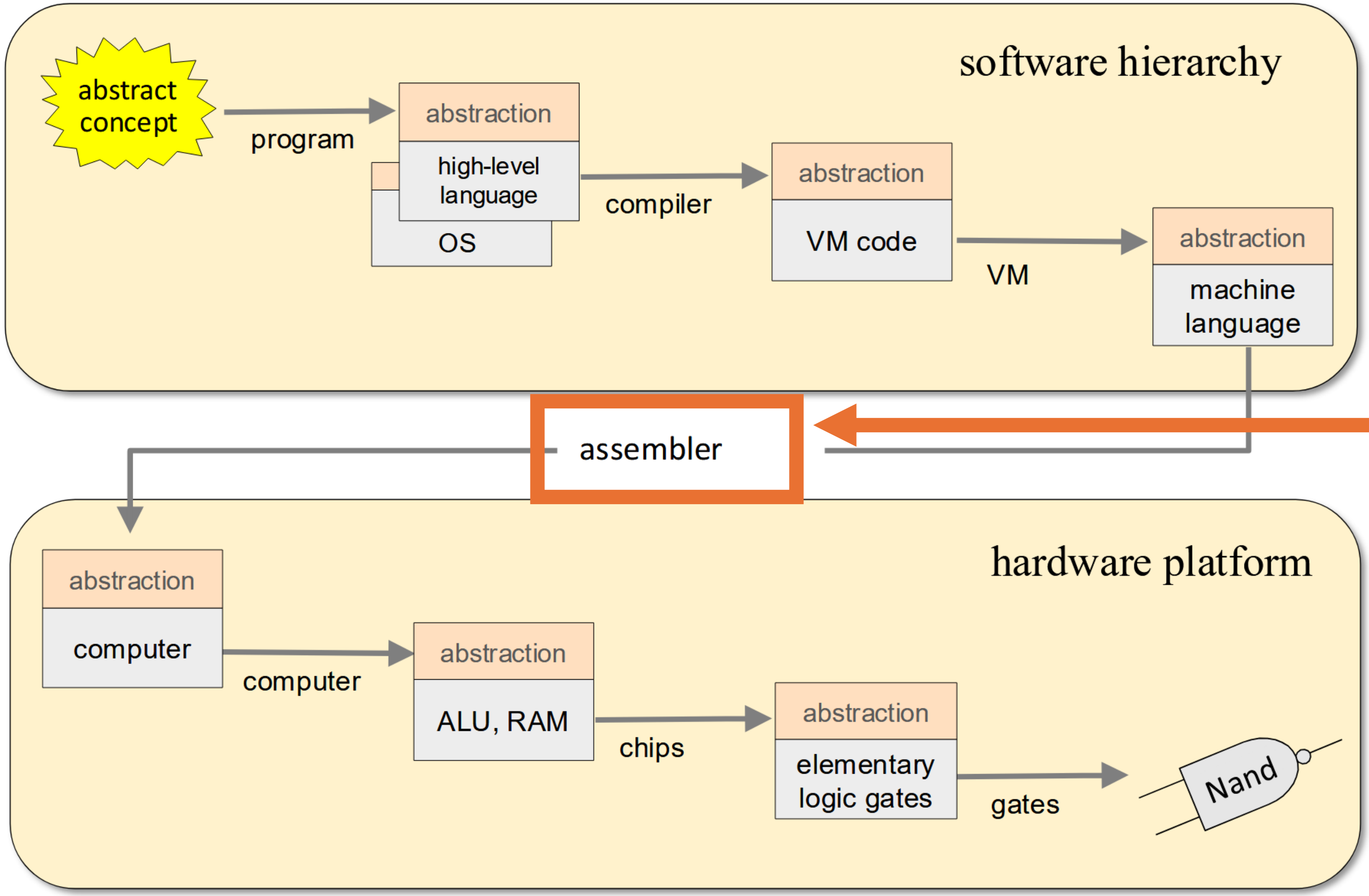
Today: finish talking about these components

The big picture: Hack



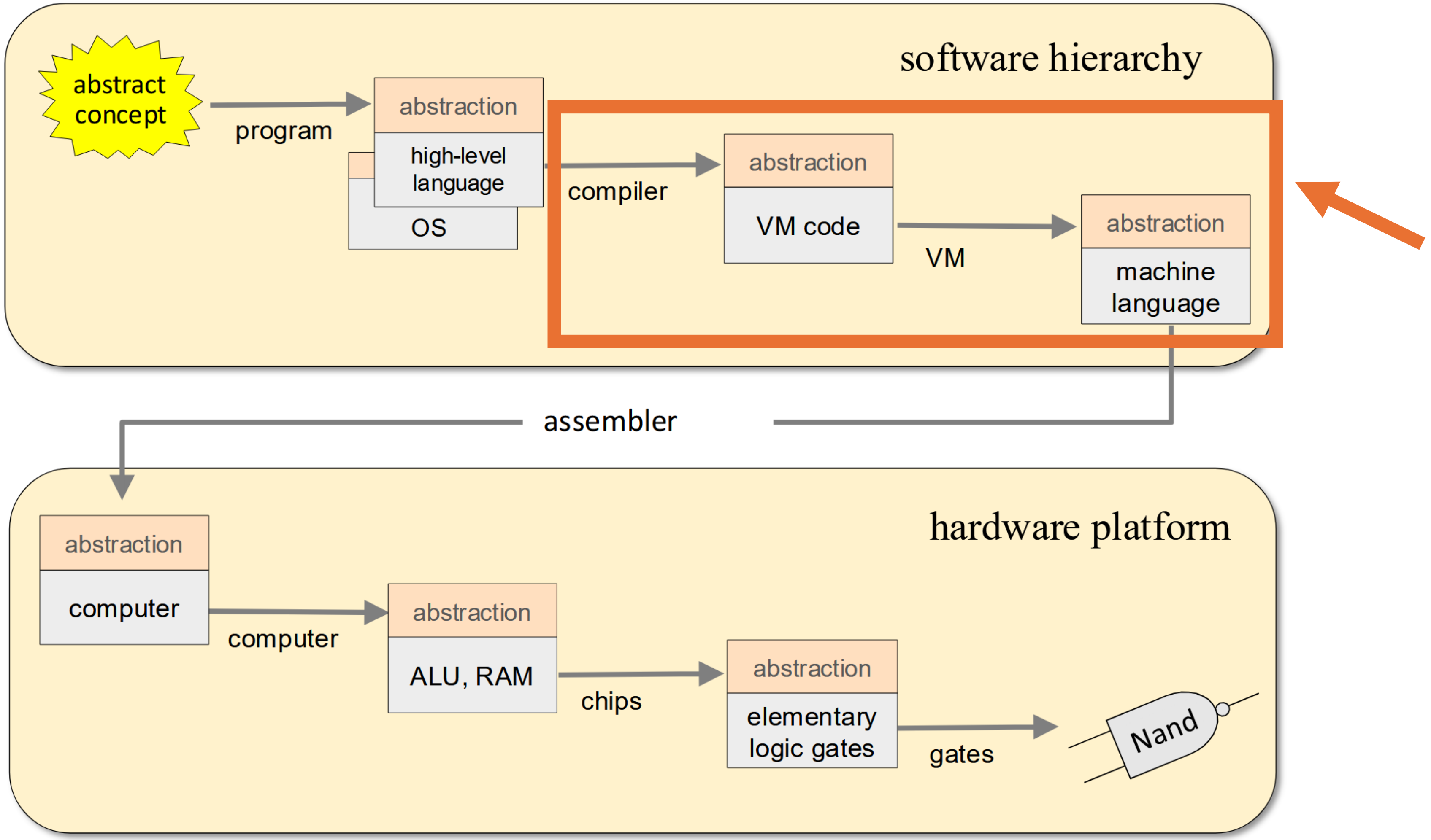
Next: How our hardware components fit together and how we can program it

The big picture: Hack



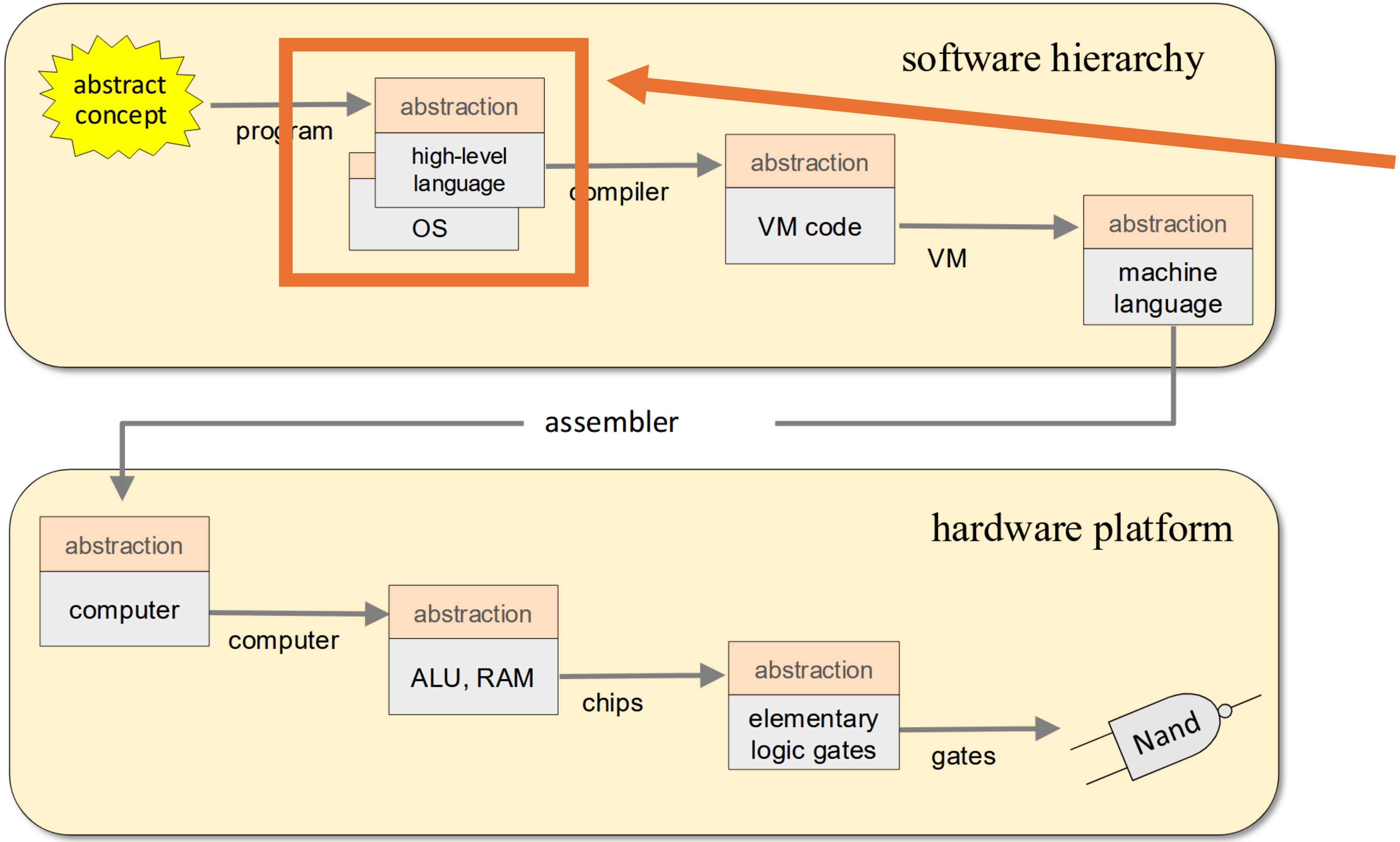
Next: Human readable machine instructions

The big picture: Hack



Compiling high-level code (like C or JAVA) into assembly language

The big picture: Hack



(Maybe) Implement OS-level system libraries for the keyboard, screen,memory, etc

Gates revisited: theory

Recall: Representations in binary

A binary variable can have two possible states (0 or 1)

Two binary variables can have 4 possible states (00, 01, 10, 11)

Three binary variables can have _____ possible states

N binary variables can have have ______ possible states

Boolean functions

A **Boolean function** operates on Boolean variables and returns a Boolean variable

examples: AND, OR, NOT, NAND, etc

Notation:

$f(x, y)$ → AND(x, y)

x *f* y → x AND y

Example: How many Boolean functions exist over two binary variables?

A	B	Out
0	0	?
0	1	?
1	0	?
1	1	?

Example Boolean functions of two variables

Function	0	0	1	1
	0	1	0	1
Constant 0	0	0	0	0
x	0	0	1	1
Equivalence	1	0	0	1
Constant 1	1	1	1	1

Exercise: Consider the function “if A then B”

A	B	f(A, B)
0	0	1
0	1	1
1	0	0
1	1	1

Construct a composite gate that implements this function

Example: How many Boolean functions exist over N binary variables?

We can represent a lot of functions using just binary variables….

Fun fact: Any Boolean function can be realized using only Nand

Nand(x, y)

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

And(x, y)

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

Or(x, y)

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

Not(x)

A	Out
0	1
1	0

Not(x) =

And(x, y) =

Or(x, y) =

Fun fact: Any Boolean function can be realized using only Nand

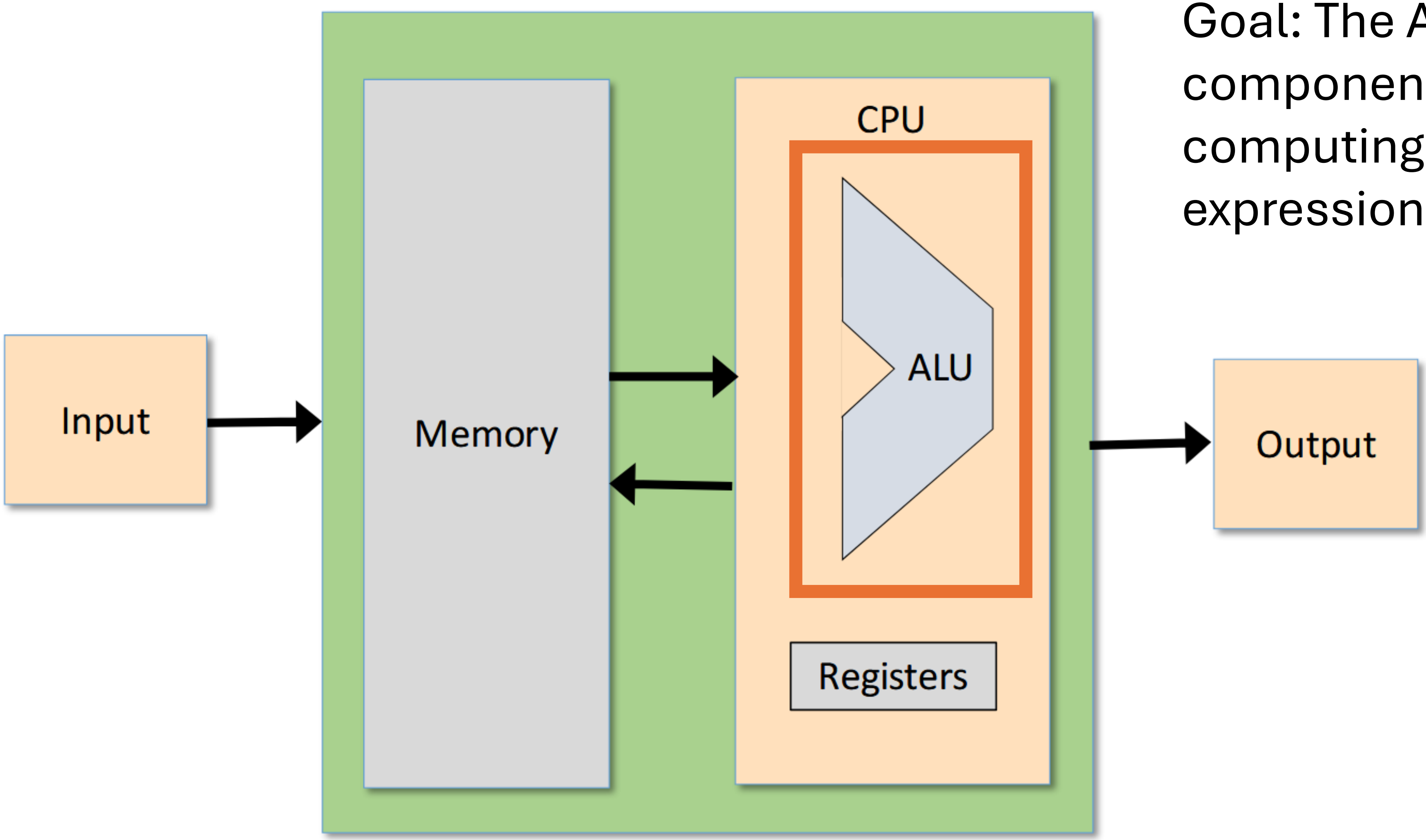
Proof:

Any Boolean function can be expressed as a truth table.

Any truth table can be expressed as a Boolean function using only Not, And, and Or (using Disjunctive Normal Form)

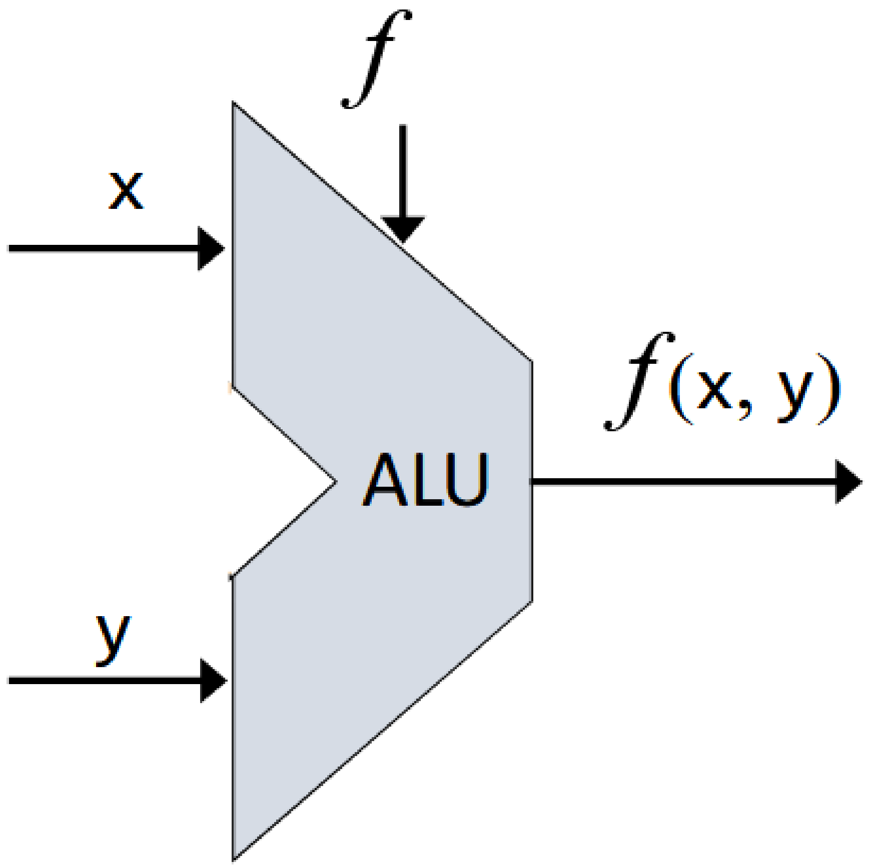
Note that And, Not, and Or can be defined using Nand

Arithmetic Logic Unit (ALU)



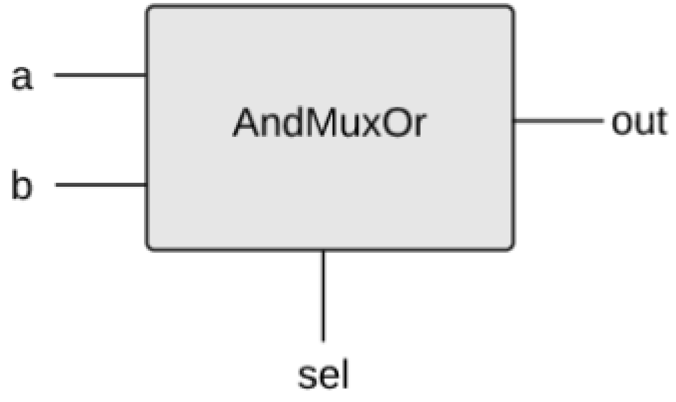
Goal: The ALU is a component for computing expressions

ALU



Idea: Computes a given function on two n-bit input values, and outputs an n-bit value

Exercise/Review: Using Mux logic to build a programmable gate




```
if (sel == 0)
    out = a And b
else
    out = a Or b
```

Suppose we want to build a gate that behaves like AND when sel = 0 and behaves like OR when sel = 1.

On the next slide:

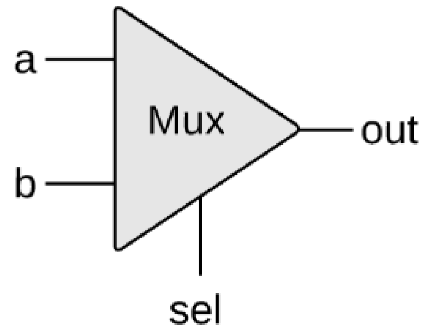
1. Draw the truth table
2. Draw the corresponding gate diagram

Exercise/Review: Using Mux logic to build a programmable gate

a	b	sel	output

Exercise/Review: Multiplexor

Express Mux in terms of and, or, not



if (sel == 0)
    out = a
else
    out = b

a	b	sel	out
0	0	0	**0**
0	1	0	**0**
1	0	0	**1**
1	1	0	**1**
0	0	1	**0**
0	1	1	**1**
1	0	1	**0**
1	1	1	**1**

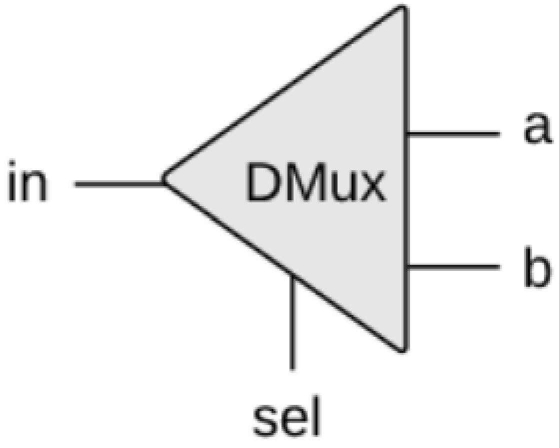
sel	out
0	**a**
1	**b**

abbreviated truth table

Exercise/Review: Demultiplexor

Idea: Route a single input value to one of several destinations

Express DMux in terms of and, or, not



if (`sel == 0`)

  {`a, b`} = {`in, 0`}

else

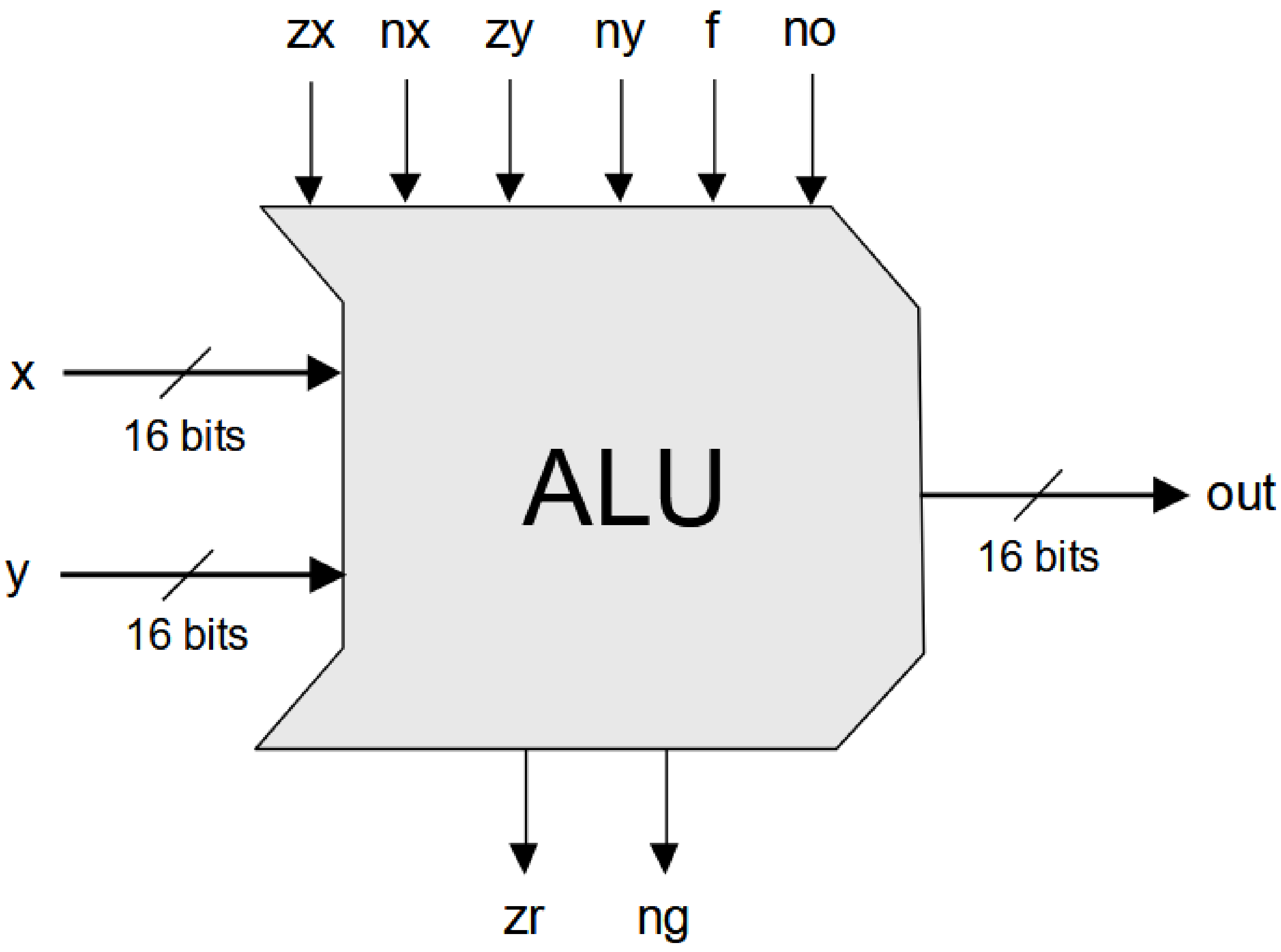
  {`a, b`} = {`0, in`}

in	sel	a	b
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

ALU implementation

Inputs:
- x: 16-bit value
- y: 16-bit value
- zx: if true, sets x to zero
- nx: if true, sets x to !x
- zy: if true, sets y to zero
- ny: if true, sets y to !y
- f: if true, computes out = x + y; else computes out = x AND y
- no: if true, sets out = !out

Outputs:
- out: 16-bit value
- zr: true when out == 0; false otherwise
- ng: true when out < 0; false otehrwise



ALU Example:

Derive settings that compute the Boolean function out(x,y) = x&y

zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out = x&y	if no then out=!out	out(x,y)
						x&y

ALU Example:

Derive settings that compute the Boolean function out(x,y) = x+y

zx	nx	zy	ny	f	no	out
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out = x&y	if no then out=!out	out(x,y)
						x+y

ALU Example:

Derive settings that compute the Boolean function out(x,y) = 0

zx	nx	zy	ny	f	no	out
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out = x&y	if no then out=!out	out(x,y)
						0

ALU Example:

Derive settings that compute the Boolean function out(x,y) = 1

zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out = x&y	if no then out=!out	out(x,y)
						1

ALU Example:

Verify that the following settings compute the Boolean function out(x,y) = x-1

zx	nx	zy	ny	f	no	out
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out = x&y	if no then out=!out	out(x,y)
0	1	1	1	1	1	x+1

ALU Example

Verify that the previous settings computes the Boolean function
out(x,y) = x+1

Let x = 0x6 and y = 0x9

How it works: Boolean algebra Insights

x + 1 = !(!x + 1111)

–x = !(x + 1111)

x – y = !(!x + y)

x = x & 1111

0 = x & 0000

and others…

Exercise: Outputs

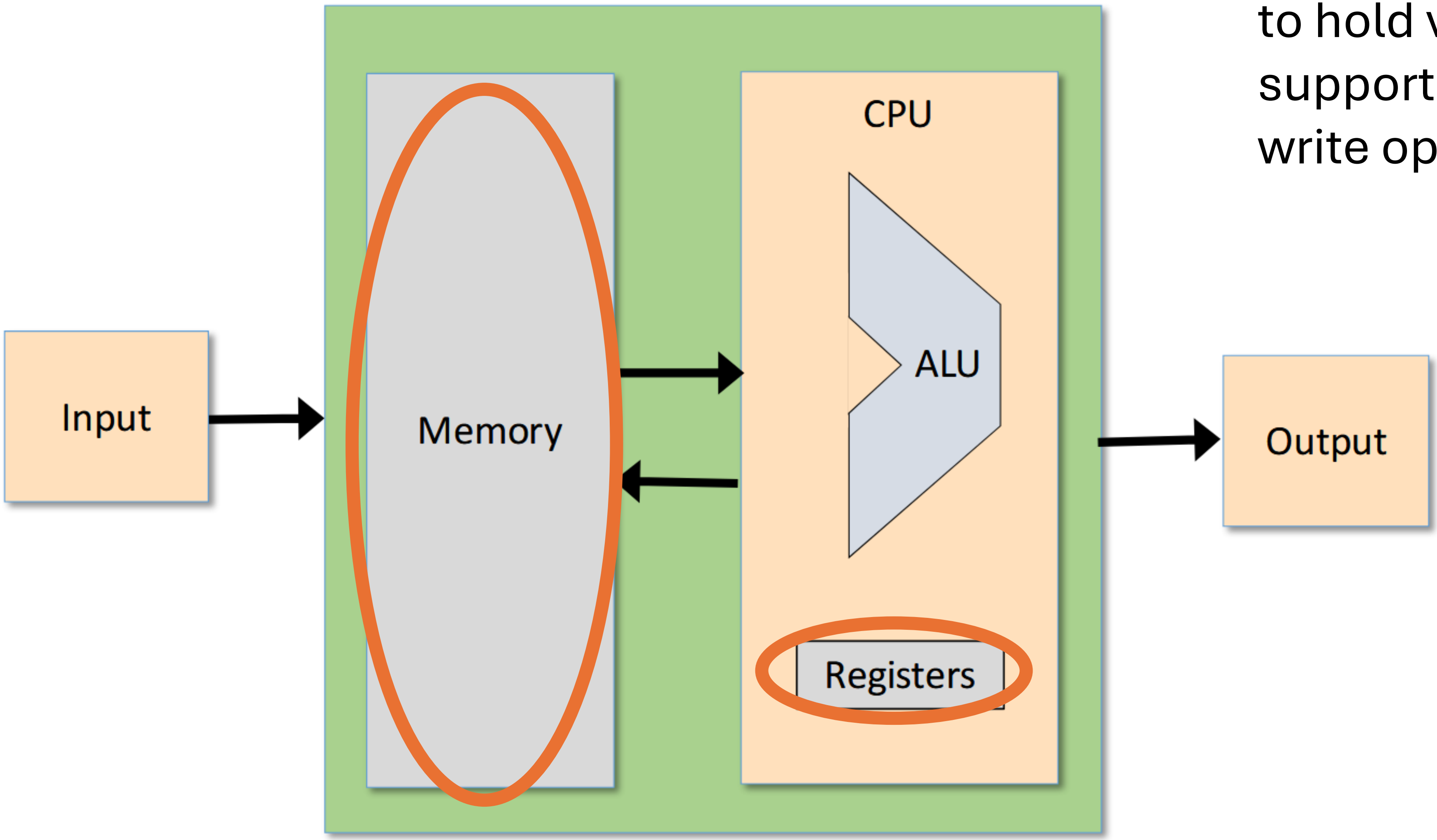
How can we test if the output is zero?

How can we test if the output is negative?

Hint: You can access single bits from a value, e.g. x[2]

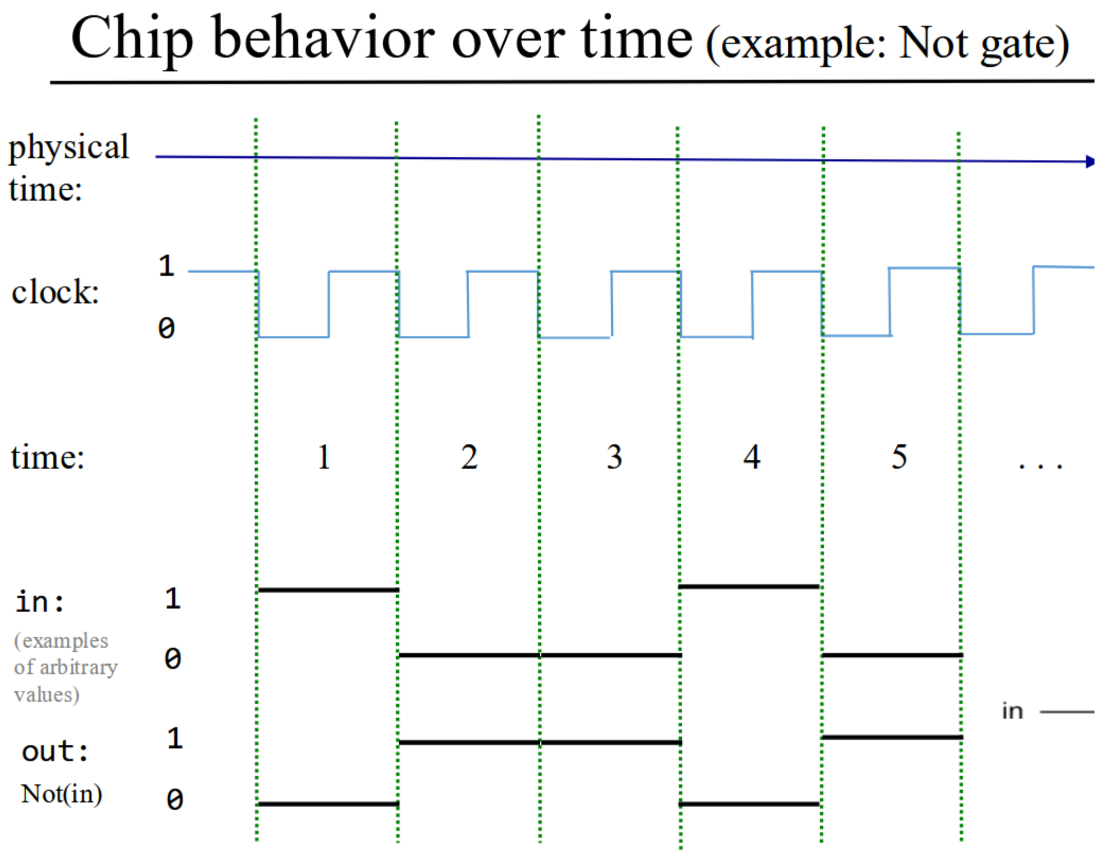
Exercise: Connect components to implement the ALU

Memory



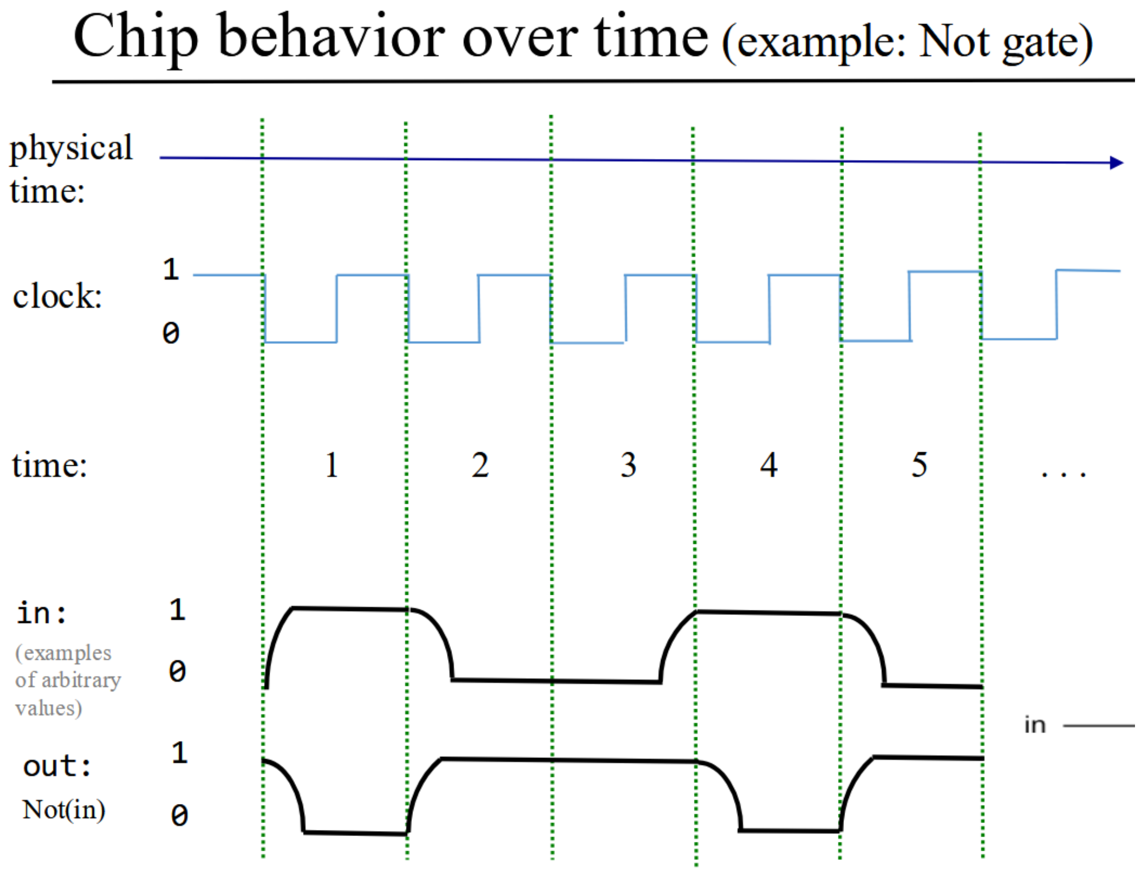
Goal: Memory needs to hold values and support read and write operations

Recall: Representing time in hardware



Desired / idealized behavior of the in and out signals:

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



Actual behavior of the in and out signals:

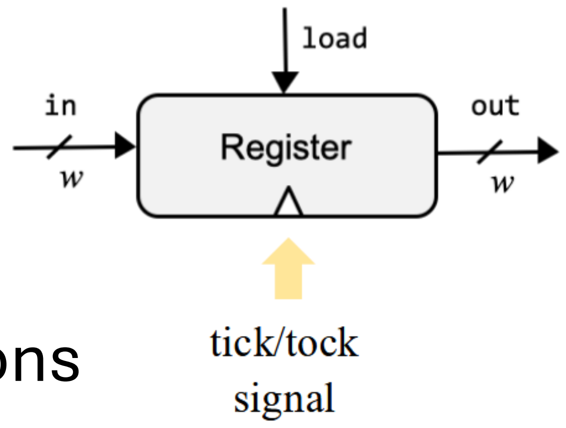
Influenced by physical time delays

Time and memory

Memory only works along side a concept of time

Need to create a mechanism that "knows" what its value was at the previous time step

Hardware needs time to settle into high/low values



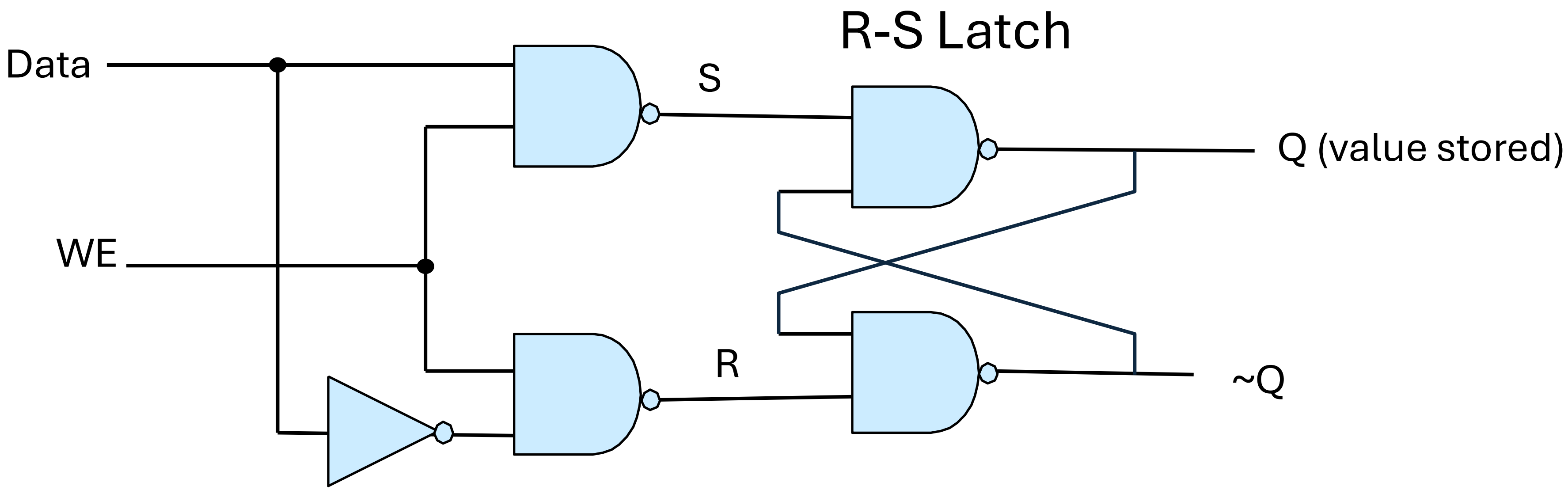
Solution: Clock time step is long enough to hide fluctuations

Only change state when each time cycle ends

Clock hardware is based on an oscillator

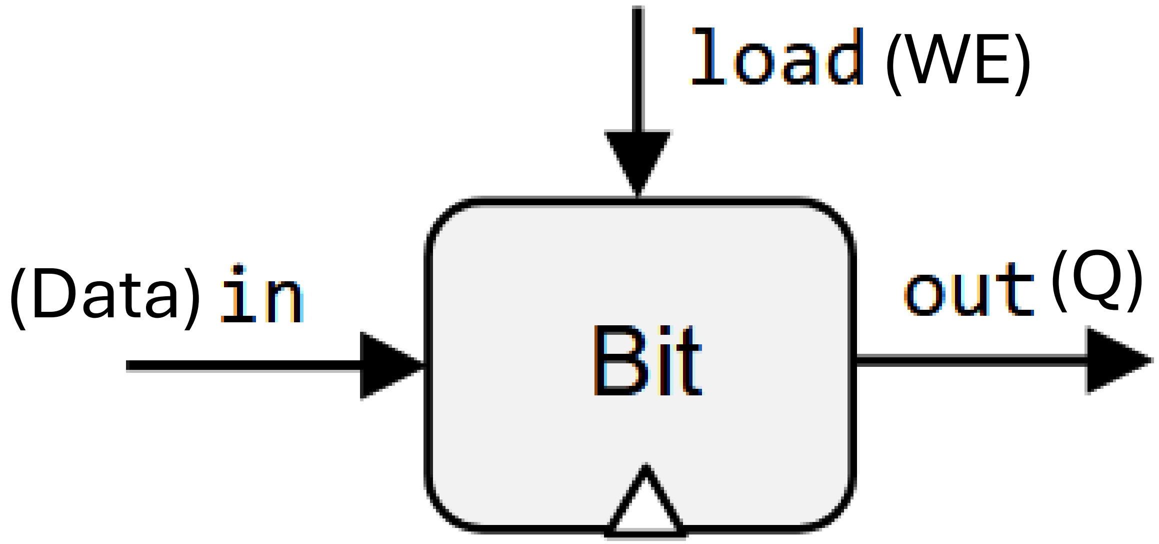
NOTE: In contrast, combinatorial gates react immediately to input

Recall: Gated R-S Latch



Data	WE	S	R	Q
0	0			
0	1			
1	0			
1	1			

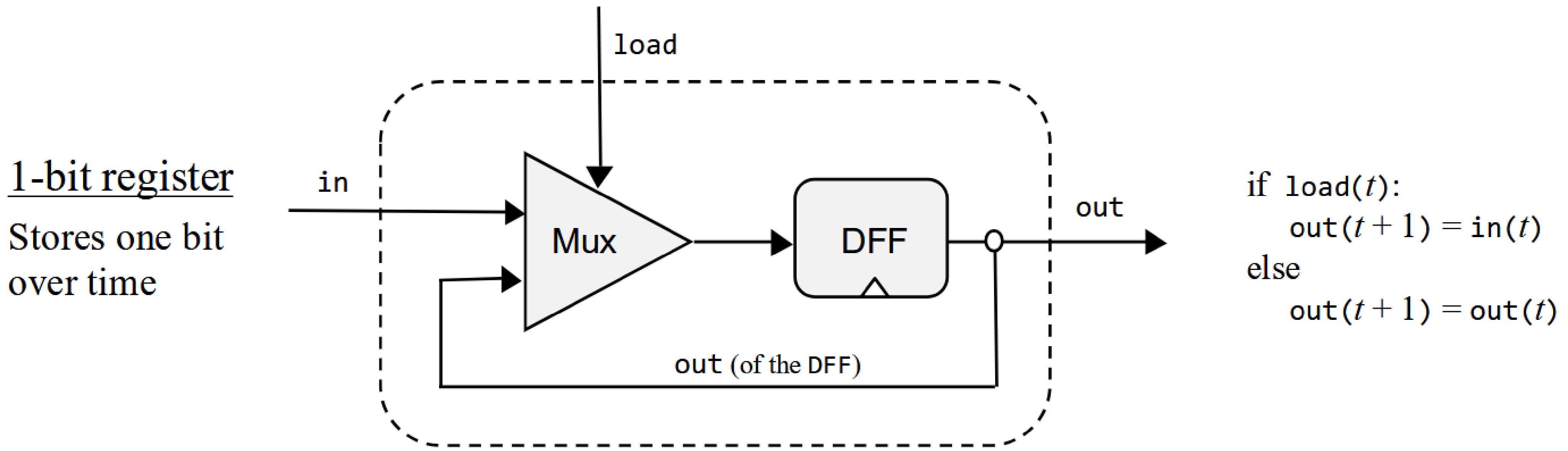
Data flip-flop (DFF): 1-bit register



1-bit register

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

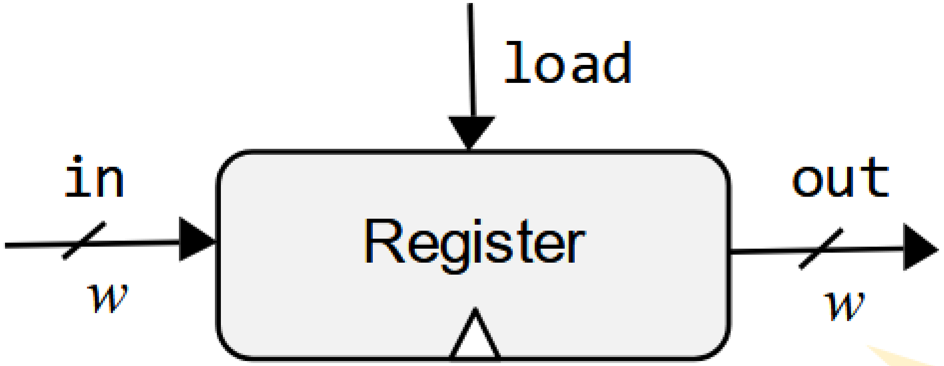
Data flip-flop (simplified implementation)



To read, simply get the value of out

To write, set in = newValue and load = true

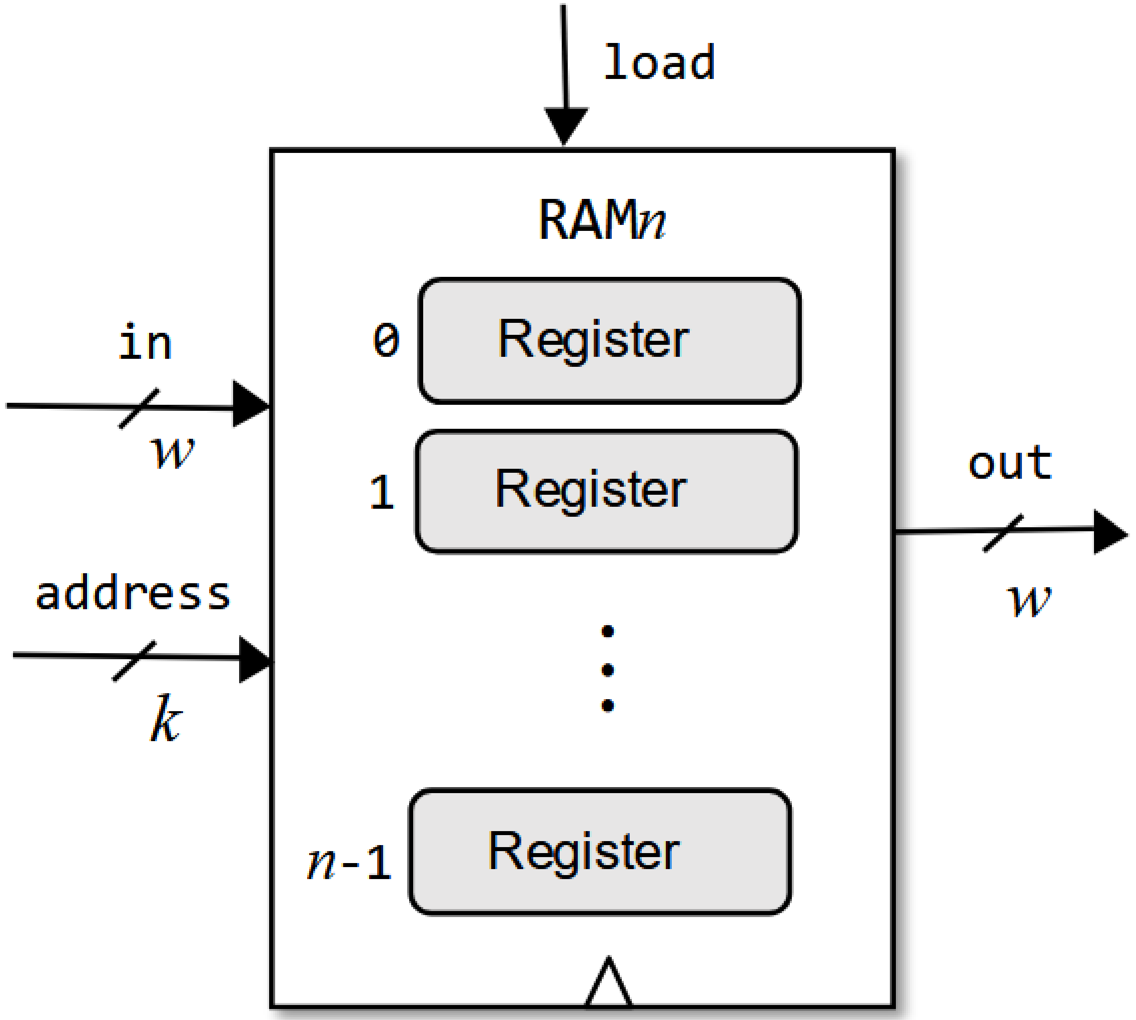
Register



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

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

Random Access Memory (RAM)



Question: If RAM has N registers, how many bits do we need for the address?

Example: Suppose RAM has 8 registers.

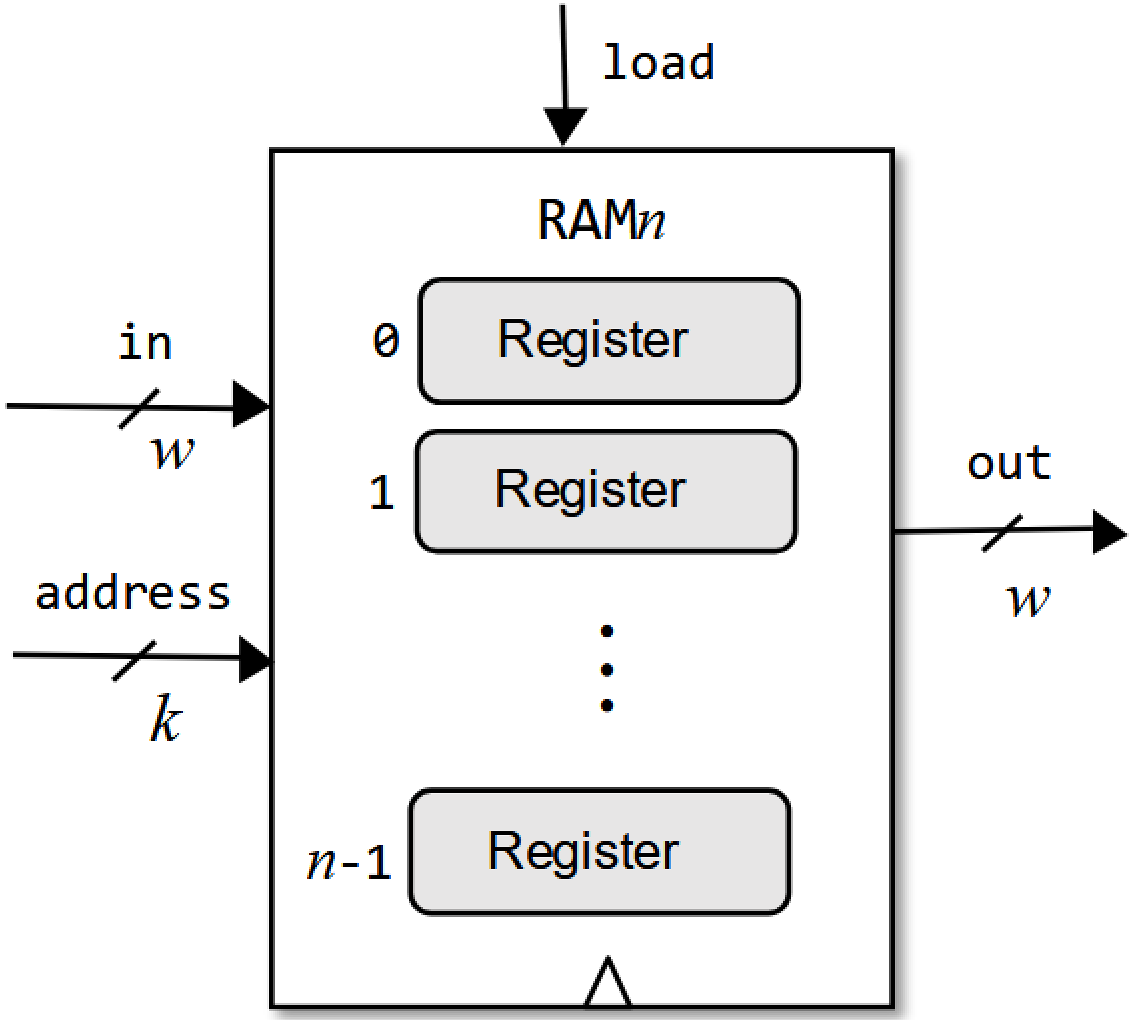
RAM Behavior

If load == 0, the RAM maintains its state

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

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

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

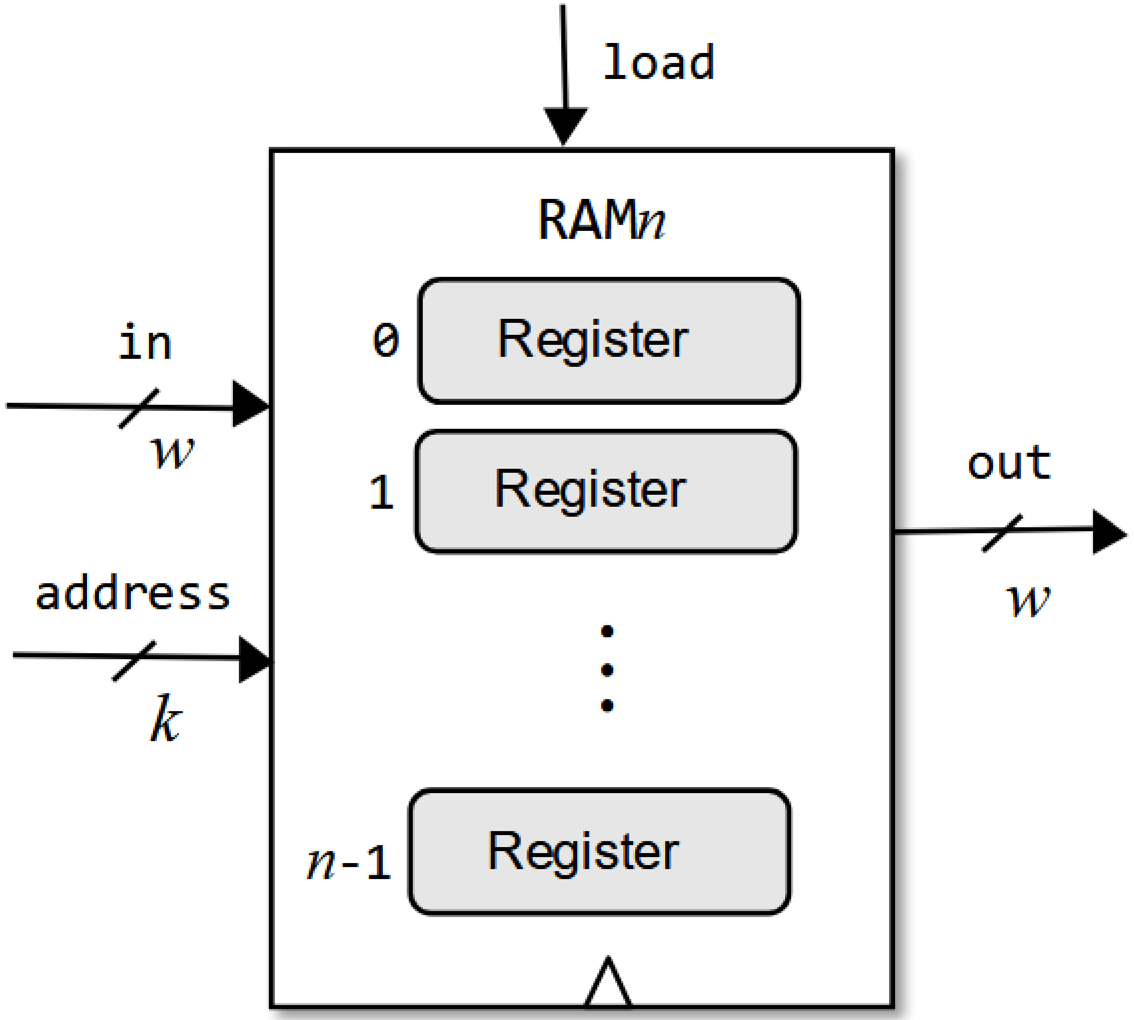


RAM Usage

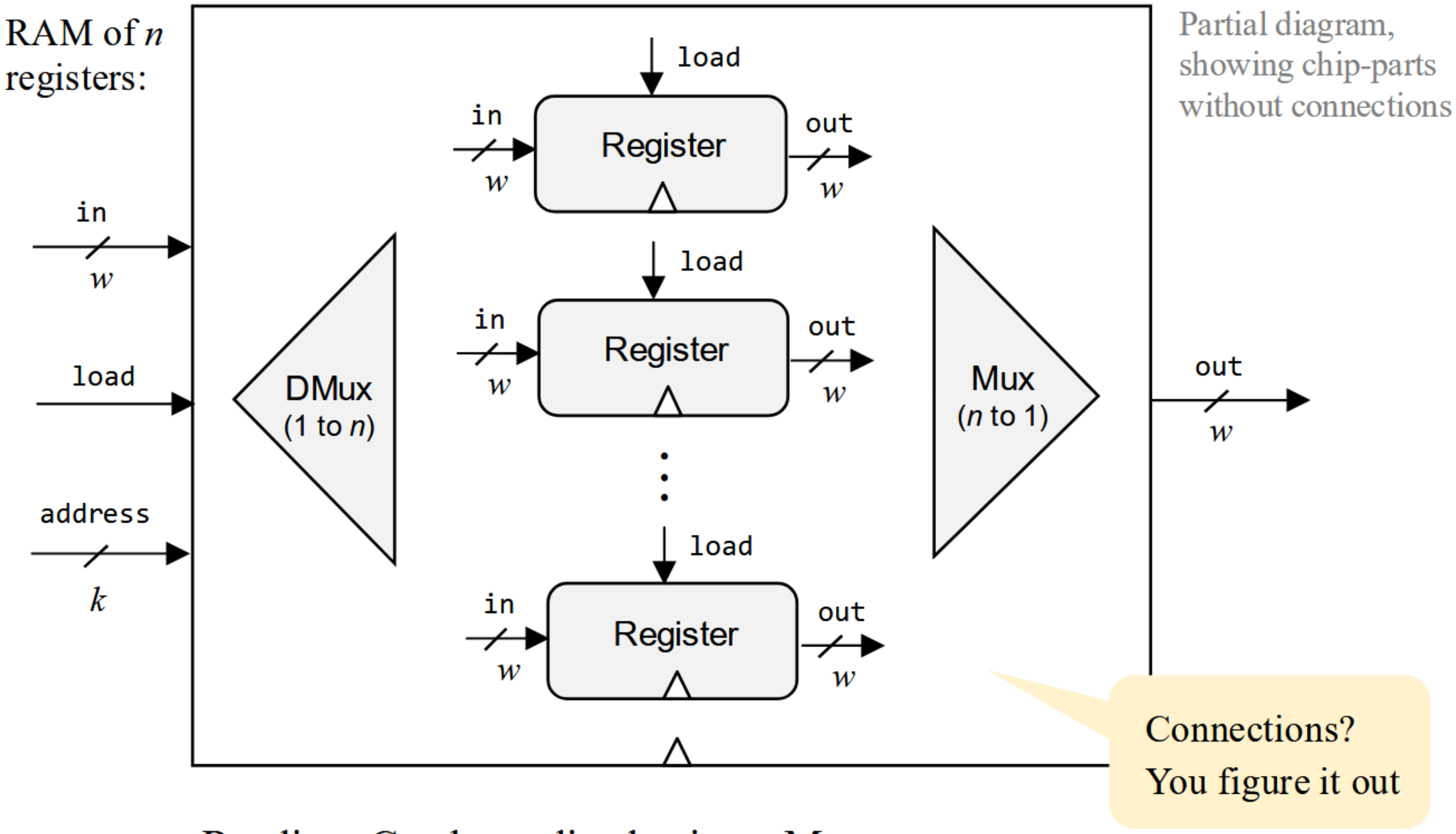
To read:
set address = i
probe out
(out = RAM[i] always)

To write:
set address = i
set in = newValue
set load = true
(R[i] = newValue)

Any random register can be read/written in one time cycle (thus the name RAM)



RAM implementation



Reading: Can be realized using a Mux

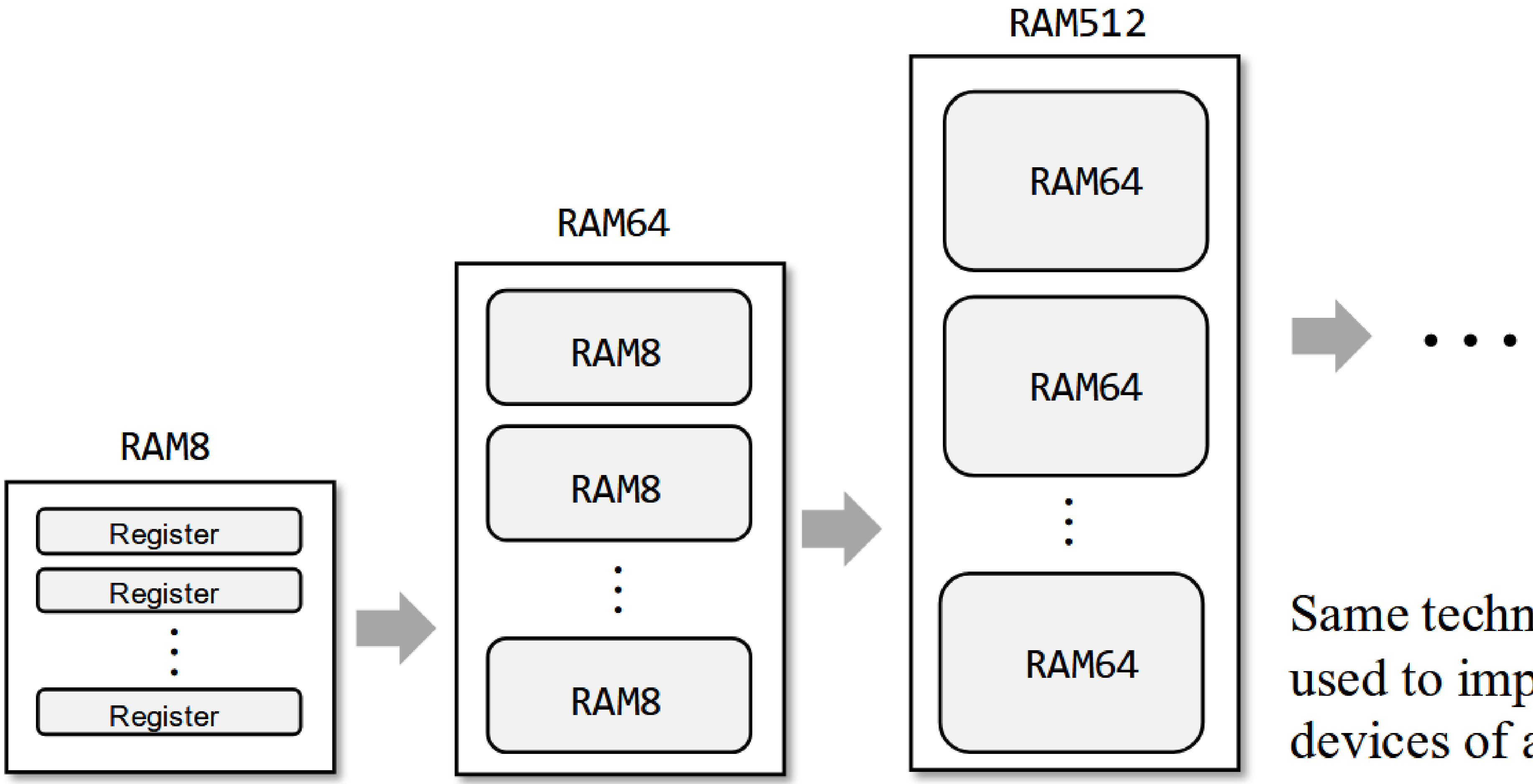
Writing: Can be realized using a DMux

Direct access works like so:

The storage behavior uses sequential logic

But, the addressing logic uses Mux / DMux chips, which are combinational.

Larger blocks of RAM

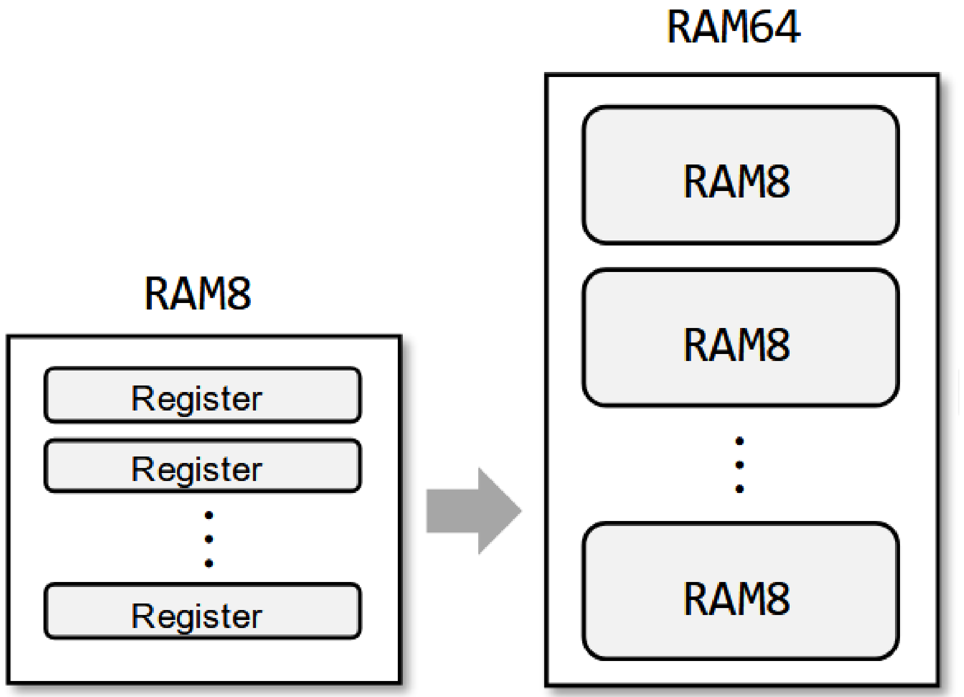


Same technique can be used to implement RAM devices of any size

Example: RAM addressing

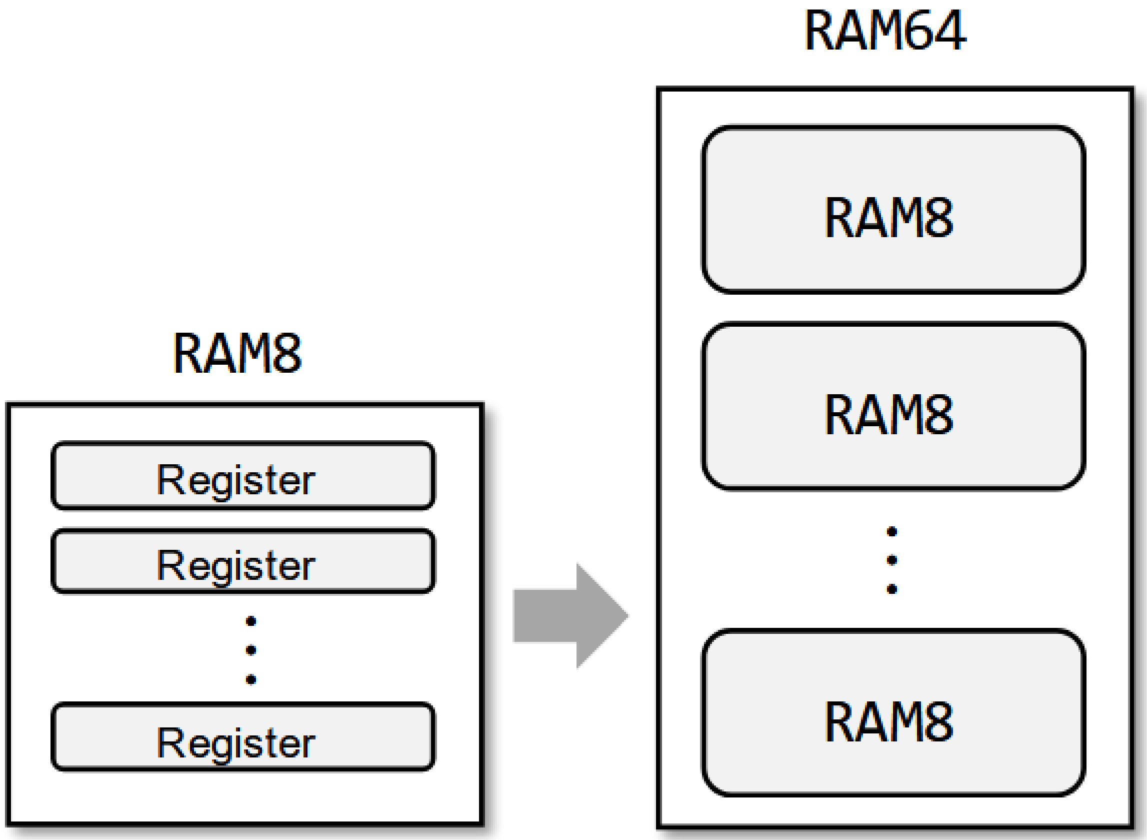
Question: If RAM has N registers, how many bits do we need for the address?

Example: Suppose RAM has 8 registers.



Example: RAM addressing

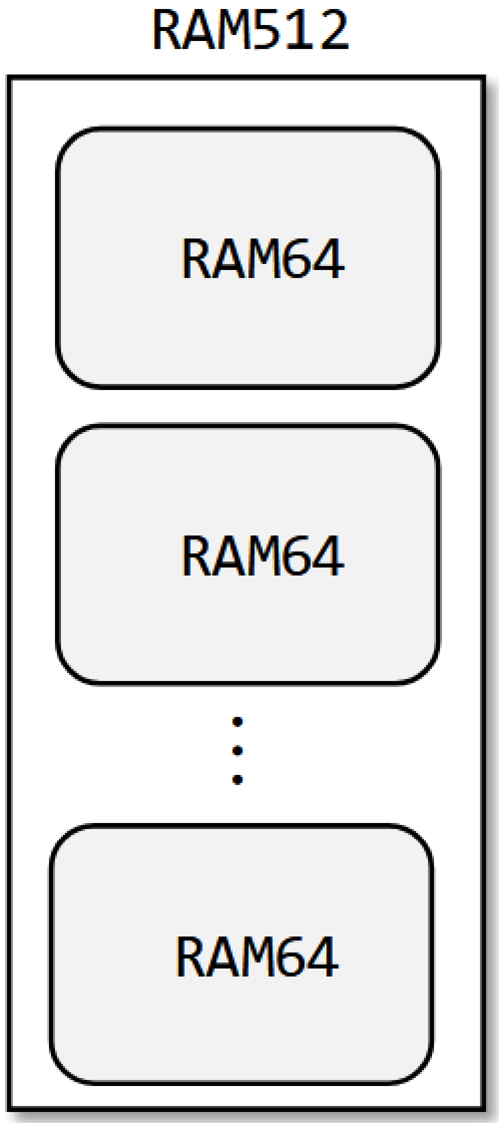
Suppose we are working with RAM64. How many bits do we need?
What are the base addresses of each subblock?



Example: RAM addressing

Suppose we are working with RAM512. How many bits do we need?

What are the base addresses of each subblock?

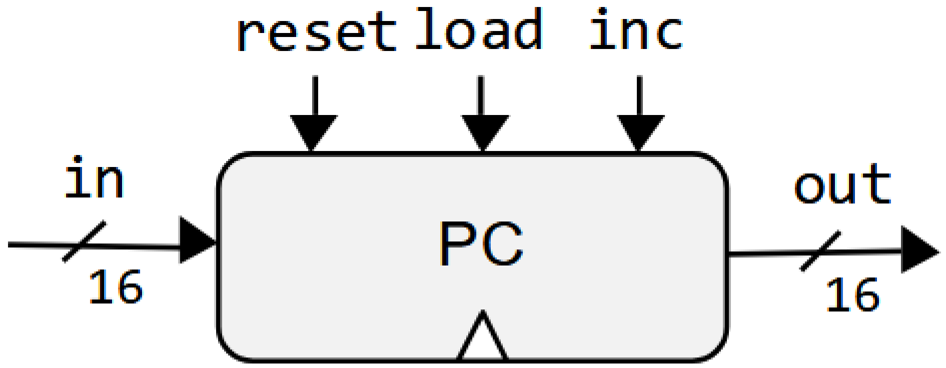


RAM chips needed for Hack

NOTE: In our simulator, all addresses are 16 bits; however, only a subset of bits are used for each memory layer

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

Counter



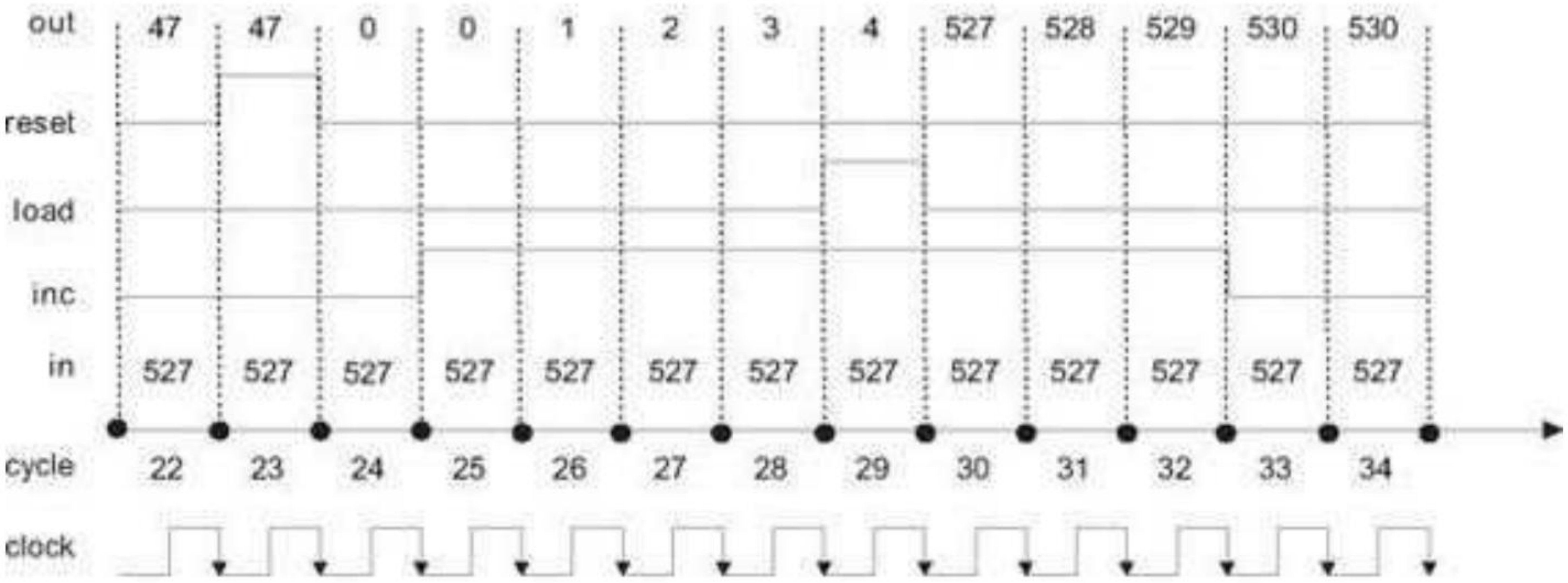
if reset(t): out($t+1$) = 0
else if load(t): out($t+1$) = in(t)
else if inc(t): out($t+1$) = out(t) + 1
else out($t+1$) = out(t)

Implementation is based on a register (sub-class)

Additional gates (Mux16, Inc16) are added to the DFF to support the different features: reset, load, inc

Will be used for storing the address of the next instruction

Counter Example



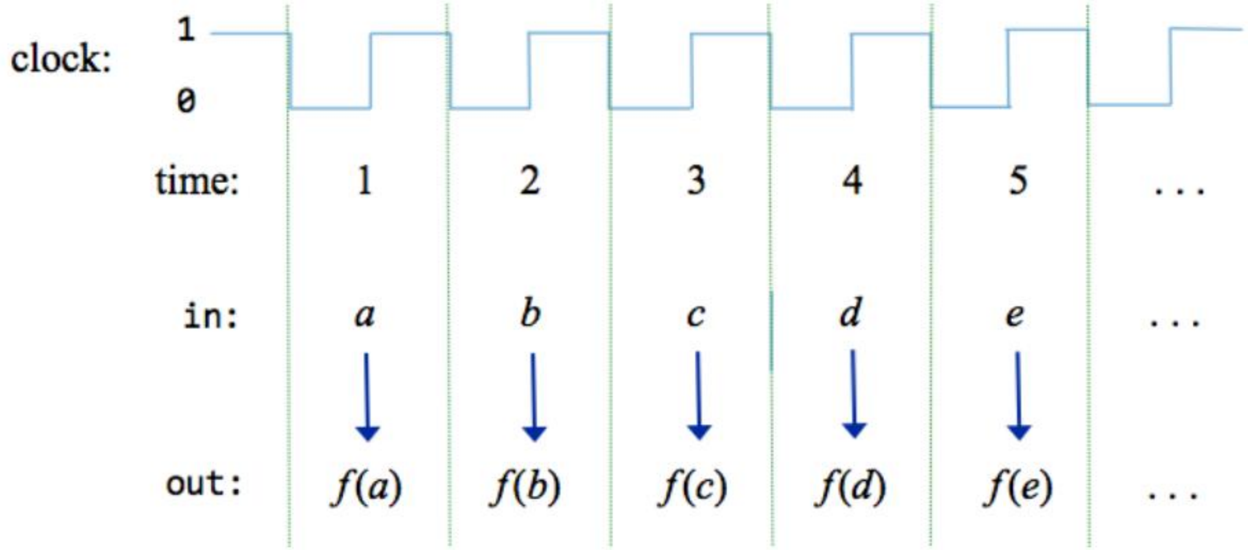
Recap: Sequential vs. Combinatorial Logic

Combinational logic

The output depends on current inputs only

The clock is used to stabilize outputs

Used for building chips that do calculations



Sequential logic

The output depends on:

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

Used for building memory chips (registers, RAM)