Lecture 4 – Sequential Logic and ALU

25/10/2024
Dr. Libin HONG

Outline

- Sequential Logic Circuit

- Sequential Chips

- Arithmetic Logical Unit

Learning Outcome

- To be able to understand sequential logic circuit
- To be able to understand the key concepts of sequential logic chip
- To be able to implement simple sequential logic chip in HDL
- To be able to understand the general concepts of ALU

Introduction

- All the chips we’ve seen so far were combinational
- Combinational chips compute functions that **depend solely on combinations of their input values**
  - The chip’s inputs were just “sitting there” fixed and unchanging
  - The chip’s output was a pure function of the current inputs, and **did not depend on anything that happened previously**
  - The output was computed “instantaneously”
- This style of gate logic is sometimes called:
  - Time independent logic
  - Combinational logic
  - Memory less

Sequential Logic Circuits

- So far we ignored the issue of time
- In order to maintain states, we need to be able to store and recall values
- Sequential Logic Circuits
  - Output depends **not only on the present value** of its input signals but **on the sequence of past inputs**, the input history as well

Memory Elements

- Memory elements are needed to preserve data over time
- Memory elements are built from sequential chips
- Implementation of memory elements involves synchronization, clocking and feedback loops
- Most of this complexity can be embedded in the operating logic of very low-level sequential gates called **flip-flops**
- Using flips-flops as elementary building blocks – we will build all the memory devices employed by modern day computers

Time

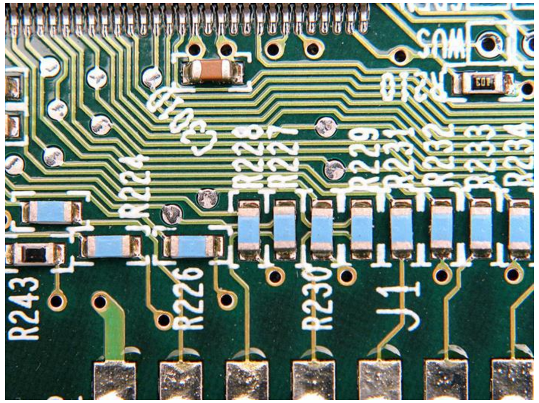
- The hardware must support maintaining “state”

```
x = 17
```

- The hardware must support computations over time

```
for i = 0 … 99:
    sum = sum + a[i]
```

- The hardware must handle the physical time delays associated with calculating and moving data from one chip to another
  - Can not ask computer to do something faster than its physics

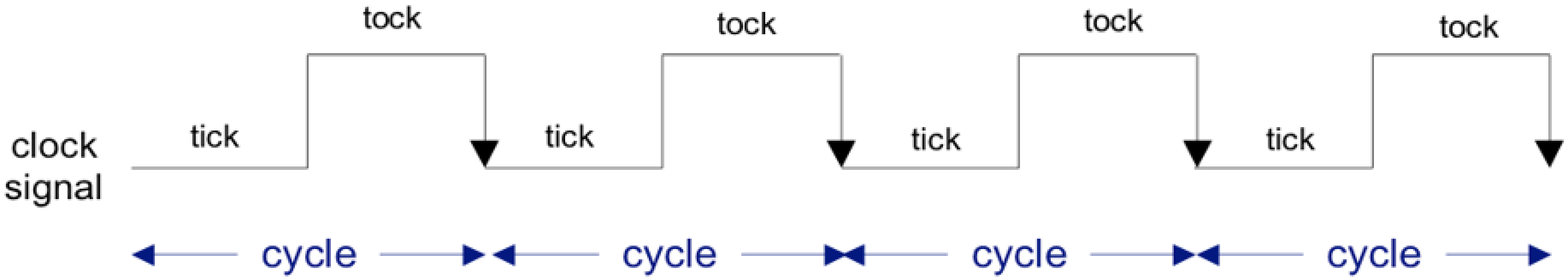


Clock

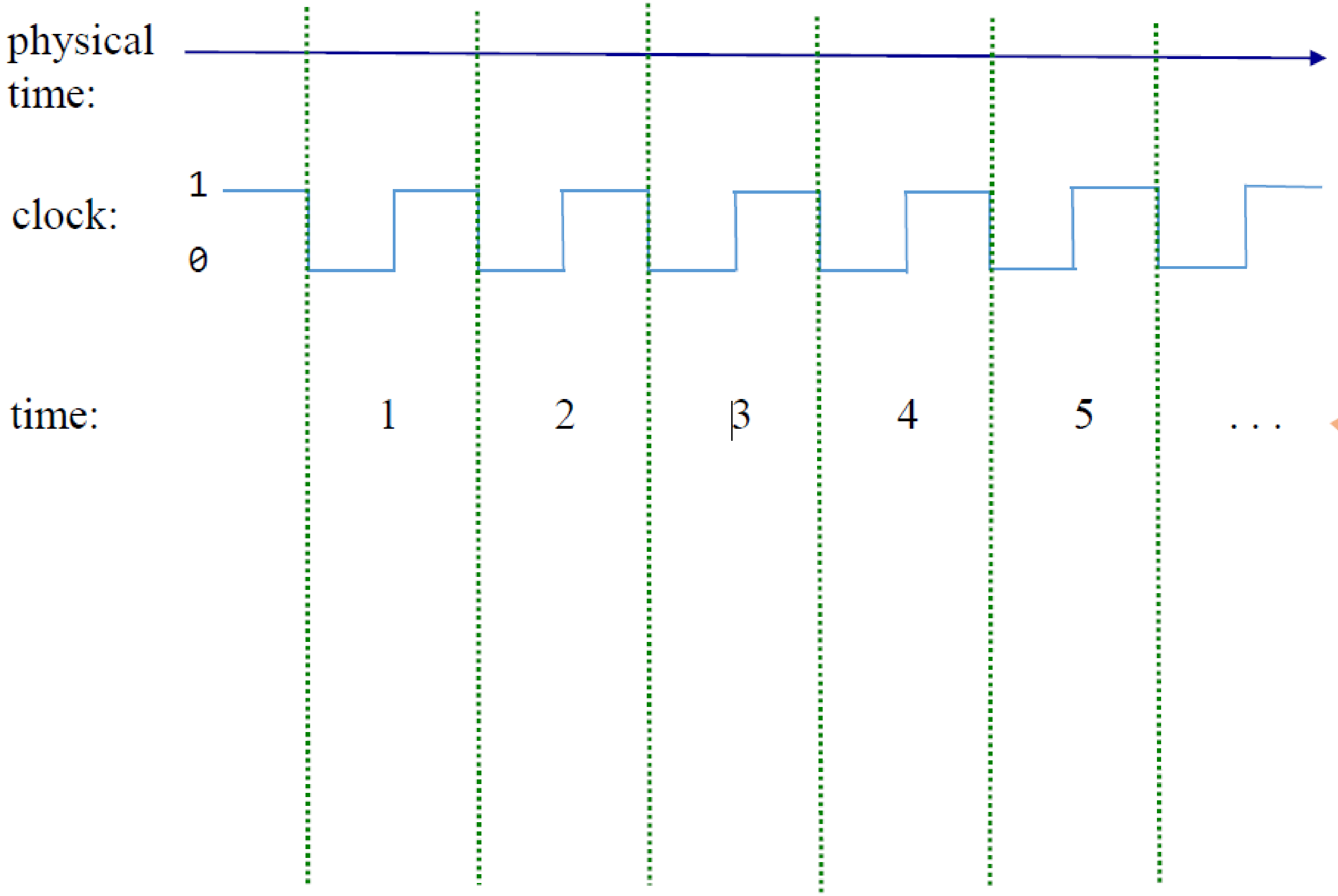
- Almost all computers are constructed using a clock that determines when events take place in hardware
- The clock delivers a **continuous train of alternating signals**
- The hardware implementation is based on an oscillator that alternates between the beginning phases labelled:
  - 0-1, low-high, tick-tock
- The elapsed time **between the beginning of a tick and the end of a subsequent tock is called a cycle**
- A clock phases tick and tock is represented by a binary signal (0 and 1)

Clock

- In our jargon, a clock cycle = tick-phase(low), followed by a tock-phase(high)
- In real hardware, the clock is implemented by an oscillator



Physical time/Clock time



Arrow of time:

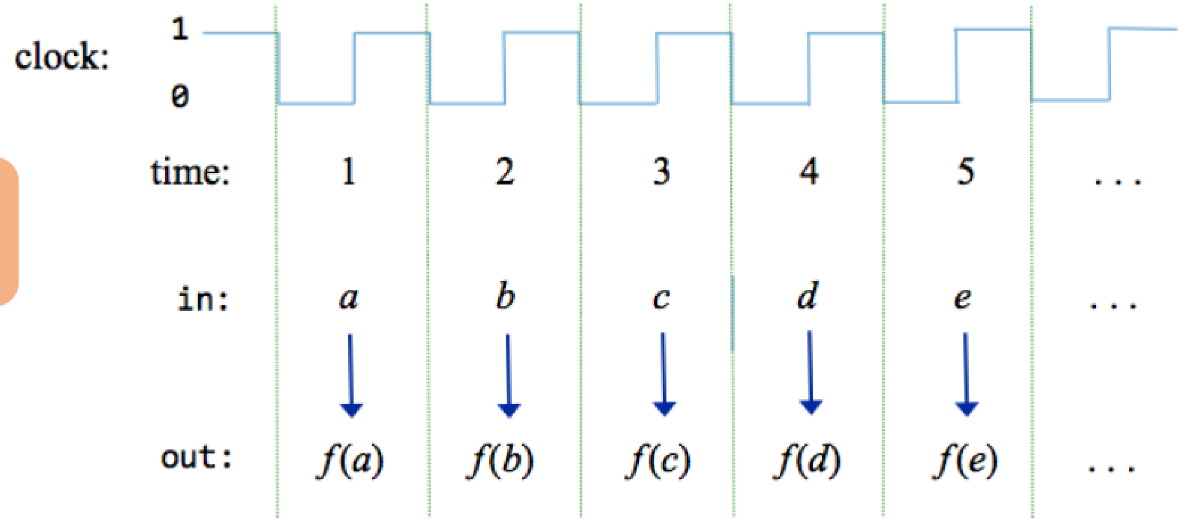
Continuous

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

Combinational logic / Sequential logic

Combinational

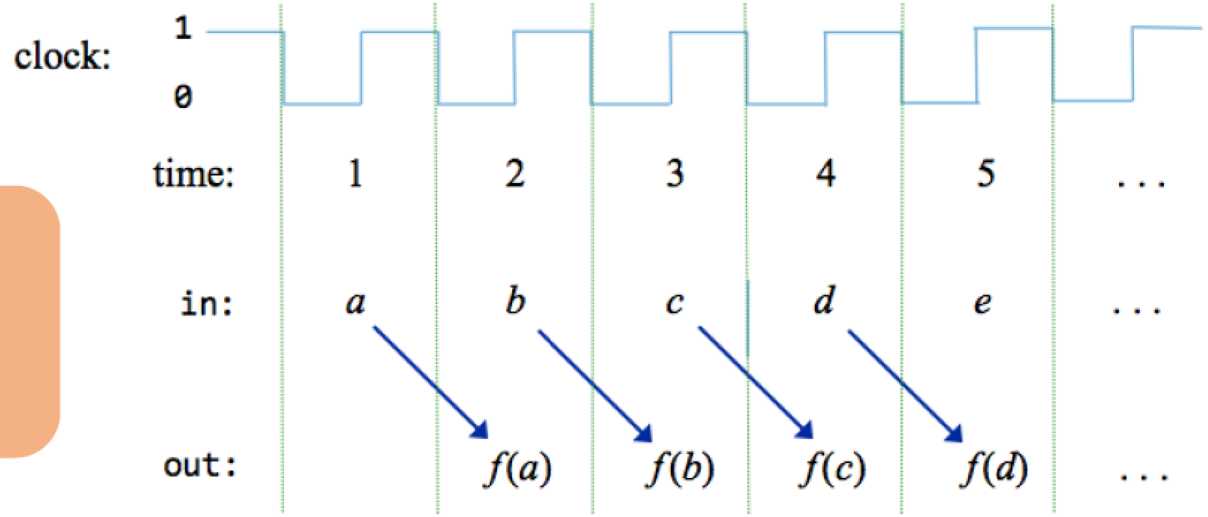
The output is a pure function of the present input only



Sequential logic:

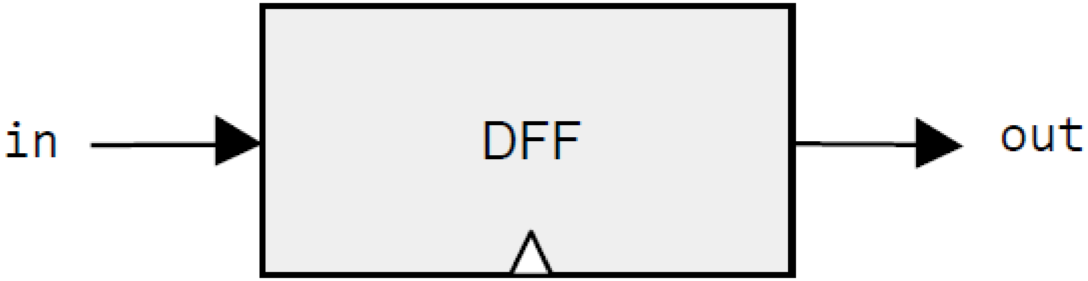
The output depends on:

- the present input (optionally)
- the history of the input (creates a memory effect).



Flip Flops

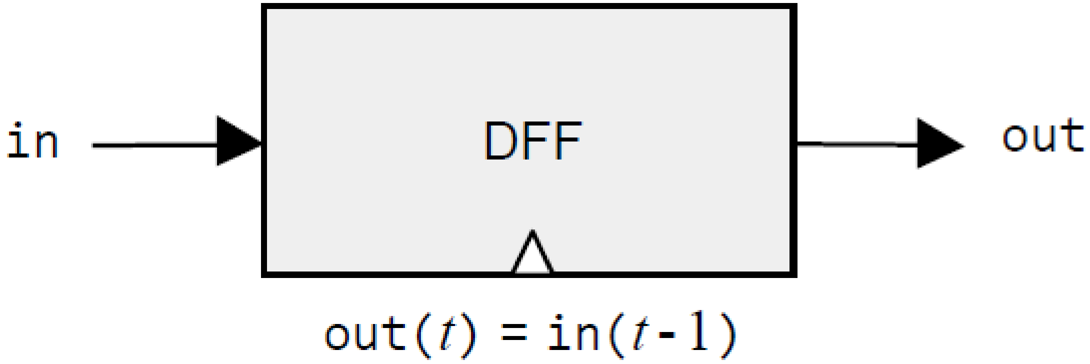
- The flip flop is the most elementary sequential element in the computer
- Data Flip Flop (DFF): the simplest **state keeping** gate (built-in)



out(t) = in(t-1)

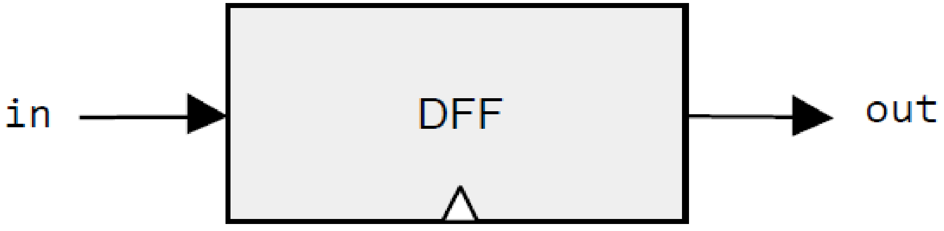
- Contains a **single** bit **input** and a **single** bit **output**

Flip Flops

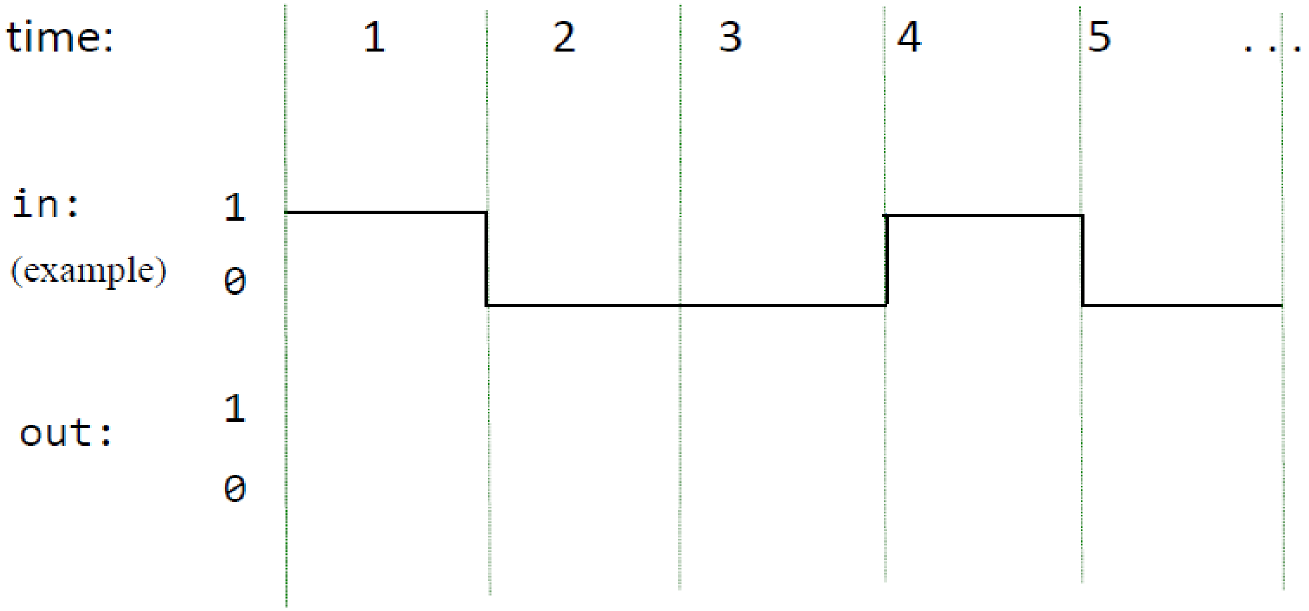


- The gate outputs its previous input: out(t)= in$(t$-1$)$
- Implementation: a gate that can flip between two stable states:
  - Remembering 0/Remembering 1
  - Also can be made from looping NAND gates

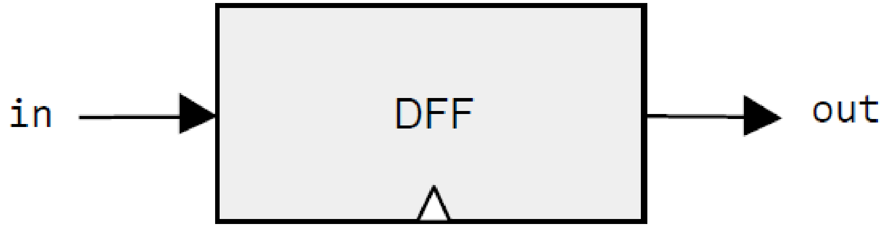
Flip-Flop



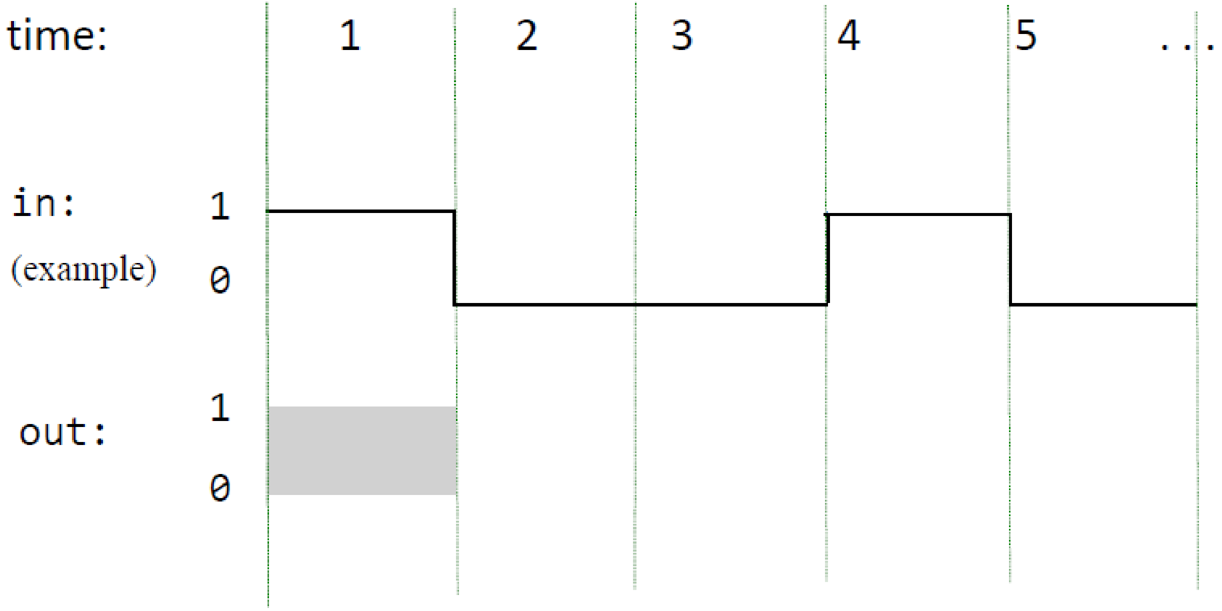
out(t) = in(t-1)



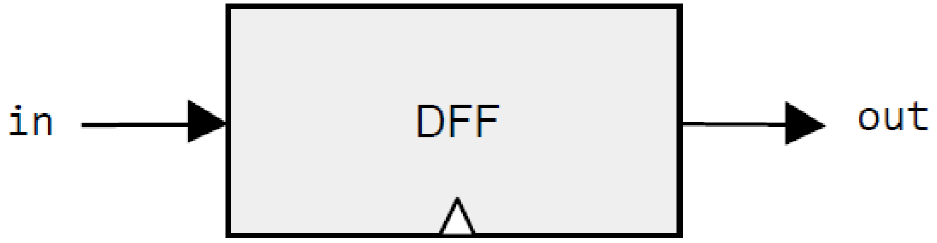
Flip-Flop



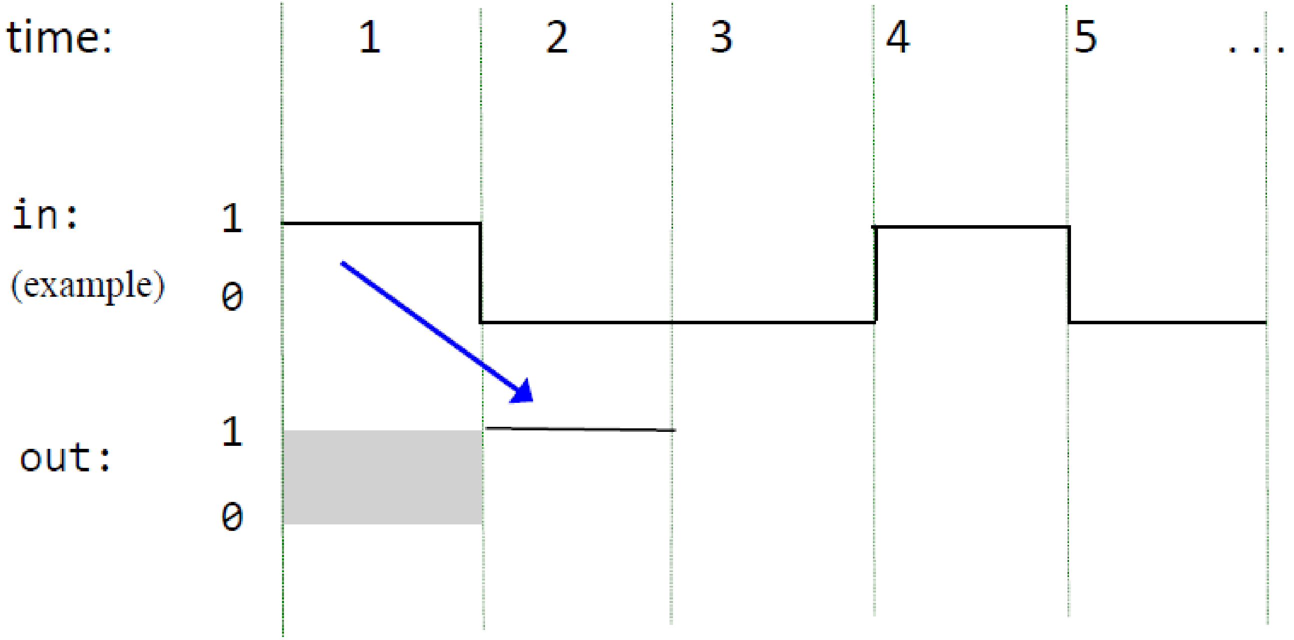
out(t) = in(t-1)



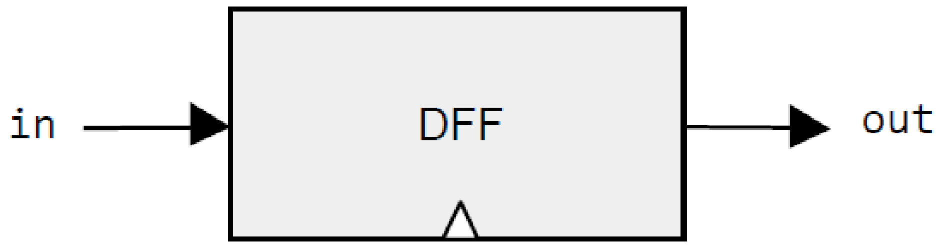
Flip-Flop



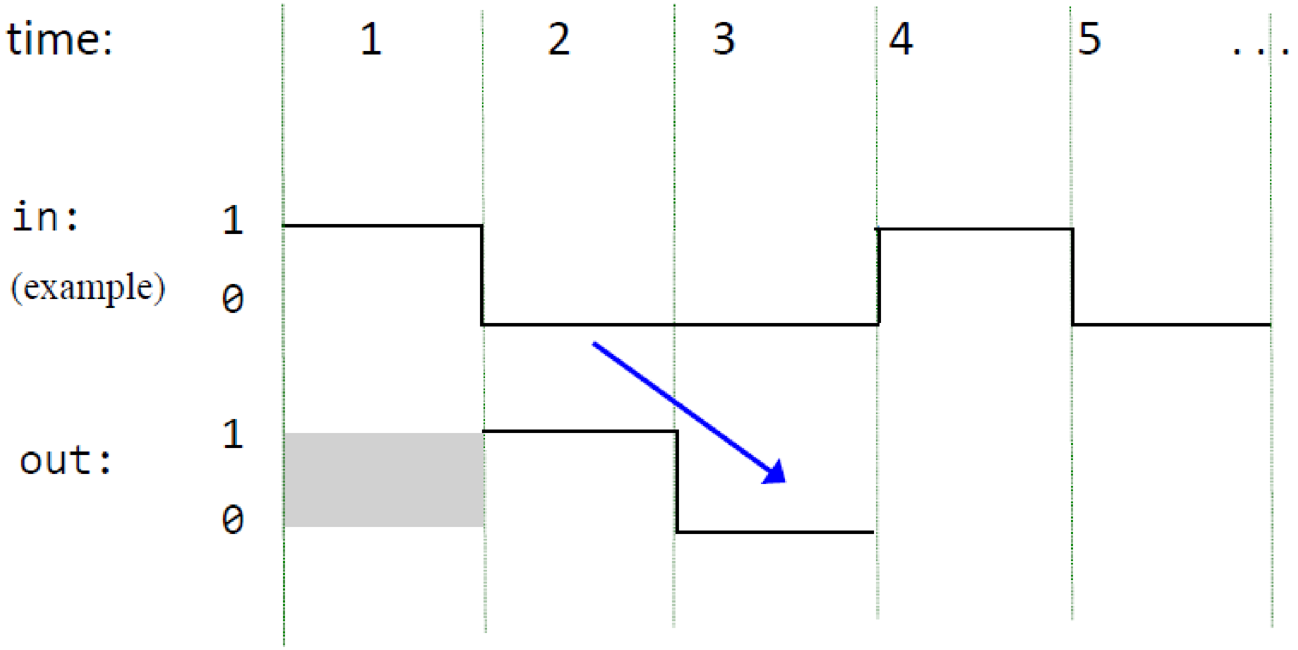
out(t) = in(t-1)



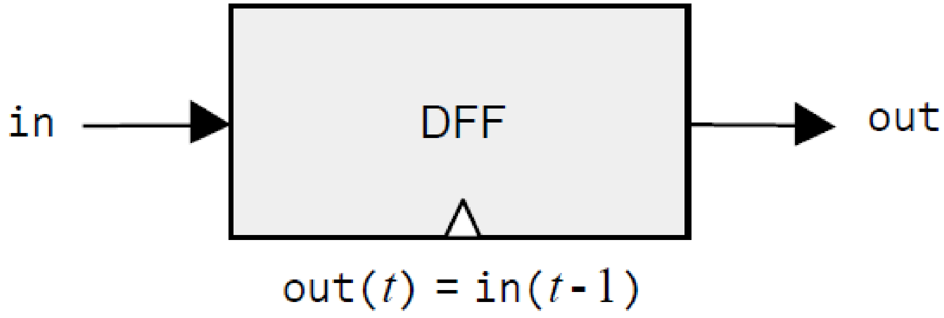
Flip-Flop

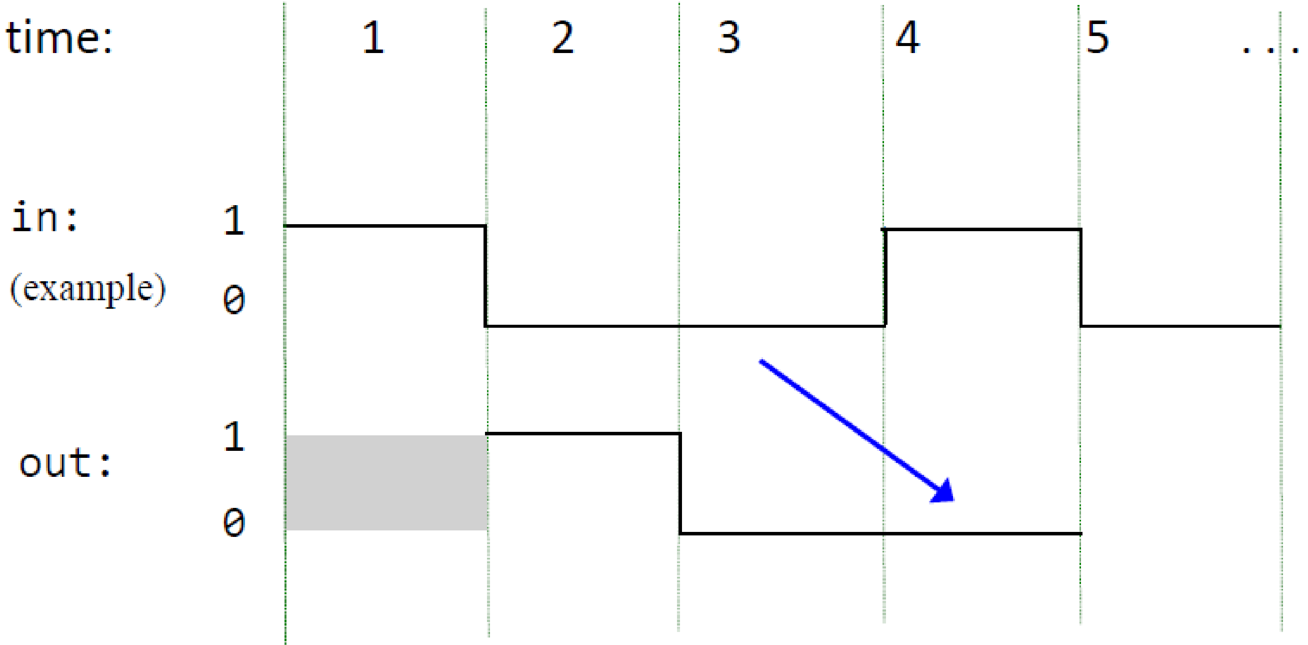


out(t) = in(t-1)

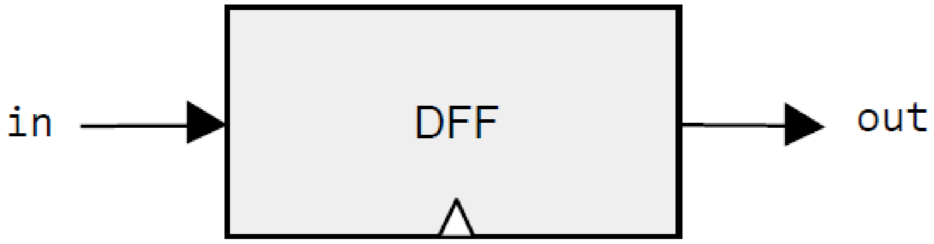


Flip-Flop

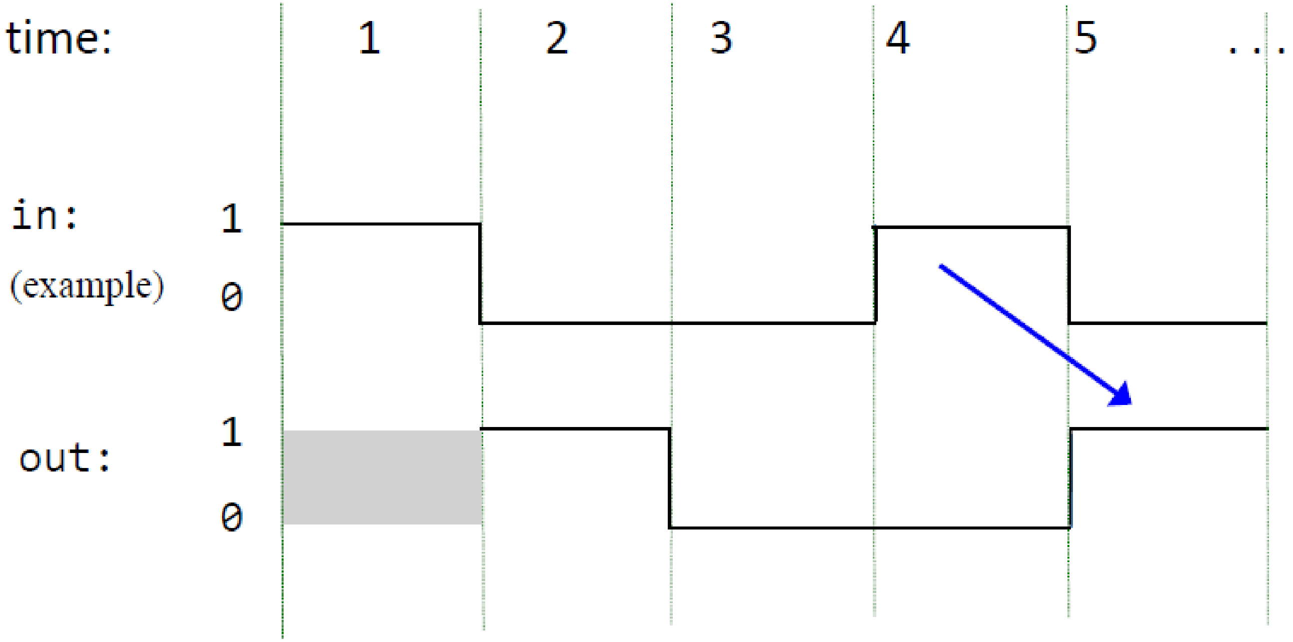




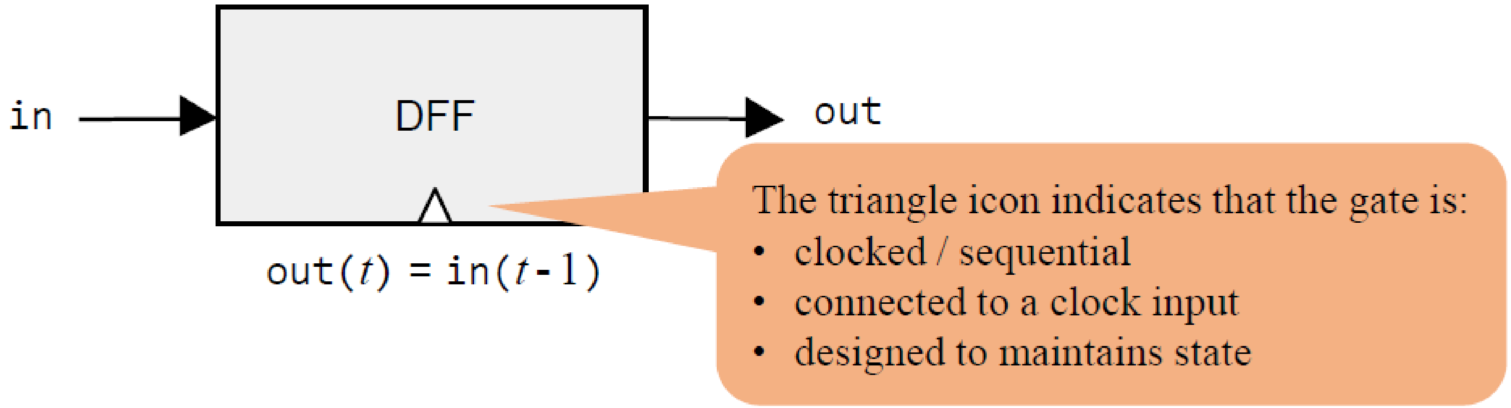
Flip-Flop



out(t) = in(t-1)



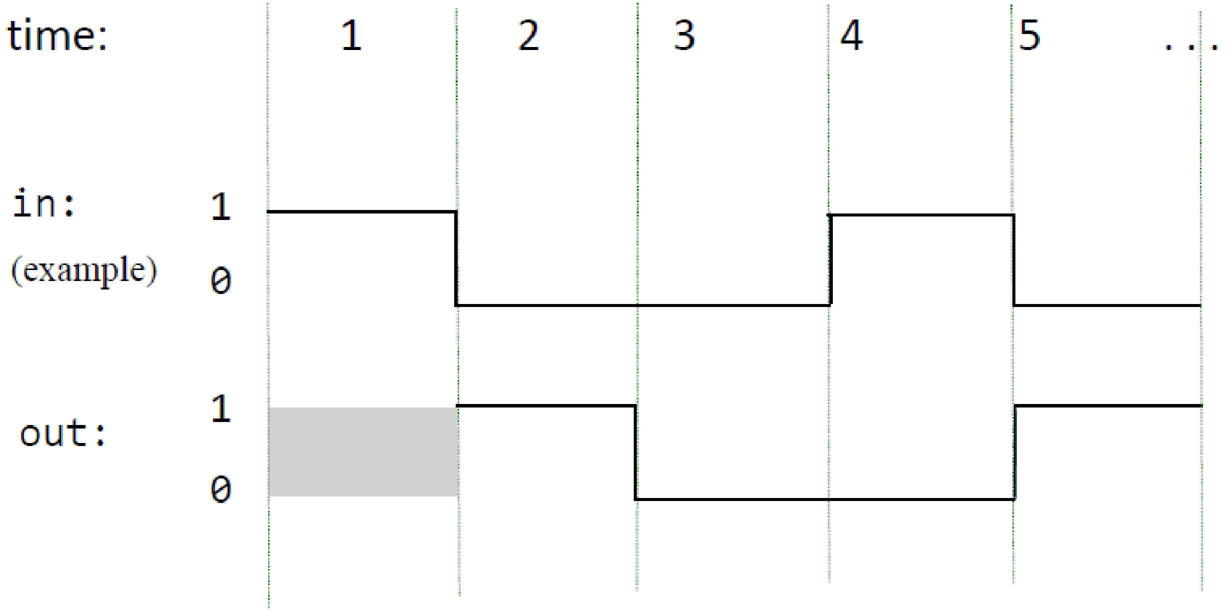
Flip-Flop



out(t) = in(t-1)

The triangle icon indicates that the gate is:

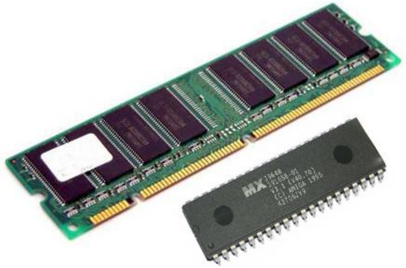
- clocked / sequential
- connected to a clock input
- designed to maintains state



Data and Time in DFF

out(t)= in(t-1)

- ***in*** is the gate's input value
- ***out*** is the gate's output value
- ***t*** is the current clock cycle
- ***t-1*** is the previous clock cycle
- **t+1** is the next clock cycle
- This elementary behavior can form the basis of all the hardware devices that computers use to maintain state



Sequential Chips

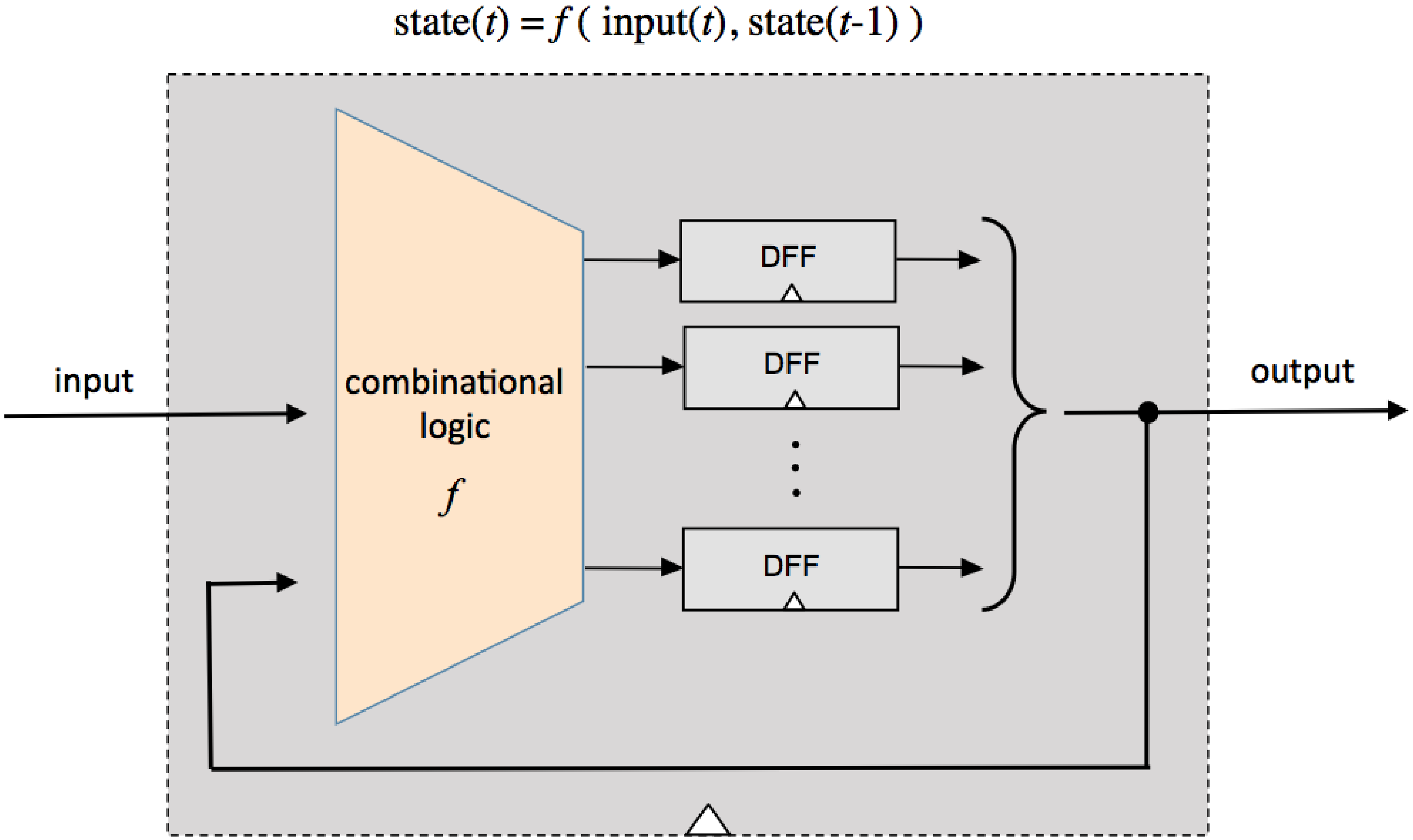
- Sequential chips are capable of:
  - Maintaining state
  - Acting on the state, and on the current inputs

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

- Example: DFF
  - The DFF state: the value of the input from the previous time unit
- Example: RAM
  - The RAM state: the current values of all its registers
  - Given some address (input), the RAM emits the value of the selected register
- All combinational chips can be constructed from NAND gates
- All sequential chips can be constructed from DFF gates, and combinational chips

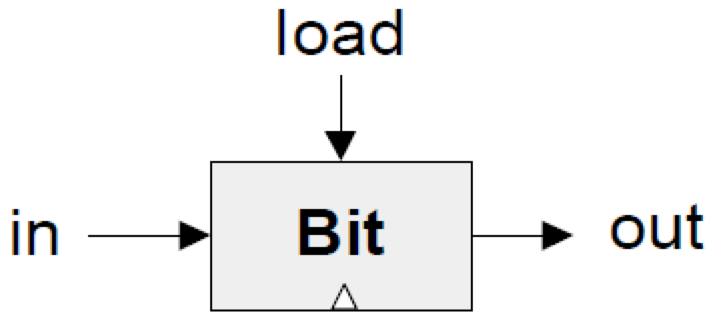
Sequential Chips

- Calculate -> Save



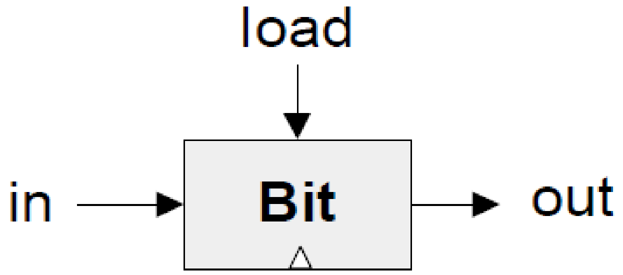
Register

- A register is a storage device that can "**store**" or "**remember**" a value over time
- Typically is composed of flip flops
- 1-bit register:
  - Store (maintain) a bit
  - Until it is instructed to load(store) another bit

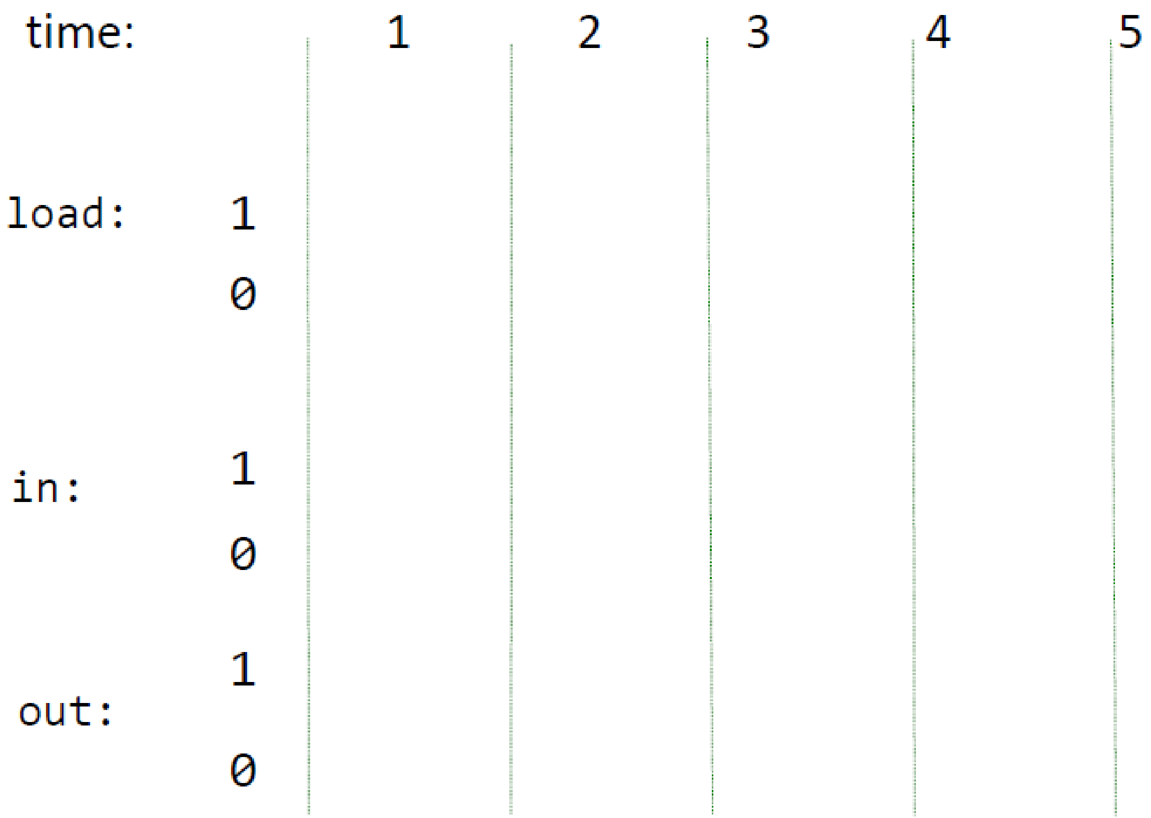


$$\begin{array}{ll} \text{if } \mathtt{load}(t) \text{ then} & \mathtt{out}(t+1) = \mathtt{in}(t) \\ \text{else} & \mathtt{out}(t+1) = \mathtt{out}(t) \end{array}$$

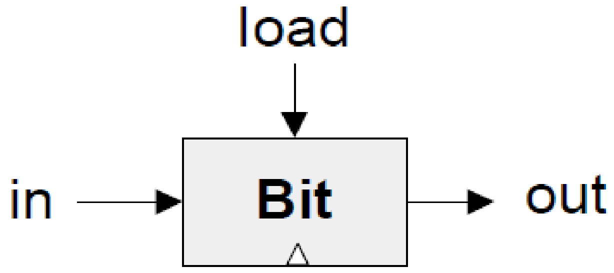
1-bit Register



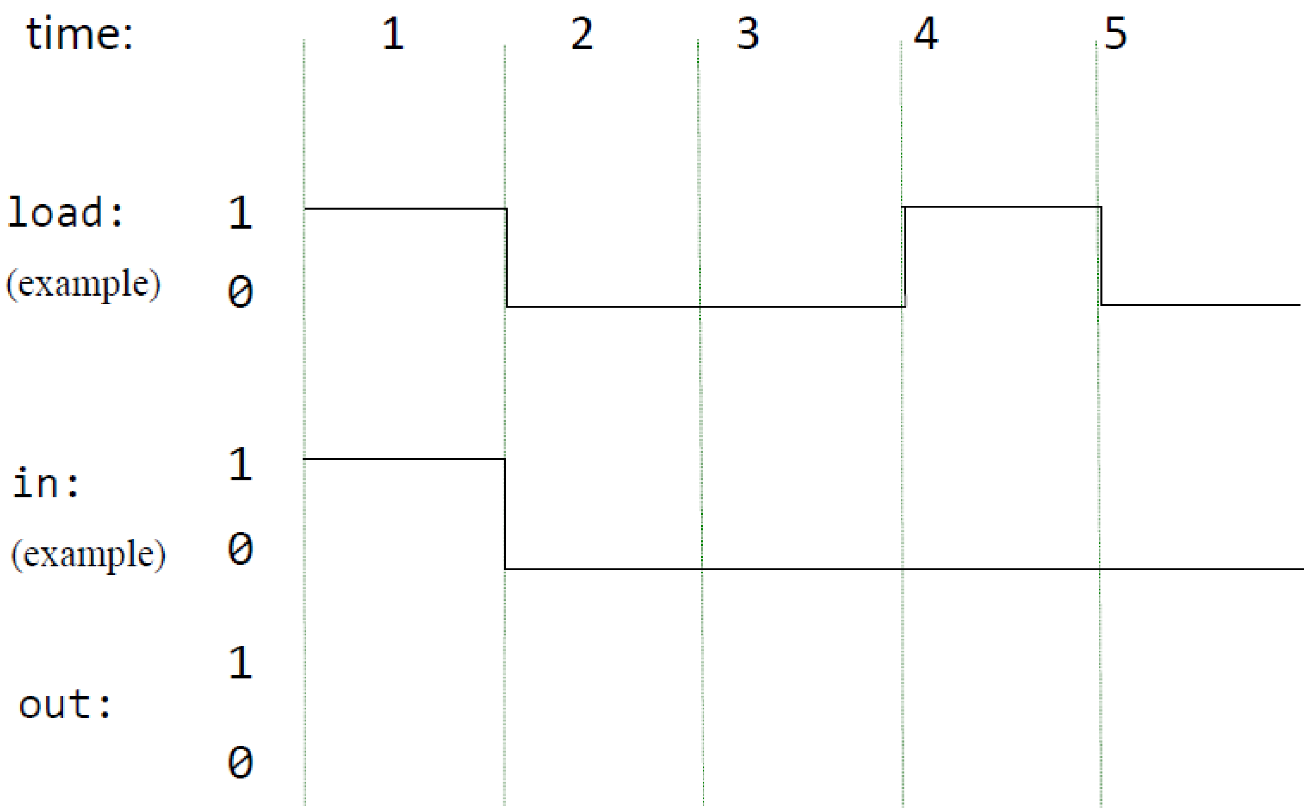
if `load`(t) then `out`$(t+1)$ = `in`(t)
else `out`$(t+1)$ = `out`(t)



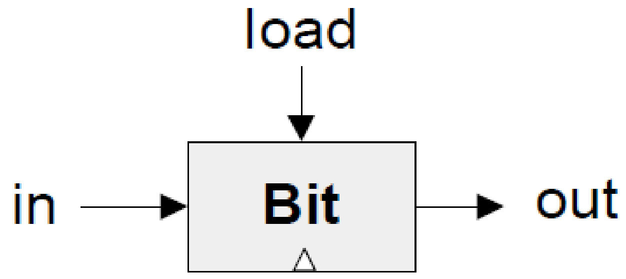
1-bit Register



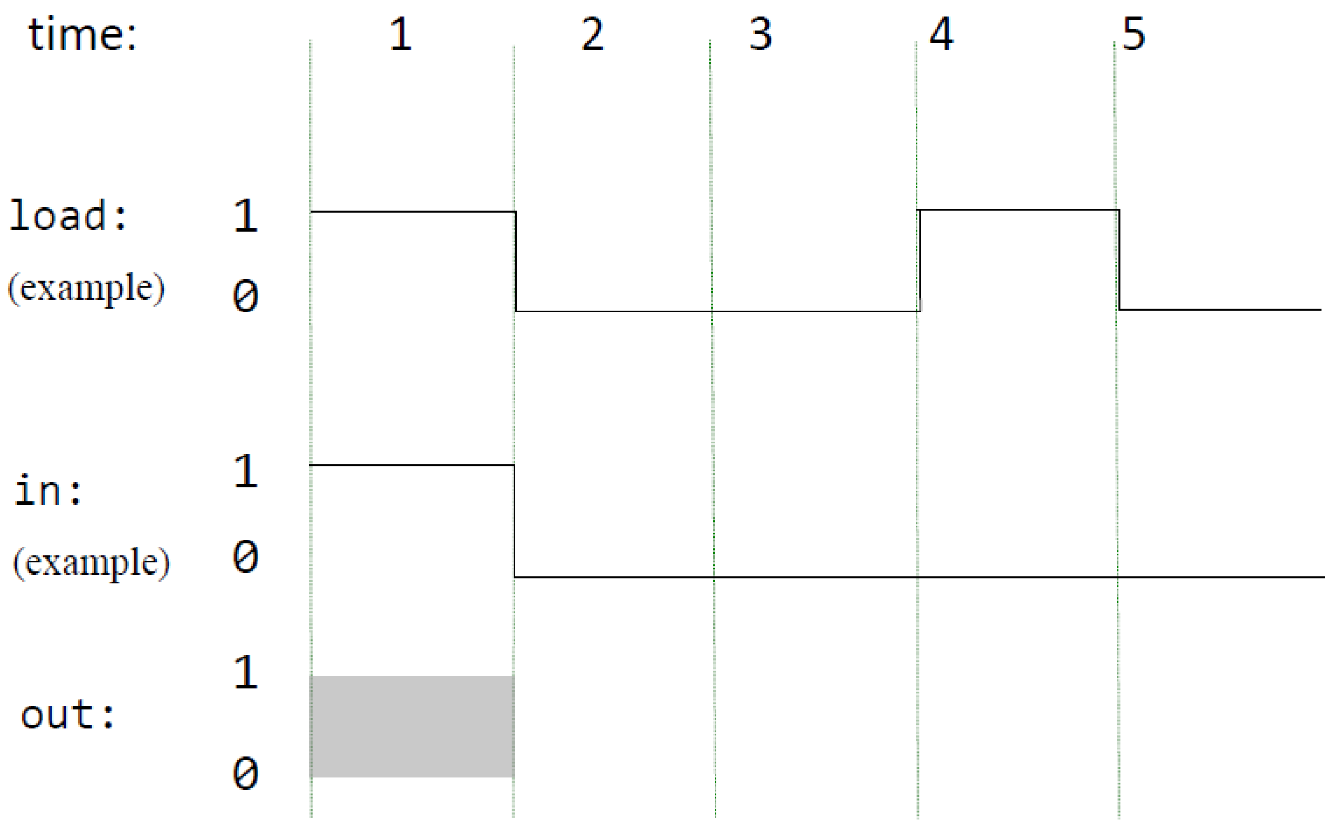
if `load`(t) then `out`$(t+1)$ = `in`(t)
else `out`$(t+1)$ = `out`(t)



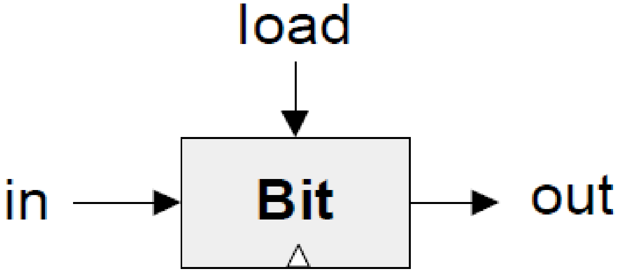
1-bit Register



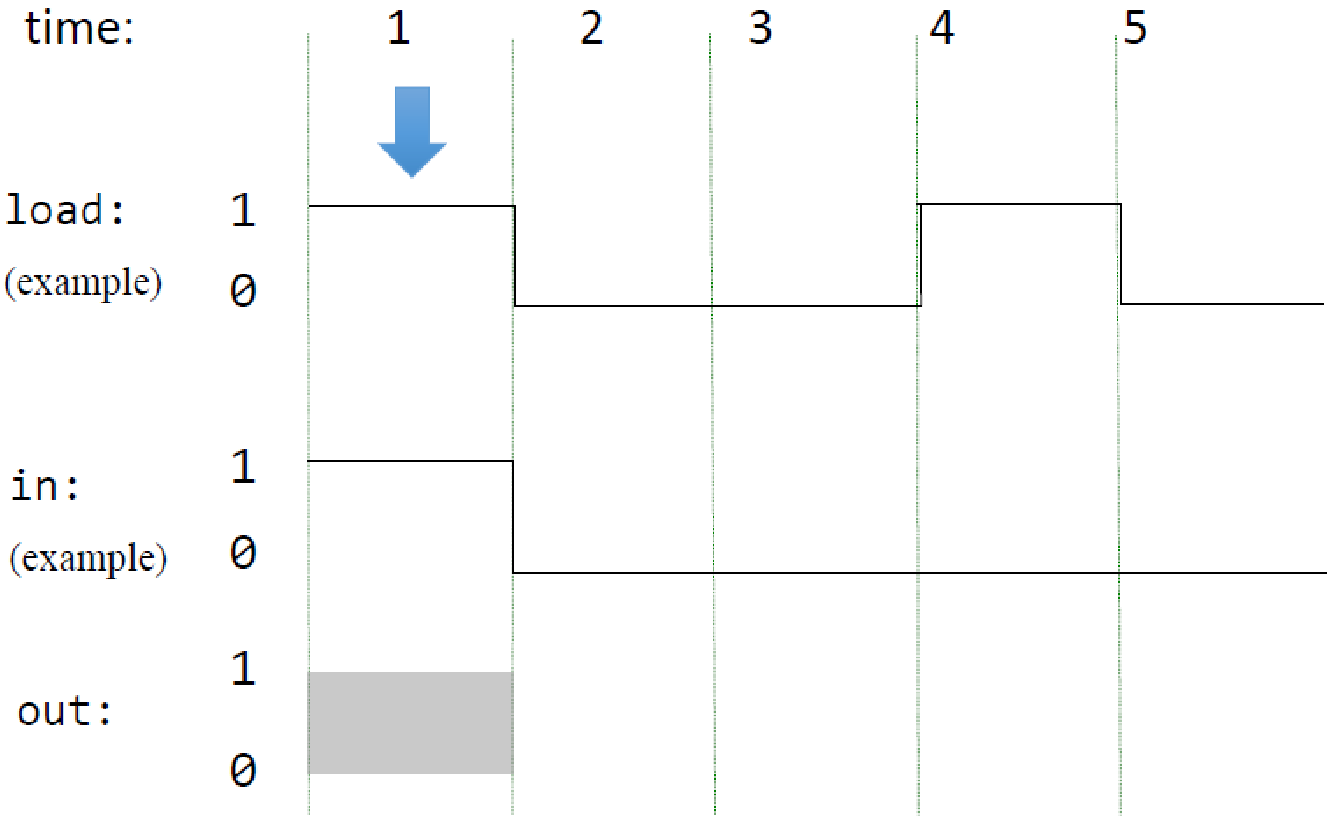
if load(t) then out$(t+1)$ = in(t)
else out$(t+1)$ = out(t)



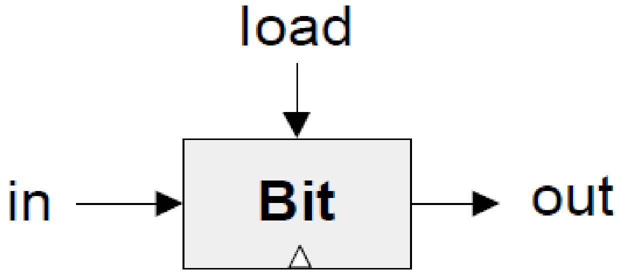
1-bit Register



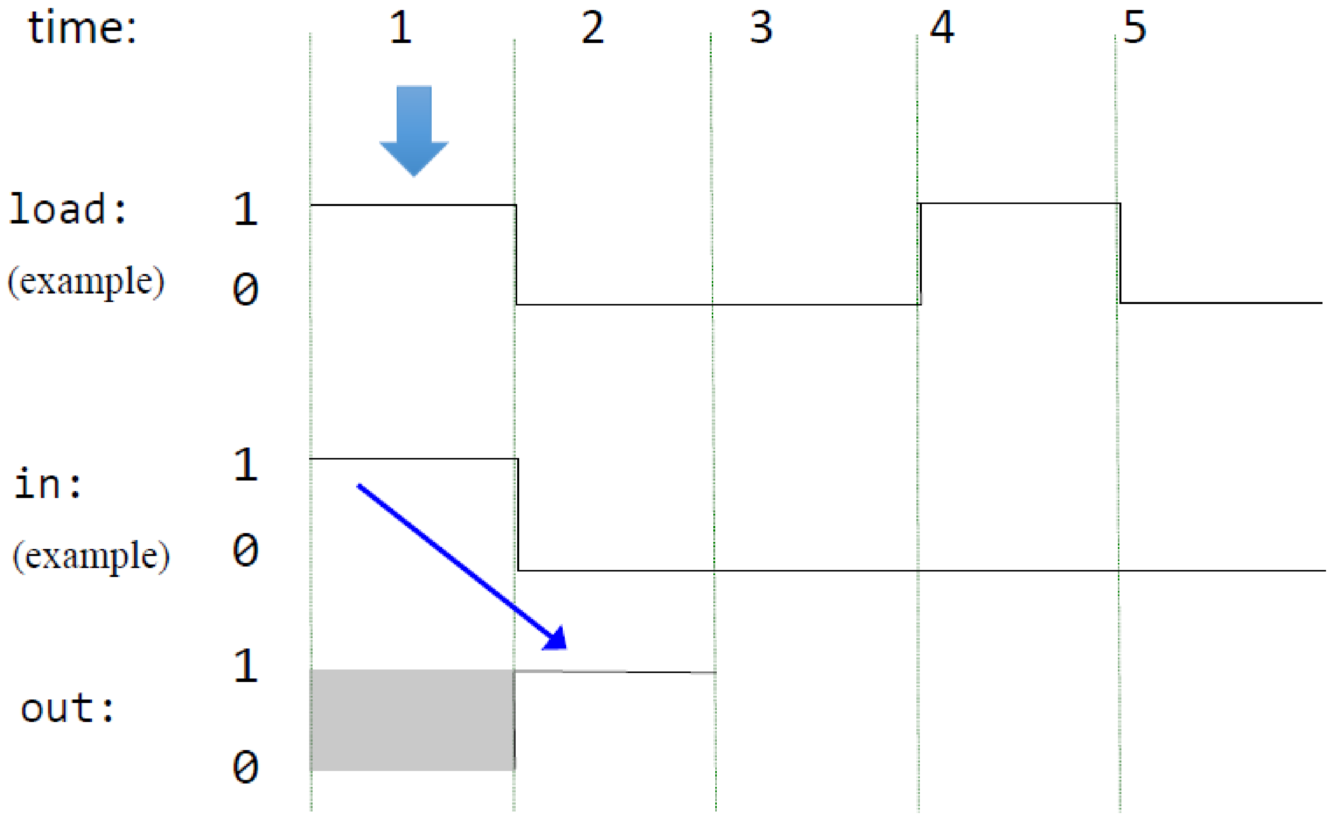
if $\texttt{load}(t)$ then $\texttt{out}(t+1) = \texttt{in}(t)$
else $\texttt{out}(t+1) = \texttt{out}(t)$



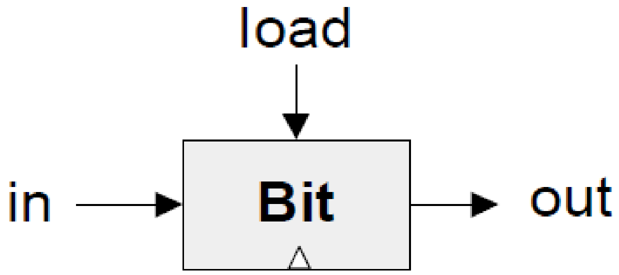
1-bit Register



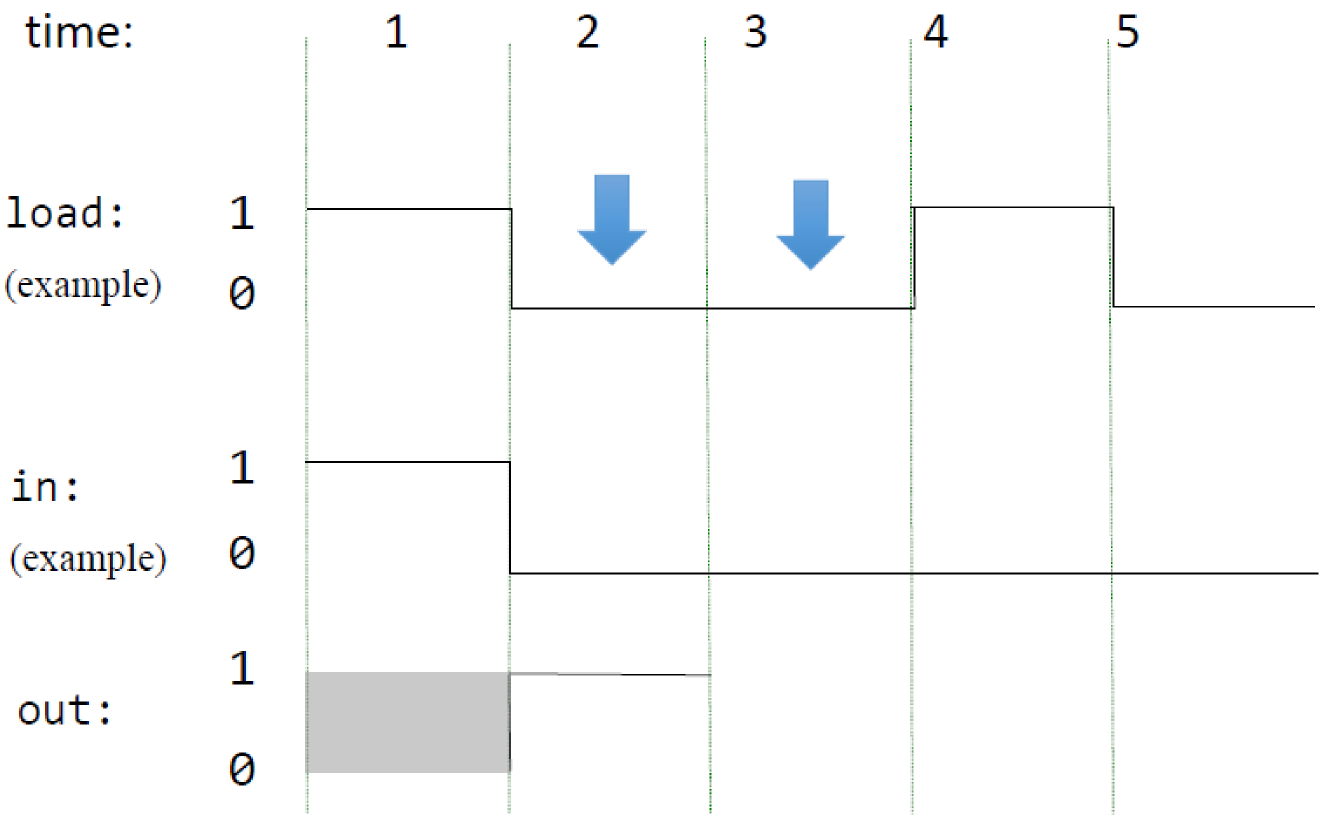
if `load(`t`)` then `out(`$t+1$`)` = `in(`t`)`
else `out(`$t+1$`)` = `out(`t`)`



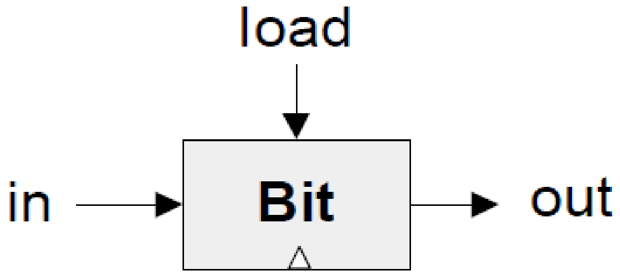
1-bit Register



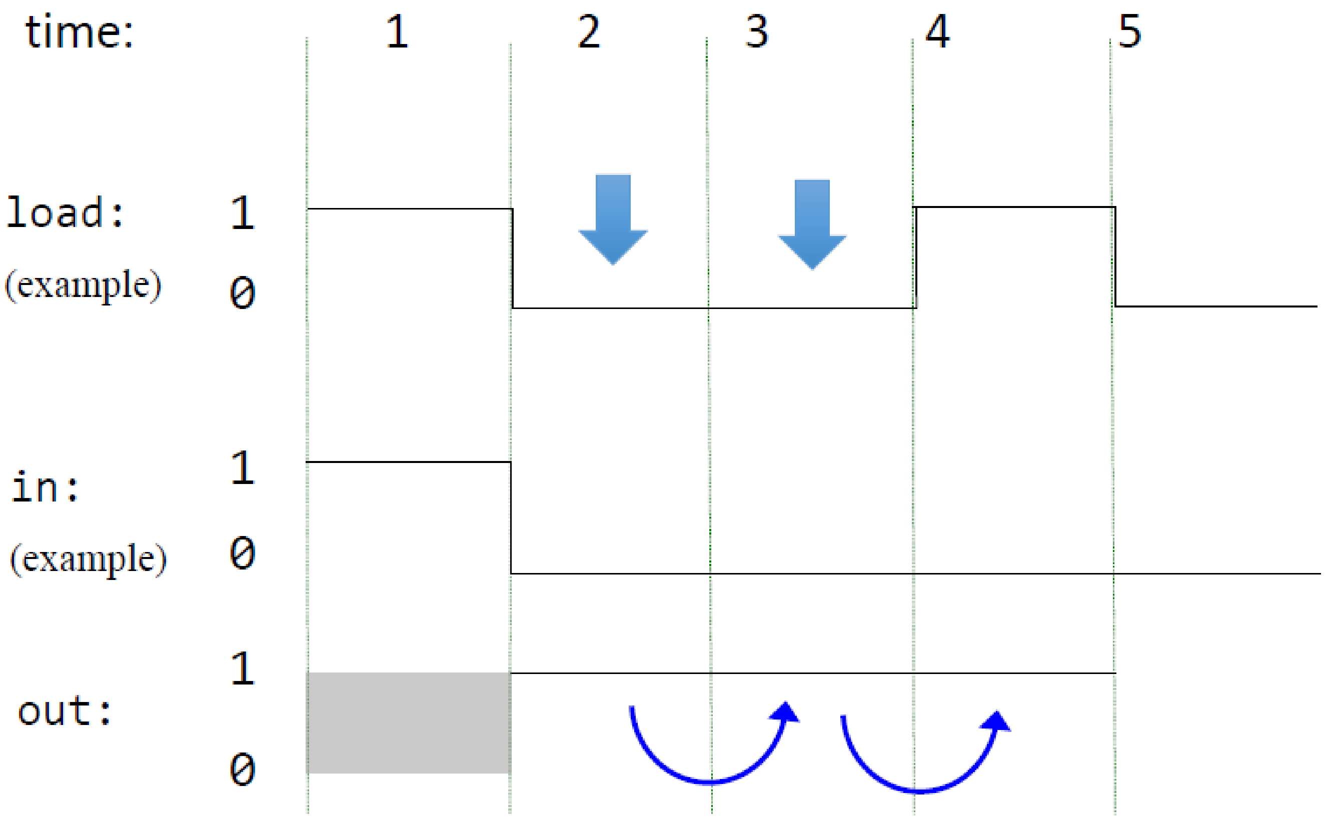
if load(t) then out($t+1$) = in(t)
else out($t+1$) = out(t)



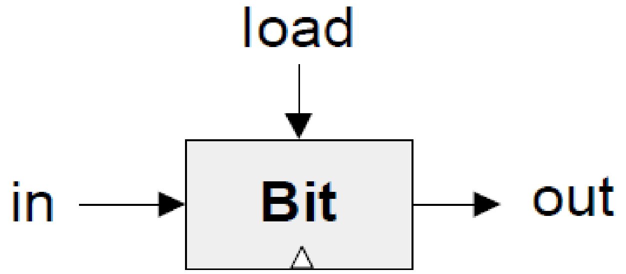
1-bit Register



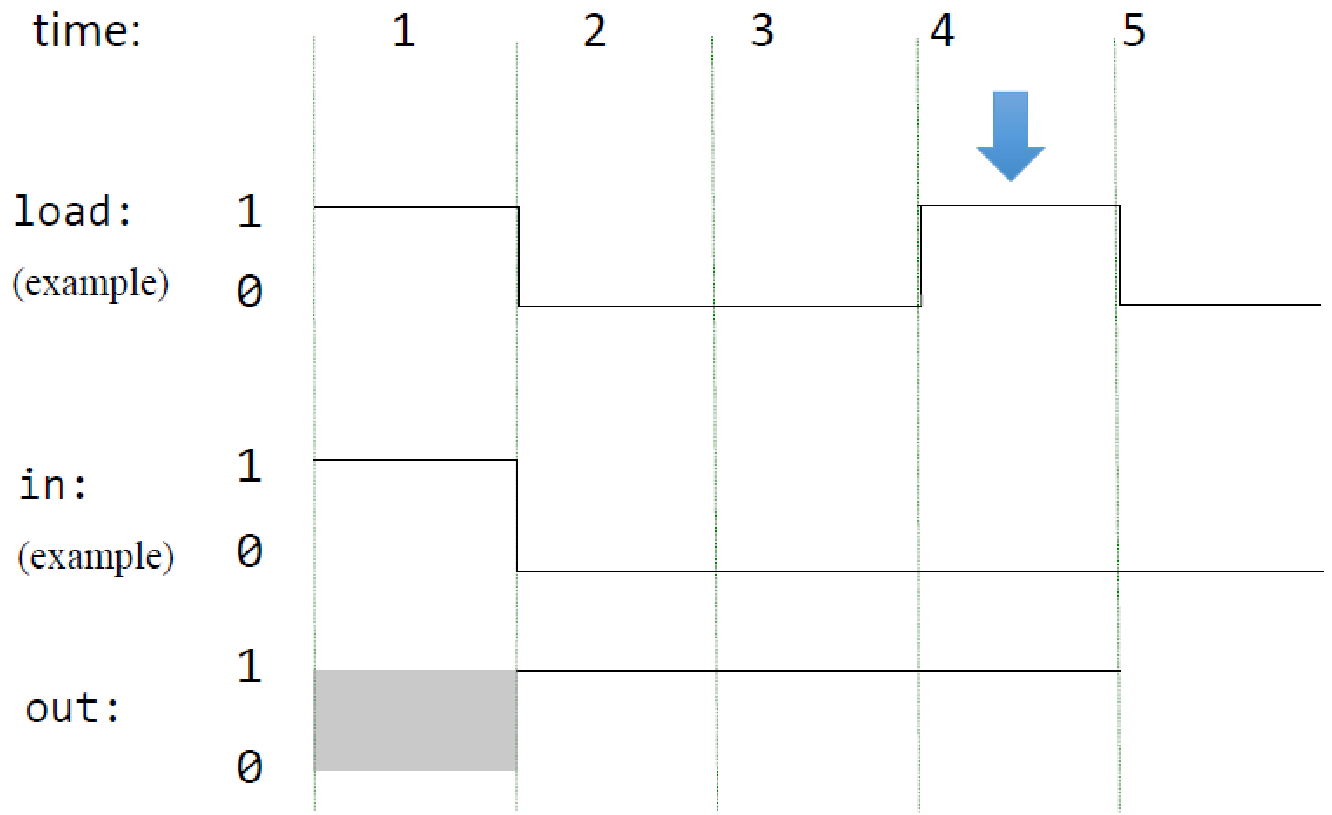
if load(t) then out($t+1$) = in(t)
else out($t+1$) = out(t)



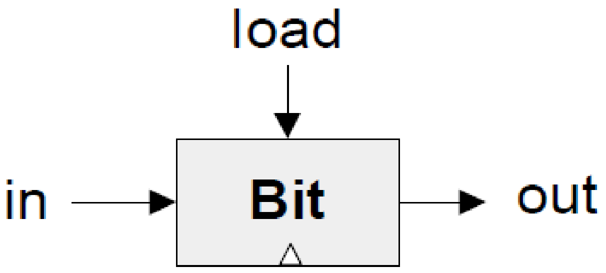
1-bit Register



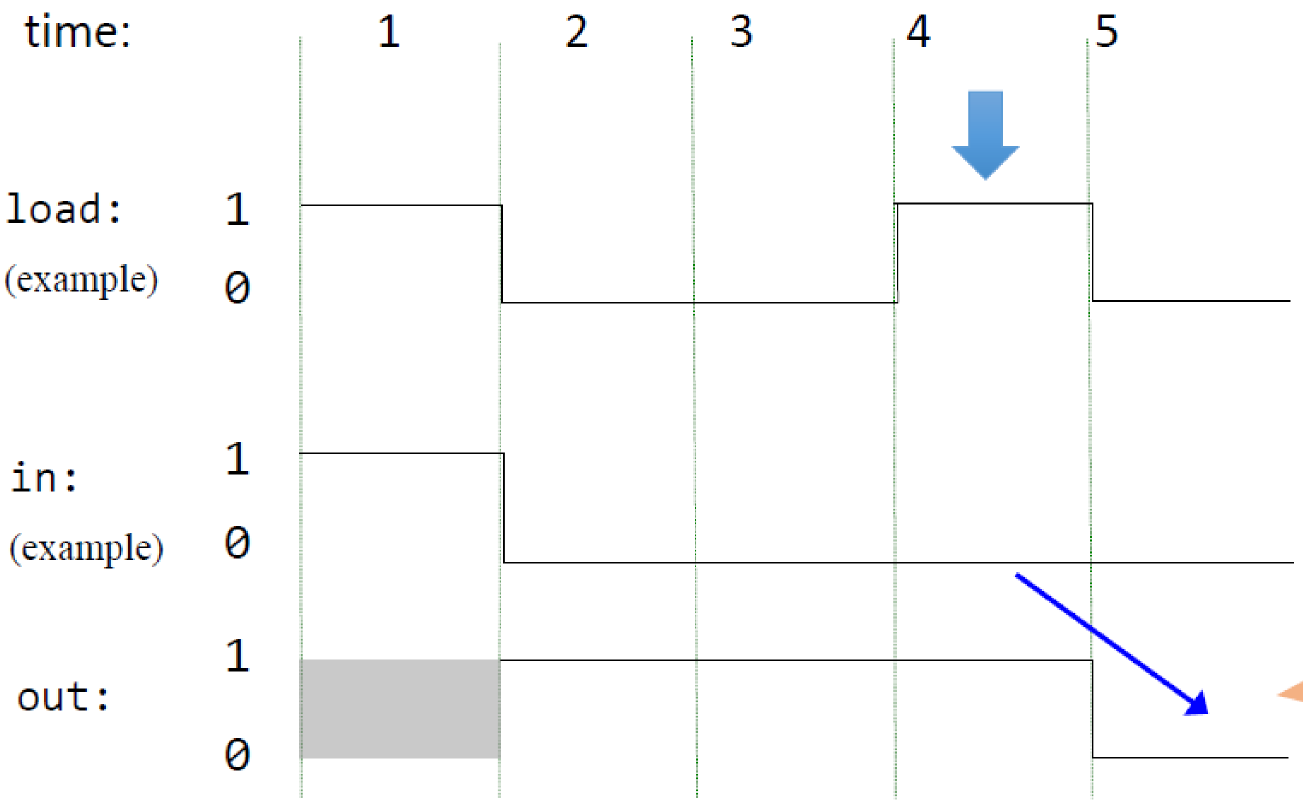
if `load`(t) then `out`$(t+1)$ = `in`(t)
else `out`$(t+1)$ = `out`(t)



1-bit Register



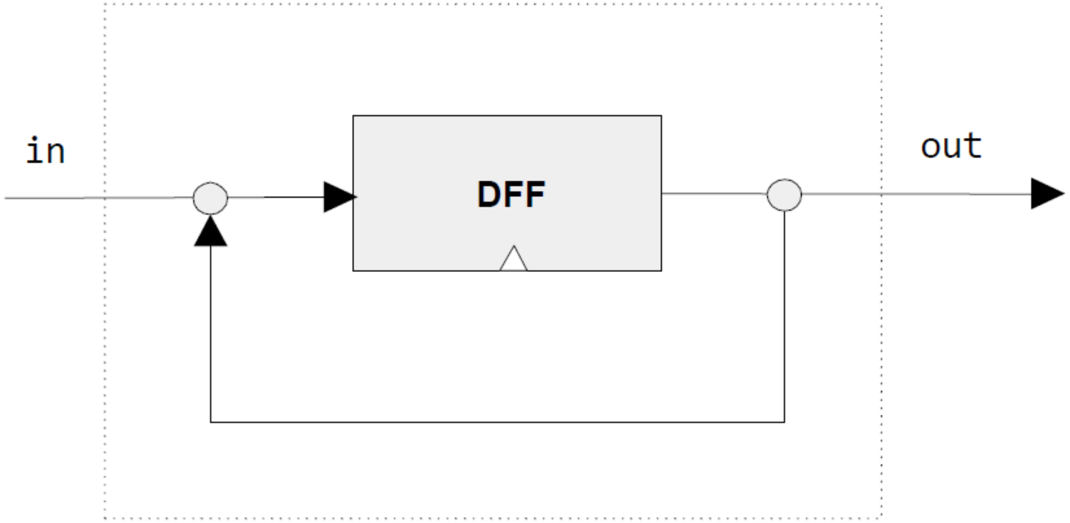
```
if load(t) then out(t+1) = in(t)
else            out(t+1) = out(t)
```



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

1-bit Register: Implementation

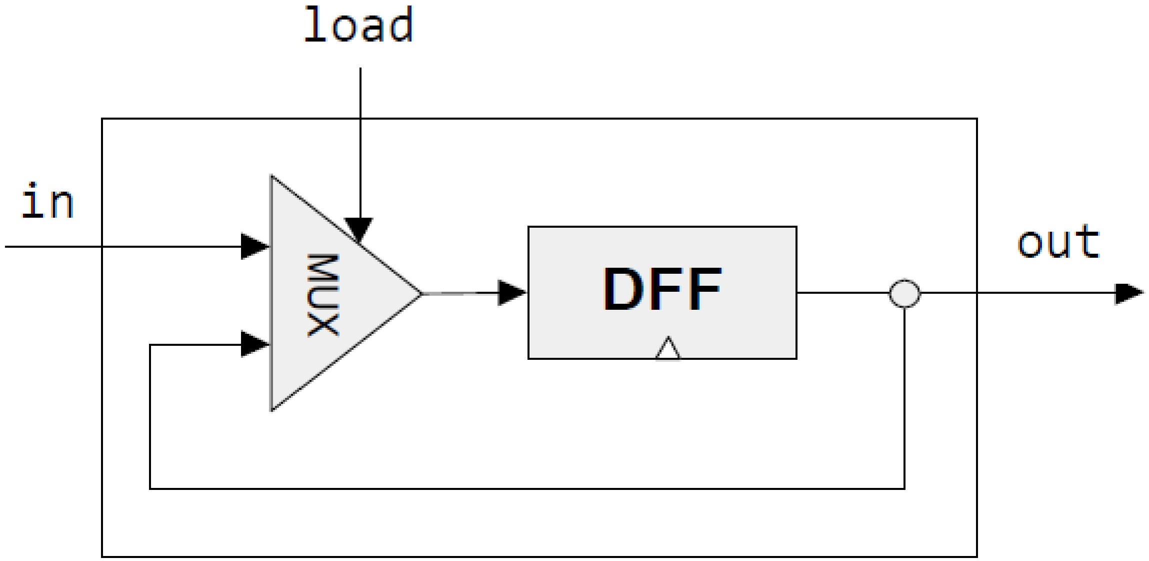
- This first objective of 1-bit register suggests that it can be implemented from a DFF by simply feeding the output back into its input, creating a device below



- It won’t work as:
  - There is no way to load data
  - Could have 2 different values as input

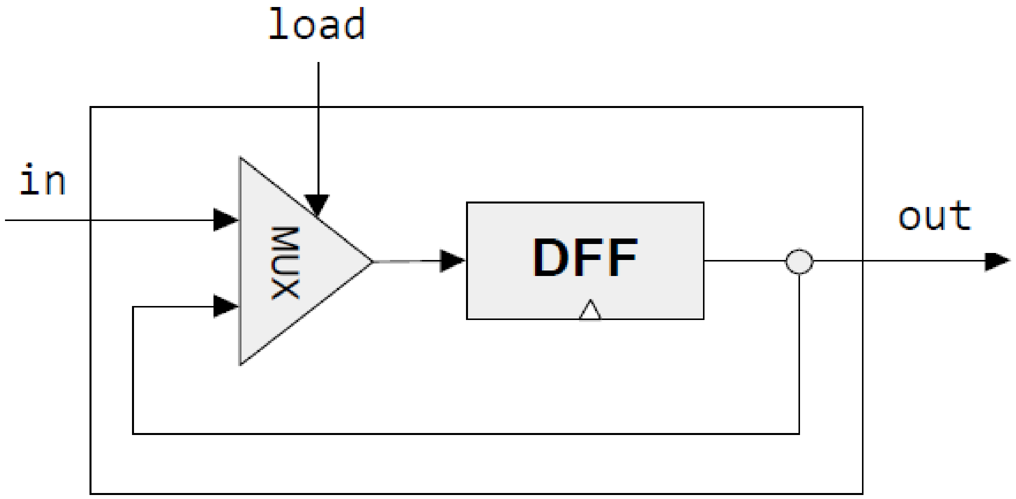
1-bit Register: Implementation

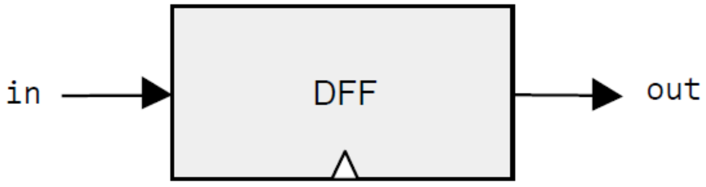
- One way to solve the problem is to include a multiplexor into the design
- The select bit of the Mux can become the load bit!



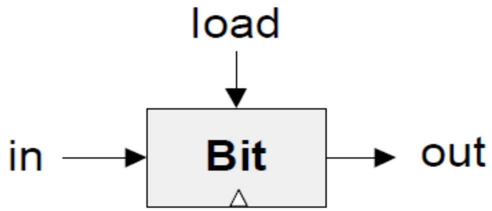
1-bit Register

```
if load == 1
  out = in
else
  out = b
```

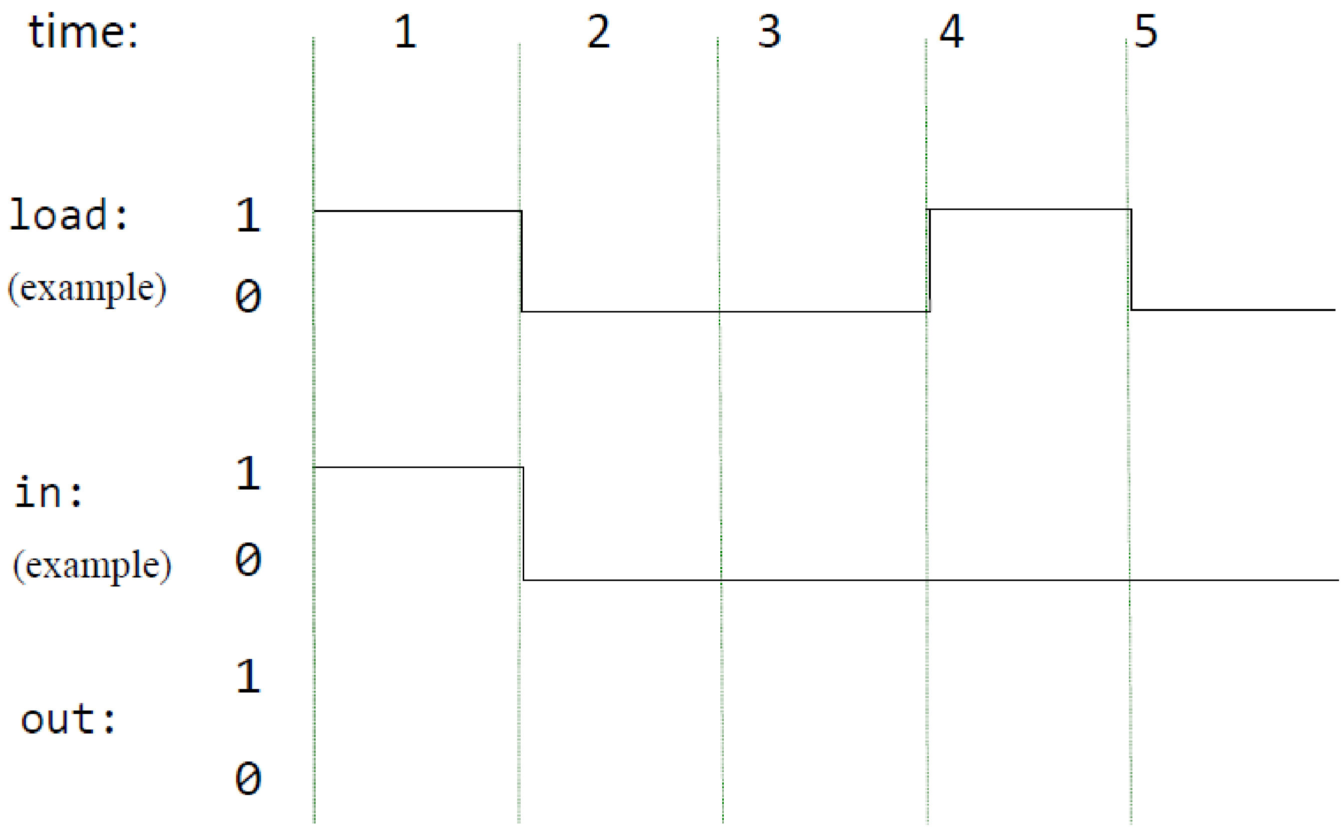




out(t) = in(t-1)

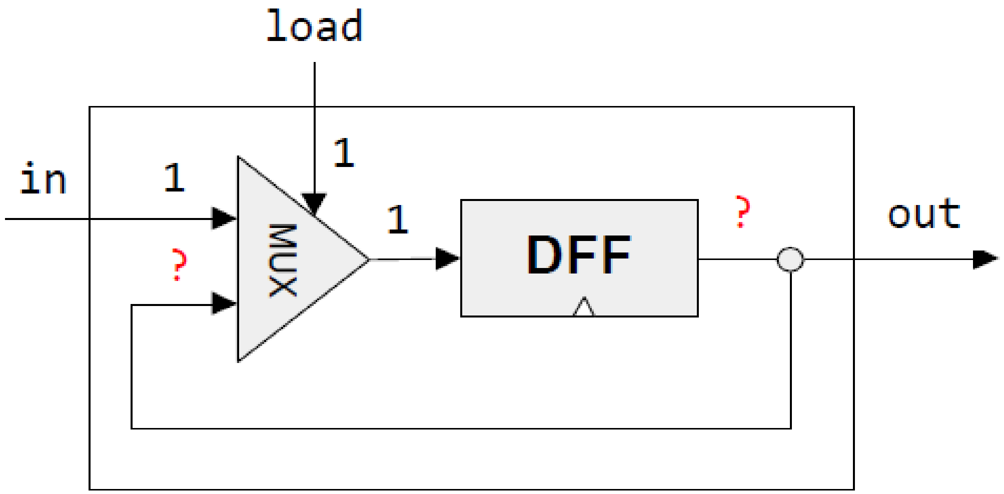


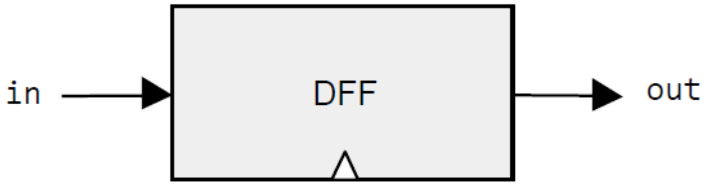
if load(t) then out(t+1) = in(t)
else out(t+1) = out(t)



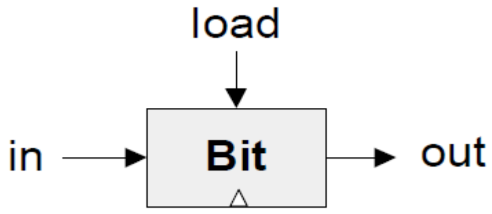
1-bit Register

```
if load == 1
  out = in
else
  out = b
```

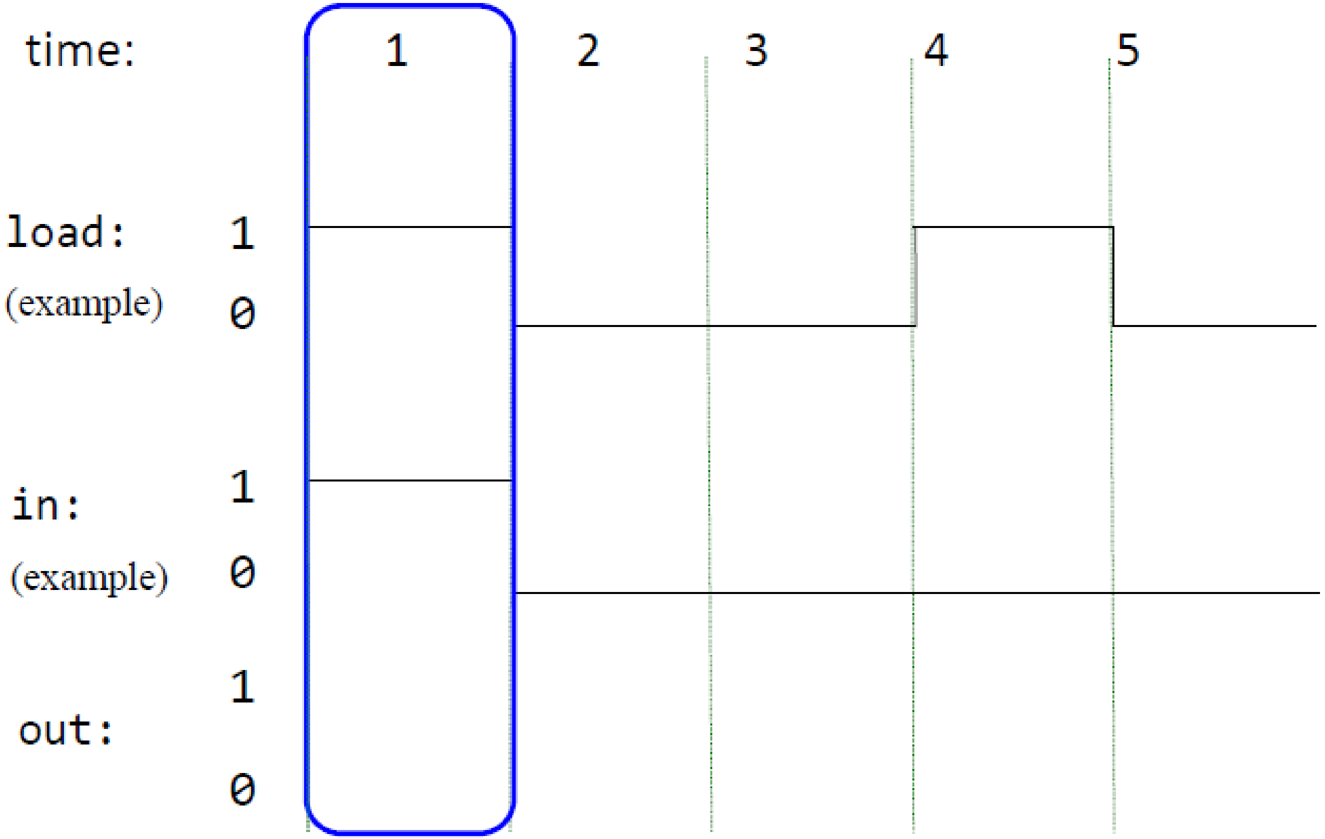




$\text{out}(t) = \text{in}(t-1)$

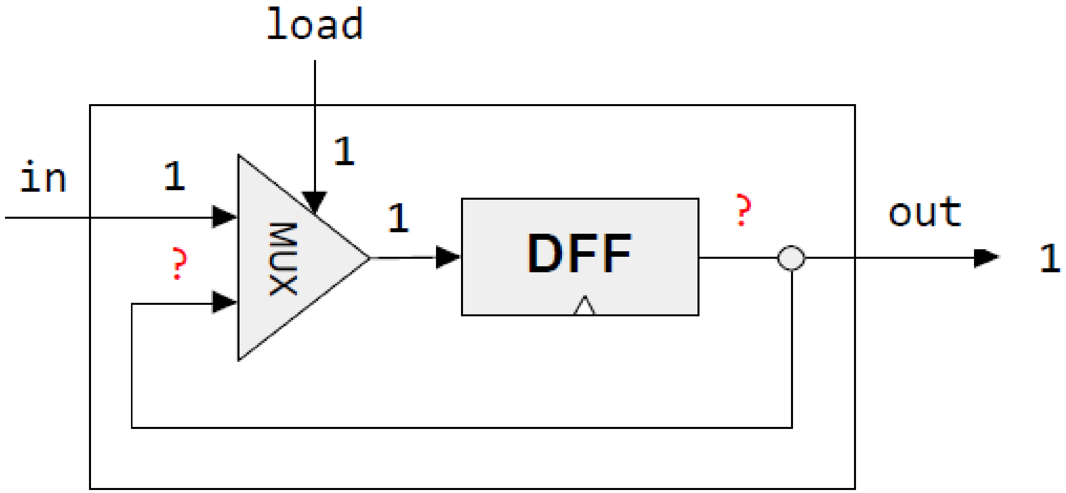


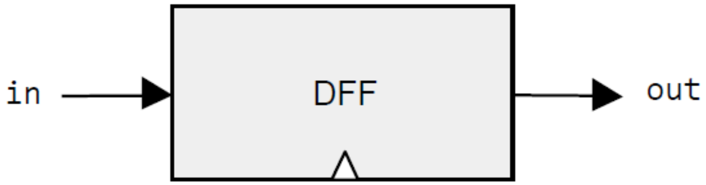
if $\text{load}(t)$ then $\text{out}(t+1) = \text{in}(t)$
else $\text{out}(t+1) = \text{out}(t)$



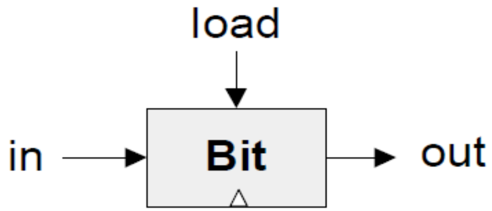
1-bit Register

```
if load == 1
   out = in
else
   out = b
```

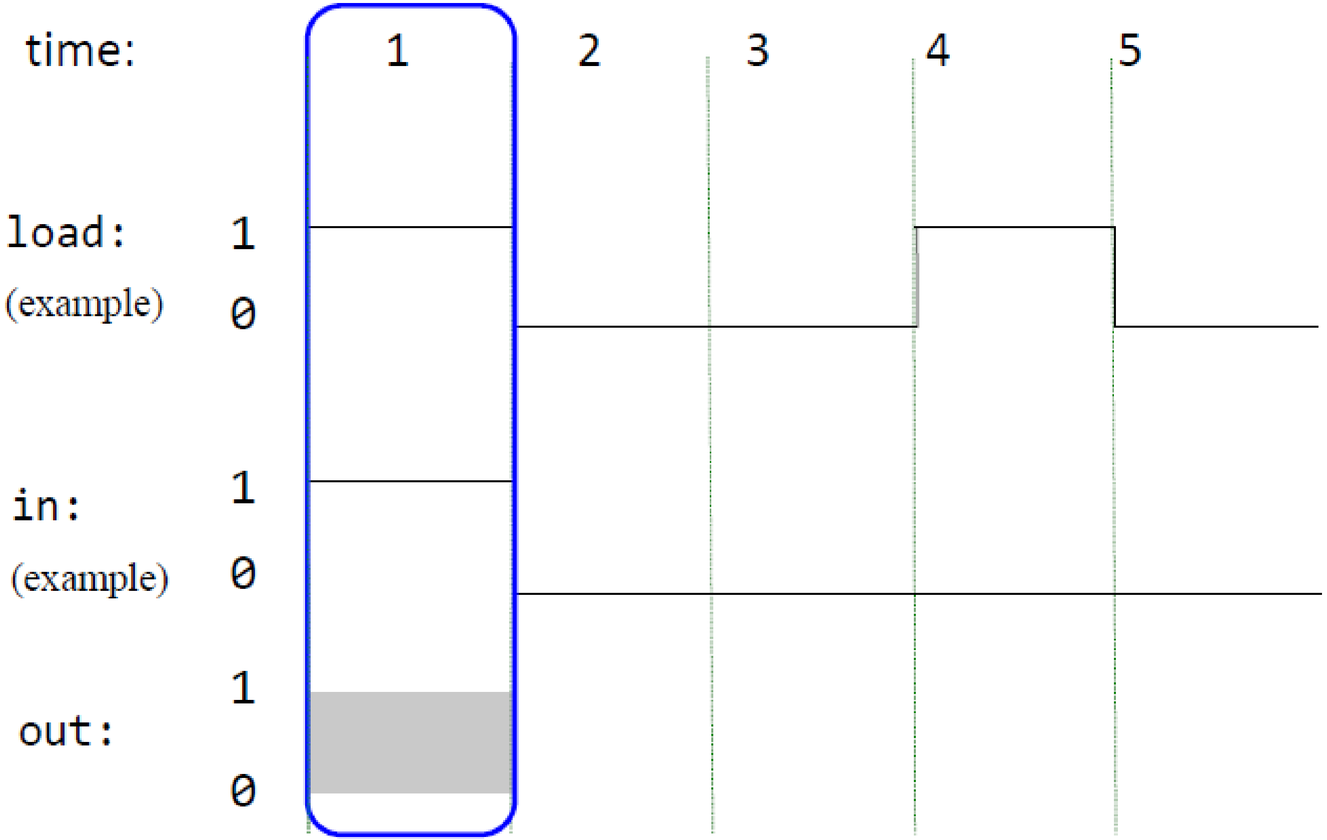




out(t) = in(t-1)

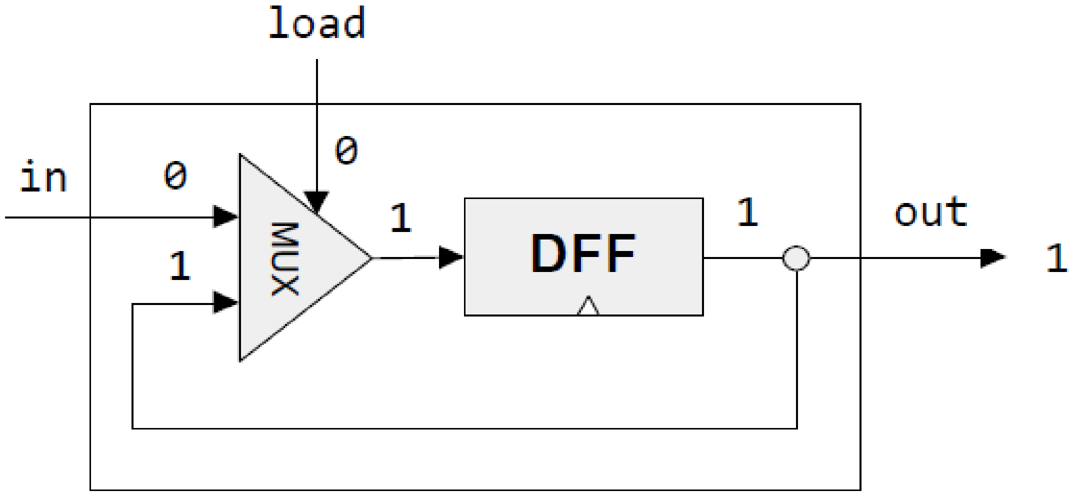


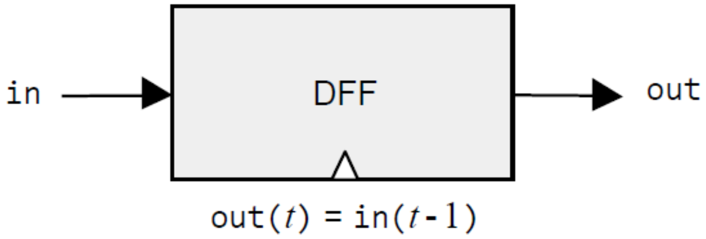
if load(t) then out(t+1) = in(t)
else out(t+1) = out(t)



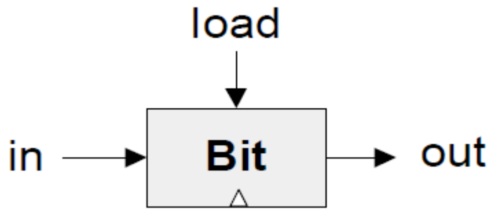
1-bit Register

```
if load == 1
  out = in
else
  out = b
```

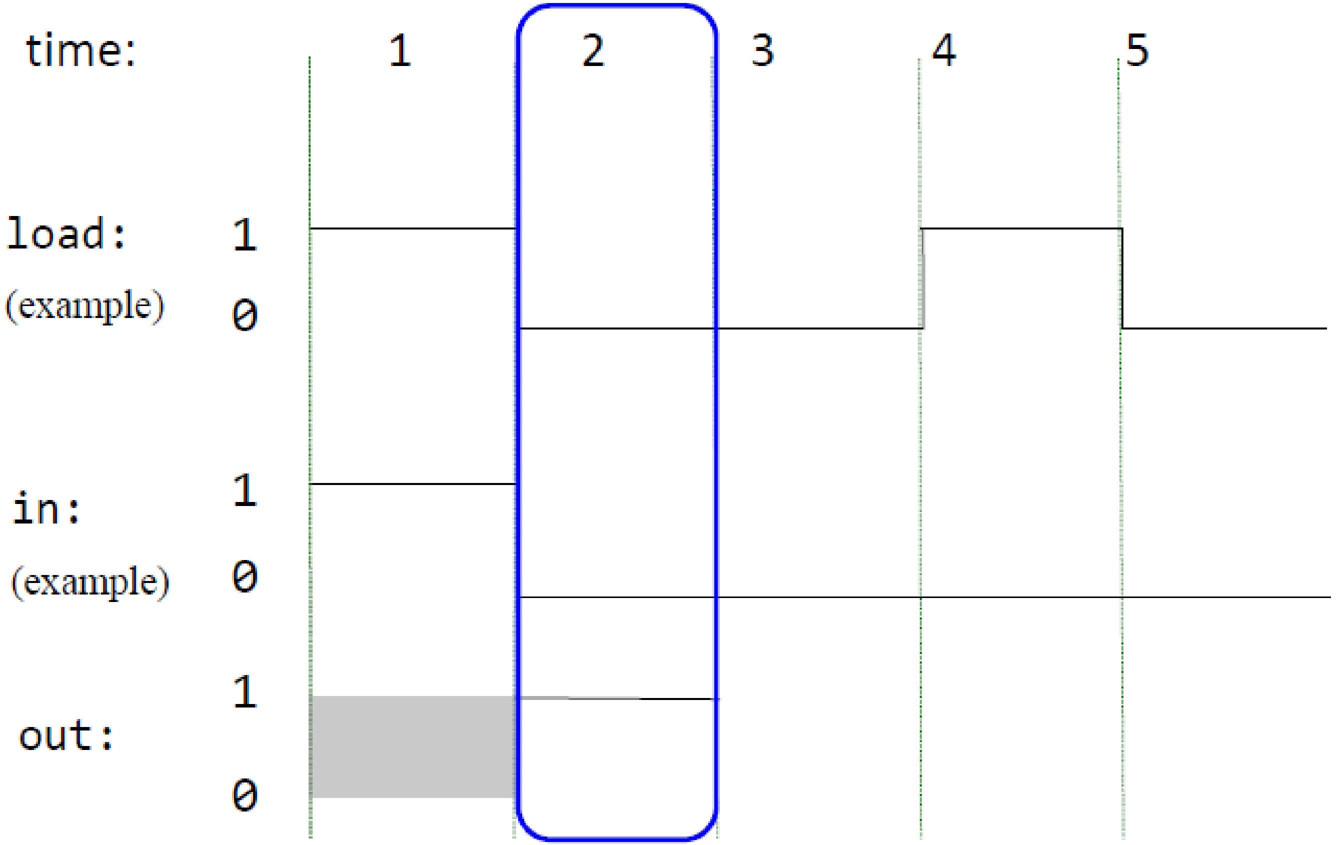




out(t) = in(t-1)

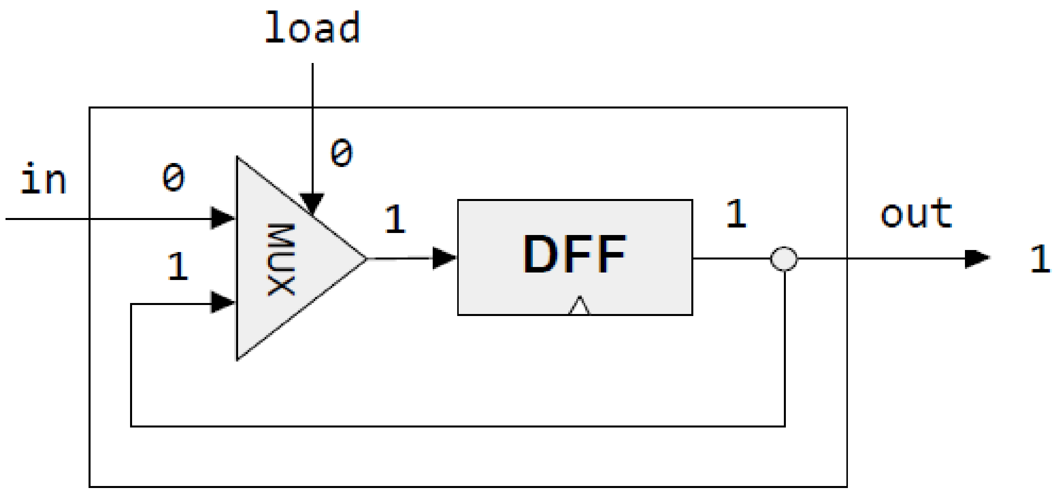


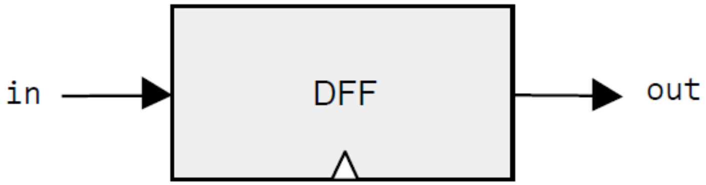
if load(t) then out(t+1) = in(t)
else out(t+1) = out(t)



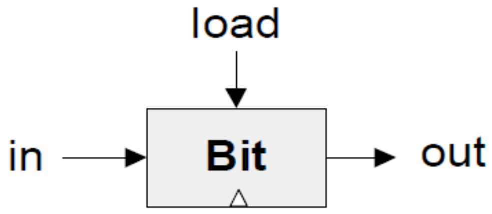
1-bit Register

```
if load == 1
  out = in
else
  out = b
```

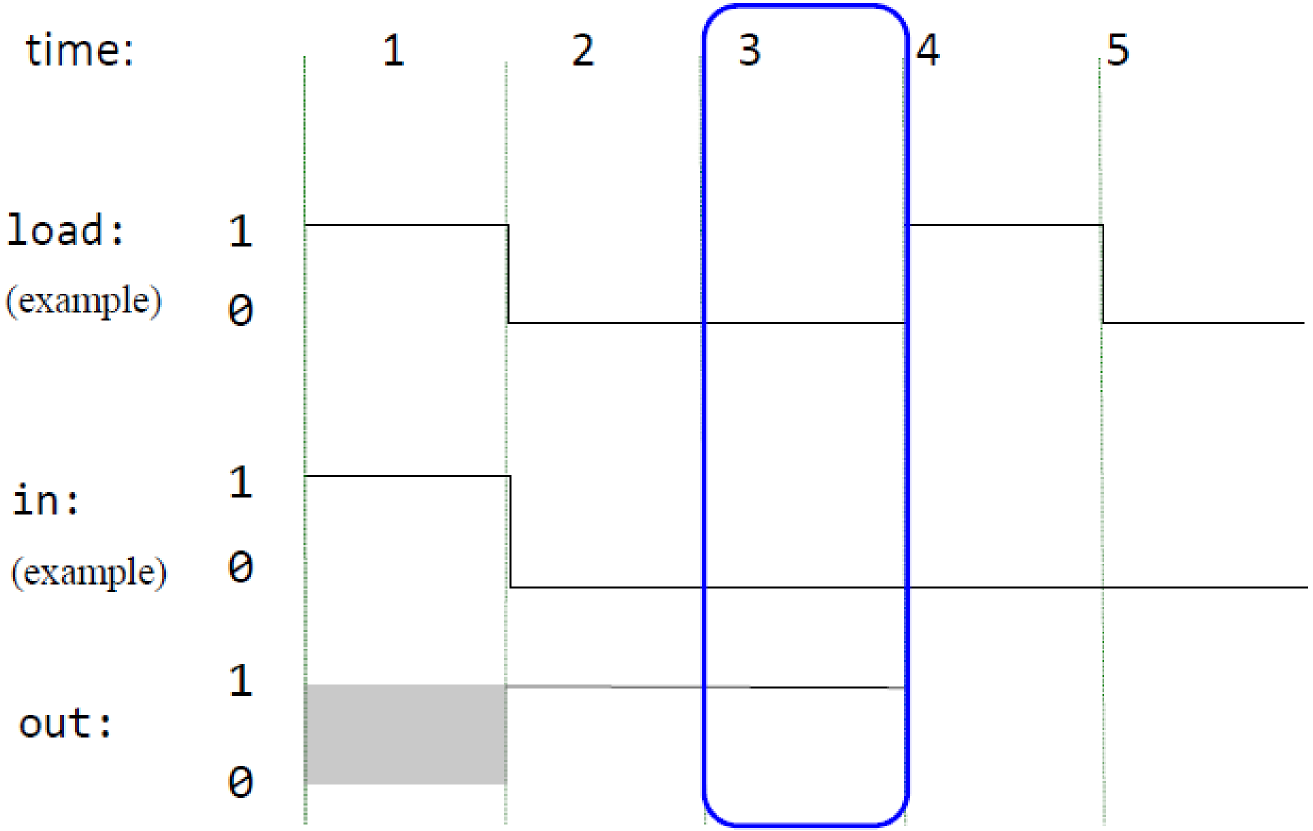




$\text{out}(t) = \text{in}(t-1)$

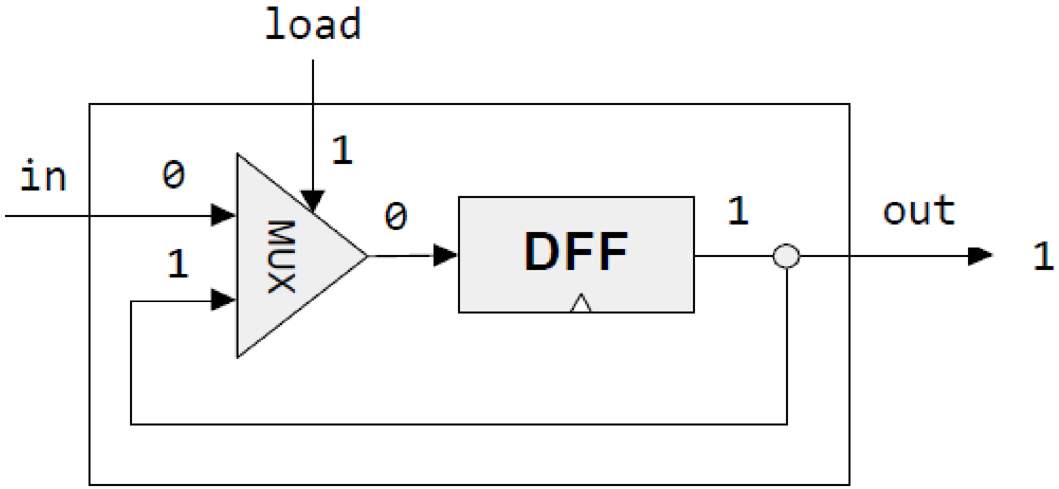


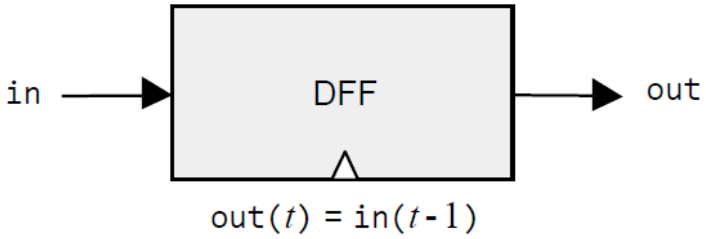
if $\text{load}(t)$ then $\text{out}(t+1) = \text{in}(t)$
else $\text{out}(t+1) = \text{out}(t)$

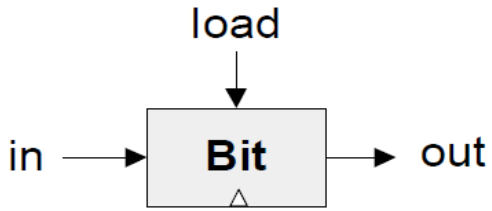


1-bit Register

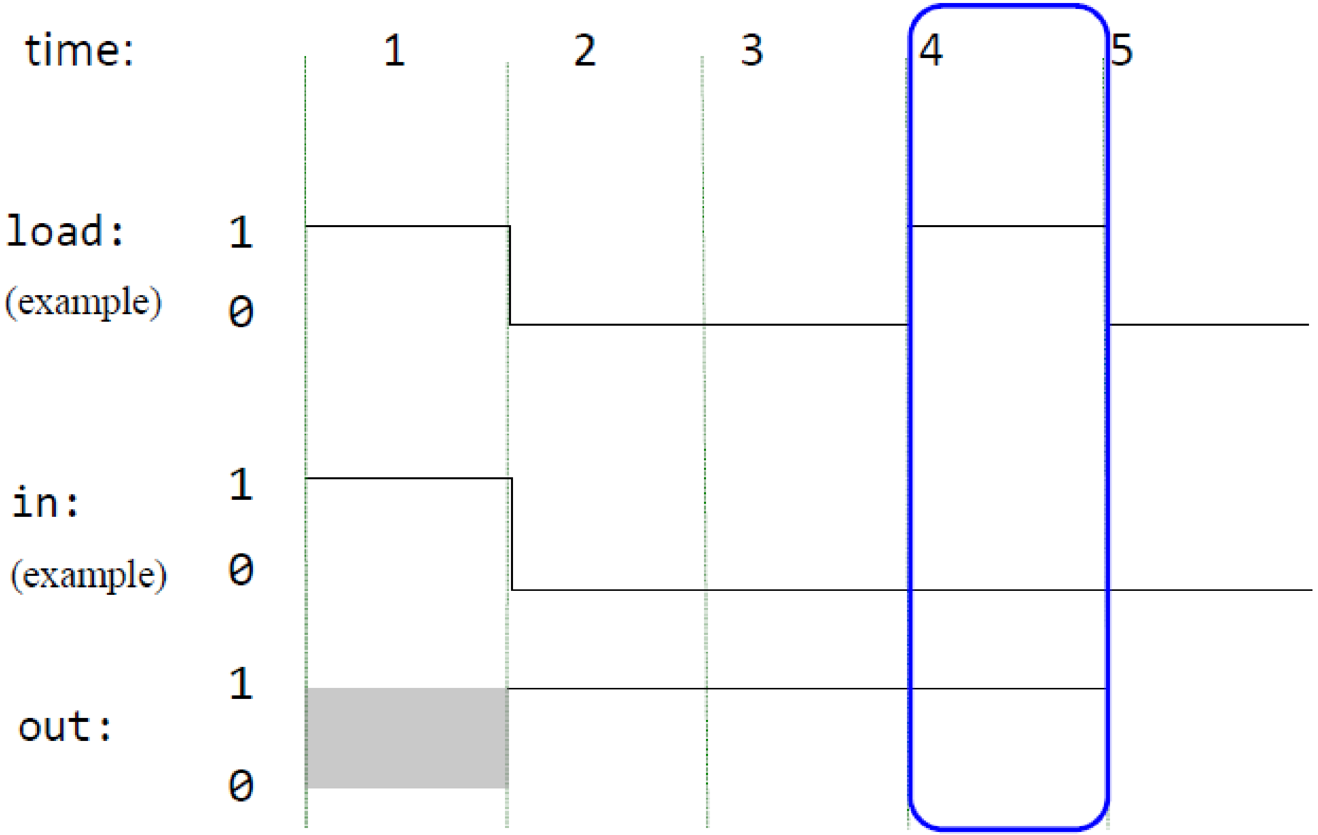
```
if load == 1
  out = in
else
  out = b
```





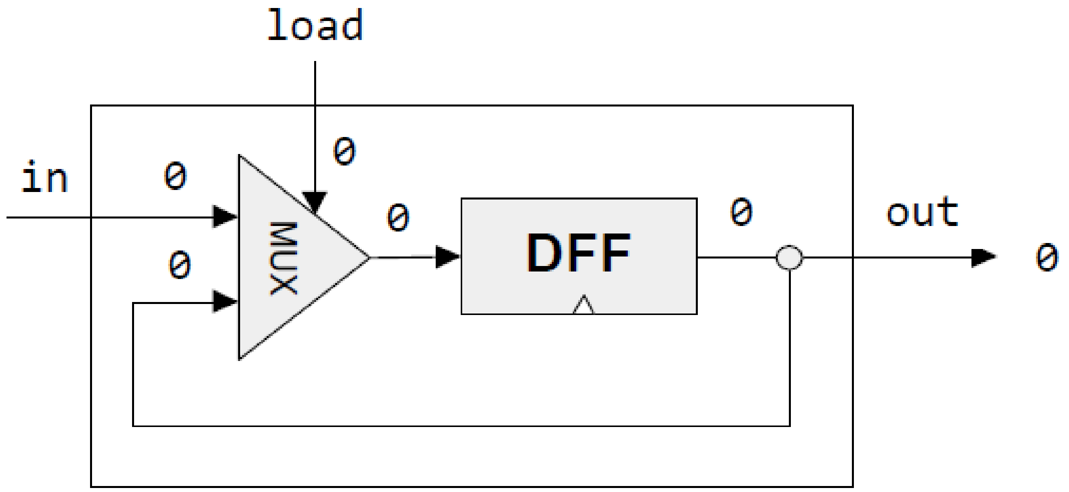


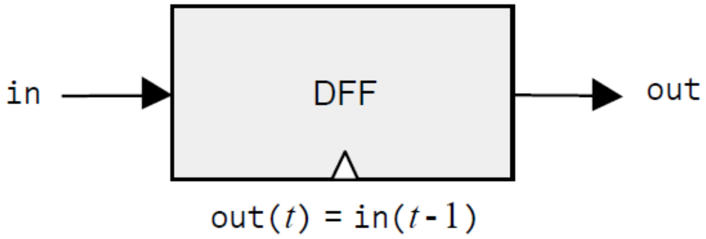
if load(t) then out(t+1) = in(t)
else out(t+1) = out(t)

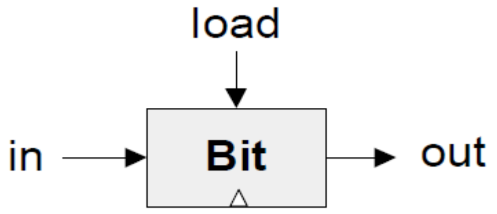


1-bit Register

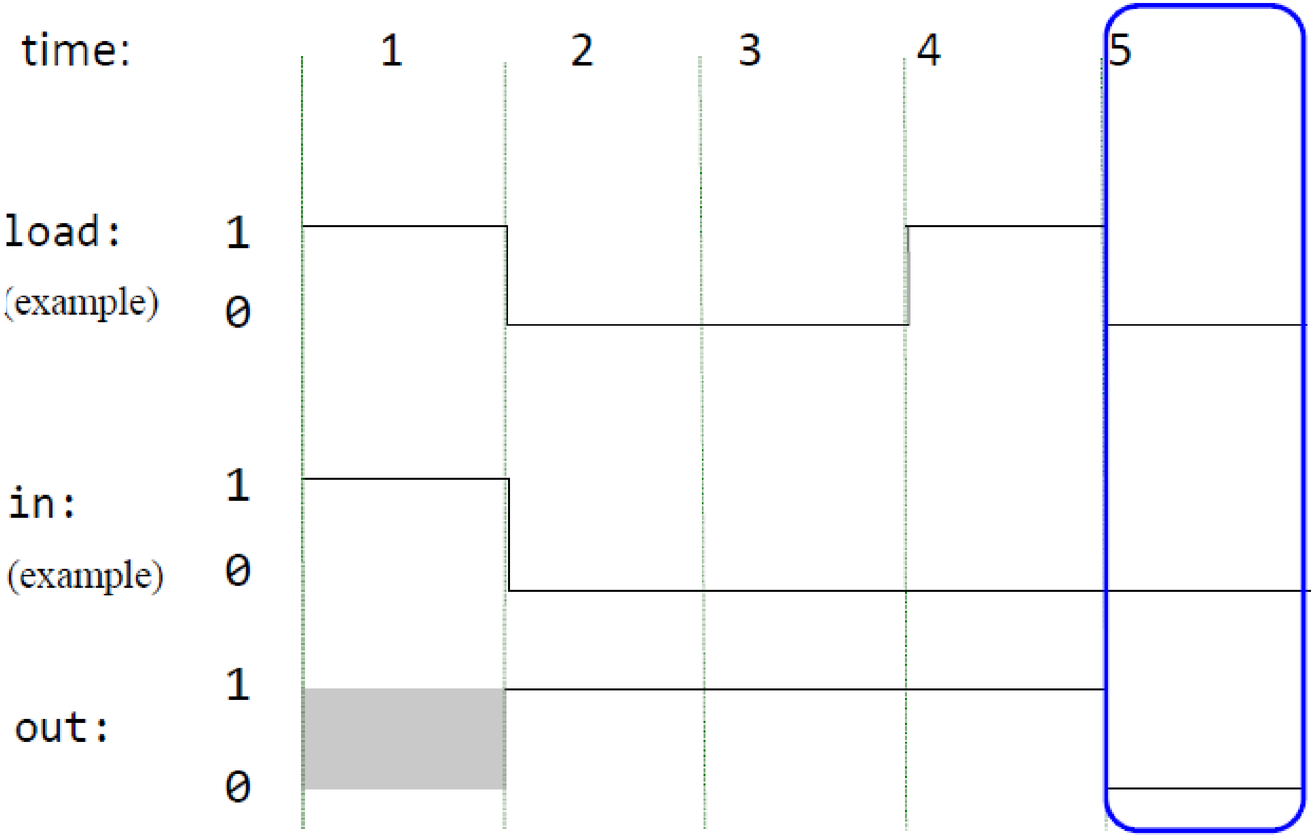
```
if load == 1
  out = in
else
  out = b
```



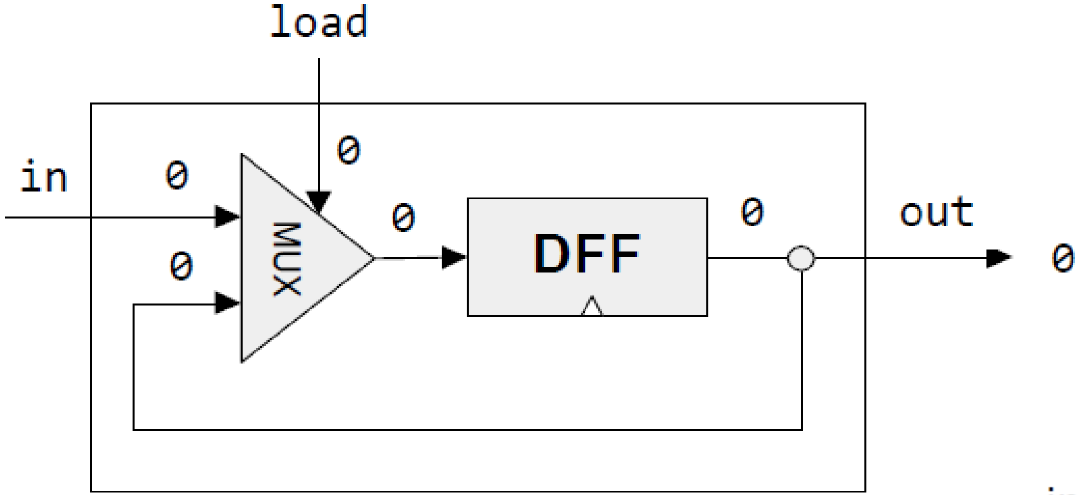


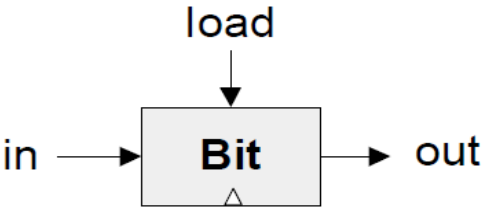


if load(t) then out(t+1) = in(t)
else out(t+1) = out(t)

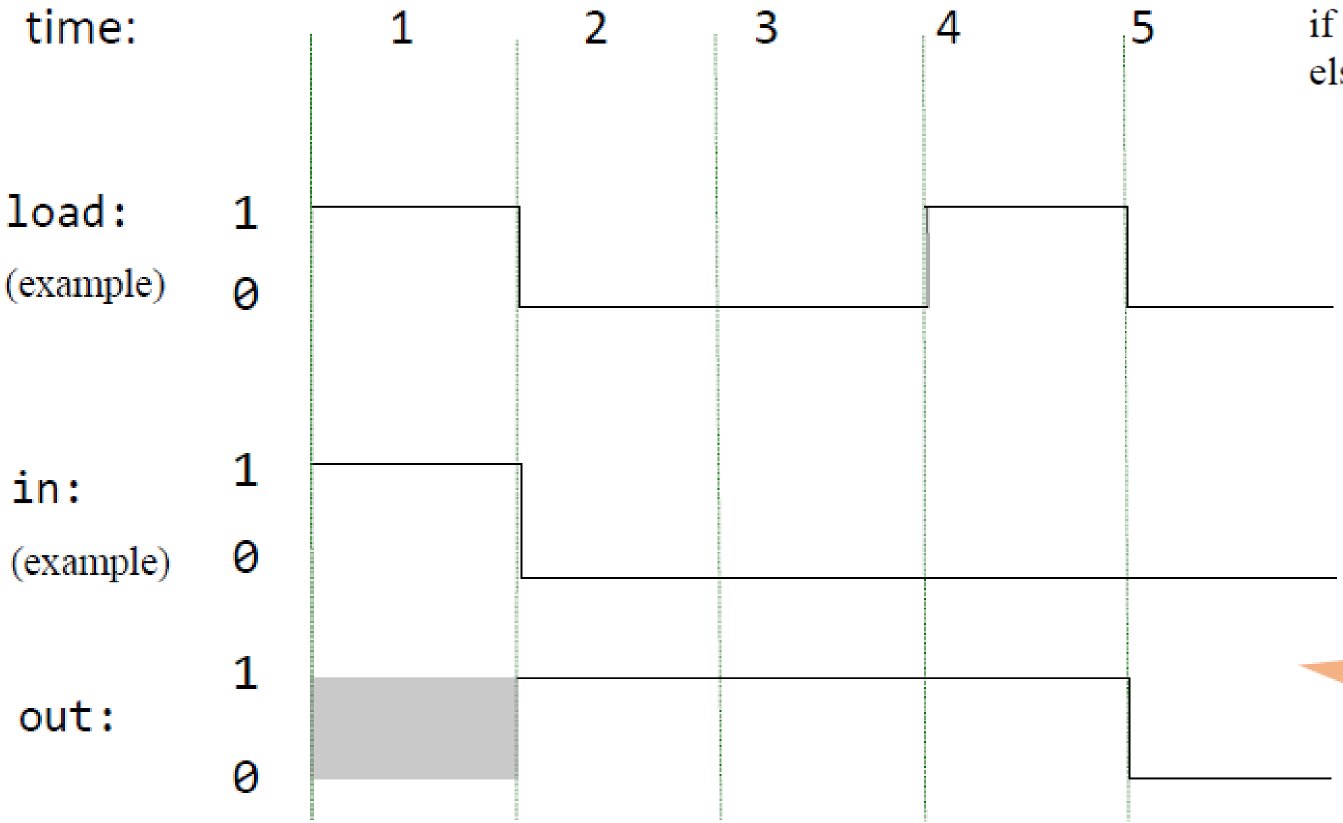


1-bit Register



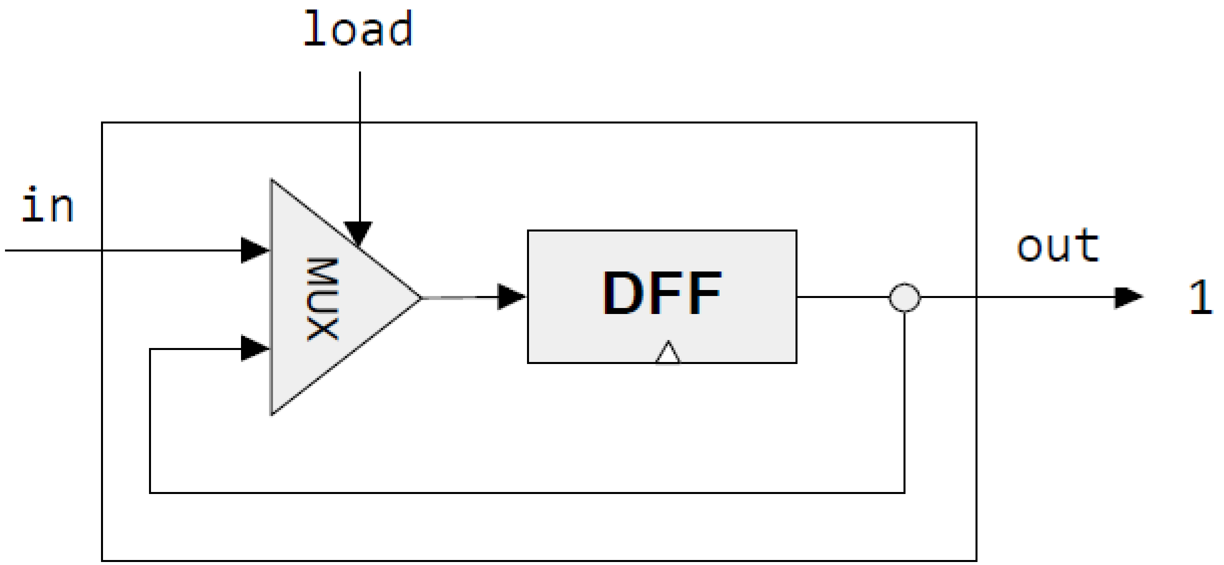


if load(*t*) then out(*t*+1) = in(*t*)
else out(*t*+1) = out(*t*)



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

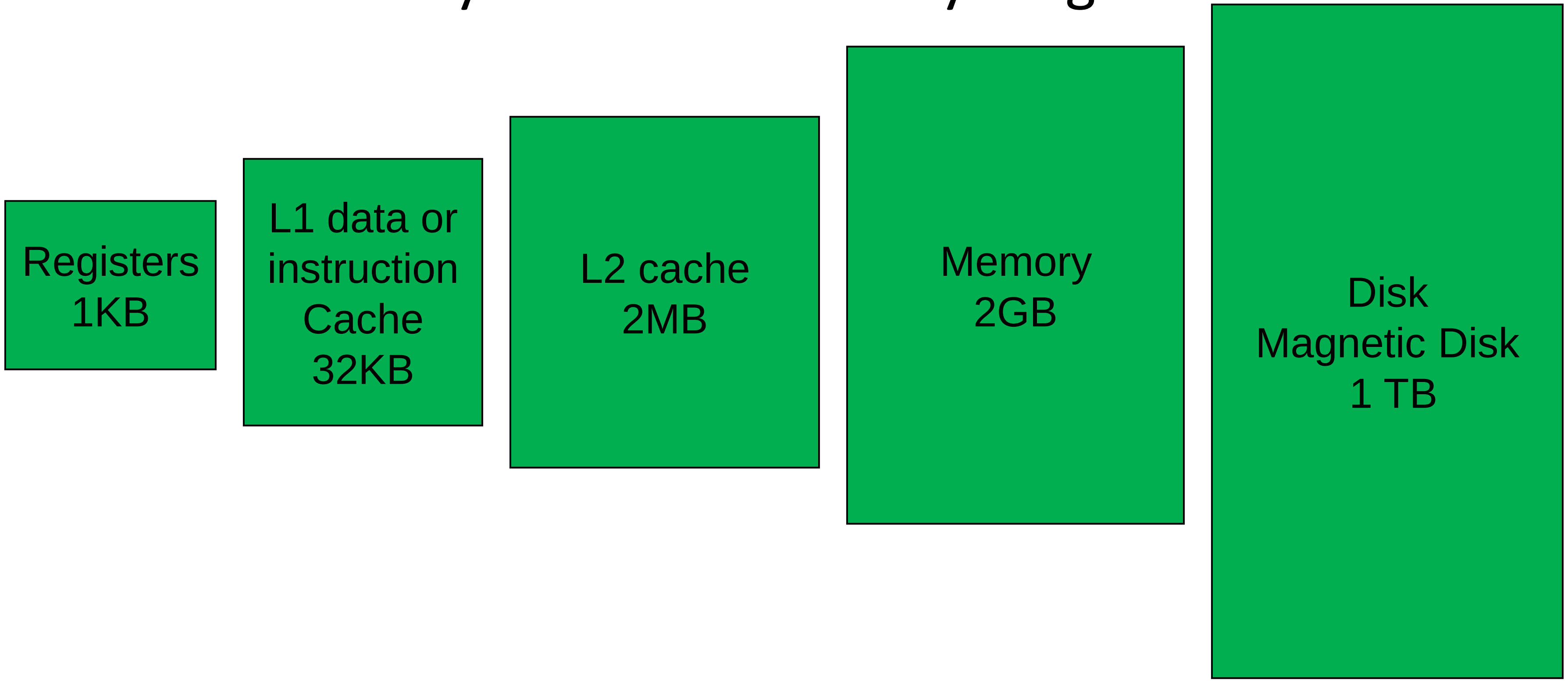
1-bit Register in HDL



```
/**
* 1-bit register.
* If load[t]=1 then out[t+1] = in[t]
*             else out does not change (out[t+1]=out[t])
*/
CHIP Bit {
IN in, load;
OUT out;
PARTS:
Mux (a=dffOut, b=in, sel=load, out=muxOut);
DFF (in=muxOut, out=dffOut, out=out);
}
```

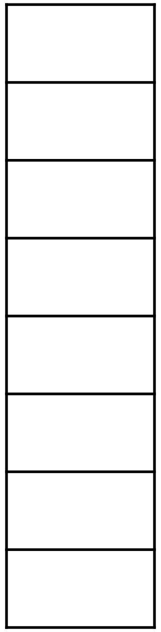
Memory Hierarchy

- As it goes further, capacity and latency increase
- Once we have basic ability to represent words, we can proceed to build memory banks of arbitrary length



Memory (RAM)

- Random-Access Memory (RAM)
  - Traditionally packaged as a chip
  - Basic storage unit is normally a cell (one bit per cell)
  - Multiple RAM chips form a memory
- RAM should be able to access randomly chosen words, with no restriction in the order in which they are accessed
  - Assign each word in the n-register RAM a unique address (an integer between 0 to n-1)
  - Given an address j, the individual register with the address j can be selected

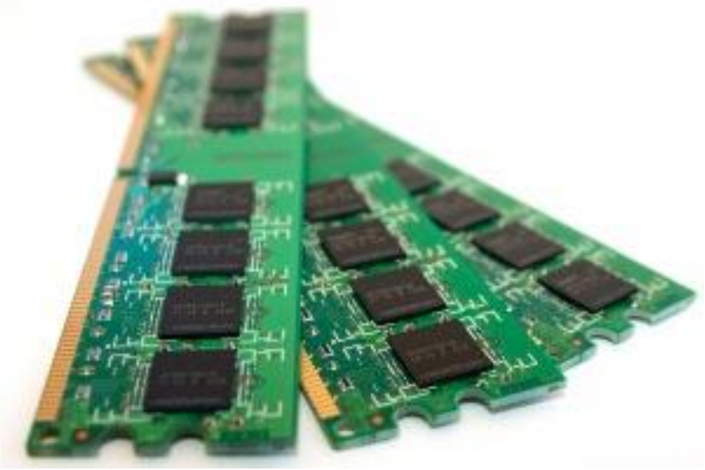


RAM

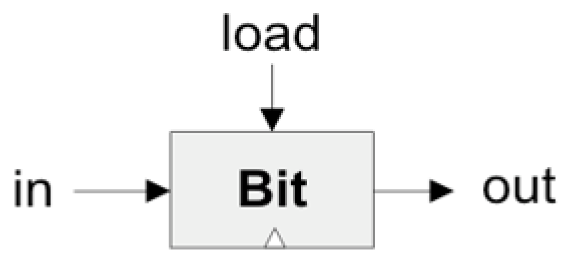
- Volatile Memory: only maintains its data while the device is powered

- Random-Access Memory has:
  - A data input
  - An address input
  - A load bit (read is load=0, write is load=1)

- Types of RAM:
  - SRAM (static RAM)
    - Generally faster and requires less dynamic power
    - More expensive to produce
    - Often used as cache memory for the CPU in modern computers
  - DRAM (dynamic RAM)
    - Slower
    - Less expensive to produce

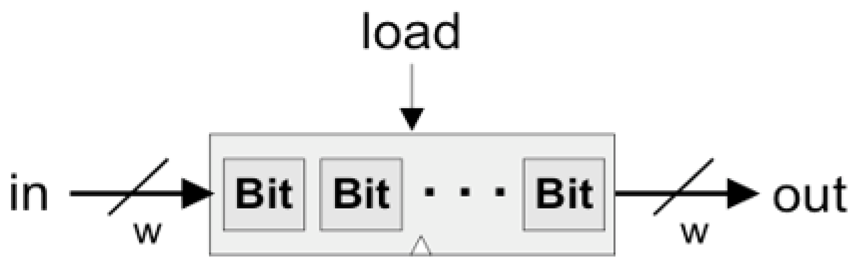


Multi-bit Registers



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

1-bit register

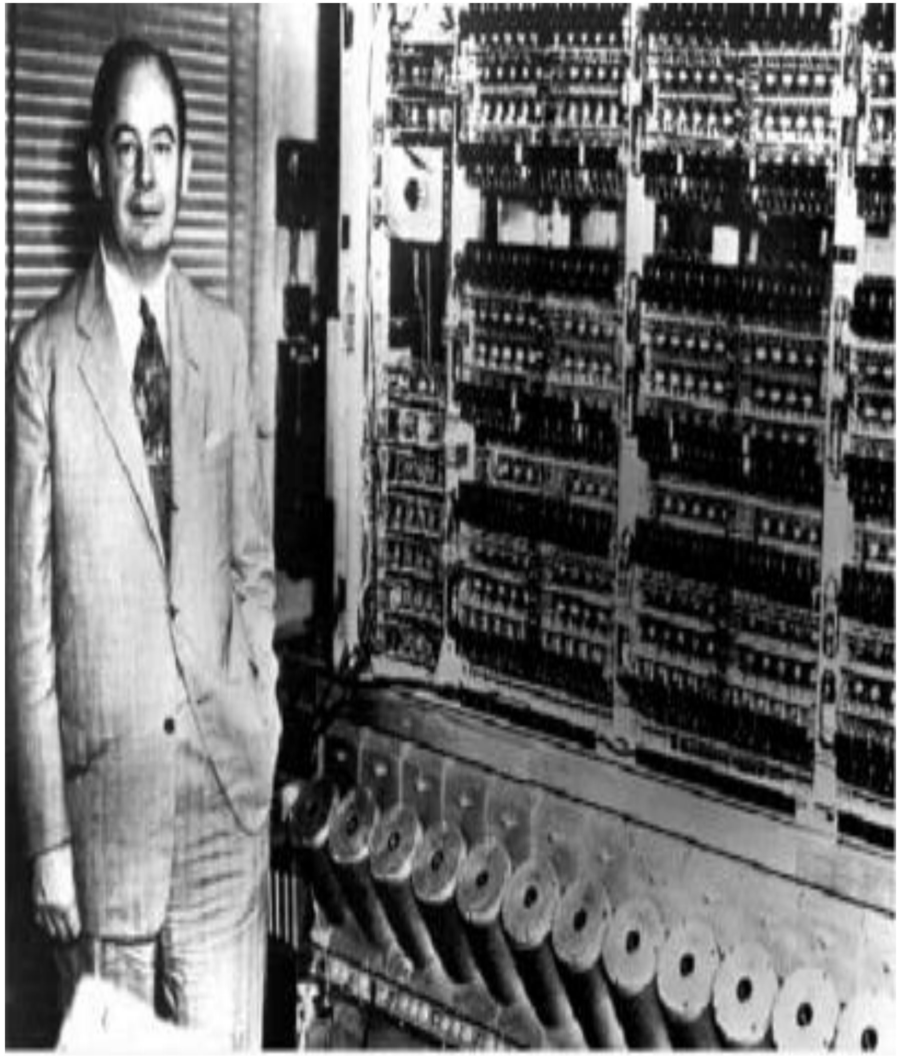


if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

w-bit register

- Word width (*w*):
  - 16 bit, 32 bit, 64 bit, …(doesn’t matter, once you understand 1 bit you understand all others)
  - A *w* bit register can be created from an array of *w* 1 bit registers

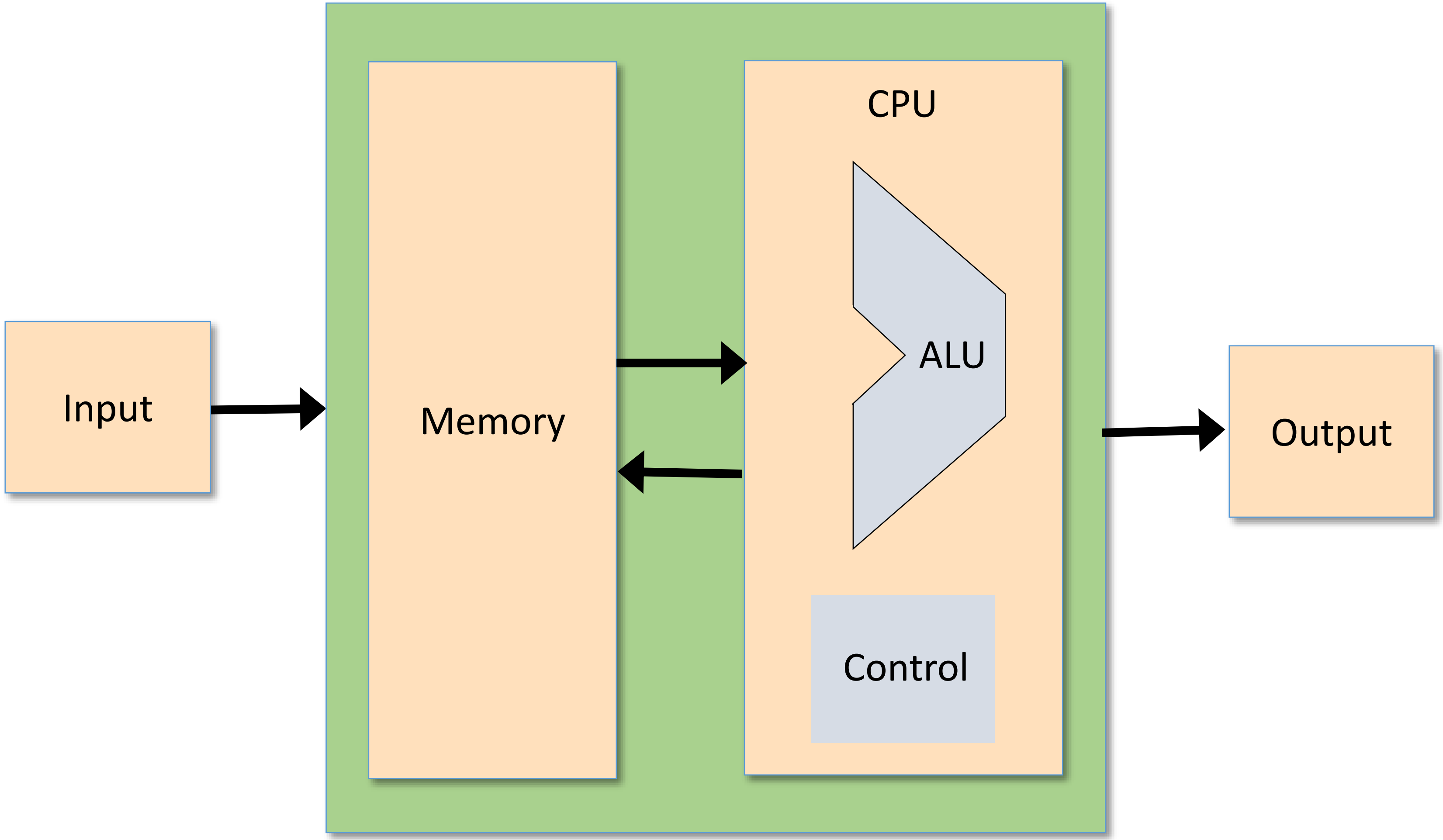
The IAS (von Neumann) Machine



- In 1945, John von Neumann wrote a report on the stored program concept, at Institute for Advanced Study (IAS), in Princeton

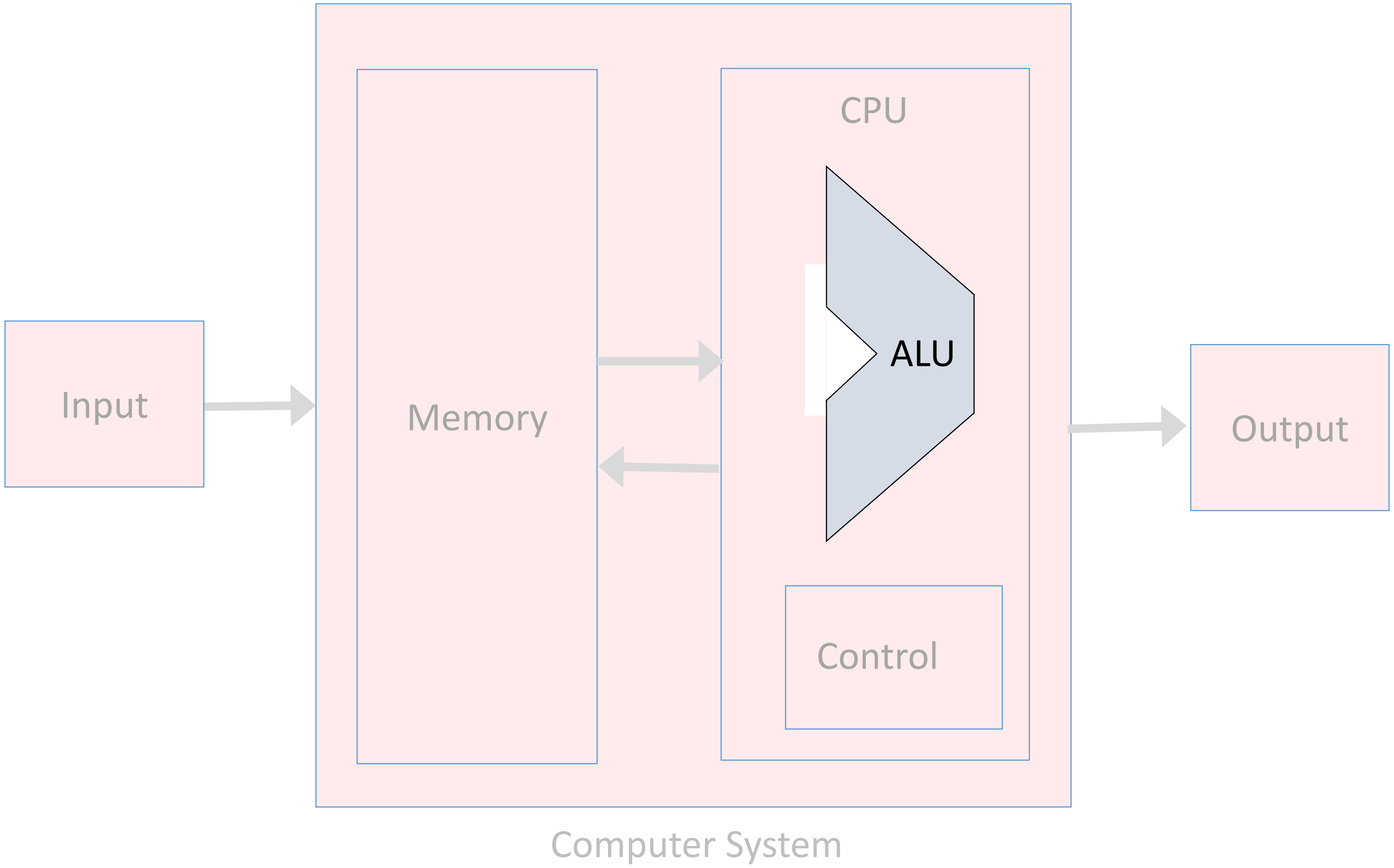
- Almost all of today’s computers have the same general structure as the IAS - referred to as von Neumann machines
    - A memory storing both instructions and data
    - A processing unit performing arithmetic and logical operations
    - A control unit interpreting instructions from memory and executing

Von Neumann Architecture



Computer System

The Arithmetic Logical Unit

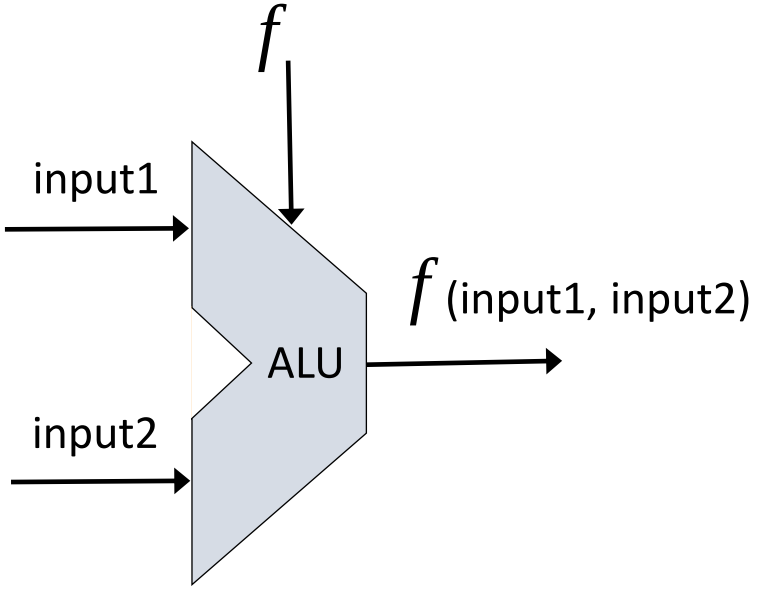


Arithmetic Logical Unit

- **A combinational circuit** that performs arithmetic and bitwise operations on integers represented as binary numbers.
- Input the **data** and some **code for the operation**
- Output will be some **data** and any **additional information**
- ALUs perform simple functions, because of this they can be executed at high speeds (i.e., very short propagation delays)

The Arithmetic Logical Unit

The ALU computes a function on the two inputs, and outputs the result



f: one out of a family of pre-defined arithmetic and logical functions

- ❑ Arithmetic functions: integer addition, multiplication, division, ...
- ❑ logical functions: And, Or, Xor, …

Which functions should the ALU perform?
A hardware / software tradeoff.

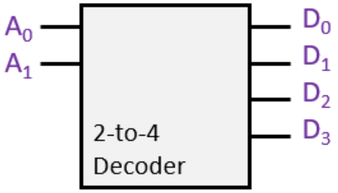
Example: 8-bit ALU using Logic Gates

- 4 operations (3 logical operations and 1 arithmetical operation)

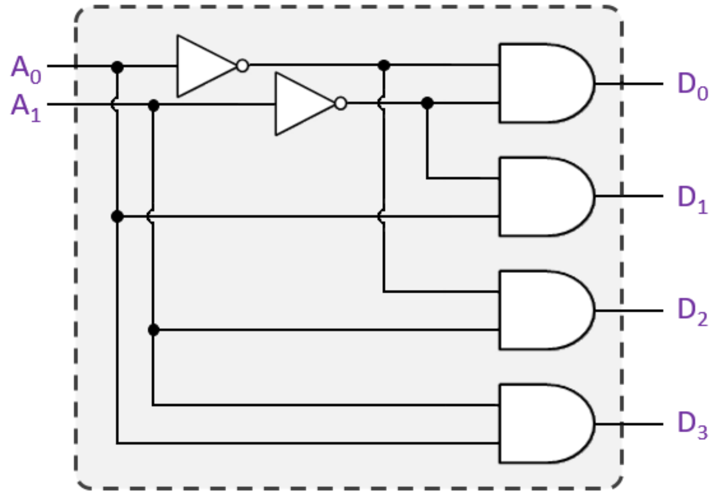
F_0F_1	Output
00	NOT A
01	A OR B
10	A AND B
11	A + B

- Use a 2-to-4 binary decoder that can decode 2-bit instructions
- The Logic unit will apply NOT/OR/AND gates
- The Arithmetic unit will use a full adder

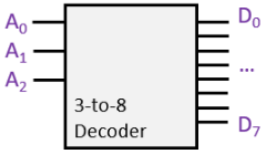
Binary Decoder



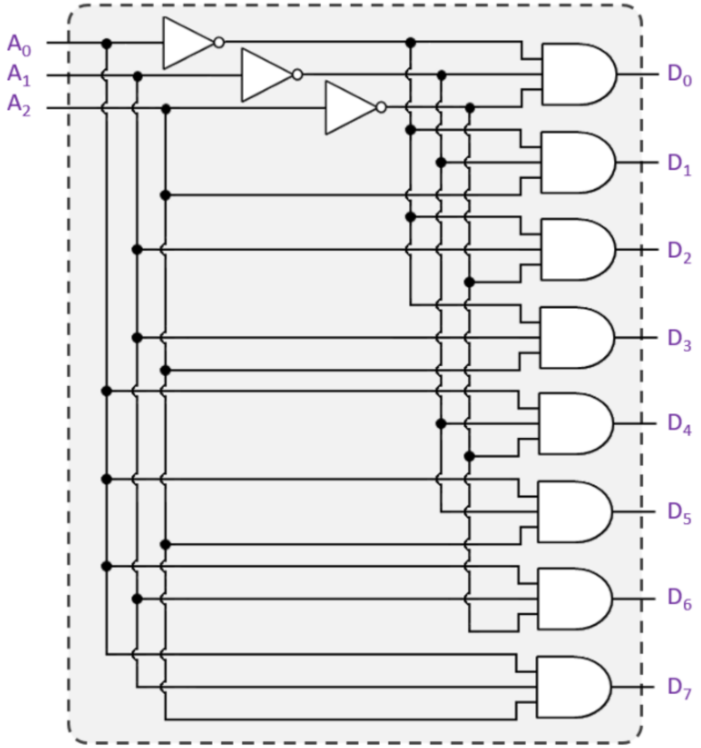
INPUT A_1A_0	OUTPUT $D_3D_2D_1D_0$
00	000**1**
01	00**1**0
10	0**1**00
11	**1**000



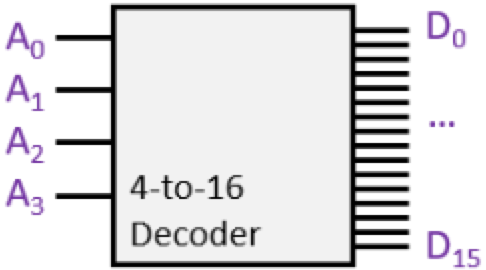
2-to-4 Binary Decoder Logic Gates Diagram



INPUT $A_2A_1A_0$	OUTPUT $D_7D_6D_5D_4D_3D_2D_1D_0$
000	0000000**1**
001	000000**1**0
010	00000**1**00
011	0000**1**000
100	000**1**0000
101	00**1**00000
110	0**1**000000
111	**1**0000000

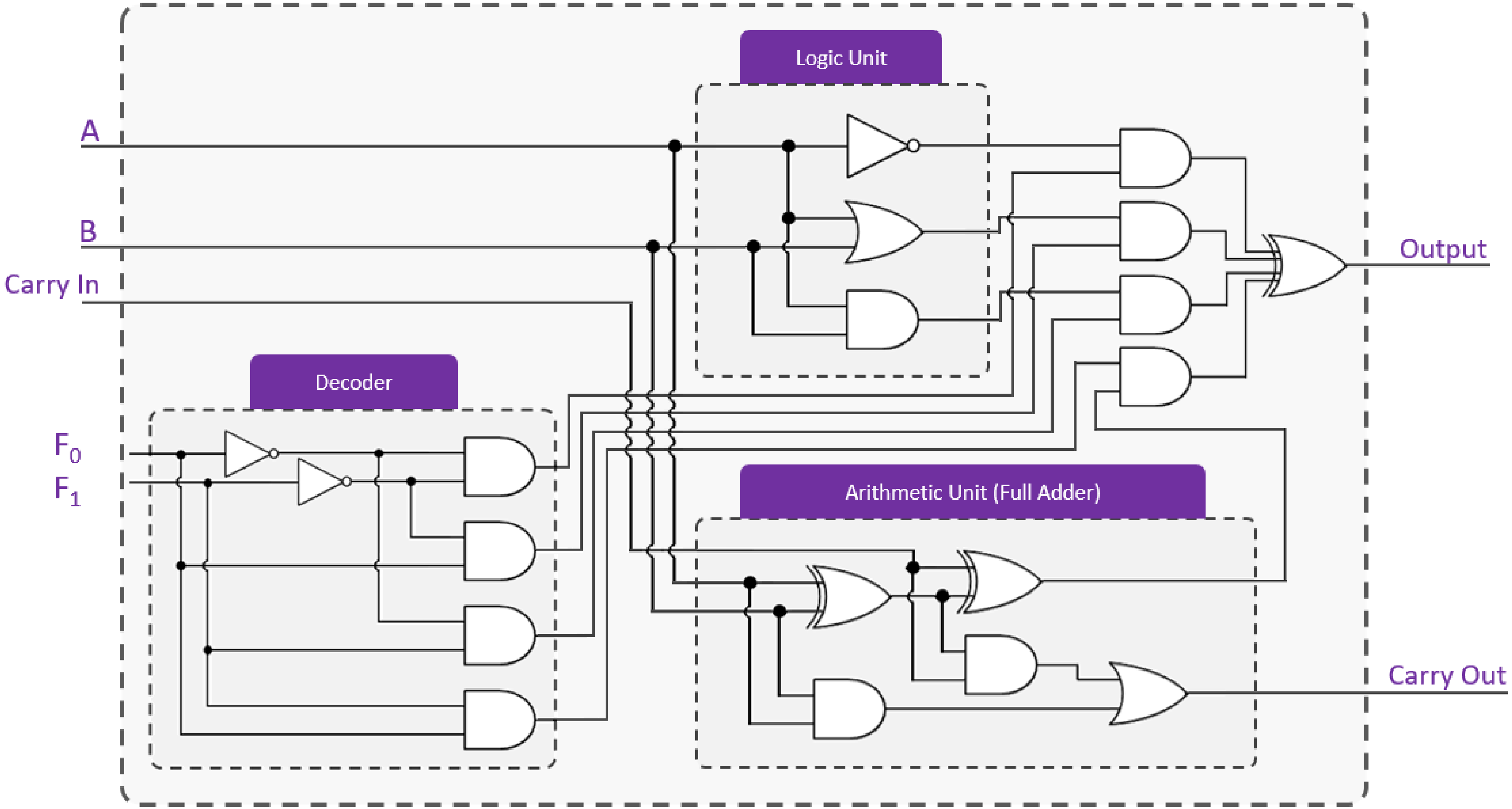


3-to-8 Binary Decoder Logic Gates Diagram



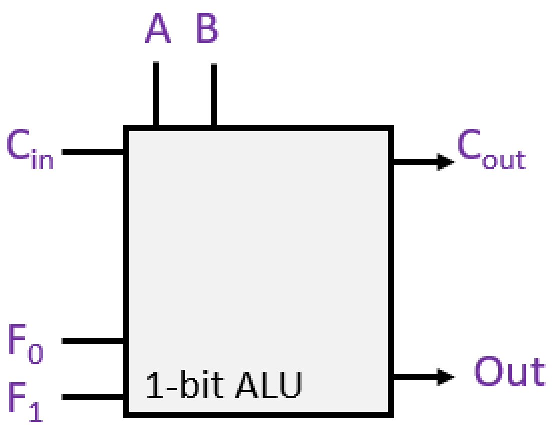
INPUT $A_3A_2A_1A_0$	OUTPUT D_{15}……D_0
0000	000000000000000**1**
0001	00000000000000**1**0
0010	0000000000000**1**00
0011	000000000000**1**000
0100	00000000000**1**0000
0101	0000000000**1**00000
0110	000000000**1**000000
0111	00000000**1**0000000
1000	0000000**1**00000000
1001	000000**1**000000000
1010	00000**1**0000000000
1011	0000**1**00000000000
1100	000**1**000000000000
1101	00**1**0000000000000
1110	0**1**00000000000000
1111	**1**000000000000000

1-bit ALU

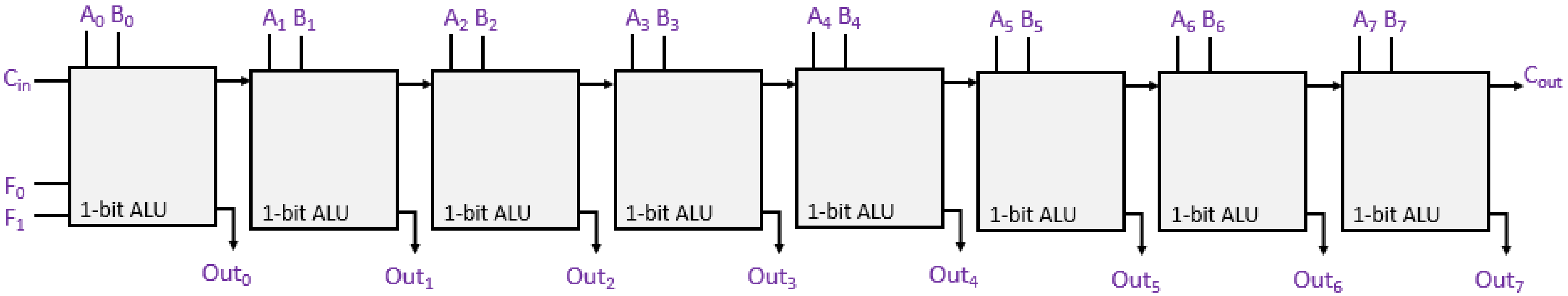

Logic Unit
A
B
Carry In
Output
Decoder
F_0
F_1
Arithmetic Unit (Full Adder)
Carry Out

8-bit ALU

- We can simplify the 1-bit ALU representation as:

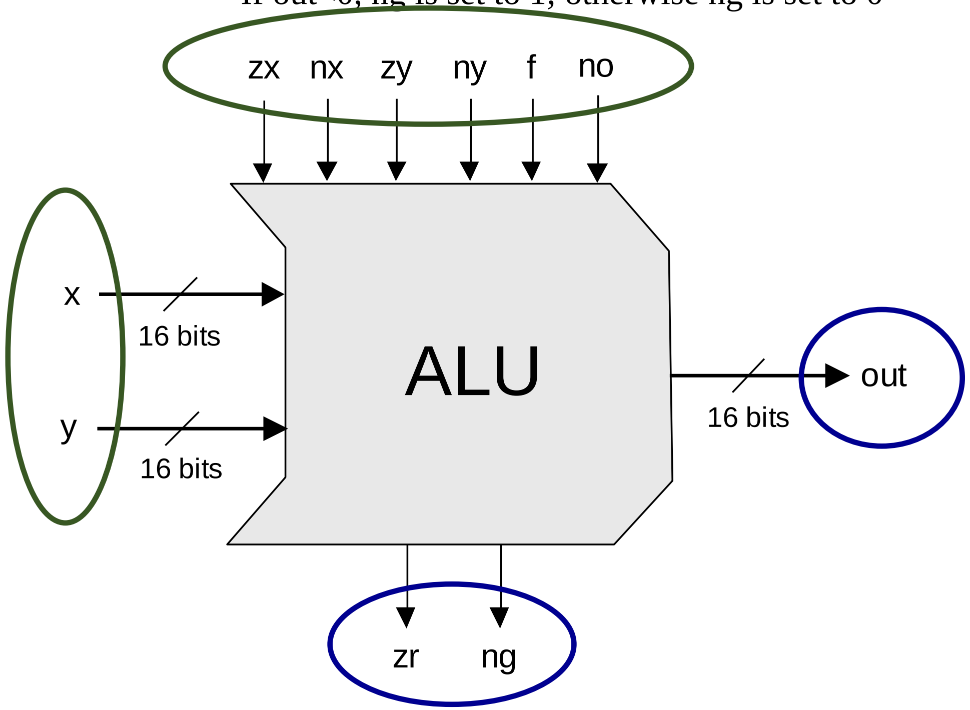


- By connecting eight 1-bit ALUs together, we obtain an 8-bit ALU:



The Hack ALU

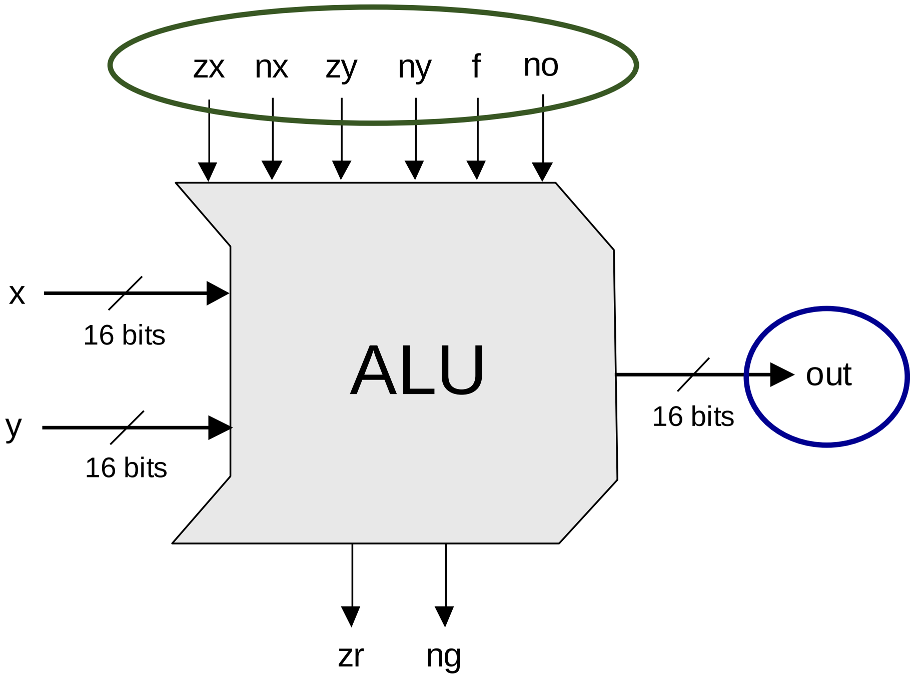
- Operates on two 16-bit, two’s complement values
- Outputs a 16-bit, two’s complement value
- Which function to compute is set by six 1-bit inputs
- Computes one out of a family of 18 functions
- Also outputs two 1-bit values
  - if the ALU output is 0, zr is set to 1; otherwise zr is set to 0
  - If out<0, ng is set to 1; otherwise ng is set to 0



out
0
1
-1
x
y
!x
!y
-x
-y
x+1
y+1
x-1
y-1
x+y
x-y
y-x
x&y
x|y

The Hack ALU

To cause the ALU to compute a function, set the control bits to one of the binary combinations listed in the table.



control bits

zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

ALU

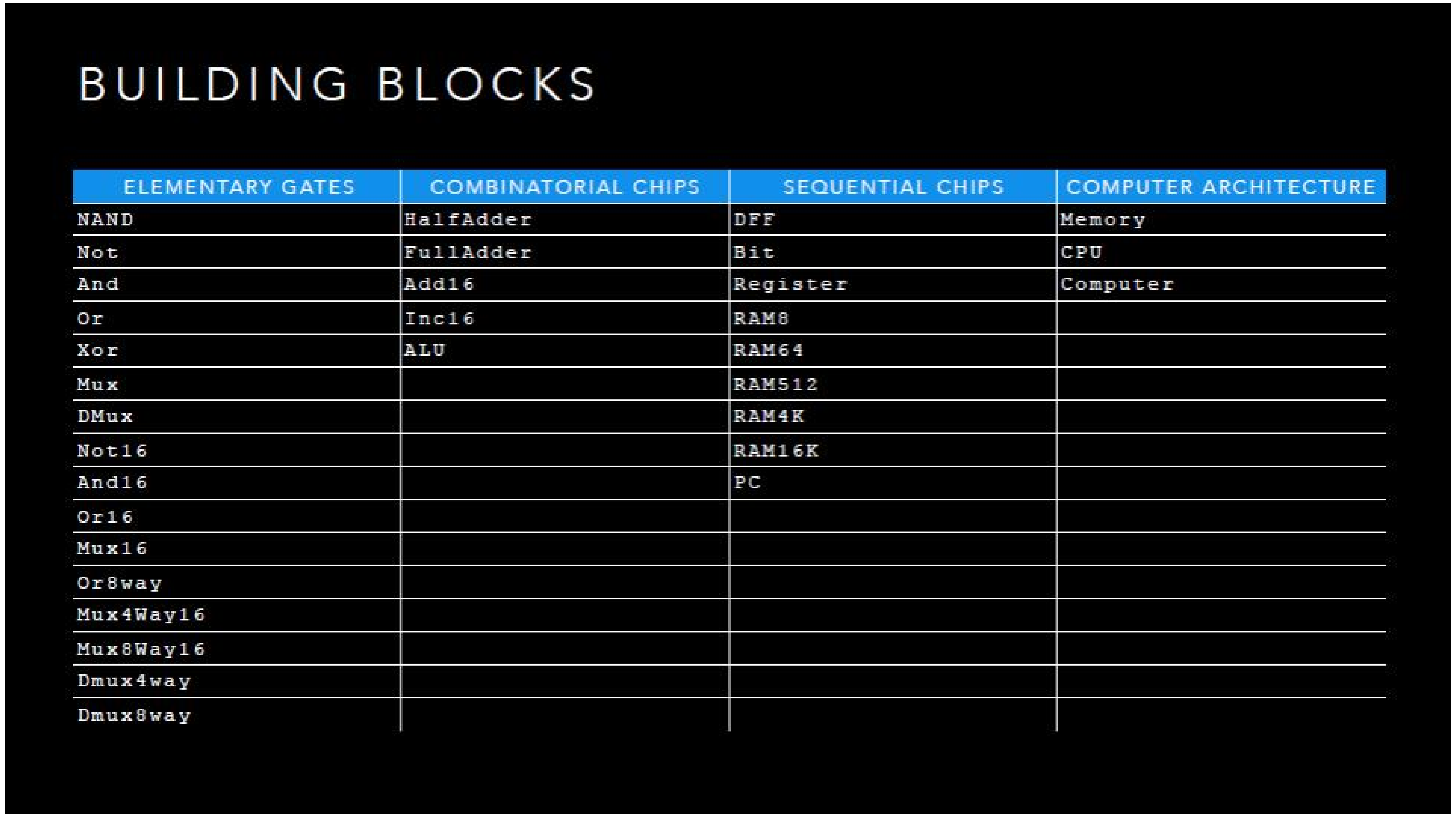
- Each of the 6 control bits instruct the ALU to carry out a certain elementary operation
- Taken together, the combined effects of these operations causes the ALU to compute a variety of useful functions
- 2^6 = 64 different functions, but only 18 are documented in the ALU table
- This “Hack” ALU is carefully designed by the authors of the text book to do what it’s supposed to do

Custom ALUs

- We can make ALUs to perform the specialized set of tasks
- Example
  - Multiply x by y
  - Divide x by 2
  - Calculate an factorial
  - Detect overflow
  - Decision making (eg. Is x bigger than y)

Summary

- Sequential Logic Circuit
  - Clock
  - Flip-Flop
- Sequential Chips
  - 1-bit register
- Arithmetic Logical Unit
  - Von Neumann Architecture
  - Hack ALU



BUILDING BLOCKS

ELEMENTARY GATES	COMBINATORIAL CHIPS	SEQUENTIAL CHIPS	COMPUTER ARCHITECTURE
NAND	HalfAdder	DFF	Memory
Not	FullAdder	Bit	CPU
And	Add16	Register	Computer
Or	Inc16	RAM8	
Xor	ALU	RAM64	
Mux		RAM512	
DMux		RAM4K	
Not16		RAM16K	
And16		PC	
Or16			
Mux16			
Or8way			
Mux4Way16			
Mux8Way16			
Dmux4way			
Dmux8way			