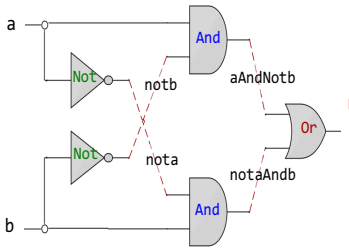
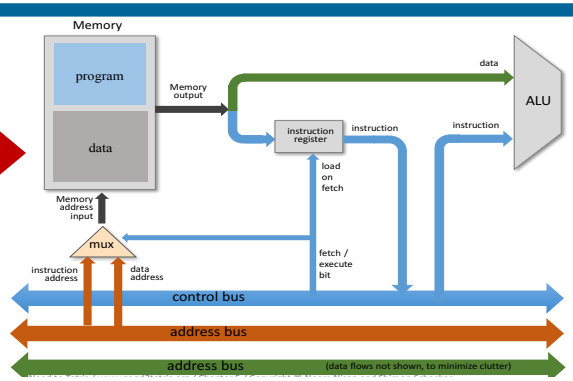
Computer Organization & Assembly Language Programming




```
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
  Not(in=a, out=nota);
  Not(in=b, out=notb);
  And(a=nota, b=b, out=w1);
  And(a=a, b=notb, out=w2);
  Or(a=w1, b=w2, out=out);
}
```




```
@R1
D=M
@temp
M=D
```

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
```

Lecture # 11

Design of Registers

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

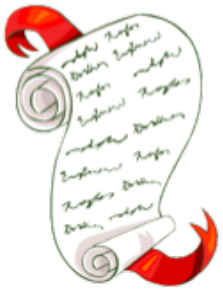
```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

Slides of first half of the course are adapted from:
https://www.nand2tetris.org
Download s/w tools required for first half of the course from the following link:
https://drive.google.com/file/d/0B9c0BdDJz6XpZUh3X2dPR1o0MUE/view

Instructor: Muhammad Arif Butt, Ph.D.

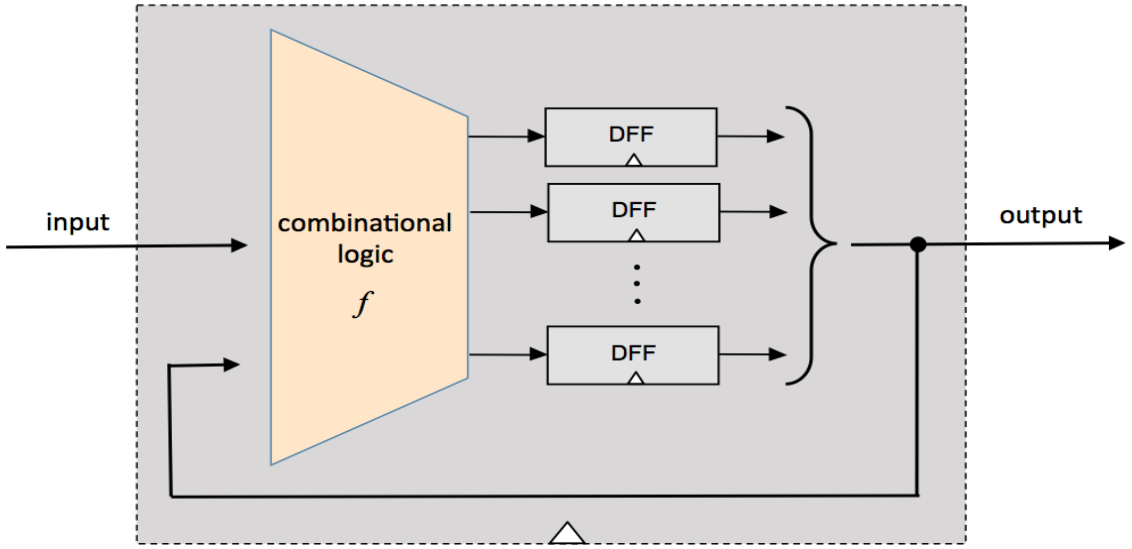
Today's Agenda

- Review of Sequential Chips
- What are Registers
- Design of 1-bit Register
- HDL for 1-bit Register
- Design of 16-bit Register
- HDL for 16-bit Register



Review of Sequential Chips

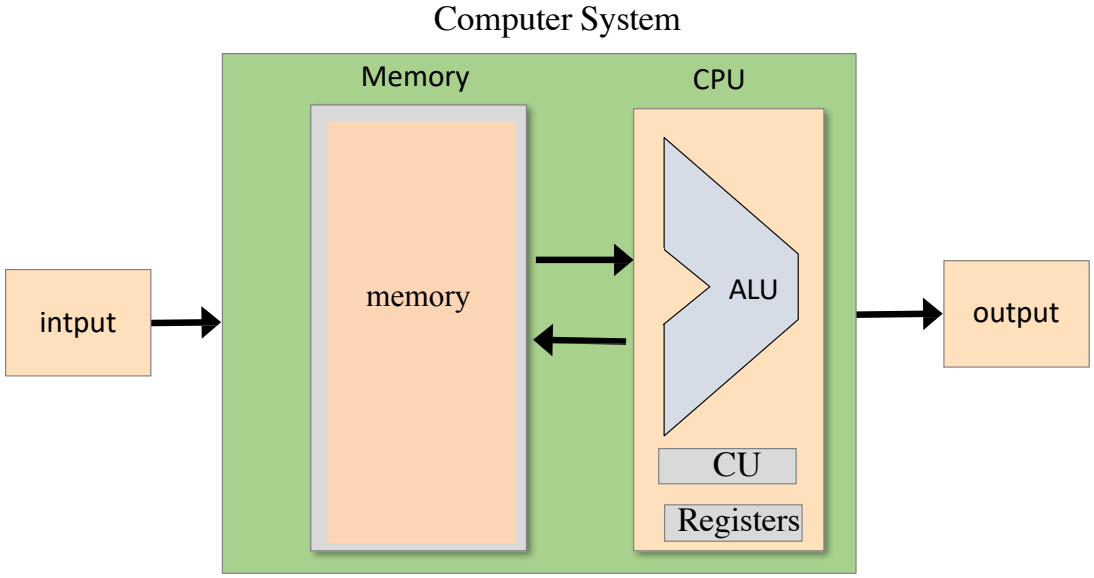
```
state(t) = f (state(t-1), input(t))
```



- Sequential chips are capable of maintaining state, and, optionally acting on the state, and on the current input
- The simplest and most elementary sequential chip is DFF, which maintain a state, i.e., the value of the input from the previous time unit
- Using DFF we can design registers, and using registers we can design RAM, whose state is the current values of all its registers. Given an address, the RAM emits the value of the selected register
- All combinational chips are constructed from NAND gates, while all sequential chips are constructed from DFF gates, and combinational chips

CPU Registers

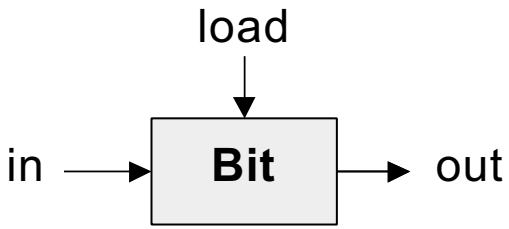
CPU Registers



- A register is a small memory place inside the CPU that may hold data, memory address or instruction
- The size of registers in a 64-bit computer must be of 64 bits
- In our Hack computer is a 16 bit computer, so the registers we are going to design will be of 16 bits
- There are several different classes of CPU registers which works in coordination with the computer memory to run operations efficiently. (More on it later)

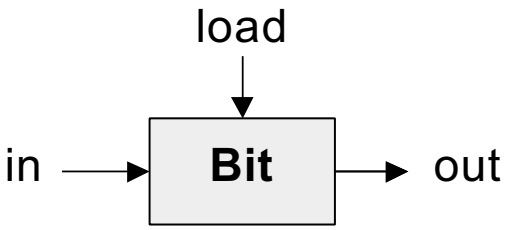
1-Bit Register

1-Bit Register API



- A single-bit register, which we call Bit, or binary cell, is designed to store a single bit of information (0 or 1)
- The chip interface diagram shows that it has two input pins and one output pin. The input pin carries a data bit, the load pin enables the cell for writes, and an output pin that emits the current state of the cell
- When you read the out pin of the binary cell, you will always get whatever is the state of the binary cell
- To write the binary cell, we set the load bit to 1, now what ever is there on the input bit will be stored inside the binary cell and will be available on the out pin in the next clock cycle
- When the load bit is zero, the chip keep remembering the last input that was loaded into it for infinity until a new load operation is performed

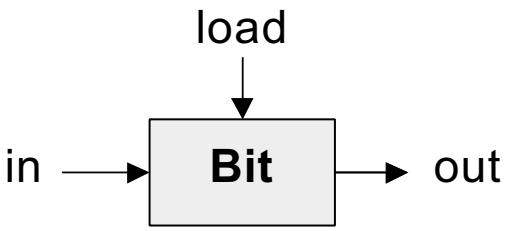
Sequential Chips: 1-Bit Register



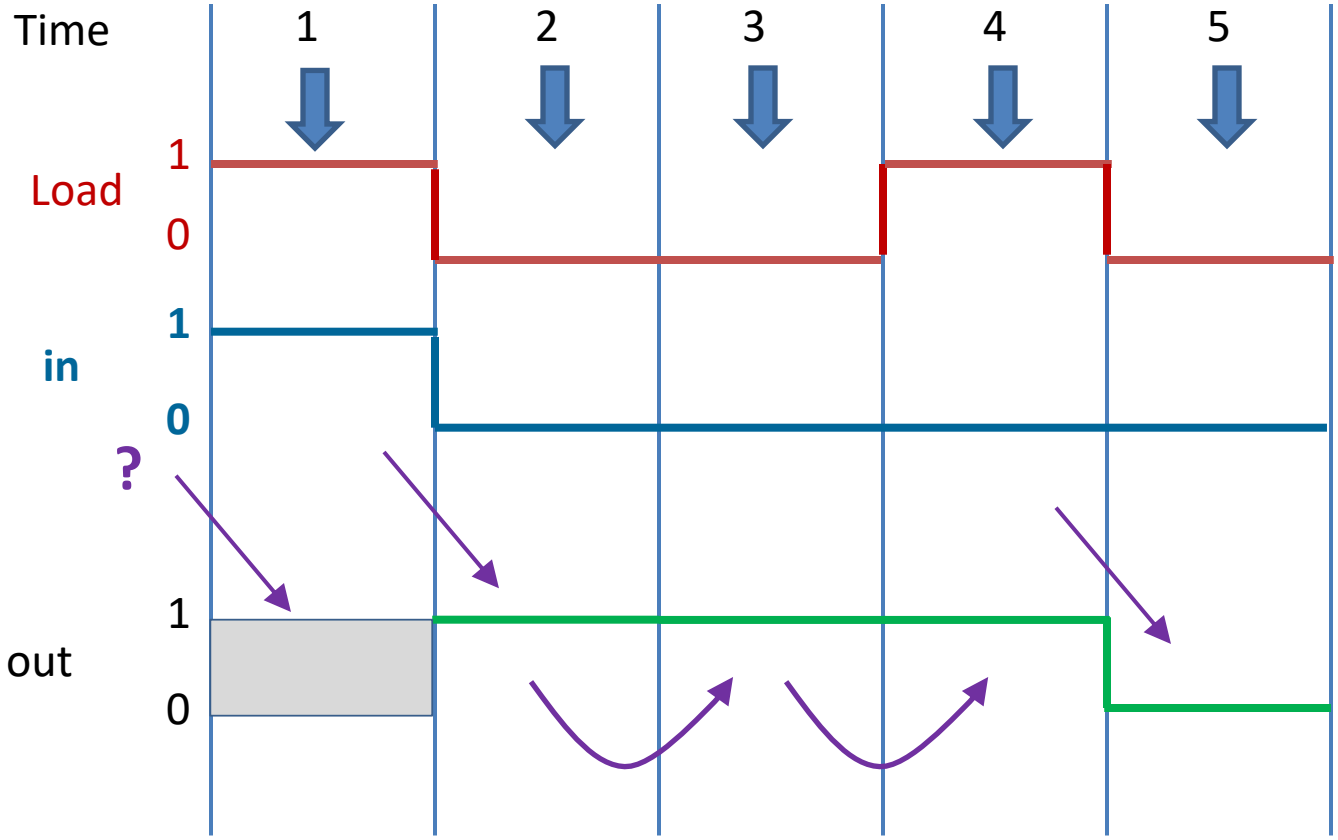
Chip name: Bit
Inputs: in, load
Outputs: out
Function:
```
If load(t) then
   out(t+1) = in(t)
else
   out(t+1) = out(t)
```

- Goal: Remember an input bit forever, until requested to load a new value
- More accurately:
  - Stores a bit until...
  - Instructed to load, and store, another bit

Sequential Chips: 1-Bit Register

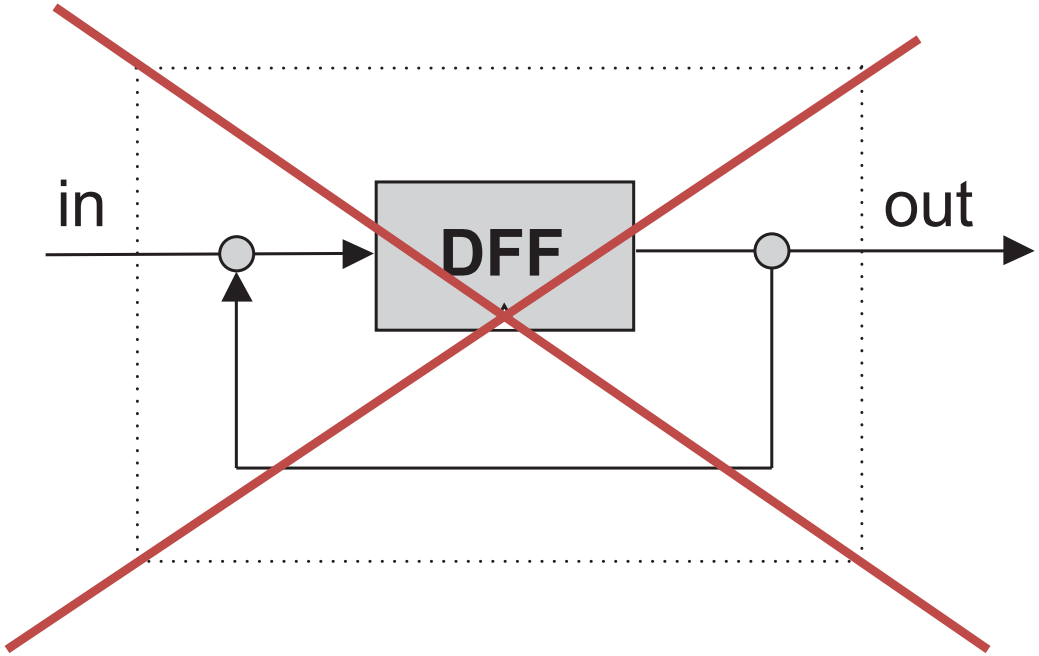


Chip name: Bit
Inputs: in, load
Outputs: out
Function:
```
If load(t) then
   out(t+1) = in(t)
else
   out(t+1) = out(t)
```

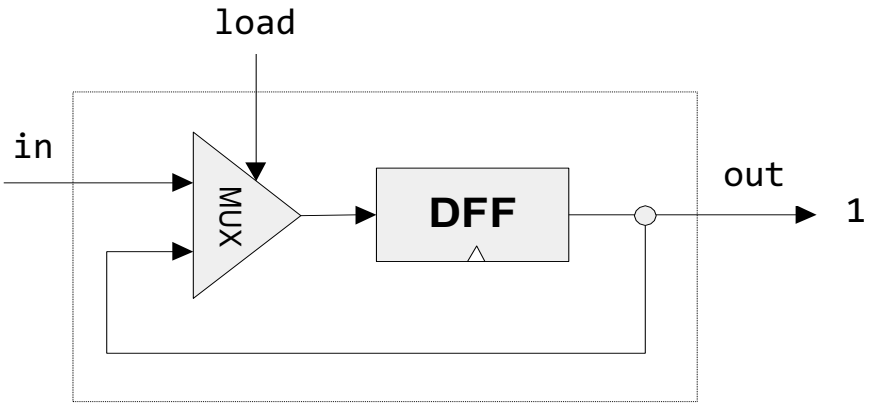


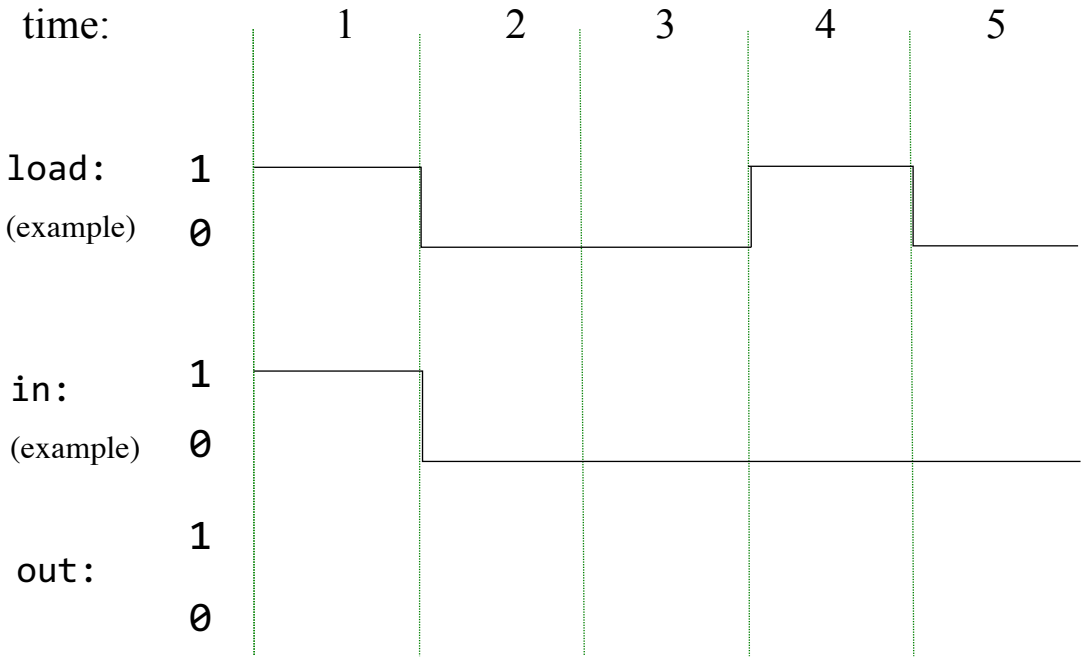
1-Bit Register Implementation

1-Bit Register Implementation

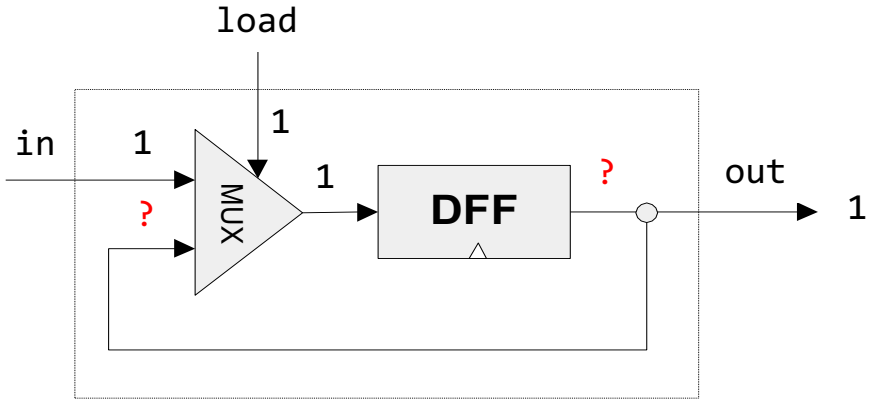


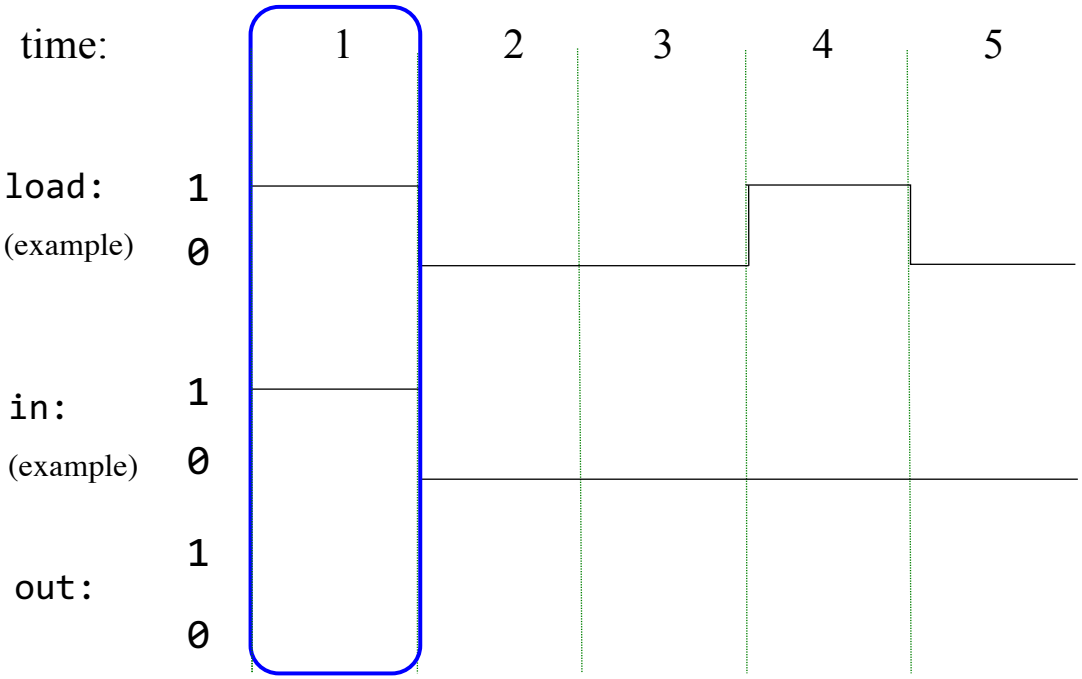
1-Bit Register Implementation



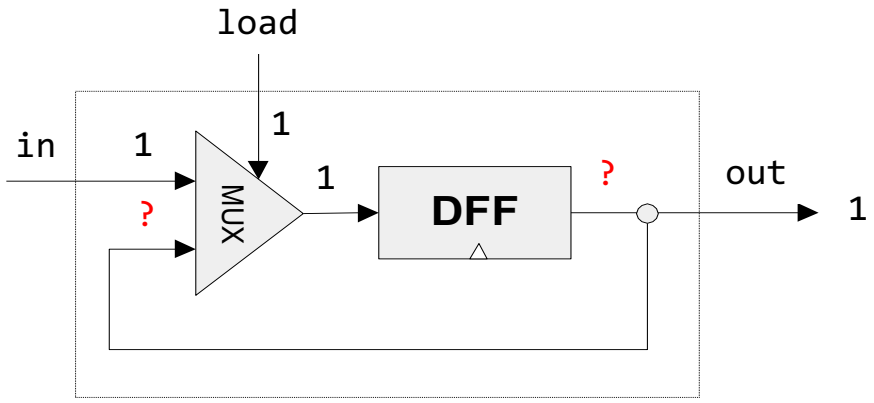


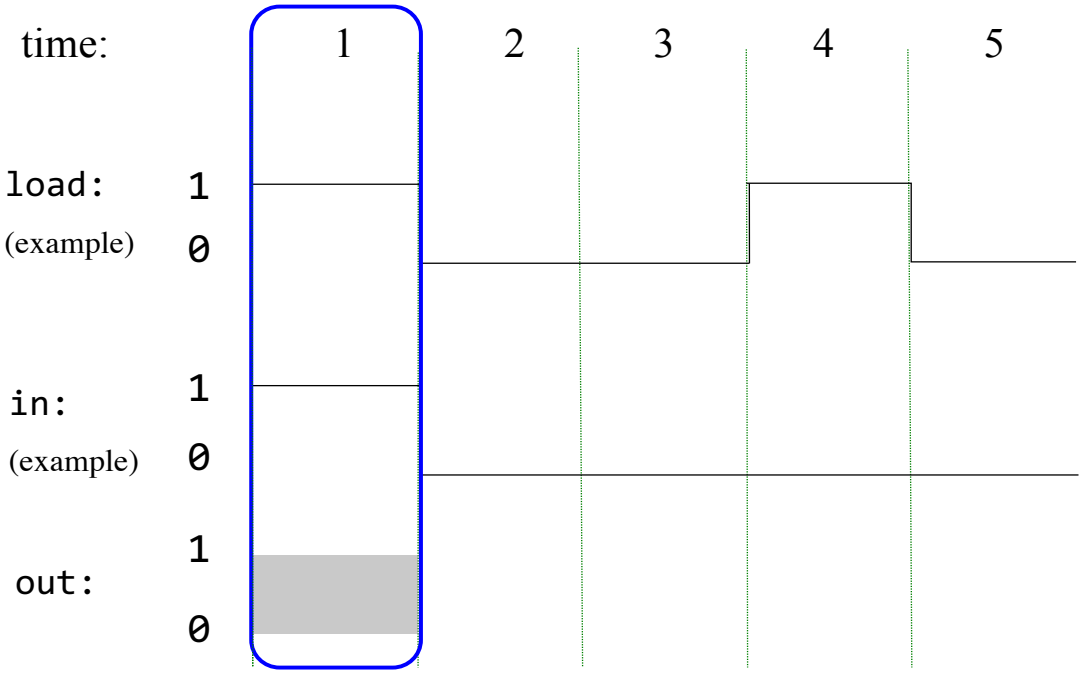
1-Bit Register Implementation



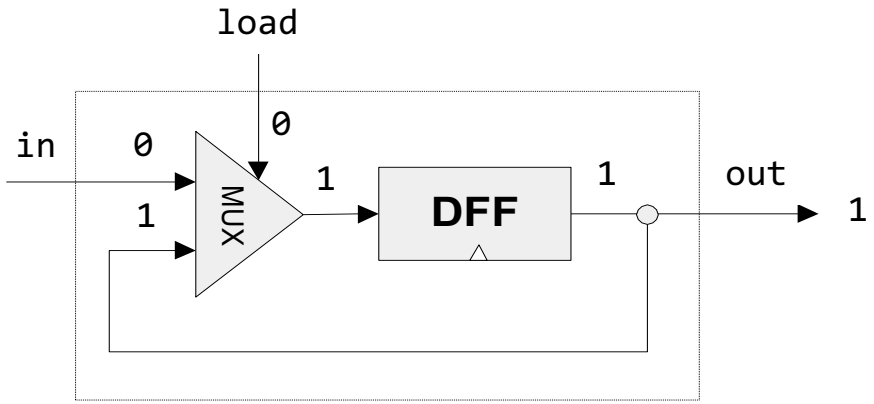


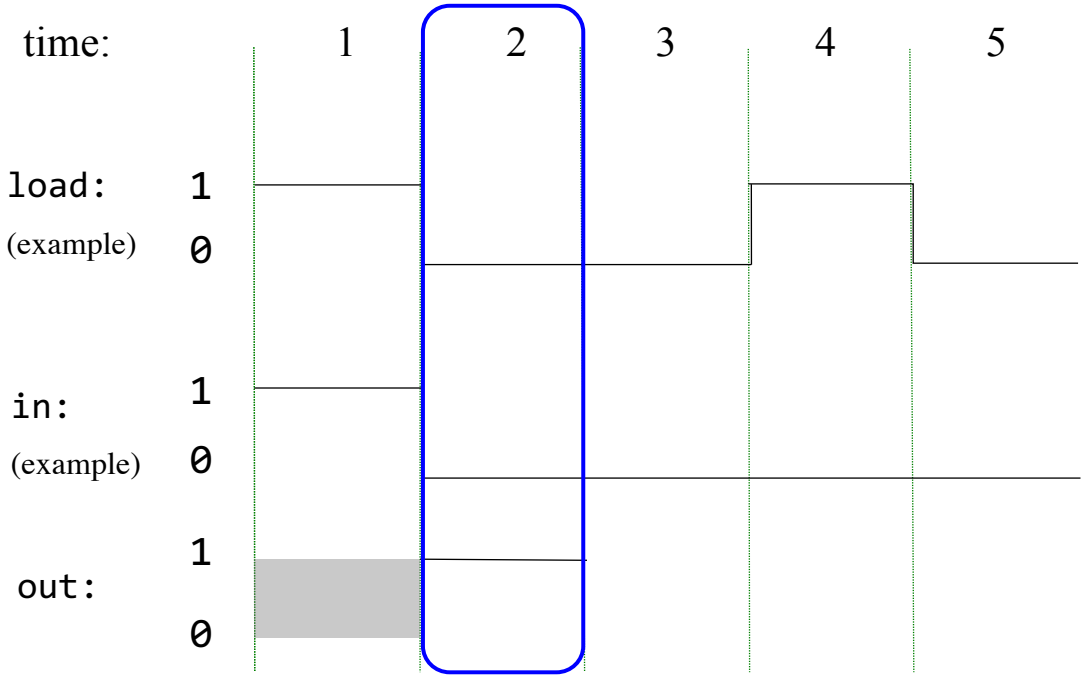
1-Bit Register Implementation



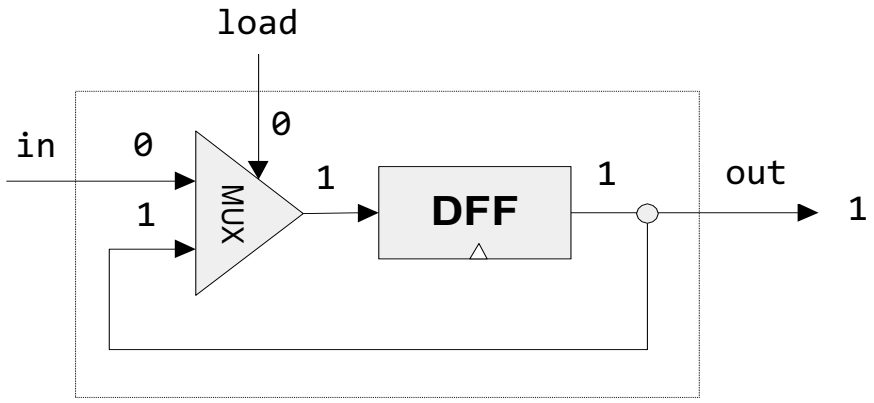


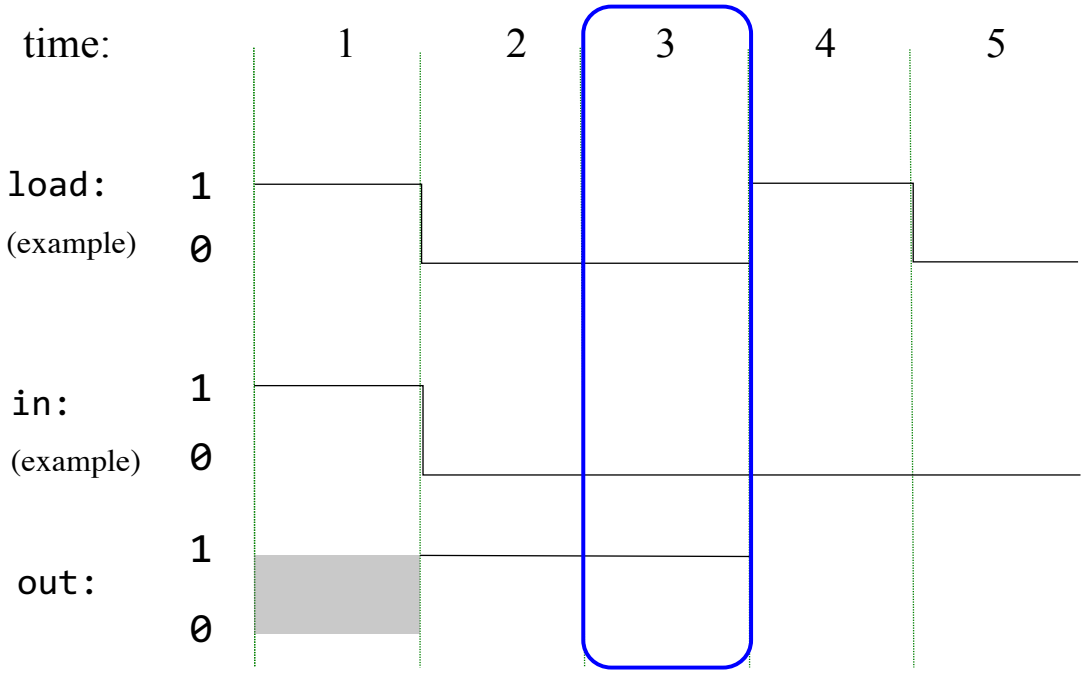
1-Bit Register Implementation



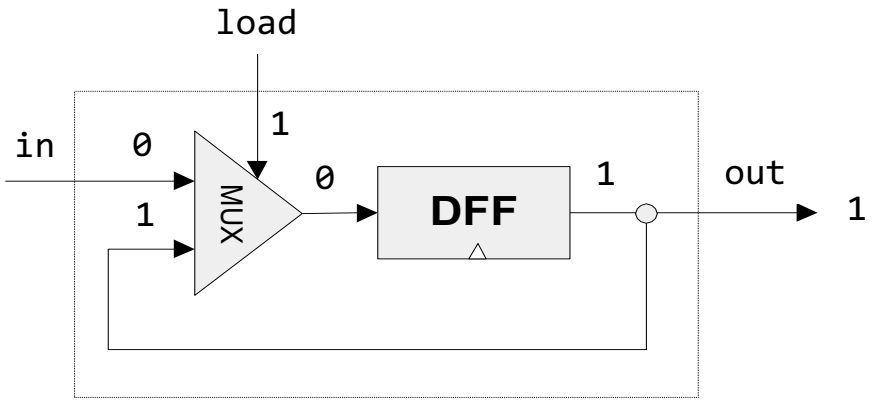


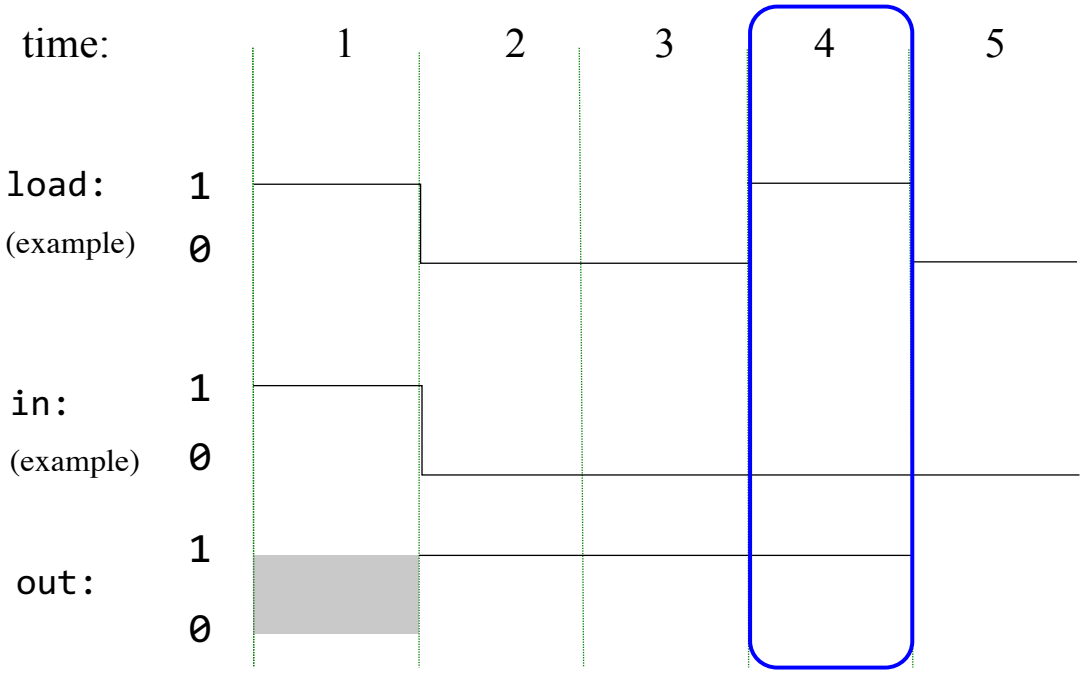
1-Bit Register Implementation



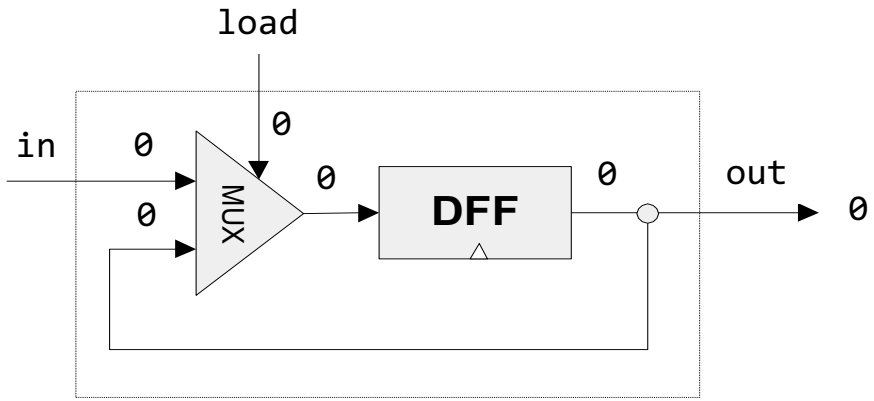


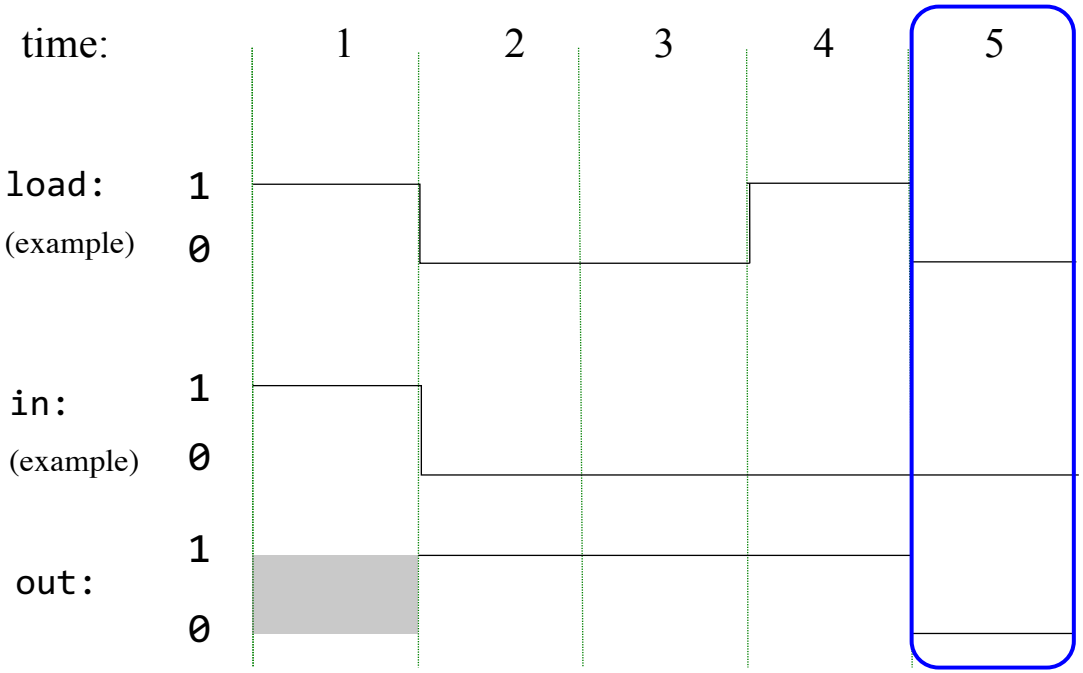
1-Bit Register Implementation



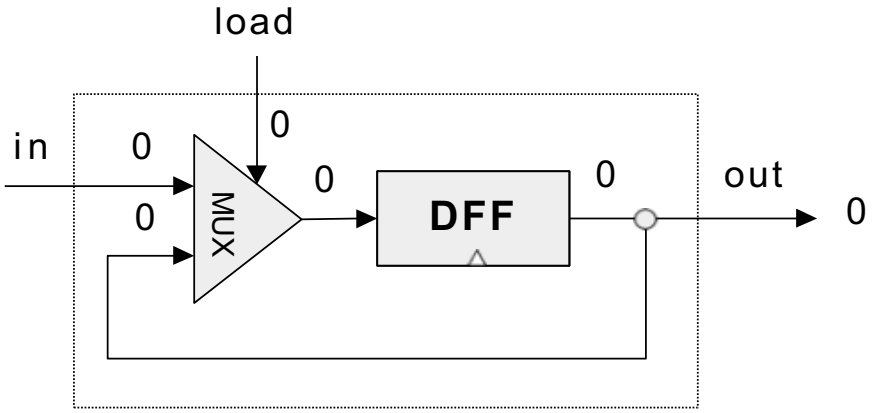


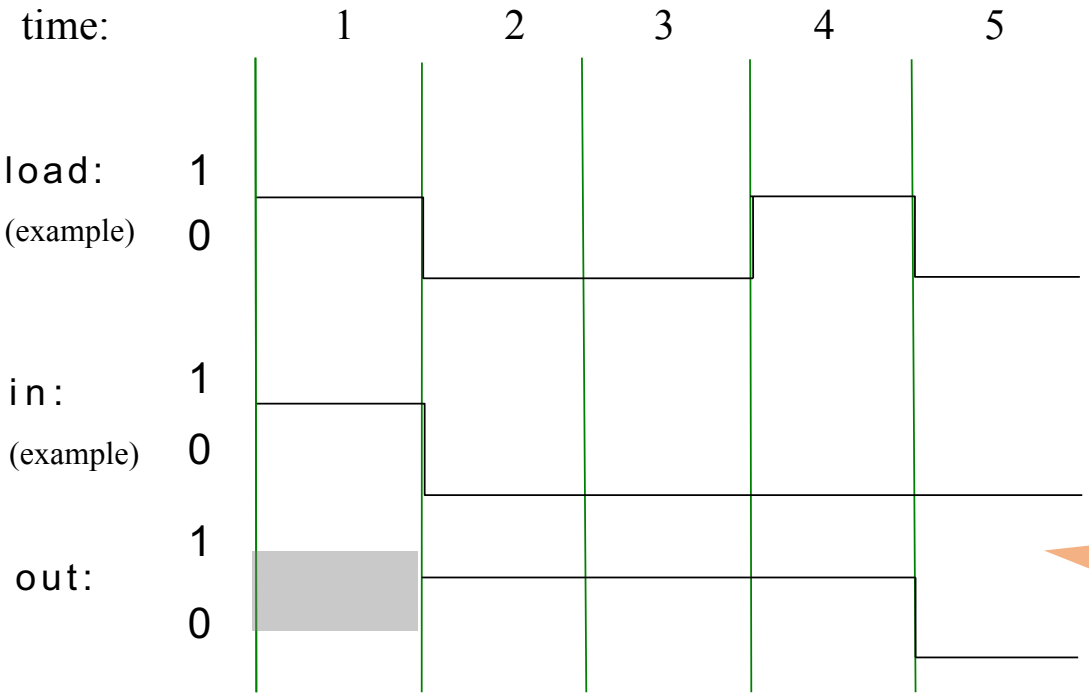
1-Bit Register Implementation





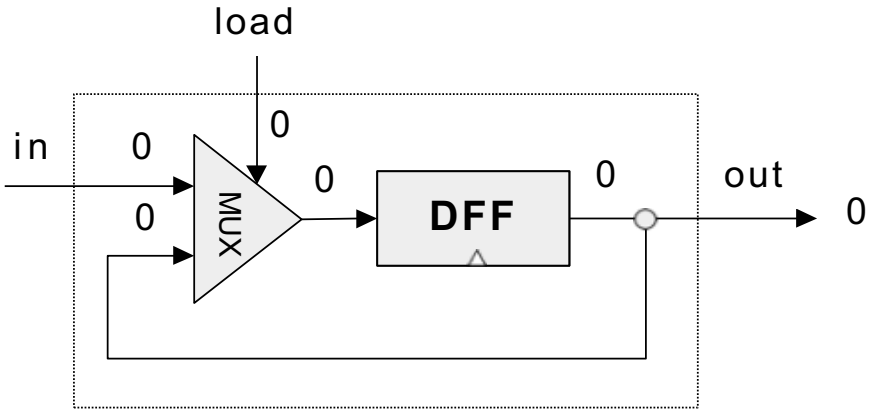
Computers Are Flexible





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

HDL for 1-bit Register



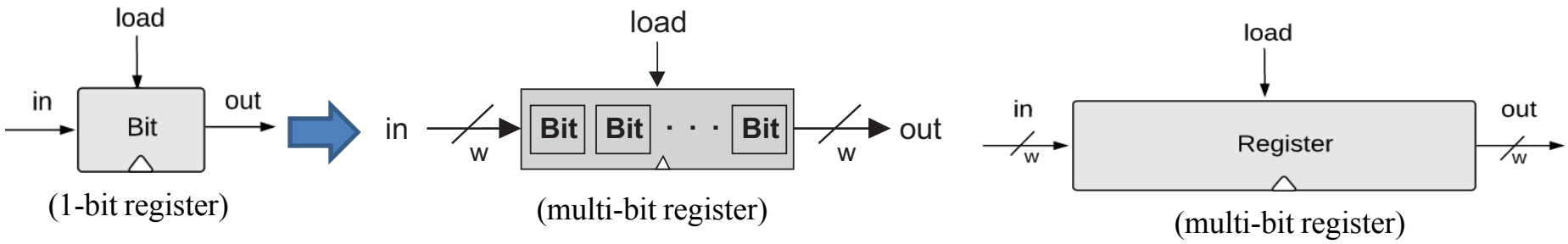
Bit.hdl

```
/** 1-bit register:
  * If load[t] == 1 then.  //load input in the ff
  *     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=sendBack, b=in, sel=load, out=MuxOut);
     DFF(in=MuxOut, out=sendBack, out=out);
}
```

1-Bit Register Demo

Multi-Bit Register

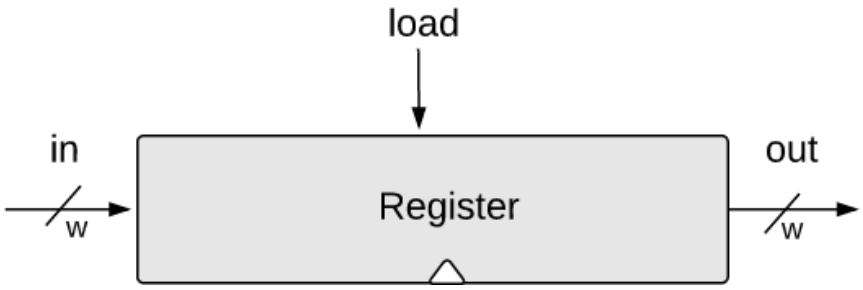
Multi-Bit Register



- A register is actually a group of flip-flops, each flip flop capable of storing one bit of information
- An n-bit register consists of a group of n flip-flops capable of storing n bits of binary information. In this course we will focus on designing of 16-bit registers for our computer
- A 16 bit register can be created from an array of 16 1-bit registers

16 Bit Register API

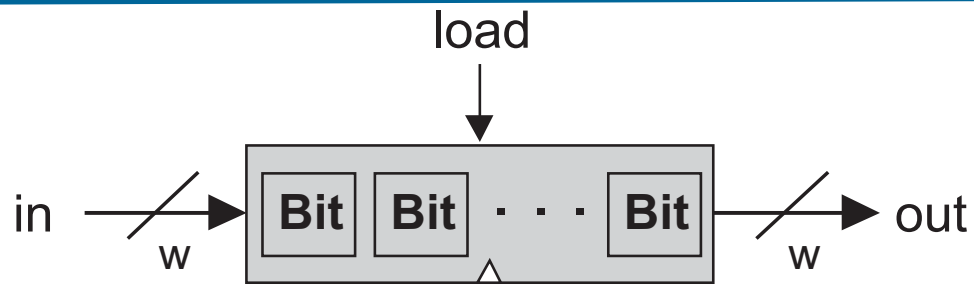
- The API of the 16 bit Register chip is essentially the same as the 1-bit register, except that the input and output pins are designed to handle multi-bit values
- The interface diagram and API of a 16-bit register is shown below
- The Bit and Register chips have exactly the same read/write behavior:
  - **Read:** To read the contents of a register, we simply probe its output
  - **Write:** To write a new data value **d** into a register, we put **d** in the **in** input and set the load input to 1. In the next clock cycle, the register commits to the new data value, and its output starts emitting **d**, and it will keep emitting this new value forever till the time we decide to write a new value in it



Chip name: Register
Inputs: in[16], load
Outputs: out[16]
Function:

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

HDL for 16-bit Register



Register.hdl

```
/**
 * 16-bit register:
 * If load[t] == 1 then out[t+1] = in[t]
 * else out does not change
 */
CHIP Register {
     IN in[16], load;
     OUT out[16];
   PARTS:
     Bit(in=in[0], load=load, out=out[0]);
     Bit(in=in[1], load=load, out=out[1]);
     Bit(in=in[2], load=load, out=out[2]);
     Bit(in=in[3], load=load, out=out[3]);
       . . . .
     Bit(in=in[15], load=load, out=out[15]);
}
```

```
CHIP Bit {
     IN in, load;
     OUT out;
   PARTS:
     Mux(a=sendBack, b=in, sel=load, out=MuxOut);
     DFF(in=MuxOut, out=sendBack, out=out);
}
```

16-Bit Register Demo

Things To Do

- Practice writing HDL of the register chips and verify their behavior by loading and running them on the h/w ssimulator. You can download the .hdl, .tst and .cmp files of above chips from the course bitbucket repository:

https://bitbucket.org/arifpucit/coal-repo/

- Practice the different timing diagrams that we have used to describe the behavior of D-flip flop and the single bit register on a piece of paper by yourself.

Coming to office hours does NOT mean you are academically week!