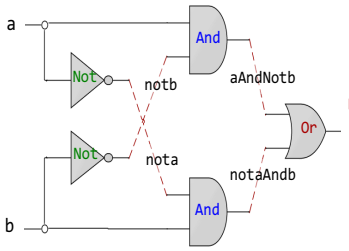
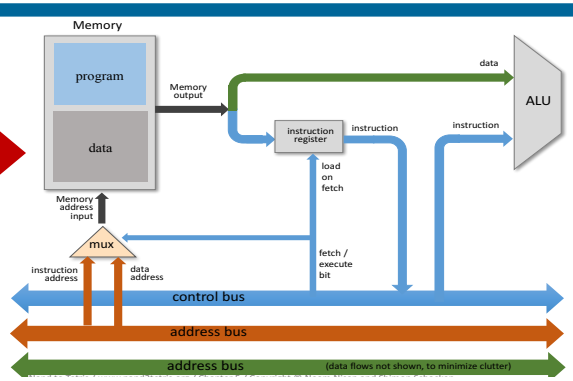
Digital Logic Design




```
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 # 18-19

Registers, Memory and Counters

```
#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

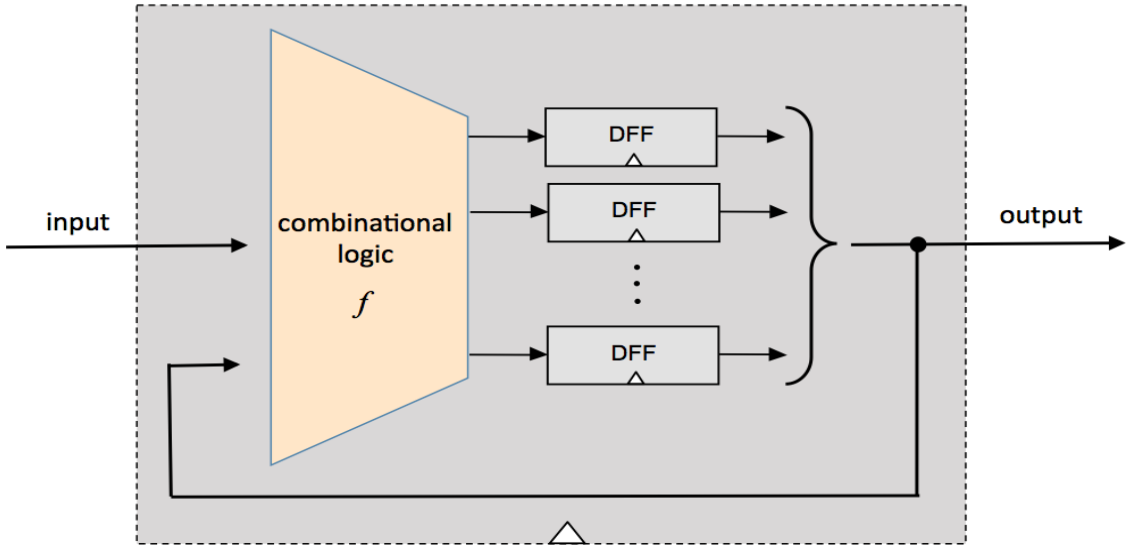
Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

- Review of Sequential Chips
- What are **Registers**?
- Design and HDL of 1-bit to 16-bit Registers
- Concept of **Memory** Hierarchy
- Multi-Byte Read/Write
- Design and HDL of Random Access Memory (8, 64, 512, 4K, 16K words)
- What are **Counters**?
- Why do we need Counter for our Hack Computer
- Concept of Program Counter
- Design and Implementation of PC for Hack Computer
- Demo on H/W Simulator

Review of Sequential Chips

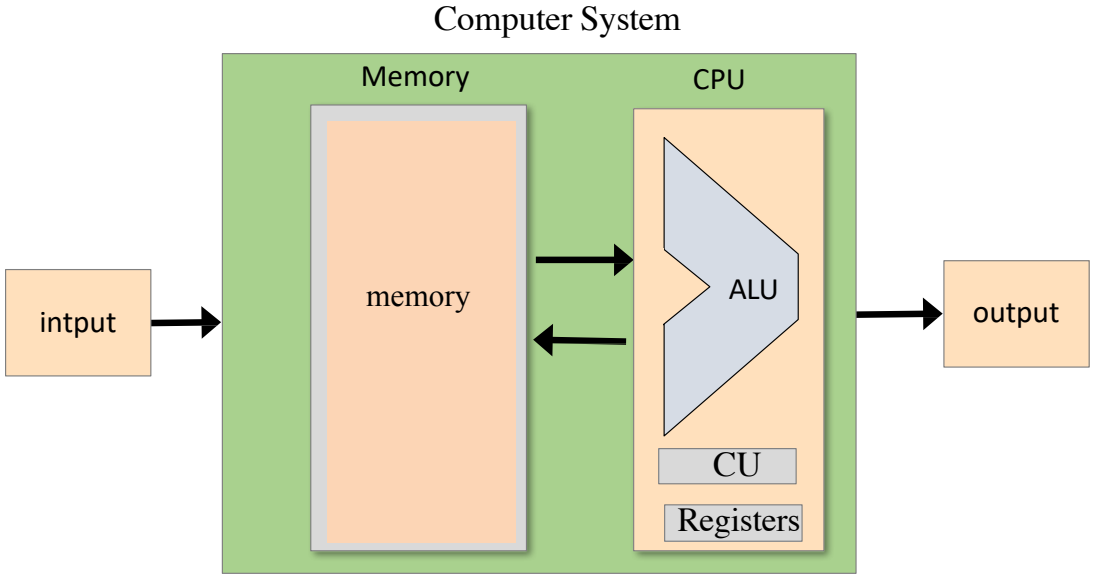
```
state(t) = f (state(t-1), input(t))
```



- 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

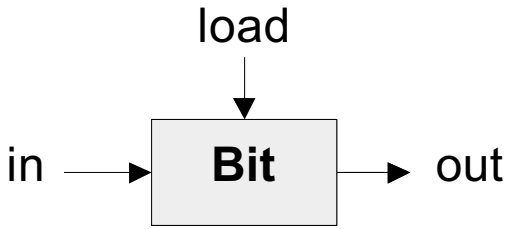
CPU Registers



- 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)

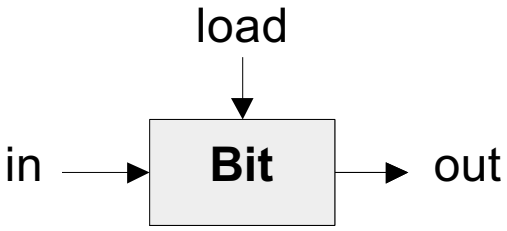
1-Bit Register

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

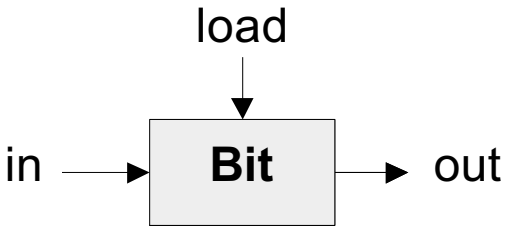
Sequential Chips: 1-Bit Register



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

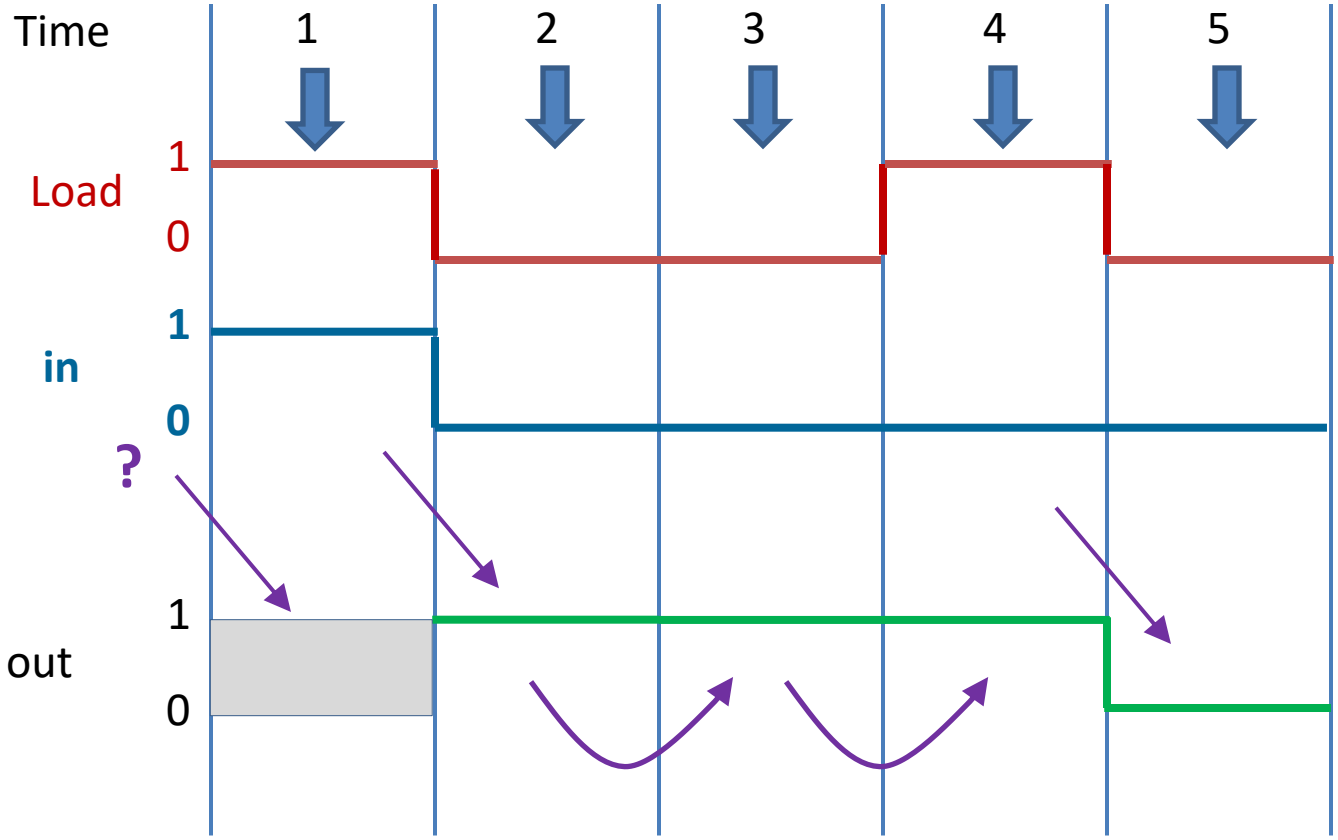
- Goal: Remember an input bit forever, until requested to load a new value
- More accurately:
  - Stores a bit until...
  - Instructed to load, and store, another bit

Sequential Chips: 1-Bit Register



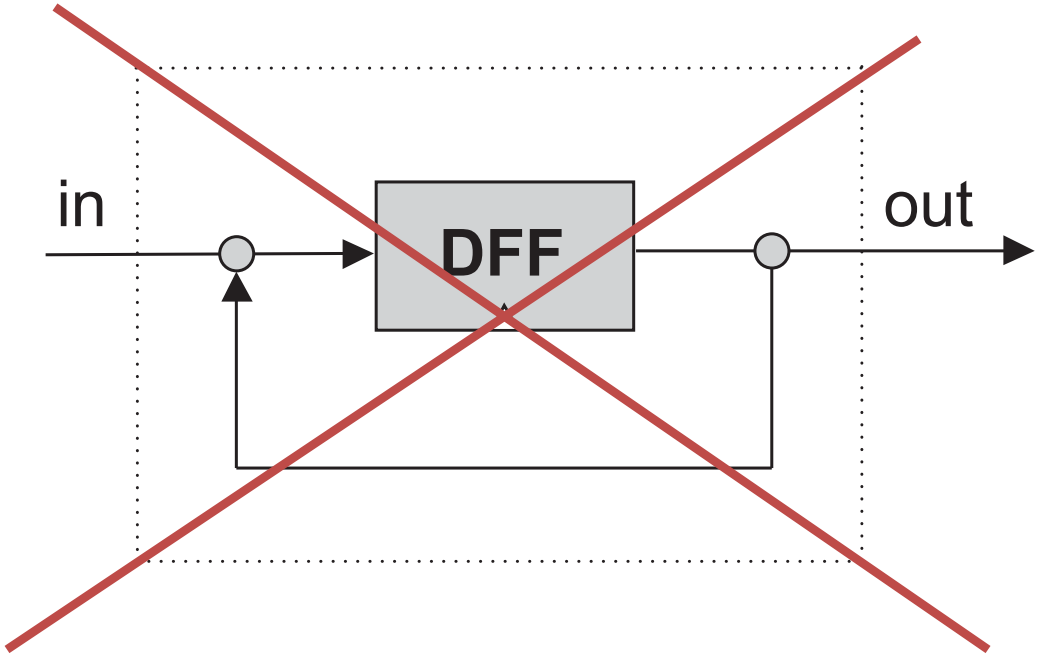
Chip name: Bit
Inputs: in, load
Outputs: out
Function:

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

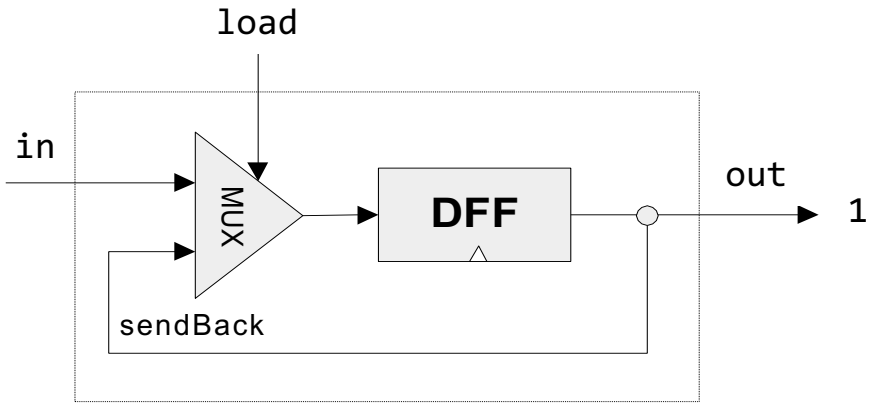


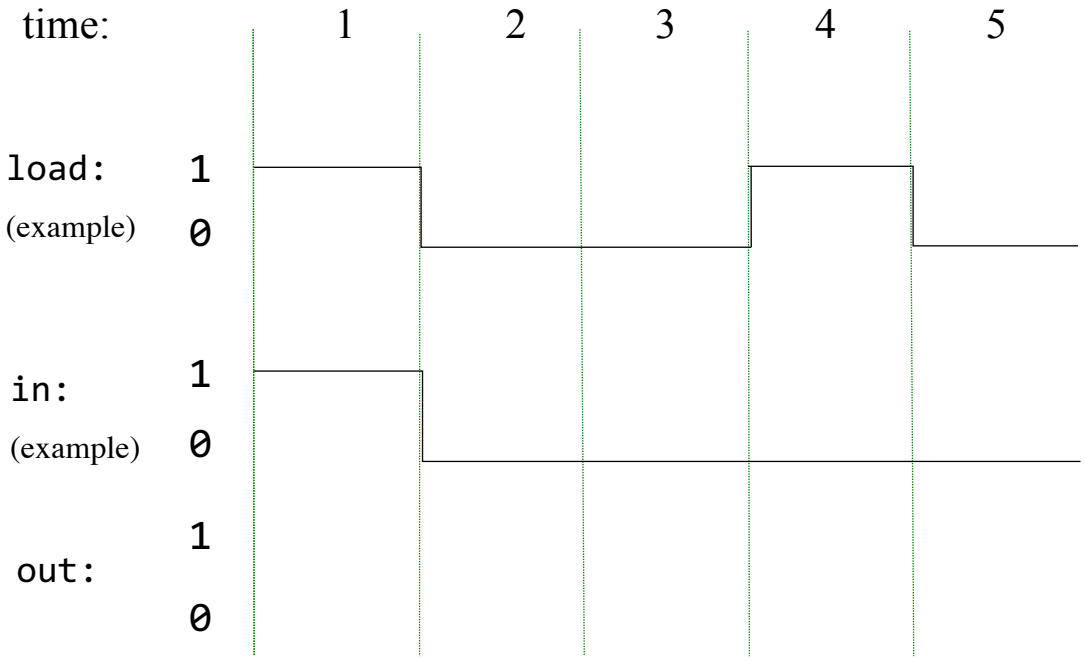
1-Bit Register Implementation

1-Bit Register Implementation

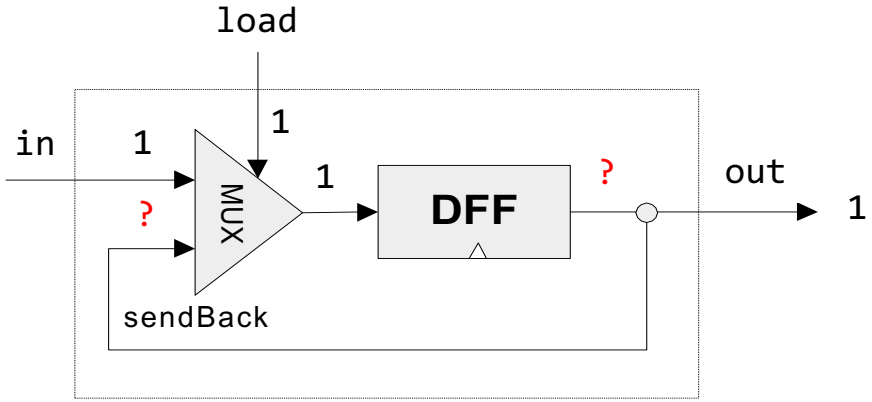


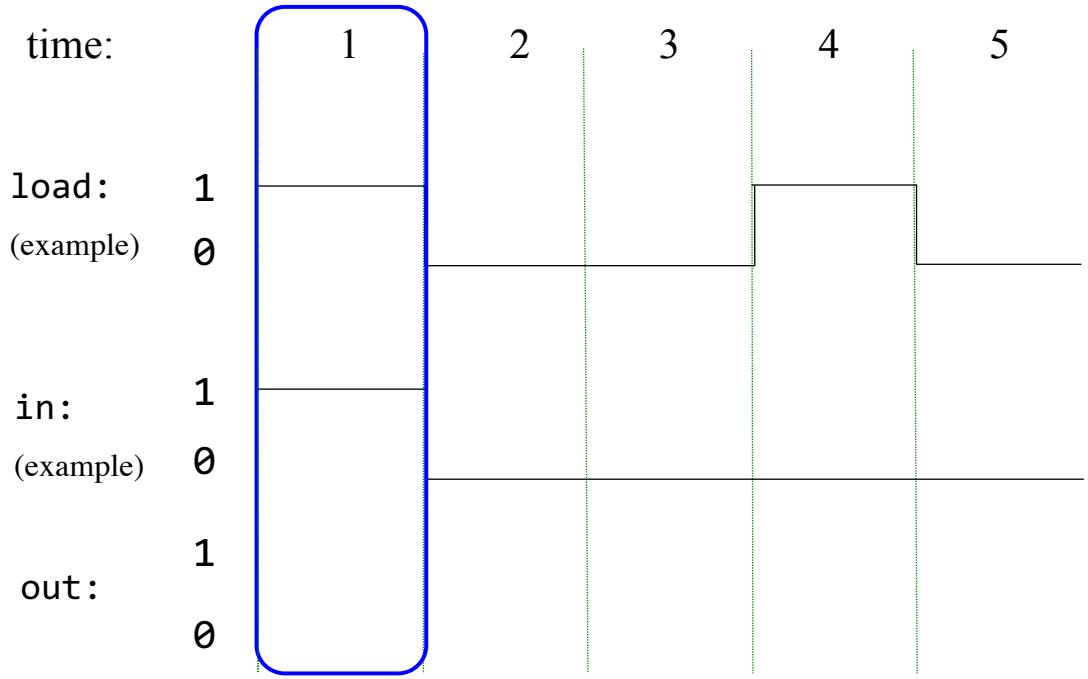
1-Bit Register Implementation



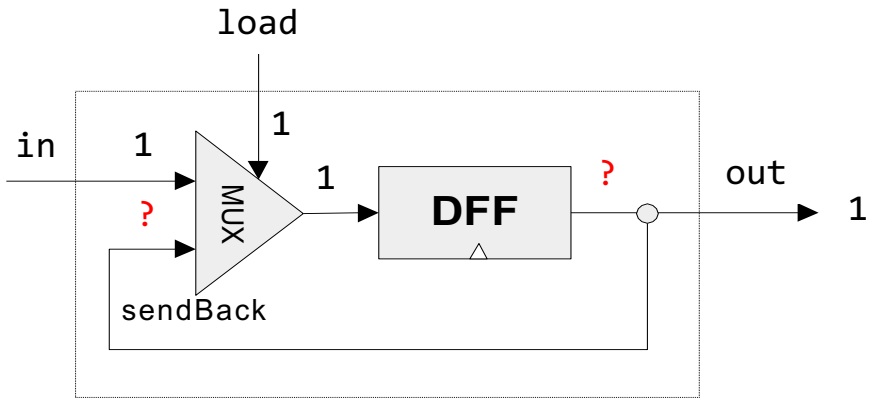


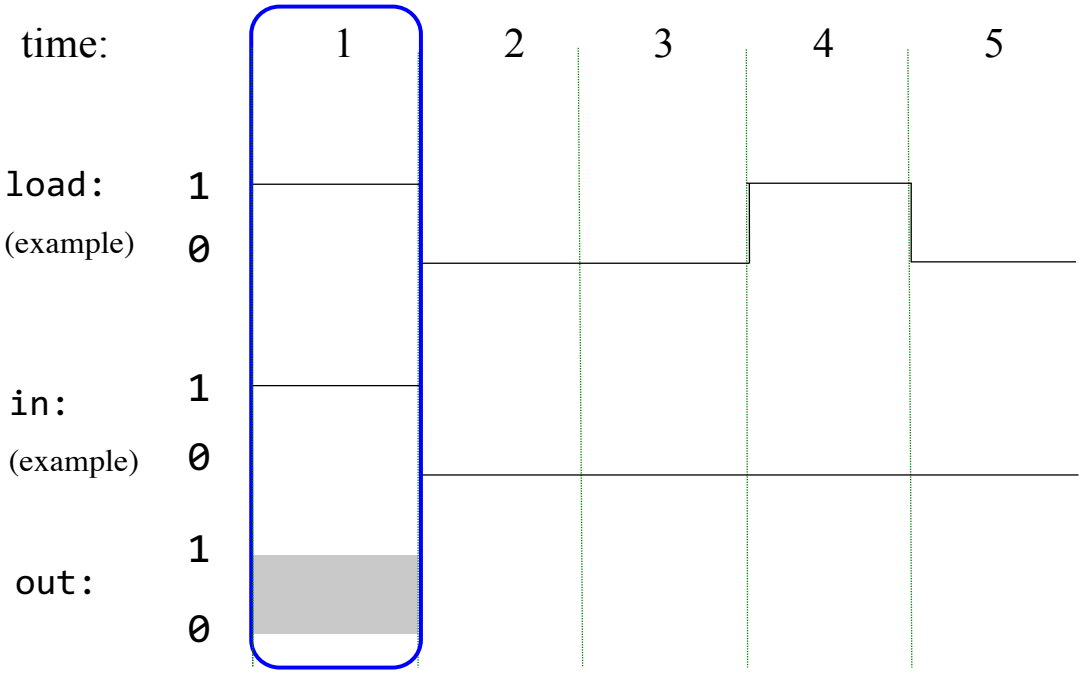
1-Bit Register Implementation



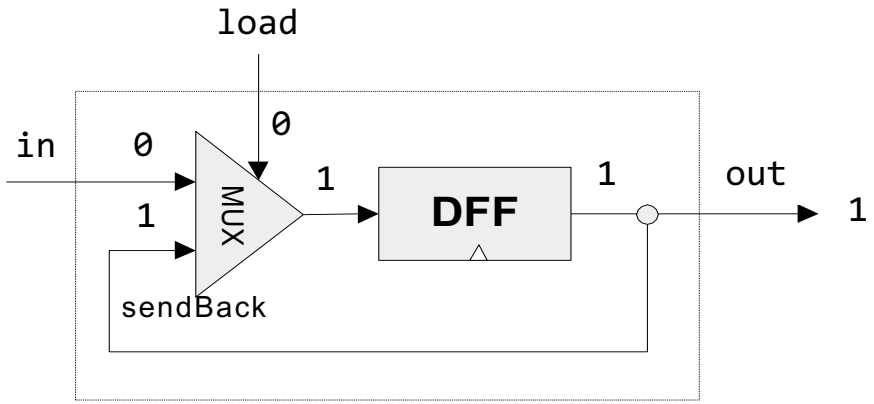


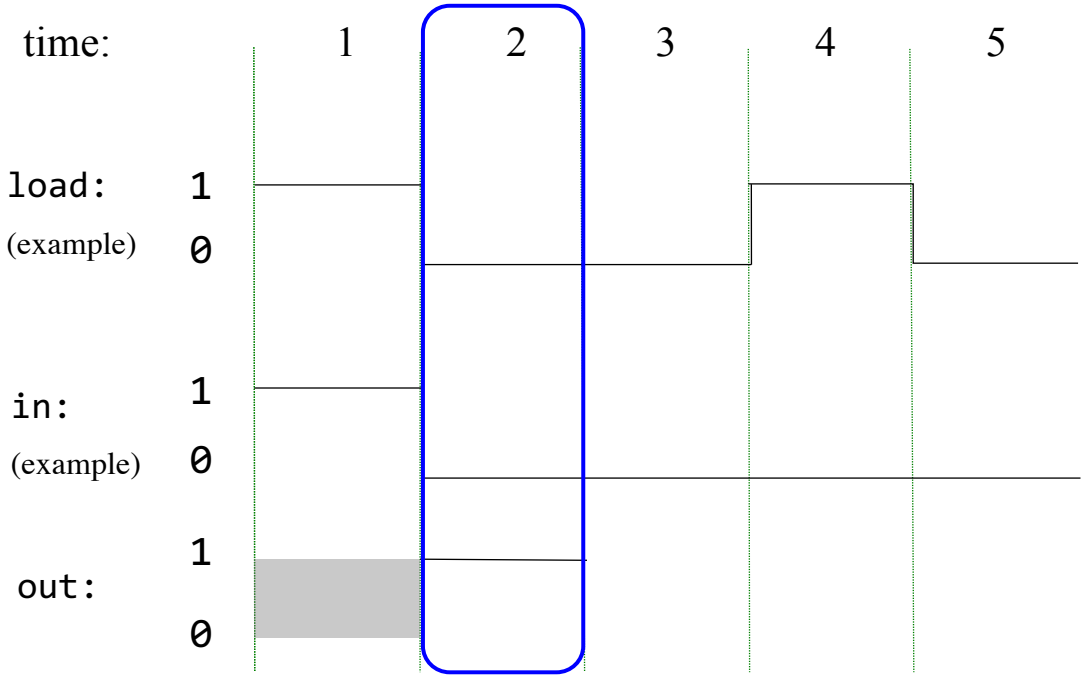
1-Bit Register Implementation



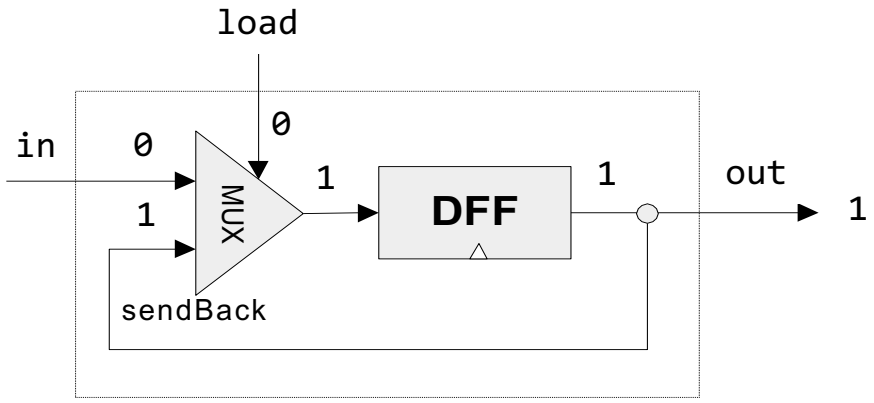


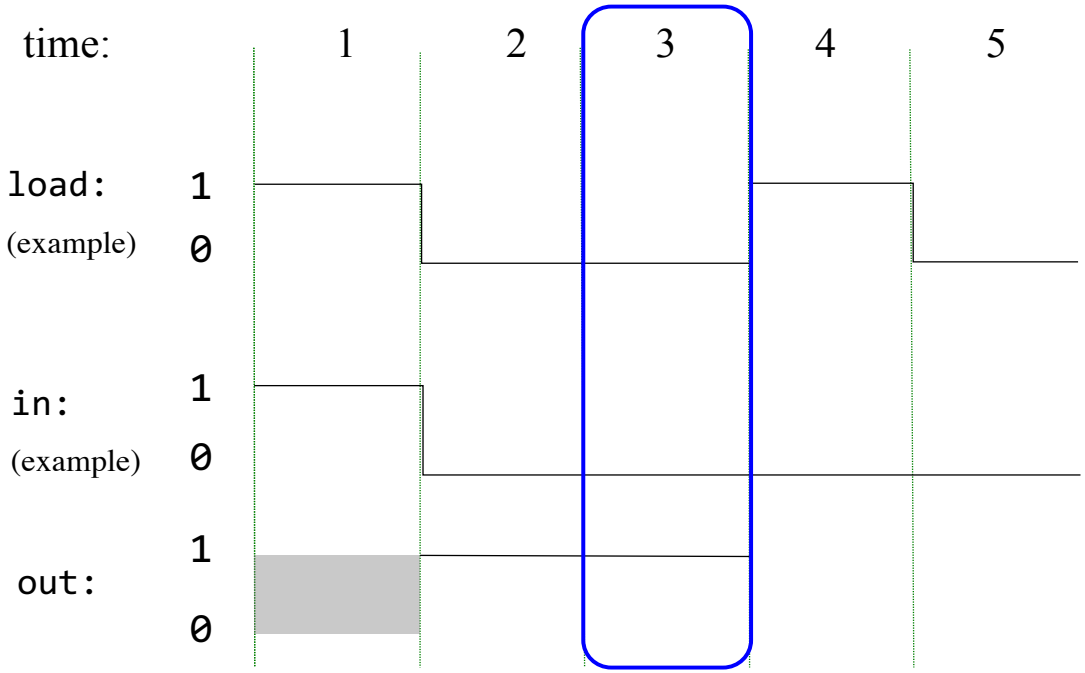
1-Bit Register Implementation



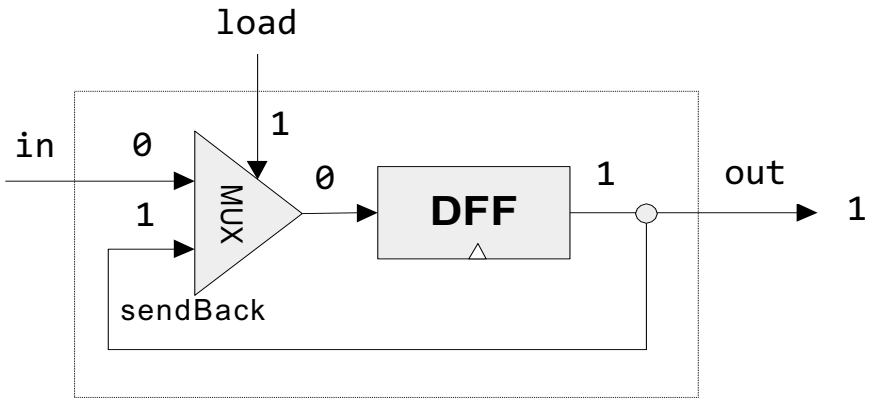


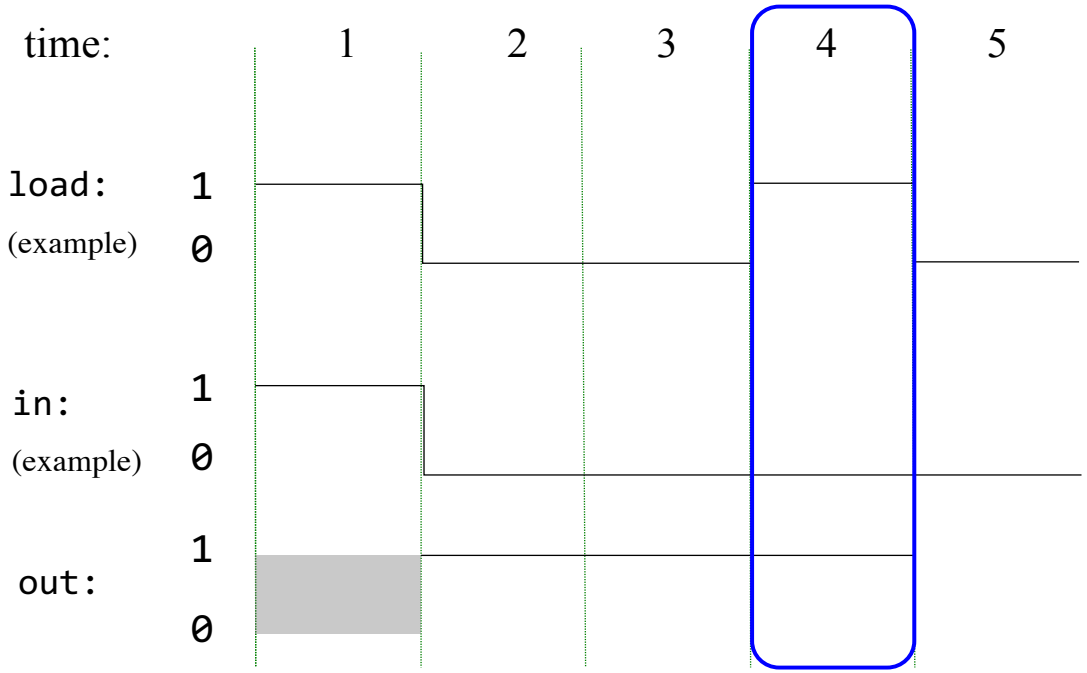
1-Bit Register Implementation



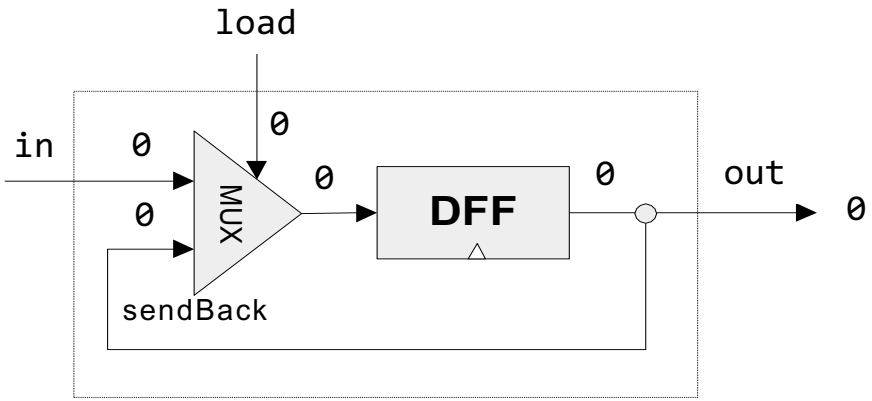


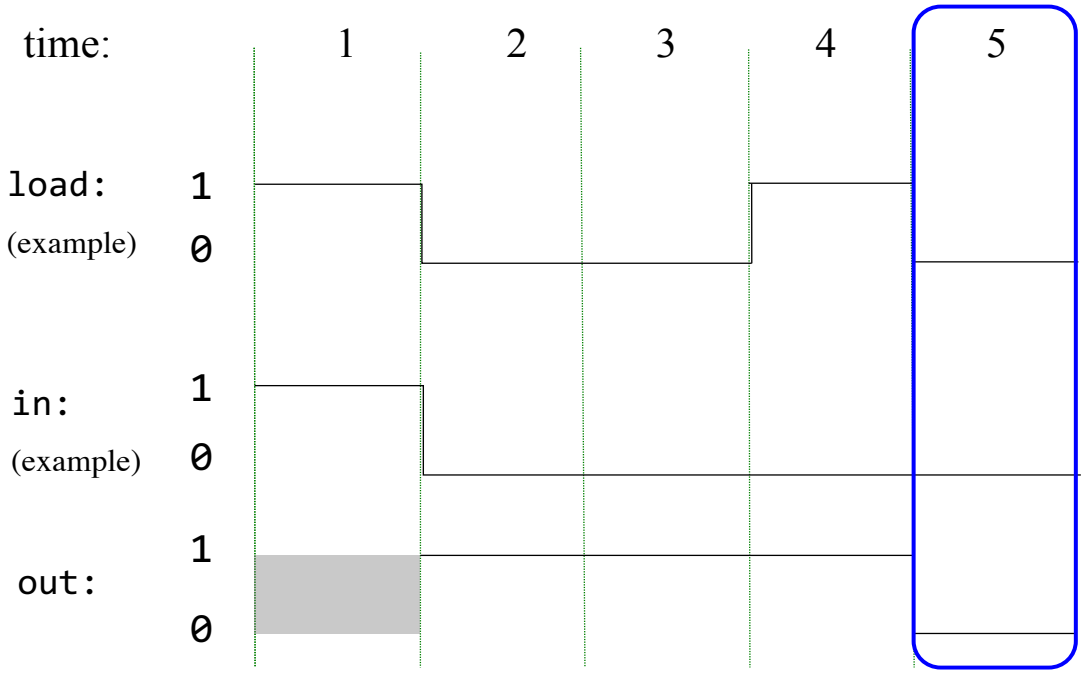
1-Bit Register Implementation



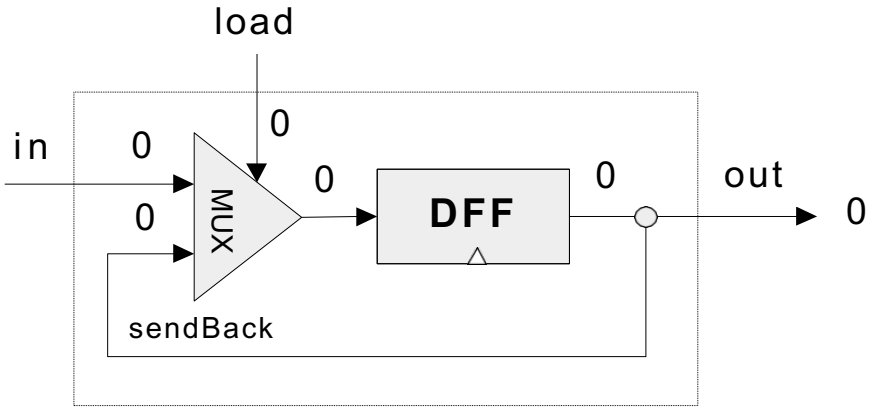


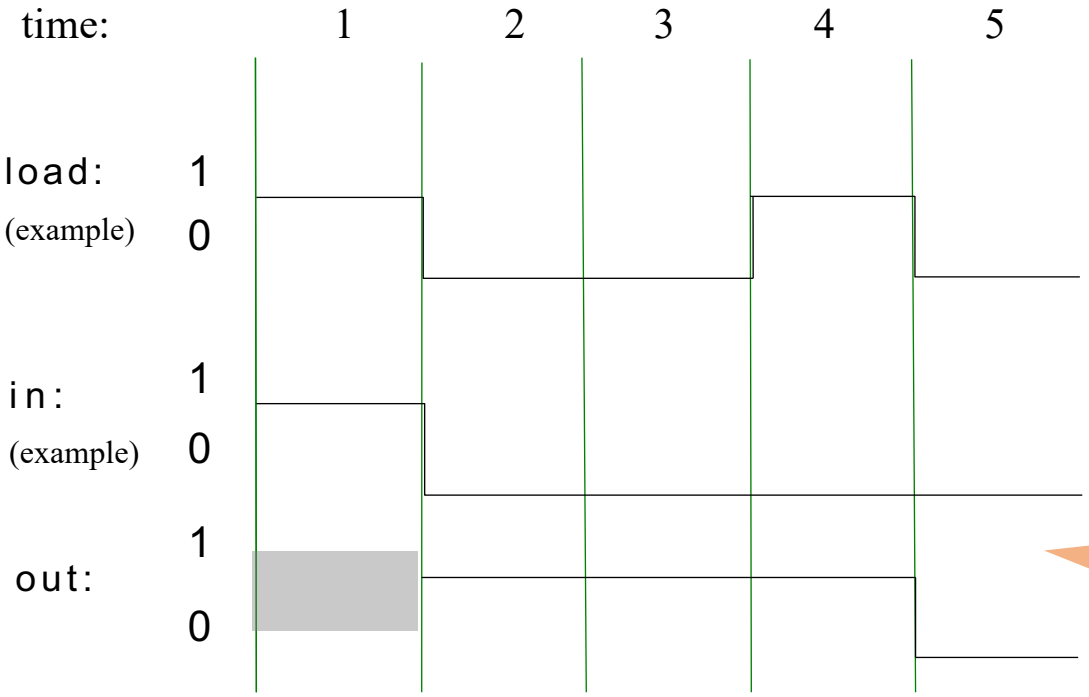
1-Bit Register Implementation





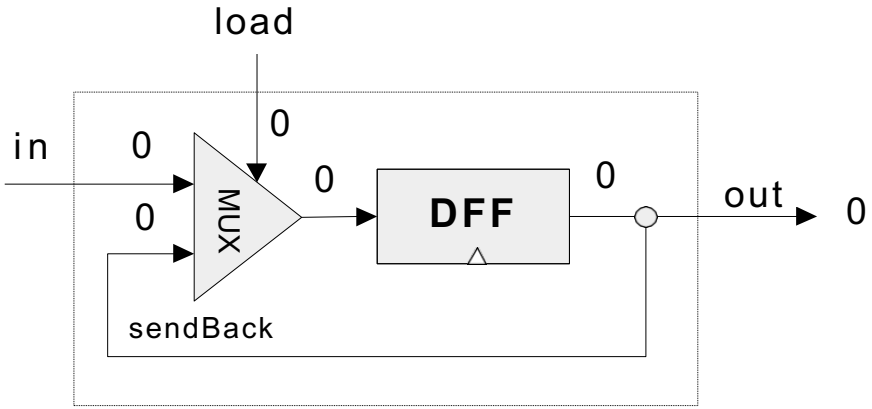
Computers Are Flexible





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

HDL for 1-bit Register



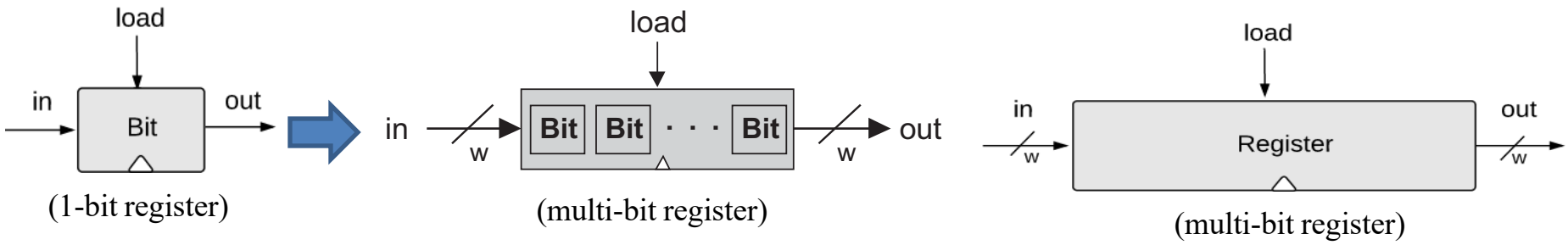
Bit.hdl

```
/** 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

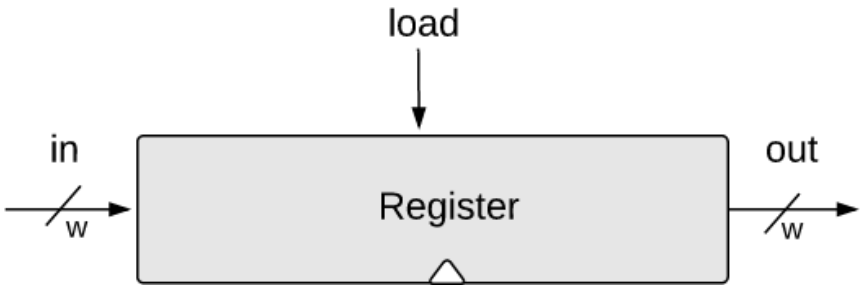
Multi-Bit Register



- 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

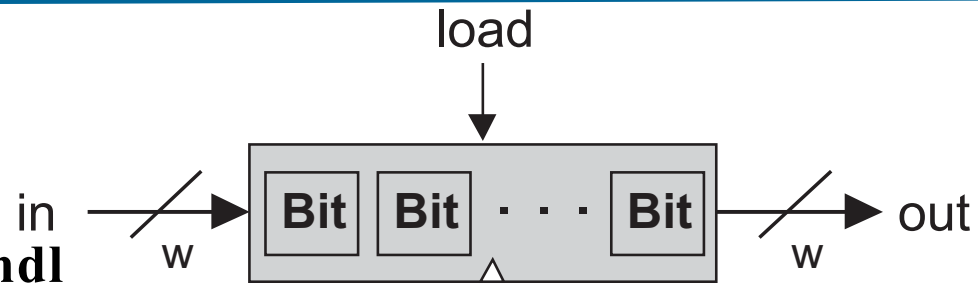
- The API of the 16 bit Register chip is essentially the same as the 1-bit register, except that the input and output pins are designed to handle multi-bit values
- The interface diagram and API of a 16-bit register is shown below
- The Bit and Register chips have exactly the same read/write behavior:
  - **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

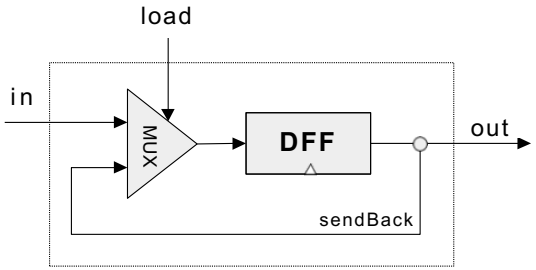


Chip name: Register
Inputs: in[16], load
Outputs: out[16]
Function:

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

HDL for 16-bit Register





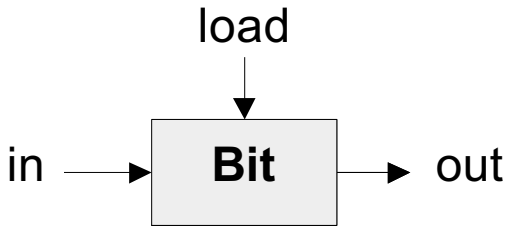
Register.hdl

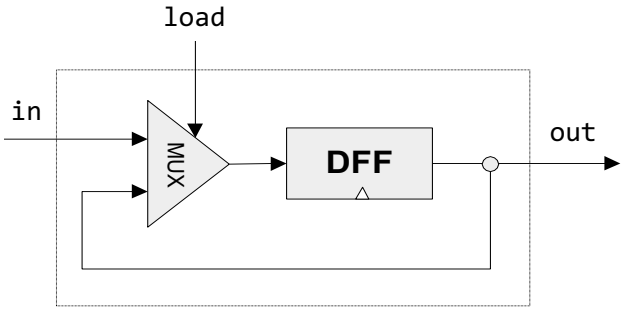
```
/**
 * 16-bit register:
 * If load[t] == 1 then out[t+1] = in[t]
 * else out does not change
 */
CHIP Register {
     IN in[16], load;
     OUT out[16];
   PARTS:
     Bit(in=in[0], load=load, out=out[0]);
     Bit(in=in[1], load=load, out=out[1]);
     Bit(in=in[2], load=load, out=out[2]);
    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

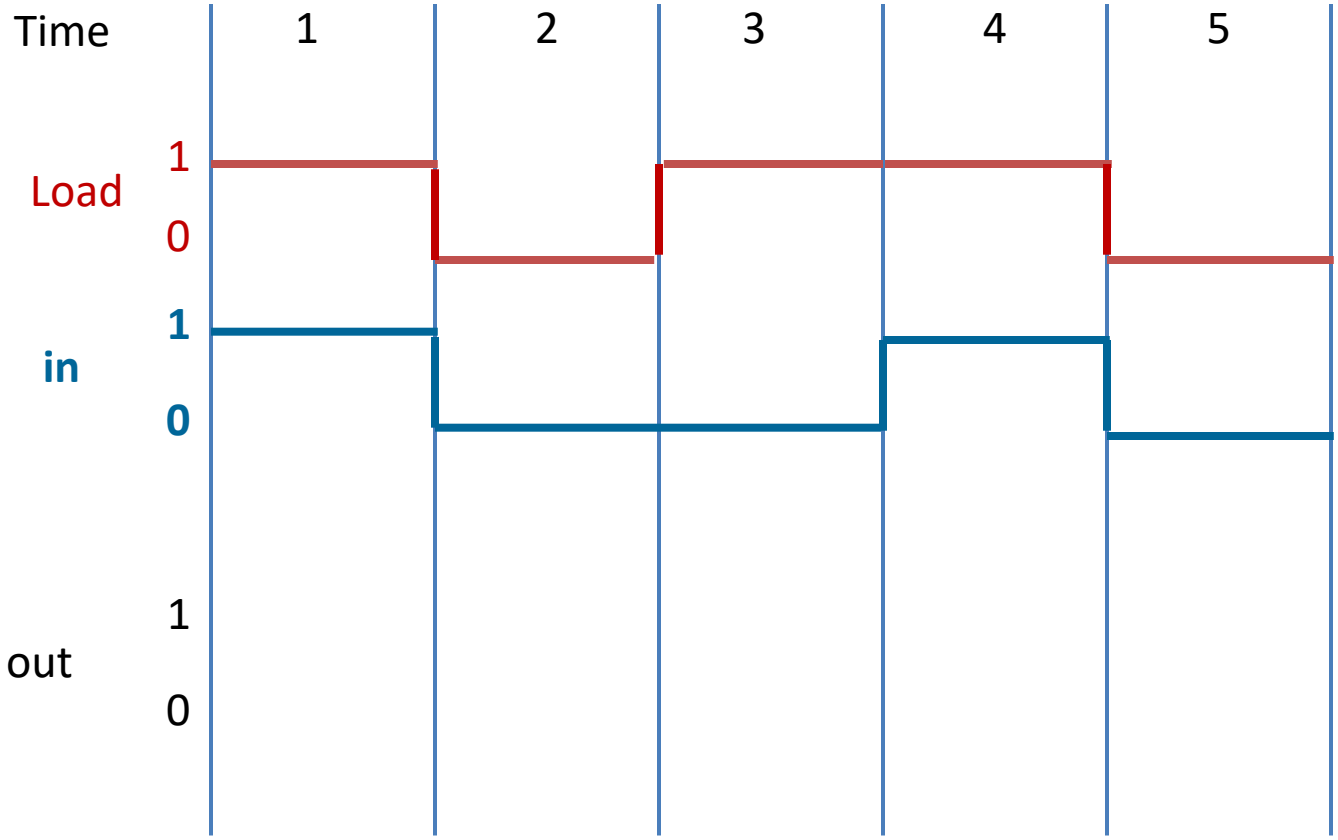
```
CHIP Bit {
     IN in, load;
     OUT out;
   PARTS:
     Mux(a=sendBack, b=in, sel=load, out=MuxOut);
     DFF(in=MuxOut, out=sendBack, out=out);
}
```

16-Bit Register Demo

Class Quiz:

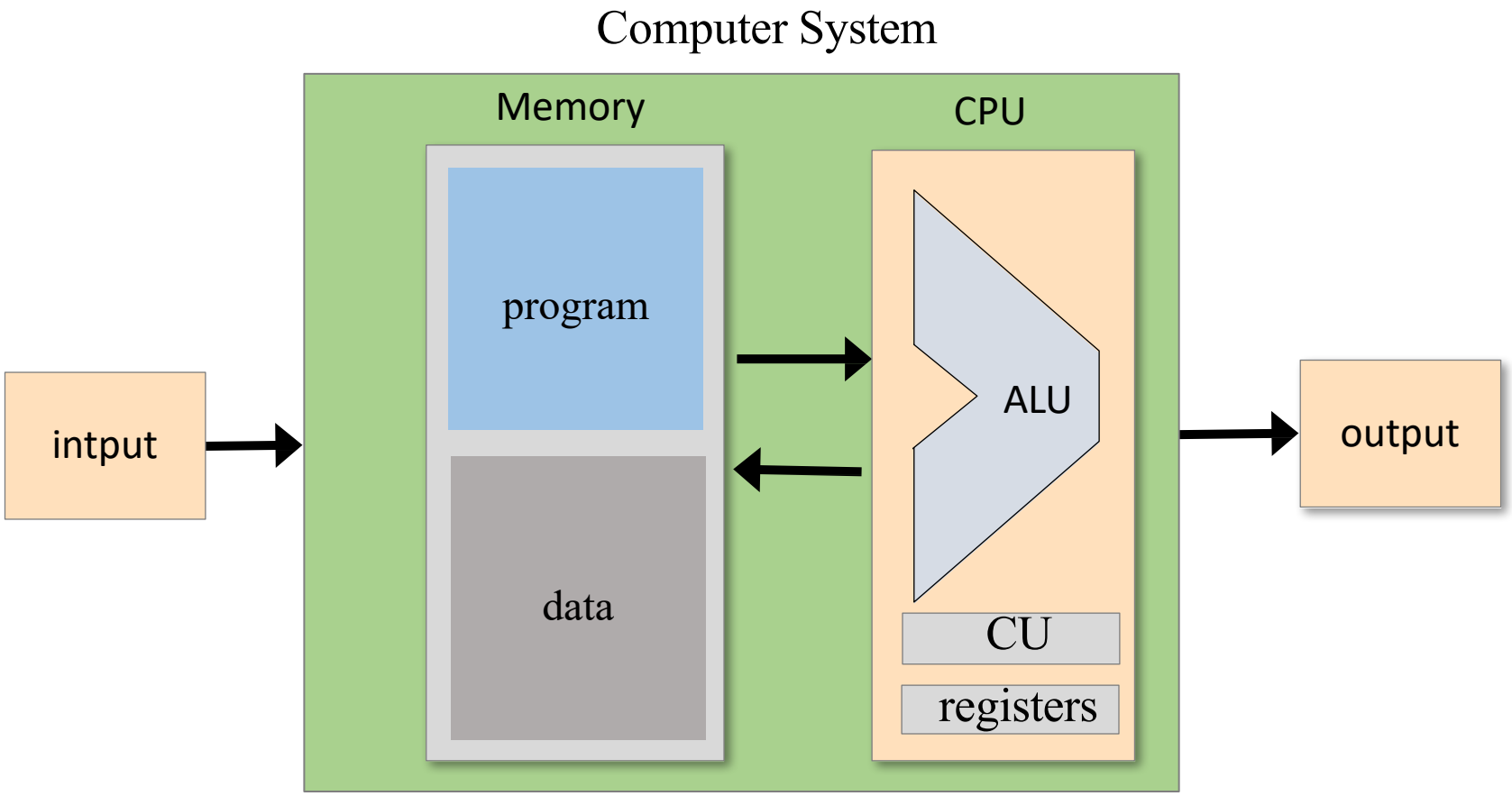

load
in
Bit
out


load
in
MUX
DFF
out


Time
1
2
3
4
5
Load
1
0
in
1
0
out
1
0

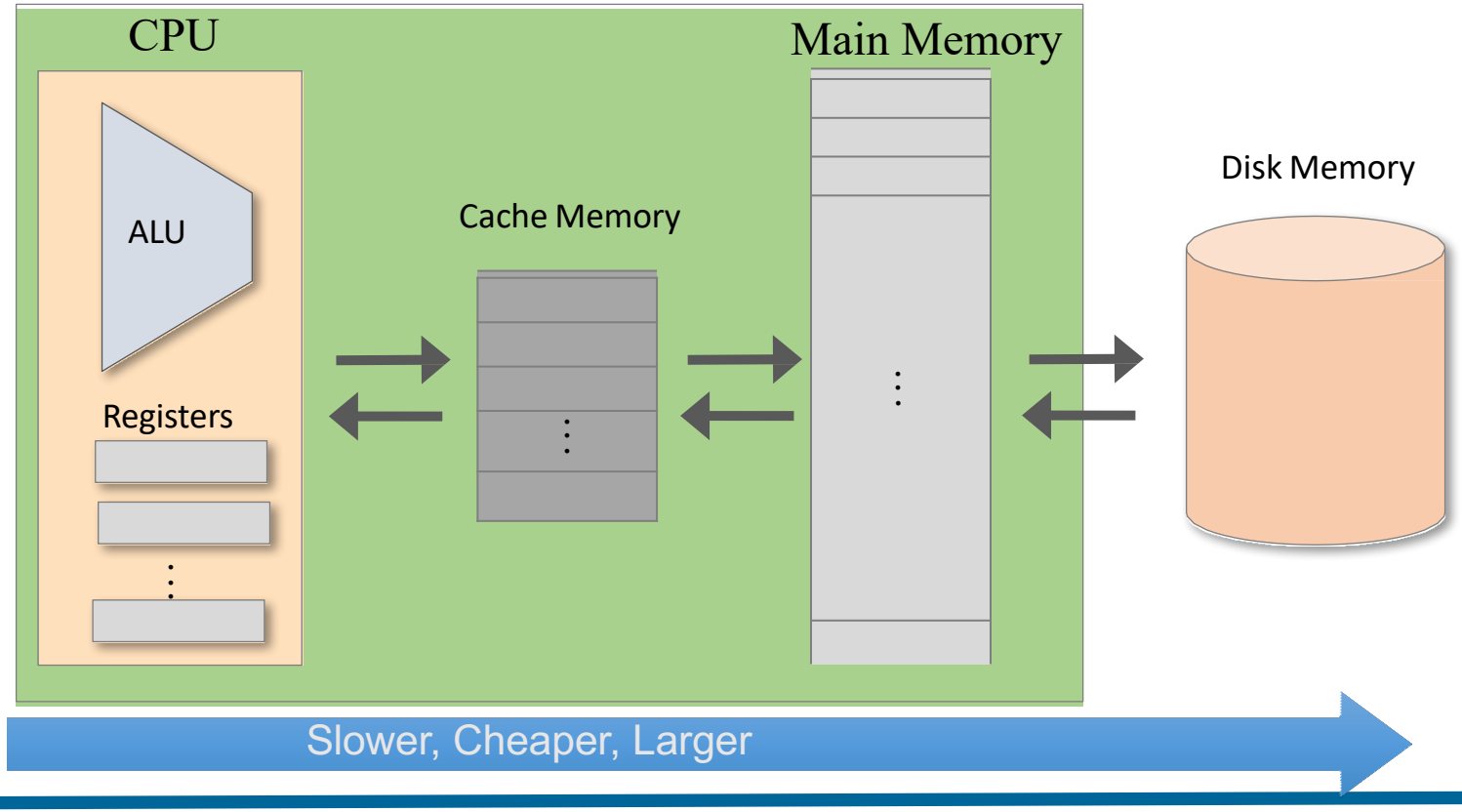
Memory Overview

Stored Program Concept



Memory Hierarchy

- Accessing a memory location is expensive, as we need to supply an address and then read/write the contents of that location of memory. Moreover, moving the memory contents into the CPU and vice versa also takes time
- **Solution:** Memory Hierarchy



Access Times of Memory

Memory Unit	Example Size	Typical Speed
Registers	16, 64-bit registers	1 nanosecond
Cache memory	4 - 8 Megabytes (L1 and L2)	5-60 nanoseconds
Primary Storage	2 - 32+ Gigabytes	100-150 nanoseconds
Secondary Storage	500 Gigabytes – 4+ Terabytes	3-15 milliseconds

Memory Sizes/Capacity

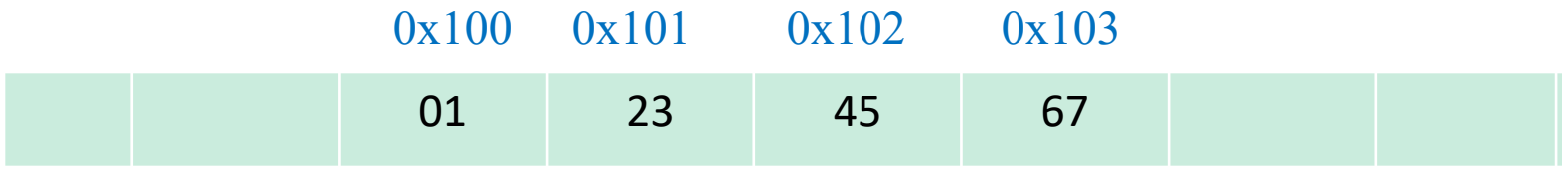
Decimal Term	Abbreviation	Value	Binary Term	Abbreviation	Value	%Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
Gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%`
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exabibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

Random Access Memory (RAM)

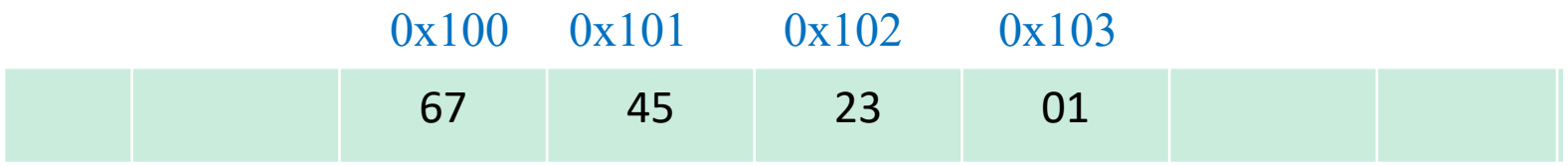
- The computer's main memory is also called the Random Access Memory, because irrespective of the RAM size, every word gets selected instantaneously, at more or less the same time
- It is also known as read/write memory as it allows CPU to read as well as write data and instructions into it
- RAM is a microchip implemented using semiconductors. There are two categories of RAM
  - **Dynamic RAM (DRAM):** It is made up of memory cells where each cell is composed of one capacitor and one transistor. DRAM must be refreshed continually to store information. The refresh operation occurs automatically thousands of times per second. DRAM is slower and less-expensive
  - **Static RAM (SRAM):** It retains the data as long as power is provided to the memory chip. It needs not be refreshed periodically. SRAM uses multiple transistors for each memory cell. It does not use capacitor. SRAM is often used as cache memory due to its high speed. SRAM is more expensive than DRAM

Multi-Byte Ordering

- All 32 bit machines load and store 32 bits of data (word) with each operation. The question is how are the bytes of a multi-byte variable ordered in memory?
- Consider a 32 bit variable having a value of 0x01234567, that needs to be stored at address 0x100
- There are two conventions that the h/w designers can follow:
  - **Big Endian:** Most significant byte is written at the lowest address byte (MSB first). Used by MIPS and Internet



  - **Little Endian:** Least significant byte is written at the lowest address byte (LSB first). Used by x86 and ARM

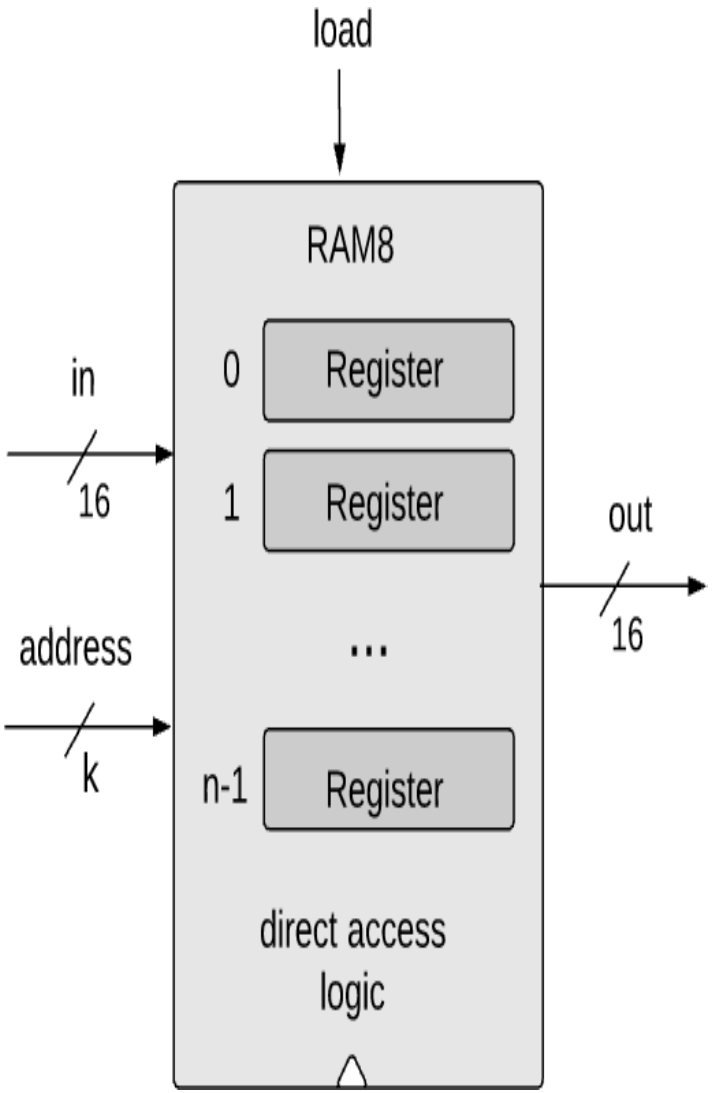


Designing Random Access Memory

Design of RAM

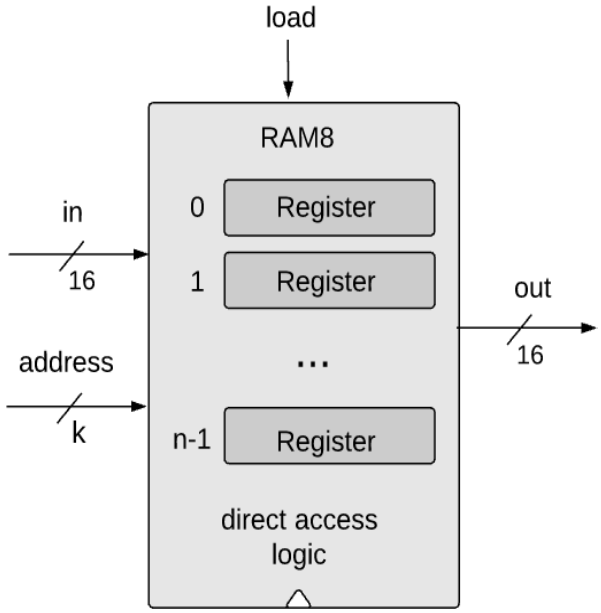
- RAM is an array of ***n w-bit registers***, equipped with direct access circuitry. The number of registers **(n)** and the width of each register **(w)** are called the memory's size and width respectively
- In simple words, you can think of RAM as a sequence of n addressable registers with addresses 0 to n-1
- At any given time only one register in the RAM is selected. It is this register whose value is available on *out* during a read operation. Similarly, it is this register whose contents will be over written during a write operation
- Now to select a register we need its address. Address width varies with the number of registers/words in the RAM, e.g., for RAM8 the address size is 3 bits

$$\mathbf{k} = \text{Address bits} = \log_2 \mathbf{n}$$



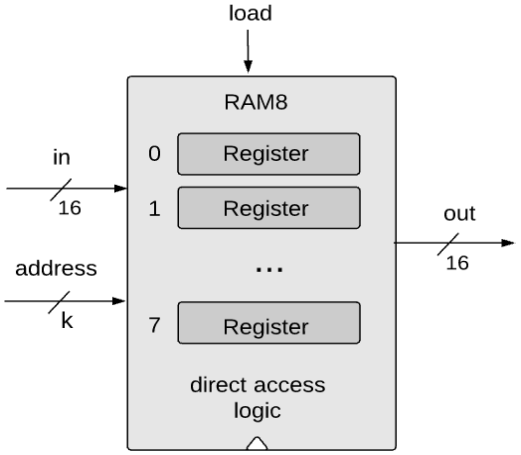
Read/Write Logic of RAM

- At any given point of time: one register in the RAM is selected, all the other registers are irrelevant
- **Read:** To read the contents of register number ***i***,
  - Set address = ***i***
  - Result: The RAM's output pin ***out*** emits the state of the register ***i***. This is a combinational operation, independent of the clock
- **Write:** To write a new data value ***d*** into register number ***i***,
  - Set address = ***i***
  - Set ***in*** = ***d***
  - Set ***load*** = **1**
  - Result: The state of register ***i*** becomes ***d*** and form next clock cycle onwards, ***out*** emits ***d***



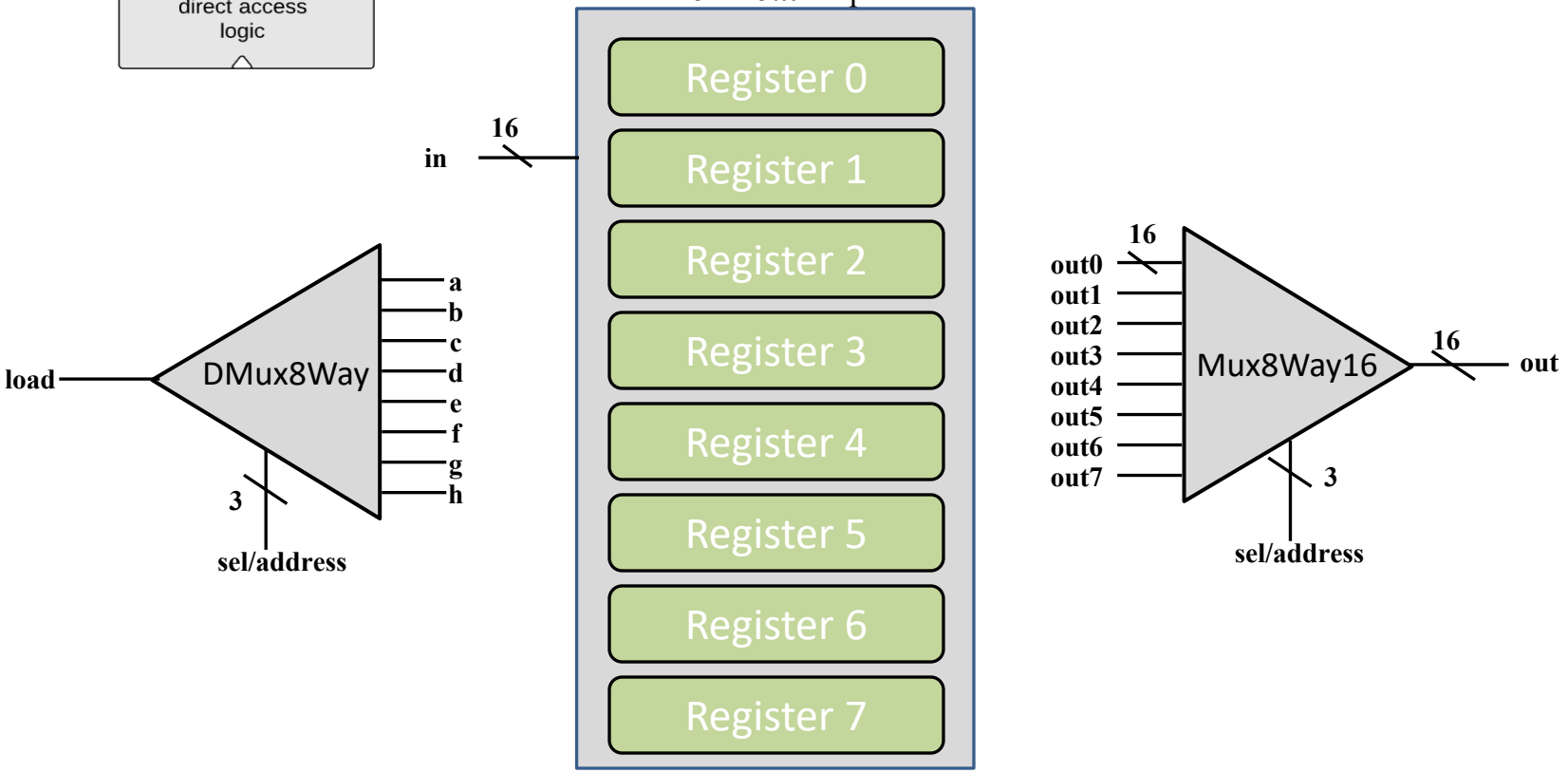
Chip Name	Size (n)	Address bits (k)
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

8-Register/words RAM

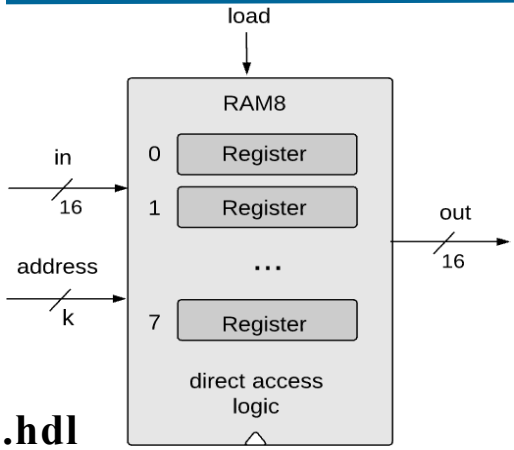


Implementation tips:

- Memory of **8** registers, each 16 bit-wide. ***Out*** holds the value stored at the memory location specified by address. If **load**==1, then the **in** value is loaded into the memory location specified by **address** (the loaded value will be emitted to **out** from the next time step onward)
- Feed the 16 bit ***in*** value to all the registers, simultaneously
- **Write:** Use DMux8Way chip to select one of the eight registers specified by address
- **Read:** Use Mux8Way16 chip to select the 16 bit contents of register specified by address on 16 bit ***out*** output



8-Register/words RAM

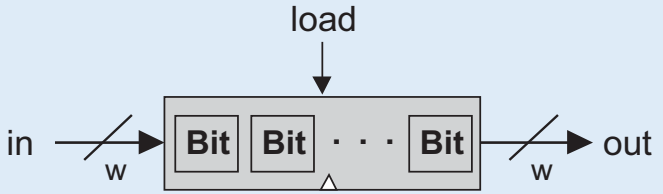


Implementation tips:

- Memory of **8** registers, each 16 bit-wide. ***Out*** holds the value stored at the memory location specified by address. If load==1, then the in value is loaded into the memory location specified by address (the loaded value will be emitted to out from the next time step onward)
- Feed the 16 bit ***in*** value to all the registers, simultaneously
- Use DMux8Way chip to select one of the eight registers specified by address
- Use Mux8Way16 chip to select the 16 bit contents of register specified by address on 16 bit ***out*** output

RAM8.hdl

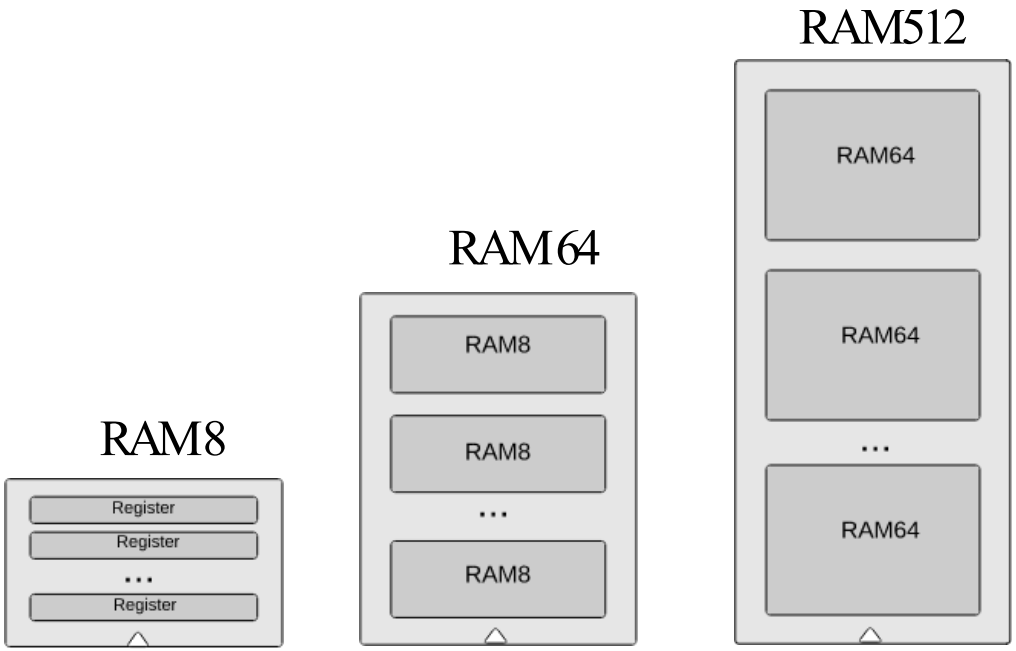
```
CHIP RAM8 {
   IN in[16], load, address[3];
   OUT out[16];
   PARTS:
   DMux8Way(in=load, sel=address, a=load0, b=load1, c=load2, d=load3, e=load4, f=load5, g=load6, h=load7);

   Register(in=in, load=load0, out=out0);
   Register(in=in, load=load1, out=out1);
   Register(in=in, load=load2, out=out2);
   Register(in=in, load=load3, out=out3);
   Register(in=in, load=load4, out=out4);
   Register(in=in, load=load5, out=out5);
   Register(in=in, load=load6, out=out6);
   Register(in=in, load=load7, out=out7);

   Mux8Way16(a=out0, b=out1, c=out2, d=out3, e=out4, f=out5, g=out6, h=out7, sel=address, out=out);
}
```




```
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]);
}
```

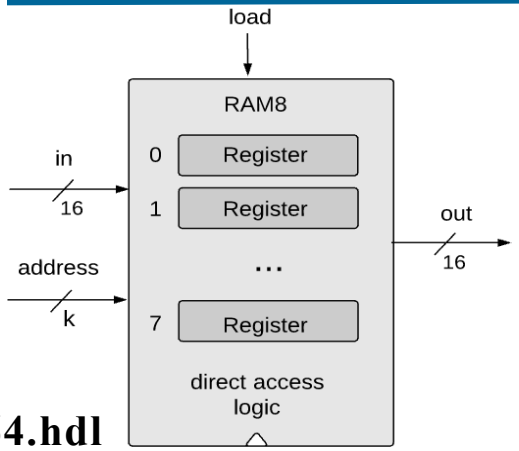
RAM8 Demo

Designing Larger Size RAM Chips



Same technique can be used to implement RAM4K and RAM16K

64-Register/words RAM



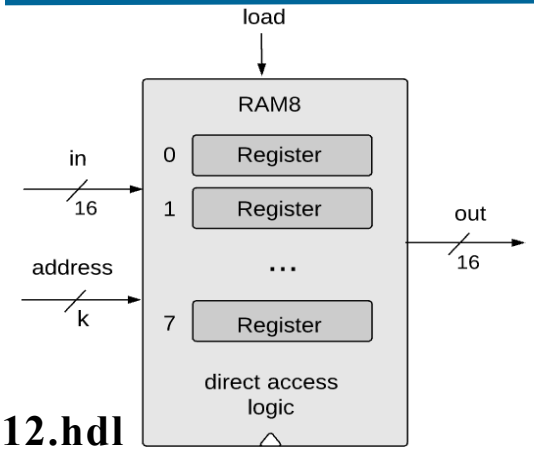
Implementation tips:

- Memory of **64** registers, each 16 bit-wide. ***Out*** holds the value stored at the memory location specified by address. If load==1, then the in value is loaded into the memory location specified by address (the loaded value will be emitted to out from the next time step onward)
- Feed the 16 bit ***in*** value to all the registers, simultaneously
- Use DMux8Way chip to select one of the eight registers specified by address
- Use Mux8Way16 chip to select the 16 bit contents of register specified by address on 16 bit ***out*** output

RAM64.hdl

```
CHIP RAM64 {
   IN in[16], load, address[6];
   OUT out[16];
   PARTS:
   DMux8Way(in=load, sel=address[3..5], a=load0, b=load1, c=load2, d=load3, e=load4, f=load5, g=load6, h=load7);

   RAM8(in=in, load=load0, address=address[0..2], out=out0);
   RAM8(in=in, load=load1, address=address[0..2], out=out1);
   RAM8(in=in, load=load2, address=address[0..2], out=out2);
   RAM8(in=in, load=load3, address=address[0..2], out=out3);
   RAM8(in=in, load=load4, address=address[0..2], out=out4);
   RAM8(in=in, load=load5, address=address[0..2], out=out5);
   RAM8(in=in, load=load6, address=address[0..2], out=out6);
   RAM8(in=in, load=load7, address=address[0..2], out=out7);

   Mux8Way16(a=out0, b=out1, c=out2, d=out3, e=out4, f=out5, g=out6, h=out7, sel=address[3..5], out=out);
}
```

512-Register/words RAM



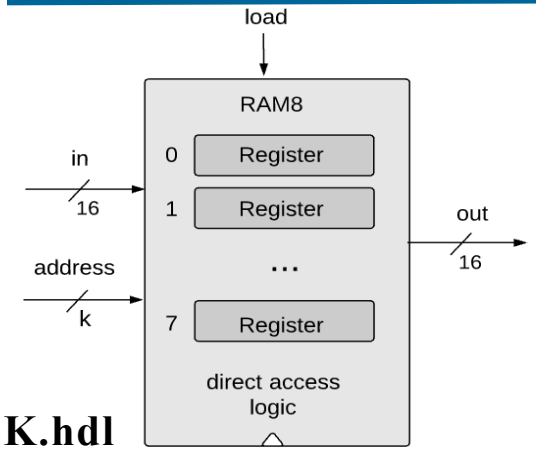
Implementation tips:

- Memory of **512** registers, each 16 bit-wide. ***Out*** holds the value stored at the memory location specified by address. If load==1, then the in value is loaded into the memory location specified by address (the loaded value will be emitted to out from the next time step onward)
- Feed the 16 bit ***in*** value to all the registers, simultaneously
- Use DMux8Way chip to select one of the eight registers specified by address
- Use Mux8Way16 chip to select the 16 bit contents of register specified by address on 16 bit ***out*** output

RAM512.hdl

```
CHIP RAM512 {
   IN in[16], load, address[9];
   OUT out[16];
   PARTS:
   DMux8Way(in=load, sel=address[6..8], a=load0, b=load1, c=load2, d=load3, e=load4, f=load5, g=load6, h=load7);

   RAM64(in=in, load=load0, address=address[0..5], out=out0);
   RAM64(in=in, load=load1, address=address[0..5], out=out1);
   RAM64(in=in, load=load2, address=address[0..5], out=out2);
   RAM64(in=in, load=load3, address=address[0..5], out=out3);
   RAM64(in=in, load=load4, address=address[0..5], out=out4);
   RAM64(in=in, load=load5, address=address[0..5], out=out5);
   RAM64(in=in, load=load6, address=address[0..5], out=out6);
   RAM64(in=in, load=load7, address=address[0..5], out=out7);

   Mux8Way16(a=out0, b=out1, c=out2, d=out3, e=out4, f=out5, g=out6, h=out7, sel=address[6..8], out=out);
}
```

4K-Register/words RAM



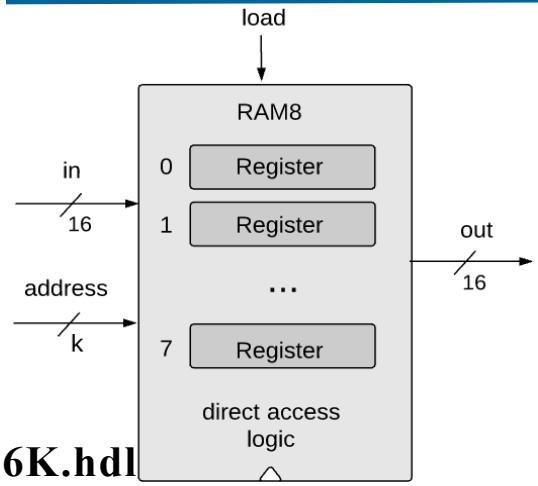
Implementation tips:

- Memory of **4K** registers, each 16 bit-wide. ***Out*** holds the value stored at the memory location specified by address. If load==1, then the in value is loaded into the memory location specified by address (the loaded value will be emitted to out from the next time step onward)
- Feed the 16 bit ***in*** value to all the registers, simultaneously
- Use DMux8Way chip to select one of the eight registers specified by address
- Use Mux8Way16 chip to select the 16 bit contents of register specified by address on 16 bit ***out*** output

RAM4K.hdl

```
CHIP RAM4K {
   IN in[16], load, address[12];
   OUT out[16];
   PARTS:
  DMux8Way(in=load, sel=address[9..11], a=load0, b=load1, c=load2, d=load3, e=load4, f=load5, g=load6, h=load7);

   RAM512(in=in, load=load0, address=address[0..8], out=out0);
   RAM512(in=in, load=load1, address=address[0..8], out=out1);
   RAM512(in=in, load=load2, address=address[0..8], out=out2);
   RAM512(in=in, load=load3, address=address[0..8], out=out3);
   RAM512(in=in, load=load4, address=address[0..8], out=out4);
   RAM512(in=in, load=load5, address=address[0..8], out=out5);
   RAM512(in=in, load=load6, address=address[0..8], out=out6);
   RAM512(in=in, load=load7, address=address[0..8], out=out7);

  Mux8Way16(a=out0, b=out1, c=out2, d=out3, e=out4, f=out5, g=out6, h=out7, sel=address[9..11], out=out);
}
```

16K-Register/words RAM



Implementation tips:

- Memory of **16K** registers, each 16 bit-wide. ***Out*** holds the value stored at the memory location specified by address. If load==1, then the in value is loaded into the memory location specified by address (the loaded value will be emitted to out from the next time step onward)
- Feed the 16 bit ***in*** value to all the registers, simultaneously
- Use DMux8Way chip to select one of the eight registers specified by address
- Use Mux8Way16 chip to select the 16 bit contents of register specified by address on 16 bit ***out*** output

RAM16K.hdl

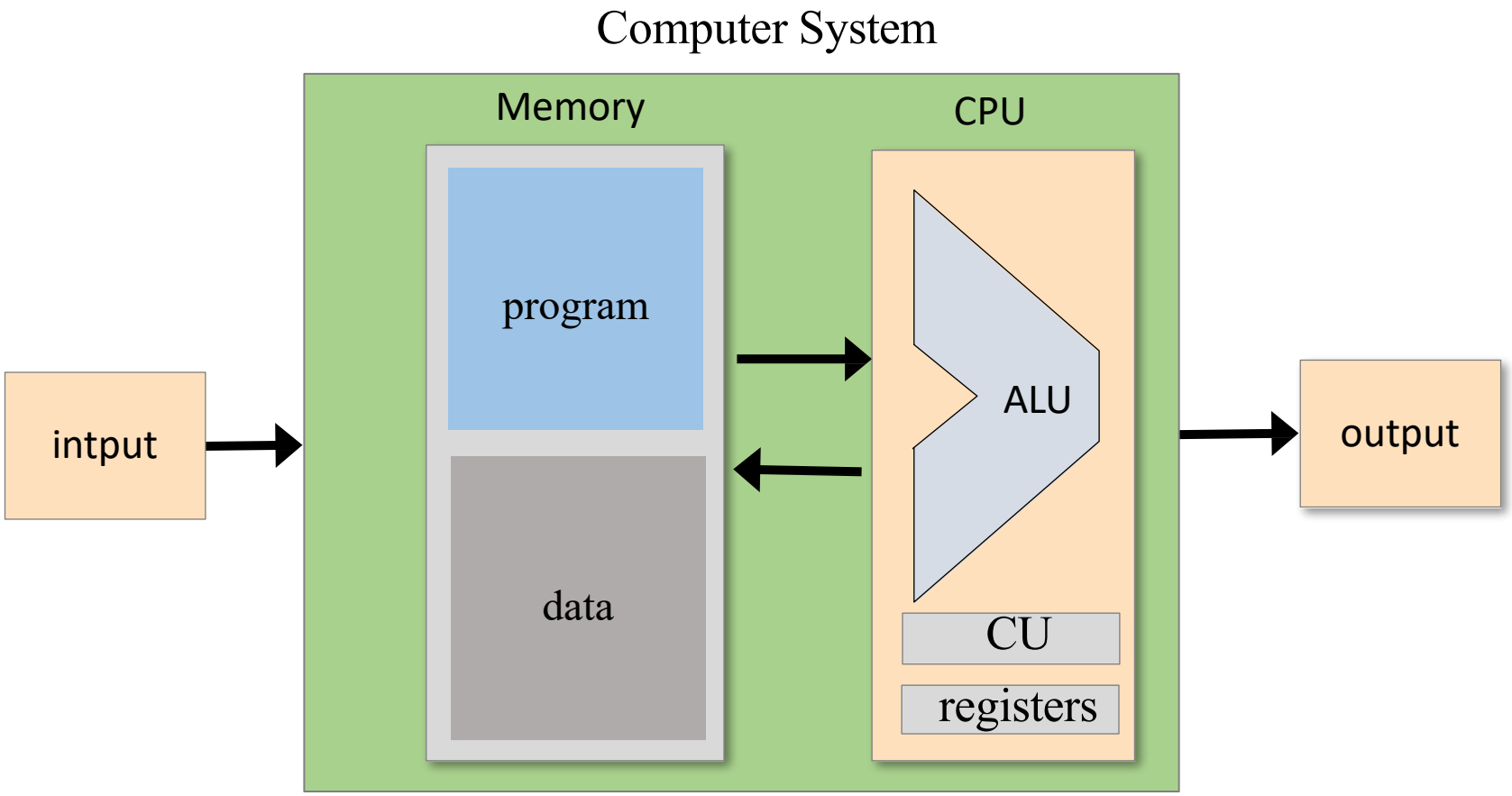
```
CHIP RAM16K {
   IN in[16], load, address[14];
   OUT out[16];
   PARTS:
   DMux4Way(in=load, sel=address[12..13], a=load0, b=load1, c=load2, d=load3);

   RAM4K(in=in, load=load0, address=address[0..11], out=out0);
   RAM4K(in=in, load=load1, address=address[0..11], out=out1);
   RAM4K(in=in, load=load2, address=address[0..11], out=out2);
   RAM4K(in=in, load=load3, address=address[0..11], out=out3);

   Mux4Way16(a=out0, b=out1, c=out2, d=out3, sel=address[12..13], out=out);
}
```

Counters

Overview of Counters

- A counter is a special type of register that goes through a pre-determined sequence of states upon the application of input pulses
- **Counters are used for:**
  - Counting the number of occurrences of an event
  - Keeping time or calculating amount of time between events
  - Baud rate generation
- **A w-bit counter consists of two main elements:**
  - A w-bit register to store a w-bit value
  - A combinational logic to
    - Compute the next value (according to a specific counting function)
    - Load a new value of user/programmer choice
    - Reset the counter to a default value
- **Examples:**
  - Simple Up/Down Binary Counters
  - BCD Counter(s)
  - Gray Code Counter
  - Ring Counter
  - Johnson Counter

Why we Need Counter Chip for Hack CPU



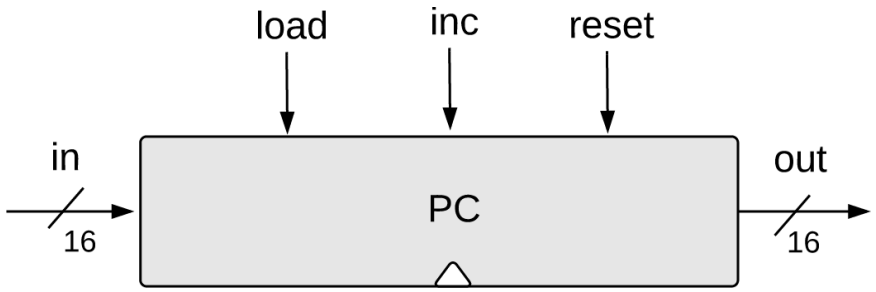
Why we Need Counter Chip for Hack CPU

- Consider a counter chip designed to contain the address of the instruction that the computer should fetch and execute next
- In most cases, the counter has to simply increment itself by 1 in each clock cycle, thus causing the computer to fetch the next instruction in the program
- In other cases, we may want the program to ***jump to an instruction at memory address n***, so the programmer want to set the counter to a value of ***n***, rather than its default counting behavior with ***n+1***, ***n+2***, and so forth
- Finally, the program's execution can be restarted anytime by resetting the counter to 0, assuming that the address of the program's first instruction
- In short, we need a loadable and resettable counter

Program Counter Register

- Every computer has a special register called the Program Counter, normally called the PC, which keeps track of the instruction to be fetched and executed next
- The PC is designed to support three possible control operations:
  - **Reset:** Fetch the first instruction **PC = 0**
  - **Next:** Fetch the next instruction **PC++**
  - **Goto:** Fetch instruction at address **n** **PC = n**

Counter Abstraction



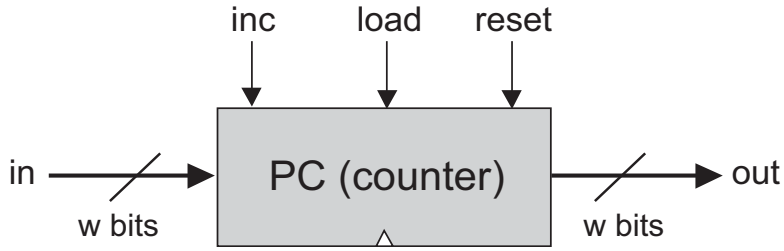

```
if reset[t] = 1 then
     out[t+1] = 0
else if load[t] = 1 then
     out[t+1] = in[t]
else if inc[t] = 1 then
     out[t+1] = out[t] + 1
else out[t+1] = out[t] //do nothing
```

PC = 0

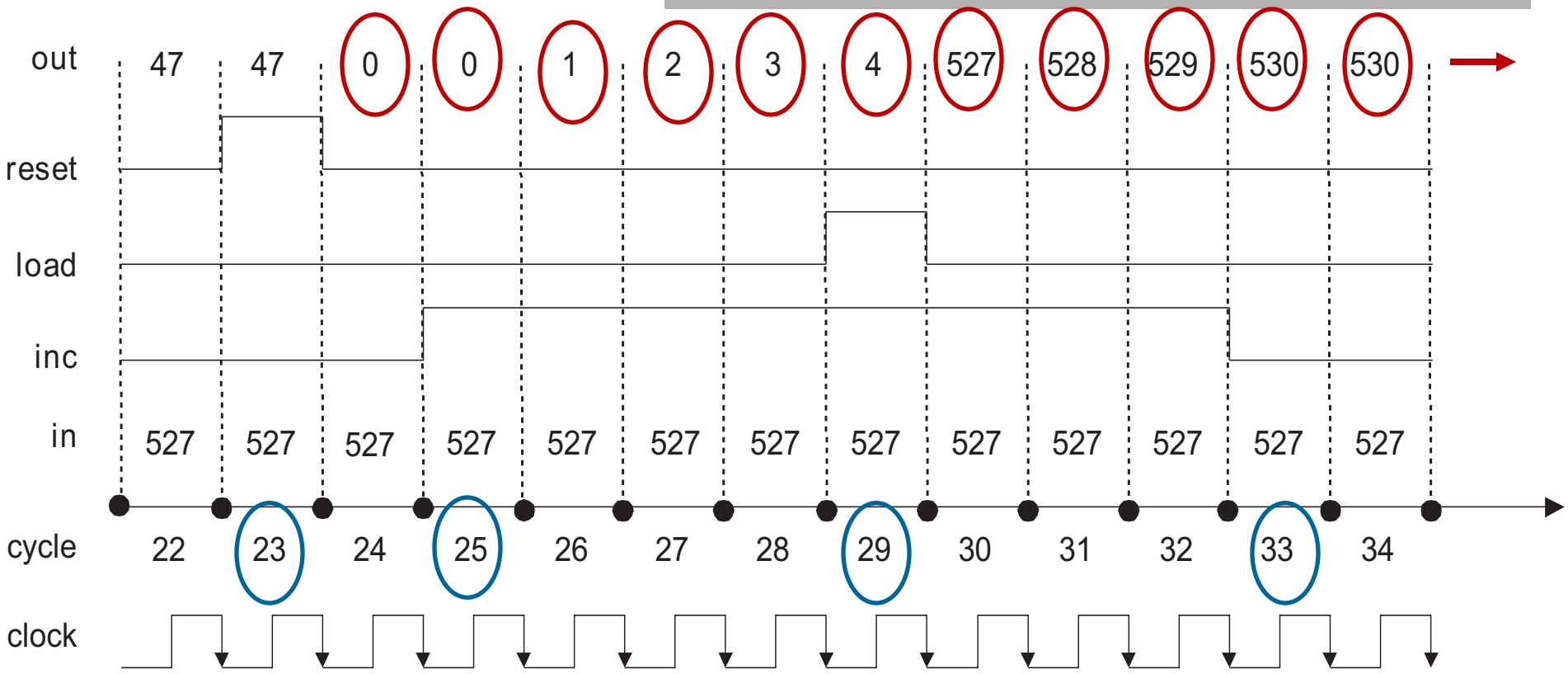
PC = in

PC++

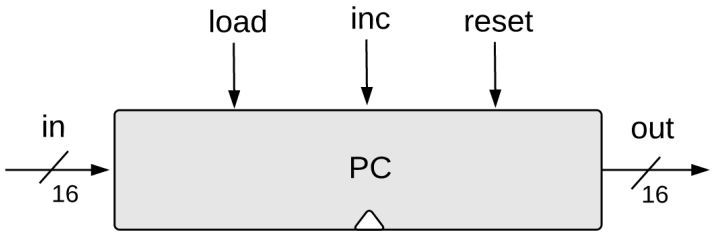
Counter Simulation




```
Chip name:  PC  // 16-bit counter
Inputs:     in[16], inc, load, reset
Outputs:    out[16]
Function:   If reset(t-1) then out(t)=0
               else if load(t-1) then out(t)=in(t-1)
                    else if inc(t-1) then out(t)=out(t-1)+1
                         else out(t)=out(t-1)
Comment:    "=" is 16-bit assignment.
            "+" is 16-bit arithmetic addition.
```



16 Bit Program Counter Implementation



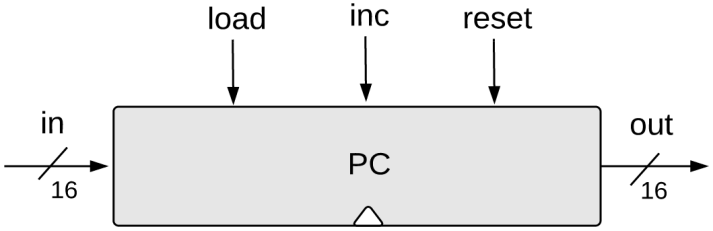

```
CHIP Register {
     IN in[16], load;
     OUT out[16];
   PARTS:
     Bit(in=in[0], load=load, out=out[0]);
     Bit(in=in[1], load=load, out=out[1]);
     Bit(in=in[2], load=load, out=out[2]);
     Bit(in=in[3], load=load, out=out[3]);
           . . . .
     Bit(in=in[15], load=load, out=out[15]);
}
```

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

```
CHIP Inc16 {
  IN in[16];
  OUT out[16];
  PARTS:
   Add16(a=in, b[0]=true, out=out);
}
```

```
CHIP Add16 {
  IN a[16], b[16];
  OUT out[16];
  PARTS:
   HalfAdder(a=a[0], b=b[0], sum=out[0], carry=carry0);
   FullAdder(a=a[1], b=b[1], c=carry0, sum=out[1], carry=carry1);
   FullAdder(a=a[2], b=b[2], c=carry1, sum=out[2], carry=carry2);
   FullAdder(a=a[3], b=b[3], c=carry2, sum=out[3], carry=carry3);
   .....
   FullAdder(a=a[14], b=b[14], c=carry13, sum=out[14], carry=carry14);
   FullAdder(a=a[15], b=b[15], c=carry14, sum=out[15], carry=carry15);
}
```

16 Bit Program Counter Implementation



PC.hdl

```
CHIP PC {
   IN in[16], load, inc, reset;
   OUT out[16];

   PARTS:
      Inc16(in=regContent, out=incremented);
//if (inc == 1)
      Mux16(a=regContent, b=incremented, sel=inc, out=value1);
//else if (load == 1)
      Mux16(a=value1, b=in, sel=load, out=value2);
//else if (reset == 1)
      Mux16(a=value2, b=false, sel=reset, out=value3);
//else
      Register(in=value3, load=true, out=regContent, out=out);
}
```

```
CHIP Mux16 {
  IN a[16], b[16], sel;
  OUT out[16];
  PARTS:
   Mux(a=a[0], b=b[0], sel=sel, out=out[0]);
   Mux(a=a[1], b=b[1], sel=sel, out=out[1]);
             . . . .
   Mux(a=a[1], b=b[15], sel=sel, out=out[15]);
}
```

Program Counter Demo

Things To Do

- Perform testing of the chips designed in today's session on the h/w simulator. You can download the .hdl, .tst and .cmp files of above chips from the course bitbucket repository:

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

- Interested students should also try to design, implement and simulate binary down counter, cascaded BCD counter, Gray Counter, Ring counter, and Johnson counter

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