Coursera: Build a Modern Computer

Questions:

- How are computers built?
- How do you go from transistors to full computer programs?
- What are the different levels of abstraction involved in a computer?

Goals:

- Understand how a computer works
- "Build" a computer from the simplest parts

Notes

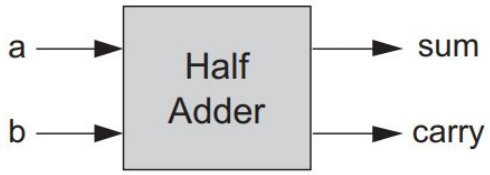
Week 1

- Abstraction is fundamental in computers
- **Everything starts with transistors**
- **Transistors allow us to build "logic gates"**
- There's the AND logic gate, the OR logic gate, the NOT logic gate, and others…
- **You can build all other logic gates from NAND (not and) gates** (and also NOR gates, but NAND gates are more efficient due to EE constraints)
- Computer chips usually just have millions of NAND gates
- With logic gates you can evaluate boolean logic (0's and 1's as inputs and outputs)

Week 2

- Why do we need logic gates? To evaluate boolean logic and do operations on binary numbers
- **We only need two types of operations to successfully add (and subtract!) binary numbers: Adding two bits, and adding three bits**
- Adding two bits is self-explanatory.
- We do this with a chip called **"Half Adder"** and we can use the logic gates we built earlier

Inputs		Outputs	
a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



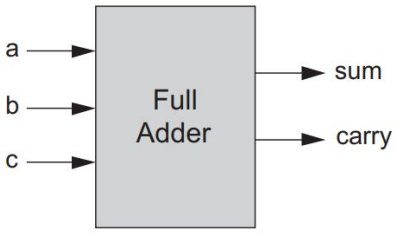
- Why add three bits? Because when we add two binary numbers, much like when we add decimal numbers, we can have “carry” numbers that are “carried” to the next operation.

E.g.: Adding 01 and 11

```
                 1
    01          01
+   11    ->   +11
    ----        -----
                101
```

- We do this with a **“Full Adder,”** which can be built by using two Half Adders

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

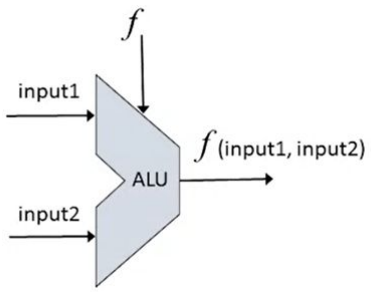


- Once we have these adders, we can add any number of bits (16, 32, 64, etc.)
- How do we subtract? To subtract x from y, we add the negative of x to y
- In binary, negative numbers are represented with two’s complement:

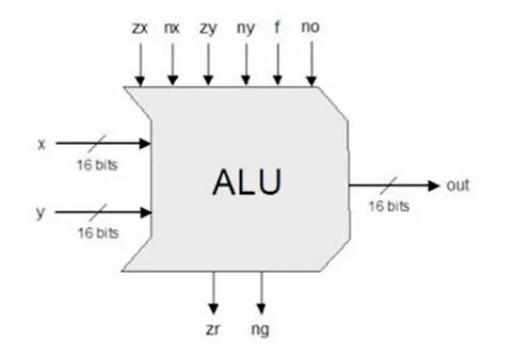
0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	-8	(8)
1001	-7	(9)
1010	-6	(10)
1011	-5	(11)
1100	-4	(12)
1101	-3	(13)
1110	-2	(14)
1111	-1	(15)

- Two’s complement makes it easy to do operations with negative numbers in binary
- To convert a number to its negative, you flip all its bits and add 1
- Therefore, fundamentally, all operations on bits boil down to two processes
- The **Arithmetic Logic Unit, or ALU, uses these two bit operations, plus the logic gates we built, to do operations (either logical(and, or, xor) or mathematical(adding, subtracting)) on larger bits.** The ALU is a central part of the CPU.

- The operations the ALU can handle are custom for every computer. This is a general representation:



- For this project, the detailed ALU will look more like this:



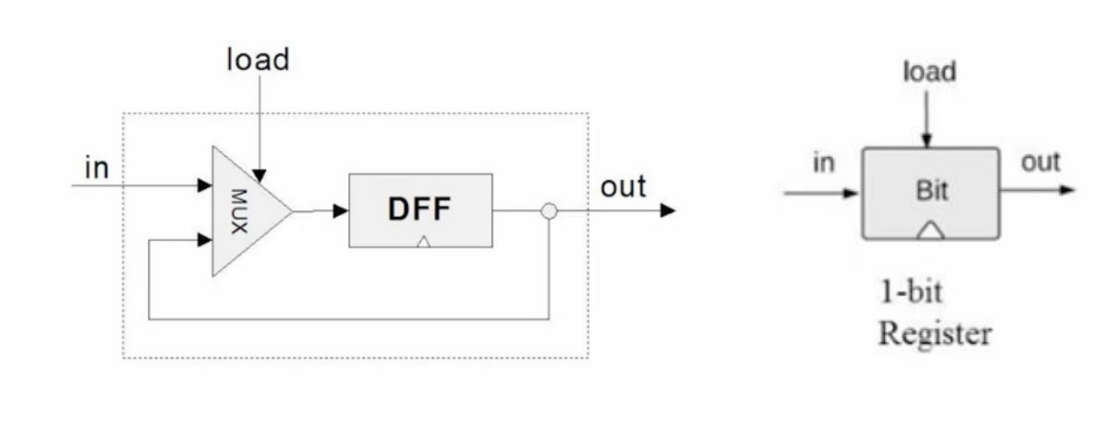
- It has two 16-bit inputs, one 16-bit output, six input function bits, and two control output bits
- These are the functions the ALU can handle in this project. The functions happen sequentially in order:
    - zx(zero x): if true, turn x to 0
    - ng(negate x): if true, turn x to -x
    - zy(zero y): if true, turn y to 0
    - ny(negate y): if true, turn y to -y
    - f(function): if true, do addition. If false, do &(“anding)
    - no(negate output): if true, turn output to -output
    - out(output)

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
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	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

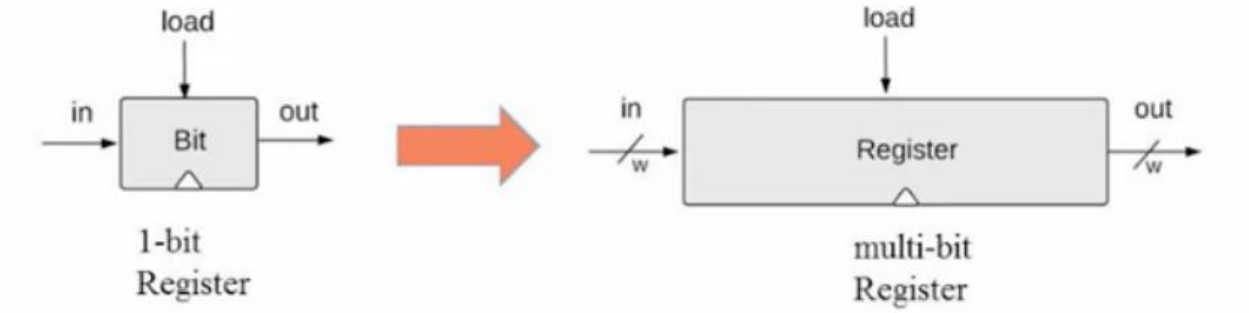
- The two control output bits are:
    - zr(zero): 1 if output is 0
    - ng(negative): 1 if output is negative

Week 3

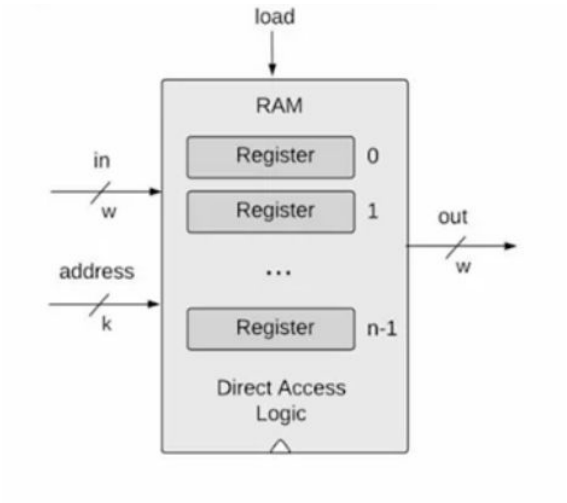
- It's not only needed to do calculations. We also need to have a sense of "time" or sequentiality
- Sequentiality allows us to chain operations (have the output of one function as the input of another)
- Flip Flops / Latches allow us to remember states by flipping between and storing two states (1 or 0)
- Flip Flops / Latches allow us to remember states **for a single time unit** (or step)
- If we want to remember a state for longer, we use flip flops/latches to build a 1-bit register. Here's one implementation, using the multiplexer chip that we built before.



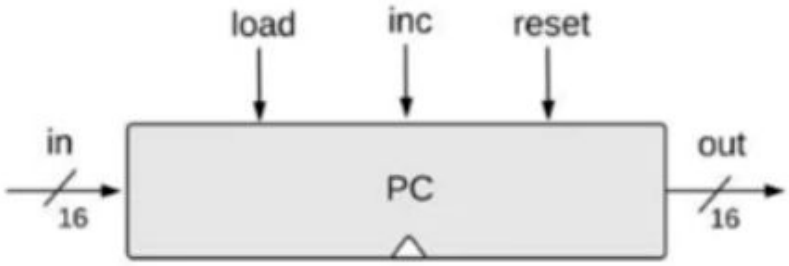
-
- A multi-register is multiple 1-bit registers in a row (8, 16, 32, 64, etc)



- RAM is made up of a sequence of n addressable registers stores temporary data and instructions



- At any point in time, only one register in the RAM is selected using an address
- To have a bigger RAM, we continue stacking these chips to get 8, 64, 512, 4k, 16k bytes of addressable memory
- A Program Counter is a chip that can keep track of which instruction should be executed next
- The PC contains the address of the instruction that will be fetched and executed
- It can reset (fetch first instruction), next (fetch next instruction), and goto(go to instruction n)



Week 4

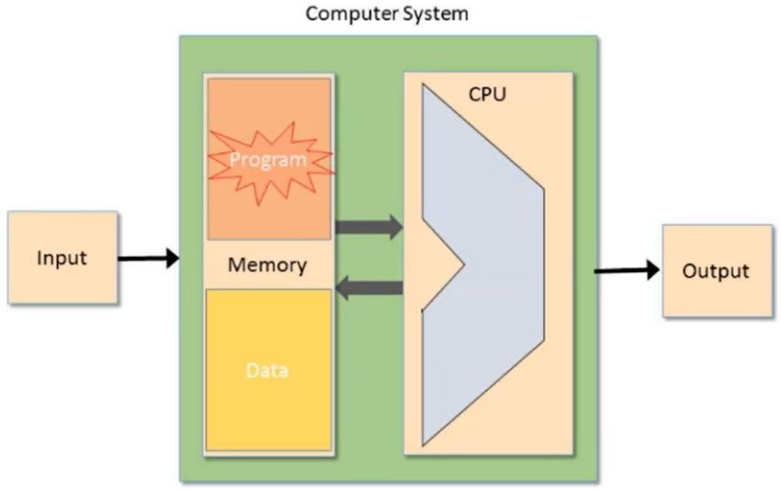
- To instruct the computer to do operations (so that we can run programs), we use instructions. These instructions are in reality a sequence of bits (e.g.: 0100101010). To make this easier to read, we can have a symbolic interpretation of the same instruction (e.g.: ADD R2 R5). Humans can program in the “Assembly” language and then the assembler can convert these instructions into a bit sequence that the machine can execute.
- The assembly language we will be using will be the Hack programming language
- The Hack language has a range of functionality: ways to retrieve memory from a register, ways to signal an operation, ways to jump to a set of instructions, etc.

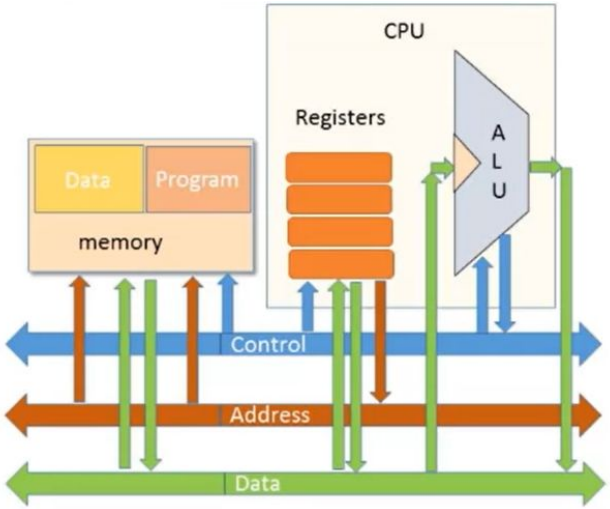
- The screen and the keyboard (and probably other devices) have configurations that affect the RAM on the computer
- For the screen, there is a section of the RAM dedicated to it, where each bit represents one pixel. If the bit is on (1), the pixel on the screen is black. If it is off (0), the pixel is white.
- For the keyboard, one of the RAM registers is dedicated to the output. If the keyboard is not pressed, this register contains 0. If a key is pressed, the value of this key is shown in the keyboard in ASCII

Week 5

CPU Archtecture

- Universal Turing Machine (Theory) and Von Neumann Architecture (Practical): One computer can run any program

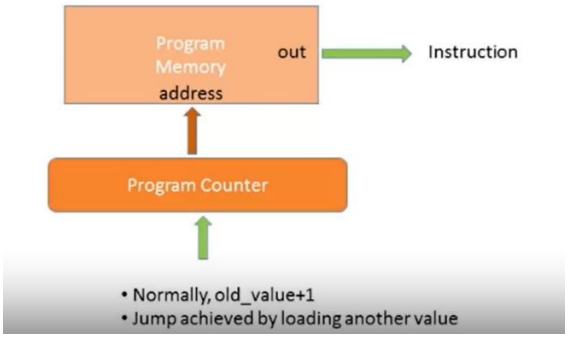




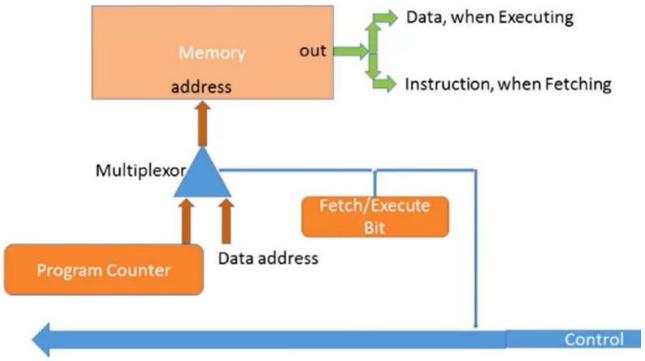
- Two main parts: the memory and the CPU.
- Three types of information pass between the memory and the CPU: Control bits, Addresses, and Data
- This information is passed by wires, which we call busses. There is a Control bus, an Address bus, and a Data bus
- The ALU connects to the Data bus (to perform operations on data) and the Control bus (to know which operation to execute and to signal other parts of the system)
- The Registers in the CPU (temporary) need to be connected to the Data bus (to do and store operations) and the Address bus (to connect to memory)
- The Memory connects to the Address bus (to store and send objects in memory) and the Data bus (to send and receive data)
    - Data Memory: Needs to get an address where to store the data, plus a way to send/receive the data. Connected to Address bus (input) and Data bus (input/output)
    - Program Memory: Needs to get an address to put the next instruction and can send data or control instructions to the rest of the system. Connected to Address bus (input) and Data and Control busses (Output)

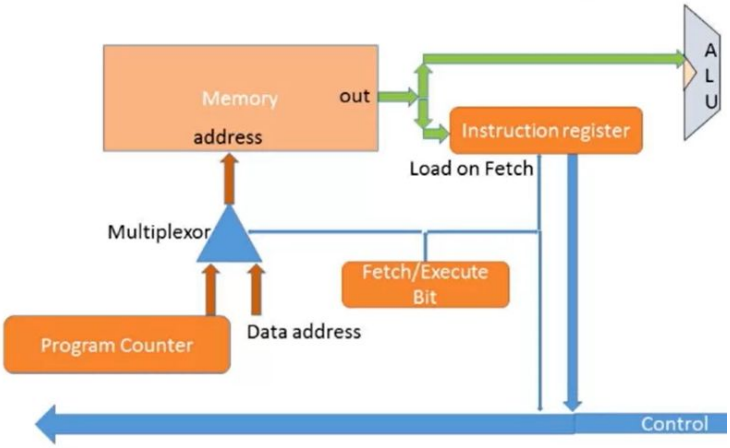
Fetch-Execute Cycle

- Put location of next instruction into address of program memory and get the instruction code itself by reading the memory at that location
- We use the program counter to know which instruction to access in the program memory

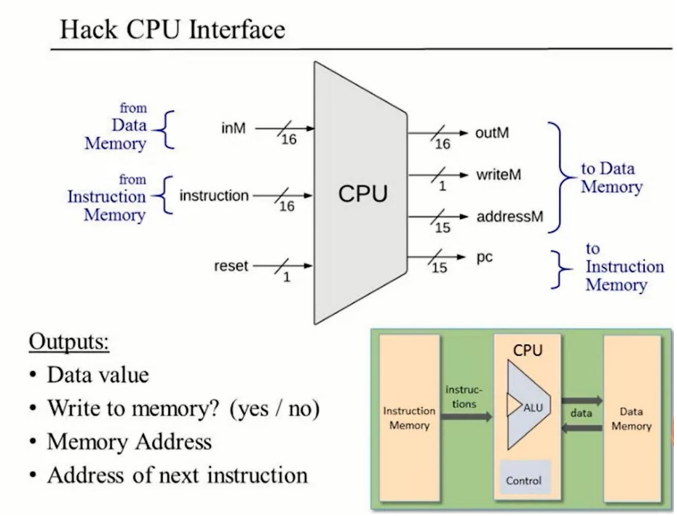


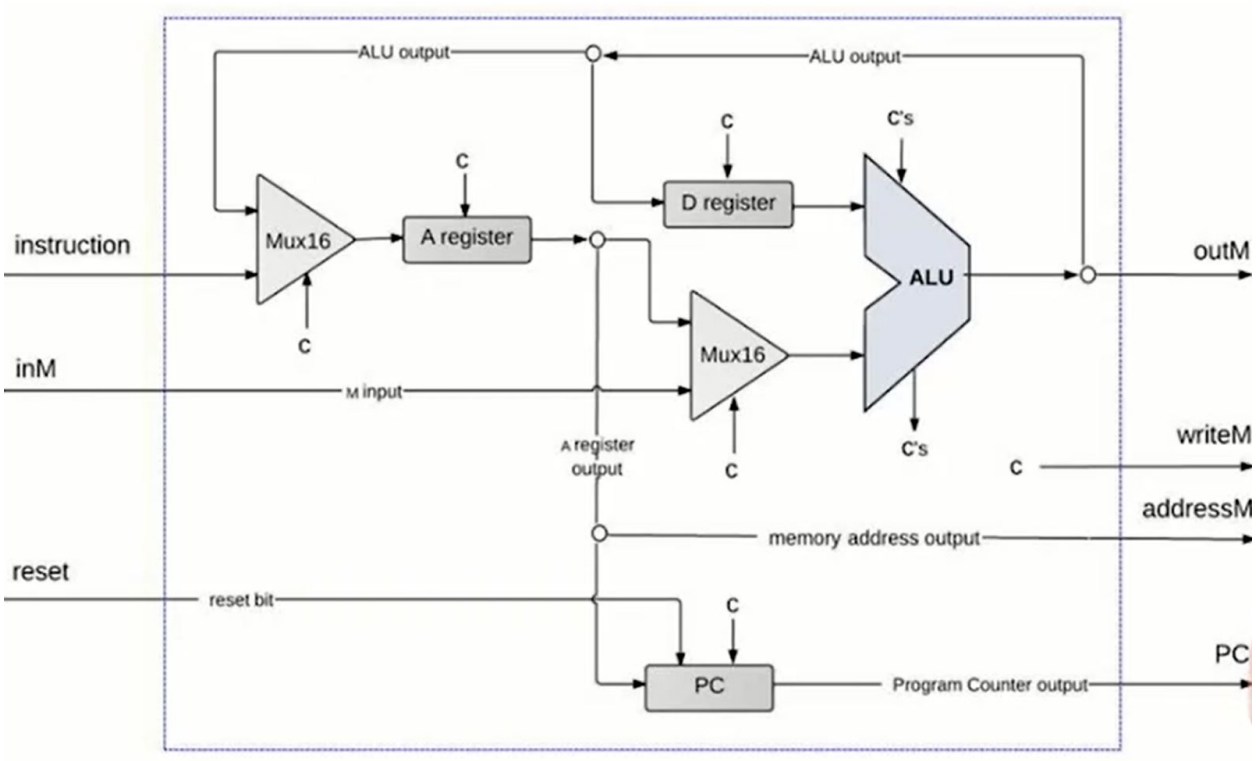
- The instruction code already has all the information to execute what to do
- The instruction feeds into the Control bus to control the rest of the system
- To avoid a clash between the Fetch and the Execute cycles, we use a mulitplexer to set the memory address in the appropriate location for the corresponding cycle





- A shortcut to address this problem is to separate the memory into two sections: the data memory and the program memory. This is known as the Harvard architecture.

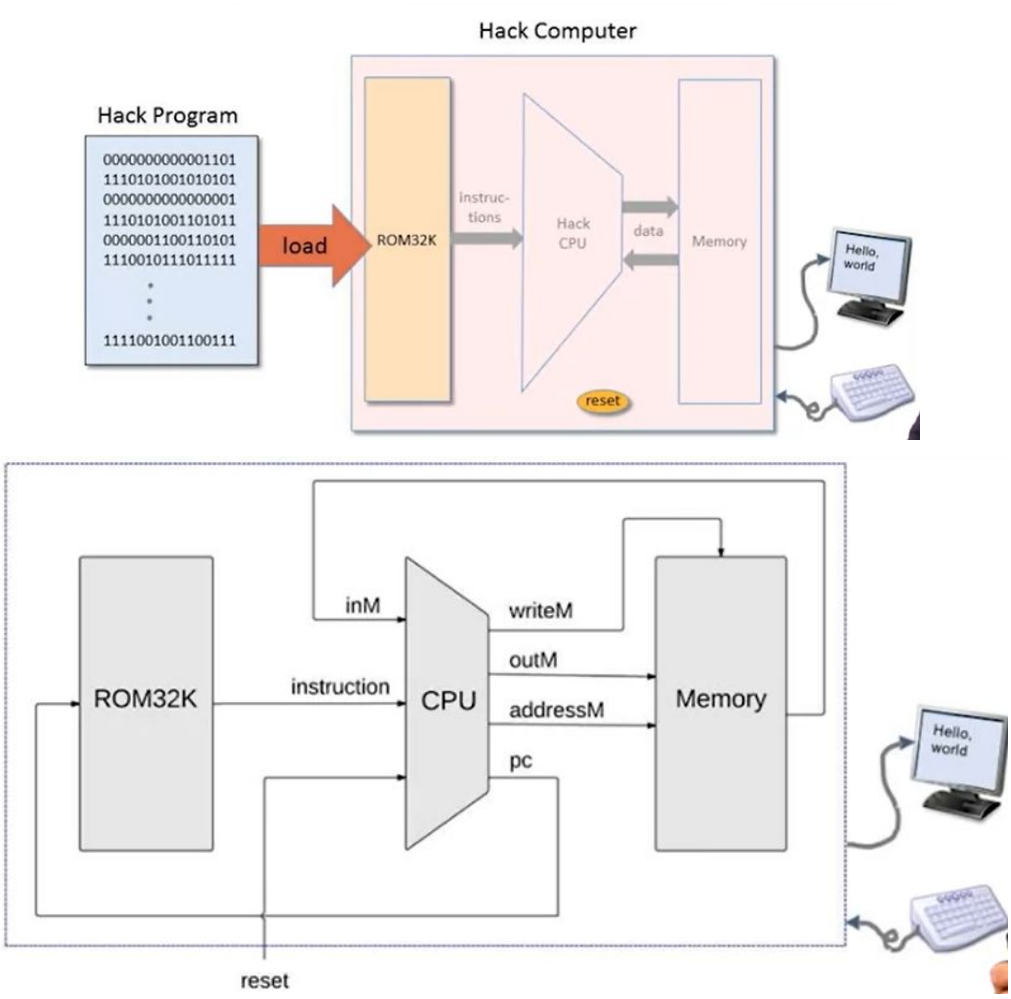




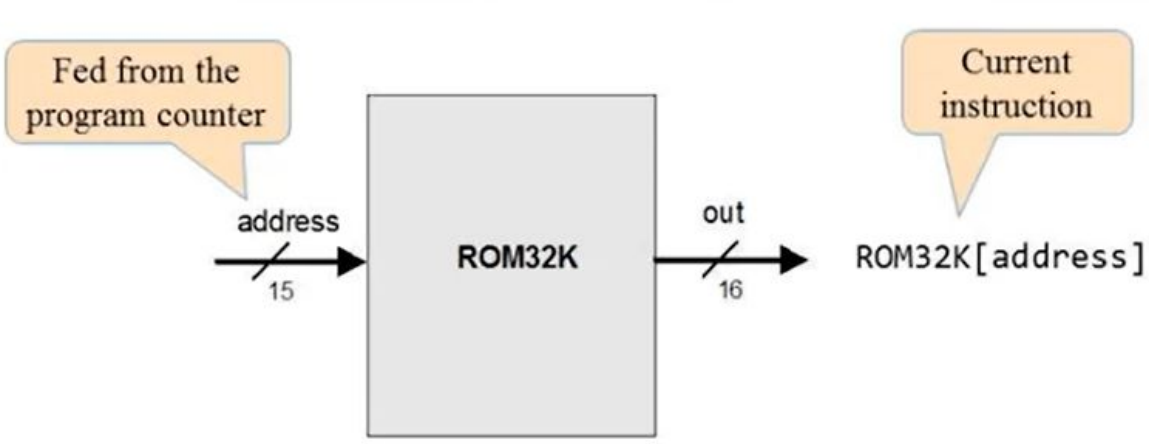
- How does the CPU execute an instruction? The CPU receives an instruction in binary, like the following: `111accccccdddjjj`
- The format of this instruction has been established previously so that the CPU knows what to do.
- Each section of bits in the instruction tells the CPU which control bits to activate to signal what operations to run in the ALU, where to store the output in the registers, and how to change the Program Counter to indicate where to jump after the operation is executed.

Building a Computer

- How do we put everything together?



- The CPU communicates to the RAM, which stores the memory registers plus the screen and keyboard mappings.
- The instruction code of the program that we want to execute is loaded to the ROM (Read-Only Memory). Once the reset button is pressed, the Program Counter starts executing these instructions.
- How do we load a program into the ROM? Multiple ways. Video Games, for example, use plug-and-play chips.
- How does the ROM work? It receives as input the address that we want to access, and it outputs the value of the program at that address.


Fed from the program counter
address
15
ROM32K
out
16
Current instruction
ROM32K[address]