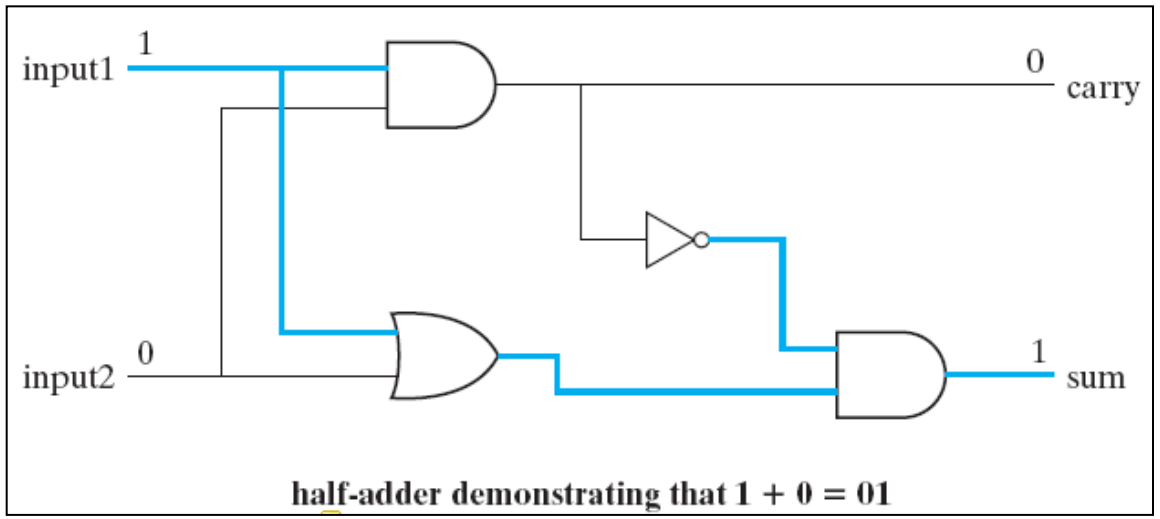
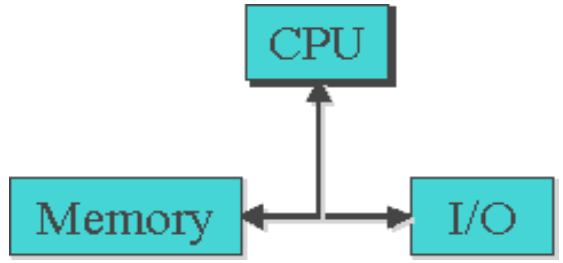
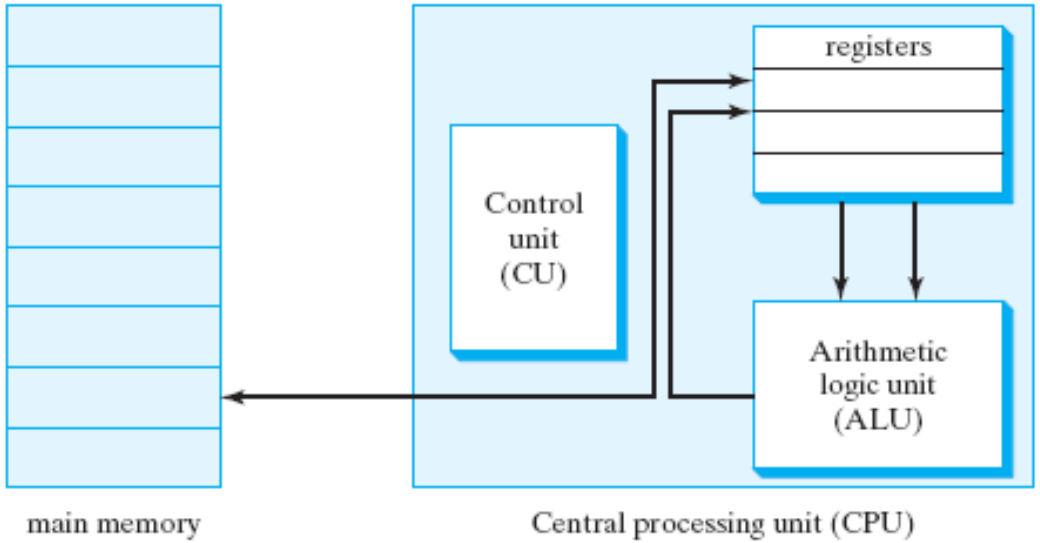
2. From logic gates to computer components



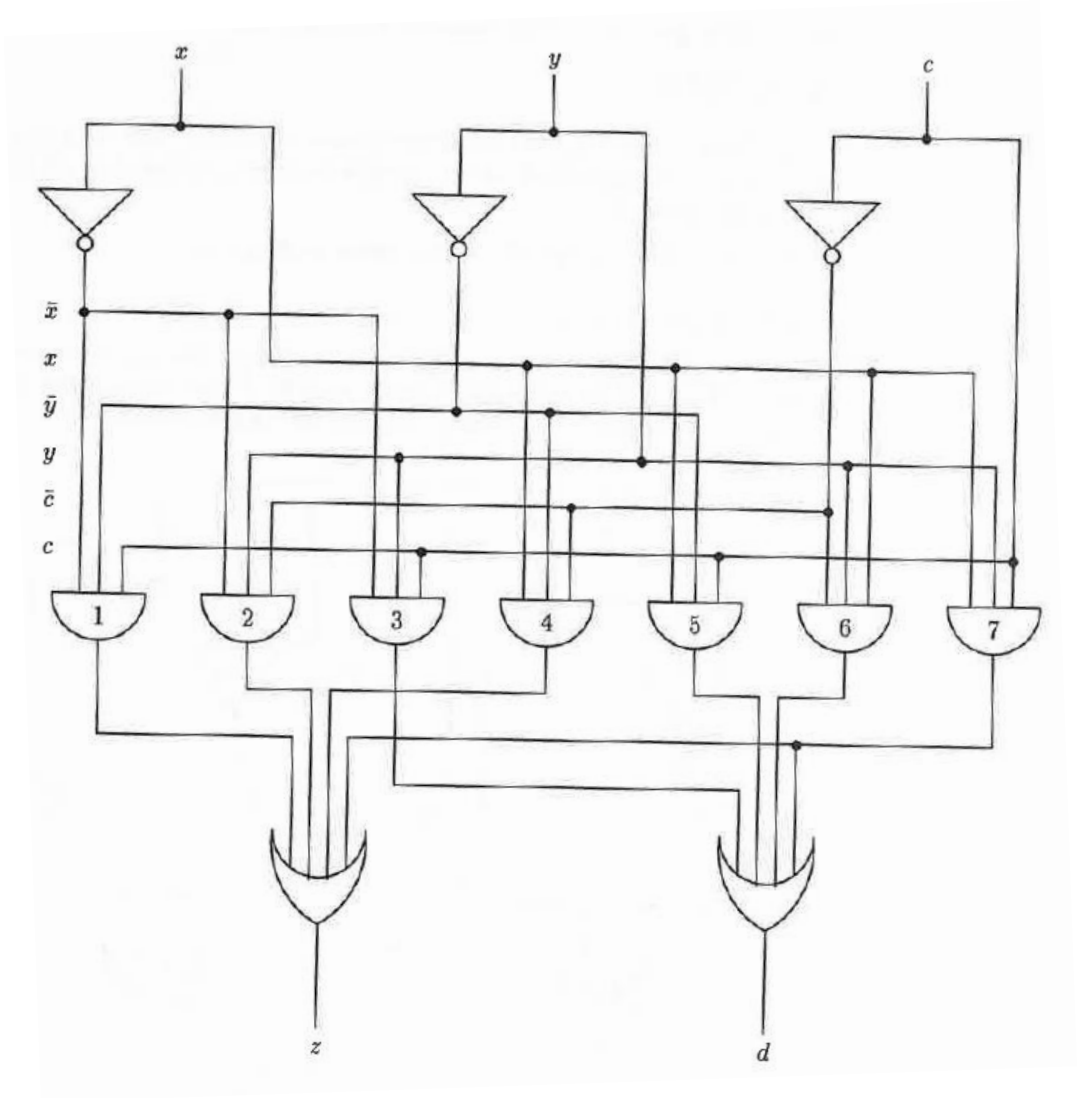
half-adder demonstrating that 1 + 0 = 01





Connections between logic gates

- When drawing more complex logic circuits, it is often needed to cross the lines that connect gates to each other
  - Is there a junction or not?
- Good way to improve notation:
  - If there's a black dot ("node") marked at the point of intersection, it's a junction
  - If there's no black dot, then the signals don't get mixed up
- Usually this notation is not actually needed, but it makes life easier
  - Especially in case feedback loops occur



Half adder

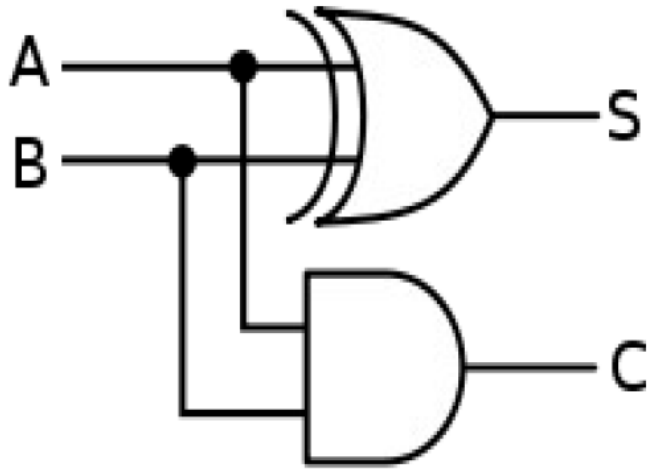
- A logic circuit that performs the addition of two 1-bit numbers is called a *half adder*
- Inputs: A and B
- Outputs: sum bit S and carry bit C
- Can be constructed in multiple ways, but the easiest way would be to use a XOR gate for the sum bit and an AND gate for the carry bit
- Disjunctive normal forms for output bits would be

$$S = AB' + A'B$$

$$C = AB$$

Why are there two outputs?

Input		Output	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Ripple-carry addition algorithm

- Let's revise how we have learned to perform addition of regular base 10 numbers in elementary school:

```
  1          0            4 5 6
4 5 6      4 5 6          8 2 9
8 2 9      8 2 9      ------------
-------    -------      1 2 8 5
      5          8 5
```

- Here we've used the *Ripple-carry addition algorithm*:
  - If the sum of numbers in a column goes to double digits, we carry the first number to the next column (in this example, 6+9 = 15, so we write 5 below and carry the 1 to the next column)
- The same logic applies for addition of binary numbers
  - So, the "carry bit" C is literally a carry bit!
  - Binary addition rules:

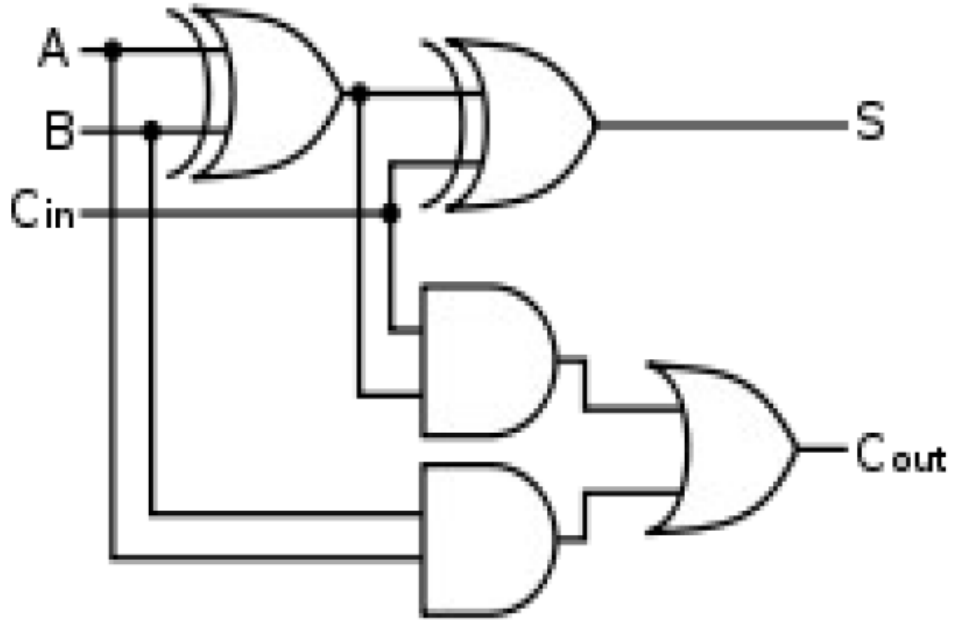
```
  0      1      0      1
+ 0    + 0    + 1    + 1
---    ---    ---    ----
  0      1      1     10
```

Full adder

- Calculation of numbers which are more than 1 bit long requires taking this carry bit into account in further columns
- This can be done by constructing a *full adder* from two half adders and an OR gate

Truth Table

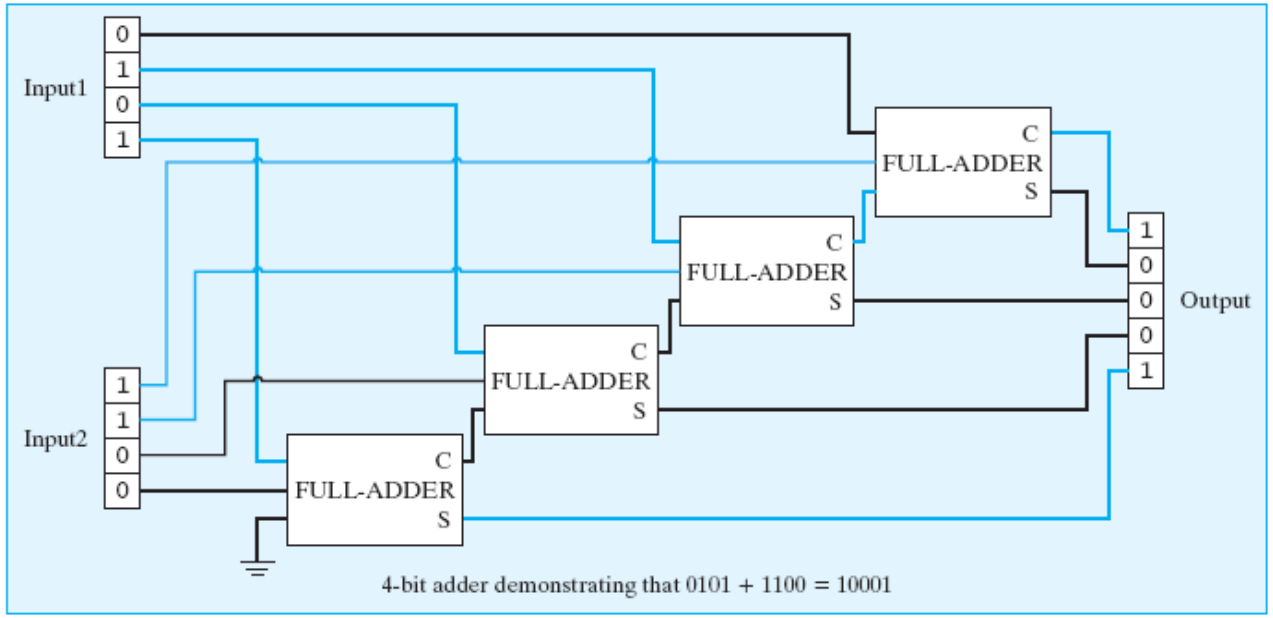
A	B	Carry-in	Sum	Carry-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Adder circuits

- By combining adders, we can construct *adder circuits* that are capable of performing addition calculations for much larger numbers
- Generally speaking: a circuit that performs the addition of two n-bit numbers consists of n pcs of full adder modules

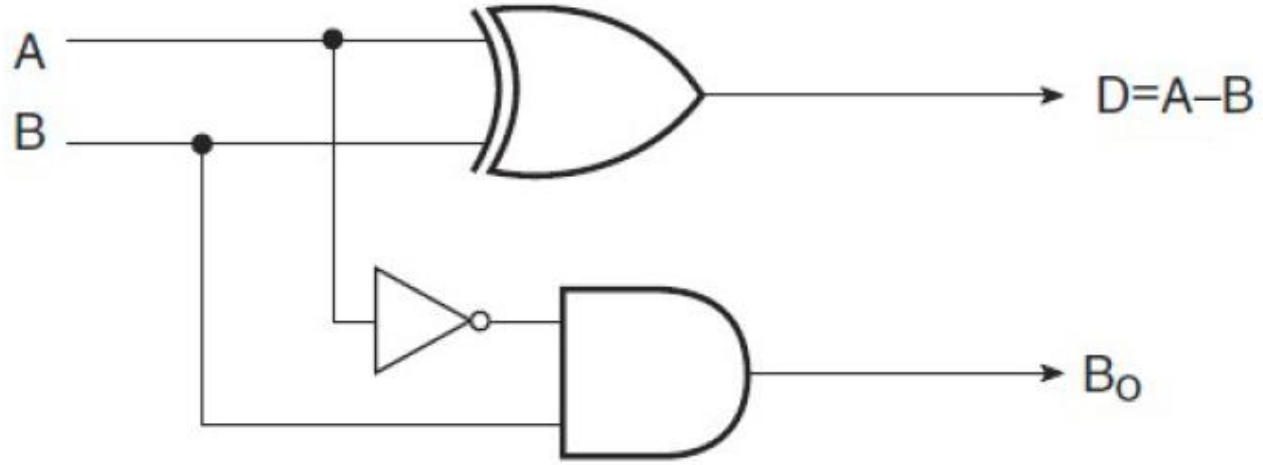
  - Example: 4-bit full adder
  - The last carry bit can be taken into account (if the end result is n+1 bits long), or if the output is also 4-bit, it can be neglected (overflow)



4-bit adder demonstrating that 0101 + 1100 = 10001

Half subtractor

- Likewise, a subtraction of two 1-bit numbers is possible to do using logic gates
- A circuit that does this is called a *half subtractor*
  - Identical to half adder with the exception of a NOT gate before the AND gate
- n-bit full subtractors can be constructed, too



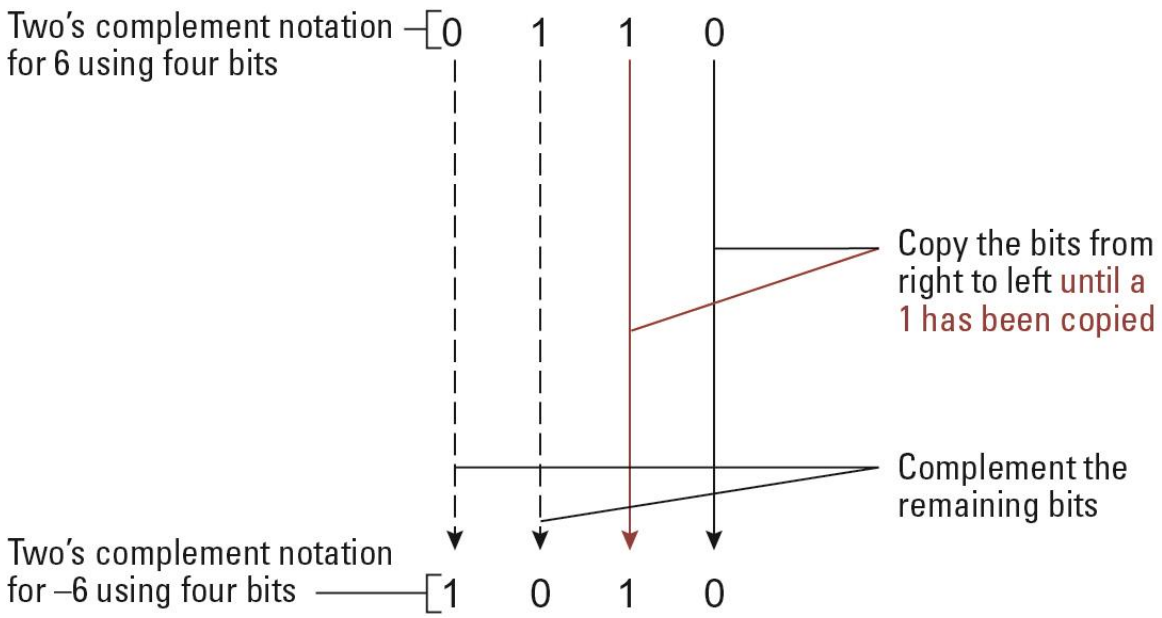
A	B	Bo	Di
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

$Di = A'B + AB'$

$Bo = A'B$

Two's complement

- Instead of half/full subtractors it is possible to use *two's complement:*
  - Transform the number that's to be subtracted to its complement (remember from FoIP?)
  - Then perform the addition as before



a. Using patterns of length three

Bit pattern	Value represented
011	3
010	2
001	1
000	0
111	–1
110	–2
101	–3
100	–4

b. Using patterns of length four

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	–1
1110	–2
1101	–3
1100	–4
1011	–5
1010	–6
1001	–7
1000	–8

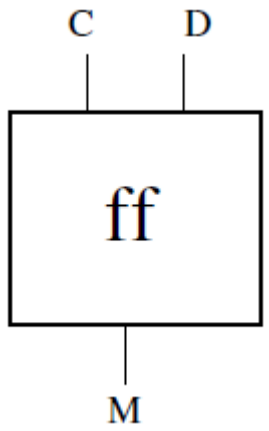
Two's complement, calculation examples

- Some examples calculated both in base 10 and in base 2 using two's complement
- (This is revision from FoIP course)

Problem in base 10	Problem in two's complement	Answer in base 10
3 + 2	0011 + 0010 = 0101	5
−3 + −2	1101 + 1110 = 1011	−5
7 + −5	0111 + 1011 = 0010	2

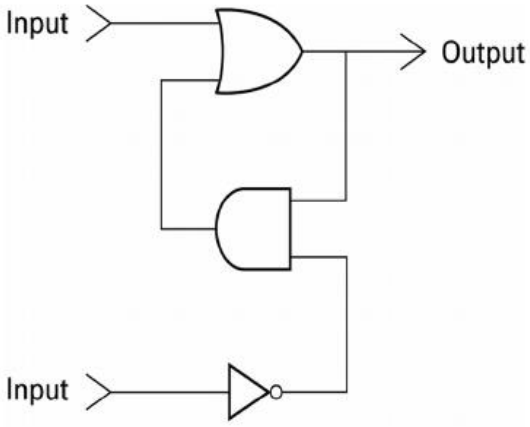
Flip-flops

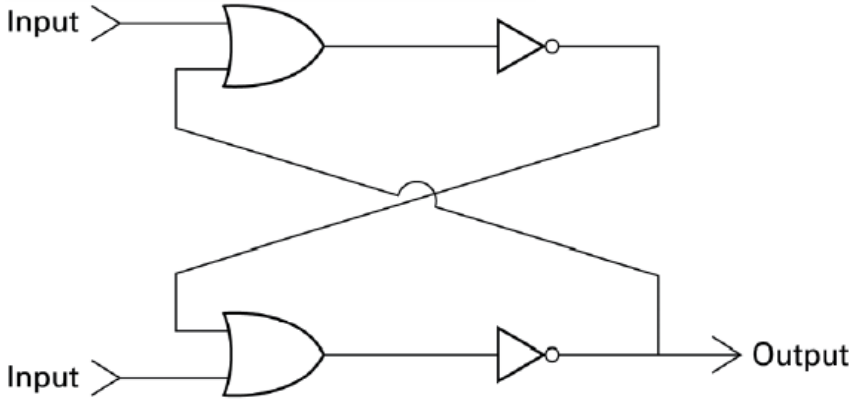
- A *flip-flop* is a logic circuit that has memory: its action depends not only on the inputs, but also on what's happened before
- In other words, it's capable of storing a bit value
- Flip-flops can be asynchronous or synchronous
  - Asynchronous flip-flops are called *latches*, and they change their state right away when the input signal changes
  - Synchronous flip-flops are controlled via clock signal; they change their state "on their turn"
- A flip-flop has two inputs:
  - Control bit (C)
  - Data bit (D)
- The output bit depends on C, D and previous value of M

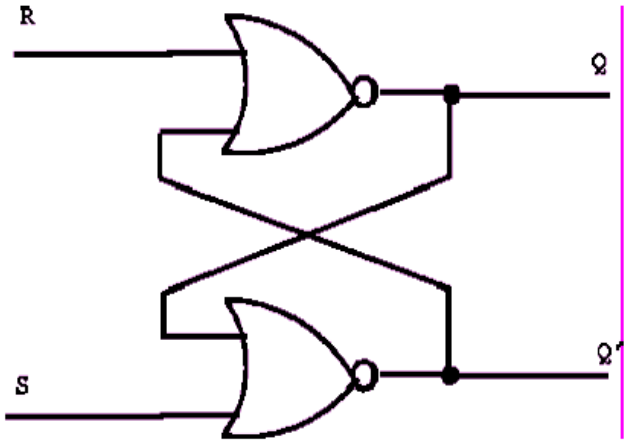


Flip-flops

- A flip-flop can be constructed in several ways using different gates – for example
  - OR, AND & NOT gates
  - 2 OR gates + 2 NOT gates (or, simply, 2 NOR gates)
- Common feature in all of them is a *feedback loop* (or two)
- Even a case of two outputs (result and its complement) is possible

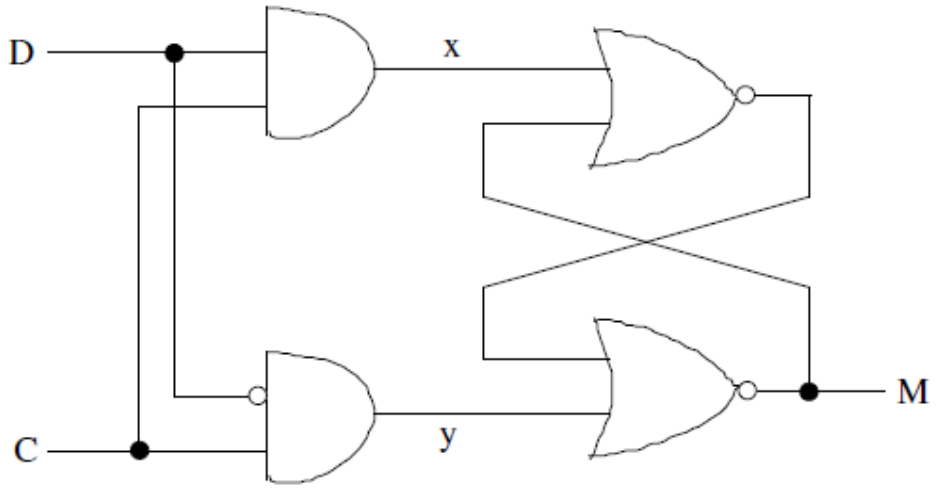






Flip-flops

- Because flip-flop remembers history, the truth table for a flip-flop must be constructed in such a way that the 3rd input is M_t and the output is M_{t+1}
- Some flip-flops may be equipped with a control part in order to prevent unwanted input combinations
  - This might be because they would cause the circuit to be unstable

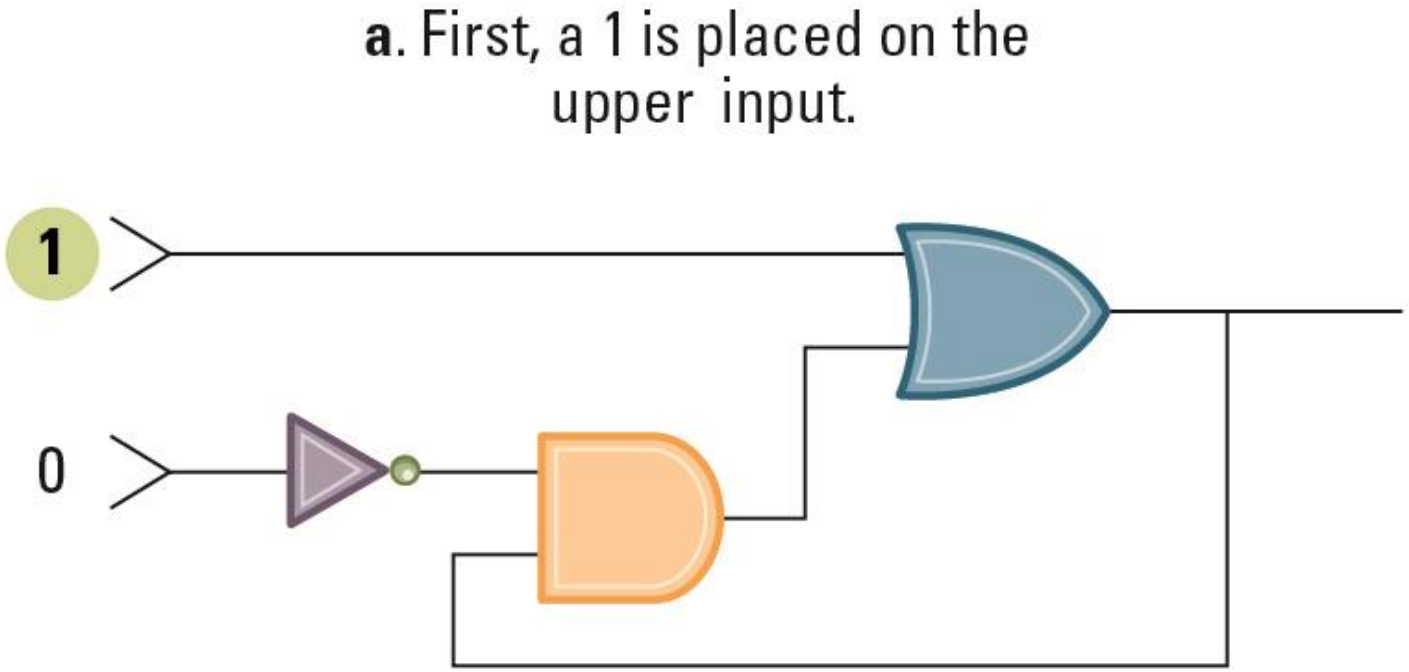


x	y	M_t	M_{t+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1

Here the control part (two AND gates) causes that x and y can never be 1 at the same time.

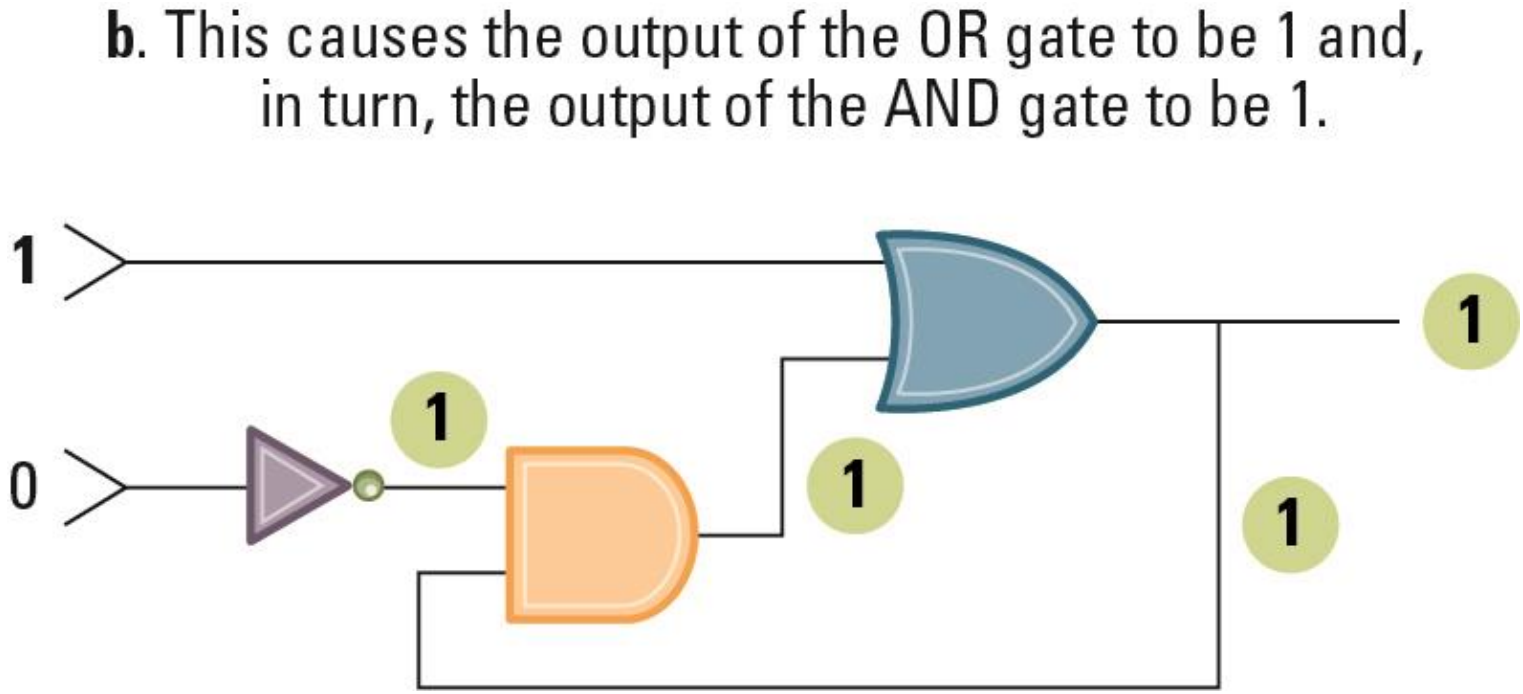
Simple flip-flop operation

- Let’s examine how the following simple flip-flop operates:



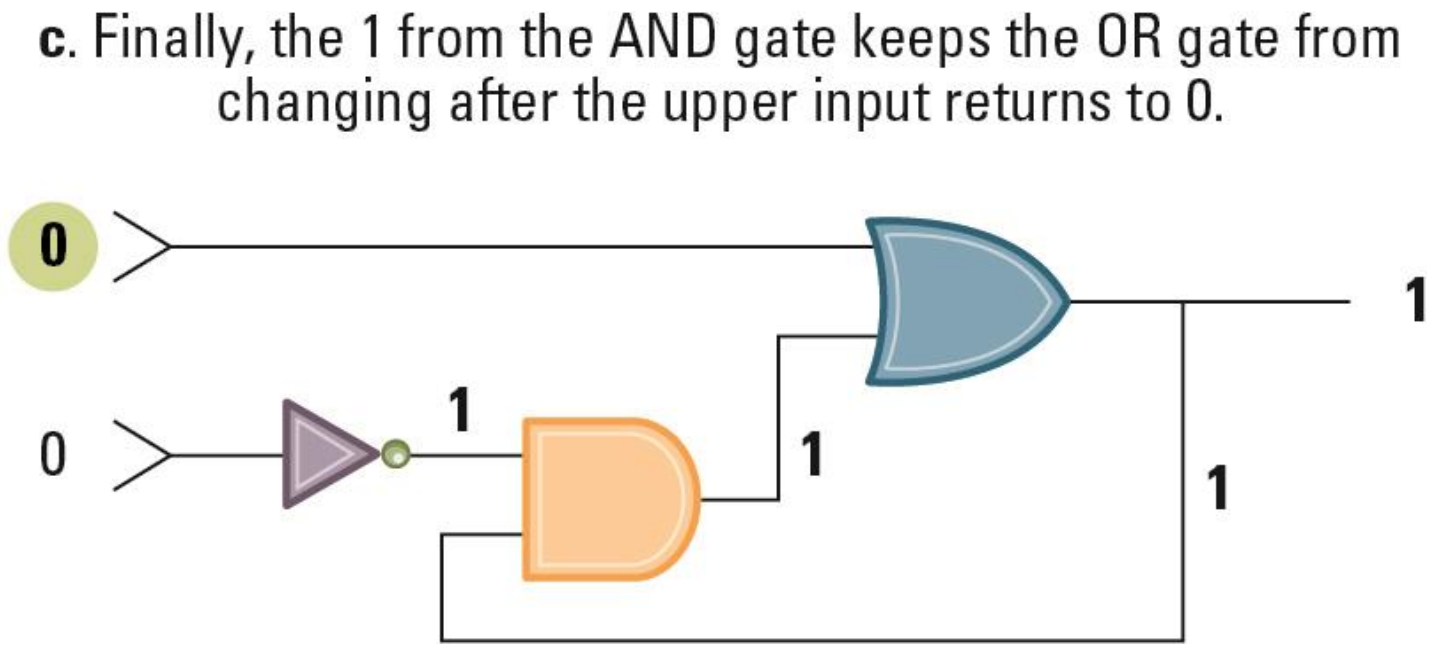
Simple flip-flop operation

- Mark in the diagram what the values will be
- We notice, that the flip-flop is stable in this state



Simple flip-flop operation

- Notice then, that changing the value of the upper input causes no changes
- The state of the flip-flop can be changed only by changing the lower input to 1
  - When the lower input is 1, the flip-flop has no memory (output is decided by upper input)

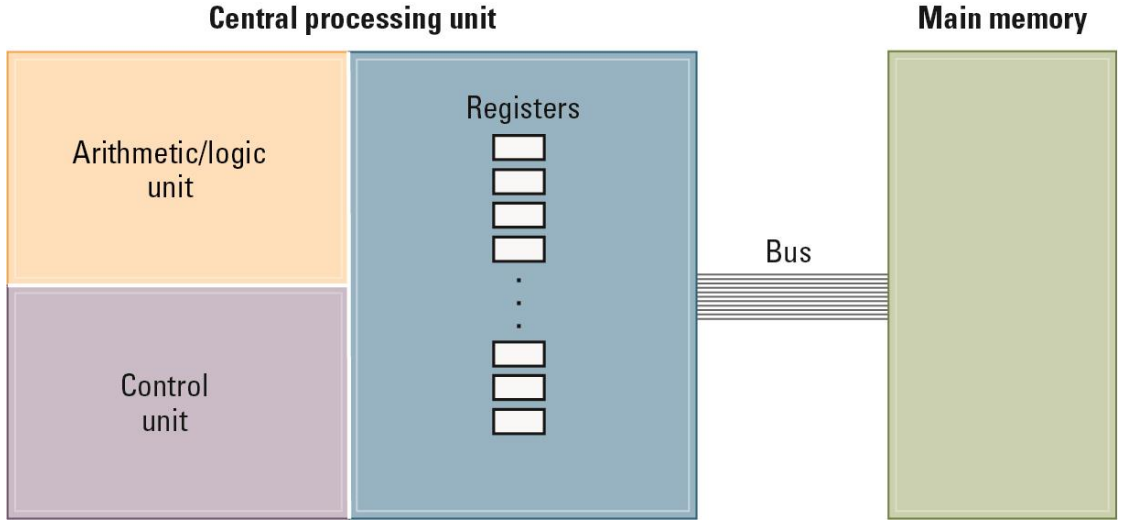


Core components of a computer

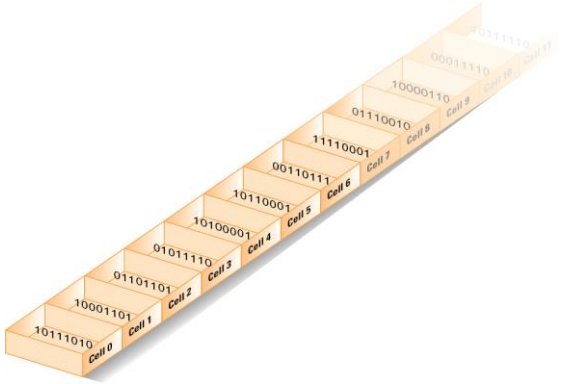
- Computer is basically a logic circuit that consists of **very many** gates
- The structure is very modular, so it consists of components
- At least the following *logic components* (not parts!) are needed:
  - Arithmetic/logic unit (ALU)
  - Register unit
    - General-purpose registers
    - Special-purpose registers (instruction register, program counter)
  - Main memory
  - Buses
  - Clock
  - Input/Output (I/O)

Central Processing Unit (CPU)

- *Central Processing Unit* (known as "processor" or CPU) controls all manipulation of data; it contains three main parts:
  - Arithmetic/logic unit (ALU), which performs calculations and logic comparisons
  - Control unit, which is a "bookkeeper" of how the execution of the program is going – it coordinates the machine's activities
  - Registers, which are built of flip-flops; the data needed by the program is temporarily stored here for rapid access
- CPU is connected to the main memory
- Connects via a bus



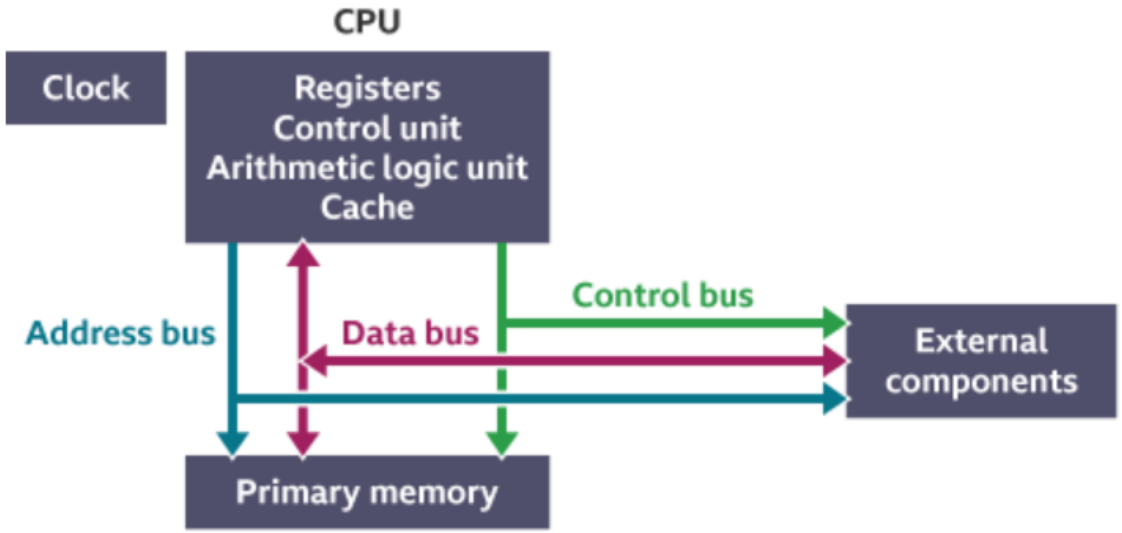
Types of memory



- Generally, there are three types of memory in a basic computer
- Registers, which are inside the CPU
  - Rapid access, can be read and written; data needed immediately
- Main memory
  - Random Access Memory (RAM); data which is needed in near future
  - Quick, but not as quick as registers (because data has to be fetched via a bus)
  - Individual memory cells can be accessed in any order
  - Often Dynamic (DRAM), which is volatile: memory is cleared when the power is turned off
- Mass storage
  - Read-Only Memory (ROM); data which might be needed “at some point”
  - Relatively cheap and non-volatile (information is permanently stored)
  - Quite slow (due to memory type & slower bus connection)

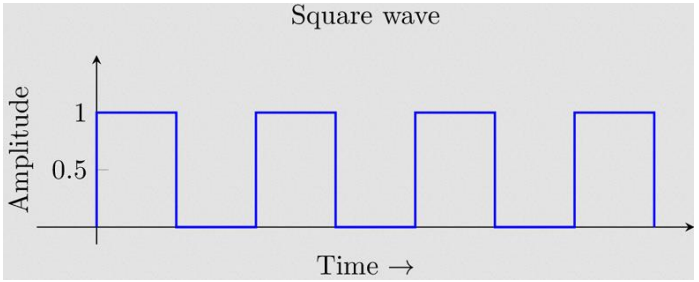
Buses

- Different components in a computer are connected by *buses*, which transfer (or rather copy) information between components
- Bus bandwidth = the total amount of bits that can be transferred in a unit of time; usually this matches the word size of the computer (8bit, 16bit, 32bit, 64bit…)
- Buses can be divided in three groups:
  - Address bus: referral to desired memory address
  - Data bus: the data itself
  - Control bus: keeps the components informed



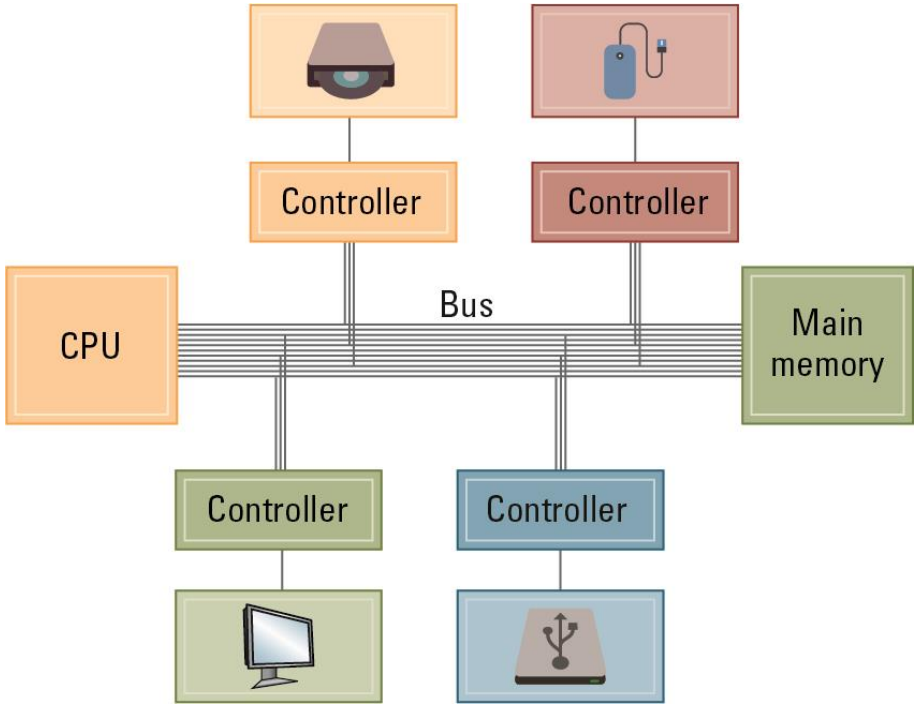
Clock

- Components of the computer need a control signal in order to perform in a synchronized fashion
- Control signal, for example, commands when a flip-flop can save its input and orders when different components have the permission to transfer information to certain buses
- Control signal is created by the control logic and the clock circuit which vibrates
  - The clock generates a square wave (that consists of alternating 1 and 0 signals) on a certain frequency
  - During one signal the computer executes one elementary event
  - Therefore, the performance of the computer is partially dependent on the clock frequency



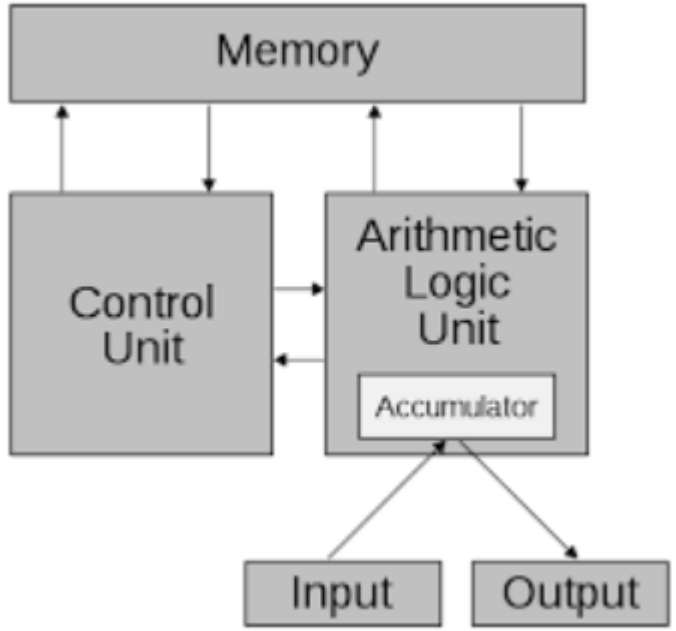
Input/Output (I/O)

- In order to transfer information to and from the computer, different controllers (and also converters and communication protocols) are needed
- These devices are not essential, but often improve the possibilities of the computer - as well as user experience
- Connect to bus
- For example:
  - Monitor
  - Keyboard
  - Mouse
  - Printer
  - External drives
  - Internet connection



Von Neumann architecture

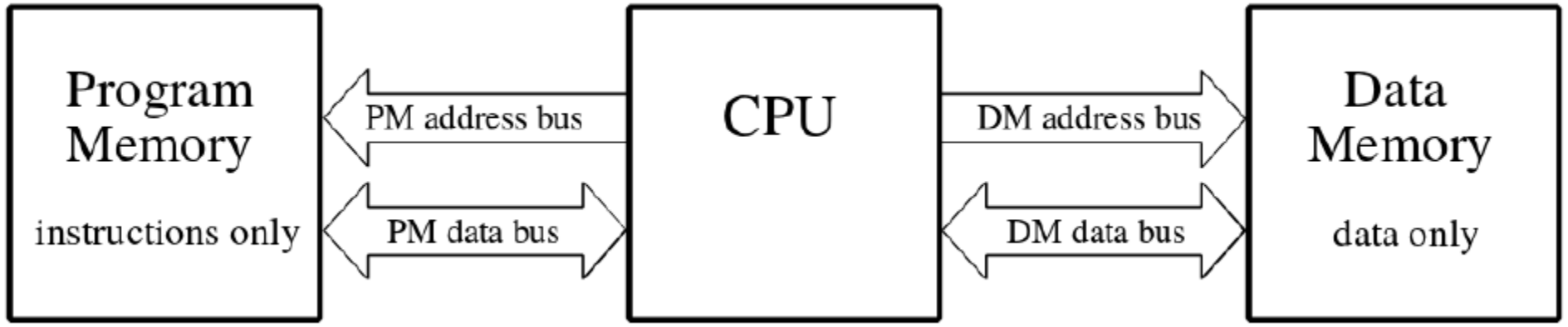
- Early computers were not flexible; they were made to perform one program, and if the program needed to be changed, it required actual re-wiring of the CPU
- John von Neumann* came up with a solution and published it in 1945
  - *Stored-Program Concept*: a program can be encoded and stored in main memory just like data
  - Control unit extracts the program from memory, decodes the instructions and executes
  - Therefore, if we want to change the program, we only need to change the contents of the memory – no need for hardware changes!



*Actually, the main development was made by a research team at University of Pennsylvania, led by J. P. Eckert.

Harvard architecture

- One major disadvantage in von Neumann architecture is the von Neumann bottleneck: because only one bus can be accessed at a time, the CPU spends quite a lot of time just sitting idle and waiting for data to arrive
- Harvard architecture improved this by separating data memory and program memory – thus allowing the use of 2 buses simultaneously, which results in less CPU idle time and better performance
  - Downside: this architecture is expensive to manufacture

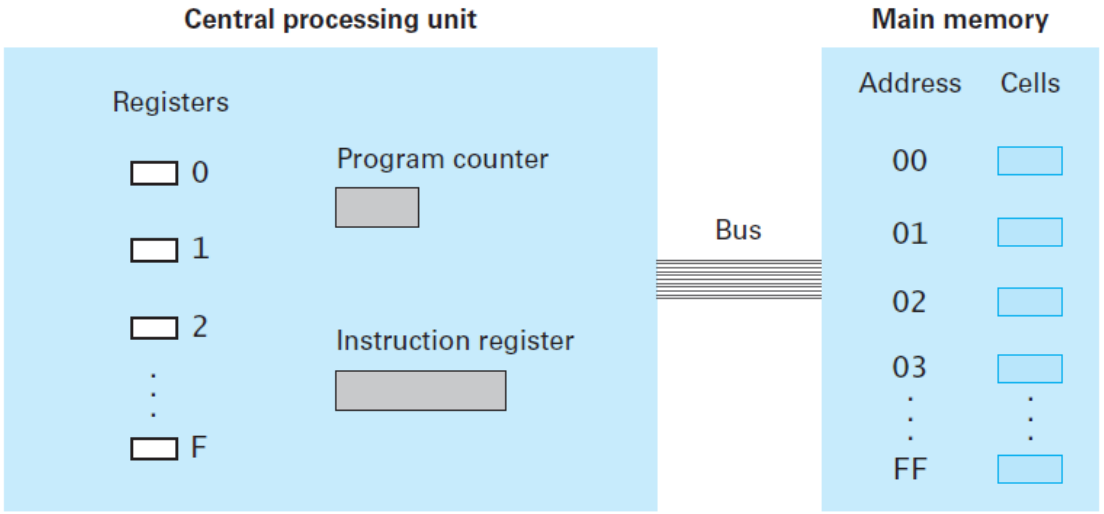


Machine language

- In order to apply the stored-program concept, CPUs are designed to recognize instructions encoded as bit patterns
- This collection of instructions plus its encoding system is called machine language
  - Using this we can write machine-level instructions
- How many instructions should a CPU be able to decode and execute?
  - The less, the better – if we want efficiency in calculation
  - The more, the better – if we want ease of programming
- Two types of CPU philosophies:
  - RISC (Reduced Instruction Set Computer); few simple and efficient instructions
    - Used in mobile devices & consumer electronics, f.ex. ARM processors (Advanced RISC Machine)
  - CISC (Complex Instruction Set Computer); large set of convenient and powerful instructions
    - Used in desktop and laptop computers, f.ex. Intel & AMD processors
- Due to differences, RISC & CISC CPUs can't be easily compared in performance

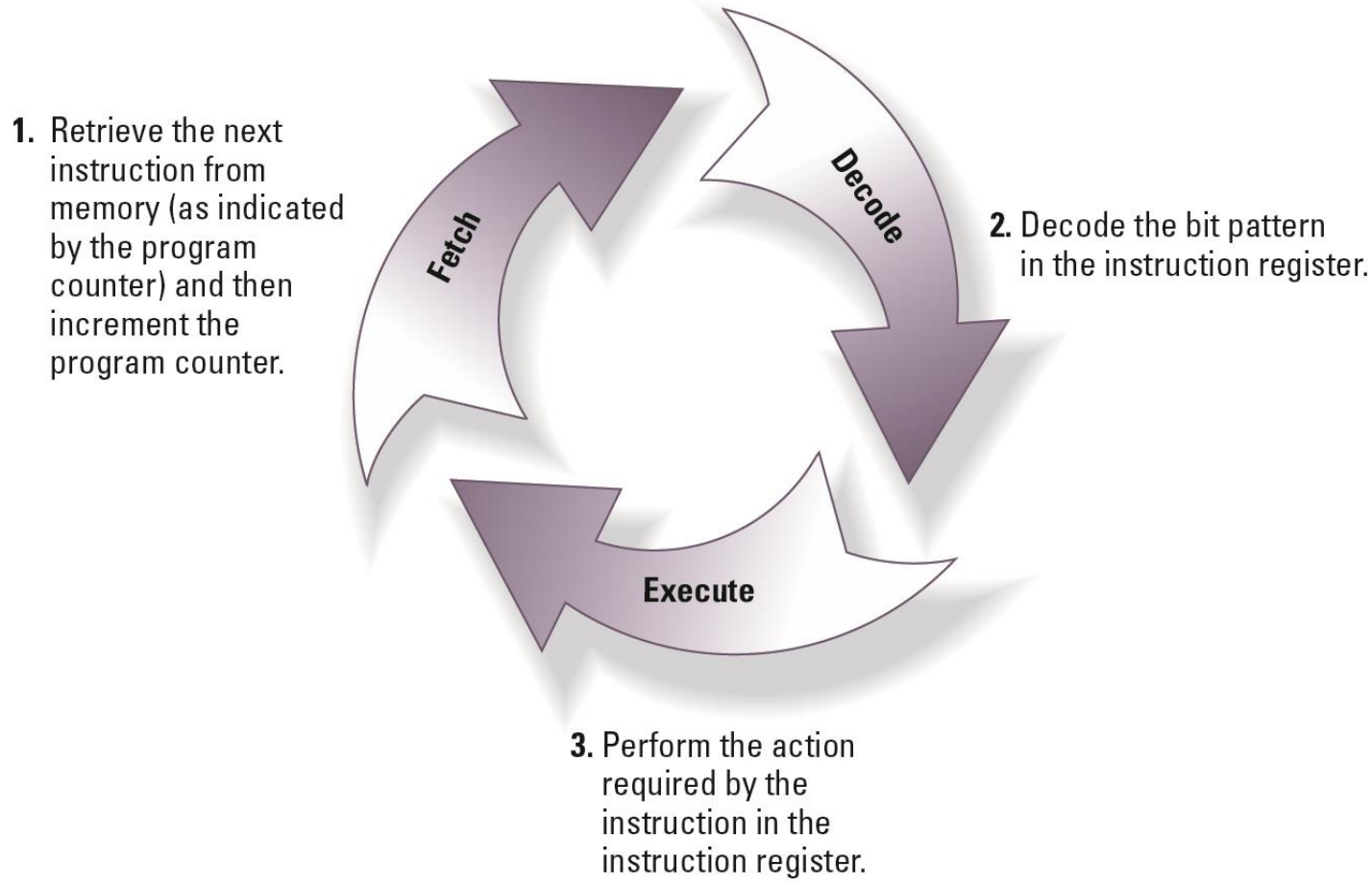
Machine cycle example

- Let's consider how a typical computer encodes instructions
- Our example computer has
  - 16 general-purpose registers (labeled 0…15)
  - 16-bit instruction register
  - 256 main memory cells, each with a capacity of 8 bits (labeled 0…255)
- Binary representation of these would be inconvenient, so we refer to registers and memory cells by hexadecimal notation:
  - Registers labeled 0…F
  - Memory cells labeled 00…FF



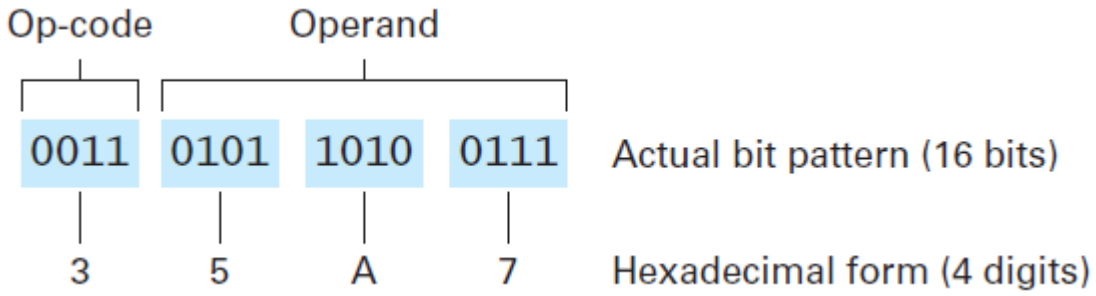
Machine cycle

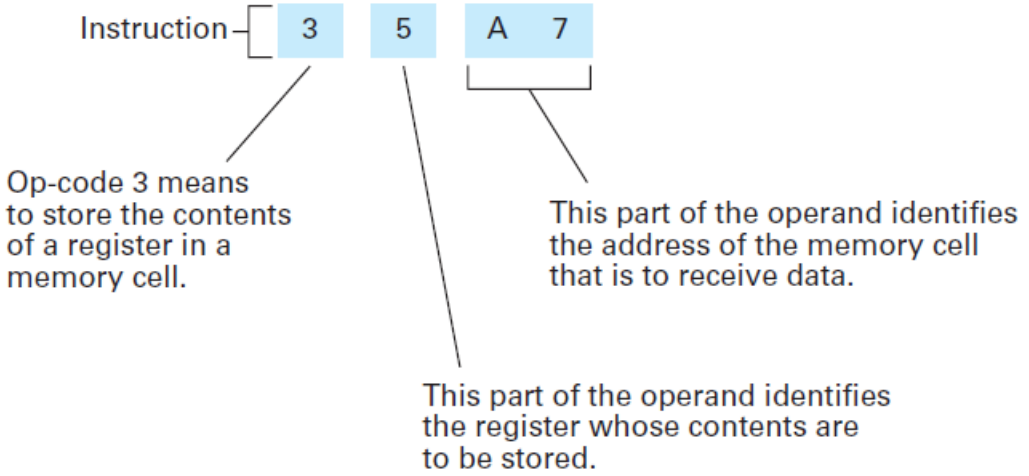
- CPU executes programs by repeating a three-phase process known as the machine cycle
- Phases are
  - Retrieve
  - Decode
  - Execute
- This process is repeated until the program halts



Machine cycle example

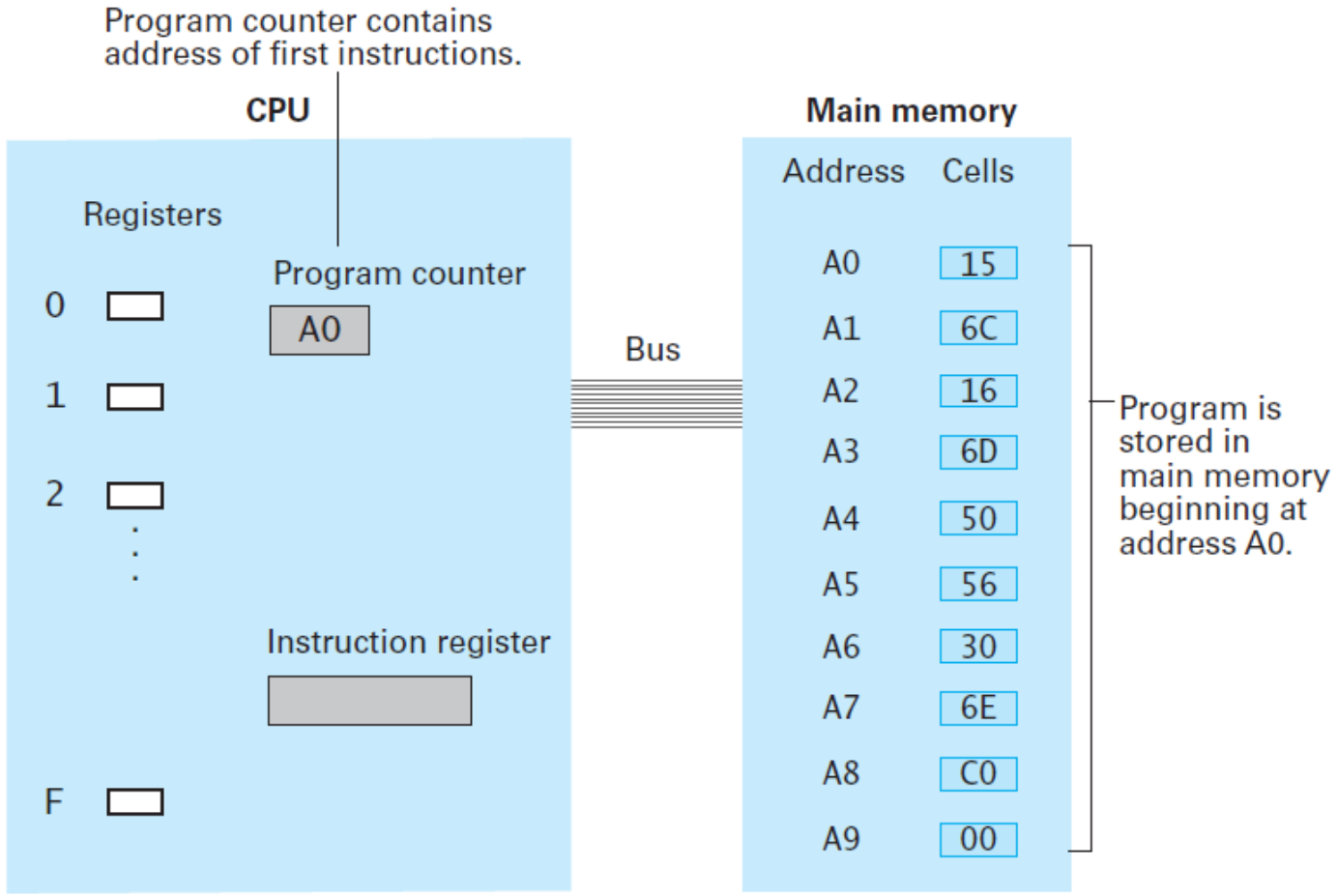
- Length of all instructions is 16 bits, which consists of:
  - Op-code (first 4 bits), which specifies the operation (LOAD, STORE, JUMP,…)
  - Operand (following 12 bits), which clarifies the input of the operation and where do we put the output of the operation





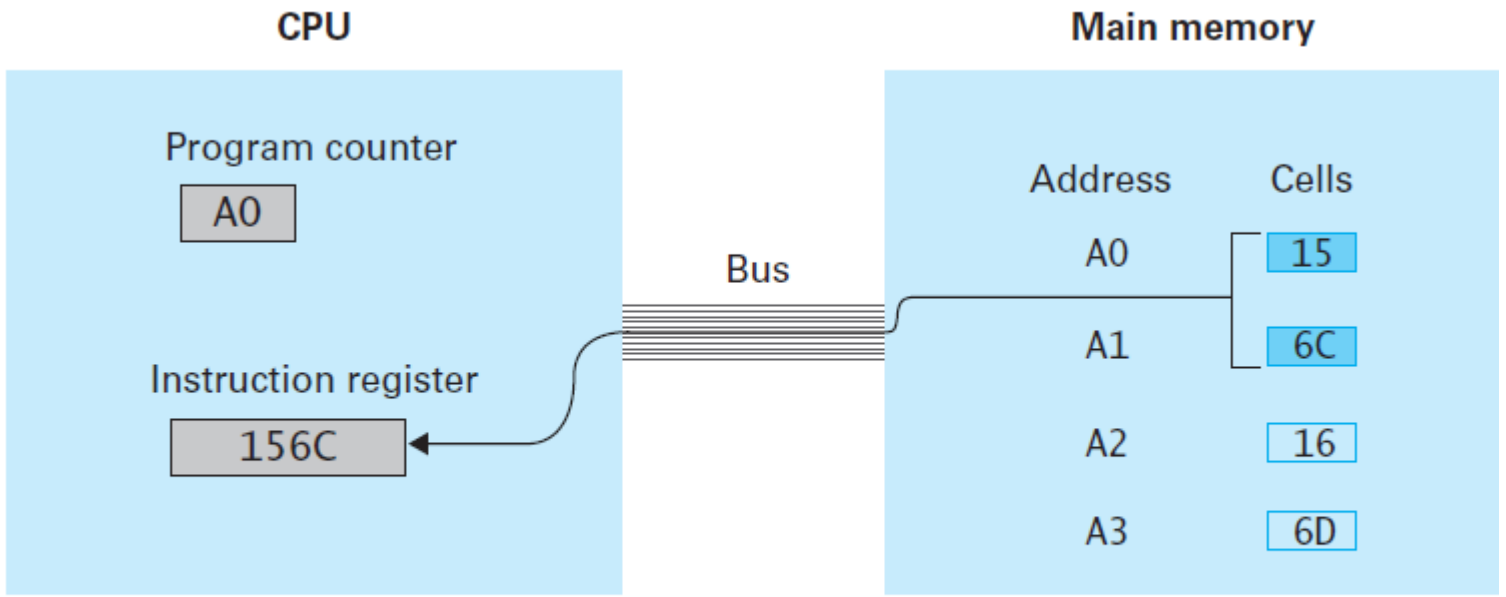
Machine cycle example

- Initial state: program is stored in main memory, starting at A0.
- Program is started by setting the program counter to A0



Machine cycle example

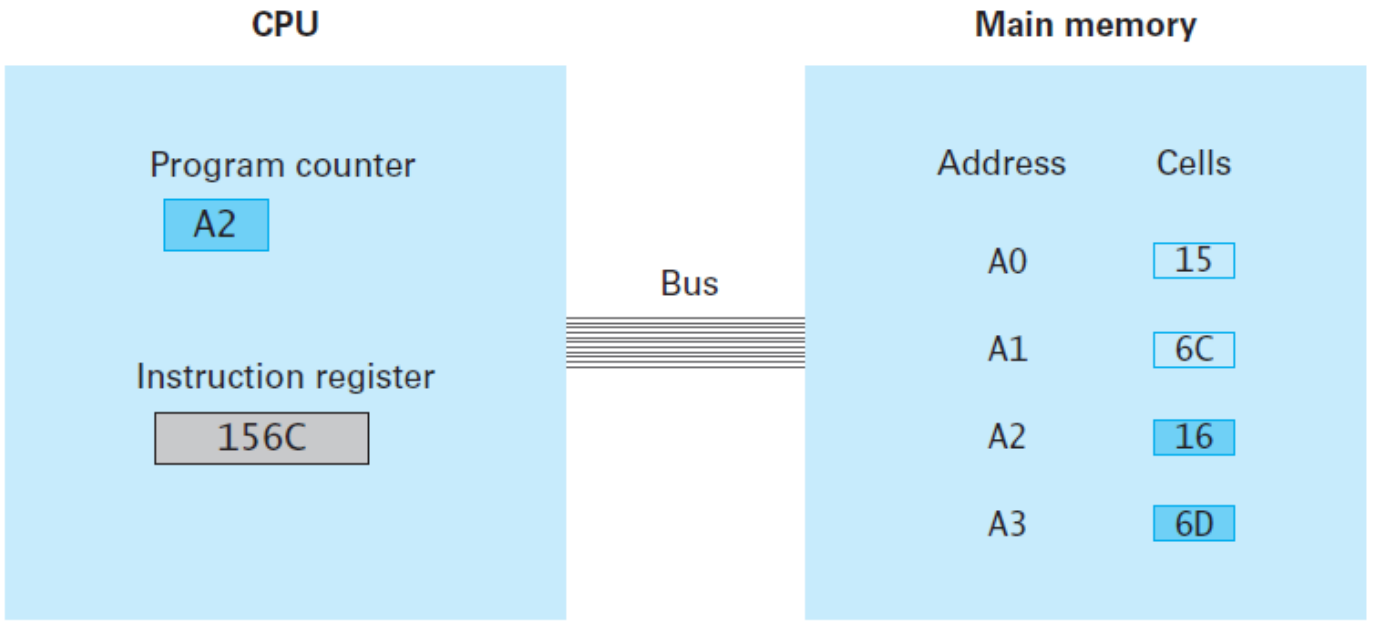
- At the beginning of the fetch step, instructions beginning from A0 are fetched
- Because the memory cells are 8bit and the instruction register is 16bit, the program fetches the contents of two consecutive memory cells (A0 and A1)



a. At the beginning of the fetch step the instruction starting at address A0 is retrieved from memory and placed in the instruction register.

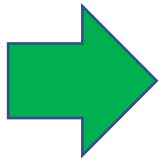
Machine cycle example

- At the end of fetch step, the value of program counter is set to A2
- Decoding the instruction:
  - Op-code 1
  - Operand 56C
- Execute
- Fetch next instruction: 166D etc.



b. Then the program counter is incremented so that it points to the next instruction.

Op-code	Operand	Description
1	RXY	LOAD the register R with the bit pattern found in the memory cell whose address is XY. *Example:* 14A3 would cause the contents of the memory cell located at address A3 to be placed in register 4.



"Load data in memory cell at address 6C to register 5."

Thank you for listening!