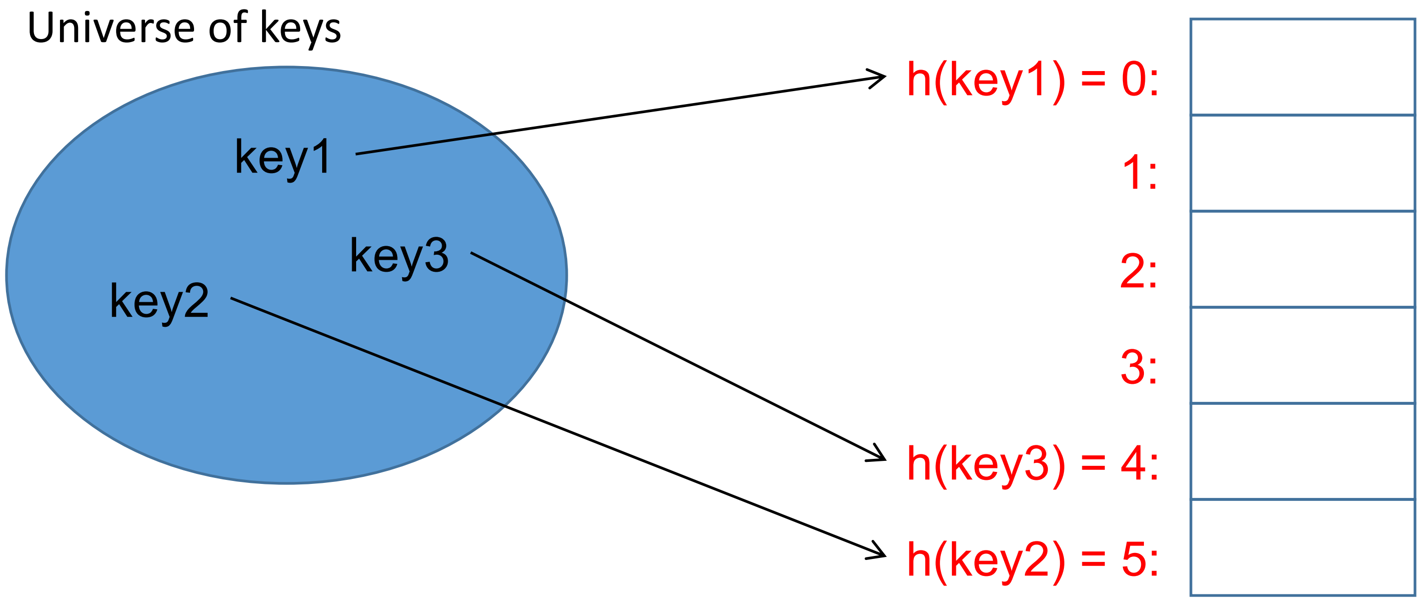
Exam 2 Review

Data Organization

Hashing

- A "hash function" `h(key)` that maps a key to an array index in 0..*k*-1.
- To search the array `table` for that key, look in `table[h(key)]`



A hash function `h` is used to map keys to hash-table (array) slots. Table is an array bounded in size. The size of the universe for keys may be larger than the array size. We call the table slots buckets.

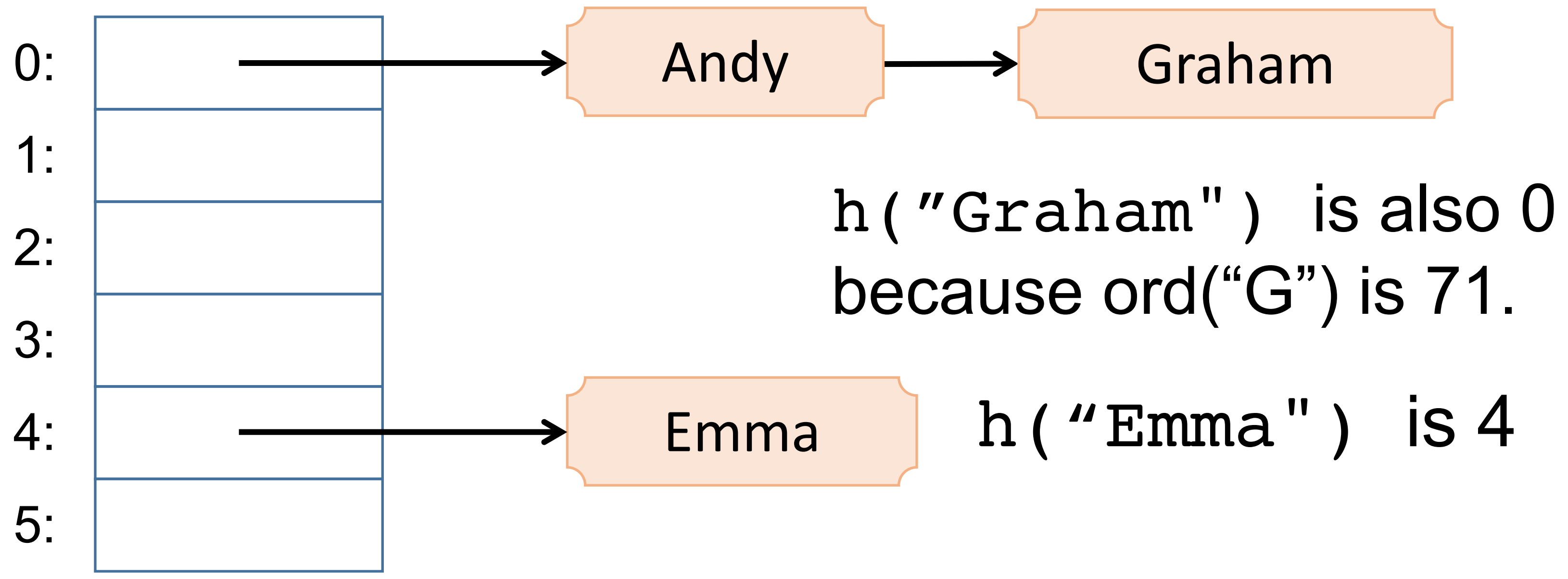
Example: Hash function

- Suppose we have (key,value) pairs where the key is a string such as (name, phone number) pairs and we want to store these key value pairs in an array.
- We could pick the array position where each string is stored based on the first letter of the string using this hash function:

```
def h(str):
  return (ord(str[0]) - 65) % 6
```

Note ord('A') = 65

Add Element “Graham”



In order to add Graham’s information to the table we had to form a link list for bucket 0.

Some Dictionary Operations

- `d[key] = value --` Set `d[key]` to `value`.
- `del d[key] --` Remove `d[key]` from `d.` Raises a an error if `key` is not in the map.
- `key in d --` Return True if `d` has a key `key`, else False.
- `items() --` Return a new view of the dictionary’s items ((key, value) pairs).
- `keys() --` Return a new view of the dictionary’s keys.
- `pop(key[, default])` If `key` is in the dictionary, remove it and return its value, else return `default`. If `default` is not given and `key` is not in the dictionary, an error is raised.

Source: https://docs.python.org/

Data Representation

Compression: Information Content

- We measure information content in bits
  - This is related to the fact that we can represent 2^k different things with *k* bits.
  - Turn the idea around and if we want to represent *M* different things, we need $\log_2 M$ bits
- **But** this is only true if the *M* things all have the same probability

Compression: Information Content

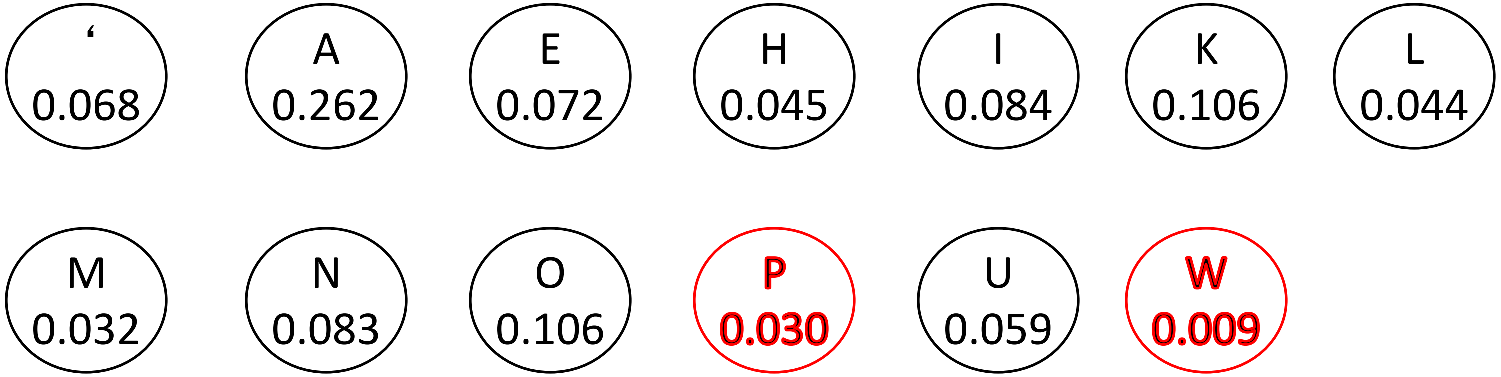
- We measure information content in bits
  - This is related to the fact that we can represent 2^k different things with k bits.
  - Turn the idea around and if we want to represent M different things, we need $\log_2 M$ bits
- **But** this is only true if the M things all have the same probability

Compression: Huffman Coding Process

1. Assign character codes
    a. Obtain character frequencies
    b. Use frequencies to build a *Huffman tree*
    c. Use tree to assign variable-length codes to characters (store them in a table)
2. Use table to encode (compress) ASCII source file to variable-length codes
3. Use tree to decode (decompress) to ASCII

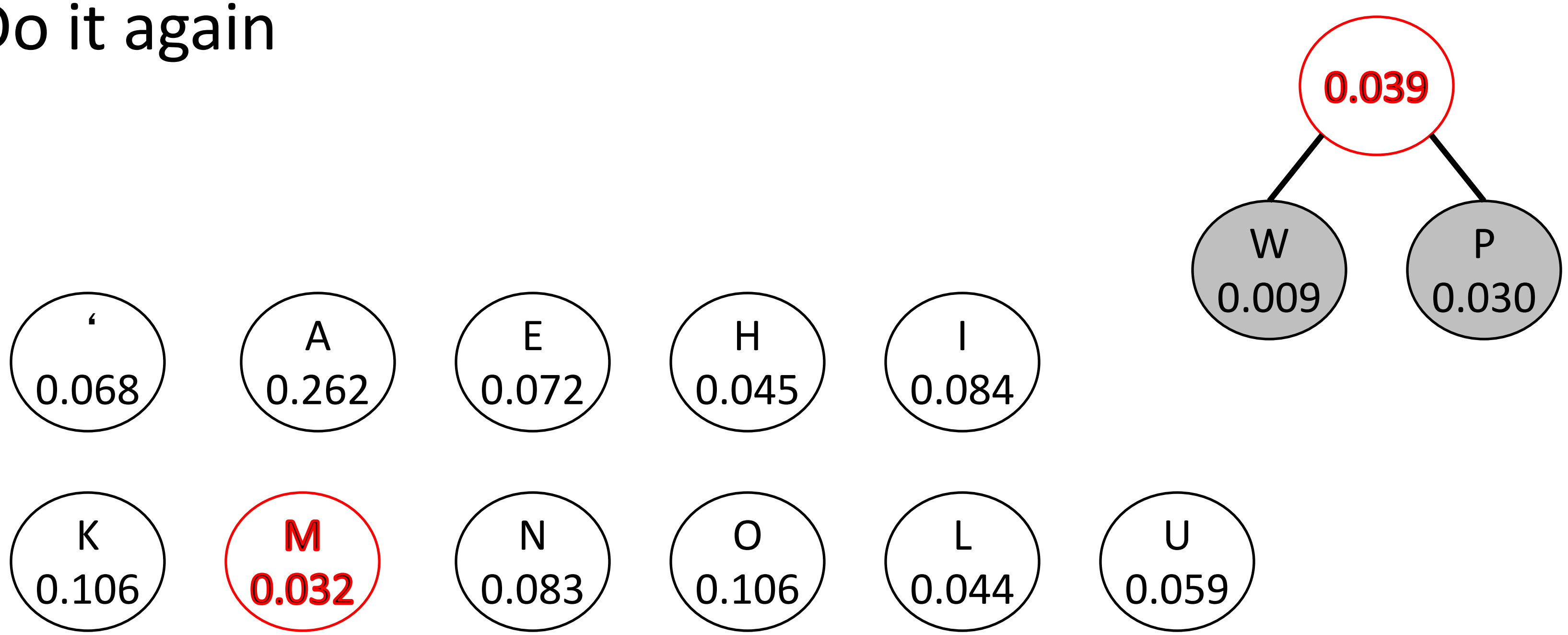
Building The Huffman Tree

- We use a tree structure to develop the unique binary code for each letter.
- Start with each letter/frequency as its own single-node tree
- Find the **two lowest-frequency** trees



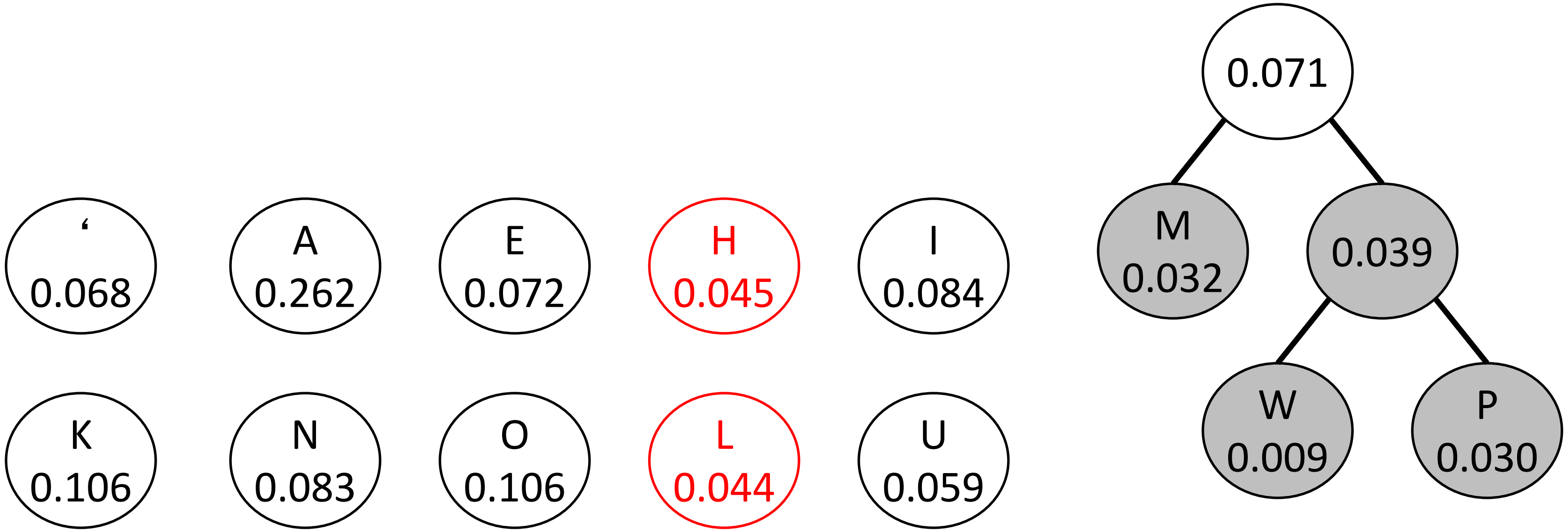
Building The Huffman Tree

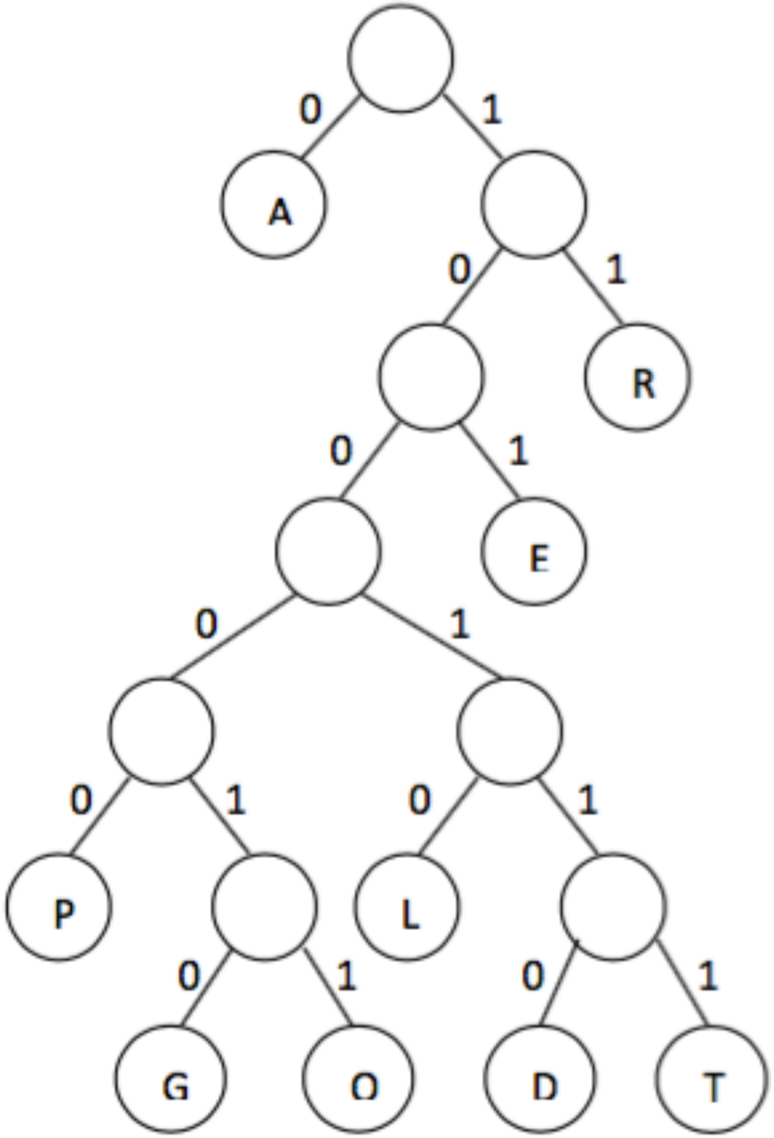
- Combine **two lowest-frequency** trees into a tree with a new root with the sum of their frequencies.
- Do it again



Building The Huffman Tree

- …and again, as many times as possible





10001011101010011 1

Bits to encode each letter?

Bits to re-encode the word above?

Computer Organization

Boolean Logic (Algebra)

- Computer circuitry works based on Boolean Logic (Boolean Algebra) : operations on True (1) and False (0) values.

A	B	$A \wedge B$ (A AND B) (conjunction)	$A \vee B$ (A OR B) (disjunction)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

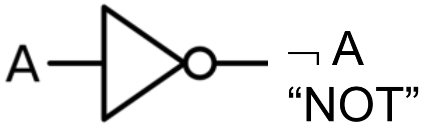
A	$\neg A$ (NOT A) (negation)
0	1
1	0

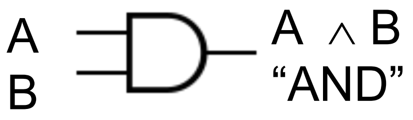
- A and B in the table are Boolean variables, AND and OR are operations (also called functions).

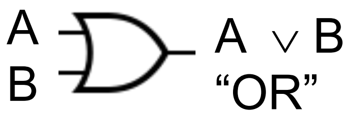
AND, OR, NOT Gates

A	B	$A \wedge B$ (A AND B) (conjunction)	$A \vee B$ (A OR B) (disjunction)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

A	$\neg A$ (NOT A) (negation)
0	1
1	0

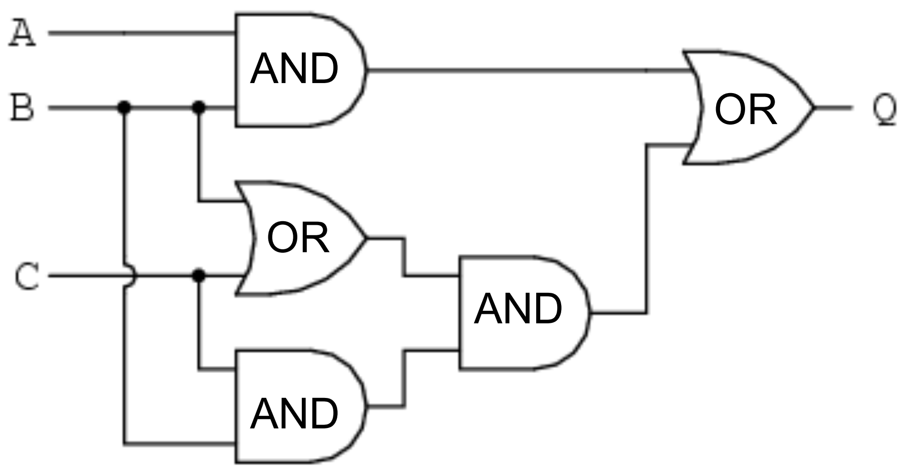






Truth tables define the input - output behavior of logic gates.

Truth Table of a Circuit



Q = (A ∧ B) ∨ ((B ∨ C) ∧ (C ∧ B))

A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	
1	1	0	
1	1	1	

Describes the relationship between inputs and outputs of a device

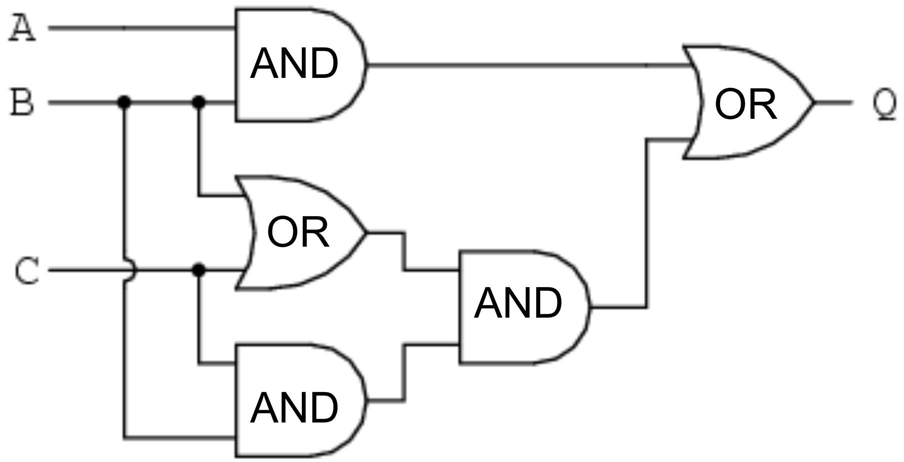
http://www.allaboutcircuits.com/vol_4/chpt_7/6.html

Describing Behavior of Circuits

- Boolean expressions
- Circuit diagrams
- Truth tables

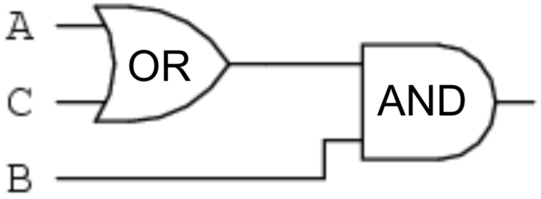
Equivalent notations

Logical Equivalence



$Q = (A \wedge B) \vee ((B \vee C) \wedge (C \wedge B))$

A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



$Q = B \wedge (A \vee C)$

This smaller circuit is logically equivalent to the one above: they have the same truth table. By using laws of Boolean Algebra we convert a circuit to another equivalent circuit.

Laws for the Logical Operators $\wedge$ and $\vee$ (Similar to $\times$ and +)

- Commutative: $A \wedge B = B \wedge A$ $\quad A \vee B = B \vee A$
- Associative: $A \wedge B \wedge C = (A \wedge B) \wedge C = A \wedge (B \wedge C)$
  $A \vee B \vee C = (A \vee B) \vee C = A \vee (B \vee C)$
- Distributive: $A \wedge (B \vee B) = (A \wedge B) \vee (A \wedge C)$
  $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$
- Identity: $A \wedge 1 = A$ $\quad A \vee 0 = A$
- Dominance: $A \wedge 0 = 0$ $\quad A \vee 1 = 1$
- Idempotence: $A \wedge A = A$ $\quad A \vee A = A$
- Complementation: $A \wedge \neg A = 0$ $\quad A \vee \neg A = 1$
- Double Negation: $\neg \neg A = A$

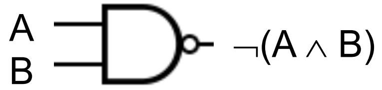
- Start
- Commutativity: $A \wedge B = B \wedge A$
- Distributivity $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$
- Associativity (& Commutativity)
  $(A \wedge B \wedge C = (A \wedge B) \wedge C = A \wedge (B \wedge C)$
- Idempotence $A \wedge A = A$
- Commutativity: $A \wedge B = B \wedge A$
- Idempotence $A \vee A = A$
- Commutativity: $A \wedge B = B \wedge A$
- Distributivity $(A \wedge B) \vee (A \wedge C) = A \wedge (B \vee C)$

- $(x \wedge y) \vee ((y \vee z) \wedge (z \wedge y))$
- $(x \wedge y) \vee ((z \wedge y) \wedge (y \vee z))$
- $(x \wedge y) \vee (z \wedge y \wedge y) \vee (z \wedge y \wedge z)$
- $(x \wedge y) \vee (z \wedge (y \wedge y)) \vee (y \wedge (z \wedge z))$
- $(x \wedge y) \vee ((z \wedge y) \vee (y \wedge z))$
- $(x \wedge y) \vee ((z \wedge y) \vee (z \wedge y))$
- $(x \wedge y) \vee (z \wedge y)$
- $(y \wedge x) \vee (y \wedge z)$
- $y \wedge (x \vee z)$

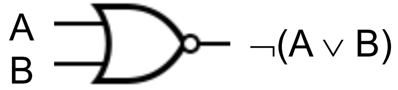
More gates (NAND, NOR, XOR)

A	B	A nand B	A nor B	A xor B
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

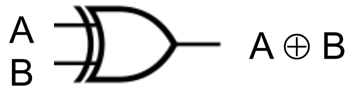
- nand (“not and”): A nand B = not (A and B)



- nor (“not or”): A nor B = not (A or B)



- xor (“exclusive or”):
  A xor B = (A and not B) or (B and not A)



DeMorgan's Law

Nand: $\neg(A \wedge B) = \neg A \vee \neg B$

```
if not (x > 15 and x < 110):  ...
```

is logically equivalent to

```
if (not x > 15) or (not x < 110): ...
```

Nor: $\neg(A \vee B) = \neg A \wedge \neg B$

```
if not (x < 15 or x > 110): ...
```

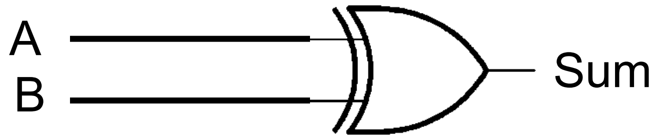
is logically equivalent to

```
if (not x < 15) and (not x > 110): ...
```

Adding Binary Numbers

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0

Adding two 1-bit numbers without taking the carry into account

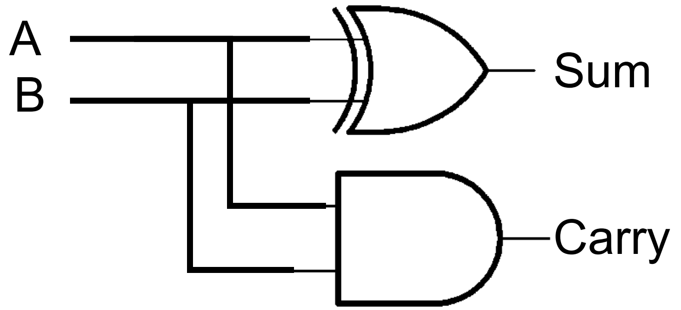


Sum = $A \oplus B$

How can we handle the carry?

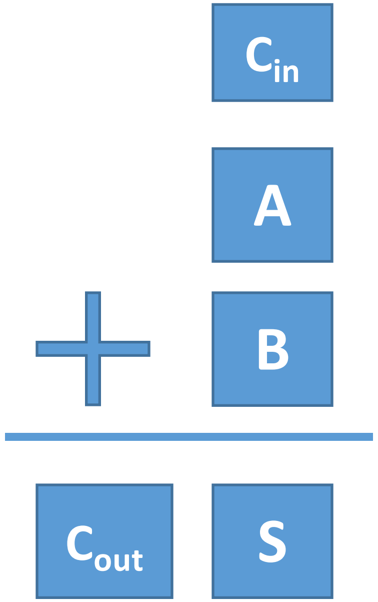
Adding Binary Numbers

A:	0	0	1	1
B:	0	1	0	1
	---	---	---	---
	0	1	1	1 0



Half Adder: adds two single digits

A Full Adder



A	B	C_{in}	C_{out}	S
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

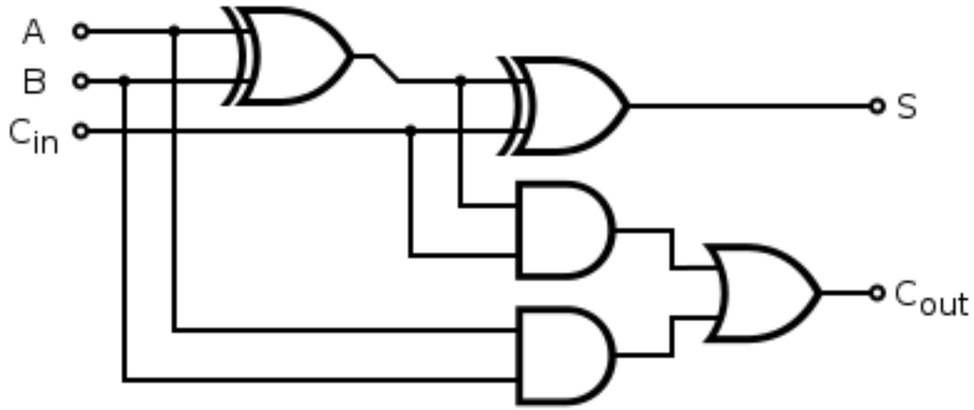
S: 1 when there is an odd number of bits that are 1

C_{out} : 1 if both A and B are 1 or, one of the bits and the carry in are 1.

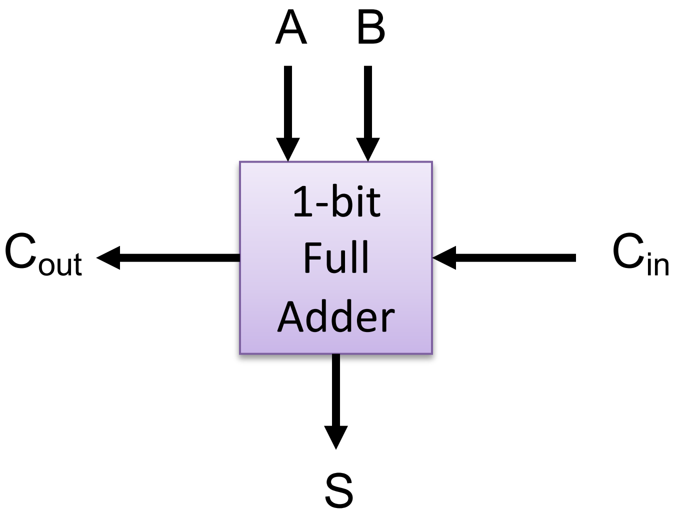
$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = ((A \oplus B) \wedge C_{in}) \vee (A \wedge B)$$

Full Adder (FA)

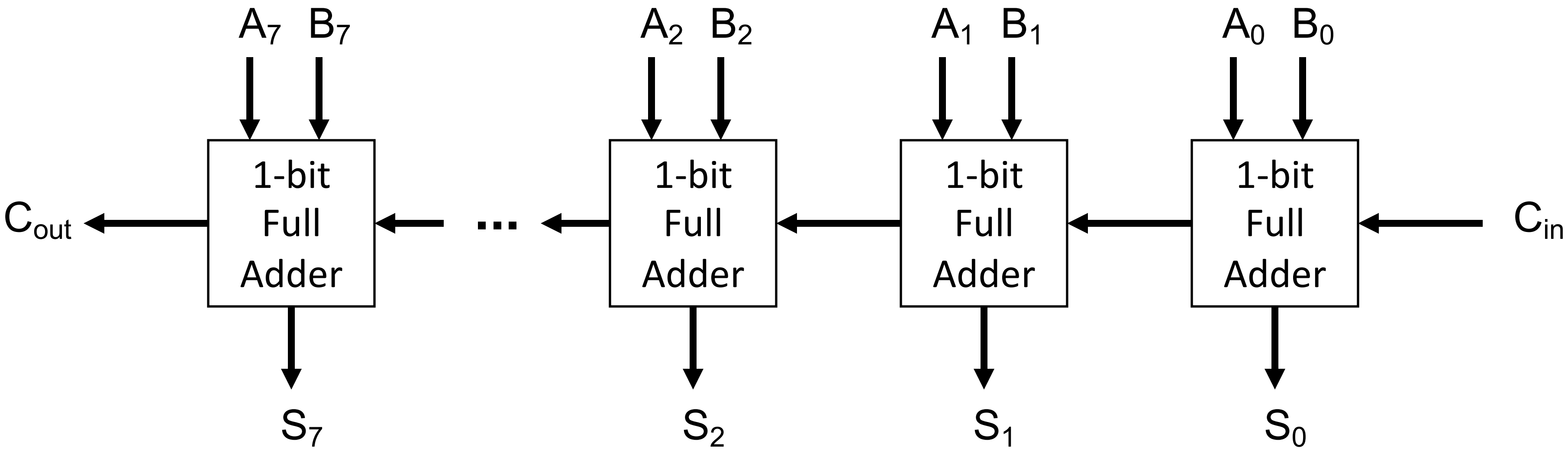


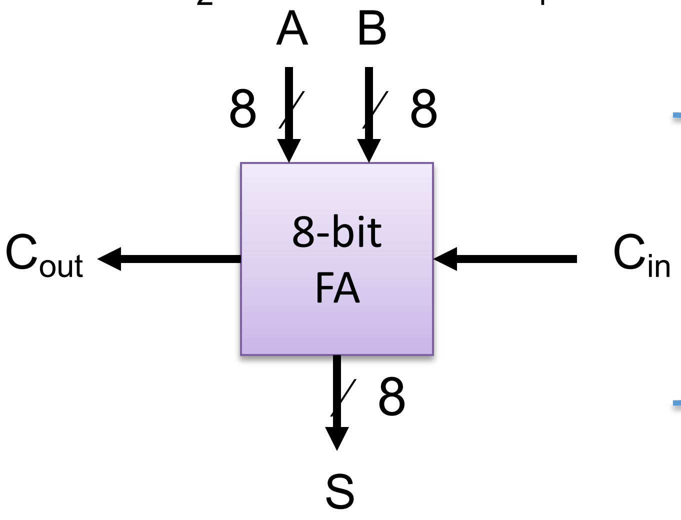
$S = A \oplus B \oplus C_{in}$
$C_{out} = ((A \oplus B) \wedge C_{in}) \vee (A \wedge B)$



More abstract representation of the above circuit. Hides details of the circuit above.

8-bit Full Adder



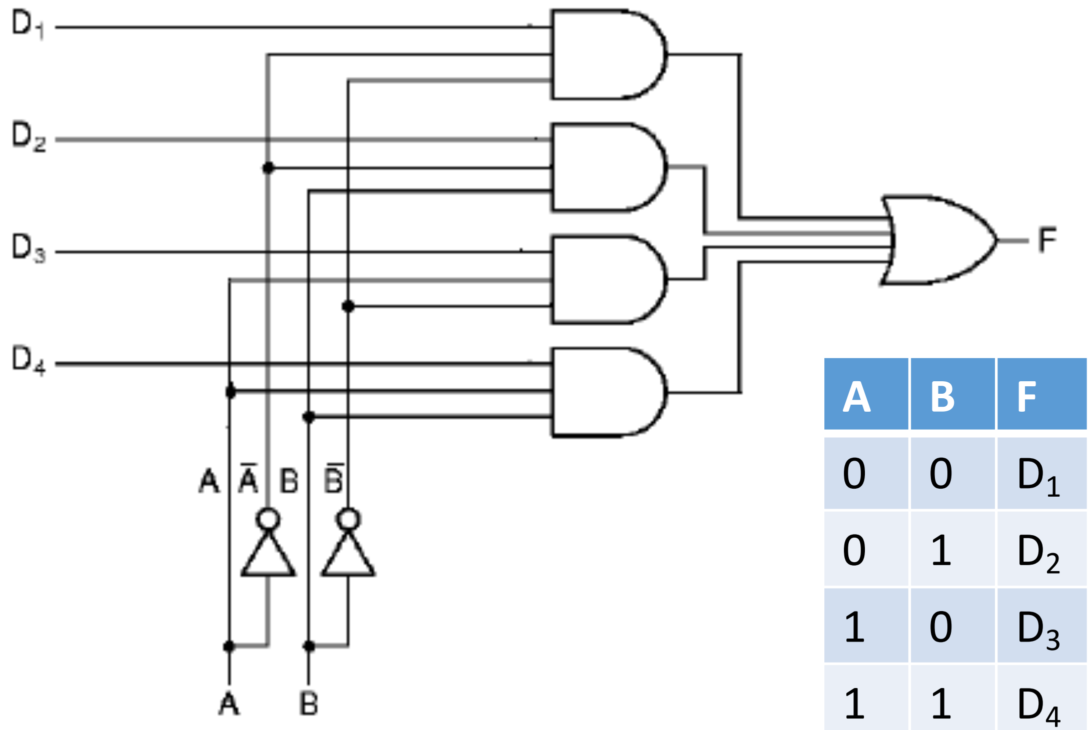


More abstract representation
of the above circuit.
Hides details of the circuit above.

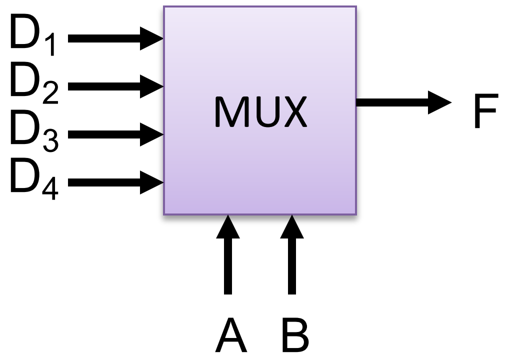
Multiplexer (MUX)

- A multiplexer chooses one of its inputs.

  2^n input lines, n selector lines, and 1 output line



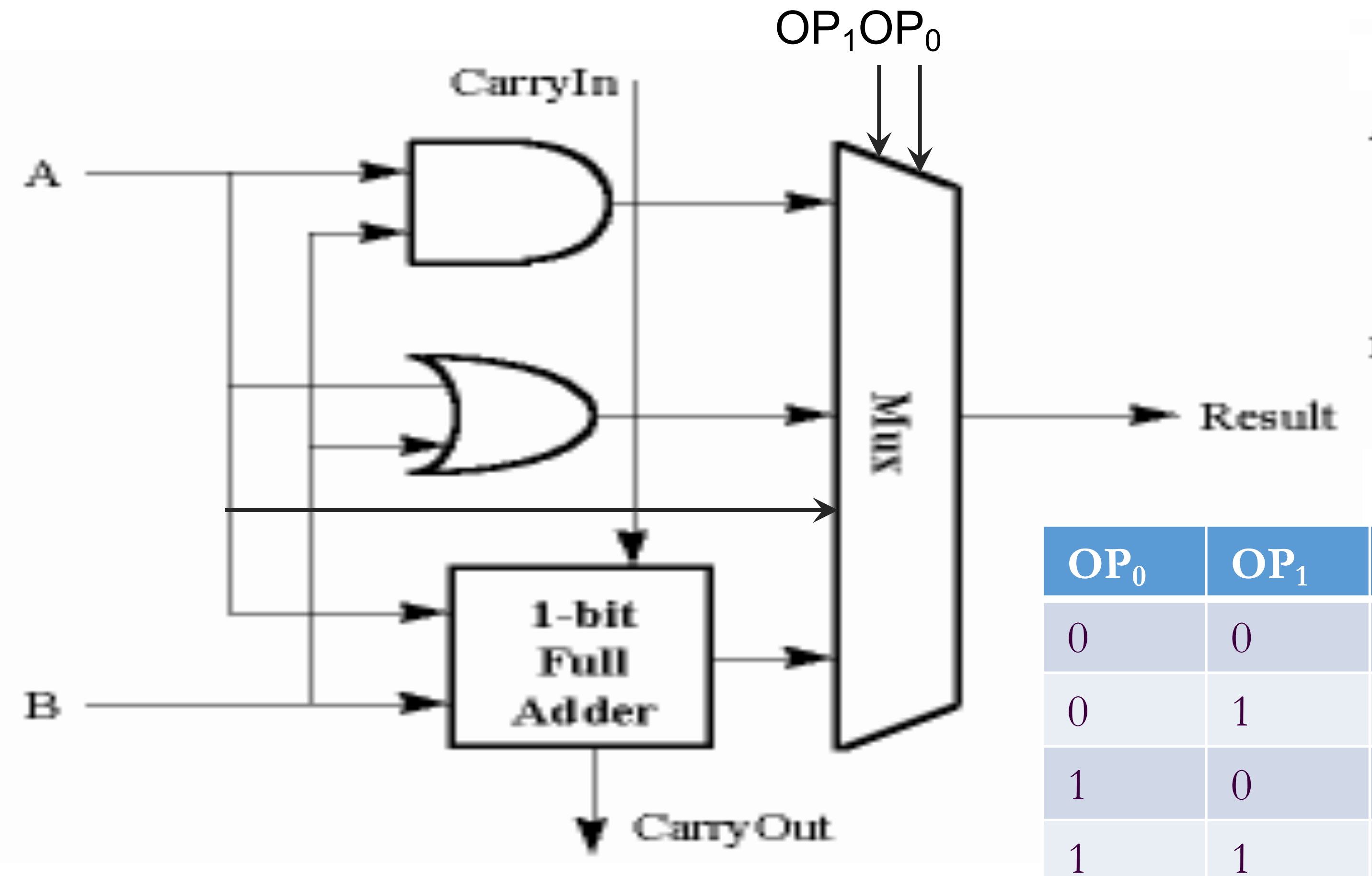
A	B	F
0	0	D_1
0	1	D_2
1	0	D_3
1	1	D_4

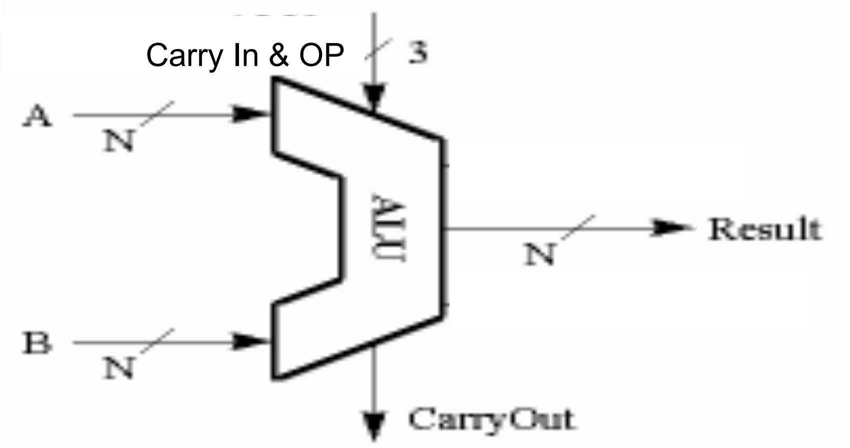


hides details of the circuit on the left

http://www.cise.ufl.edu/~mssz/CompOrg/CDAintro.html

Arithmetic Logic Unit (ALU)





OP_0	OP_1	F
0	0	$A \wedge B$
0	1	$A \vee B$
1	0	A
1	1	$A + B$

http://cs-alb-pc3.massey.ac.nz/notes/59304/l4.html

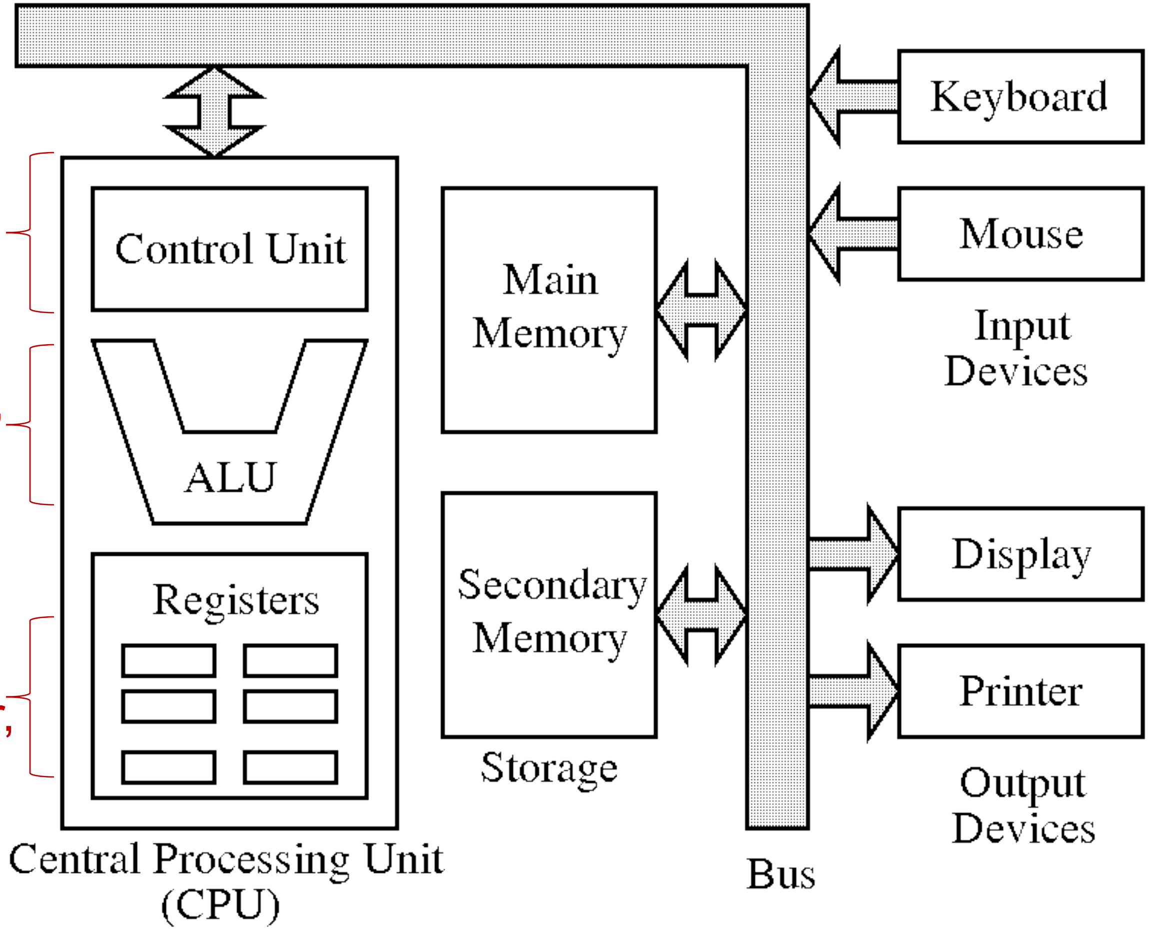
Depending on the OP code Mux chooses
the result of one of the functions (and, or, identity, addition)

Stored Program Computer

instruction fetch, decode, execute

adder, multiplier, multiplexor, Etc.

program counter, instruction register, Etc.



Two specialized registers: the instruction register holds the current instruction to be executed and the program counter contains the address of the next instruction to be executed.

Fetch-Decode-Execute Cycle

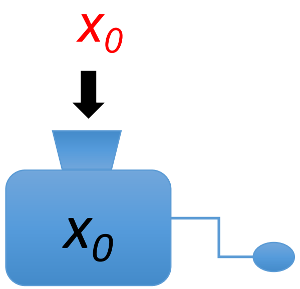
- Modern computers include **control logic** that implements the **fetch-decode-execute** cycle introduced by John von Neumann:
  - Fetch next instruction from memory into the instruction register.
  - Decode instruction to a control signal and get any data it needs (possibly from memory).
  - Execute instruction with data in ALU and store results (possibly into memory).
  - Repeat.

Note that all of these steps are implemented with circuits of the kind we have seen in this unit.

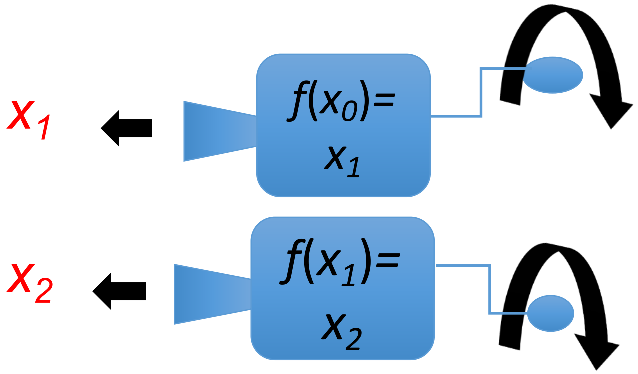
Randomness in Computation

(Pseudo) Random Number Generator

- A (software) machine to produce sequence x_1, x_2, x_3, x_4, x_5,… from x_0
- Initialize / **seed:**



- Get pseudorandom numbers (f is a function that computes a number):



- Idea: **internal state determines the next number**

Simple PRNGs

- *Linear congruential generator formula:*
  $x_{i+1} = (a\ x_i + c)\ \%\ m$
- a, c, and m are **constants**
- Good enough for many purposes
- **…if** a, c, and m are properly chosen

Picking the constants a, c, m

(1) c and m relatively prime

(2) a-1 divisible by all prime factors of m

(3) if m a multiple of 4, so is a-1

- Example: prng1 (a = 1, c = 7, m = 12)
  - Factors of 7: <u>1</u>, 7   Factors of 12: <u>1</u>, 2, 3, 4, 6, 12
  - 0 is divisible by all prime factors of 12 → true
  - if 12 is a multiple of 4 , then 0 is also a multiple of 4 → true
- prng1 will have a period of 12

Random integers in Python

- To generate random integers in Python, we can use the `randint` function from the `random` module.
- `randint(a,b)` returns an integer n such that

  $a \leq n \leq b$ (note that it's **inclusive**)

```
>>> from random import randint
>>> randint(0,15110)
12838
>>> randint(0,15110)
5920
>>> randint(0,15110)
12723
```

Some functions from the **random** module

```
>>> [ random() for i in range(5) ]
[0.05325137538696989, 0.9139978582604943, 0.614299510564187, 0.32231562902200417,
0.8198417602039083]
>>> [ uniform(1,10) for i in range(5) ]
[4.777545709914872, 1.8966139666534423, 8.334224863883207, 3.006025360903046, 8.968660414003441]
>>> [ randrange(10) for i in range(5) ]
[8, 7, 9, 4, 0]
>>> [ randrange(0, 101, 2) for i in range(5) ]
[76, 14, 44, 24, 54]
>>> colors = ['red', 'blue','green', 'gray', 'black']
>>> [ choice(colors) for i in range(5) ]
['gray', 'green', 'blue', 'red', 'black']
>>> [ choice(colors) for i in range(5) ]
['red', 'blue', 'green', 'blue', 'green’]
```

Monte Carlo methods

Idea: run many experiments with random inputs to approximate an answer to a question.

We might be unable to answer the question any other way, or an *analytical* (logical, mathematical, exact) solution might be too expensive.

Monte Carlo method for the hungry dice player

```
def average_winnings(runs) :
    # runs is the number of experiments to run
    total = 0
    for n in range(runs) :
        total = total + dice_game()
    return total/runs
>>> [round(average_winnings(10),2) for i in range(5)]
[85.8, 94.8, 120.7, 123.3, 90.0]
>>> [round(average_winnings(100),2) for i in range(5)]
[105.97, 102.95, 107.74, 134.4, 114.54]
>>> [round(average_winnings(1000),2) for i in range(5)]
[106.84, 107.11, 105.59, 104.28, 106.41]
>>> [round(average_winnings(10000),2) for i in range(5)]
[104.94, 105.71, 105.81, 105.74, 104.62]
```

The Clueless Student

A clueless student faced a pop quiz: a list of the 24 Presidents of the 19th century and another list of their terms in office, but scrambled. The object was to match the President with the term. If the student guesses a random one-to-one matching, how many matches will be right out of the 24, on average?

The Umbrella Quandary

- Mr. X walks between home and work every day
- He likes to keep an umbrella at each location
- But he always forgets to carry one if it’s not raining
- If the probability of rain is *p,* how many trips can he expect to make before he gets caught in the rain? (Assuming that if it’s not raining when he starts a trip, it doesn’t rain during the trip.)