

# Randomness in Computation

## Random Number Generators



# Announcements

- PS8 Due Tomorrow
- PA9 Due July 28. At 11:59. Note that this is SUNDAY
- PS 9 Due July 30 (?)
  
- Lab 9 Tonight (Graphics)

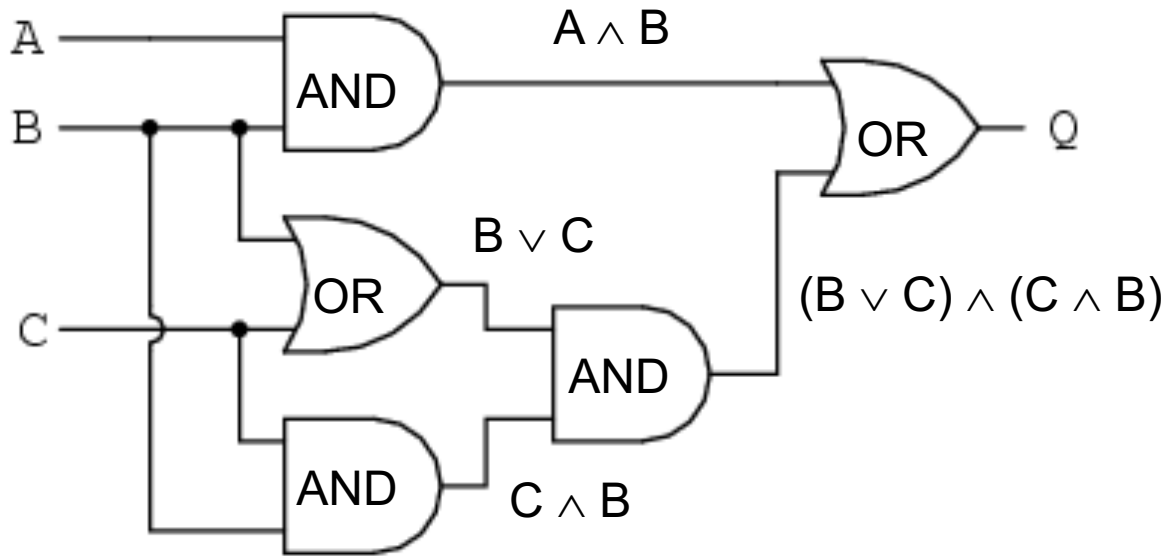
# Today

- Random Number Generation
- Using Pseudo Random Numbers

# Yesterday:

- Computer Organization:
  - Boolean Logic
  - Circuits
  - Organizing and Combining Circuits
- Levels of Abstraction

# Truth Table of a Circuit

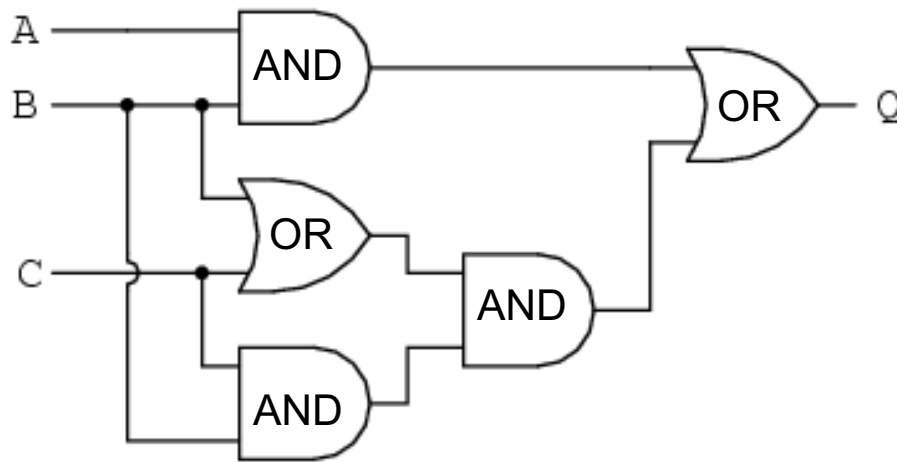


A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

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

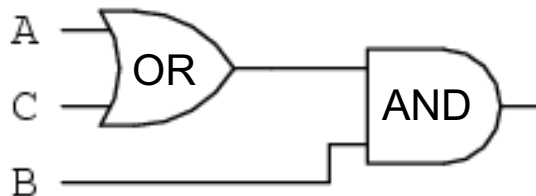
Describes the relationship between inputs and outputs of a device

# 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

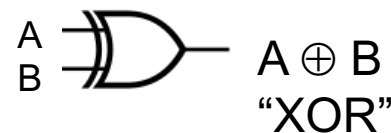
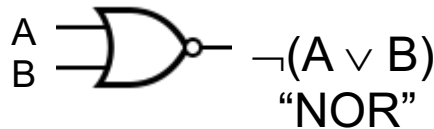
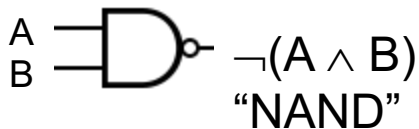
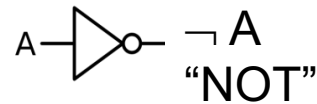
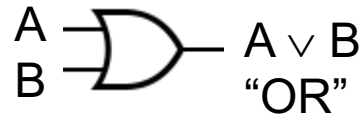
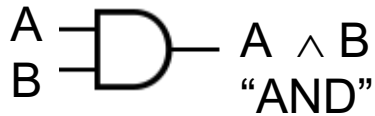


$$Q = B \wedge (A \vee C)$$

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.

# Describing Behavior of Circuits

- Boolean expressions
  - Circuit diagrams
  - Truth tables
- Equivalent notations



# The circuit

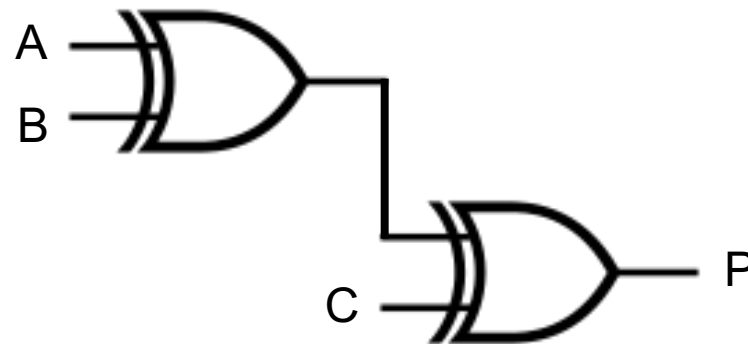
## 3-bit odd parity checker

$$P = (\neg A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge \neg C) \vee (A \wedge B \wedge C)$$

A	B	C	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$$P = (A \oplus B) \oplus C$$

logically  
equivalent





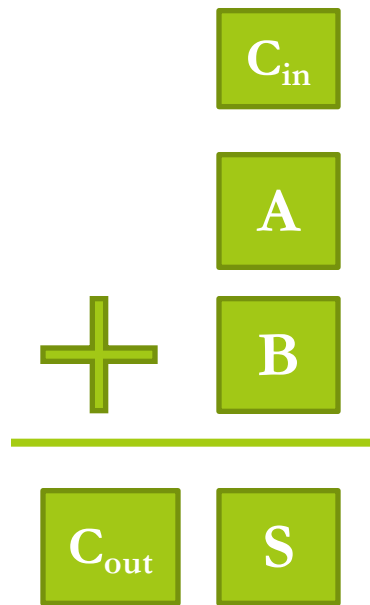
# Using Minterms to Construct a Boolean Function from a Truth Table

(bonus slide)

As presented by Alvarado et. al. in *CS for All*:

1. Write down the truth table for the Boolean function that you are considering
2. Delete all the rows from the truth table where the value of the function is 0
3. For each remaining row create a “minterm” as follows:
  - a. For each variable that has a 1 in that row write the name of the variable. If the input variable is 0 in that row, write the variable with a negation symbol.
  - b. Take their conjunction (AND them together)
4. Combine all of the minterms using OR (take their disjunction)

# A Full Adder



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

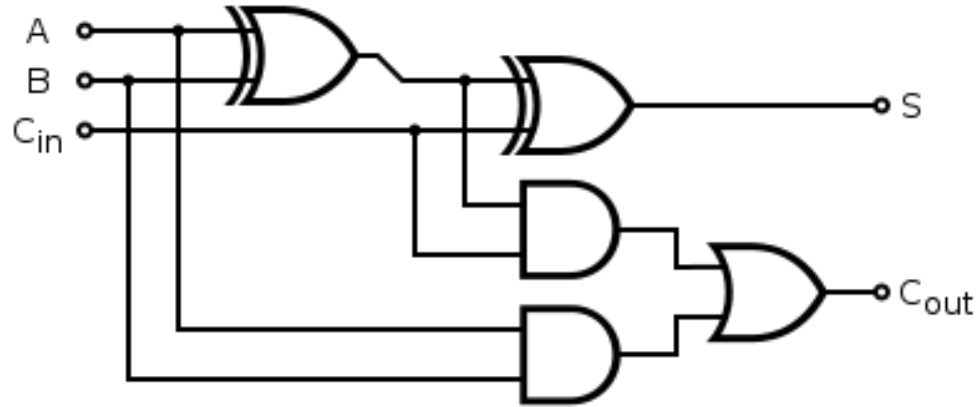
$C_{out}$ : 1 if both  $A$  and  $B$  are 1 or, one of the bits and the carry in are 1.

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

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

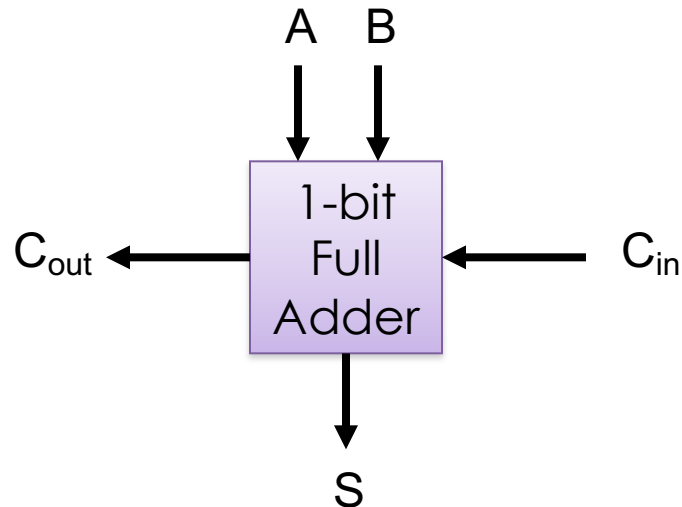
$$C_{out} = ((A \oplus B) \wedge C_{in}) \vee (A \wedge B)$$

# Full Adder (FA)



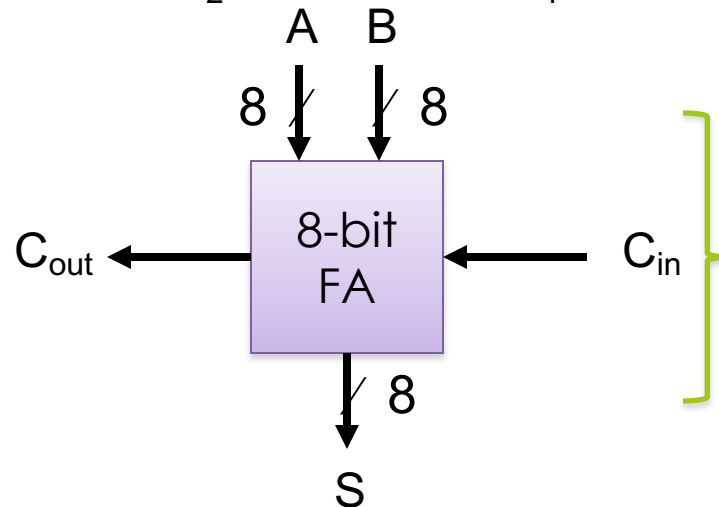
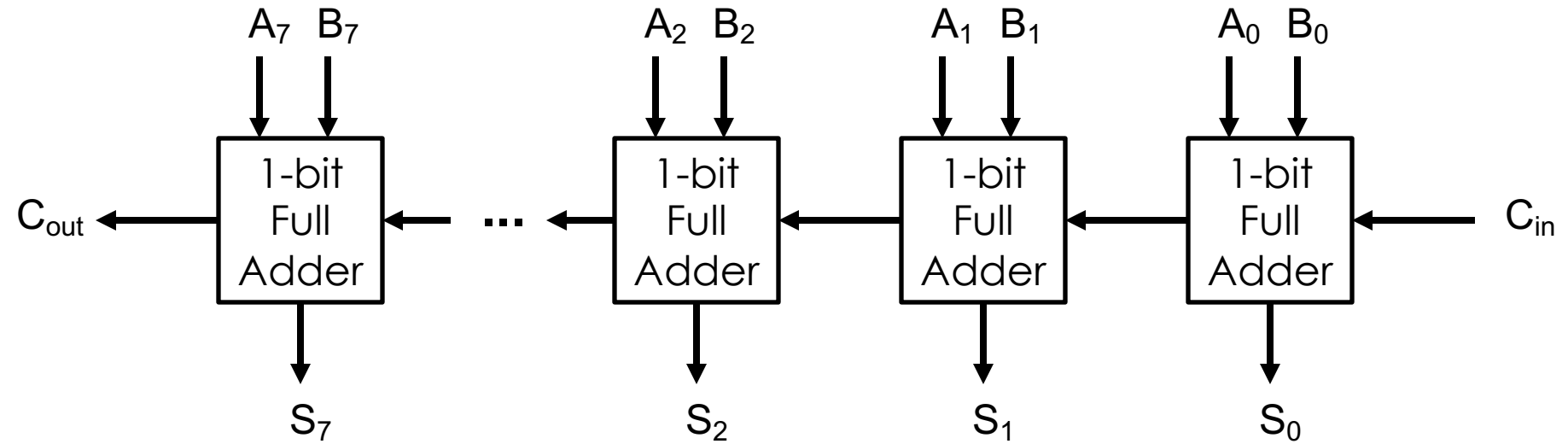
$$S = A \oplus B \oplus C_{in}$$

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



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

# 8-bit Full Adder

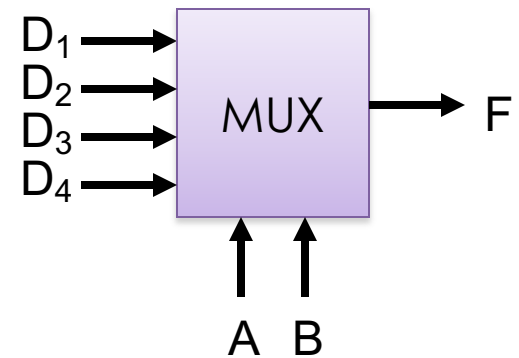
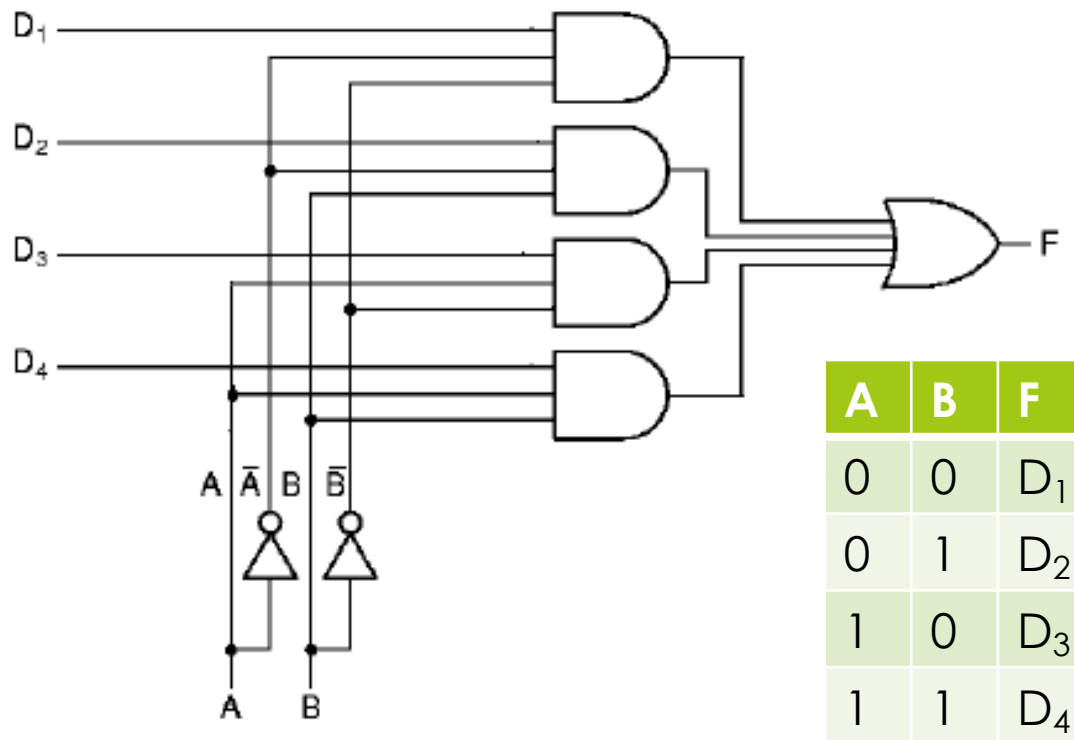


More abstract representation of the above circuit. Hides details of the circuit above.

# Multiplexer (MUX)

- A multiplexer chooses one of its inputs.

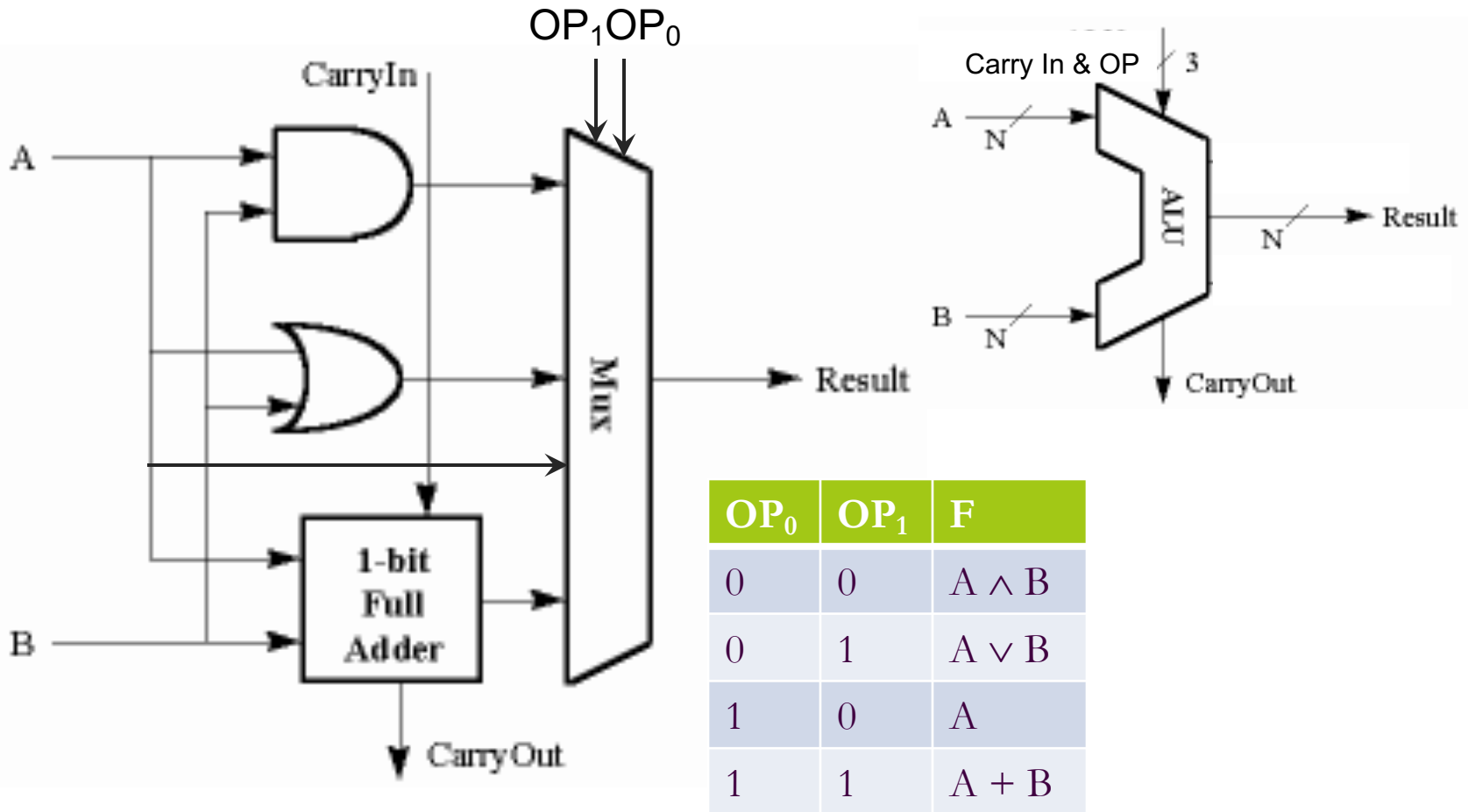
$2^n$  input lines,  $n$  selector lines, and 1 output line



hides details of the circuit on the left

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

# Arithmetic Logic Unit (ALU)



<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)

# Memory Layout

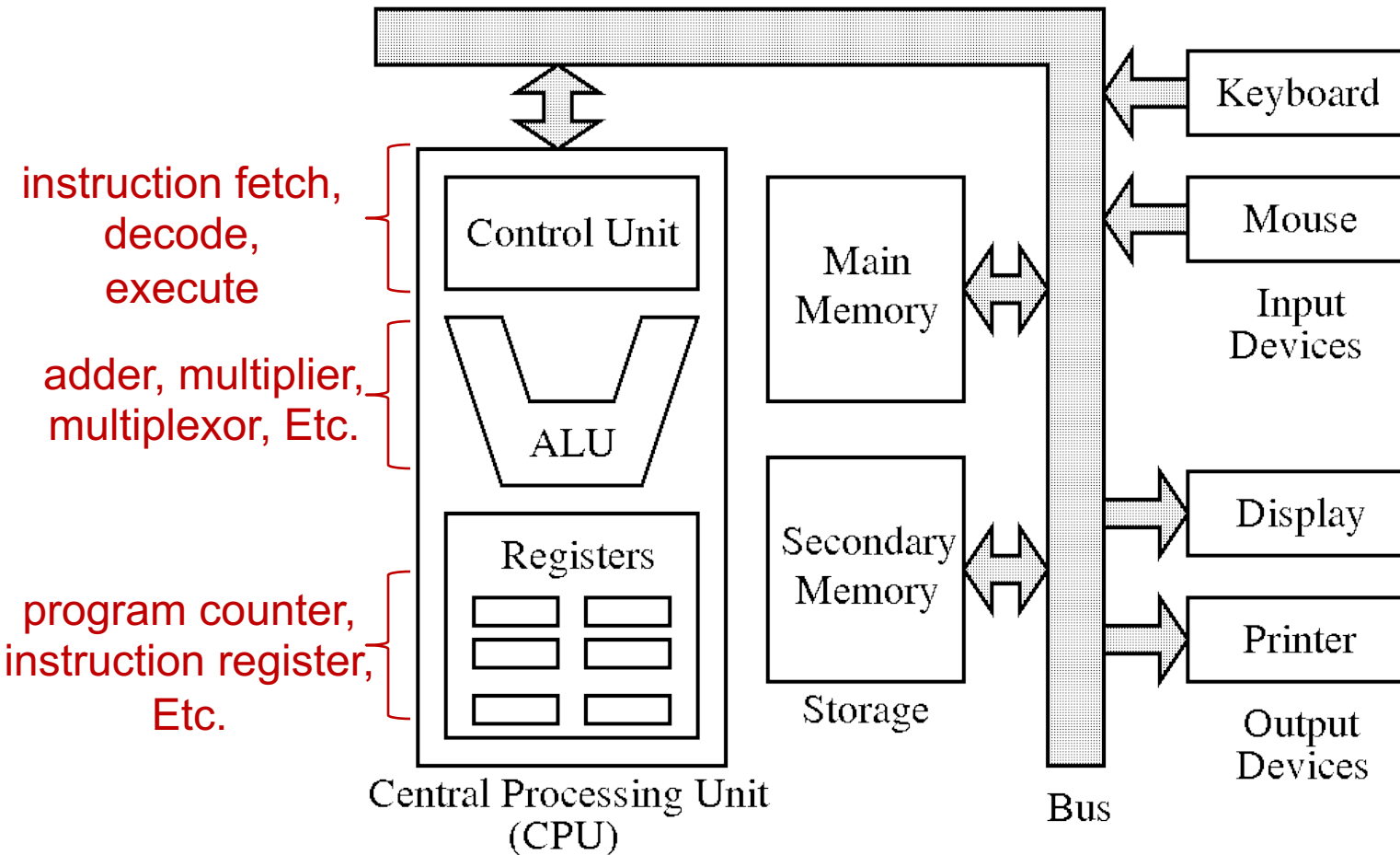
Address	Content
100:	50
104:	42
108:	85
112:	71
116:	99

We saw this picture in Unit 6. It hid the bit representation for readability. Assumes that memory is byte addressable and each integer occupies 4 bytes .

Address	Content
01100100:	... 01100100
01101000:	... 01010100
01101100:	... 01010101
01110000:	... 01000111
01110100:	... 01100011

In this picture and in reality, addresses and memory contents are sequences of bits.

# Stored Program Computer



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.



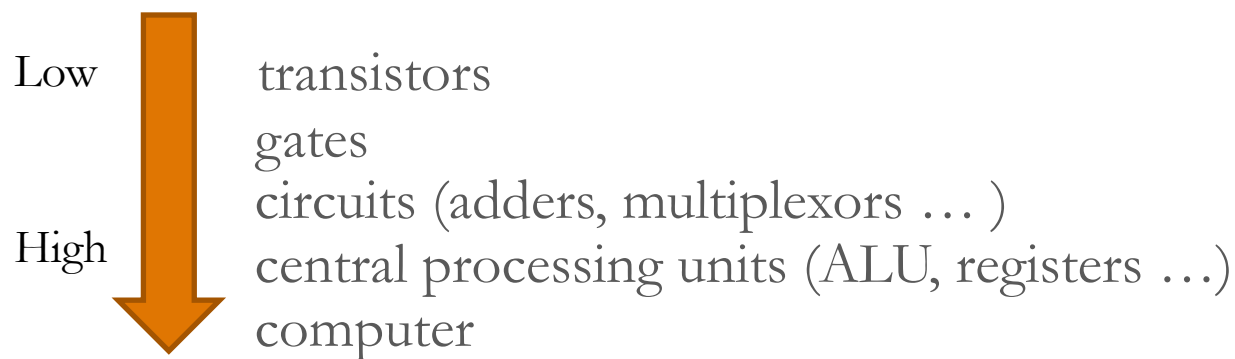
# Fetch-Decode-Execute Cycle

- 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.*

# Using Abstraction in Computer Design

- We can use layers of abstraction to hide details of the computer design.
- We can work in any layer, not needing to know how the lower layers work or how the current layer fits into the larger system.



- A component at a higher abstraction layer uses components from a lower abstraction layer without having to know the details of how it is built. It only needs to know what it does.




# Random Number Generators

# Overview

- The concept of randomness
- Pseudorandom number generators
  - Linear congruential generators
- Using random number generators in Python

# Randomness in Computing

- Determinism: input  predictable output
- Sometimes we want **unpredictable** outcomes
  - Games, cryptography, modeling and simulation, selecting samples from large data sets, randomized algorithms
- We use the word “randomness” for *unpredictability, having no pattern*

# What is Randomness?

Tricky philosophical and mathematical question

Consider a sequence of integers. When is it  
“random”?

# Some Randomness Properties

- A random sequence should **not be biased**: as length increases no element should be any more frequent than others

an unfair coin is **biased** (*Long sequences will have more heads*):

**H T T H H T H H T T H T H H H T H ...**

Long sequences will have more heads than tails

- It's not good enough to be unbiased: consider  
010101010101010101010101010101...

**Unbiased** but **predictable**

# Some Randomness Properties

- A random sequence should **be unpredictable**. What does that mean?
- We might try: given some part of the sequence, the next element is **equally likely** to be any possible number.
- But consider the *Champernowne sequence*  
011011100101110111100010011010...  
(all binary integers concatenated in order)
  - 0 and 1 are equally likely: half of each in long enough sequence)!
  - but we always know the next digit!



# Some Randomness Properties

- A random sequence is *dense* in a precise sense measured by *information entropy*.
- Low-entropy sequences are predictable, e.g. in English we know the next letter is more likely to be *e* than *z*.
- Unpredictable sequences have high entropy.
- But so do some predictable ones!

# Some Randomness Properties

- A random sequence is *incompressible*: there's no short rule describing what comes next
- E.g. digits of  $\pi$   
14159265358979323846264338327950288...look random, but they can be generated by a rule so they are not a random sequence
- A procedure for calculating  $\pi$  is a *compressed* representation of  $\pi$

# Random sequence should be

- **Unbiased** (no “loaded dice”)
- **Information-dense** (high entropy)
  - Unpredictable sequence have high entropy
- **Incompressible** (no short description of what comes next)

**But** there are sequences with these properties that are **predictable anyway!**



# Randomness is slippery

- In summary, “random” means something like this:  
***No gambling strategy is possible that allows a winner in the long run.***
  - Non-randomness: can be detected and proved
  - Randomness: hard-to-impossible to prove
- Often we settle for
  - “this sequence passes some tests for randomness”:
  - high entropy
  - passes chi-square test
  - ...
- Example: see <http://www.fourmilab.ch/random/>

# Randomness in computing

Why do we want it? How do we get it?

# Why Randomness in Computing?

- Internet gambling and state lotteries
- Simulation
  - (weather, evolution, finance [oops!], physical and biological sciences, ...)
- Monte Carlo methods and randomized algorithms
  - (evaluating integrals, ...)
- Cryptography
  - (secure Internet commerce, BitCoin, secret communications, ...)
- Games, graphics, and many more

# True Random Sequences

- Precomputed random sequences.
  - For example, *A Million Random Digits with 100,000 Normal Deviates (1955)*: A 400 page reference book by the RAND corporation
    - 2500 random digits on each page
    - Generated from random electronic pulses
- True Random Number Generators (TRNG)
  - Extract randomness from physical phenomena
    - atmospheric noise, times for radioactive decay
- Drawbacks:
  - Physical process might be biased  
(produce some values more frequently)
  - Expensive
  - Slow

# LavaRand





# Pseudorandom Sequences

- *Pseudorandom number generator (PRNG):*

Algorithm that produces a sequence that looks random  
(i.e. passes some randomness tests)

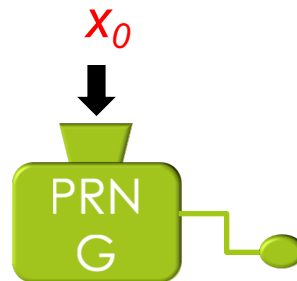
- **The sequence cannot be really random!**

An algorithm produces known output, by definition.

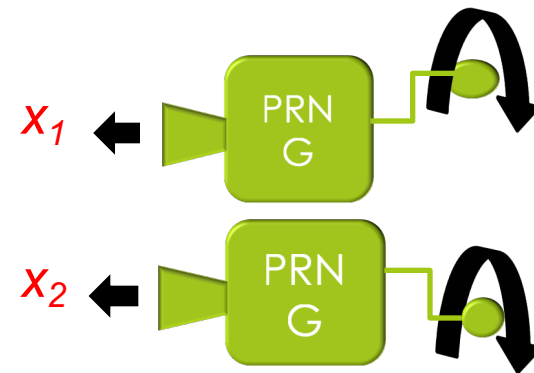
# (Pseudo) Random Number Generator

- A (software) machine to produce sequence  $x_1, x_2, x_3, x_4, x_5, \dots$  from  $x_0$

- Initialize / seed:



- Get pseudorandom numbers:

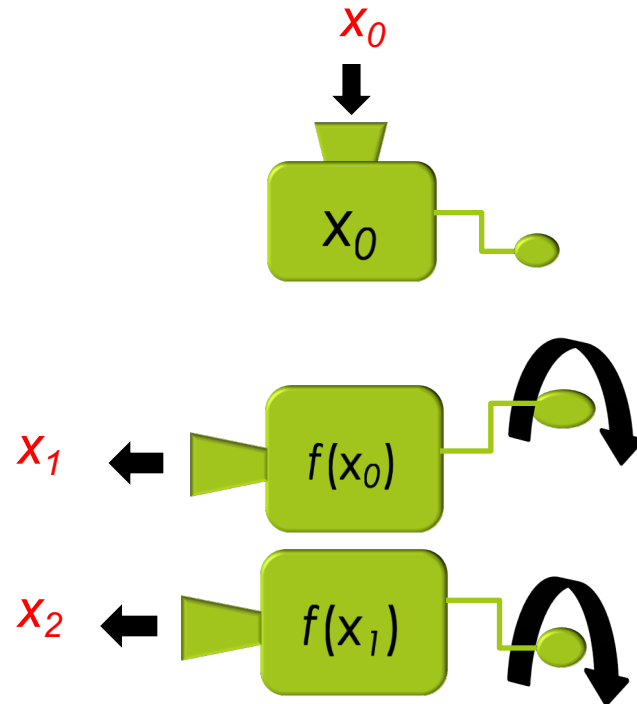


# (Pseudo) Random Number Generator

- A (software) machine to produce sequence  $x_1, x_2, x_3, x_4, x_5, \dots$  from  $x_0$

- Initialize / seed:

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



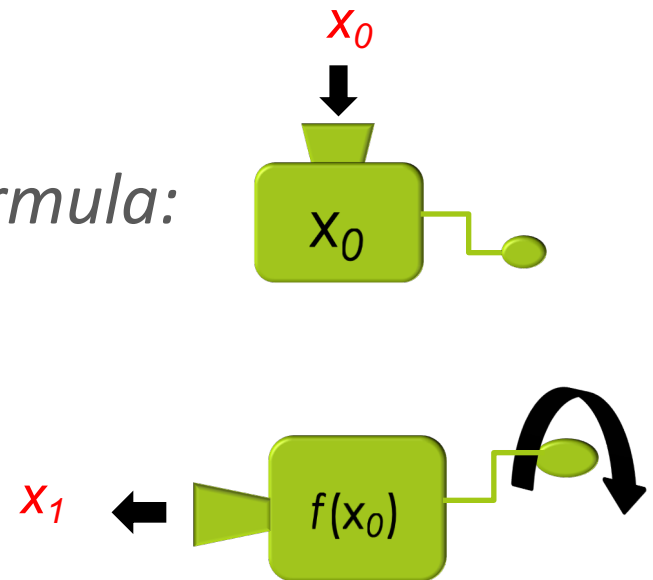
Idea: **internal state determines the next number**

# 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

# Example Linear Congruential Generator (LCG)

```
current_x = 0           # global internal state/ seed
def prng_seed(s) :     # seed the generator
    global current_x
    current_x = s

def prng_fn(n):        # LCG (a = 1, c = 7, m = 12)
    return (n + 7) % 12)

def prng() :          # state updater
    global current_x
    current_x = prng_fn(current_x)
    return current_x
```

First 12 numbers: 1, 8, 3, 10, 5, 0, 7, 2, 9, 4, 11, 6

**Does this look random to you?**

# Example LCG

First 20 numbers:

5, 0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10,  
5, 0, 7, 2, 9, 4, 11, 6, ?

Random-looking?

- What do you think the next number in the sequence is?
- **Moral: just eyeballing the sequence not a good test of randomness!**
- This generator has a **period** that is too short:  
It repeats too soon.

# Another PRNG

```
def prng2(n):  
    return (n + 8) % 12 # a=1, c=8,  
    m=12  
  
>>> prng_seed(6)  
>>> for i in range(12)  
[8, 4, 0, 8, 4, 0, 8, 4, 0, 8, 4, 0]
```

Random-looking?

**Moral: choice of  $a$ ,  $c$ , and  $m$  crucial!**

# PRNG Period

Let's define the PRNG *period* as the number of values in the sequence before it repeats.

5, 0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10,  
5, 0, 7, 2, 9, 4, 11, 6, ...

**prng1, period = 12** next number = (last number + 7) mod 12


8, 4, 0, 8, 4, 0, 8, 4, 0, 8, ...

**prng2, period = 3** next number = (last number + 8) mod 12

**We want the longest period we can get!**



# Picking the constants $a$ , $c$ , $m$

- **Large value for  $m$** , and appropriate values for  $a$  and  $c$  that work with this  $m$   
 a very long sequence before numbers begin to repeat.
- Maximum period is  $m$

# Picking the constants $a$ , $c$ , $m$

The LCG will have a **period of  $m$**  (the maximum) if and only if:

1.  $c$  and  $m$  are *relatively prime*  
(i.e. the only positive integer that divides both  $c$  and  $m$  is 1)
2.  $a-1$  is **divisible by all prime factors** of  $m$
3. If  $m$  is a **multiple of 4**, then  $a-1$  is also a multiple of 4

*(Number theory tells us so)*

If  $c$  and  $m$  are not relatively prime, then  $c = pk$  and  $m = qk$  for some  $k$ .

-> After  $q/p$  iterations you come back to the seed

# 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: `prng_fn` ( $a = 1, c = 7, m = 12$ )
  - Factors of 7: 1, 7    Factors of 12: 1, 2, 3, 4, 6, 12
  - 0 is divisible by all prime factors of 12  $\rightarrow$  true
  - if 12 is a multiple of 4, then 0 is also a multiple of 4  $\rightarrow$  true
- `prng_fn` will have a period of 12

# Exercise for you

(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  $a-1$

$$x_{i+1} = (5x_i + 3) \text{ modulo } 8$$

$$x_0 = 4$$

$$a = 5$$

$$c = 3$$

$$m = 8$$

- What is the period of this generator? Why?
- Compute  $x_1, x_2, x_3$  for this LCG formula.

# Exercise for you

(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  $a-1$

$$x_{i+1} = (5x_i + 3) \text{ modulo } 8$$

$$x_0 = 4$$

$$a = 5$$

$$c = 3$$

$$m = 8$$

- ▣ Factors of 3: 1, 3    Factors of 8: 1, 2, 4
- ▣ 4 is divisible by all prime factors of 8  $\rightarrow$  true
- ▣ If 8 is a multiple of 4, then 0 is also a multiple of 4  $\rightarrow$  true

# LCGs in the Real World

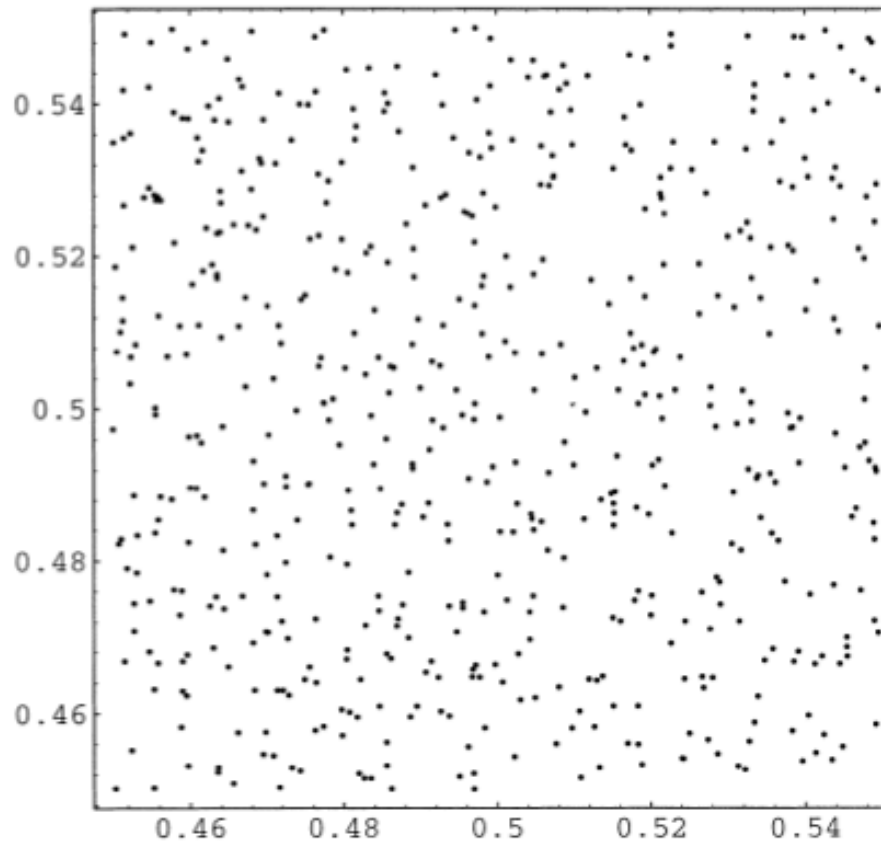
- glibc (used by the compiler gcc for the C language):  
 $a = 1103515245, c = 12345, m = 2^{32}$
- *Numerical Recipes* (popular book on numerical methods and analysis):  
 $a = 1664525, c = 1013904223, m = 2^{32}$
- Random class in Java:  
 $a = 25214903917, c = 11, m = 2^{48}$

# Some pitfalls of PRNGs

- ❑ **Predictable seed.** Example: famous Netscape security flaw caused by using system time as seed.
- ❑ **Equal sequences** of random numbers are possible when running many applications at the same time.
- ❑ **Hidden correlations** between successive values in the sequence of  $x$  values.
- ❑ High quality but **slow**

# Finding hidden correlations

*P. Hellekalek / Mathematics and Computers in Simulation 46 (1998) 485–505*



Looking good!

Fig. 1. LCG( $2^{31}$ , 65539, 0, 1) Dimension 2: Zoom into the unit interval.



# Finding hidden correlations

*P. Hellekalek/Mathematics and Computers in Simulation 46 (1998) 485–505*

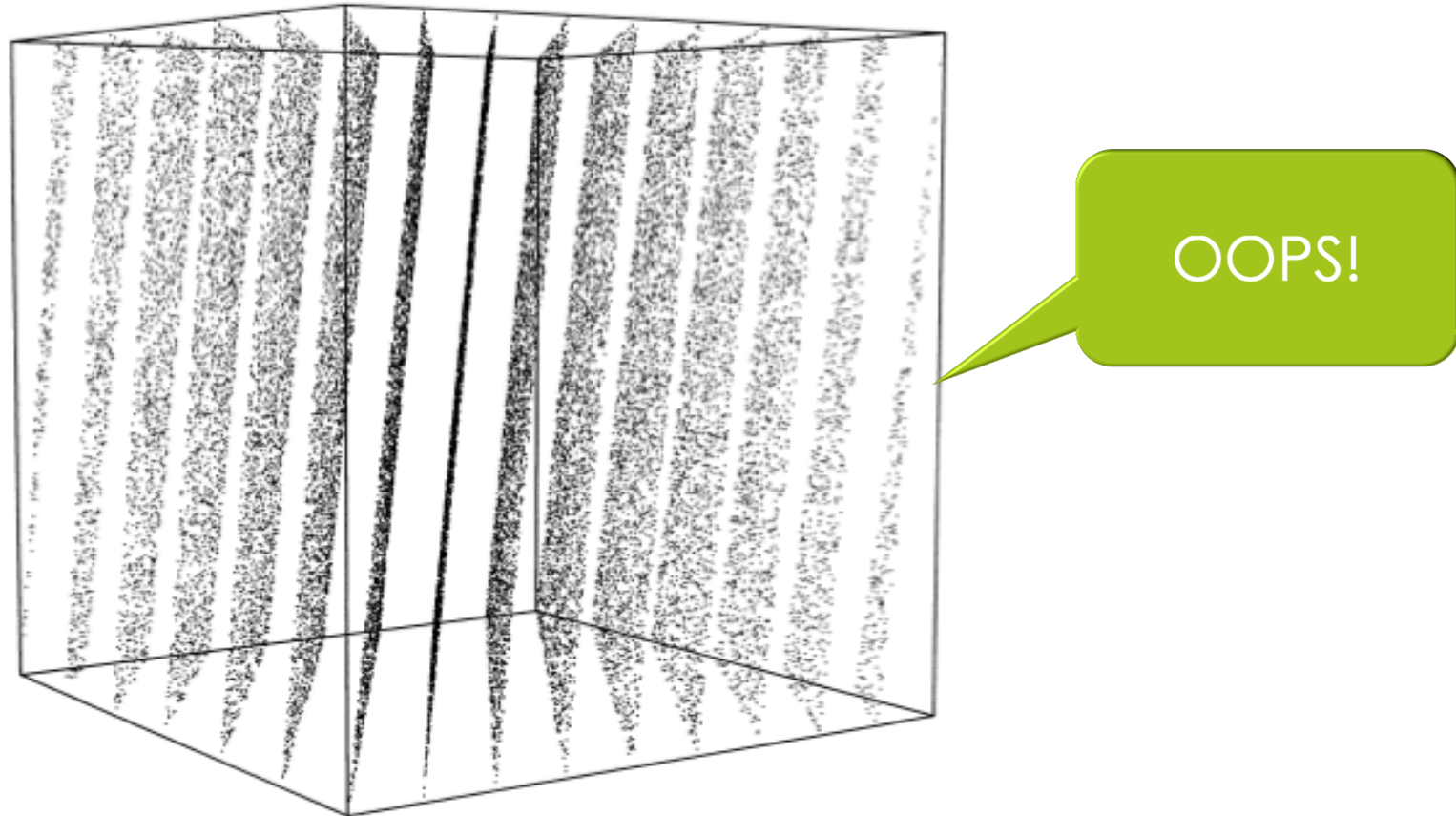


Fig. 2. LCG( $2^{31}$ , 65539, 0, 1) Dimension 3: The 15 planes.

# Advice from an expert

- Get expert advice 😊

*A good generator is not so easy to find if one sets out to design it by oneself, without the necessary mathematical background. On the other hand, with the references and links we supply, a good random number generator designed by experts is relatively easy to find.*

– P. Hellekalek

# Using random number generators in Python

# 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
```

# List Comprehensions

- One output from a random number generator not so interesting when we are trying to see how it behaves

```
>>> randint(0, 99)
```

```
42
```

So what?

- To easily get a list of outputs

```
>>> [ randint(0,99) for i in range(10) ]
```

```
[5, 94, 28, 99, 34, 49, 27, 28, 65, 65]
```

```
>>> [ randint(0,99) for i in range(5) ]
```

```
[69, 51, 8, 57, 12]
```

```
>>> [ randint(101, 200) for i in range(5) ]
```

```
[127, 167, 173, 106, 115]
```

# 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(10, 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']
```

```
>>> sample(colors, 2)  
['gray', 'red']
```

```
>>> [ sample(colors, 2) for i in range(3) ]  
[['gray', 'red'], ['blue', 'green'],  
['blue', 'black']]
```

```
>>> shuffle(colors)  
>>> colors  
['red', 'gray', 'black', 'blue', 'green']
```

In practice...



# Adjusting Range

- Suppose we want to use our LCG with period  $n$  ( $n$  is very large)
- ... but we want to play a game involving dice.  
(each side of a die has a number of spots from 1 to 6)
- How do we take an integer between 0 and  $n$ , and obtain an integer between 1 and 6?
  - Forget about our LCG and use `randint(?, ?)`

*Great, but how did they do that?*

what values  
should we use?



# Adjusting Range



- Specifically: our LCG is the Linear Congruential Generator of glib (period =  $2^{31} = 2147483648$ )
- We call `prng()` and get numbers like 1533190675, 605224016, 450231881, 1443738446, ...
- We define:

```
def roll_die():  
    roll = prng() % 6 + 1  
    assert 1 <= roll and roll <= 6  
    return roll
```

- What's the smallest possible value for `prng() % 6` ?
- The largest possible?

# Random range



- ▣ Instead of rolling dice, we want to pick a random (US) presidential election year between 1788 and 2012
  - ▣ election years always divisible by 4
- ▣ We still have the same LCG with period 2147483648.

What do we do?

- ▣ Forget about our LCG and use `randrange(1788, 2013, 4)`

*Great, but how did they do that?*

# Random range



- Remember, `prng()` gives numbers like  
1533190675, 605224016, 450231881, 1443738446, ...

```
def election_year() :  
    year = ?  
    assert 1788 <= year and year <= 2012 and year % 4 == 0  
    return year
```



# Random range

- First: Let's generate a random number 0, 1, 2,... One for every election year.
- Think, *how many numbers are there in the range we want? That is, how many elections from 1788 to 2012?*
  - 2012 – 1788? No!
  - $(2012 - 1788) / 4$ ? Not quite! (there's one extra)
  - $(2012 - 1788) / 4 + 1 = 57$  elections

```
def election_year() :  
    election_number = prng() % ( (2012 - 1788) // 4 + 1)  
    assert 0 <= election_number and election_number <= 56  
    year = ?  
    assert 1788 <= year and year <= 2012 and year % 4 == 0  
    return year
```

# Random range



- Okay, but now we have random integers from 0 through 56
  - good, since there have been 57 elections
  - bad, since we want years, not election numbers 0 ... 56

```
def election_year() :  
    election_number = prng() % ( (2012 - 1788) // 4 + 1)  
    assert 0 <= election_number and election_number <= 56  
    year = election_number * 4 + 1788  
    assert 1788 <= year and year <= 2012 and year % 4 == 0  
    return year
```

# Random range



Sample output:

```
>>> [ election_year() for i in range(10) ]  
[1976, 1912, 1796, 1800, 1984, 1852, 1976, 1804, 1992, 1972]
```

The same reasoning will work for a random sampling of any arithmetic series. *Just think of the series and let the random number generator take care of the randomness!*

- How many different numbers in the series?  
If there are  $k$ , randomly generate a number from 0 to  $k$ .
- Are the numbers separated by a constant (like the 4 years between elections)?  
If so, multiply by that constant.
- What's the smallest number in the series?  
Add it to the number you just generated.