B35APO: Computer Architectures

Lecture 01. Introduction

Pavel Píša
pisa@fel.cvut.cz

Petr Štěpán
stepan@fel.cvut.cz

6. March, 2024

Outline

Motivation

What you can do to speed up your program:

- Use a more powerful computer:
  - Increase CPU performance/throughput
    - CPU frequency
    - CPU efficiency - how many operations can it perform in 1 one clock cycle
- Change program:
  - Improve memory efficiency
  - Parallelize program:
    - Increase number of utilized CPU cores
    - Use instructions for parallel processing

Motivation

Does parallelization have some limitations:

- You can never parallelize an entire program
- Amdahl's law
  - α fraction (percent) of the program cannot be parallelized
  - the rest $(1 - \alpha)$ of the program can be accelerated to spent $\frac{1-\alpha}{p}$ time fraction when p processors are used
  - acceleration ratio for p processors $S(p) = \frac{\alpha+1-\alpha}{\alpha+\frac{1-\alpha}{p}} = \frac{p}{1+\alpha\cdot(p-1)}$
  - The acceleration limit for infinite processors is $S(p \to \infty) = lim_{p\to\infty} \frac{p}{1+\alpha\cdot(p-1)} = \frac{1}{\alpha}$
    - For example, for $\alpha = 0.3$ limit is $S(p \to \infty) = 3.\overline{3}$, i.e. the program cannot be accelerated more than $3.\overline{3}$ times.
- In practice, the more processes you have, the more difficult it is to prepare data for parallelization.

Motivation

Why to study computer architectures:

- Learn how the computer works when executes your program and where are the opportunities to make the program more efficient
  - find out what limits the a computer the computation speed (processor speed, memory size, main memory latency, number of processor cores) and test/choose another HW
  - find out if the program can be modified to use better available resources
    - modify memory access pattern/order/data structures to optimize memory throughput and cache
    - modify the program to use less branch and jump instructions (less stalls and flushes of speculative work)
    - parallelize the computation, use specialized HW - GPU, accelerator units e.g. Coral USB or Intel Neural Compute Stick 2.
- Demand for graduates combining artificial intelligence and embedded systems knowledge
- If the computer is only BlackBox for the programmer, then the resulting programs are almost certainly inefficient.

Content of the Course lectures

All the basic components of your computer will be presented:

- CPU - Central Processing Unit (Processor)
- memory hierarchy - cache/RAM/external storage (disk/SSD)
- Input and Output (I/O) - keyboard, mouse, display, network card, HW driver principles
- Exceptions and Interrupts - efficient collaboration between user program, operating system, CPU and HW

Motivation to attend lectures:

- You'll learn topics and it will be easier to prepare for the exam
- If you answer the quiz question correctly at the end of the lecture, you will get an activity point
  - Not every lecture, first scored quiz next week
  - Limit of total activity points for the course is 10

Seminaries Plan and Assessment

- 4 homeworks (smaller tasks) - 36 points
  - 2 C-language programs
  - 2 form based quizzes
  - requirement to pass 3 from 4 homeworks at their specified minimal level
- Semester project - 24 points
  - Team project – pairs (or individual)
  - Educational hardware kits designed for the course (MZ_APO)
- Optional tasks and or activity during exercises/lectures – up to 10 points

Grading	Points
A	>=90
B	80 – 89.9
C	70 – 79.9
D	60 – 69.9
E	50 – 59.9
F	<50

Exam:

- written part 30 points, min 15 points
- oral part ± 10 points

Followup Courses

If you are interested in this subject, the following subjects are related to it:

- BE4M35PAP – Advanced Computer Architectures
- B3B38VSY – Embedded Systems
- BE4M38AVS – Application of Embedded Systems
- B4B35OSY – Operating Systems
- BE5B35LSP – Logic Systems and Processors

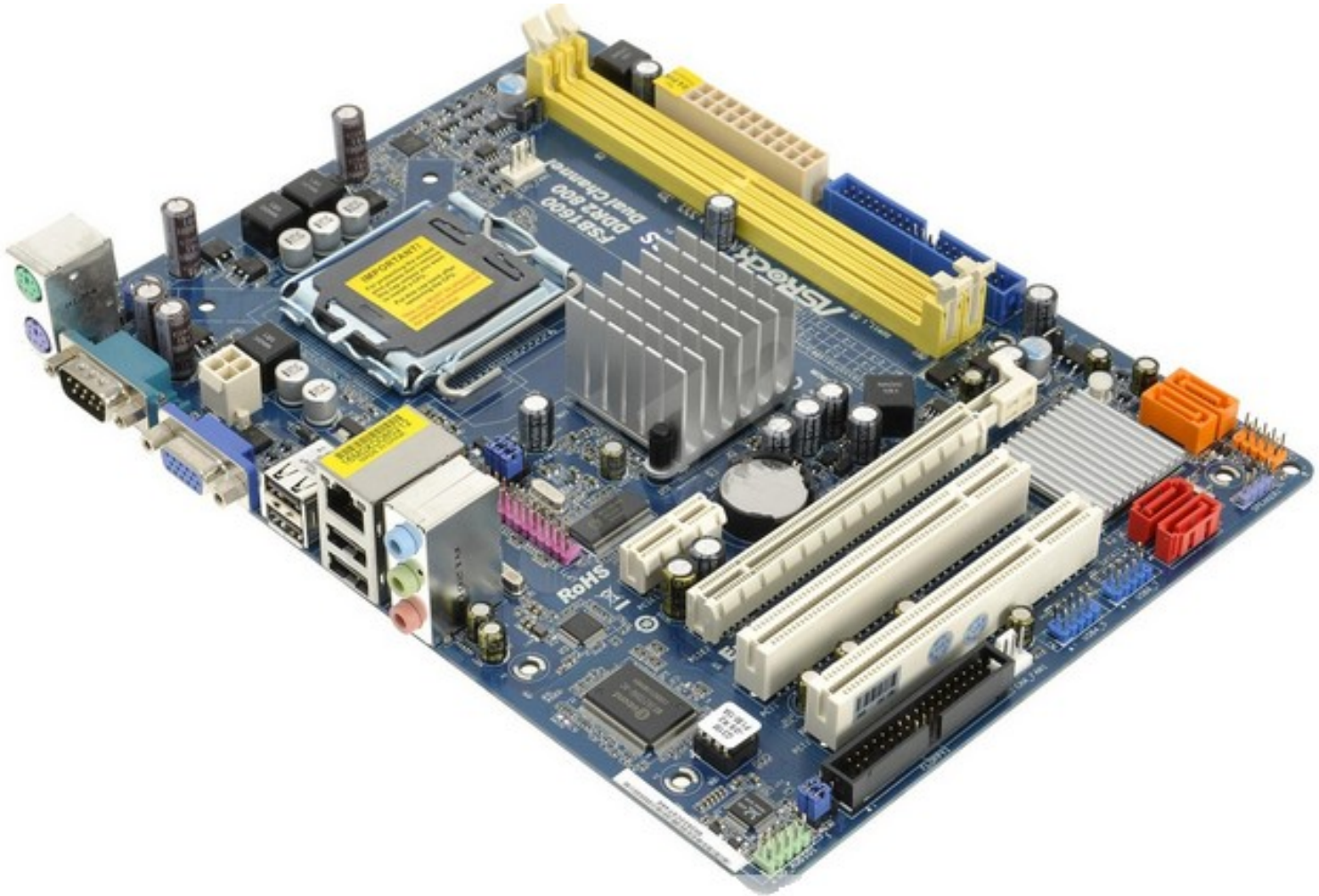
Literature and Resources

- PATTERSON, David A. a John L. HENNESSY. Computer organization and design RISC-V edition: the hardware/software interface. Second Edition. Cambridge: Elsevier, [2021]. ISBN 978-0-12-820331-6. (12 kusů v ústřední knihovně ČVUT)
- web:
  - `https://cw.fel.cvut.cz/wiki/courses/b35apo/`
  - `https://dcenet.felk.cvut.cz/apo/`
  - `https://comparch.edu.cvut.cz/`
- Courses at other universities:
  - MIT 6.004/6.191 – Computation Structures (public resources `https://computationstructures.org/`)
  - Computation Structures | Electrical Engineering and Computer Science | MIT OpenCourseWare (2015)
  - Computer System Architecture | Electrical Engineering and Computer Science | MIT OpenCourseWare (2005)
- Other courses at CTU:
  - FIT: BIE-APS.21 Architectures of Computer Systems

Outline

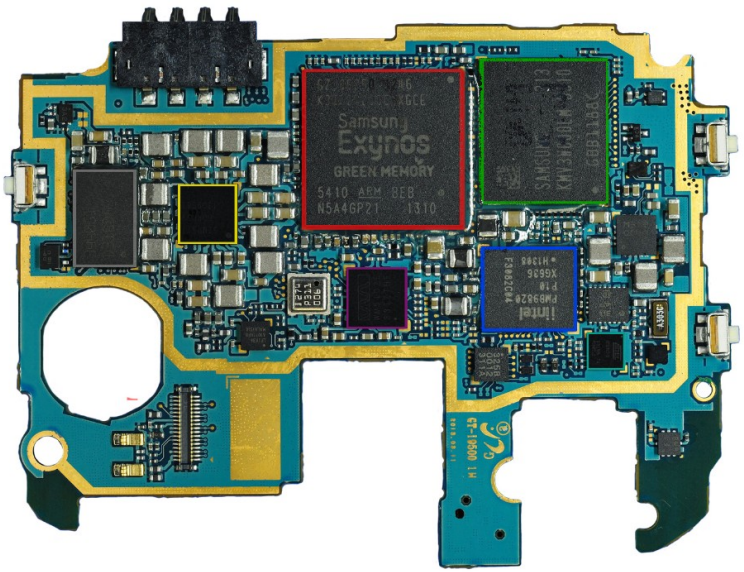
What is Inside Computer

Mainboard (of the Computer):



What is Inside Computer

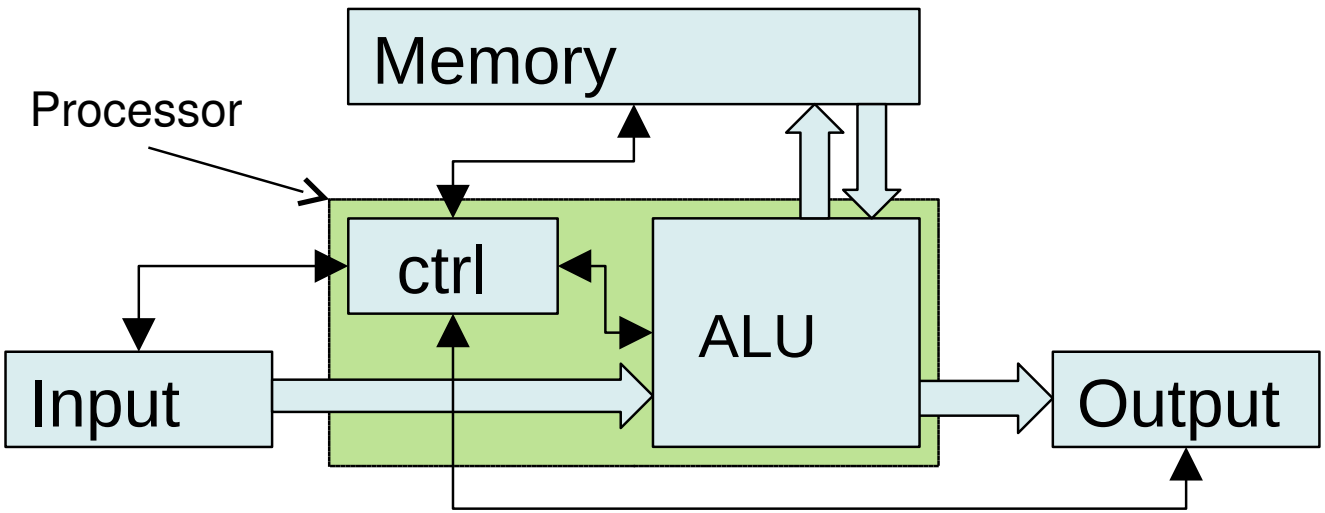
Disassembled mobile phone:



von Neumann

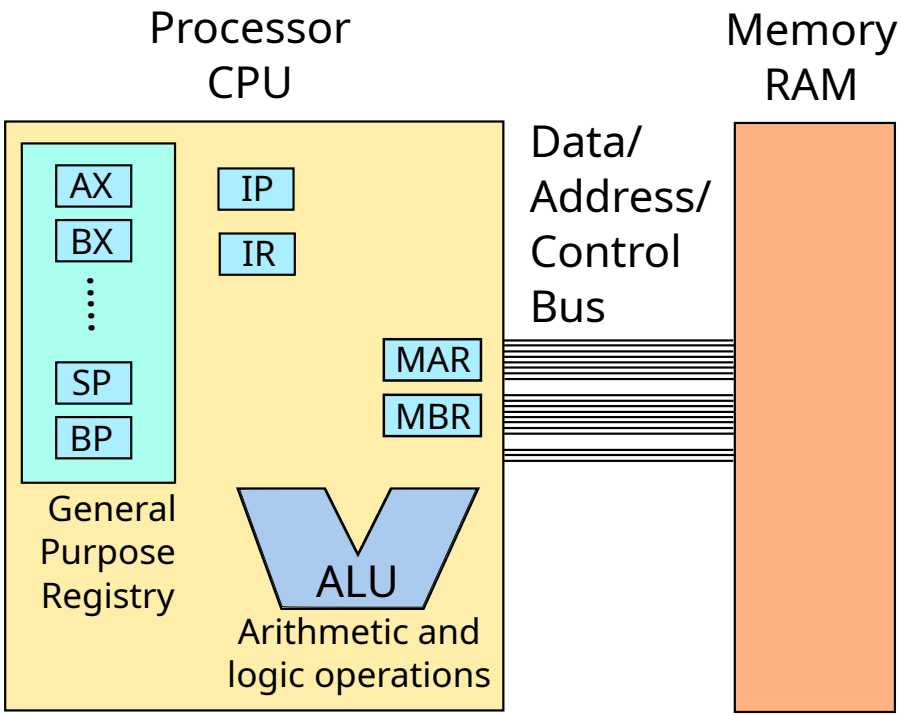
The general computer concept presented by Johnem von Neumannem (1903-1957), Hungarian-American mathematician, physicist:

- Processor - Central Processing Unit - CPU
- Memory, Random-access Memory
- Input/Output



CPU – Central Processing Unit

- general purpose (user/integer) registers (GPRs) – usually 8, 16, 32 or 64-bits wide according to CPU architecture
- CPU fetches instructions from the memory (basically in order) and executes each fetched and decoded instruction



- PC (program counter) or IP (instruction pointer) – special register holding address of the instruction to be executed
- ALU (arithmetic-logic unit) – CPU component which proceed add, subtract, multiply, divide and other arithmetic and logical operations

Main Memory

Memory holds data values - bytes, words.
If you already know some programming language then you can consider memory as array of same sized elements, i.e. for C-language:

```
unsigned char RAM[16 * 1024 * 1024 *1024]; // 16GiB RAM
```

The memory allows to read from specified location (address):

```
register R10 = RAM[address];
```

and write to specified address (source/target is usually one of GPRs):

```
RAM[address] = R10;
```

Address is numeric index into array which is usually encoded/transferred in binary repreznetation on the signals connecting CPU to the memory

Processor Instructions

- The instructions encode all (limited set of operations) that the processor can perform
- The basic instructions are:
  - store a constant in the register
  - load data from memory into the register
  - perform a mathematical operation on the registers and store the result in the register
  - store data from the register in memory
  - compare two numbers
  - perform other instructions according to the result of the comparison (change the PC to a value other than the following instruction = perform a jump or branch in the program)
- all programs, whether created in C language, Python, even programs performing very complex calculations are realized (compiled into) these simple processor instructions or interpreted by program written in these instructions

Processor Instructions – Instruction Set Architecture

Instruction Set Architecture (ISA)

- is a complete instruction set specification for given chip/architecture, defines address modes, data widths, operations, encoding
- i.e. x86 (IA-32), x86-64 (AMD64, EM64T, IA-32e), ARM32, AArch64 (ARM64), AVR, MIPS, RISC-V
- given ISA specifies:
  - the list of known processor machine instruction set
  - supported data types, their widths and encoding (integers, signed integers, real/floating point numbers, vector types)
  - the set of user visible general purpose register, optionally floating point ones and special status and control ones
  - addressing modes (how the address to memory location can be formed)
  - memory organization (if byte accessible, word, halfword, etc.)

Two Basic Concepts How to Architect ISA

RISC

Reduced Instruction Set Computer

- Usually a smaller number of instructions
- All instructions have the same same width and encoding rules (sometimes half length aliases for more dense encoding)
- Less number and simpler addressing modes
- Mathematical operations ALU only within registers (i.e. load-store architecture)

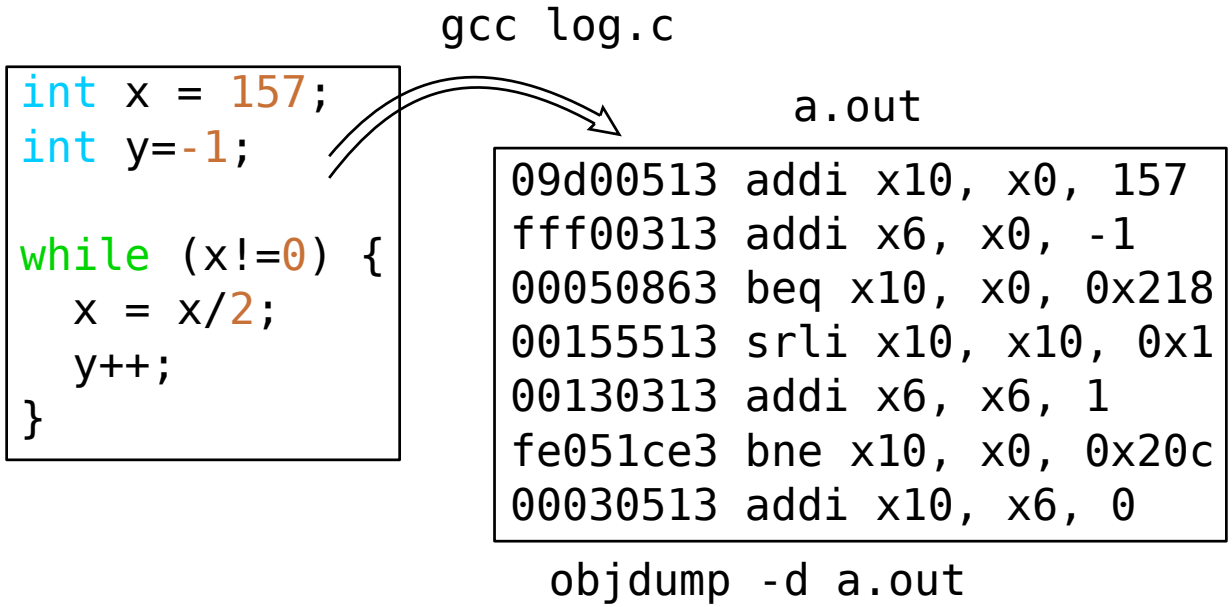
CISC

Complex Instruction Set Computer

- Usually a larger number of instructions
- The length of instructions even from 1 byte to e.g. 14 bytes, the most common instructions are the shortest
- Usually many complex addressing modes
- Data processing operations (ALU) even with values read/written to memory

Higher Level Program to Machine Level Compilation

How come you have not heard about machine instructions yet?



- Programming in assembler (assembly language) is often inefficient, hard, takes long time and application is fixed to one ISA.
- Compiler translates higher programming language directly into machine instructions of the target processor
- Cross-compilation – program translation for a different processor ISA than is used on machine to compile program

Processor – Hardware Implementation

What does a processor consist of (physically)?

- You may have heard that a processor contains, say, 16 billion transistors
- In 1965, Intel co-founder Gordon Moor formulated a law:
  - The number of transistors that can be placed on an integrated circuit doubles every 18 months or so, at the same price.
- More or less holds true until today, even though we are getting to the limits of physical possibilities.

Why do we need transistors in a processor (CPU)?

- To implement the Boolean algebra and registers (combinational and sequential logic).

We often can focus on block building system at given level of hierarchy, i.e. ISA, register transfer level RTL, logic functions, gates, transistors, semiconductor/silicon structures, atomic-gratings

Outline

Boolean Algebra – Values and Operations

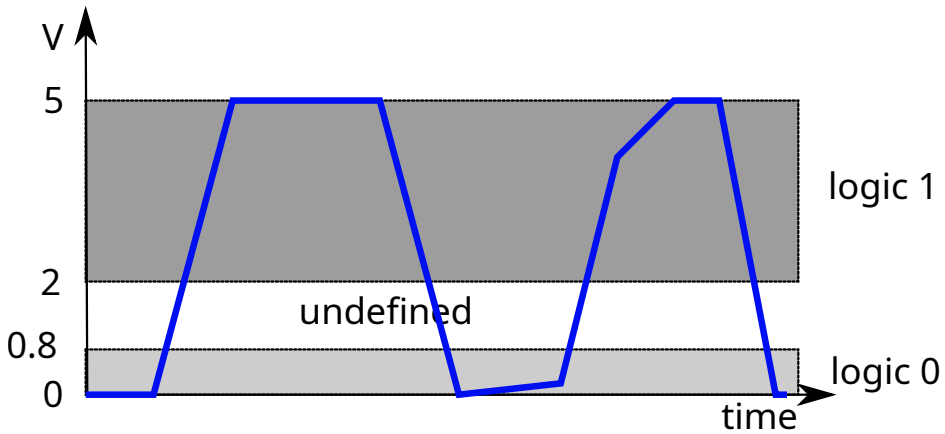
Boolean algebra is mathematical structure (see group theory):

- Only two values (states) for variables are allowed (0 and 1)
  - 0/1, or False/True, or indicator/LED is on or off, or voltage representation 0V/5V
- Addition operation (or, ||, $\vee$)
  - 0+0=0 0+1=1
  - 1+0=1 1+1=1
- Multiplication operation (and, &&, $\wedge$)
  - 0*0=0 0*1=0
  - 1*0=0 1*1=1
- Inversion, complementary element (negace, !, not, $\neg$)
  - -0 = 1
  - -1 = 0

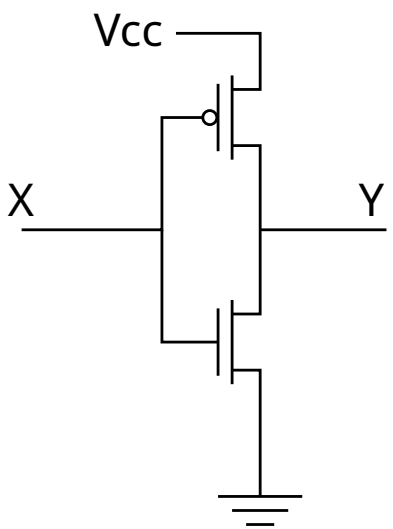
Boolean Algebra – States Representation

Boolean algebra can be implemented well with voltages and transistors

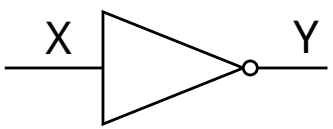
The voltage of a single conductor/signal to the ground defines a boolean value.



Example: A boolean not operation, one X input conductor, one Y output conductor.



symbol neg



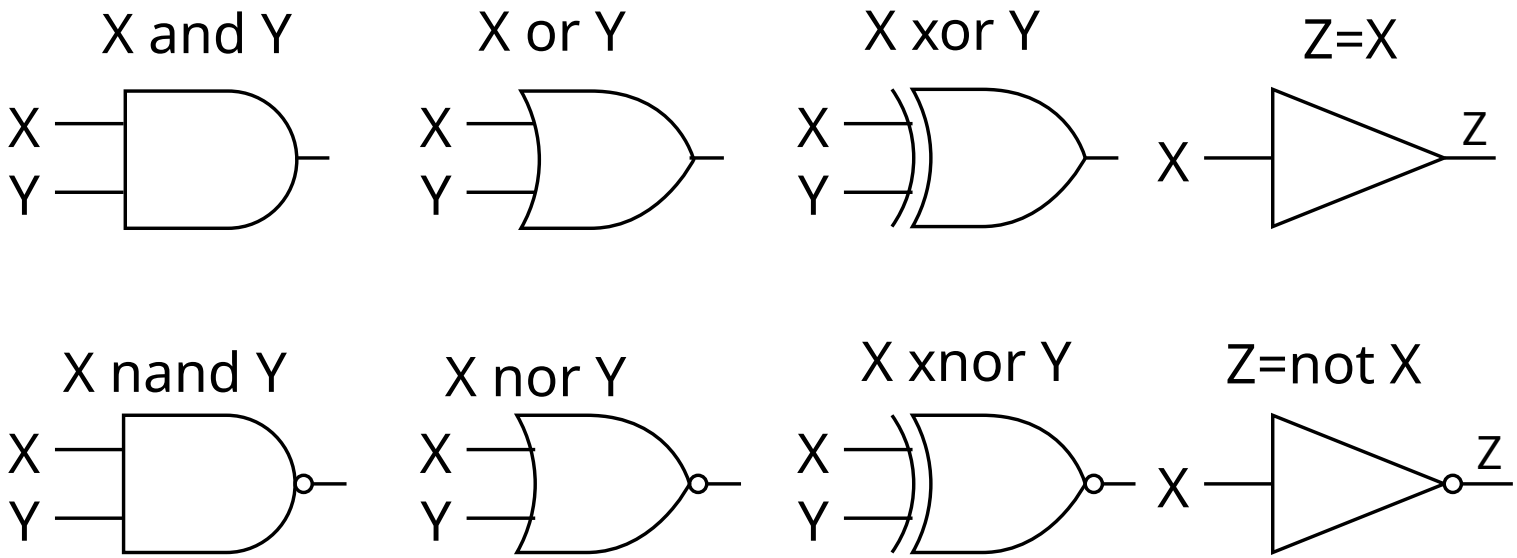
Boolean Algebra – NAND, NOR, XOR

Extended set of binary operations `nand`, `nor`, `xor` which can be decomposed to basic operations (this holds for circuit of arbitrary complexity) (`and`, `or`, `not`):

```
X nand Y  = not(X and Y)
X nor Y   = not(X or Y)
X xor Y   = (X or Y) and (not(X and Y))
          = (X or Y) and (X nand Y)
```

Boolean Algebra – Logic Gates and Symbols

List of the logic gates and their symbols for binary operations

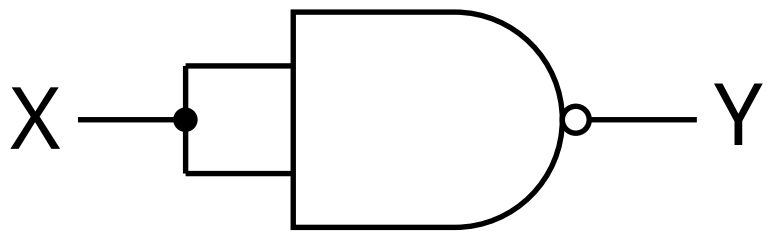


Summary table of basic logic gates:

X	Y	X and Y	X or Y	X xor Y	X nand Y	X nor Y	X xnor Y
0	0	0	0	0	1	1	1
0	1	0	1	1	1	0	0
1	0	0	1	1	1	0	0
1	1	1	1	0	0	0	1

Boolean Algebra – Quiz 1

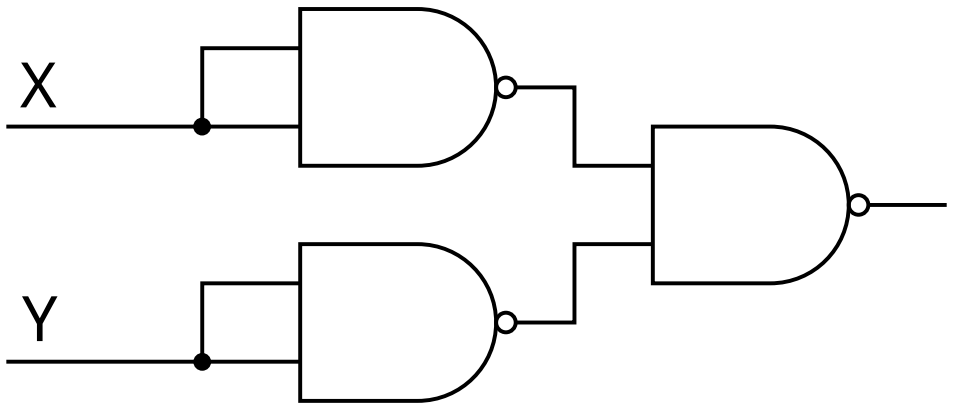
Signals/wires can also branch. What does the following circuit do?



A) it can't be connected like this
B) the result is X and Y
C) the result is X and not Y
D) the result is not X

Logic/Combinational Circuits – Quiz 2

Some functions can be converted to gates even more efficiently than via the basic logic functions and, or, not.
The nand gate is the basic gate and all other gates can be built from it.



Quiz: What is the result function equivalent to:

A X and Y

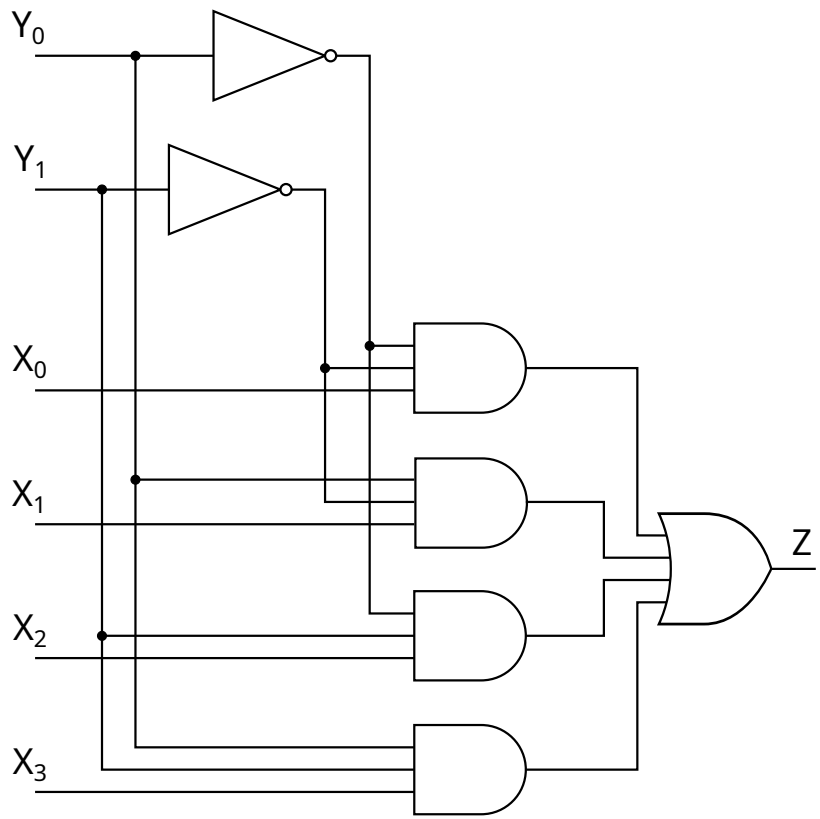
B X or Y

C X xor Y

D X nor Y

Logic/Combinational Circuits – Quiz 3

More complex circuits can be made up of basic logical elements - gates. The signals can branch, they are combined together by some logical operation. The result is always only the logical values 0 or 1.



Quiz: What is the circuit function:

A Nothing reasonable, it is just a tangle of wires

B It is a multiplexor, the value Z is one of the signals X according to the encoded value at Y

C It is a divider, the value Z is X/Y

D The value Z is 1 if $X > Y$

Binary Encoding/Representation/Numeral System – Quiz 4

Quiz: One wire represents one value, either 0 or 1. How to represent more values/symbols, like all integers from 0 to 255 (i.e. one byte)?

A One wire has 8 different voltage levels

B One wire represents 8 different 0/1 values consecutively over time

C Eight wires, each representing one of the 0/1 values at one time

D 256 wires, only one has a value of 1 others

Binary/Base-2 Numeral System

Numbers represented by more digits or bits – numeral system
Especially in the case of binary encoding

- multiple single-bit parallel conductors/wires/signals
  - usually 8, 16, 32, 64 (powers of 2)
  - sometimes only parts of word is enough, like 5-bit (32 values)
- order of conductors is important
  - each conductor represents value at given power of 2
  - conductor a_i, total value $s = \sum_{i=0}^{63} a_i * 2^i$

Outline

Addition – Two Single Bit Values

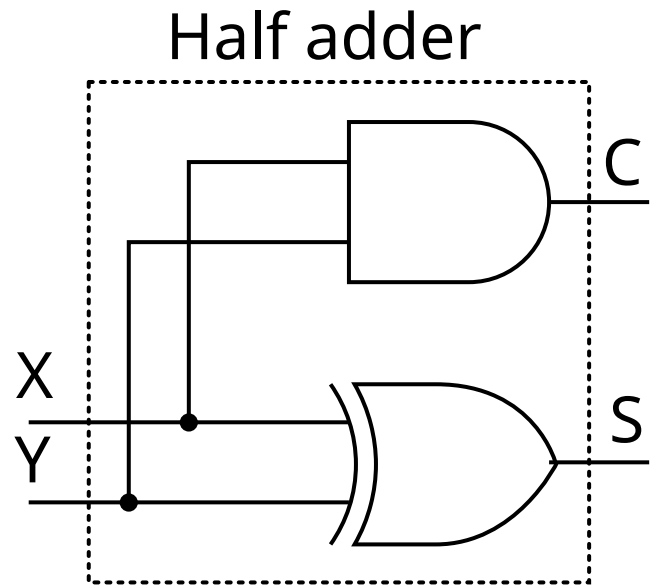
Addition of two single bit numbers:

X	Y	X+Y
0	0	00
0	1	01
1	0	01
1	1	10

The result of the arithmetic sum fits into two bits, C – carry, S – sum.

S = X `xor` Y
C = X `and` Y

This logic circuit is called a Half Adder.



Addition – Full Adder

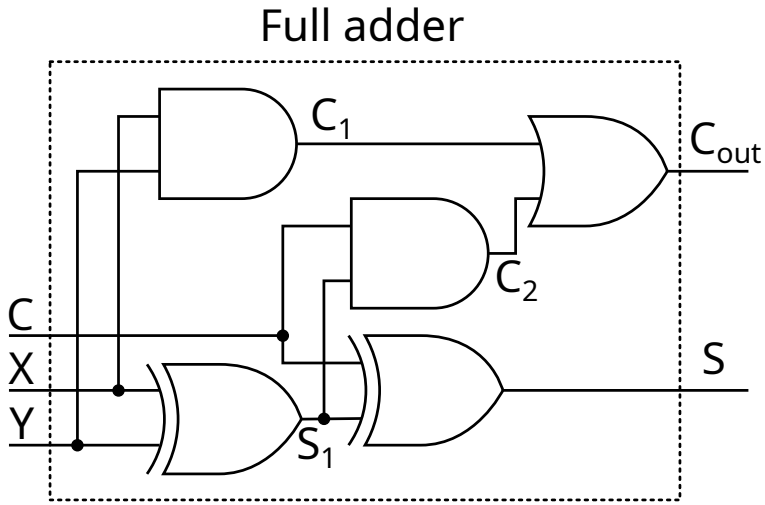
If we add multi-bit numbers we need to add three bit inputs later.

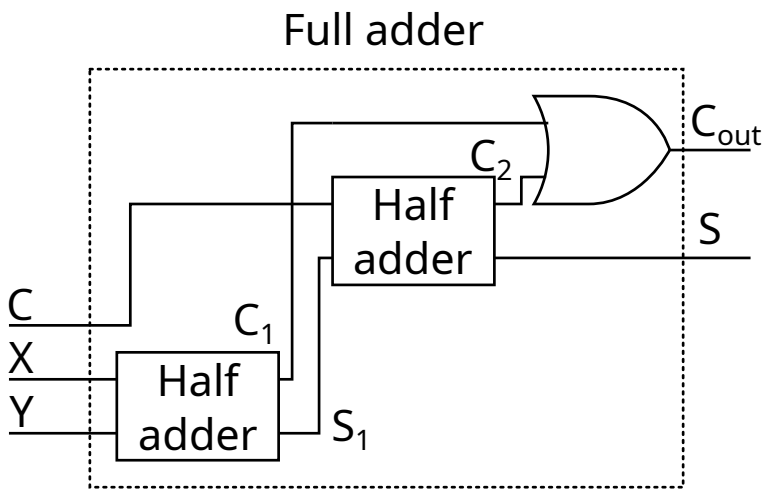
The result of the sum is again a two-bit number: C_{out} - carry, S - sum.

C	X	Y	C+X+Y
0	0	0	00
0	0	1	01
0	1	0	01
0	1	1	10
1	0	0	01
1	0	1	10
1	1	0	10
1	1	1	11

$S_1 = (X \texttt{ xor } Y)$
$S = (S_1 \texttt{ xor } C)$
$C_1 = (X \texttt{ and } Y)$
$C_2 = (S_1 \texttt{ and } C)$
$C_{out} = C_1 \texttt{ or } C_2$

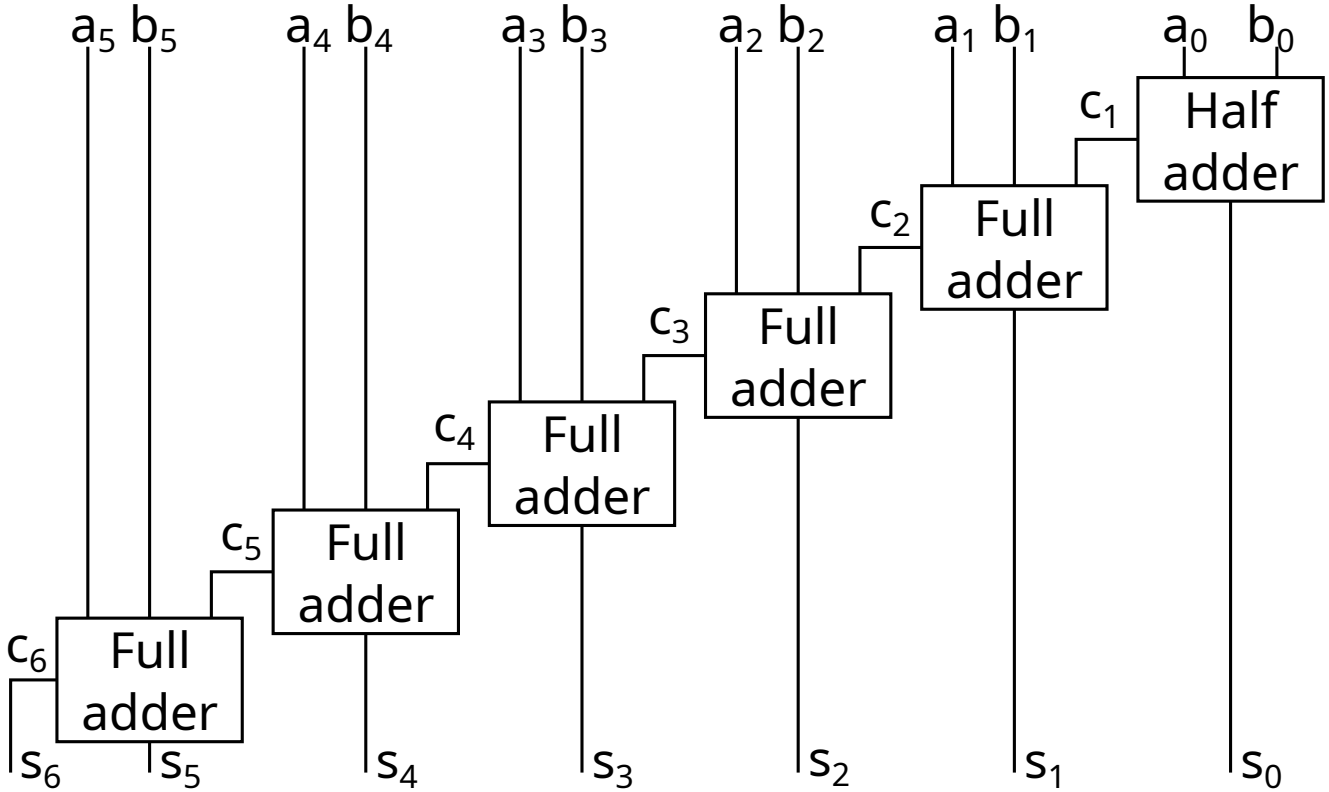
This logic circuit is called a full adder. (Full adder)





Addition – Ripple Carry Adder

The simplest multi-bit asthmatics adder is build by connection of one half adder and multiple full adders.
This type of adder is called Ripple Carry Adder (chained full-adders).



$a_5\,a_4\,a_3\,a_2\,a_1\,a_0 + b_5\,b_4\,b_3\,b_2\,b_1\,b_0 = s_6\,s_5\,s_4\,s_3\,s_2\,s_1\,s_0$ where $s_6 = c_6$

Addition – Ripple Carry Adder

What is the speed of Ripple Carry Adder?

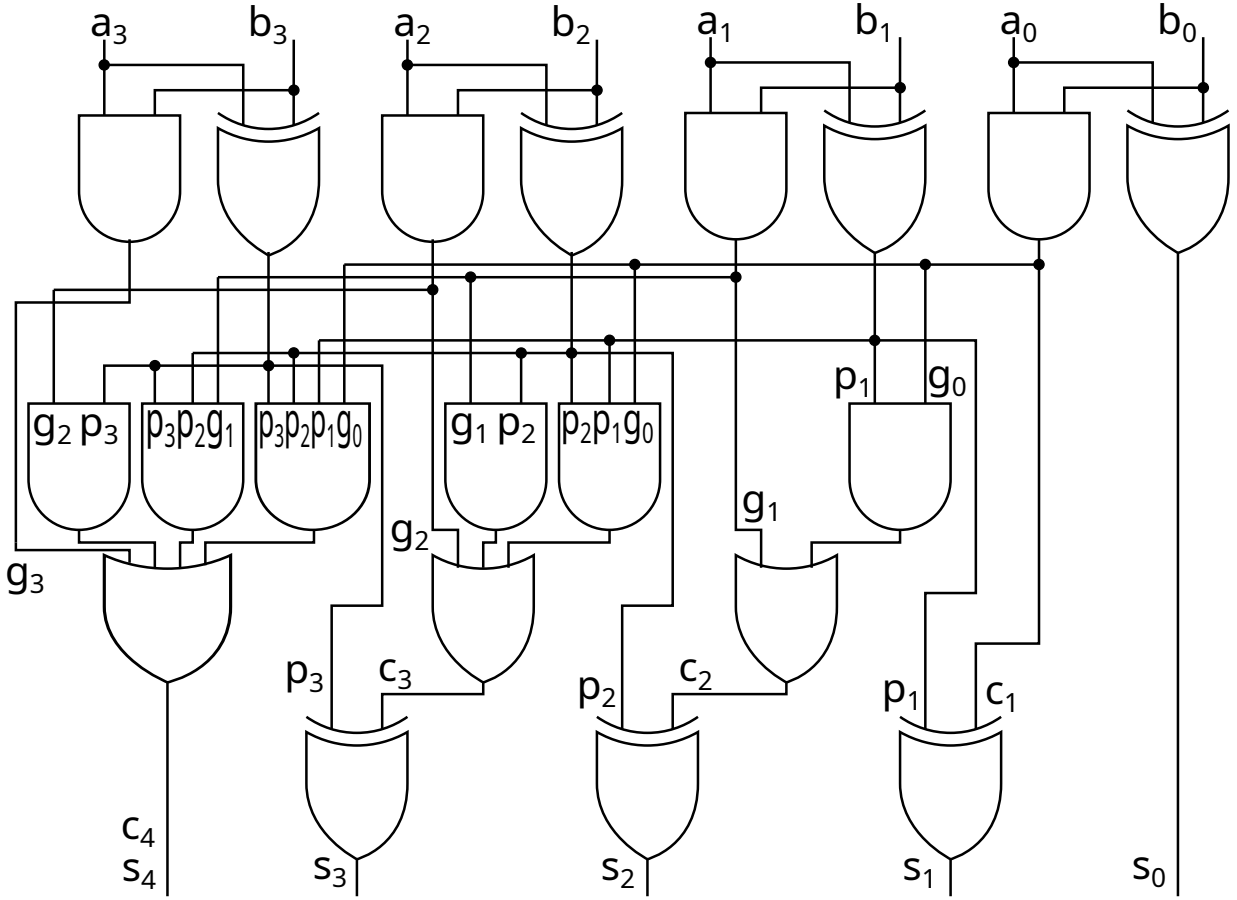
- If the signal propagation delay of one gate is N then the delay of Ripple Carry Adder for two 64-bit values is 63*2*N+N
- Is this important?
  - Yes, consider CPU clock 4GHz, one clock cycle takes 250 ps (picosecond)
  - The limit is even speed of the signal propagation ($<$ light speed) 0.3 mm/ps, it is maximal speed of information propagation
  - When about 10 ps gate propagation time is considered then Ripple Carray Adder adds 2 64-bit values in 1270 ps (that is more than 5 clock cycles – today ALU instruction latency is usually one cycle).
- Is the faster implementation possible?
  - Yes, Carry Lookahead Adder (CLA)

Addition – Carry Lookahead Adder

- Is it possible to determine C_1, C_2, ... , directly form the addends?
  - Yes, but for longer inputsit it is expensive, requires large gates count.
  - The following example consider 4-bit inputs
  - We can define two basic functions:
    - Carry generate – case $A_i = 1$ and $B_i = 1$ implicates $C_{i+1} = 1$ – carry is generated $G_i = A_i \wedge B_i$
    - carry propagate – case $A_i = 1$ or $B_i = 1$ implicates $C_{i+1} = C_i$ – carry will be propagated, of there is carry in the lower bit; $P_i = A_i$ xor B_i
  - For 4-bit input:
    - $C_1 = G_0$
    - $C_2 = G_1 \vee (C_1 \wedge P_1) = G_1 \vee (G_0 \wedge P_1)$
    - $C_3 = G_2 \vee (C_2 \wedge P_2) = G_2 \vee (G_1 \wedge P_2) \vee (G_0 \wedge P_1 \wedge P_2)$
    - $C_4 = G_3 \vee (C_3 \wedge P_3) = G_3 \vee (G_2 \wedge P_3) \vee (G_1 \wedge P_2 \wedge P_3) \vee (G_0 \wedge P_1 \wedge P_2 \wedge P_3)$
    - it is easy to extend expression for higher order bit, but it span longer and longer

Addition – Carry Lookahead Adder

The adder sums two 4-bit inputs in the time equivalent to propagation delay on four (4) serial connected gates.

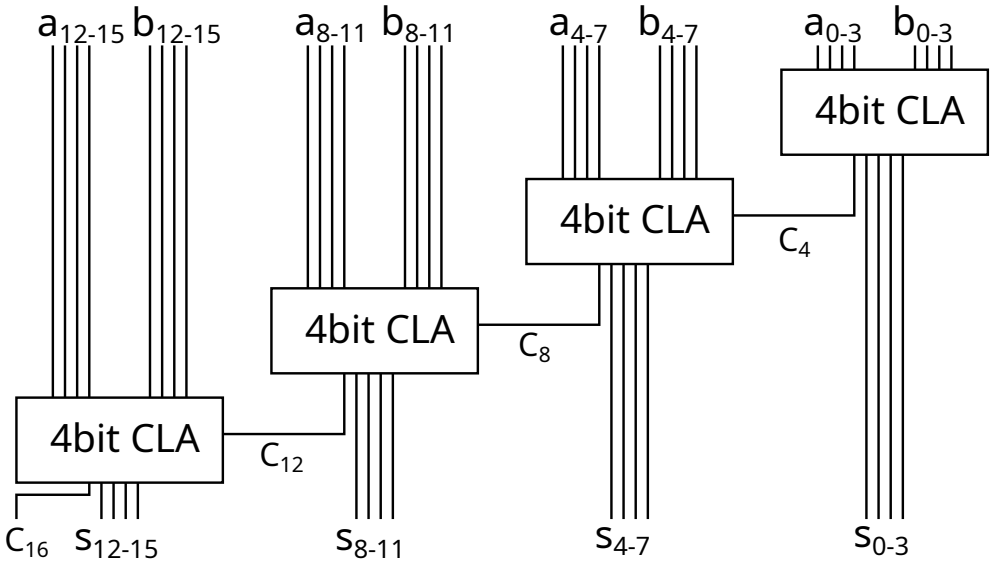


But for two 64-bit inputs, the adder would require 10^{20} gates to be built.

Addition – Carry Lookahead Adder

How to resolve reasonably fast function for larger numbers, i.e., 64-bit?

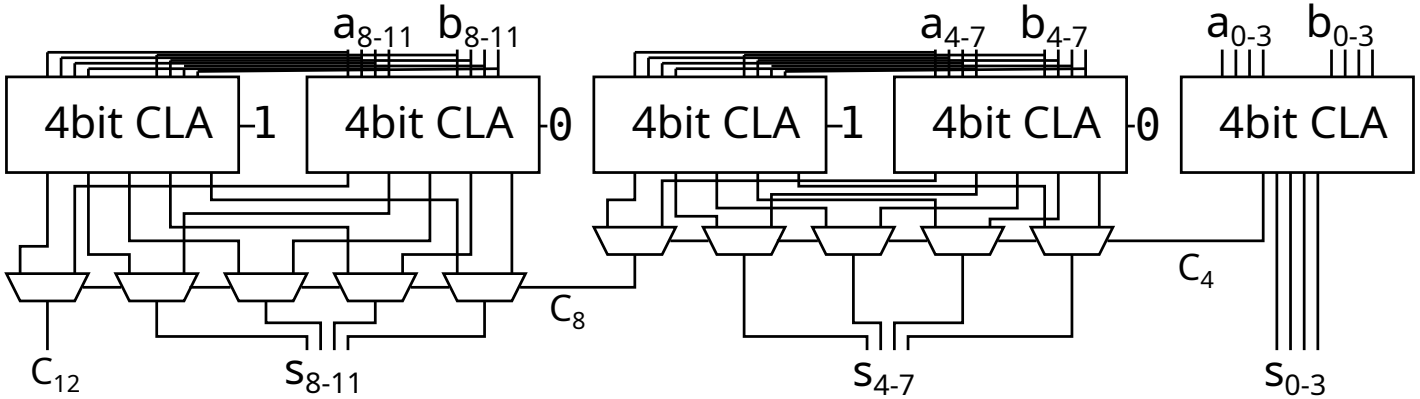
- The 4-bit adder is extended to allow C input from lower order bits
  - WARNING some gates has to be added
- We can chain these adder blocks.



- The latency/delay of 16-bit adder (at figure) is 16 gate delays
- The latency/delay of 64-bit adder is 64 gate delays, much better than 127 gate delays for simple Ripple Carry Adder.

Addition – Carry Select Adder

- It is sure that Carry is 0 or 1
- 4-bit CLA can be doubled and result is computed for both Carry inputs 0 and 1 in parallel
- Only the multiplexer control is chained which chooses result accordingly if to Carry is 0 or 1



- The adder is faster, instead of delay 4 gates will chain only delay 2 gates for multiplexer
- The speed of the 16-bit adder from the figure will be 10 gate delays
- The speed of the 64-bit adder will be 34 gate delays, which is slightly better than the 64 delay for CLA chaining.

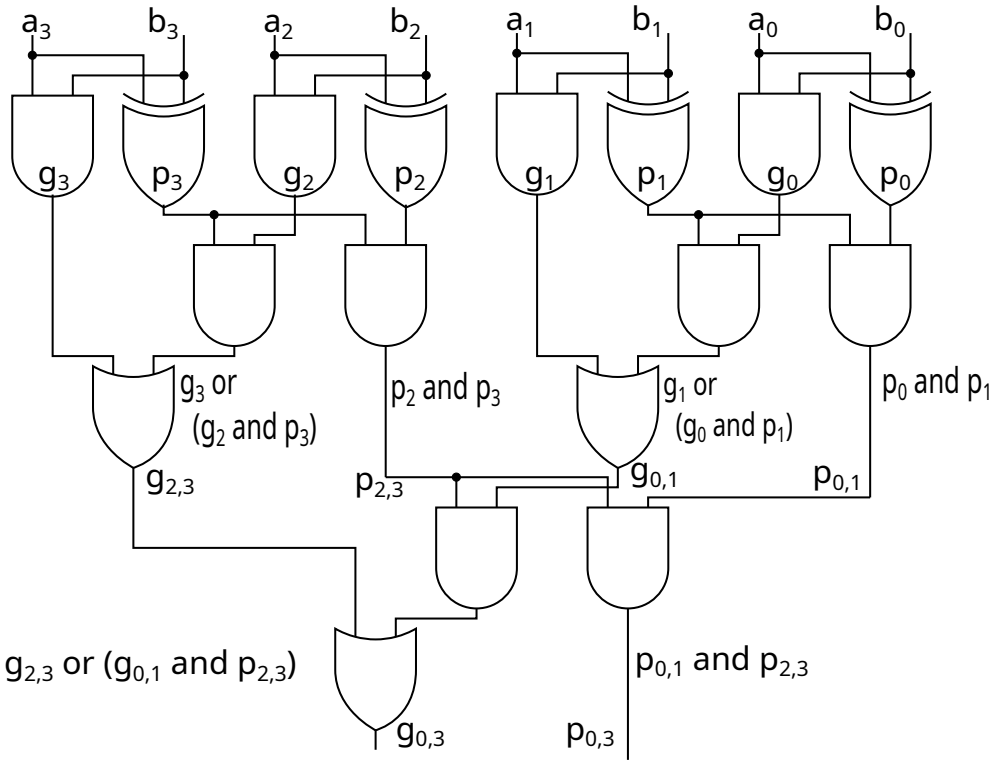
Addition – Carry Lookahead Adder, Block Version

Yet another attempt to speed up adder

- The carry generate and propagate operation can be defined for larger groups of bits:
  - If orders from i to j generate carry, then $G_{i,j} = 1$
  - If orders from i to j propagate carry, then $P_{i,j} = 1$
- Following rules are defined, how to compute $G_{i,k}$ and $P_{i,k}$ based on $G_{i,j}$, $G_{j+1,k}$,$P_{i,j}$, $P_{j+1,k}$
  - $G_{i,k} = G_{j+1,k} \vee (G_{i,j} \wedge P_{j+1,k})$
  - $P_{i,k} = P_{i,j} \wedge P_{j+1,k}$
- The initial values are old known G_i and P_i.
- That is $G_{i,i} = G_i = a_i \wedge b_i$ and $P_{i,i} = P_i = a_i$ xor b_i.

Addition – Carry Lookahead Adder, Block Gen. and Prop.

The computation of the previously defined rules can be realized as:



The time co compute pair $G_{i,k}$ and $P_{i,k}$:

- Delay for 4-bit addends – 5 gate delays
- Delay for 8-bit addends – 7 gate delays
- Delay for 16-bit addends – 9 gate delays
- Delay for 32-bit addends – 11 gate delays
- Delay for 64-bit addends – 13 gate delays

Addition – Carry Lookahead Adder, Example

Evaluation of carry generate and propagate block with use of:

- g_h, p_h – generate and propagate carry in higher (more significant) subblock
- g_l, p_l – generate and propagate carry in lower (less significant) subblock

a	0	0	1	0	1	0	0	1
b	1	0	1	1	0	1	1	1
g = a_i and b_i	0	0	1	0	0	0	0	1
p = a_i xor b_i	1	0	0	1	1	1	1	0
g = g_h or (g_l and p_h)	0		1		0		1	
p = p_h and p_l	0		0		1		0	
g = g_h or (g_l and p_h)	0				1			
p = p_h and p_l	0				0			
g = g_h or (g_l and p_h)	0							
p = p_h and p_l	0							

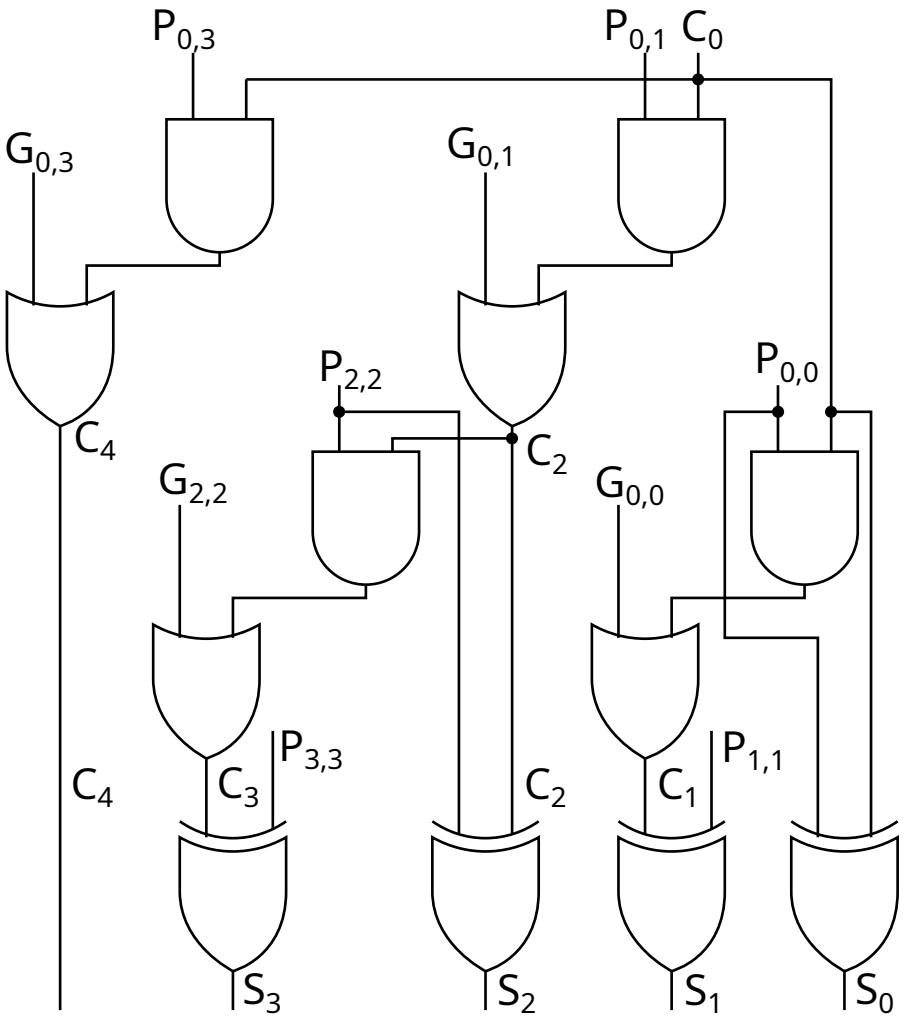
Addition – Carry Lookahead Adder, Hierarchy

It is necessary to compute all Carry C_i, in the case of hypothetical chaining we know C_0, if not then:

- We use tree hierarchy again but in the reversed order of computation $P_{i,j}$ and $G_{i,j}$:
  - If we know C_i, $G_{i,j}$ and $P_{i,j}$
  - then $C_{j+1} = G_{i,j} \vee (C_i \wedge P_{i,j})$
- The evaluation for 4-bit adder will be processed in the next specified order:
  - $C_4 = G_{0,3} \vee (C_0 \wedge P_{0,3})$, $C_2 = G_{0,1} \vee (C_0 \wedge P_{0,1})$
  - $C_3 = G_{2,2} \vee (C_2 \wedge P_{2,2})$, $C_1 = G_{0,0} \vee (C_0 \wedge P_{0,0})$
- The delay for $2\times$ wider inputs (addends) increases by 2 gates delays
- For 64-bit addends, the time required to compute all C_i is 12 gates delays.
- The complete addition for 64-bit inputs takes 26 gates delays.

Addition - Carry Lookahead Adder, Final Sum

The evaluation of C_i then can be seen from:



The time to evaluate $G_{i,k}$ pairs $P_{i,k}$:

- Delay for 4-bit addends is 5 gate delays
- Delay for 8-bit addends is 7 gate delays
- Delay for 64-bit addends is 13 gate delays

Addition – Carry Lookahead Adder, Example

The carry is evaluated from less significant blocks and g, p expressions:

- $C_i = g$ or $(p$ and $C_{i-k})$ – carry prop. or gen. from lower orders

a	0	0	1	0	1	0	0	1
b	1	0	1	1	0	1	1	1
g	0	0	1	0	0	0	0	1
p	1	0	0	1	1	1	1	0
C								
g	0		1		0		1	
p	0		0		1		0	
C								
g	0				1			
p	0				0			
C				$C_4 = 1$ or (0 and $C_0) = 1$				
g	0							
p	0							
C	$C_8 = 0$ or (0 and $C_0) = 0$							

Addition – Carry Lookahead Adder, Example

The carry is evaluated from less significant blocks and g, p expressions:

- $C_i = g$ or $(p$ and $C_{i-k})$ – carry prop. or gen. from lower orders

a	0	0	1	0	1	0	0	1
b	1	0	1	1	0	1	1	1
g	0	0	1	0	0	0	0	1
p	1	0	0	1	1	1	1	0
C								
g	0		1		0		1	
p	0		0		1		0	
C		$C_6 = 1$ or (0 and $C_4) = 1$				$C_2 = 1$ or (0 and $C_0) = 1$		
g	0				1			
p	0				0			
C			$C_4 = 1$ or (0 and $C_0) = 1$					
g	0							
p	0							
C	$C_8 = 0$ or (0 and $C_0) = 0$							

Addition – Carry Lookahead Adder, Example

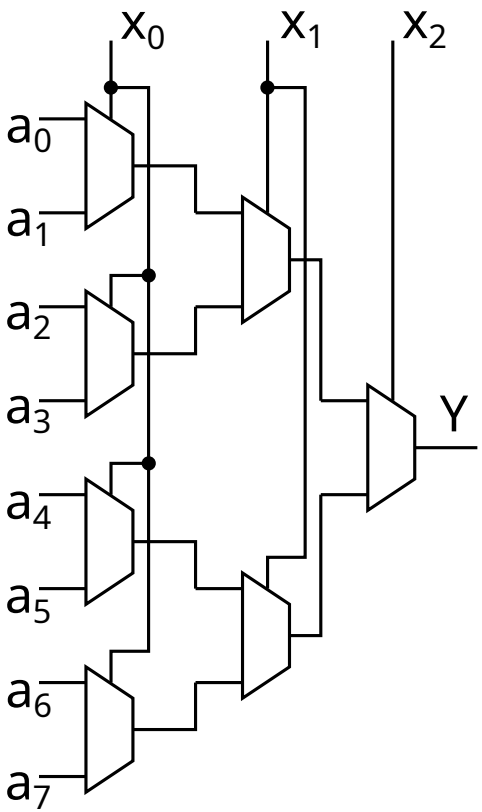
The carry is evaluated from less significant blocks and g, p expressions:

- $C_i = g$ or (p and C_{i-k}) – carry prop. or gen. from lower orders

a	0	0	1	0	1	0	0	1
b	1	0	1	1	0	1	1	1
g	0	0	1	0	0	0	0	1
p	1	0	0	1	1	1	1	0
C	$C_7 = 0$ or (0 and C_6) $= 0$		$C_5 = 0$ or (1 and C_4) $= 1$		$C_3 = 0$ or (1 and C_2) $= 1$		$C_1 = 1$ or (0 and C_0) $= 1$	
g	0		1		0		1	
p	0		0		1		0	
C		$C_6 = 1$ or (0 and C_4) $= 1$				$C_2 = 1$ or (0 and C_0) $= 1$		
g	0				1			
p	0				0			
C			$C_4 = 1$ or (0 and C_0) $= 1$					
g	0							
p	0							
C	$C_8 = 0$ or (0 and C_0) $= 0$							

Multiplexor

Similar approach (divide and conquer - create a tree structure) can be used in the multiplexor implementation:



- According to the three-bit number x the corresponding input a_x is selected for the output Y
- It is not the fastest implementation, but it is clear
- We can also use 4 input multiplex and reduce the number of tree levels.

Bit Shift Operation

- it is denoted in C-language by >> and << operators
  - The operations can be used for multiple by power of two 2 – operation <<; and for division by power of 2 – operation >>
- How to implement shift by k-bits?

k-times rotate/shift by 1 bit
It can be compared to exponentiation algorithm:

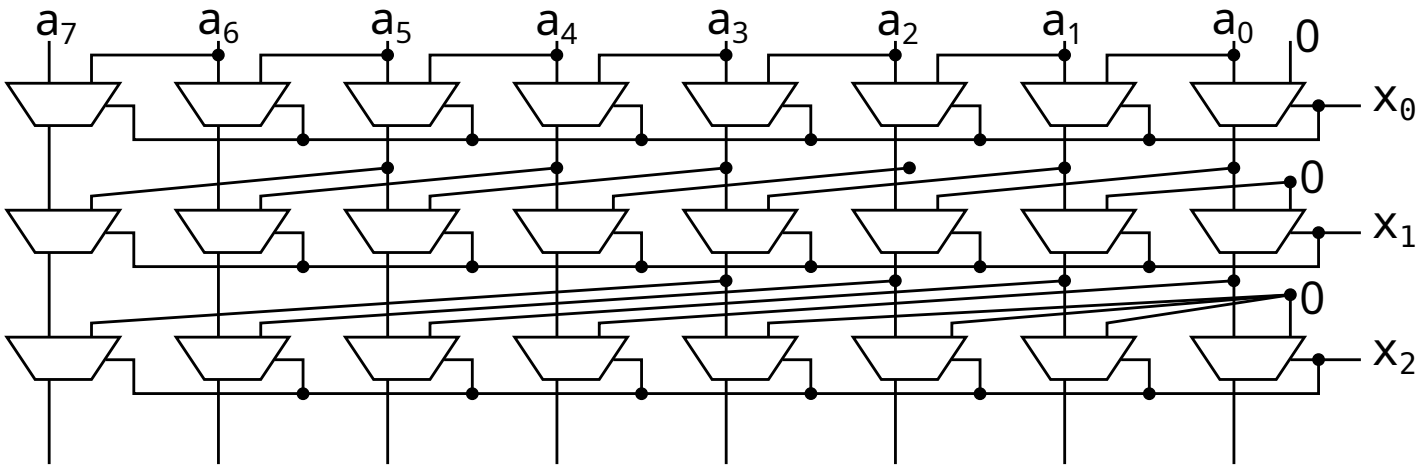
```
double Exp(double a, int k) {
  if (k == 0) return 1;
  return a * Exp(a, k - 1);
}
```

shift by 1, 2, 4, 8, 16 bits
Compare with fast exponentiation:

```
double FastExp(double a, int k) {
  if (k == 0) return 1;
  if (k % 2 == 0) {
    double i = FastExp(a, k / 2);
    return i * i;
  } else {
    return a * FastExp(a, k - 1);
  }
}
```

Barrel Shifter

Shift by 0-7 bits can be composed from shift by 1, 2, 4:



- Multiplexor in each row select ientity operation (do nothing) or shift by fixed number of bits
- Single bit input signals x_2, x_1, x_0 are quivalent to binary repreznetation of number of bits to shift
- Příklad:
  - shift by 5 bits is realized by serial combination of shift by 1 bit and shift by 4 bits ($x_2 = 1$, $x_1 = 0$, $x_0 = 1$)
  - shift by 3 bits is realized by serial combination of shift by 1 bit and shift by 2 bits ($x_2 = 0$, $x_1 = 1$, $x_0 = 1$)

Quiz

How many layers (row) Barrel shifter must have for 64 bit number, i.e. for rotations from 0 to 63? (For rotations from 0 to 7 it was 3 layers)

A 4

B 6

C 16

D 64

Feedback Quiz

How much did you understand today?

A All without a problem.

B Almost everything.

C Almost nothing.

D Absolutely nothing.