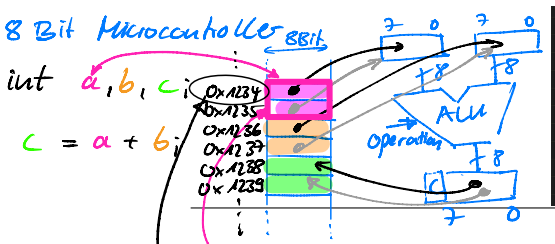
Data Structures and Algorithms

Michael Zwick

14. März 2024

Inhaltsverzeichnis




```
char c;     // 8 bit =>
            // 10000000 => -128
            // ...
            // 11111101 => -3
            // 11111110 => -2
            // 11111111 => -1
            // 00000000 => 0
            // 00000001 => 1
            // 00000010 => 2
            // 01111111 => 127
            // 10000000 => 128 @ unsigned char
```

```
27      printf("sizeof(c): %ld\n", sizeof(c) );
28      printf("sizeof(...): %ld\n", sizeof(int) );
29
30      c = 1;
31
32      printf("Value of c: %d\n", c);
33      printf("Address of c: %p\n", &c);
34
sizeof(c): 1
sizeof(...): 4
Value of c: 1
Address of c: 0x16d6973ff
→  Code
```

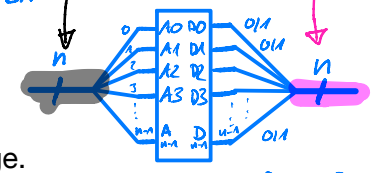
1 Introduction

YEAR 01	Fundamentals	Fundamentals	Break + Fundamentals
	• Engineering Mathematics 1 • Programming • Engineering Physics • ~~Digital Electronics~~ • Engineering Mathematics 2	• Circuit Theory • Discrete Mathematics • Data Structures & Algorithms • Electricity & Magnetism • Analogue Electronics	• Engineering Mathematics 3 • Circuit Design Fundamentals • Technical Communication* • Digital Electronics

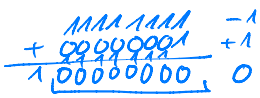
2^6 different things can be represented with 6Bit

Programming

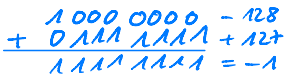
Arrays, Pointers, Addresses



- Comprehend basic terminology used in C programming language.
- Plan, implement, test and debug C programs.



- Apply different variables, arithmetic and logical expressions, control selections, and repetitions in C programs.



- Implement functions, arrays, and pointers in C programs.
- Use string and character processing, formatted input/output, file processing, and data structures in C programs.
- Design C programs for performing simple tasks using Arduino microcontroller.

Digital Electronics

- Number systems, binary codes, digital arithmetic, logic gates: AND, OR, NOT, NAND, NOR, XOR, and families. Boolean algebra, De Morgan's Theorem, combination logic circuits, Sum-of-Products, Product-of-Sums. Minimization using Karnaugh Maps
- Sequential circuits. Flip-flops, multi-vibrators, counters, shift registers, devices for arithmetic operations, multiplexers, and de-multiplexers
- Introduction to relevant applications in programmable logic devices, microprocessors, microcontrollers, etc. Digital troubleshooting and instrumentation

1.1 What we will cover

```
00000000: 23 69 6e 63 6c 75 64 65 20 3c 73 74 64 69 6f 2e
00000010: 68 3e 0a 0a 69 6e 74 20 6d 61 69 6e 28 29 0a 7b
00000020: 0a 09 69 6e 74 20 61 2c 20 62 2c 20 63 3b 0a 0a
00000030: 09 61 20 3d 20 33 3b 0a 09 62 20 3d 20 35 3b 0a
00000040: 0a 09 63 20 3d 20 61 20 2b 20 62 3b 0a 0a 7d
```

? ? ?

Data Structures and Algorithms

- How to store data efficiently?
- How to search data efficiently?
- What can be programmed at all?
- How long does it take a program to execute?

Alan Turing

```c
#include <stdio.h>

int main()
{
    char name[20];

    printf("Name: ");
    scanf("%19[^\n]", name);
    printf("Hello %s\n", name);

    return 0;
}
```

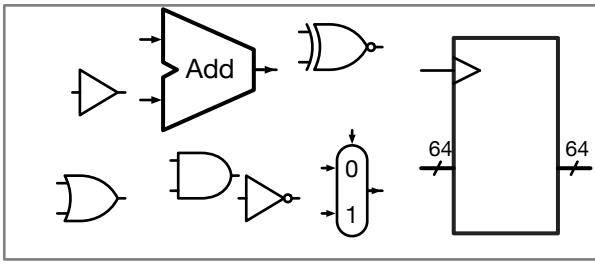
Programming

- C Syntax
- How to write simple C programs
- Arrays and pointers
- Input/output

? ? ?

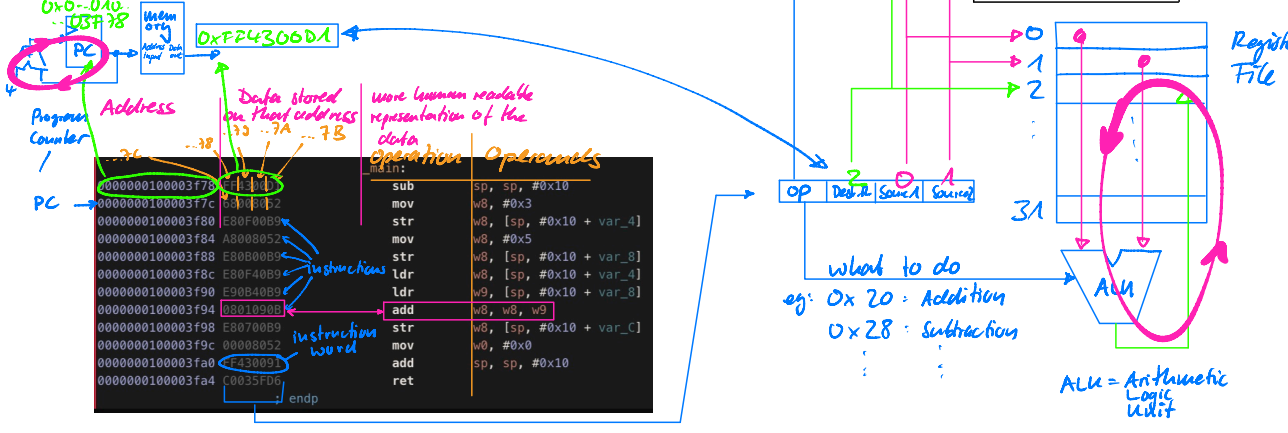
Data Structures and Algorithms

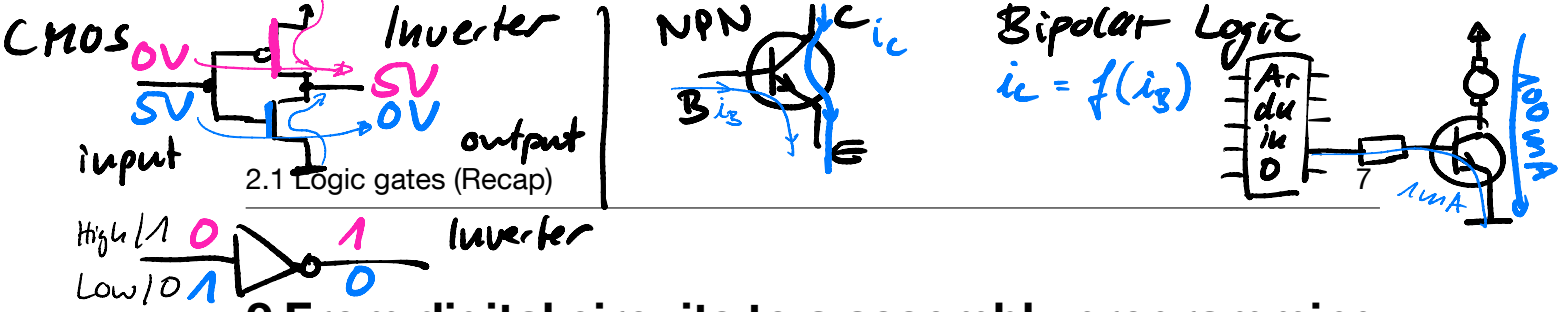
- How is a C program mapped to machine language?
- How does machine language look like?
- Processor architecture and instructions
- How to calculate execution time from a program run?



Digital Electronics

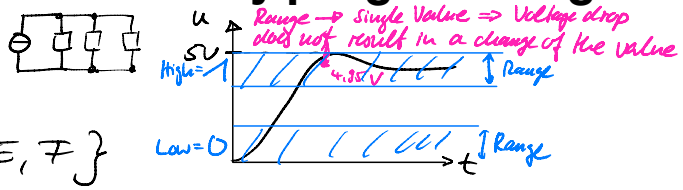
- Number systems
- Logic gates
- Boolean Algebra, minimization
- Flip-flops, registers





2 From digital circuits to a assembly programming

Binary: Symbols {0,1}
Dec: {0,1,2,3,4,5,6,7,8,9}
Hex: {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}



Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

C-File opened in a text editor:

```c
#include <stdio.h>

int main()
{
    printf("Hello.\n");
    return 0;
}
```

0010 0011

C-File opened in a hex editor:

```
00   23 69 6E 63 | 6C 75 64 65 | 20 3C 73 74 | 64 69 6F 2E   #include <stdio.
10   68 3E 0A 0A | 69 6E 74 20 | 6D 61 69 6E | 28 29 0A 7B   h>..int main().{
20   0A 09 70 72 | 69 6E 74 66 | 28 22 48 65 | 6C 6C 6F 2E   ..printf("Hello.
30   5C 6E 22 29 | 3B 0A 09 72 | 65 74 75 72 | 6E 20 30 3B   \n");..return 0;
40   0A 7D 0A 0A |             |             |               .}..
```

gcc -o p p.c

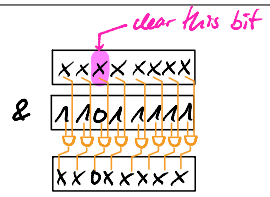
Executable binary file opened in a hex editor (excerpt):

```
0000   CF FA ED FE | 07 00 00 01 | 03 00 00 80 | 02 00 00 00 | 0E 00 00 00 | E8 02 00 00
0018   85 00 20 00 | 00 00 00 00 | 19 00 00 00 | 48 00 00 00 | 5F 5F 50 41 | 47 45 5A 45
0030   52 4F 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 01 00 00 00
0048   00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00
0060   00 00 00 00 | 00 00 00 00 | 19 00 00 00 | E8 00 00 00 | 5F 5F 54 45 | 58 54 00 00
0078   00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 01 00 00 00 | 00 10 00 00 | 00 00 00 00
0090   00 00 00 00 | 00 00 00 00 | 00 10 00 00 | 00 00 00 00 | 05 00 00 00 | 05 00 00 00
```

Executable binary file opened in a disassembler program (excerpt):

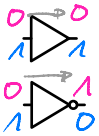
```
                    _main:
0000000100000f60        push    rbp
0000000100000f61        mov     rbp, rsp
0000000100000f64        sub     rsp, 0x10
0000000100000f68        mov     dword [rbp+var_4], 0x0
0000000100000f6f        lea     rdi, qword [aHellon]
0000000100000f76        mov     al, 0x0
0000000100000f78        call    imp___stubs__printf
0000000100000f7d        xor     ecx, ecx
0000000100000f7f        mov     dword [rbp+var_8], eax
0000000100000f82        mov     eax, ecx
0000000100000f84        add     rsp, 0x10
0000000100000f88        pop     rbp
0000000100000f89        ret
                    ; endp
```

2.1 Logic gates (Recap)



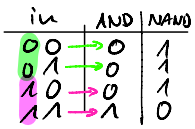

```
#include <stdio.h>

int main()
{
    unsigned char a, b, c;

    a = 0xFF;
    b = 0xDF;

    c = a & b;

    printf("c: %#x\n", c);
}
```

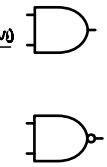
```
→ Code gcc -o p p.c
→ Code ./p
c: 0xdf
→ Code
```



Driver; forwards the input logic level to its output

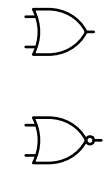
Inverter; forwards the inverse of its input logic level to the output





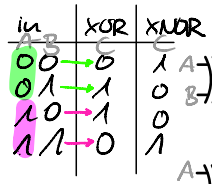
AND; provides a 1 at its output, if both inputs are 1, otherwise 0; there might be more than 2 inputs

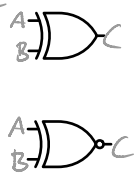
NAND; provides a 0 at its output, if both inputs are 1, otherwise 1; there might be more than 2 inputs



OR; provides a 1 at its output, if one ore more inputs are 1; otherwise the output is 0; there might be more than 2 inputs

NOR; provides a 1 at its output if no input is at 1 level; otherwise, the output will be 0, there might be more than 2 inputs





XOR, exclusive OR; provides a 1 at its output, if the inputs are different; the output will be 0 if the inputs have equal values; only 2 inputs

XNOR; provides a 1 at its output, if the inputs show equal level; otherwise, the output is 0; only 2 inputs

a) Provide the truth table for the logic gates. S.O.

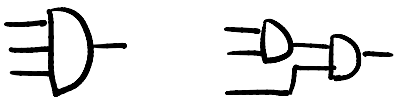
Assume integer variables a, b, c.
Assign integer variable c the value of a, but with bit no. 17 inverted.

```
#include <stdio.h>

int main()
{
    int a, b, c;
    a = 0x12345678;
    b = 1 << 17;
    c = a ^ b;
    printf("b: %#x\n", b);
    printf("c: %#x\n", c);

}
```

```
→ Code gcc -o p p.c
→ Code ./p
b: 0x20000
c: 0x12365678
→ Code
```

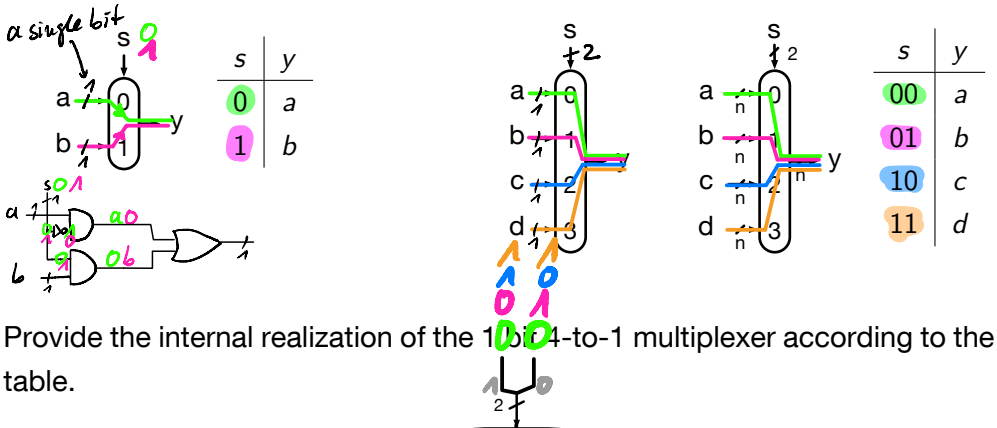
4: 0100
6: 0110

2.2 Combining basic logic gates (Recap)

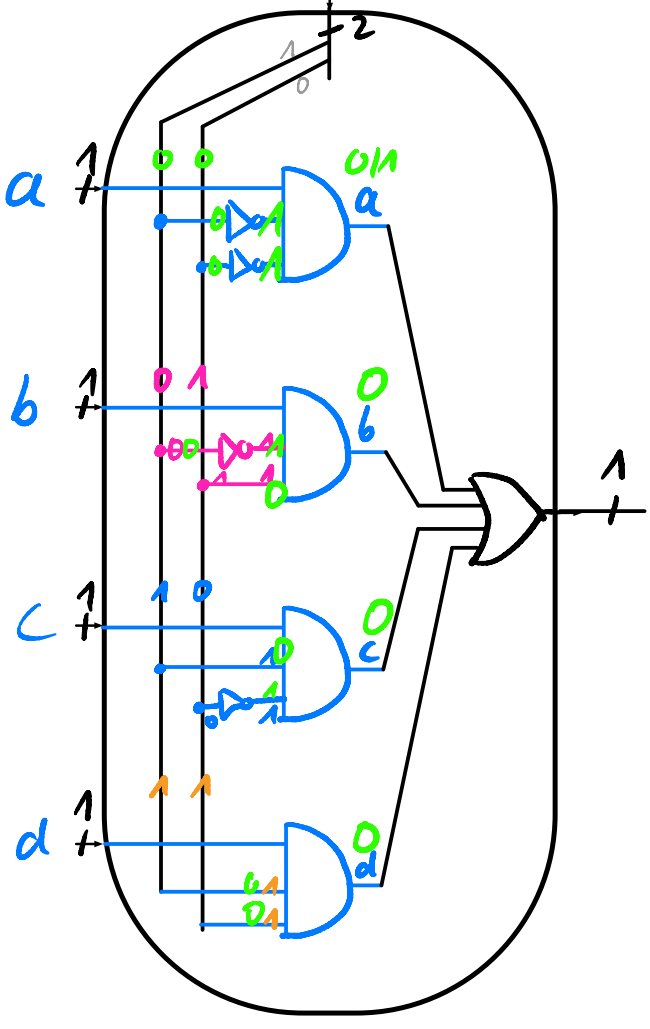


Multiplexer

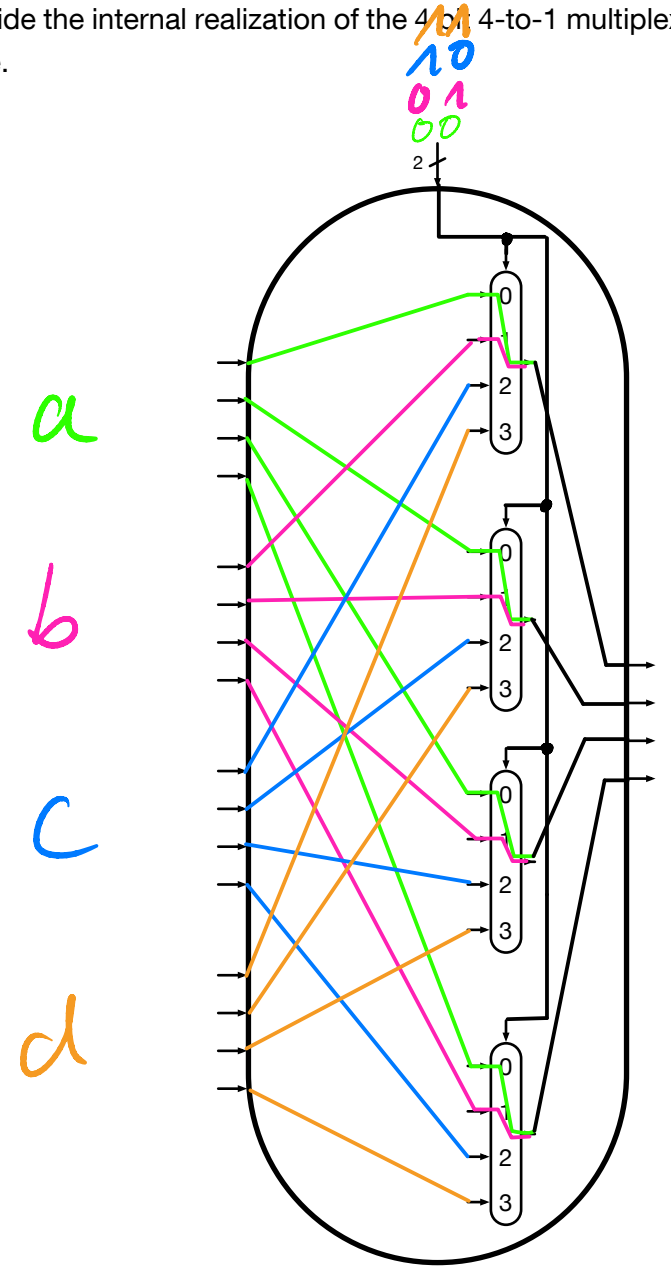
A multiplexer forwards one of several inputs to its output, depending on a control line.



a) Provide the internal realization of the 1-bit 4-to-1 multiplexer according to the truth table.

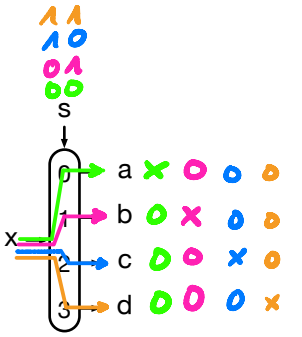


b) Provide the internal realization of the 4 bit 4-to-1 multiplexer according to the truth table.



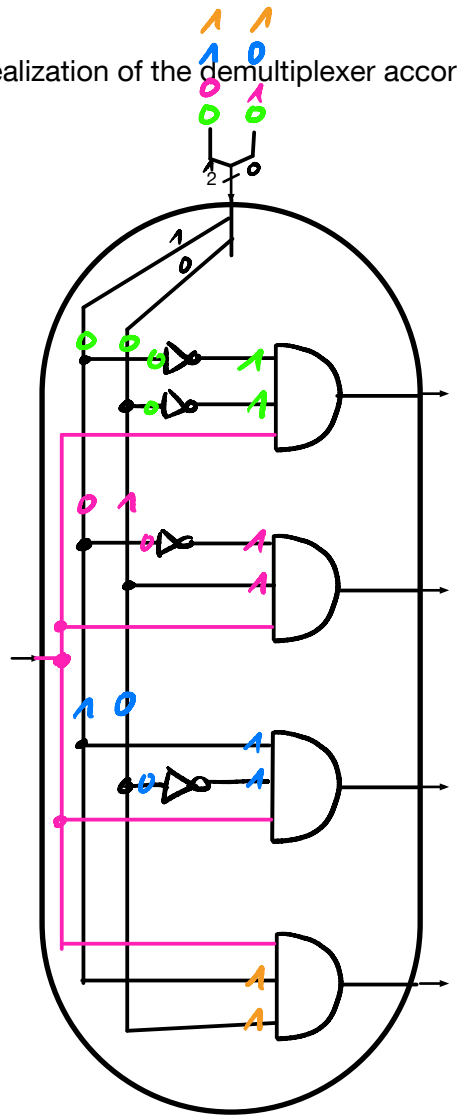
Demultiplexer

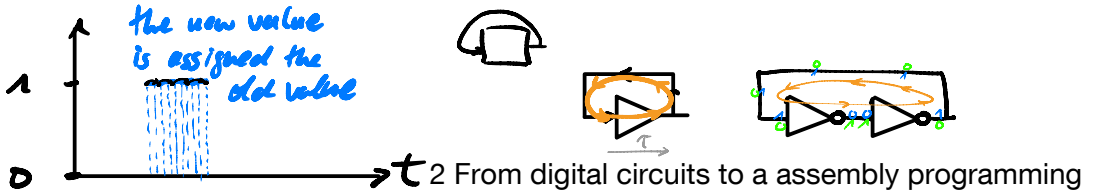
A demultiplexer provides its input to one of several outputs. The output is selected by a control line.



s	a	b	c	d
00	x	0	0	0
01	0	x	0	0
10	0	0	x	0
11	0	0	0	x

a) Provide the internal realization of the demultiplexer according to the truth table.

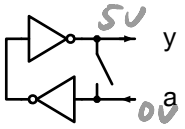




Flipflops and registers

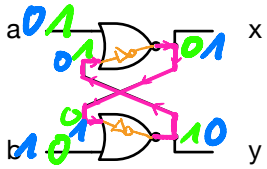
Building a feedback loop of two registers allows storing a single bit.

In order to be able to change the stored bit, you have to be able to disconnect one inverter output from the input.



Applying logic gates with more than a single input, you are able to use one input for the feedback loop and the other to set a value.

1. We need an inverter
2. We need to be able to put a specific value into the feedback loop



a	b	y	
0	0	1	} Inverter
0	1	0	
1	0	0	} if a is 1, then output y will be 0, independently of input b
1	1	0	

The following table shows how the input values to the NOR-feedback loop changes over time. At time t_1, input a changes from 0 to 1; at time t_2, a changes from 1 to 0. τ is the amount of time it takes a signal to pass any logic gate.

a) Provide values x and y with subject to the input values.

t	0	t_1	$t_1 + \tau$	$t_1 + 2\tau$	$t_1 + 3\tau$	...	t_2	$t_2 + \tau$	...
a	0	1	1	1	1	...	0	0	...
b	0	0	0	0	0	...	0	0	...
x	?	?	0	0	0	...	0	0	...
y	?	?	?	1	1	...	1	1	...

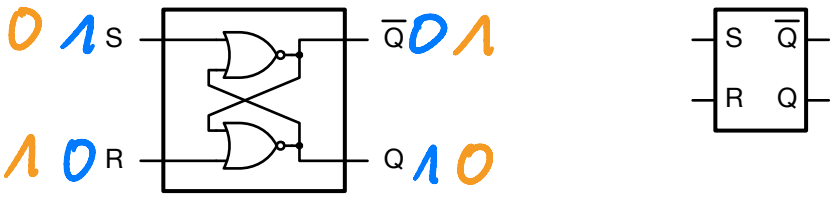
At time t_3, input b changes from 0 to 1; at time t_4 it changes bach from 1 to 0.

b) Provide values x and y with subject to the input values.

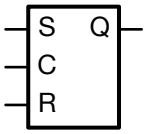
t	t_3	$t_3 + \tau$	$t_3 + 2\tau$	$t_3 + 3\tau$	...	t_4	$t_4 + \tau$	...
a	0	0	0	0	...	0	0	...
b	1	1	1	1	...	0	0	...
x	0	0	1	1	...	1	1	...
y	1	0	0	0	...	0	0	...

RS-Flipflop

The feedback NOR-circuit is known as asynchronous RS-Flipflop (RS = Reset/Set).



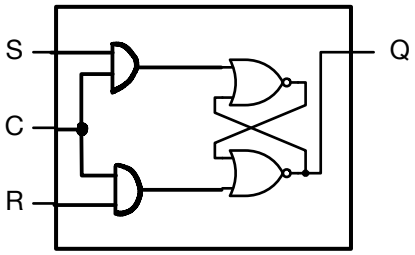
Asynchronous means that changes on the input lines will immediately be followed. Often, you would rather like to have a clk signal that enables or disables the input lines.



- If the clock signal is 0, then changes on the input lines S and R will not have any effect.
- I the clock signal is 1, then the circuit will follow changes on the input lines S and R.

As the input will be followed depending on the *state* of the clk line, the circuit is called *state* triggered or *level* triggered.

c) Provide logic gates and connection lines to create a *state triggered* RS-Flipflop.

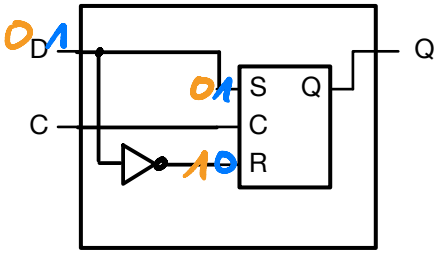


D-Flipflop (level triggered)

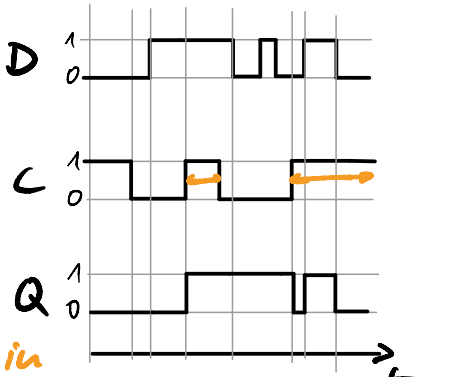
With an RS-Flipflop, you can set both lines R and S to high level, what does not make sense. You cannot set and reset simultaneously. To overcome this limitation, inputs R and S can be combined to a data input D.

- D = 1 means that S = 1 and R = 0.
- D = 0 means that S = 0 and R = 1.

d) Provide a circuit to make the RS-Flipflop become a D-Flipflop.



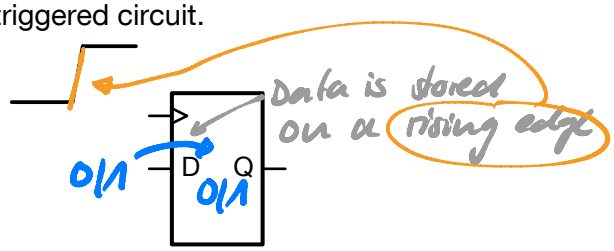
Q follows D if C=1
No change on Q if C=0



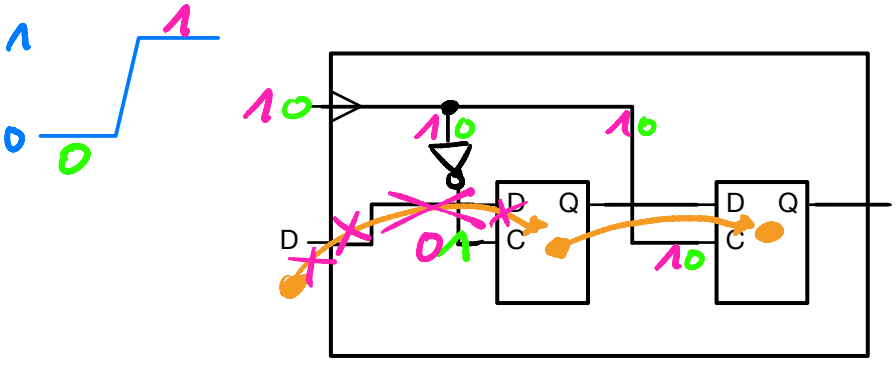
t should be small, in order to take over inputs at a specific moment

D-Flipflop (edge triggered)

Often, a circuit should only respond to input signals at a specific point in time. With an *edge triggered* D-Flipflop, inputs are stored in the circuit only on a rising edge of the clk signal, i.e. when the clock signal changes from 0 to 1. A small triangle on the clk input indicates that it's an edge triggered circuit.

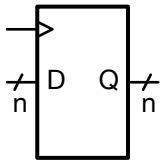


e) Complete the circuit to build an edge triggered D-Flipflop from two level triggered D-Flipflops.

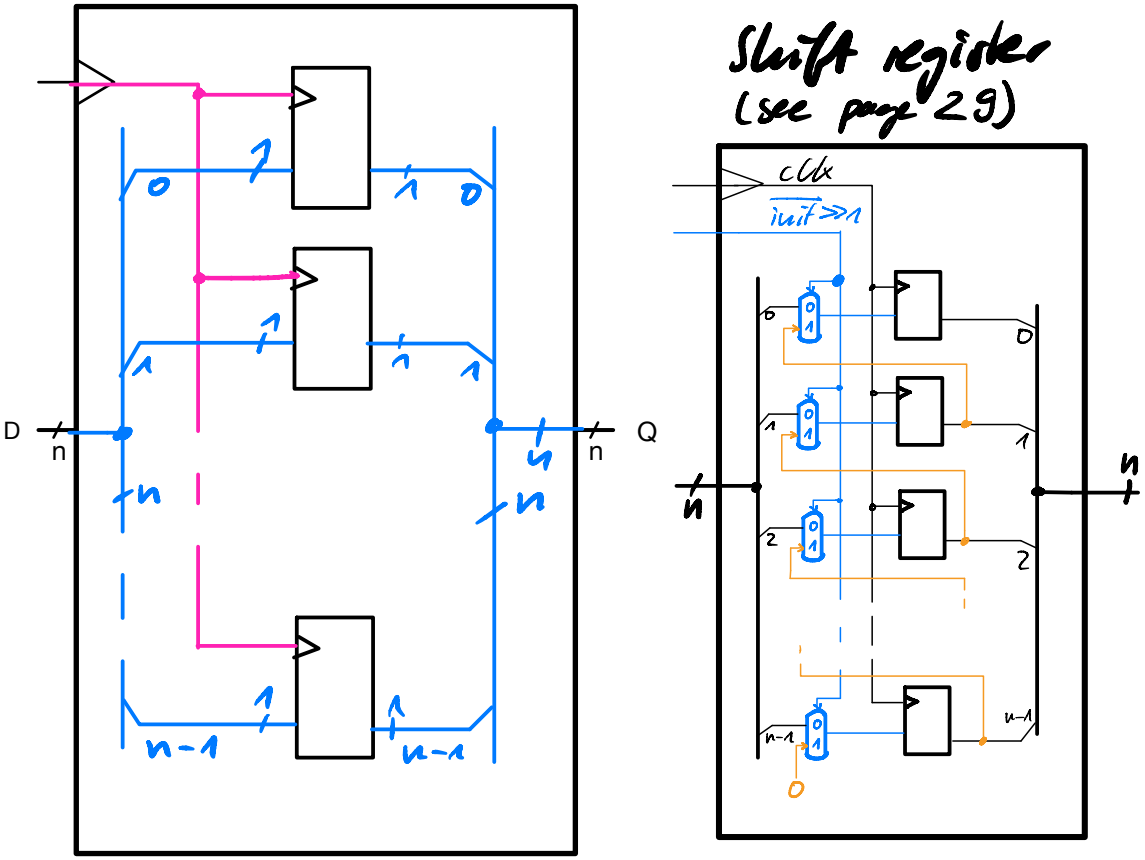


D-Register

In order to not only store a single bit, but multiple bits, you combine multiple single bit D-Flipflops to form a multi-bit edge triggered D-Flipflop, also called a D-Register.



f) Combine a set of 1 bit edge triggered D-Flipflops to implement an n bit D-Register.



Base 10: $3 + 2 = 5$ $\quad 08 + 03 = 11$ Base 2: $0 + 1 = 1$ $\quad 01 + 01 = 10$

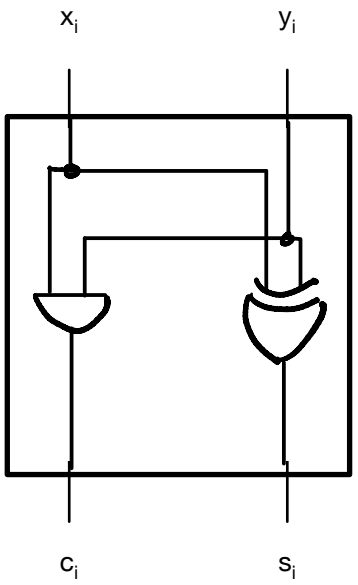
Half adder

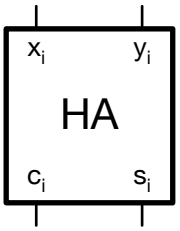
A half adder adds two single bits x and y of a dataword to the *sum* and the *carry*.

a) Provide a truth table for the half adder.

x	y	carry (AND)	sum (XOR)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

b) Provide a circuit for the half adder.

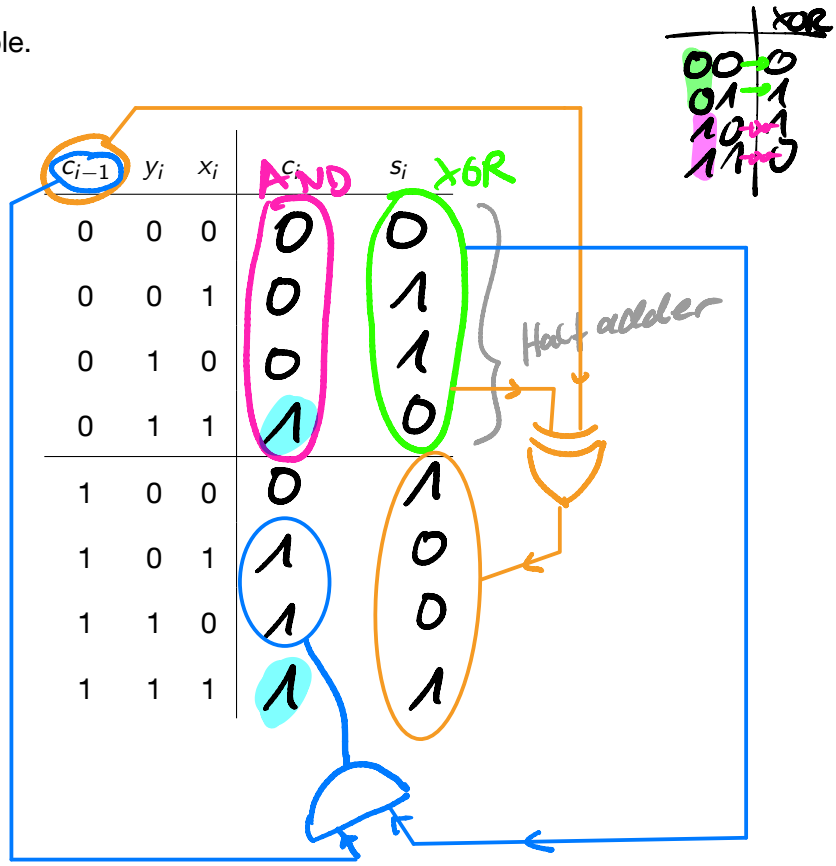




Full adder

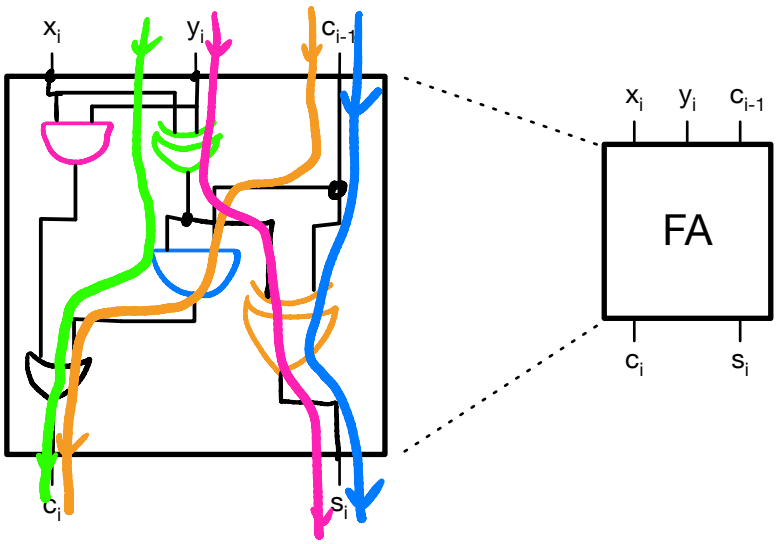
A full adder adds three single bits x, y and c_{i-1} to a *sum* s and a *carry out* c_i.

a) Provide the truth table.



c_{i-1}	y_i	x_i	c_i	s_i
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

b) Provide a circuit for the full adder.



Let τ be the time it takes a signal to pass any logic gate.

c) How many gate delays τ does it take the full adder to show the right output signal?

$c_{i-1} \mapsto s : \tau$

$c_{i-1} \mapsto c_i : 2\tau$

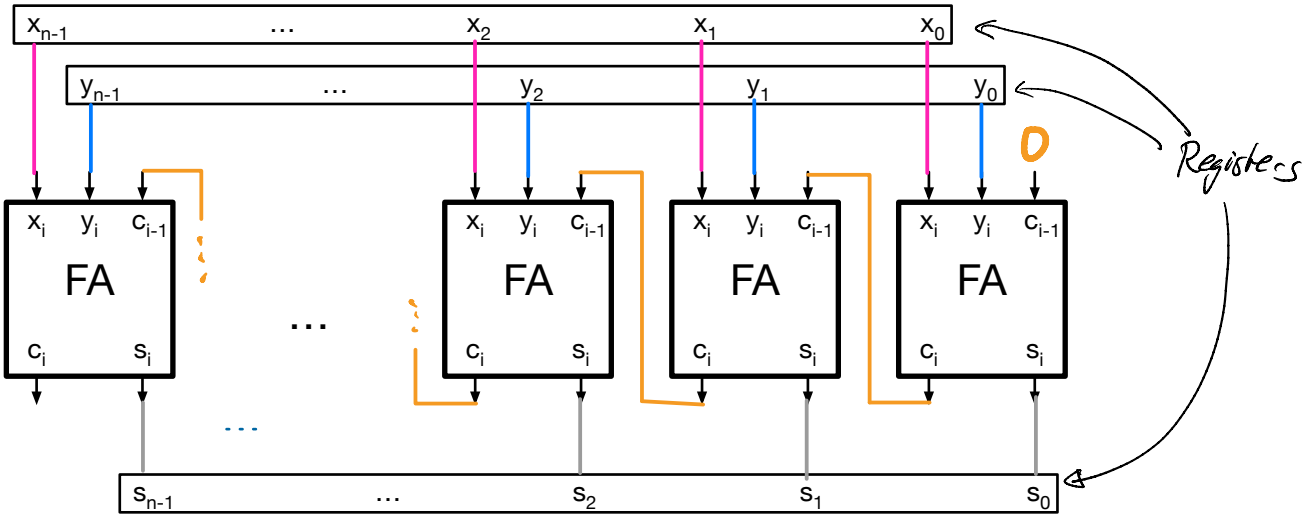
$x_i / y_i \mapsto c_i : 3\tau$

$x_i / y_i \mapsto s_i : 2\tau$

Ripple carry adder

A ripple carry adder combines multiple full adders.

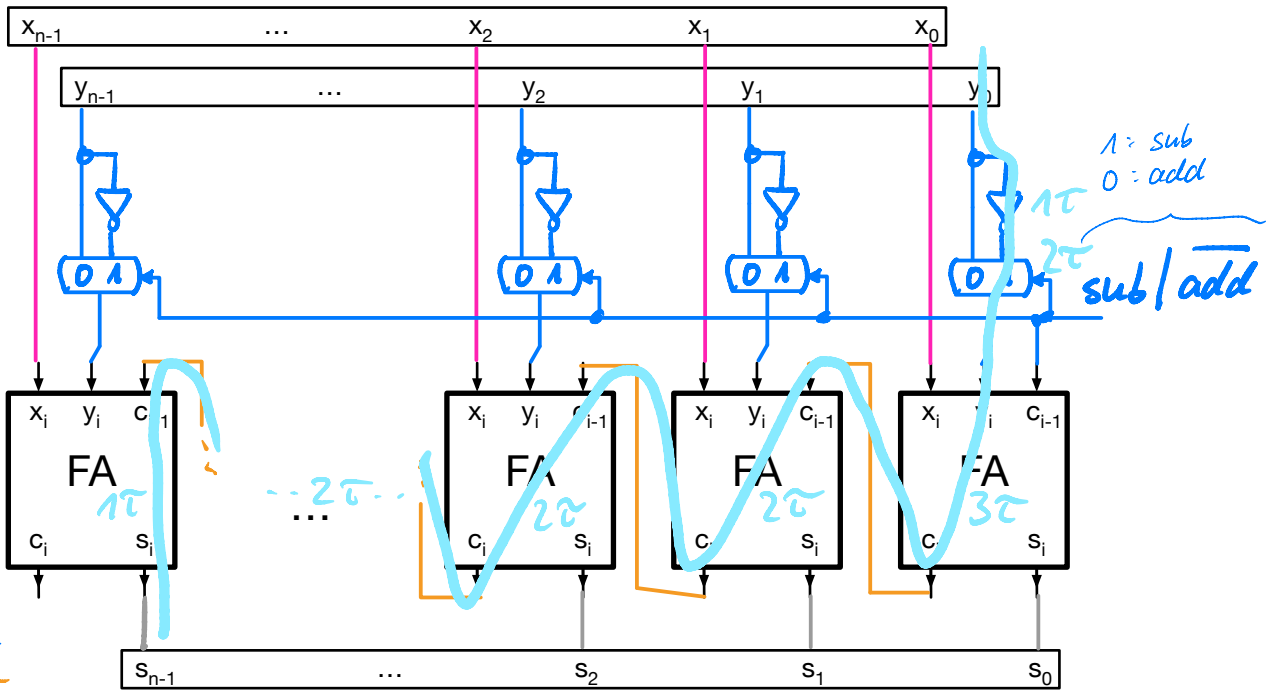
a) Add lines to connect the full adders to a ripple carry adder.



$8 - 5 = 8 + (-5)$

To allow a ripple carry adder to subtract a number, you can add the negative in 2's complement.

b) Add lines to connect the full adders and additional hardware elements to build a ripple carry adder/subtractor.



c) Provide the critical path. How many τ does it take the circuit to do a subtraction of two n bit numbers? How many τ in case n = 64?

$(1 + 2 + 3 + (n-2) \cdot 2 + 1) \cdot \tau$

$7\tau + 2n\tau - 4\tau = 2n\tau + 3\tau$

$n = 64 : (2 \cdot 64 + 3)\tau = 131\tau$

Combinational Multiplier

Convention: Multiplier · Multplicand = Product

With multiplier x and multiplicand y you do a multiplication as follows:

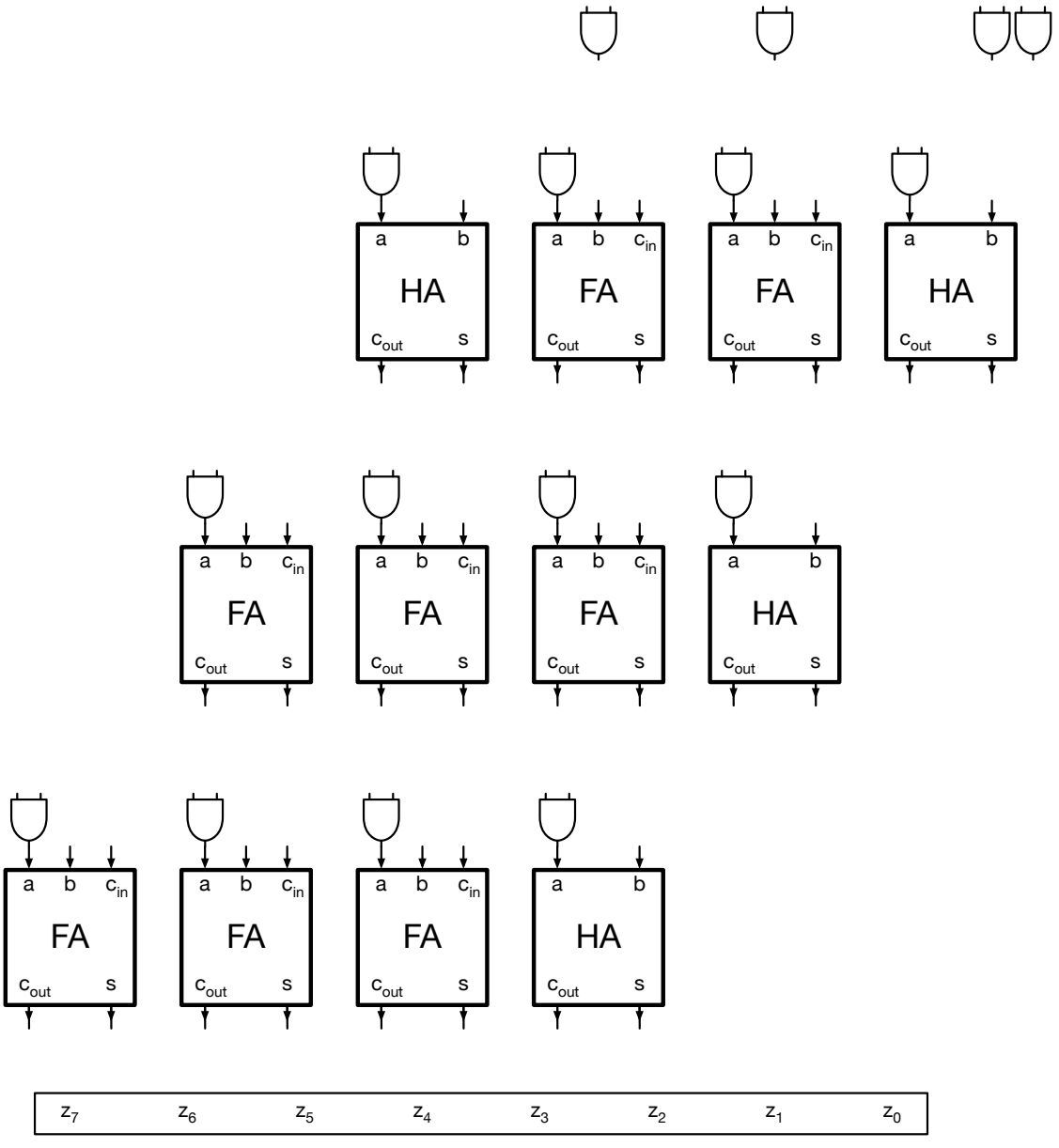
	x_3	x_2	x_1	x_0	· y_3	y_2	y_1	y_0
					$x_0 \cdot y_3$	$x_0 \cdot y_2$	$x_0 \cdot y_1$	$x_0 \cdot y_0$
+				$x_1 \cdot y_3$	$x_1 \cdot y_2$	$x_1 \cdot y_1$	$x_1 \cdot y_0$	
+			$x_2 \cdot y_3$	$x_2 \cdot y_2$	$x_2 \cdot y_1$	$x_2 \cdot y_0$		
+		$x_3 \cdot y_3$	$x_3 \cdot y_2$	$x_3 \cdot y_1$	$x_3 \cdot y_0$			
=	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0

a) Do a multiplication 123 · 17 using 4 digit numbers.

b) Convert decimal numbers 12 and 5 to 4 bit binary unsigned numbers and then do the multiplication.

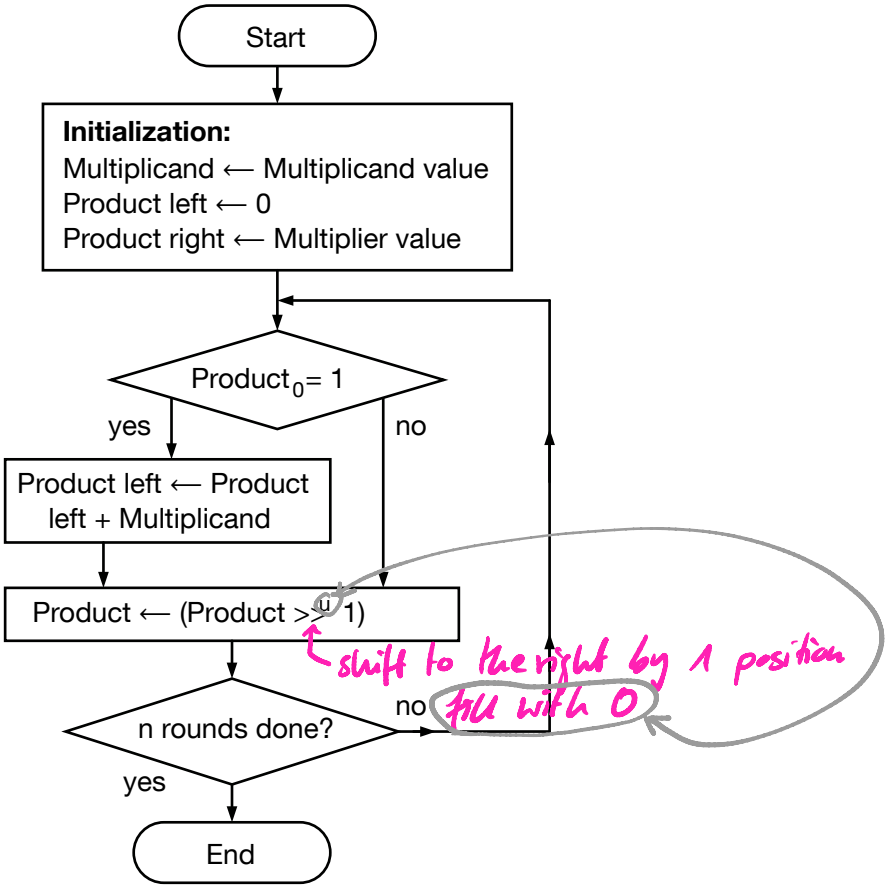
c) In the following circuit, provide lines to implement unsigned multiplication.

	x_3	x_2	x_1	x_0	$\cdot$	y_3	y_2	y_1	y_0
						$x_0 \cdot y_3$	$x_0 \cdot y_2$	$x_0 \cdot y_1$	$x_0 \cdot y_0$
$+$					$x_1 \cdot y_3$	$x_1 \cdot y_2$	$x_1 \cdot y_1$	$x_1 \cdot y_0$	
$+$				$x_2 \cdot y_3$	$x_2 \cdot y_2$	$x_2 \cdot y_1$	$x_2 \cdot y_0$		
$+$			$x_3 \cdot y_3$	$x_3 \cdot y_2$	$x_3 \cdot y_1$	$x_3 \cdot y_0$			
$=$	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0	

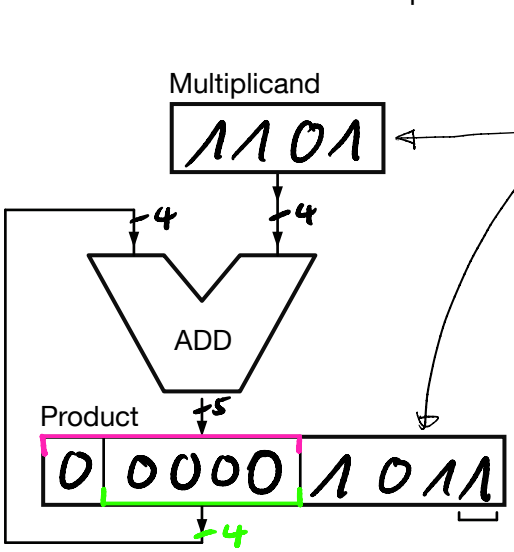


Sequential Multiplier

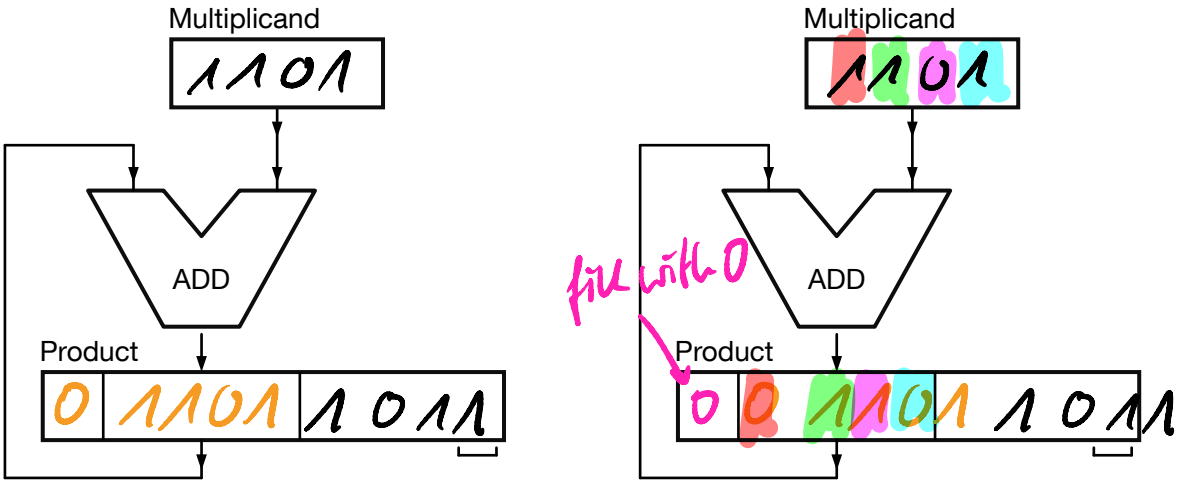
See the following multiplication flow chart to control a multiplication hardware.



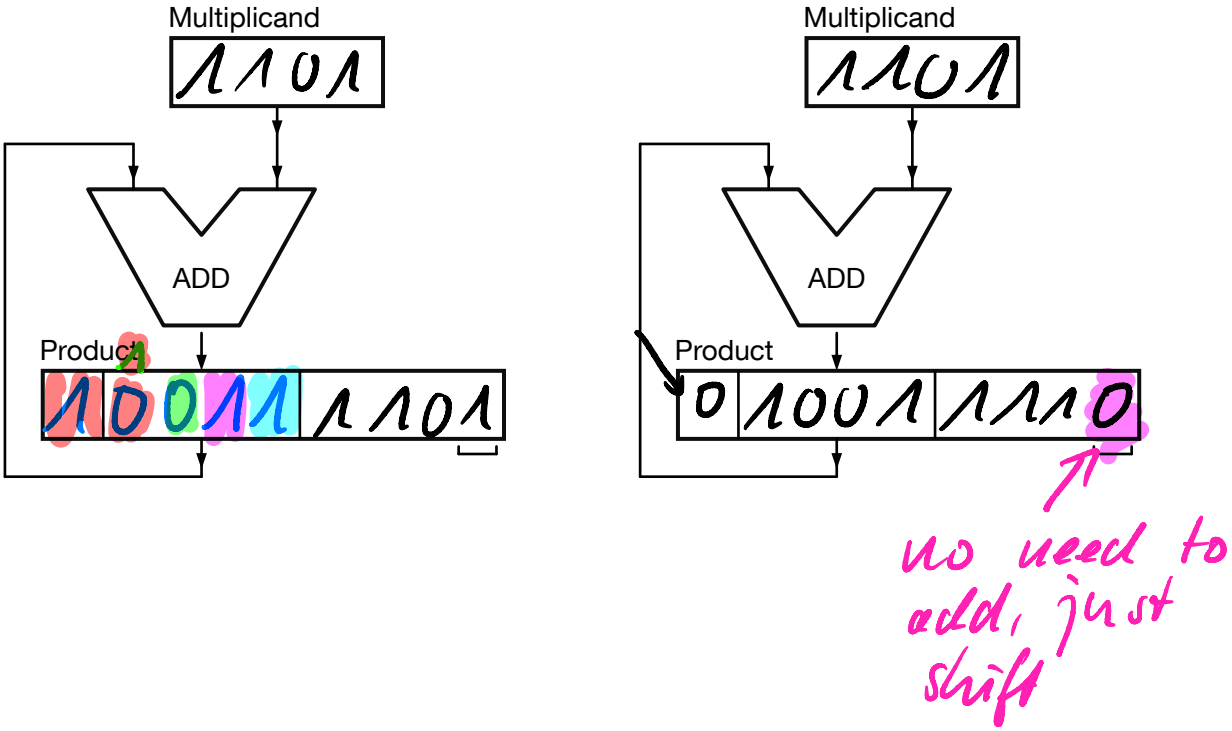
a) Provide the register contents for the initialization step for calculation 11 · 13 = 143.



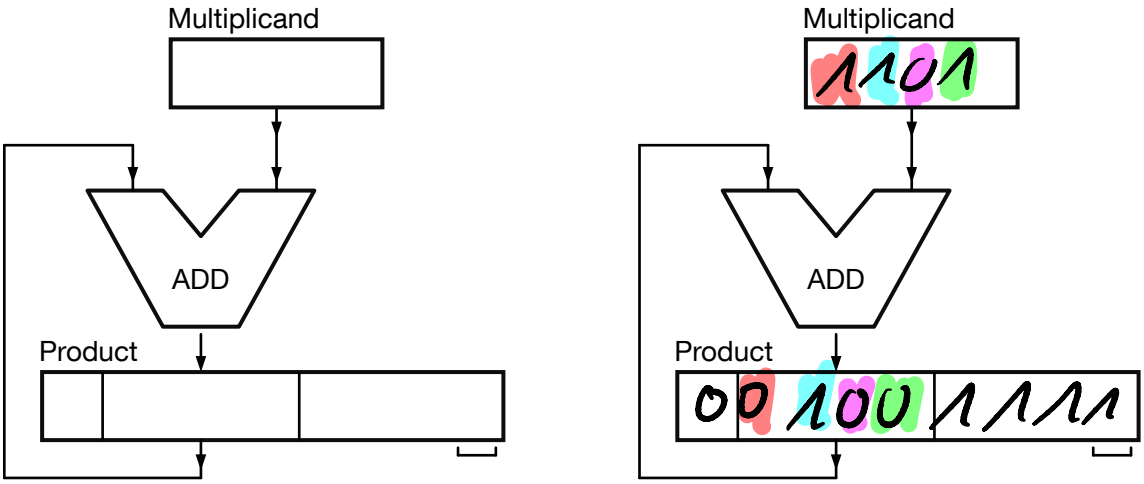
b) In the following figure, provide the data stored in the registers after the first round.



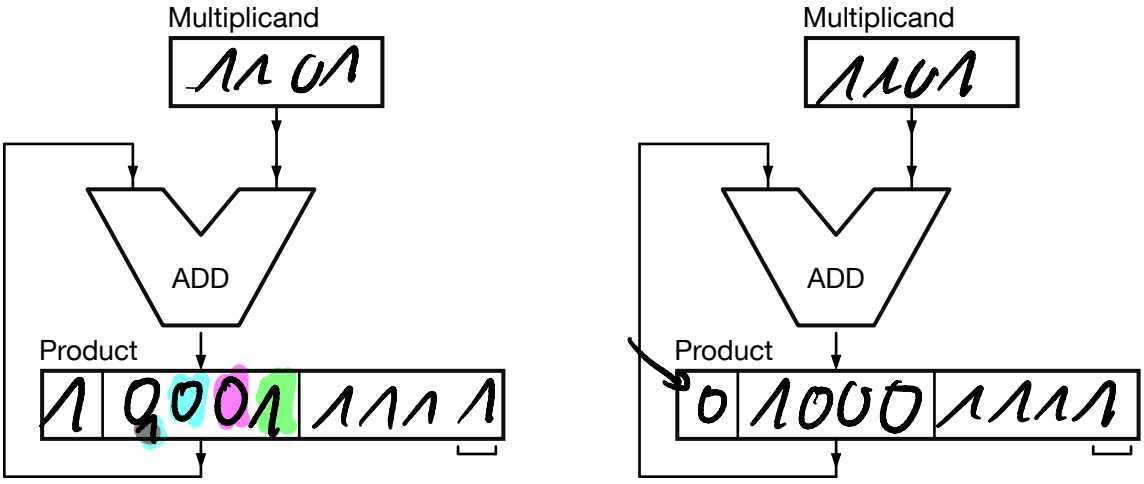
c) In the following figure, provide the data stored in the registers after round two.



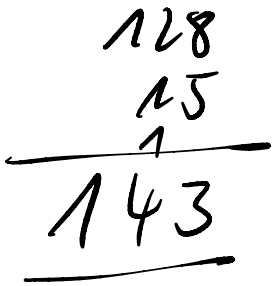
d) In the following figure, provide the data stored in the registers after round three.



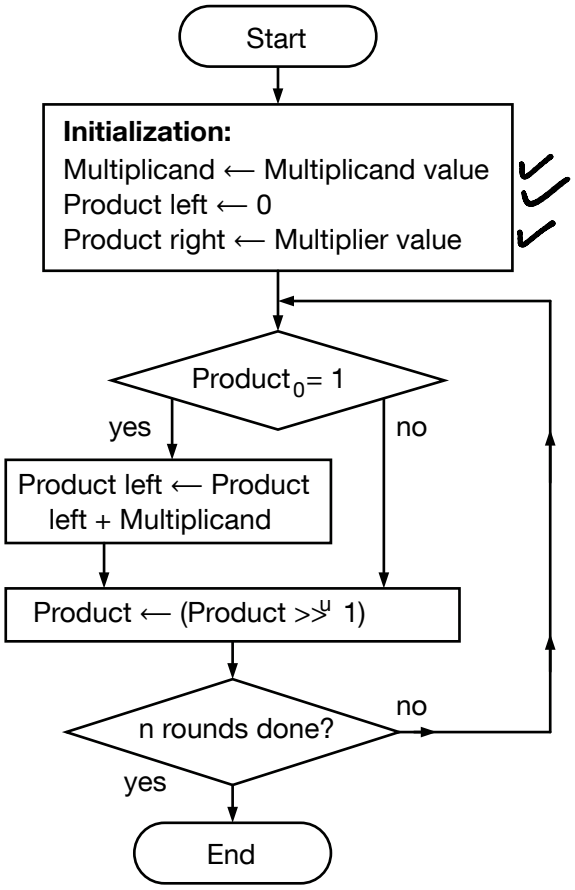
e) In the following figure, provide the data stored in the registers after the forth round.



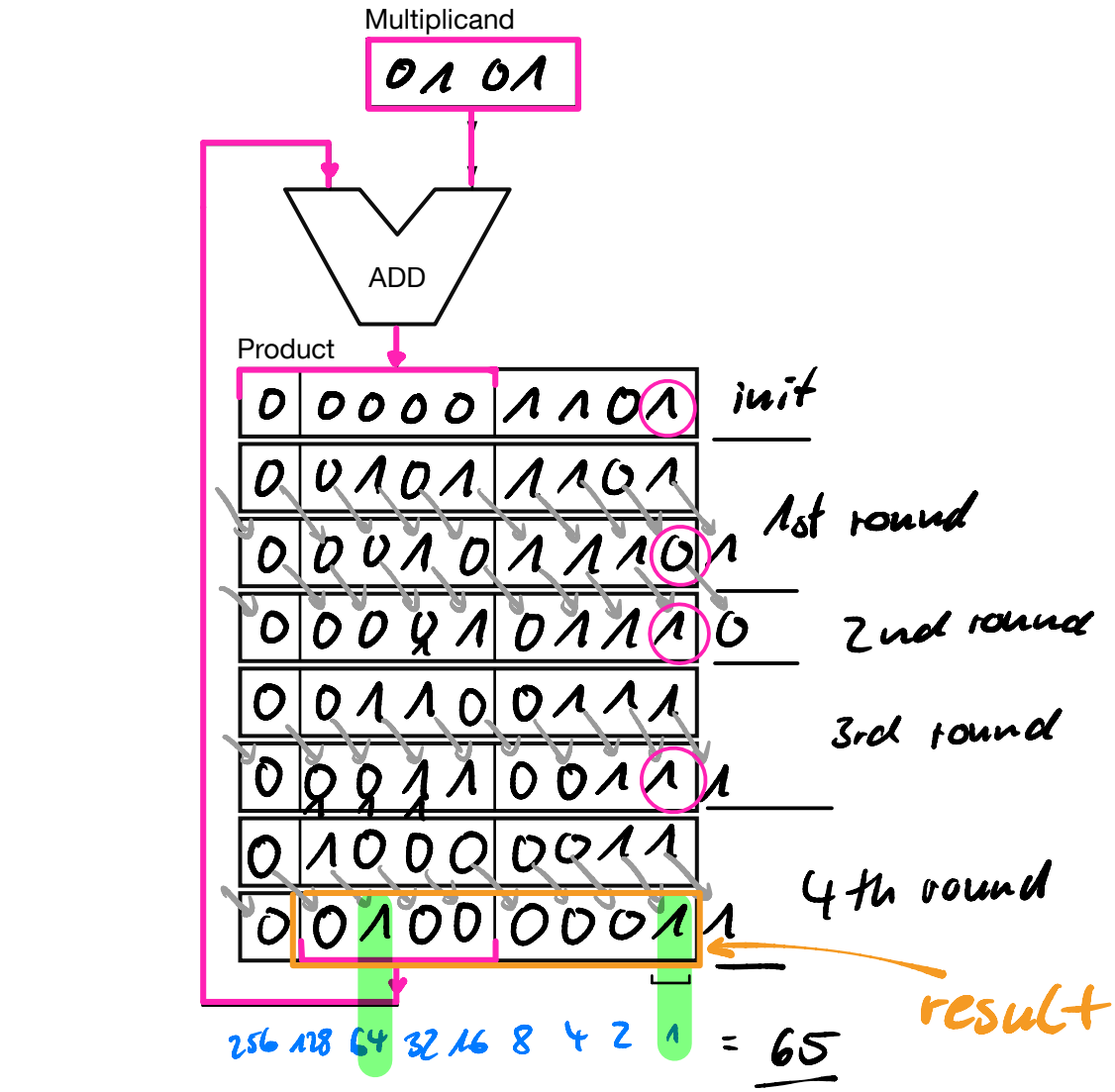
f) What is the result of the multiplication?

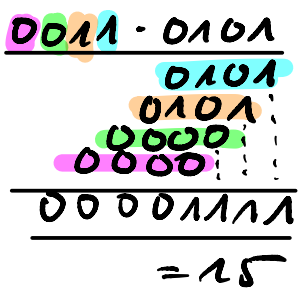


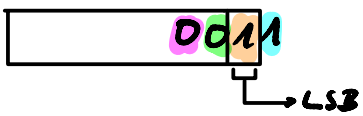
See the following control flow chart of a multiplication hardware



g) In the following figure, provide the contents of the multiplicand and product registers for the initialization step and the four calculation rounds in case of calculation 13 · 5 = 65.

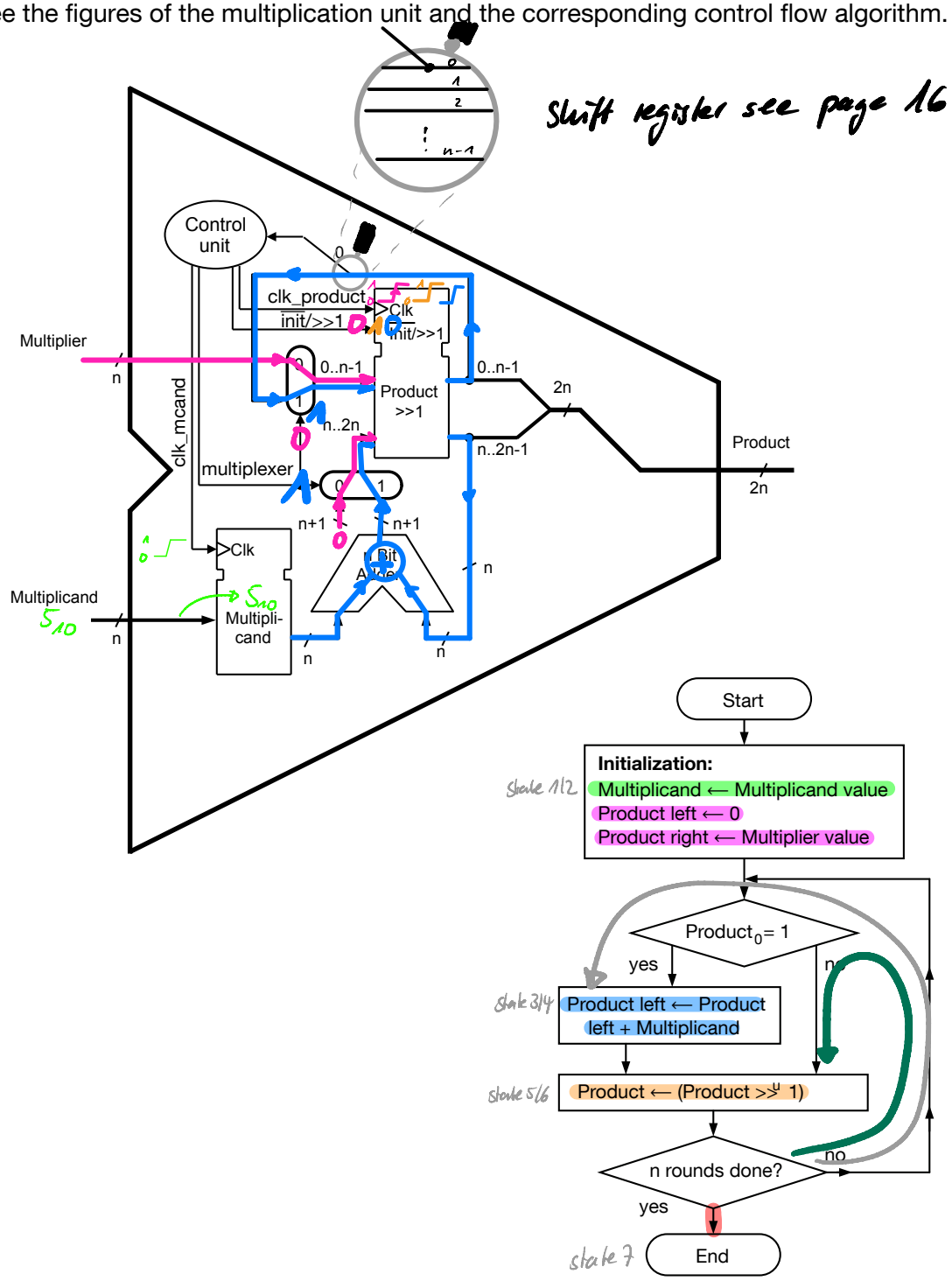






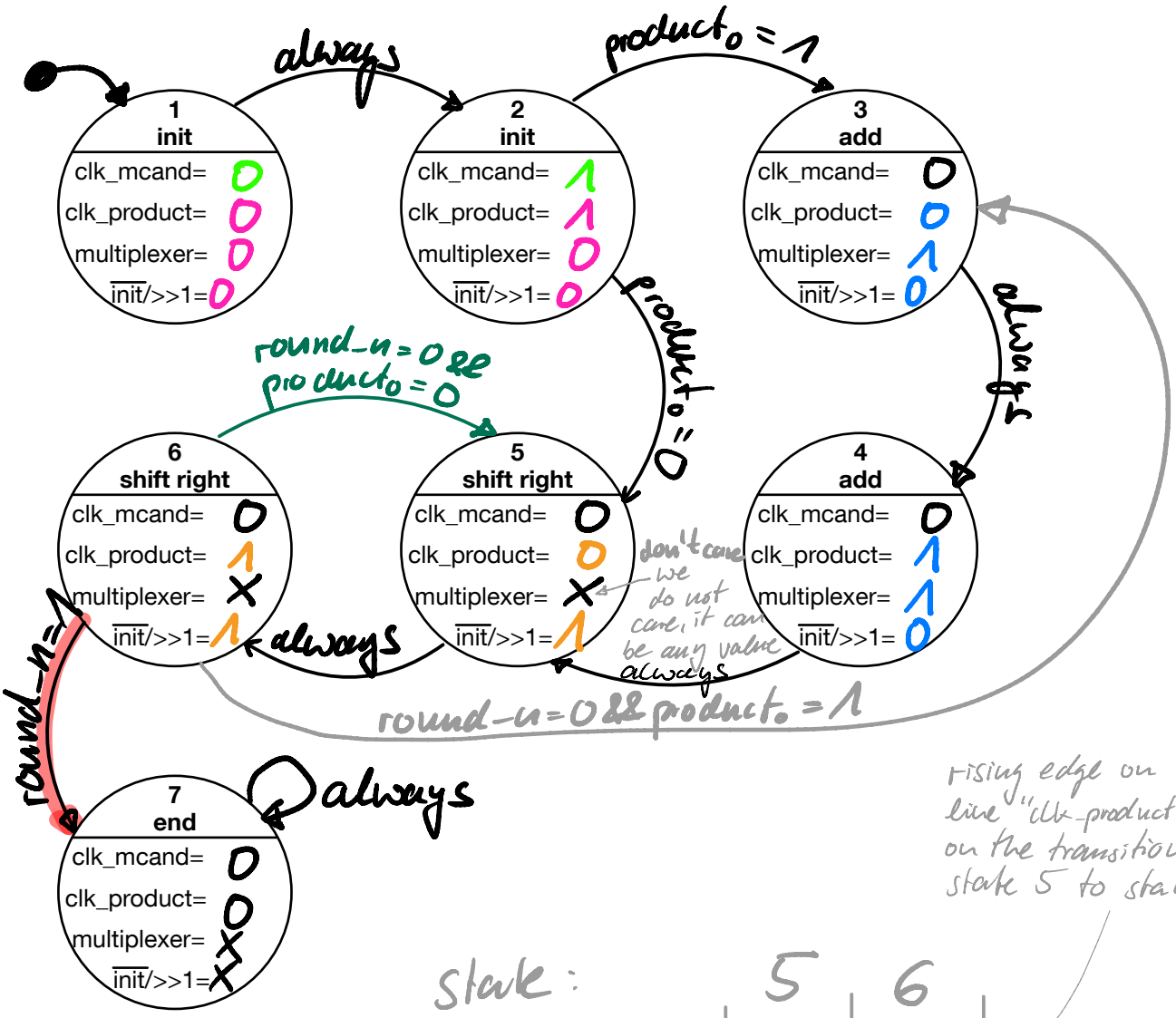
2.3 Control Unit

See the figures of the multiplication unit and the corresponding control flow algorithm.

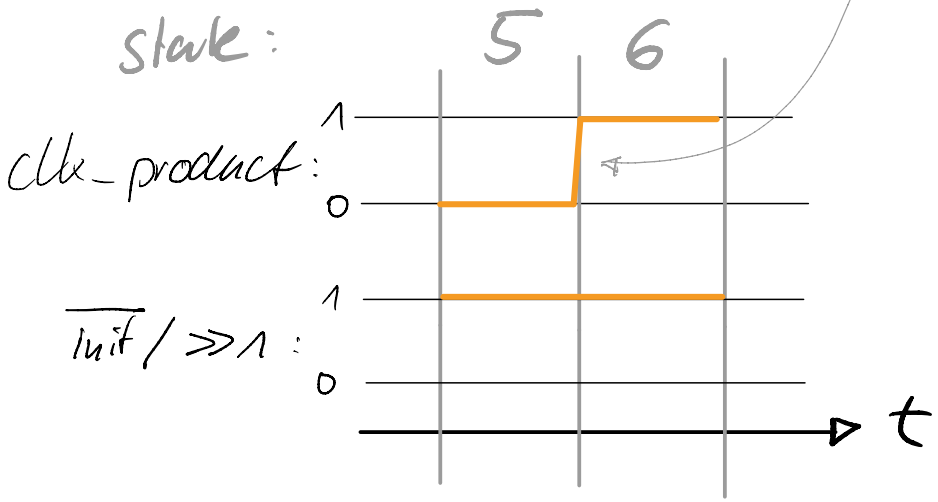


a) Complete the following state chart such that it implements the control flow algorithm to control the multiplication unit.

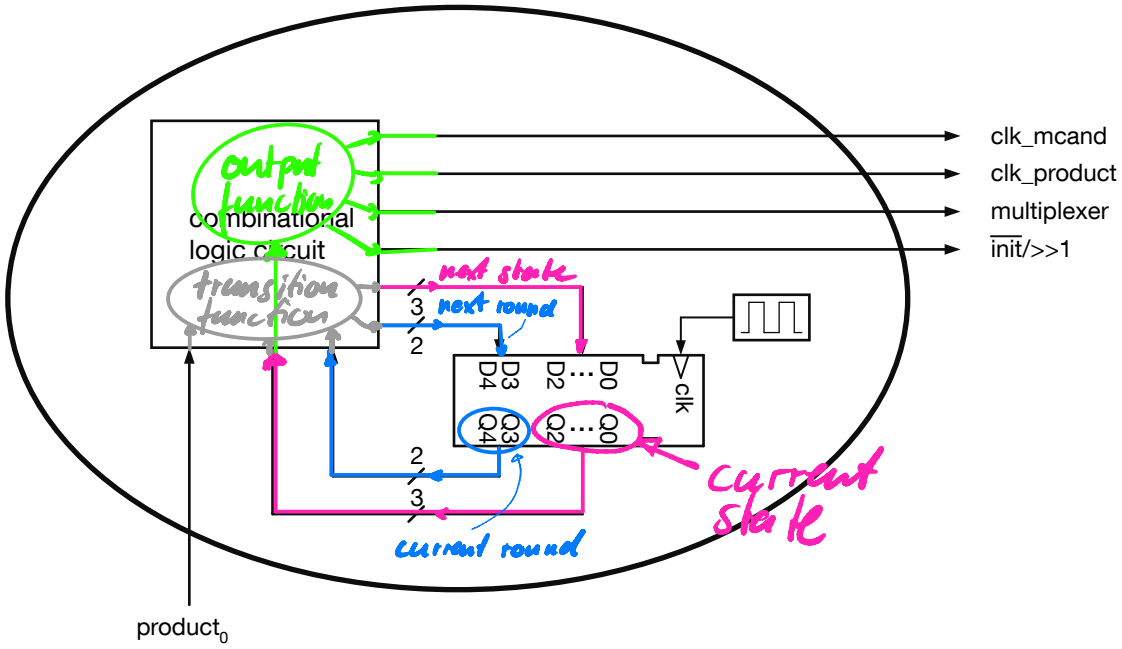
Signal round_n = { 1, if n rounds have been completed
                  { 0, otherwise



rising edge on control line "clk_product" on the transition from state 5 to state 6



The state chart will be implemented as follows:



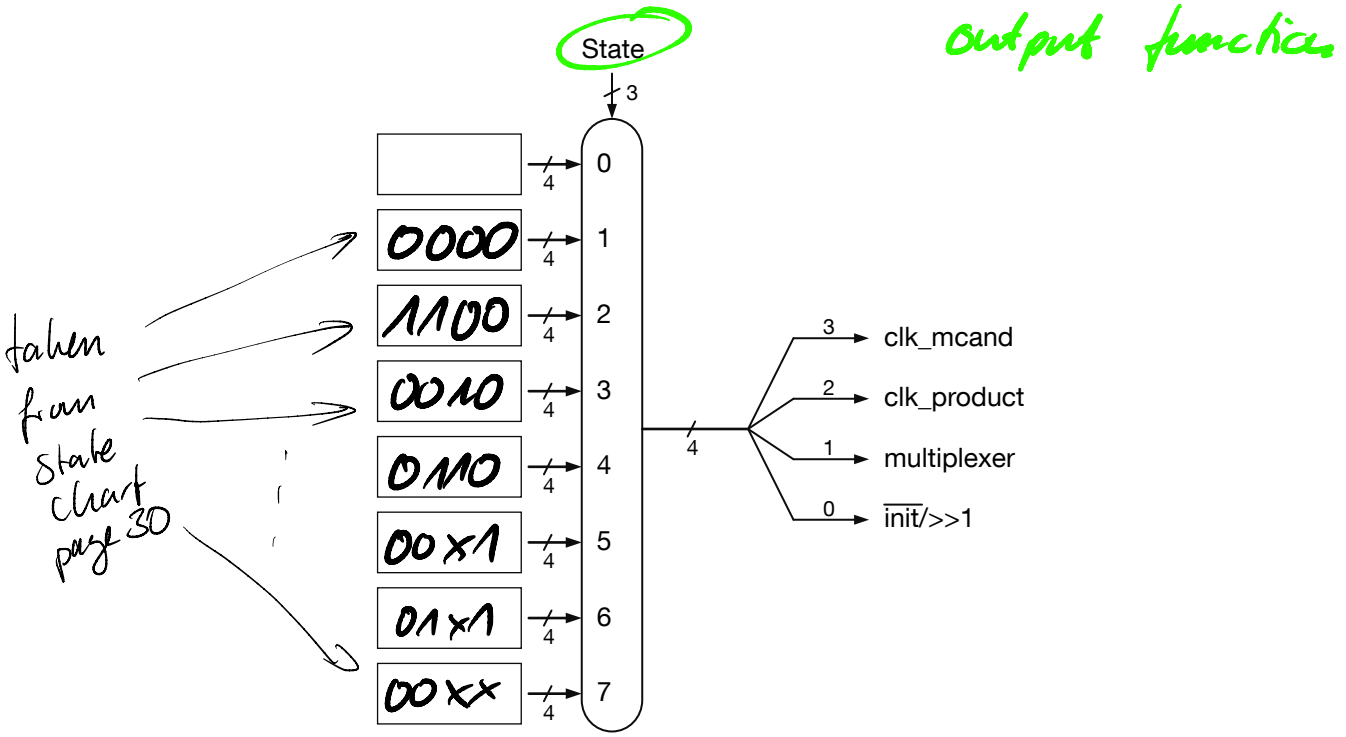
The register holds information about both the state and the number of rounds already executed.

b) Which bits will hold information about the current round and which bits will hold information about the state, if you assume a word size of n = 4.

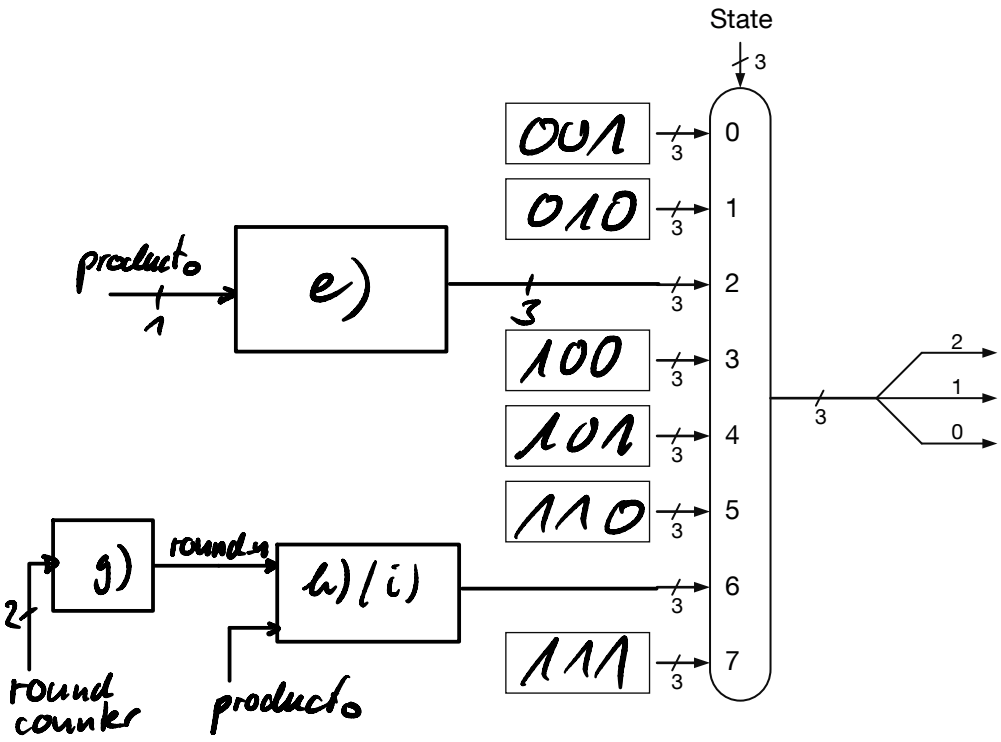
state: Q2, Q1, Q0

round: Q4, Q3

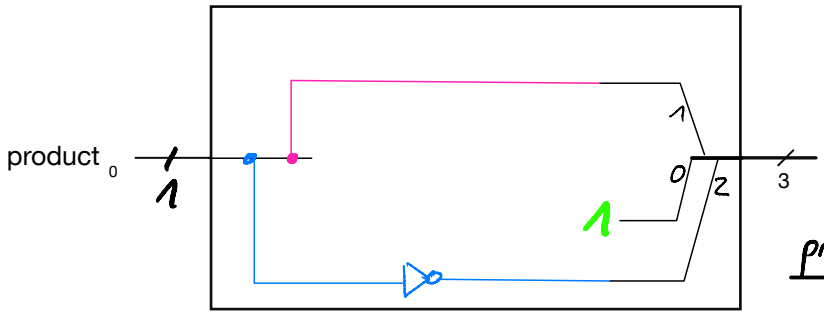
c) Provide the input to the following multiplexer to deliver the output for each state.



d) Provide the input to the following multiplexer to implement state transitions for unconditional transitions.

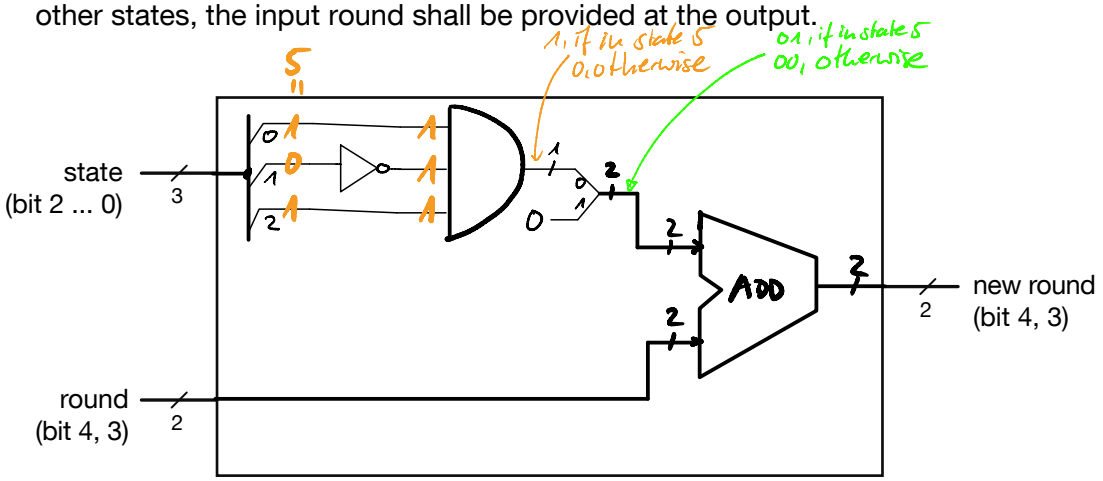


e) Complete the following figure by a circuit providing the state following state 2.

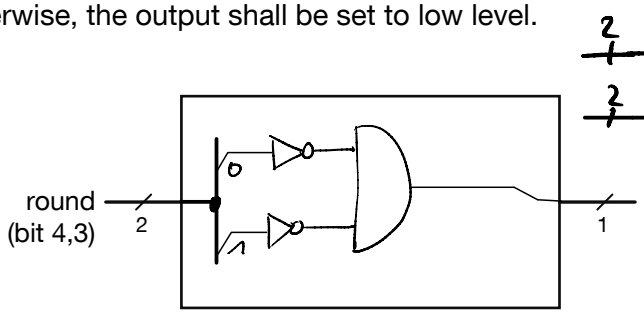


product$_0$	state$_{2\,1\,0}$	
0	101	(5 - shift)
1	011	(3 - add)

f) Complete the following round increment circuit that increments input *round* by one whenever input *state* indicates state 5 and provides the new round as output. In all other states, the input round shall be provided at the output.



g) Complete the following figure by the *round_n detector* that puts its output at high level, if both input bits are zero (round 1 = 01, round 2 = 10, round 3 = 11, round 4 = 00); otherwise, the output shall be set to low level.



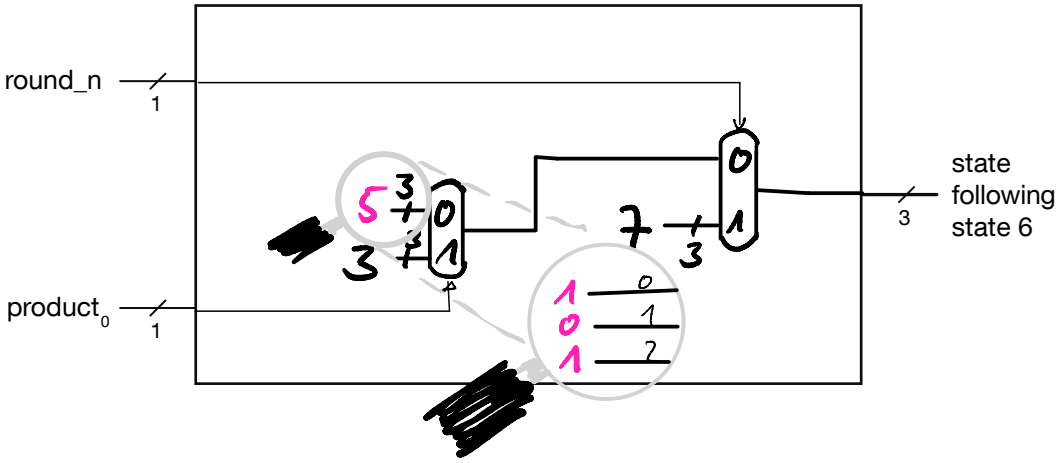
De Morgan Theorem:

$\overline{a} \wedge \overline{b} = \overline{a \vee b}$

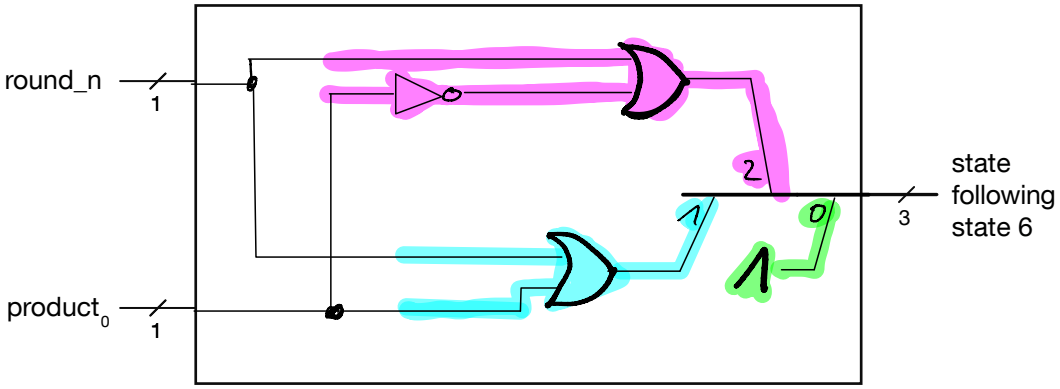
	round counter value in state 5	round counter value in state 6
1st round	00	01
2nd round	01	10
3rd round	10	11
4th round	11	(1) 00

in state 6, round_n = 1 means that the round counter has the value 00

h) In the following figure, add two multiplexers and provide an appropriate wiring in order to determine the state following state 6 depending on signals *round_n* and $product_0$

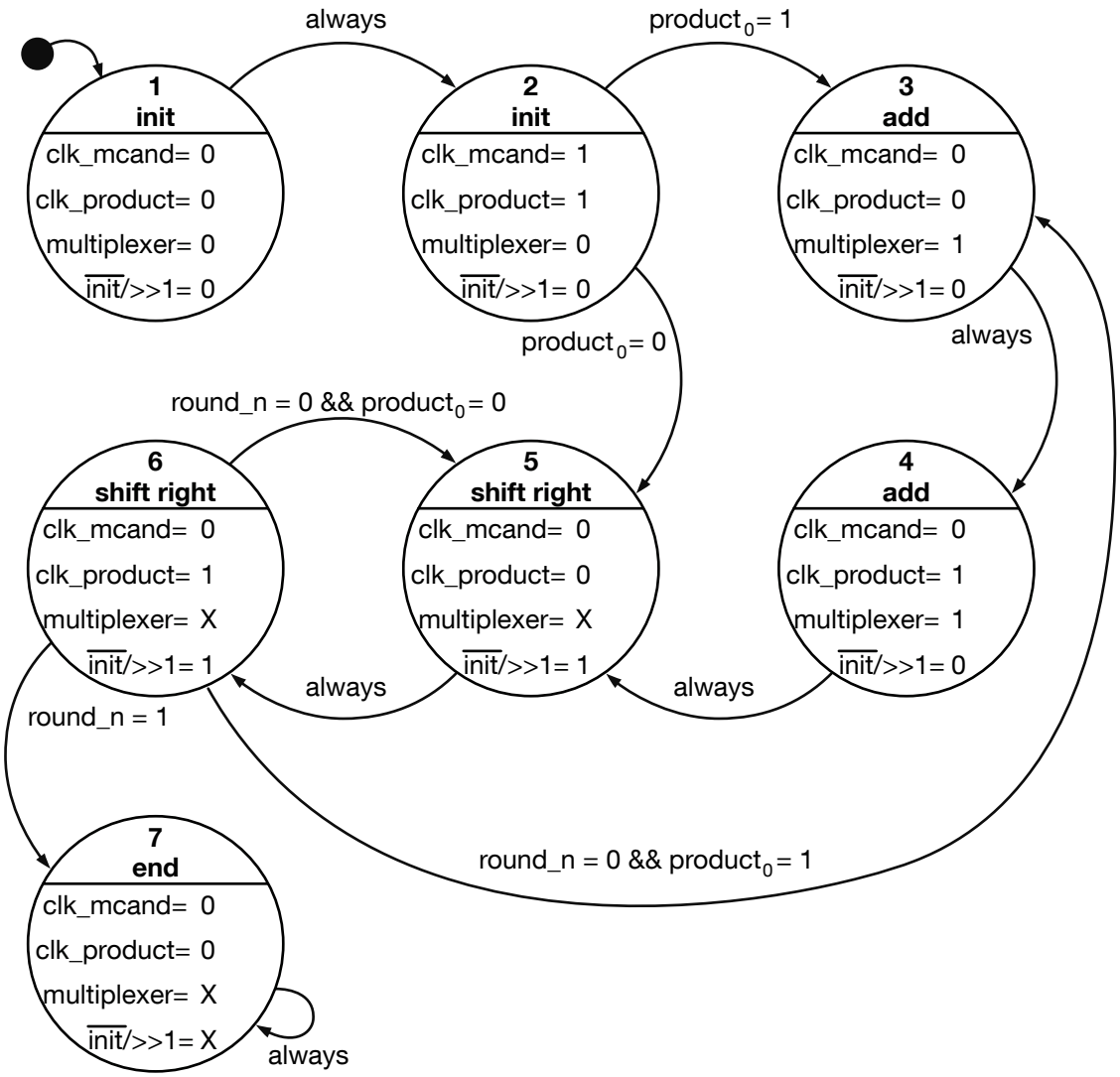


i) In the following figure, combine an inverter and two OR-gates in order to determine the state following state 6 depending on signals *round_n* and $product_0$

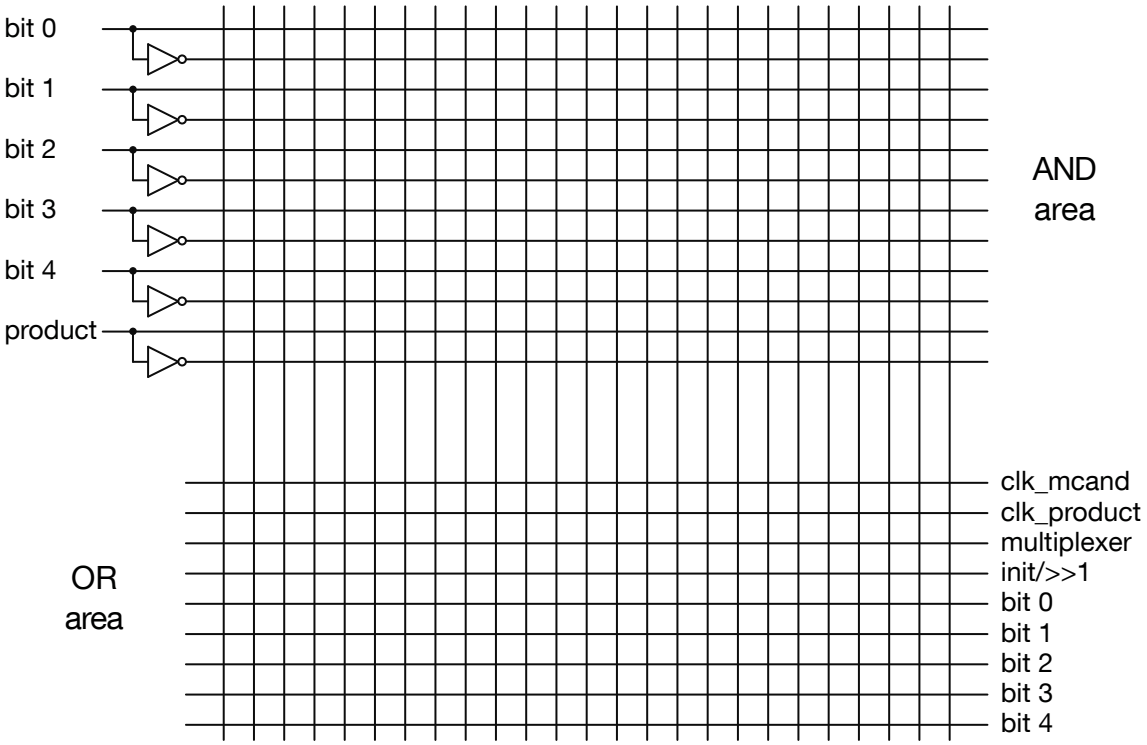


input		output	
round_n	product0	state (decimal)	state (binary) 2 1 0
1	0/1	7	1 1 1
0	0	5	1 0 1
0	1	3	0 1 1

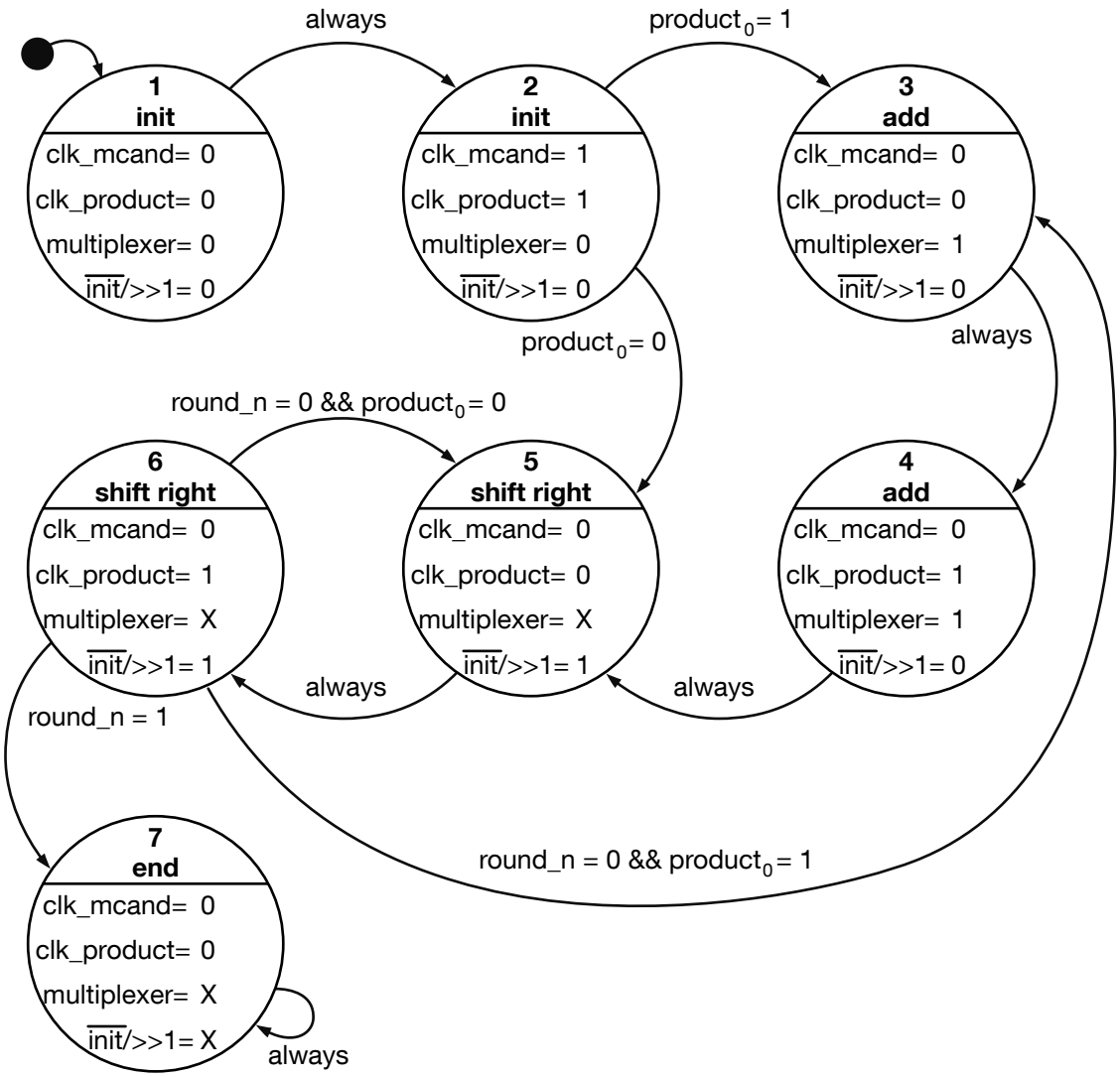
See the following state chart to control the multiplication hardware.



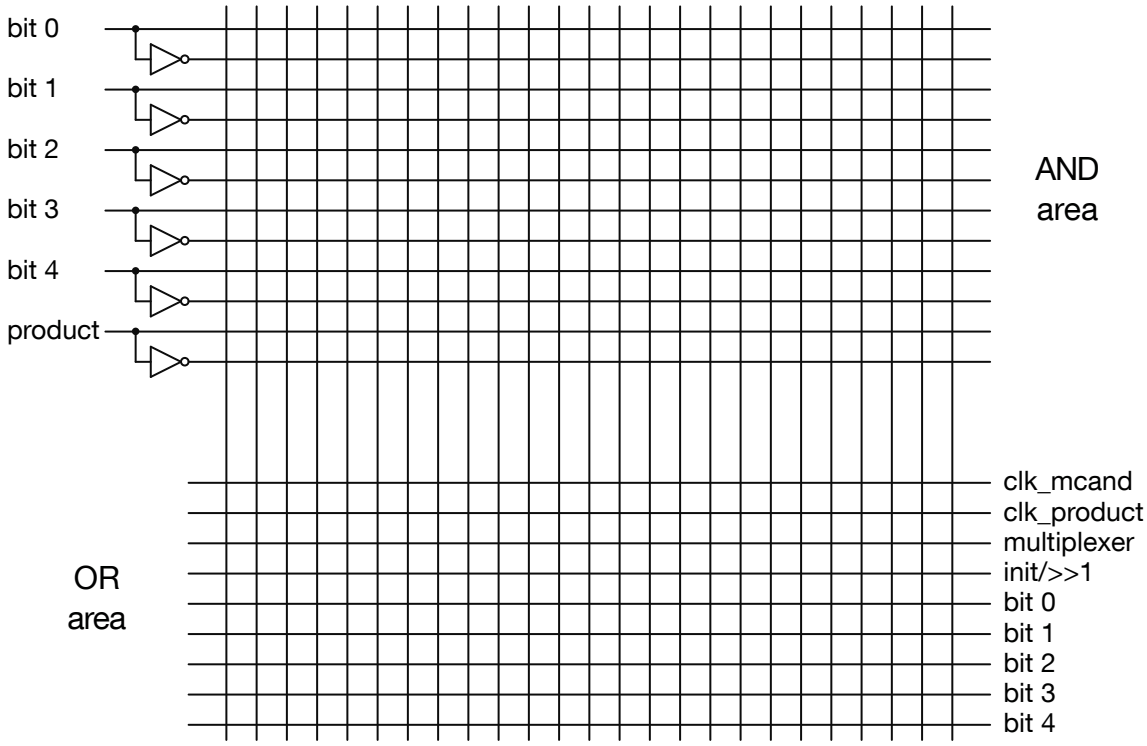
j) For the following PLA, provide appropriate connections to implement the output function of the finite state machine.



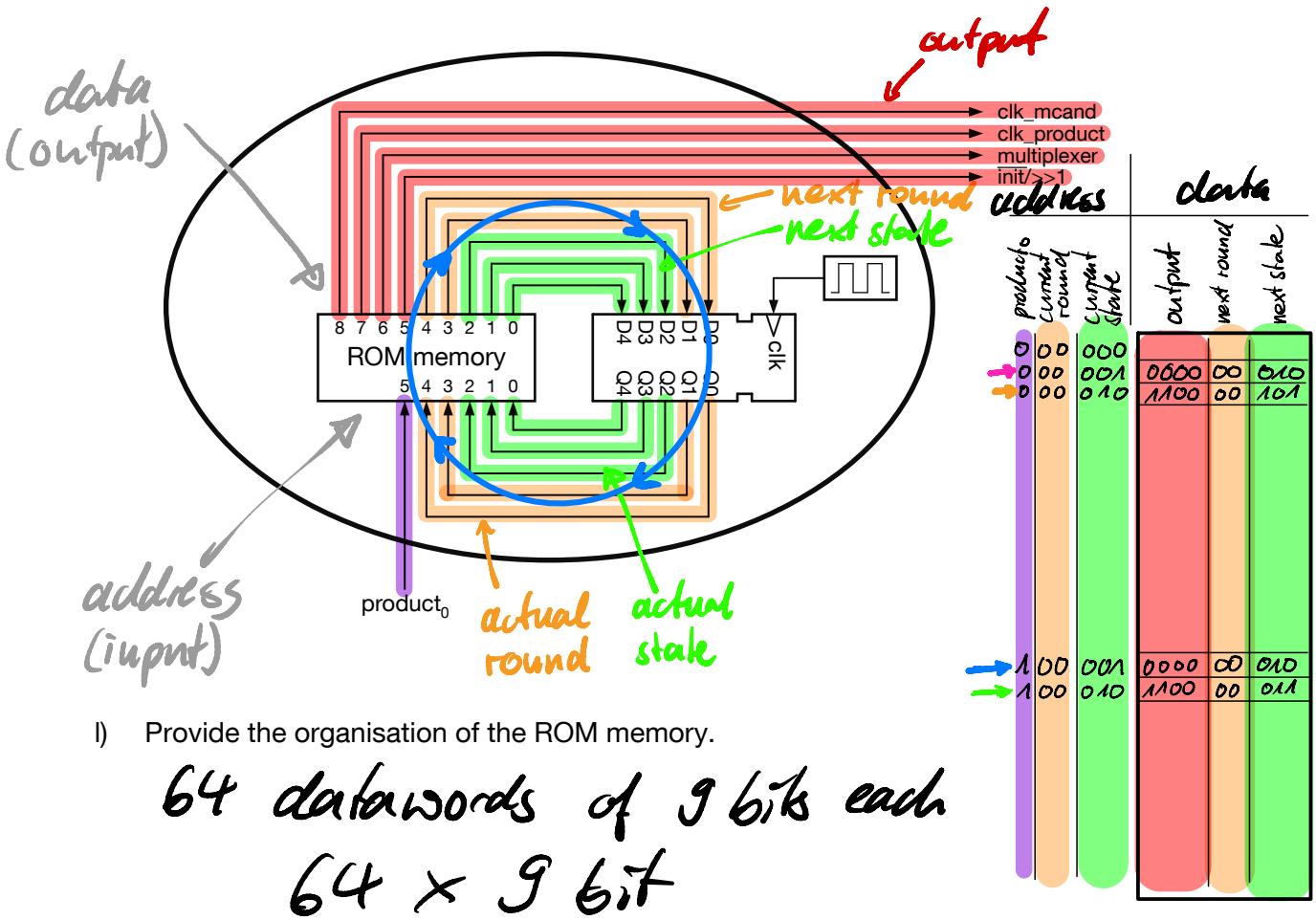
See the following state chart to control the multiplication hardware.



k) For the following PLA, provide appropriate connections to implement the state transition function of the finite state machine.



The control unit shall now be implemented with the following sequential circuit. Assume, that at beginning, the register of the circuit is reset and Q0, Q1, ... Q4 all provide 0 as output.



l) Provide the organisation of the ROM memory.

64 datawords of 9 bits each

64 × 9 bit

m) Provide the contents of the ROM memory in order to implement states 1 and 2 of the finite state machine.

state	product0	current round	current state	output	next round	next state
1	0	00	001	0000	00	010
	1	00	001	0000	00	010
2	0	00	010	1100	00	101
	1	00	010	1100	00	011

n) Provide the contents of the ROM memory in order to implement state 3 of the finite state machine.

state	product	current round	current state	output	next round	next state
3	1	00	011	0010	00	100
	1	01	011	0010	01	100
	1	10	011	0010	10	100
	1	11	011	0010	11	100

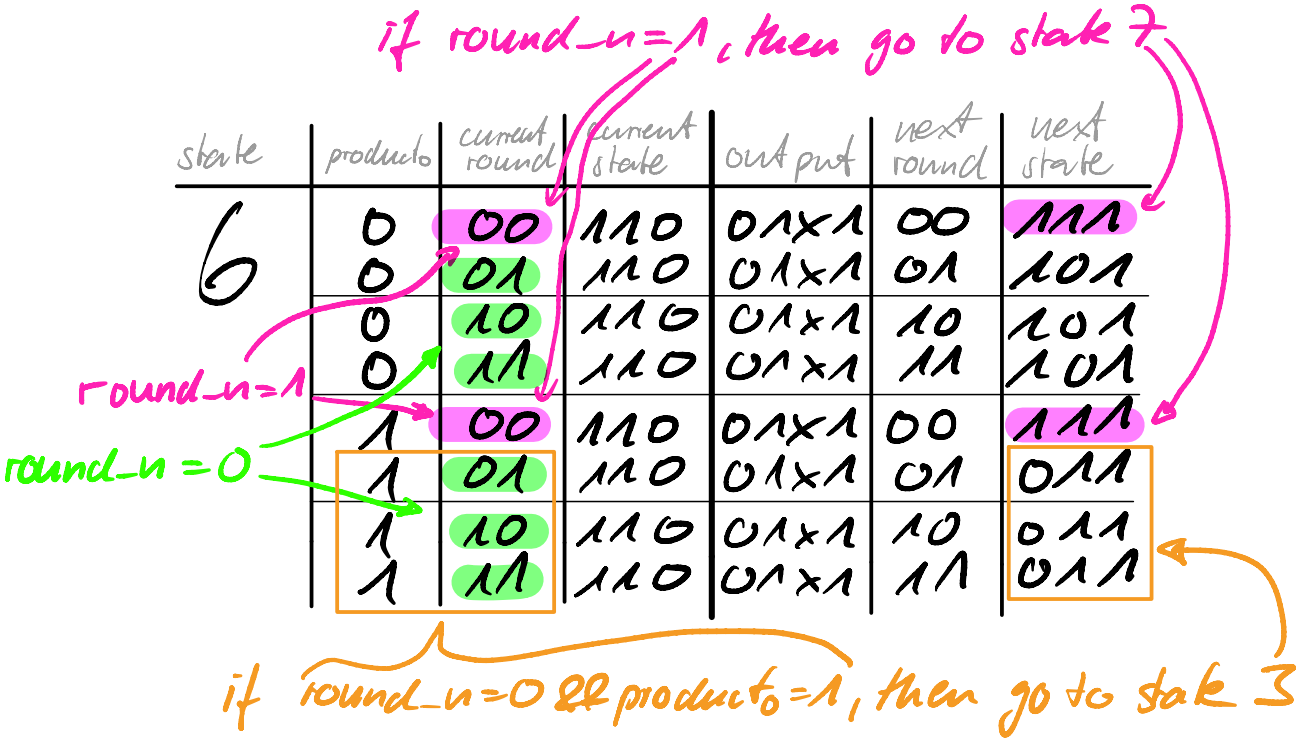
o) Provide the contents of the ROM memory in order to implement state 4 of the finite state machine.

state	product	current round	current state	output	next round	next state
4	1	00	100	0110	00	101
	1	01	100	0110	01	101
	1	10	100	0110	10	101
	1	11	100	0110	11	101

p) Provide the contents of the ROM memory in order to implement state 5 of the finite state machine.

state	product$_0$	current round	current state	output	next round	next state
5	0	00	101	00x1	01	110
	0	01	101	00x1	10	110
	0	10	101	00x1	11	110
	0	11	101	00x1	00	110
	1	00	101	00x1	01	110
	1	01	101	00x1	10	110
	1	10	101	00x1	11	110
	1	11	101	00x1	00	110

q) Provide the contents of the ROM memory in order to implement state 6 of the finite state machine.



state	product$_0$	current round	current state	output	next round	next state
6	0	00	110	01x1	00	111
	0	01	110	01x1	01	101
	0	10	110	01x1	10	101
	0	11	110	01x1	11	101
	1	00	110	01x1	00	111
	1	01	110	01x1	01	011
	1	10	110	01x1	10	011
	1	11	110	01x1	11	011

r) Provide the contents of the ROM memory in order to implement state 7 of the finite state machine.

state	product	current round	current state	output	next round	next state
7	0	00	111	00XX	00	111
	1	00	111	00XX	00	111

2.4 Universal programmable calculator

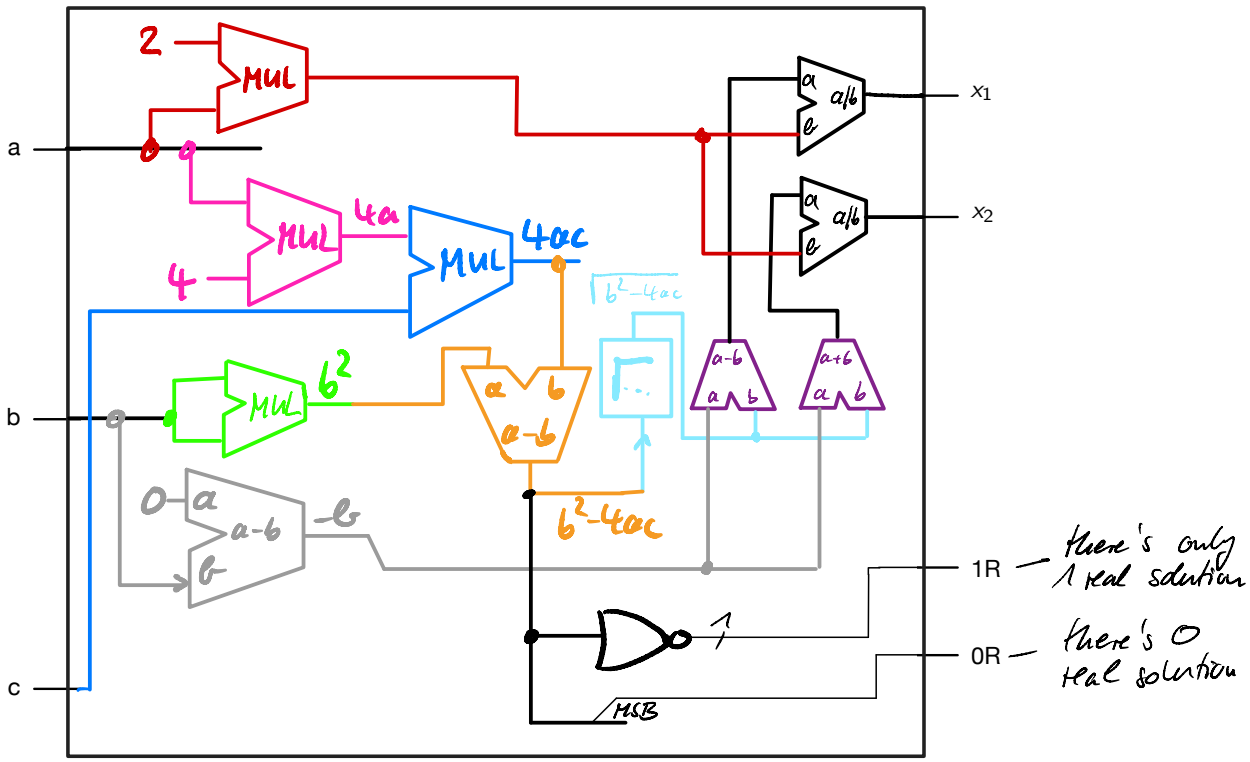
Hard coded circuit

Given the logic gates and the highler level circuits we have have built from logic gates, we are able to implement circuits that solve specific problems.
As an example, we build a hardwired circuit that solves the quadratic equation.

a) Provide and connect higher level circuits

- Adder
- Subtractor
- Multiplication unit
- Division unit
- Square root calculator

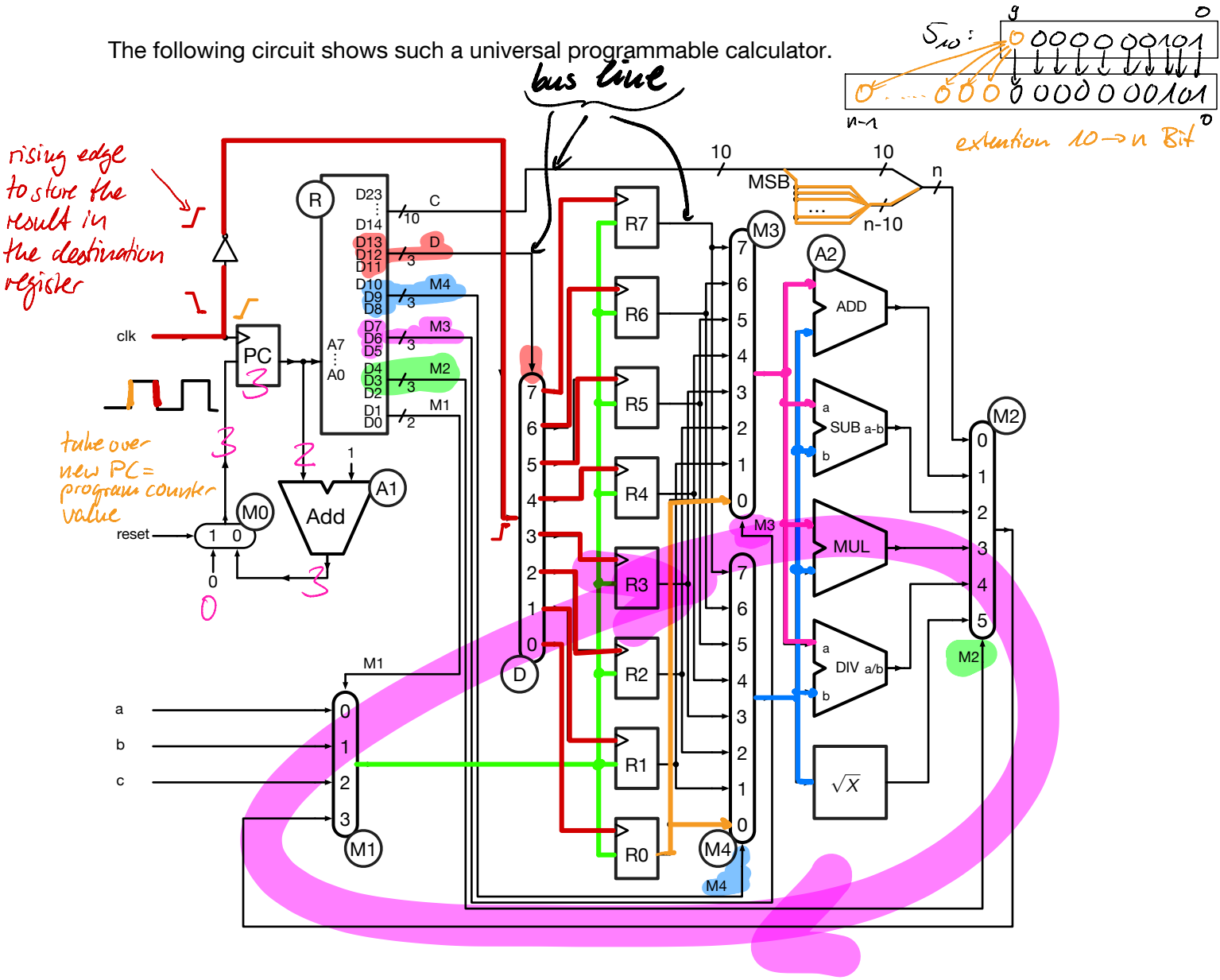
to solve the quadratic equation $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2a}$



Instruction serialization - the univeral programmable calculator

The circuit you have just drawn solves exactly one problem: The quadratic equation. If you want to solve a different problem, you have to built a different circuit.
A circuit that can be used for arbitrary calculations is called a *universal calculator* or *programmable calculator*.

The following circuit shows such a universal programmable calculator.



b) What is the purpose of the two multiplexers M3 and M4?

Select the source operands

c) How does the calculator select the mode of operation(addition, subtraction, ...)?

Multiplexer M2 selects the result of the specified operation

d) Whats the purpose of M1?

Select input a, b, c or result of the operation performed

e) What is demultiplexer D used for?

Provides the rising edge ↑ to the destination register

f) What does the inverter do?

Makes the falling edge ↓ of the clk input become the rising edge ↑ to store the result

g) Whats the purpose of register PC?

PC = program counter
stores the address of the current instruction

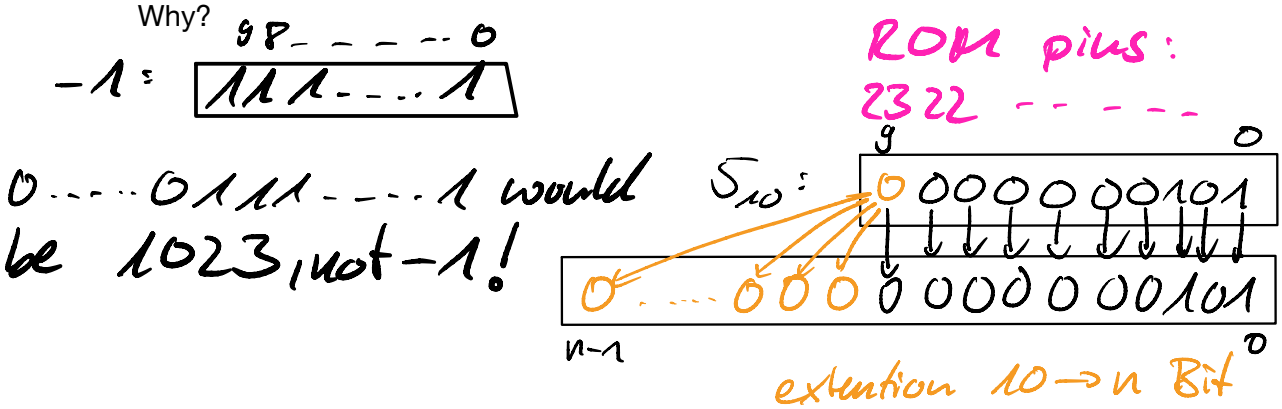
h) What is the adder A1 used for?

Increment PC to address the next/following instruction

i) What does multiplexor M0?

Enable reset to initial / start-address 0

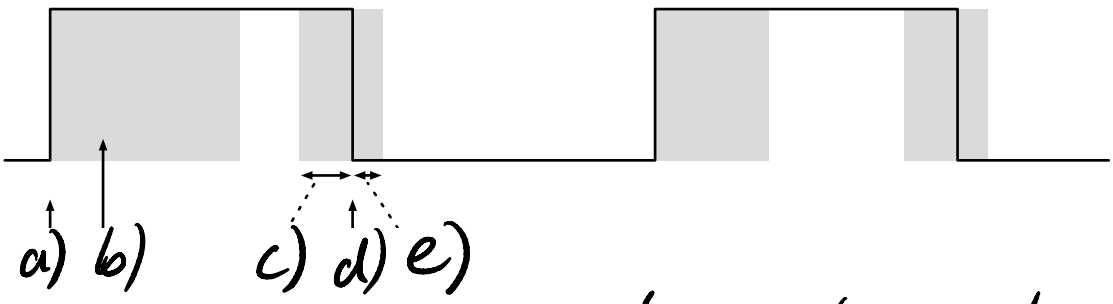
j) What happens to bit 23 of the ROM data word in the upper right part of the circuit? Why?



Bits n-1 ... 10 will be assigned D23, i.e. the MSB of the constant.
⇒ keep the value of the constant when we put the 10 bit constant on the n bit bus line

See the clk signal provided by input line *clk*.

k) Explain what the universal programmable calculator does at the inidicated position.

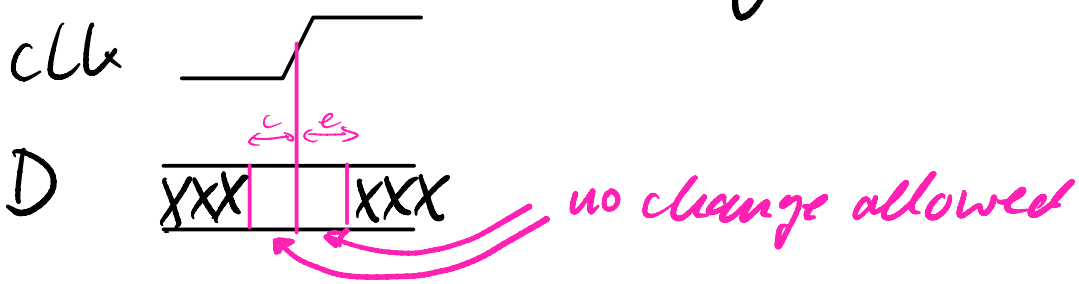


a) Rising edge on clk-input; write 0 (if reset=1) or PC+1 (if reset=0) to PC register

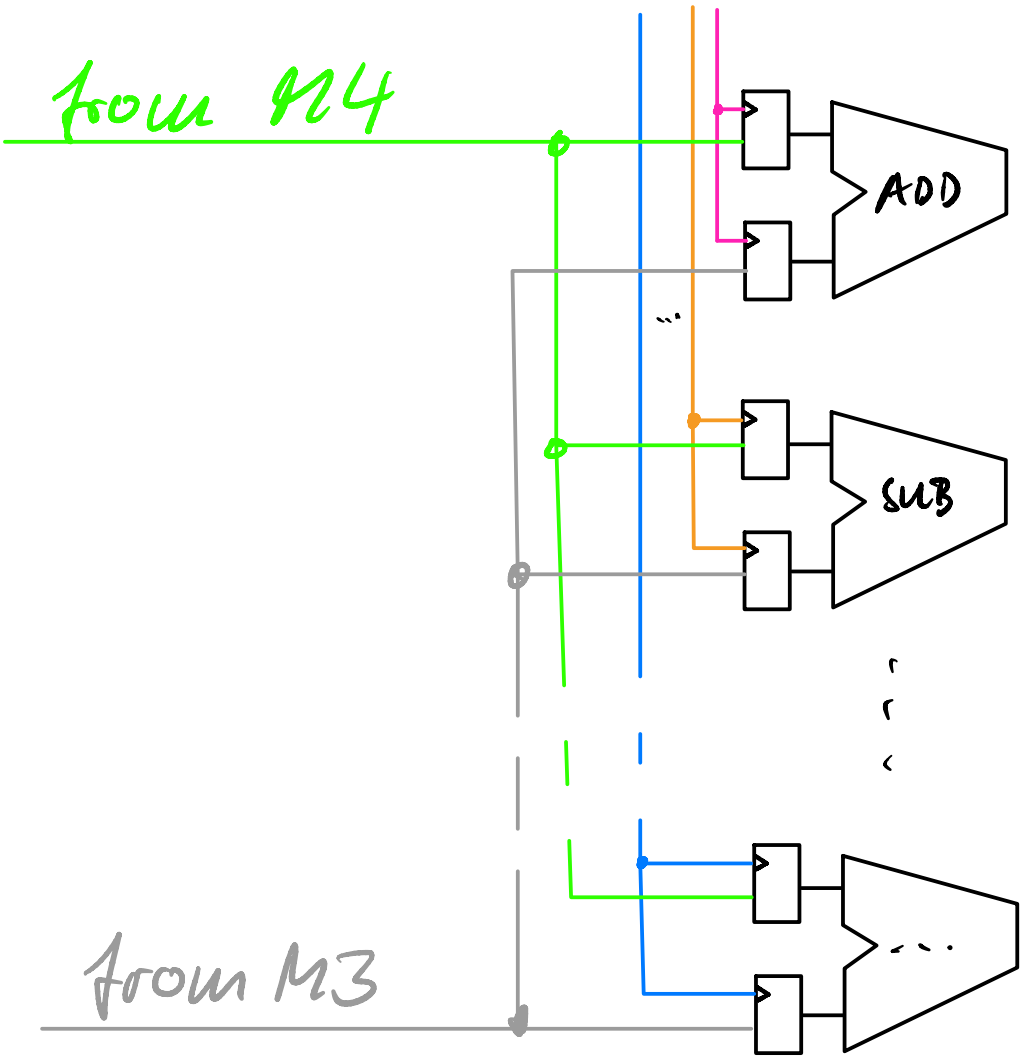
b) New PC value addresses the ROM, i.e. the next instruction; instruction is processed; result is provided at the inputs of all registers

d) falling clk-edge results in rising edge on clk-input of the selected destination register

c) setup-time } between c and e, the
e) hold-time } data to be written to the register must not change

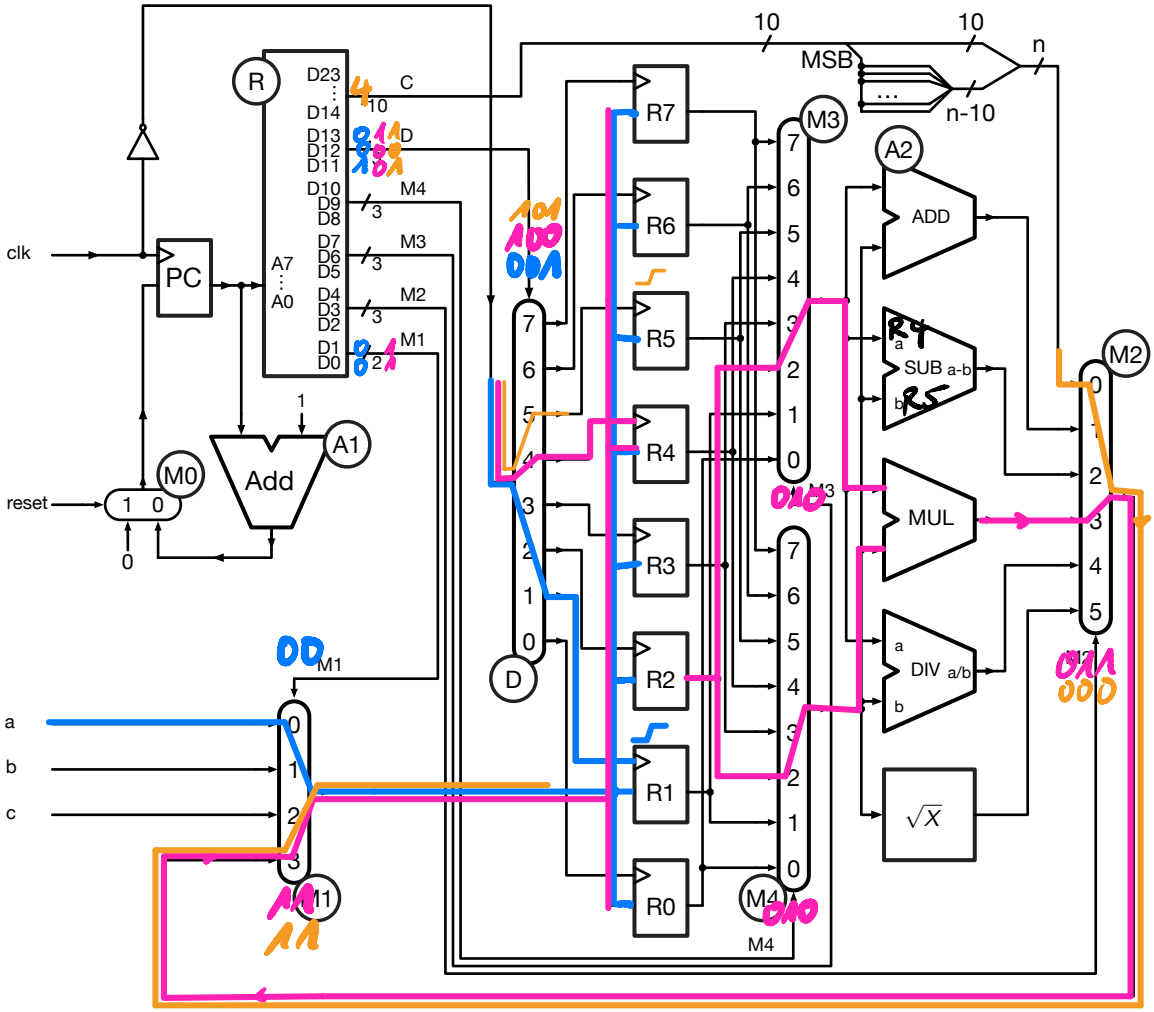


l) In CMOS circuits, much power is consumed by resistive and capacitive effects when circuit elements change their state. How could you improve the circuit that less elements change their state during operation?



provide registered input to the execution units

See again the universal programmable calculator.



Example program: Quadratic equation

In the following, we calculate the solution to the quadratic equation. I.e. we provide the bits (instruction words) to the ROM that controls the calculation circuit in such a way that it does a step-by-step calculation of the quadratic equation. For each question, follow the comments to provide the instruction words.

m) Provide binary instruction words that store inputs a, b, and c to registers R1, R2, and R3

ROM address — data stored in memory / ROM data

	C	D	M4	M3	M2	M1	Comment
00000000	X - - - X	001	XXX	XXX	XXX	00	R1 ← a
00000001	X - - - X	010	XXX	XXX	XXX	01	R2 ← b
00000010	X - - - X	011	XXX	XXX	XXX	10	R3 ← c

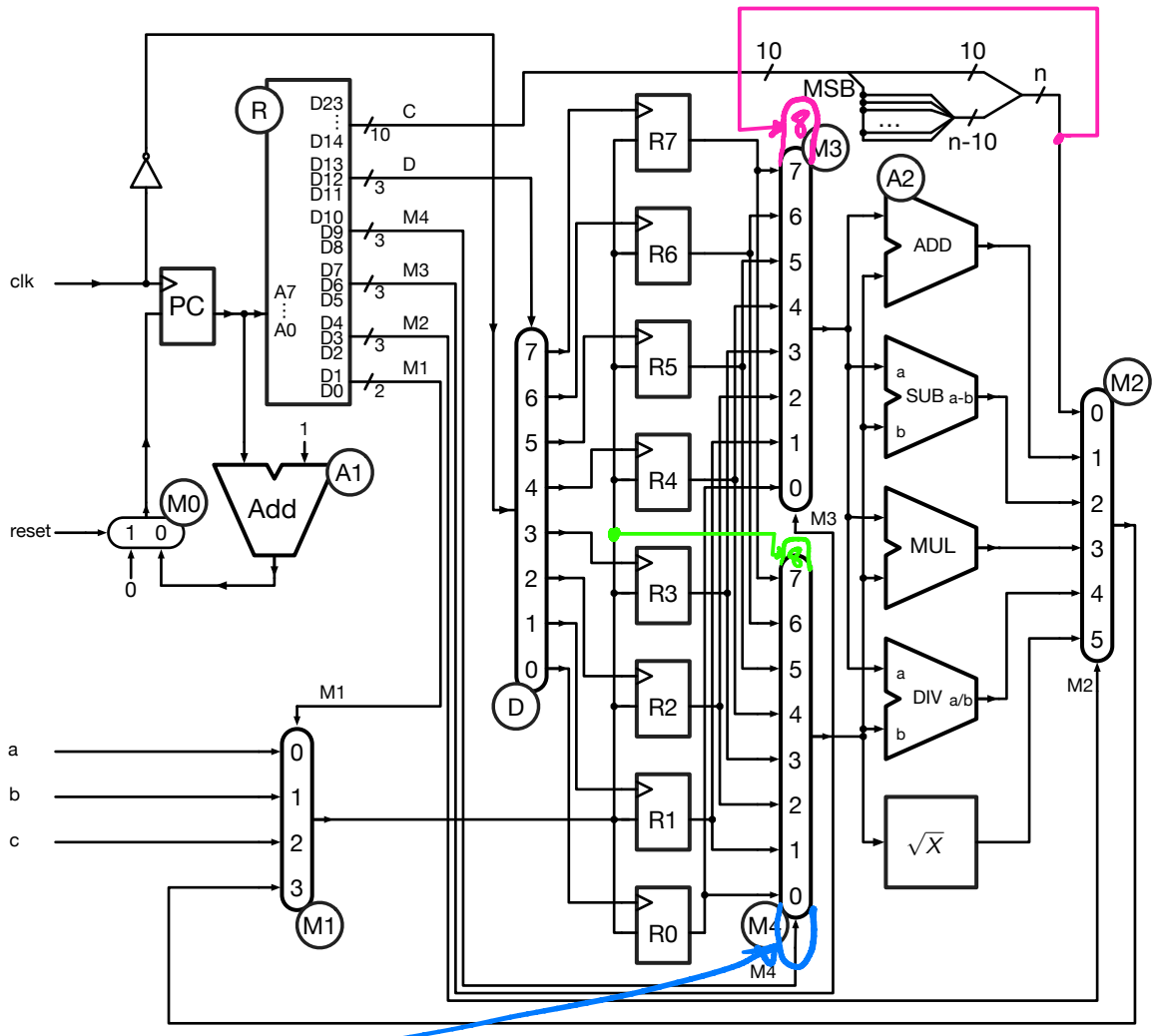
n) Provide binary instruction words to calculate $\sqrt{b^2 - 4ac}$ and store the result to register 4.

	C	D	M4	M3	M2	M1	Comment
00000011	X - - - - X	100	010	010	011	11	R4 ← b^2
00000100	0 - - - 0100	101	XXX	XXX	000	11	R5 ← 4
00000101	X - - - X	101	001 / 101	101 / 001	011	11	R5 ← 4a (R5 ← R5, R1)
00000110	X - - - X	101	101 / 011	011 / 101	011	11	R5 ← 4ac (R5 ← R5, R3)
00001111	X - - - X	100	101	100	010	11	R4 ← b^2 − 4ac (R4 ← R4 − R5)
00010000	X - - - X	100	100	XXX	101	11	R4 ← $\sqrt{b^2-4ac}$ (R4)

o) Provide binary instruction words to calculate $(-b \pm \sqrt{b^2 - 4ac})/2a$ and store the result to registers R1 and R2.

ROM address	ROM data						
	C	D	M4	M3	M2	M1	Comment
0000 1001	0 - - - - 0	1101	xxx	xxx	000	11	R5 ← 0
0000 1010	x - - - - x	1011	010	101	010	11	R5 ← 0 − 6
0000 1011	x - - - x	1110	101 100	100 101	001	11	R6 ← −6 + √
0000 1100	x - - - x	1111	100	101	010	11	R7 ← −6 − √
0000 1101	0 - - - 010	0000	xxx	xxx	000	11	R0 ← 2
0000 1110	x - - - - x	0000	000 001	001 000	011	11	R0 ← 2 · a
0000 1111	x - - - x	0001	000	110	100	11	R1 ← (−b+√)/2a
0001 0000	x - - - x	0010	000	111	100	11	R2 ← (−b−√)/2a

See again the universal programmable calculator.



Memory

M4876E58 ← Memory M2456

```
ADD  R0, R0, M2456
ADD  R0, R0, R1
ADD  R0, R0, 1
ADD  R0, R0, $1
```

special character to make „input 1" different from „number 1"

Example program: Volume of a sphere

To show that the universal programmable calculator can perform arbitrary calculations, we calculate the volume of a sphere. The radius is provided at calculator input 0.

p) Provide the binary instruction word to read input 0 to register R0.

C	D	M4	M3	M2	M1	Comment
x x - - - x	000	x x x	x x x	x x x	00	R0 ← input 0

q) Provide binary instruction words to calculate $4 \cdot r^3 \cdot 314$ and store the result to register R1. Note that $314_{10} = 100111010_2$.

C	D	M4	M3	M2	M1	Comment
0 - - - 0100	001	x x x	x x x	000	11	R1 ← 4
0100111010	010	x x x	x x x	000	11	R2 ← 314
x - - - x	001	001 010	010 001	011	11	R1 ← 4 · 314
x - - - x	001	001 000	000 001	011	11	R1 ← 4 · 314 · r
x - - - x	001	000 000	000 001	011	11	R1 ← 4 · 314 · r^2
x - - - x	001	000 000	000 001	011	11	R1 ← 4 · 314 · r^3

$$V = \frac{4}{300} \cdot r^3 \cdot \frac{314}{11}$$

288 { 256 : 01 0000 0000
32 : 00 0010 0000
00 0000 1100
01 0012 1100

r) Provide binary instruction words to divide the dataword stored in R1 by 300 and store the result in register R1.

C	D	M4	M3	M2	M1	Comment
01 0010 1100	010	x x x	x x x	000	11	R2 ← 300
x - - - - - x	001	010	001	100	11	R1 ← R1 / R2

Assembly language

Programming our univeral programmable calculator by setting the bits in the instruction word manually is a tedious job. It would be much easier to formulate the job according to the comments (rightmost column) and let a computer program do the job of translating the comment/instruction to binary format. Such a program is called *assembly program*. The language it translates to binary instruction format is called *assembly program*.
Our calculator reads two source operands/registers and calculates a result that has to be stored to a destination operand/register. I.e. the assembly language has to specify

- up to three operands (2 x source, 1 x destination), and
- the operation to be performed (addition, subtraction, ...).

Assembly program to solve the quadratic equation

The following code shows a possible assembly program to calculate the quadratic equation:

operation destination source

m)

```
INPUT   R1,0      // R1 ← a
INPUT   R2,1      // R2 ← b
INPUT   R3,2      // R3 ← c
```

n)

```
MUL     R4,R2,R2  // R4 ← b^2
SET     R5,4      // R5 ← 4
MUL     R5,R5,R1  // R5 ← 4 · a
MUL     R5,R5,R3  // R5 ← 4 · a · c
SUB     R4,R4,R5  // R4 ← b^2 − 4 · a · c
SQRT    R4,R4     // R4 ← √(b^2 − 4 · a · c)
```

o)

```
SET     R5,0      // R5 ← 0
SUB     R5,R5,R2  // R5 ← −b
ADD     R6,R5,R4  // R6 ← −b + √(b^2 − 4 · a · c)
SUB     R7,R5,R4  // R7 ← −b − √(b^2 − 4 · a · c)
SET     R0,2      // R0 ← 2
MUL     R0,R1,R0  // R0 ← 2 · a
DIV     R1,R6,R0  // R1 ← (−b + √(b^2 − 4 · a · c))/(2 · a)
DIV     R2,R7,R0  // R2 ← (−b − √(b^2 − 4 · a · c))/(2 · a)
```

This code means is just a representation of the bitwise specified instruction words we have seen before. It is much easier to read und much faster to write.

Defining an assembly language for our calculator.

- Empty program lines can be used to structure the program. They are ignored by the assembler.

- If a line is not empty
  - the first word in a line (i.e. all characters up to the first whitespace) define the instruction, and
  - the second word in the line defines the operands:
    - the operands are separated by commas
    - the first operand represents the destination
    - the second and third operands represent the source operands
  - words following the operands are expected to represent comments and are ignored.

- Instructions and Operands
  - `INPUT:` Reads data from input 0, 1, and 2 of multiplexer M1 and stores it to a register; the first operand specifies the register number, the second operand specifies the multiplexer control line.
  - `ADD, SUB, MUL, DIV:` Perform the corresponding operations; the first operand is the destination register, the second and third operand represent the source operands.
  - `SQRT:` Calculates the square root; the first operand is the destination register, the second operand is the source register.
  - `SET:` Writes a 10 bit constant data word to a register; the first operand is the destination register, the second operand represents the constant.

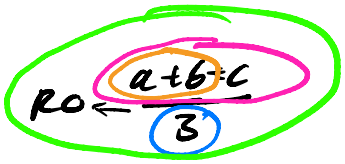
Example assembly programs

a) Provide assembly instructions to read the three sides of cuboid from input lines 0, 1, and 2, calculate the volume and store the result to register R0.

```
INPUT   R0, 0
INPUT   R1, 1
MUL     R0, R0, R1
INPUT   R1, 2
MUL     R0, R0, R1
```

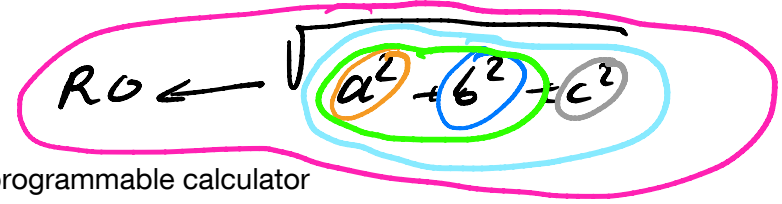
b) Provide assembly instructions to calculate the arithmetic mean of the numbers presented at inputs 0, 1, and 2 and store the result to register R0.

```
INPUT   R0, 0
INPUT   R1, 1
ADD     R0, R0, R1
INPUT   R1, 2
ADD     R0, R0, R1
SET     R1, 3
DIV     R0, R0, R1
```



Divide and conquer

Split large problem in sub-problems to solve the large problem



c) Provide assembly instructions to read the coordinates of a 3D vector from inputs 0, 1, and 2, calculate the norm and store the result to register R0.

```
INPUT  R0, 0
MUL    R0, R0, R0      R0 ← a²
INPUT  R1, 1
MUL    R1, R1, R1      R1 ← b²
ADD    R0, R0, R1      R0 ← a² + b²
INPUT  R1, 2
MUL    R1, R1, R1      R1 ← c²
ADD    R0, R0, R1      R0 ← a² + b² + c²
SQRT   R0, R0          R0 ← √(a² + b² + c²)
```

```
ADD    R0, R0, R1
ADD    R0, R0, 1
```

10 bits signed (2's complement) → −512, … 0, … 511

$A = r^2 \cdot \pi$ 314 / 314000

d) Provide assembly instructions to read the radius of a circle from input 0, calculate the area and store the result to register R0. Apply division 3141592 / 1000000 to calculate π; keep an eye on the size of the constant (10 bit).

```
INPUT  R0, 0        R0 ← r
MUL    R0, R0, R0   R0 ← r²
SET    R1, 314      R1 ← 314
SET    R2, 100      R2 ← 100
MUL    R1, R1, R2   R1 ← 31400
SET    R3, 15       R3 ← 15
ADD    R1, R1, R3   R1 ← 31415
MUL    R1, R1, R2   R1 ← 3141500
SET    R3, 92       R3 ← 92
ADD    R1, R1, R3   R1 ← 3141592
MUL    R0, R0, R1   R0 ← 3141592 · r²
DIV    R0, R0, R2   R0 ← 31415.92 · r²
DIV    R0, R0, R2   R0 ← 314.1592 · r²
DIV    R0, R0, R2   R0 ← 3.141592 · r²
```

Classification of instruction sets

Instruction set

The set of instruction that a processor understands is called the *instruction set*. Our universal programmble calculator understands 7 instructions: `ADD`, `SUB`, `MUL`, `DIV`, `FSQRT`, `INPUT`, and `SET`.

Classification in terms of complexity

CISC = Complex Instruction Set Architecture

- Very powerful instructions; a single instruction performs a number of tasks, e.g. read two operands from memory, perform an addition and store the result back to memory.
- Very popular in the 1970s where the amount of memory available to the programmer was very small; no compiler available.
- Allows faster coding on assembly level. However, for most cases assembly coding has been replaced by programming in higher languages, e.g. C, Python.
- instruction word size varies heavily (e.g. 1...16 byte for x86)

RISC = Reduced Instruction Set Architecture

- Instructions not too powerful; rather use multiple “small” instructions instead of a very powerful instruction. Moves complexity to the compiler.
  - read first operand from memory to register
  - read second operand from memory to register
  - perform addition and store result in register
  - write back result from register to memory
- Constant instruction word size (e.g. 4 byte) for any instruction available on the processor. Makes hardware easier and faster.

Classification according to the location of operands

- *Register-Memory-Architecture:* Operands might be located in registers or in memory
- *Register-Register-Architecture/Load-Store-Architecture:* Source and destination operands have to be located in registers. There are special load and store instructions to access memory which have to be specified explicitely.

Classification depending on the amount of operands

- *Three address machine:* Allows to specify up to two source operands and one operand for the destination. Example: `ADD R0,R1,R2` ⇔ `R0 = R1 + R2`
- *Two address machine:* The destination operand also acts as one source operand. Example `ADD R0,R1` ⇔ `R0 = R0 + R1`
- *One address machine:* Only one source operand has to be specified; a special purpose accumulator register is used as the other source operand and as the destination. Example: `ADD R1` ⇔ `A = A + R1`
- *Zero address machine/stack machine:* No operand has to be specified; they all reside at the top of a stack. It's just the operation that hast to be specified. Example:

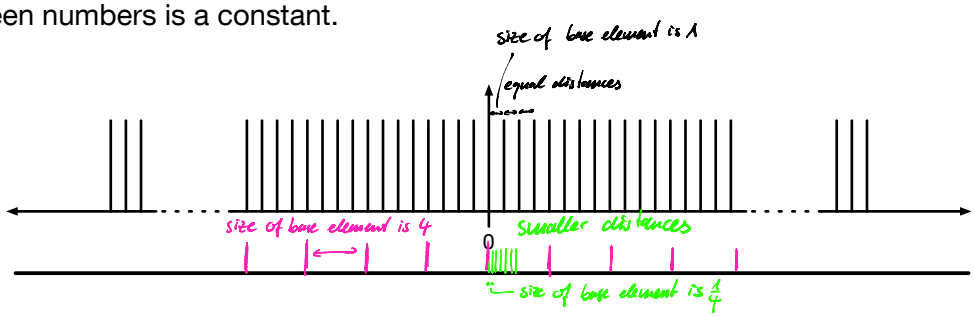
```
push 3
push 4
add
```

3 Data Structures and Algorithms

3.1 Elementary data types

Fixed point numbers

Fixed point numbers have their radix point at a fixed position. Therefore, the distance between numbers is a constant.



We will use the following fixed point data types: `char`, `short`, `int`, `long int`, `long long int` in their singed (2's complement) and unsigned variant.

a) Provide the bitpattern of 16 bit unsigned number 12.

```
→ Code gcc -o p p.c
→ Code ./p
d: 12
0000000000001100
→ Code
```

b) Provide the bitpattern of 8 bit signed number -2.

c) Provide a C program that keys in an signed integer and prints its representation in binary.

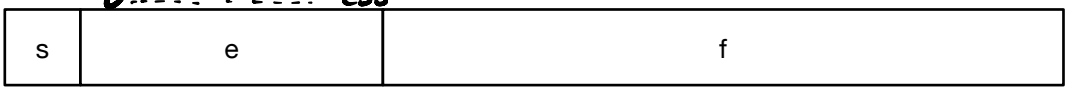
```c
#include <stdio.h>

int main()
{
    int d, i;

    printf("d: ");
    scanf("%d", &d);

    // 8 bit: sizeof(d) = 1
    // 16 bit: sizeof(d) = 2
    // 32 bit: sizeof(d) = 4
    // 64 bit: sizeof(d) = 8
    for(i = (sizeof(d) * 8) - 1; i >= 0; i--)
    {
        printf("%d", (int) ( (d >> i) & 1 ));
    }
    printf("\n");
}
```

```
→ Code gcc -o p p.c
→ Code ./p
d: 132
00000000000000000000000010000100
→ Code
```

Floting point numbers

Floating point numbers have the position of the radix encoded encoded within the numbers. This allows small distances between representable numbers around zero (high precision) and large distances for very large numbers (high range).

Floating point values are encoded as follows:



- s: 1 bit
- e: 8 bit for 32 bit numbers and 11 bit for 64 bit numbers
- f: 23 bit for 32 bit numbers and 52 bit for 64 bit numbers

The value is calculated by

- $v = (-1)^s \cdot 1.f \cdot 2^{e-K}$ for normalized floating point numbers (e not all bits set and not all bits cleared) and
- $v = (-1)^s \cdot 0.f \cdot 2^{1-K}$ for denormalized numbers (e = 0 and f $\neq$ 0).

K = 127 for 32 bit numbers (float) and K = 1023 for 64 bit numbers (double).

a) Provide the bit pattern for 32 bit floating point number with value 3.625.

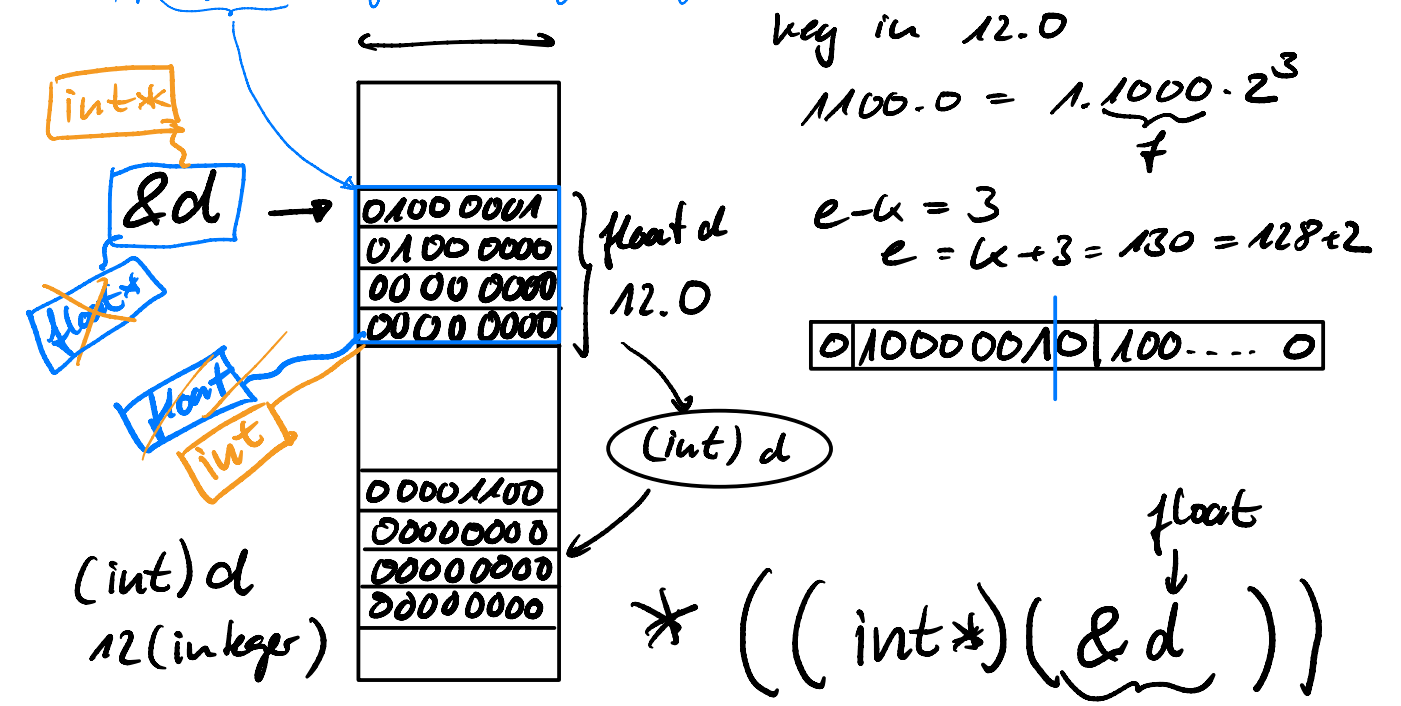
b) Provide a C program that keys in a 64 bit floating point value and prints its representation in binary.

```c
#include <stdio.h>

int main()
{
    float d = 12.;

    printf("float d: %f\n", d);
    printf("Address of float d: %p\n", &d);
    printf("Address of int d:   %p\n", (int*)(&d) );
    printf("Value of int d:     %d\n", *((int*)(&d)) );
}
```

```
float d: 12.000000
Address of float d: 0x16dc2f3fc
Address of int d:   0x16dc2f3fc
Value of int d:     1094713344
→  Code
```

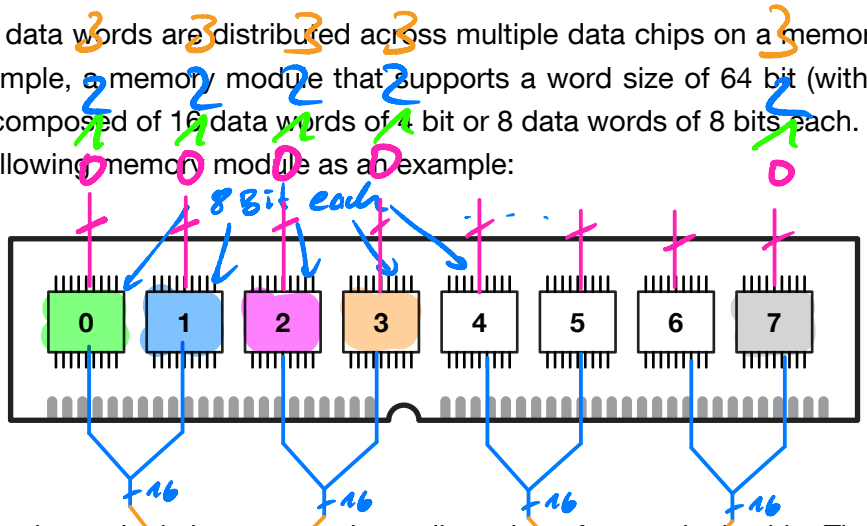
```c
int main()
{
    int i;
    float d;

    printf("d: ");
    scanf("%f", &d);

    // 8 bit: sizeof(d) = 1
    // 16 bit: sizeof(d) = 2
    // 32 bit: sizeof(d) = 4
    // 64 bit: sizeof(d) = 8
    for(i = (sizeof(d) * 8) - 1; i >= 0; i--)
    {
        printf("%d", ((*((int*)(&d))) >> i) & 1 );
    }
    printf("\n");
}
```

```
→  Code gcc -o p p.c
→  Code ./p
d: 12
01000001010000000000000000000000
→  Code
```



Alignment

Multi byte data words are distributed across multiple data chips on a memory module. As an example, a memory module that supports a word size of 64 bit (without parity) might be composed of 16 data words of 4 bit or 8 data words of 8 bits each. See the following memory module as an example:



- Accessing a single byte results in reading a byte from a single chip. The last three address bits can be used to select the chip number 0 ... 7. Address bits 3, 4, 5, ... are used to address an 8 bit memory location within the chip.
- Accessing a two byte dataword (16 bit), address bits 1 and 2 (address bit 0 is not used here) one out of four chip groups (0,1), (2,3), (4,5), and (6,7). The remaining address bits select the address within the chip. The 16 bit data word is then composed of the 2 · 8 bit of the selected chip group.
- Accessing a four byte dataword (32 bit), address bit 2 (address bits 1 and 0 are ignored) selects one of the two chip groups (0,1,2,3) or (4,5,6,7). Address bits 3, 4, 5, ... then select the address within the selected chips. The 32 bit dataword is then composed of the 4 · 8 bit of the selected chip group.
- Accessing an eight byte dataword (64 bit), address bits 0, 1, and 2 are ignored. Adress bits 3, 4, 5, ... are used to select an 8 bit memory location within the chip. The 64 bit dataword is then composed out of the 8 · 8 bit of the selected chip group.

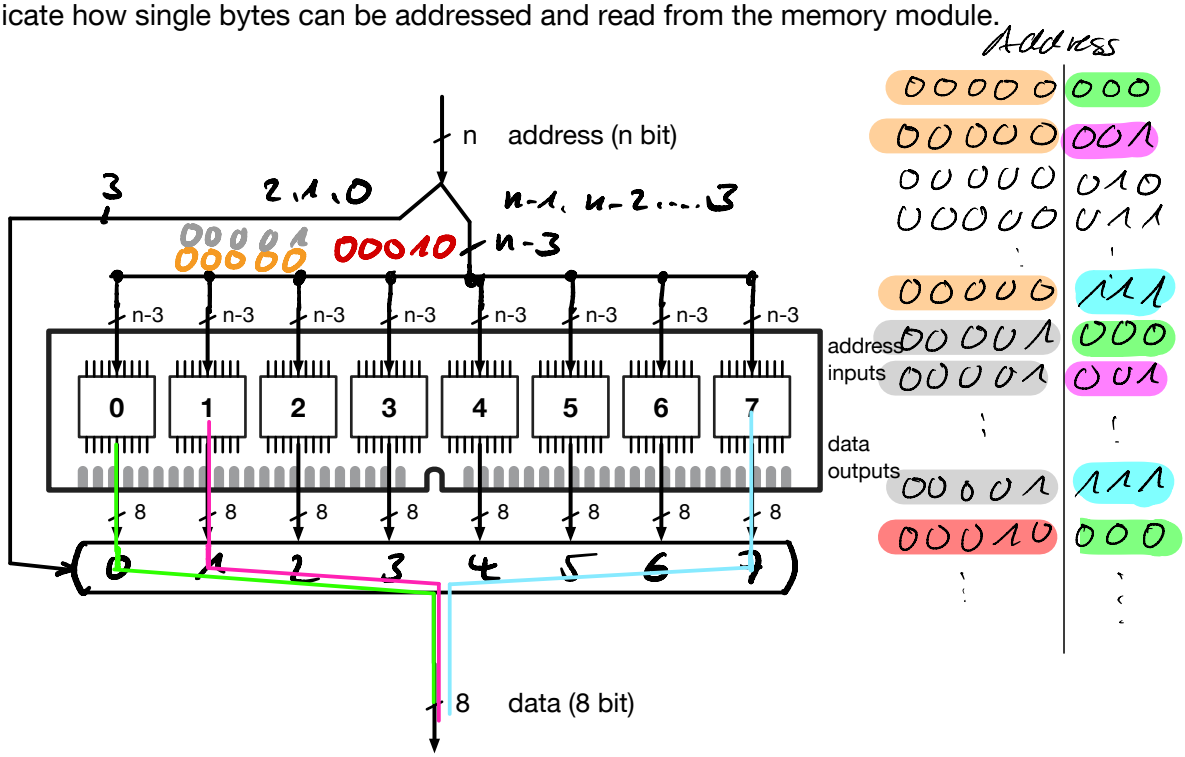
This means that accessing a multi byte dataword merges multiple byte employing fixed hardware wiring. Therefore, an n byte dataword has to reside on a memory address that is a multiple of n.

- Bytes can be placed at any address.
- Wydes can only be placed at an address that is a multiple of 2.
- Tetras can only be placed at an address that is a multiple of 4.
- Octas can only be placed at an address that is a multiple of 8.

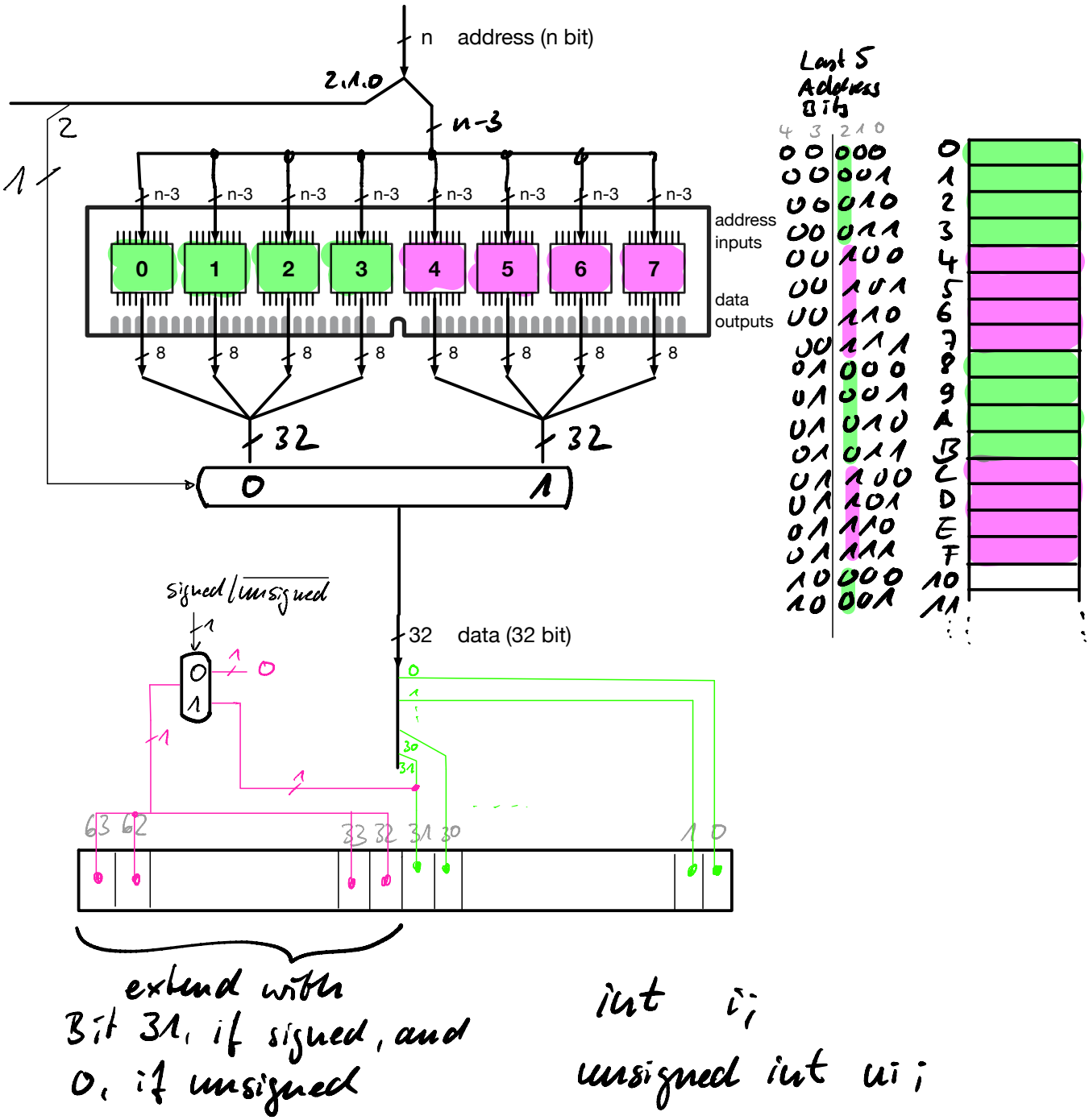
		8 bit	16 bit	32 bit	64 bit
0x0000000000000000		X	X	X	X
0x0000000000000001		X			
0x0000000000000002		X	X		
0x0000000000000003		X			
0x0000000000000004		X	X	X	
0x0000000000000005		X			
0x0000000000000006		X	X		
0x0000000000000007		X			
0x0000000000000008		X	X	X	X
0x0000000000000009		X			
0x000000000000000A		X	X		
⋮	⋮	⋮	⋮	⋮	

Alignment defines the memory locations a multi byte dataword can be placed on. For example, a 32 bit dataword starts at an address that is a multiple of 4 and also utilizes the next 3 bytes (⇒ 4 bytes in total).

a) Indicate how single bytes can be addressed and read from the memory module.



b) Indicate how 32 bit datawords (tetras) can be addressed and read from the memory module.



A data word of size n bytes has to start in memory at an address that is a multiple of n. $n \in \{1, 2, 4, 8, \ldots\}$

Big- and Little Endian

Endianness describes, how the bytes are placed *within* a multi byte dataword.

- Big Endian: The byte with the highest value is addressed (e.g. MIPS).
- Little Endian: The byte with the lowest value is addressed (e.g. x86).

Example: 32 Bit number 0x01234567 at Address 0x0000 0000 0000 0000:

	Big Endian	Little Endian
0x0000000000000000	0x01	0x67
0x0000000000000001	0x23	0x45
0x0000000000000002	0x45	0x23
0x0000000000000003	0x67	0x01

a) Is the endianness a matter for any size of data?

It's only relevant to data words that consist of multiple (≥2) bytes

b) How is 8 bit dataword 0x12 placed at memory address 0x2000 0000 0000 0000 in case of a big- and a little-endian processor?

Adresse	Big Endian	Little Endian
0x2000000000000000	0x12	0x12
0x2000000000000001		

c) How is 16 bit dataword 0x1234 placed at memory address 0x2000 0000 0000 0000 in case of a big- and a little-endian processor?

Adresse	Big Endian	Little Endian
`0x2000000000000000`	0x12	0x34
`0x2000000000000001`	0x34	0x12

d) How is 32 bit dataword 0x12345678 placed at memory address 0x2000 0000 0000 0008 in case of a big- and a little-endian processor?

Adresse	Big Endian	Little Endian
`0x2000000000000008`	0x12	0x78
`0x2000000000000009`	0x34	0x56
`0x200000000000000A`	0x56	0x34
`0x200000000000000B`	0x78	0x12

long int i = 7;

in 64 bits with all leading zeros: 0x0000 0000 1234 5678

e) How is 64 bit dataword 0x12345678 placed at memory address 0x2000 0000 0000 0008 in case of a big- and a little-endian processor?

Adresse	Big Endian	Little Endian
0x2000000000000008	0x00	0x78
0x2000000000000009	0x00	0x56
0x200000000000000A	0x00	0x34
0x200000000000000B	0x00	0x12
0x200000000000000c	0x12	0x00
0x200000000000000D	0x34	0x00
0x200000000000000E	0x56	0x00
0x200000000000000F	0x78	0x00

f) Write a short C program to determine the endianness on your machine.

0x12345678

```c
#include <stdio.h>

int main()
{
    int i = 0x12345678;

    // provide your code here
    printf("%X", *((char*)&i));
}
```

Big endian

&i → 0x12 0x34 0x56 0x78

Little endian

&i → 0x78 0x56 0x34 0x12

```c
#include <stdio.h>

int main()
{
    int i = 0x12345678;

    printf("%#x\n", *((char*)(&i)));
}
```

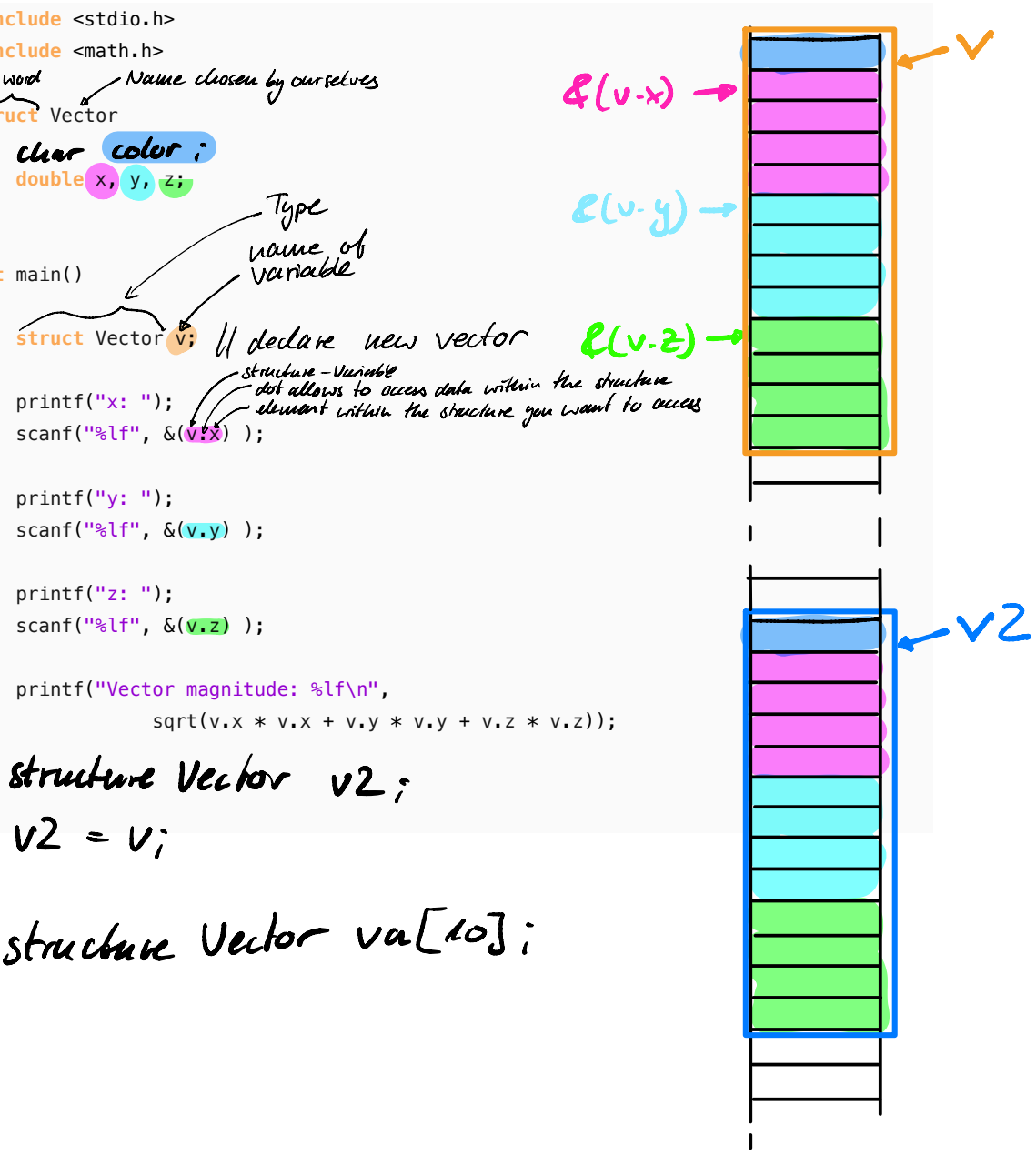
```
→ Code gcc -o p p.c
→ Code ./p
0x78
→ Code
```

My machine is little endian

Combining datatypes

Structures allow to combine multiple elements of data to a new data type. In the C programming language, this is done with keyword `struct`.

```c
#include <stdio.h>
#include <math.h>

struct Vector
{
    double x, y, z;
};

int main()
{
    struct Vector v; // declare new vector

    printf("x: ");
    scanf("%lf", &(v.x) );

    printf("y: ");
    scanf("%lf", &(v.y) );

    printf("z: ");
    scanf("%lf", &(v.z) );

    printf("Vector magnitude: %lf\n",
                sqrt(v.x * v.x + v.y * v.y + v.z * v.z));

}
```



Measuring program execution time

Measuring program execution time in a C program can be performed applying function `clock()` which is declared in time.h.

```c
#include <stdio.h>
#include <time.h>

int main()
{
    long int i, n;
    double d;

    clock_t start, end;
    double t;

    start = clock();

    printf("Enter a number: ");
    scanf("%ld", &n);

    d = 1;
    for(i = 0; i < n; i++)
    {
        d += (double) i;
    }

    end = clock();

    printf("d: %lf\n", d);
    printf("Time: %lf\n", ((double) (end - start)) / CLOCKS_PER_SEC);
}
```

a) Copy the program, compile it and run it.

b) Key in a small number (e.g. 1) and observe the execution time.

c) Key in the same small number, but wait a few seconds to press enter. Observe the execution time. What do you see?

macOS/Linux: user cpu time is measured
windows: Wall-clock time

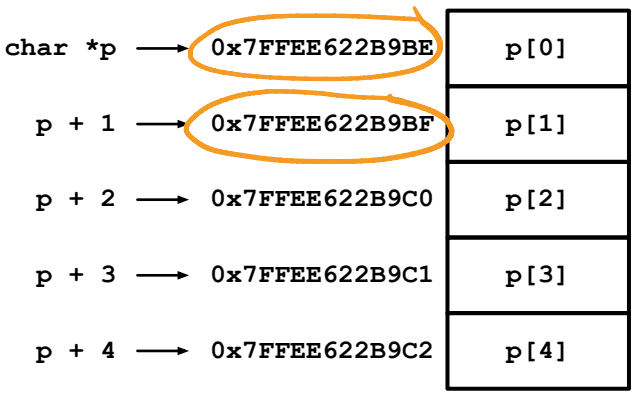
d) Observe execution time for several large numbers. Change i from type `long int` to type `int`. What do you observe?

3.2 Linear data structures

In a linear data structure, the elements are sequentially allocated or connected. Storing elements in a linear data structure you can pass all elements in a single run.

Array

An Array is a linear data structure that stores all elements one after the other in memory. Therefore, the elements can be accessed by adding an offset to a base pointer. To access element i , $0 \leq i \leq$ n-1 of an element of size s, $i \cdot s$ has to be added to the address p of the first element of the array. Note that in C, operators + and - automatically multiply i by s, i.e. adding i to a pointer on C layer adds $i \cdot s$ on machine/assembly layer.



a) Provide code to measure the distance between two adjacent elements in a `char` array. Provide the distance.

```c
#include <stdio.h>

int main()
{

    // provide your code here

}

```

```c
#include <stdio.h>
#include <time.h>

int main()
{
    char s[] = "Hello";

    printf("Distance between two elements: %ld\n", &(s[1]) - &(s[0]) );
}
```

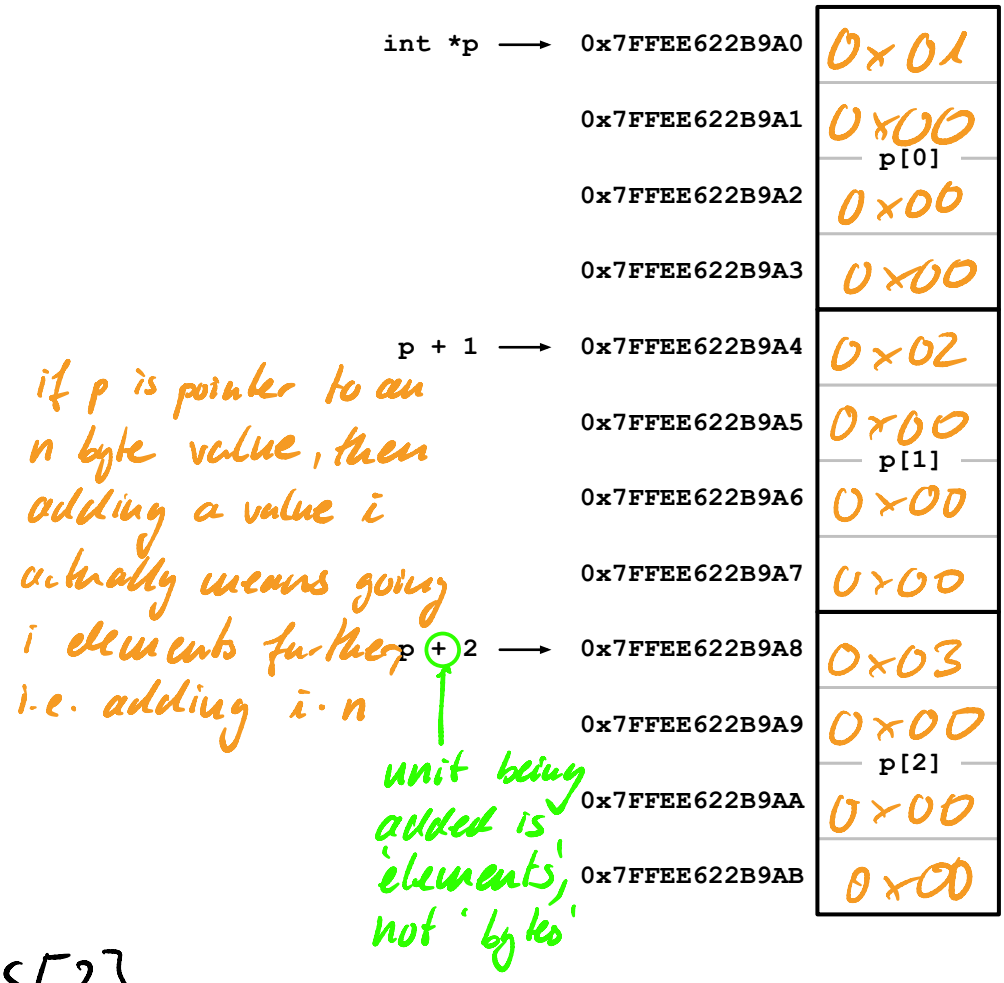
```
→  Code gcc -o p p.c
→  Code ./p
Distance between two elements: 1
→  Code
```

b) Provide code to measure the distance between two adjacent elements in an `int` array. Provide the distance. If you subtract the addresses: How do you have to cast the pointers in order to get the right value?

```c
#include <stdio.h>

int main()
{

    // provide your code here

}

```

```c
int main()
{
    int s[] = {1, 2, 3, 4, 5, 6, 7};

    // here we get a distance of 1, meaning
    // 1 element away (measuring 'elements',
    // not bytes
    printf("Distance between two elements: %ld\n",
        &(s[1]) - &(s[0]) );

    // now 1 element is 1 byte in size
    printf("Distance between two elements: %ld\n",
        ((char*)(&(s[1]))) - ((char*)(&(s[0]))) );

    // using void*
    printf("Distance between two elements: %ld\n",
        ((void*)(&(s[1]))) - ((void*)(&(s[0]))) );
}
```

```
→  Code ./p
Distance between two elements: 1
Distance between two elements: 4
Distance between two elements: 4
→  Code
```



Reading binary array data from file

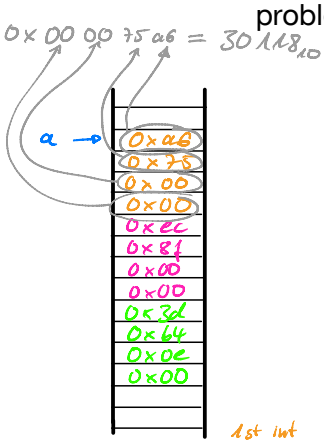
Some of the following programs will read data from binary files. Please arrange your files as follows:

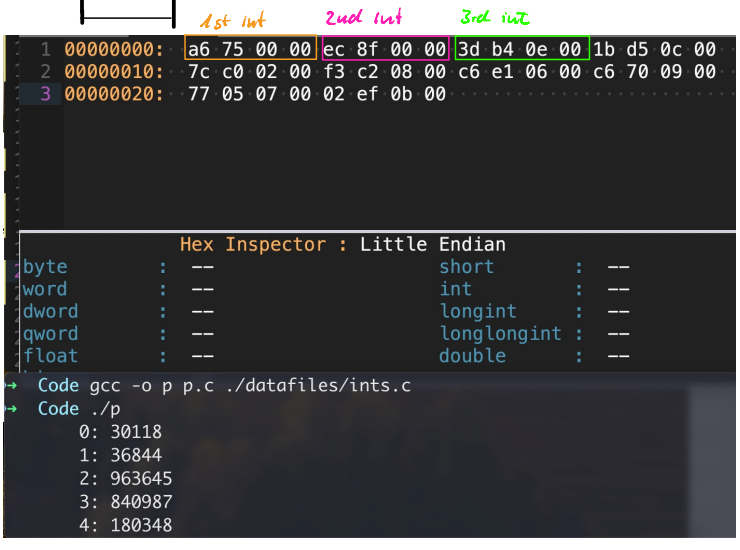
- For each program, provide a separate folder that holds the source code.
- Provide the datafiles folder on the same hierarchy level as the program folders.

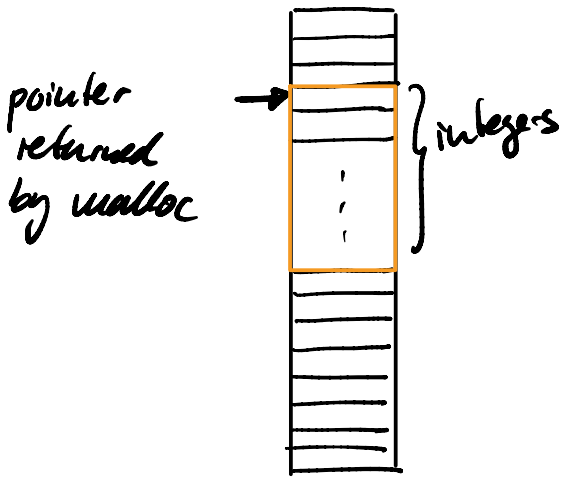
To compile programs that read data from the binary files, select one of the following options:

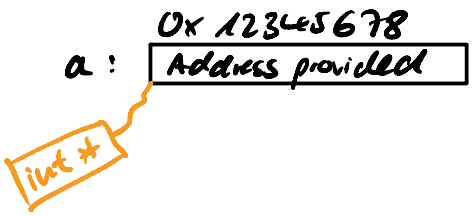
- Include the definition of function `fill_array()` in the source file (e.g. p.c) and compile as before:
  `gcc -o p p.c`
- Include the ints.h header file (`#include "../datafiles/ints.h"`) in your source file and compile your sourcefile (e.g. p.c) together with the ints.c file:
  `gcc -o p ../datafiles/ints.c p.c`
- Include the ints.h header file (`#include "../datafiles/ints.h"`) in your source file and link with libints library (built for macOS only):
  `gcc -lints -L../datafiles -o p p.c`

The following code shows the first option. Use this option if you are encountering problems using options 2 and 3.









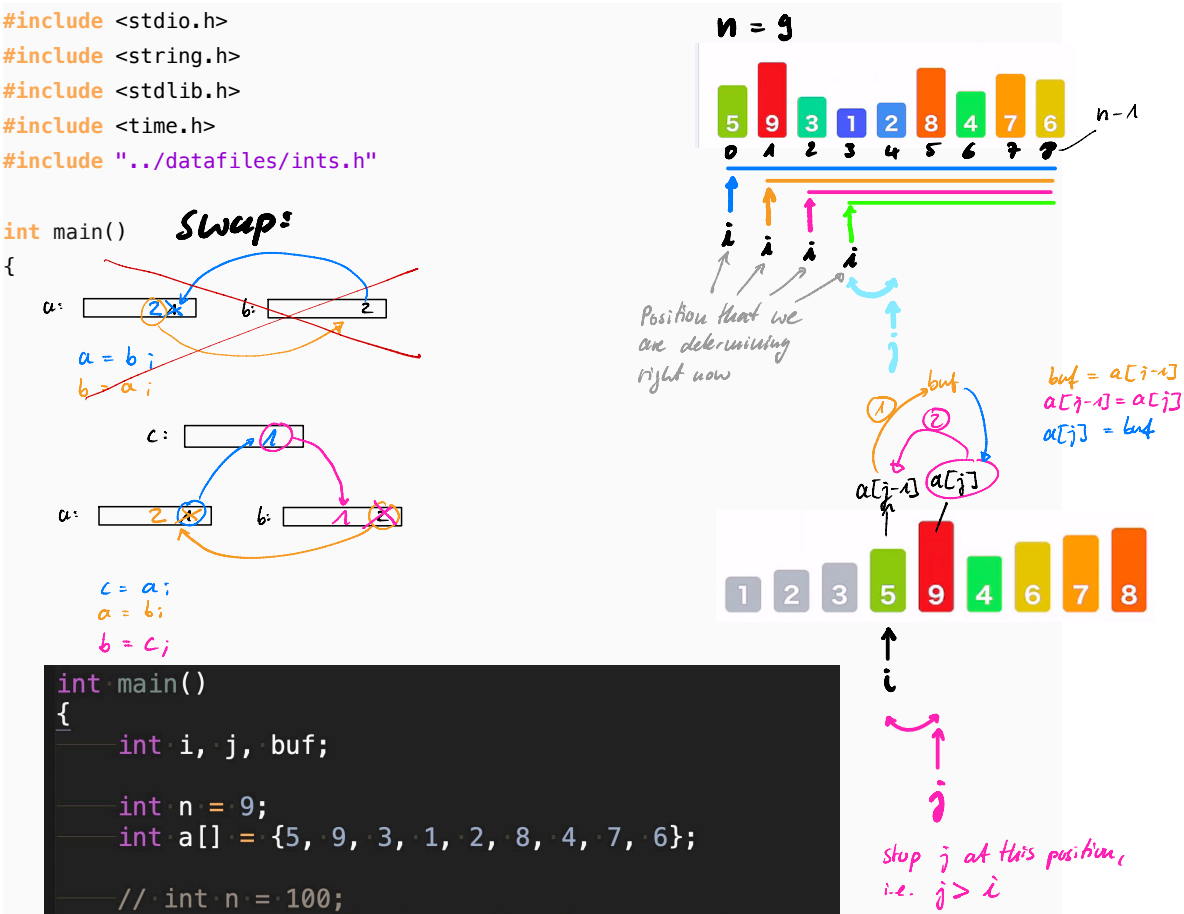
c) Copy the following program and make it run. If you are encountering problems, then the datafiles folder might not be located at the specified path.

```c
#include <stdio.h>      // for printf, scanf, ...
#include <stdlib.h>     // for malloc
#include <string.h>     // for strcpy, strcat

void fill_array(int *a, char *path, int n)
{
    char filename[200];
    char filename2[20];

    strcpy(filename, path);
    sprintf(filename2, "/%d", n);
    strcat(filename, filename2);

    printf("%s\n", filename);

    FILE *file = fopen(filename, "rb");

    if(fread(a, sizeof(int), n, file) != n)
    {
        printf("ERROR - could not read all data!\n");
    }
}


int main()
{
    // number of elements
    int n = 10;

    // allocate memory and store pointer to the first byte in a
    int *a = malloc( n * sizeof(int) );

    // copy contents of data file to allocated memory
    fill_array(a, "../datafiles/ints", n);

    // print data for inspection
    for(int i = 0; i < n; i++)
        printf("%8d: %d\n", i, a[i]);
}
```

Sorting elements in an Array

The bubblesort algorithm

a) Watch the presentation. Then, formulate the bubblesort algorithm as a C program.




```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "../datafiles/ints.h"

int main()
{
    int i, j, buf;

    int n = 9;
    int a[] = {5, 9, 3, 1, 2, 8, 4, 7, 6};

    // int n = 100;
    // int *a = malloc(sizeof(int) * amount);
    // fill_array(a, "../datafiles/ints", amount);

    // provide code for the bubblesort algorithm
    for(i = 0; i < n; i++)
    {
        for(j = n-1; j > i; j--)
        {
            if(a[j] < a[j-1])
            {
                // swap
                buf = a[j-1];
                a[j-1] = a[j];
                a[j] = buf;
            }
        }
    }

    // print the integers
    for(i = 0; i < n; i++)
        printf("%8d: %d\n", i, a[i]);

}
```

```
n: 10
Time: 5.000000 µs
```

```
n: 100
Time: 57.000000 µs
```

```
n: 1000
Time: 1161.0 µs
```

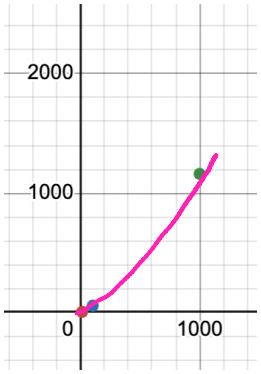
```
n: 10000
Time: 159887.0 µs
```

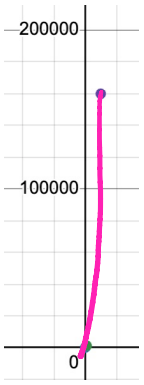
```
n: 100000
Time: 19688950.0 µs
```

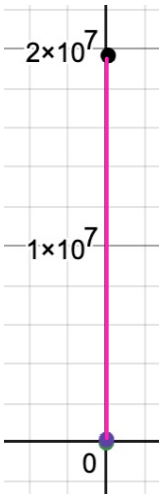

b) Run the algorithm with n = 10, 100, 1000, 10000, and 100000 numbers and measure execution time. Compare n and runtime in a table.

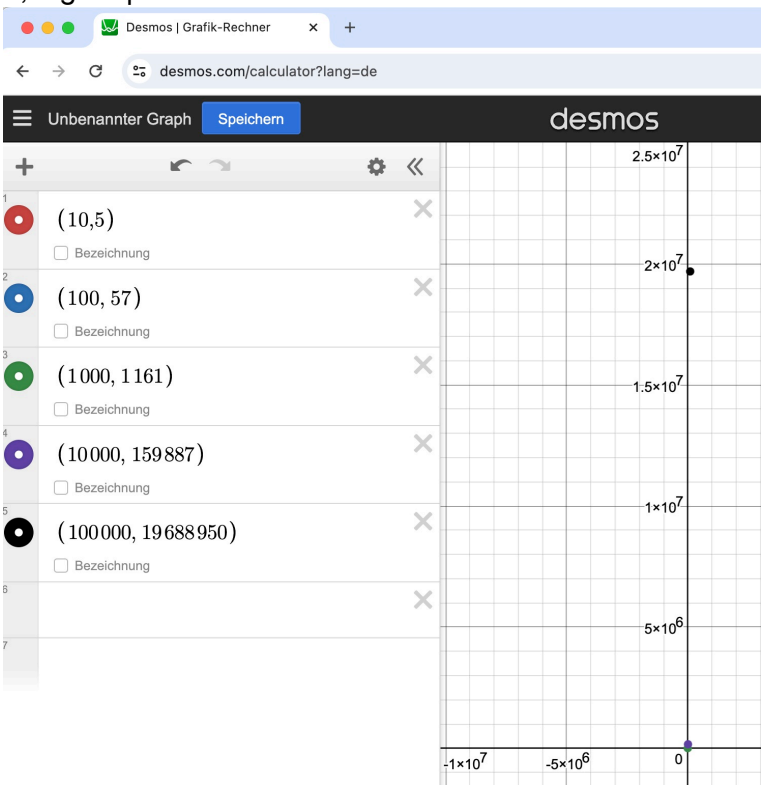
n	time (µs)
10	5
100	57
1000	1161
10000	159887
100000	19 688 950

c) Provide a graph that plots execution time over n. Look for an appropriate tool or web framework to perform such a plot, e.g. https://www.desmos.com/calculator/.









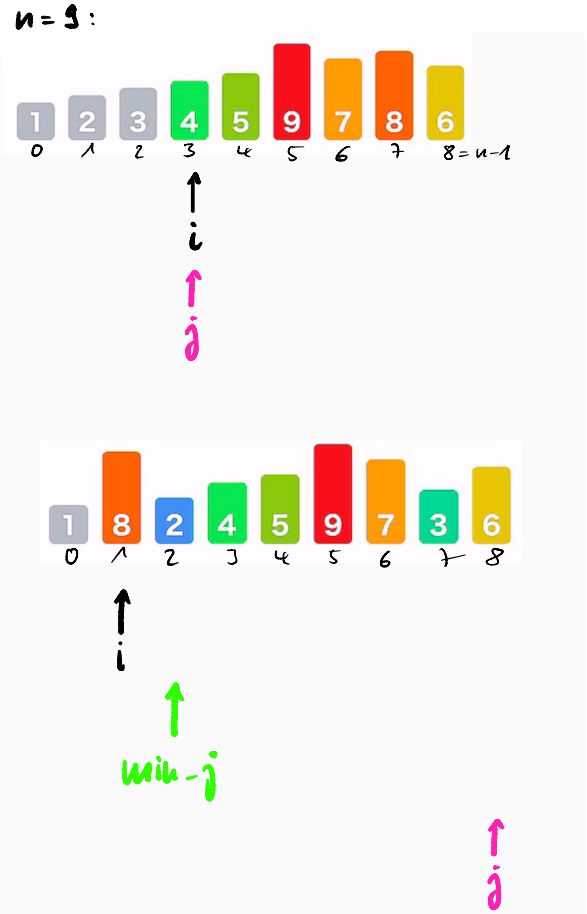
d) Estimate how the maximum runtime dependes on the amount of n.

$O(n^2)$

The selectionsort algorithm

e) Watch the presentation. Then, formulate the selectionsort algorithms as a C program.

```c
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include "../datafiles/ints.h"

int main()
{




}
```

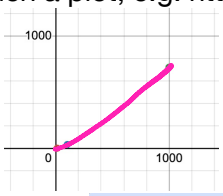


```
n: 10
Time: 4.0 µs
```
```
n: 100
Time: 37.0 µs
```
```
n: 1000
Time: 724.0 µs
```
```
n: 10000
Time: 65849.0 µs
```
```
n: 100000
Time: 6447235.0 µs
```

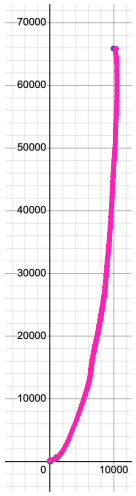

f) Run the algorithm with n = 10, 100, 1000, 10000, and 100000 numbers and measure execution time. Compare n and runtime in a table.

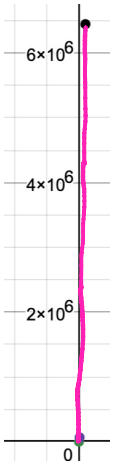
n	time
10	4
100	37
1000	724
10 000	65 849
100 000	6 447 235

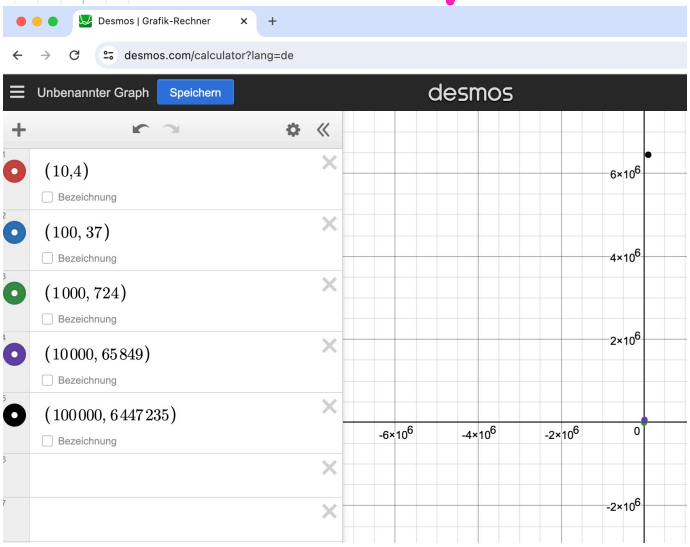
g) Provide a graph that plots execution time over n. Look for an appropriate tool or web framework to perform such a plot, e.g. https://www.desmos.com/calculator/.



better than bubblesort, but not good for large values of n







h) Estimate how the maximum runtime dependes on the amount of n.

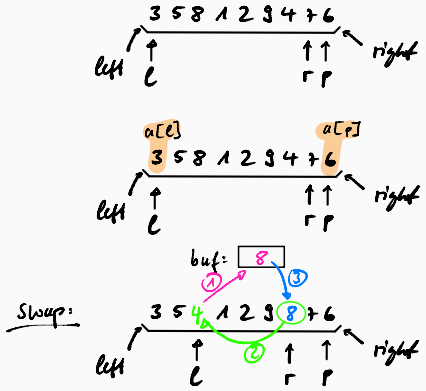
$O(n^2)$

The quicksort algorithm

If you have to sort a larger amount of numbers, the quicksort algorithm is much faster.

i) Watch the presentation. Provide the quicksort function.

```c
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include "../datafiles/ints.h"


void quicksort(int a[], int left, int right)
{
    int l, r, p;

    l = left;
    p = right;
    r = right-1;
```



```
 7  void quicksort(int a[], int left, int right)
 8  {
 9      int l, r, p, buf;
10
11      printf("Calling quicksort(a, %d, %d)\n", left, right);
12
13      l = left;
14      r = right - 1;
15      p = right;
16
17      do
18      {
19          // shift l to the right until you find an
20          // element larger or equal the pivot element
21          while( a[l] < a[p] ) l++;
22
23          // shift r to the left
24          while( a[r] >= a[p] && l < r ) r--;
25
26          // swap l and r
27          if(l < r)
28          {
29              buf = a[l];
30              a[l] = a[r];
31              a[r] = buf;
32          }
33
34          // swap meeting point (l == r) and p
35          if(l == r)
36          {
37              buf = a[r];
38              a[r] = a[p];
39              a[p] = buf;
40
41              p = l;      // pivot becomes the meeting point
42          }
43      } while(l < r);
44
45      // call quicksort recursively for the left part
46      // if there are at least two elements still to be sorted
47      if(left < p - 1)
48          quicksort(a, left, p - 1);
49
50      // call quicksort recursively for the right part
51      // if there are at least two elements still to be sorted
52      if(p + 1 < right)
53          quicksort(a, p + 1, right);
54  }
```

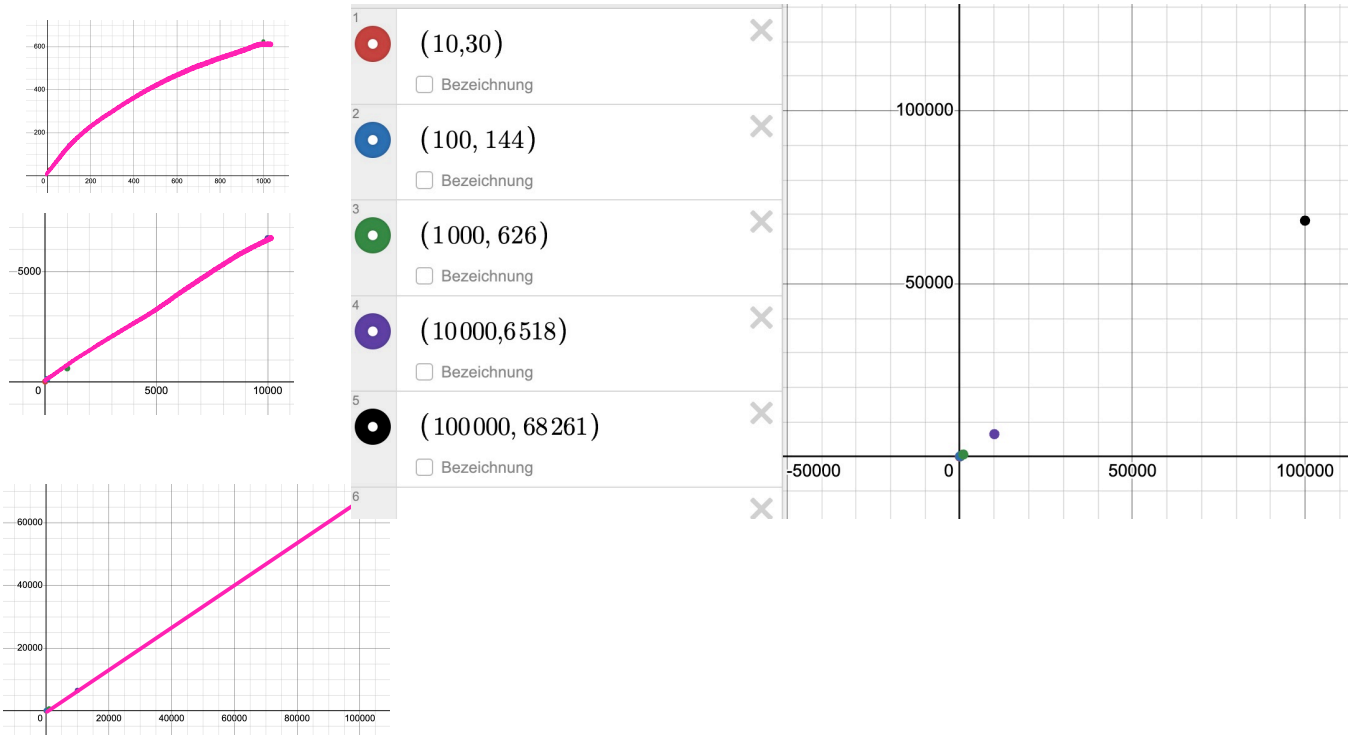
```
}
```

```
93
94  int main()
95  {
96      int n = 1000;
97      int *a = malloc( n * sizeof(int) );
98
99      clock_t start, end;
100
101     double t;
102
103     fill_array(a, "../datafiles/ints", n);
104
105     start = clock();
106
107     quicksort(a, 0, n-1);   // do the sorting
108
109     end = clock();
110
111     for(int i = 0; i < n; i++)  // print all sorted elements
112     {
113         if(i < (n - 1) && a[i] > a[i+1])
114         {
115             printf("Error at i = %d: %d > %d\n", i, a[i], a[i+1]);
116             break;
117         }
118         printf(" %d ", a[i]);
119     }
120
121     printf("\n\n");
122     printf("n: %d\n", n);
123     printf("Time: %lf µs\n", (((double) (end - start)) / CLOCKS_PER_SEC) * 1000000);
124     printf("\n");
125 }
```

j) Run the algorithm with n = 10, 100, 1000, 10000, and 100000 numbers and measure execution time. Compare n and runtime in a table.

n	time [µs]
10	30
100	144
1000	626
10 000	6518
100 000	68261

k) Provide a graph that plots execution time over n. Look for an appropriate tool or web framework to perform such a plot, e.g. https://www.desmos.com/calculator/.



l) Estimate how the maximum runtime dependes on the amount of n.

$O(n \cdot \log(n))$

4 Linked lists

Memory blocks that are allocated at different points in time (i.e. multiple calls to *malloc*) have no address relation to each other.
In order to manage various memory locations, you may store their pointers in an array.
Another way to manage multiple memory locations is by using *linked lists*.
In linked lists, the list is implemented by

- storing a pointer to the first and/or last element in the list, and
- making each element within the list point to the next element in the list, and
- making the last element in the list point to *NULL*.

If each element points to the next element only, the list is called to be a *singly linked list*.
If each element points to the *next and previous* element, the list is called to be a *doubly linked list*.

```c
struct Song * get_song()
{
    struct Song *p;

    // allocate memory
    p = malloc( sizeof(struct Song) );

    // key in Name and duration
    printf("Name: ");
    scanf("%s", p->name ); // (*p).name is iden-
    printf("Duration: ");    // tical to p->name
    scanf("%d", &(p->duration) );

    // if the user entered duration 0, we quit
    if(p->duration == 0)
    {
        free(p);
        return NULL;
    }

    return p;
}
```

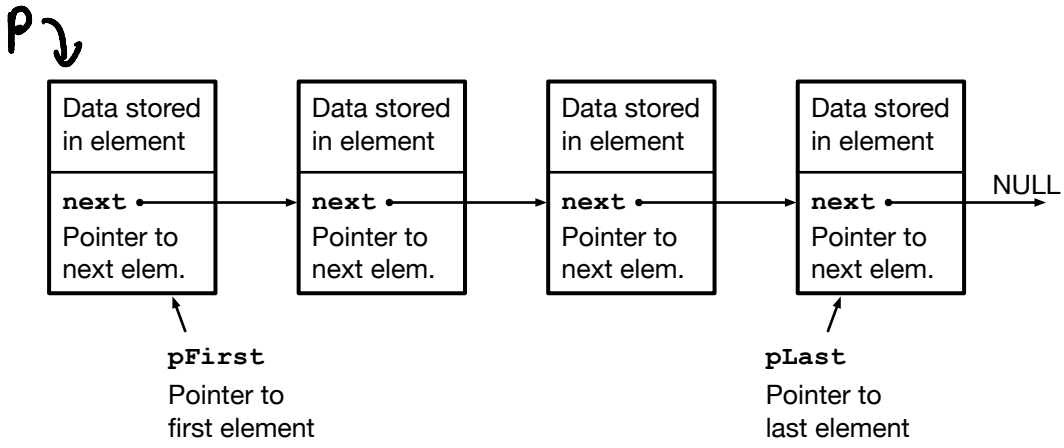
```c
int main()
{
    struct Song *p;

    // List is empty
    pFirst = pLast = NULL;

    while(1)
    {
        if( (p = get_song()) )
        {
            // add_song_at_the_end(p);
            add_song_at_the_beginning(p);
        }
        else break;
    }
    print_all_songs();
}
```

```c
void print_all_songs()
{
    struct Song *p;

    printf("\n");

    for(p = pFirst; p; p = p->next)
    {
        printf("%s (%02d:%02d)\n", p->name, p->duration/60, p->duration%60);
    }
    printf("\n");
}
```

4.1 Singly linked list

```
7  struct Song
8  {
9      char name[12];       // 12 bytes for the name
10     int duration;        // 4 bytes for the duration
11
12     // pointer to next Song
13     struct Song *next;   // 8 bytes to store the address
14 };
15
16 struct Song *pFirst, *pLast;
```

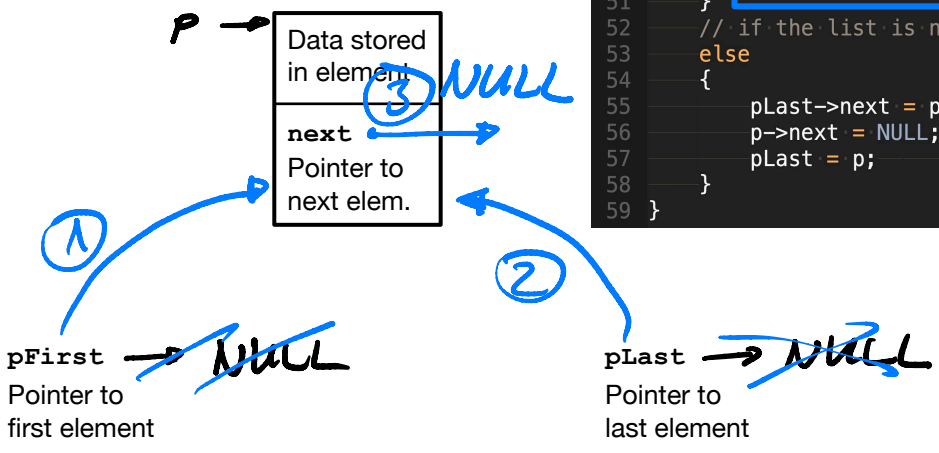
The following figure shows a singly linked list with four elements.



Adding an element p to an empty list

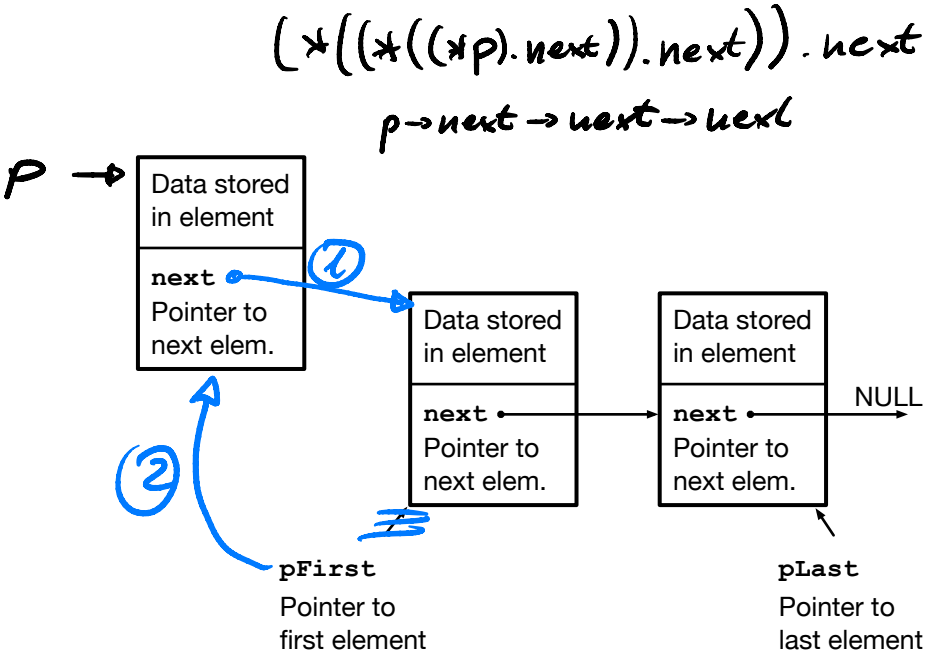
- (Validate that the list is empty by verifying that `pFirst` is equal to `NULL`.)
- Make `pFirst` and `pLast` both point to `p`: `pFirst = pLast = p`.
- Make the next-pointer of `p` point to `NULL`.

```
43 void add_song_at_the_end(struct Song *p)
44 {
45     // if the list is empty
46     if(pFirst == NULL)
47     {
48         pFirst = p;       // (1)
49         pLast = p;        // (2)
50         p->next = NULL; // (3)
51     }
52     // if the list is not empty
53     else
54     {
55         pLast->next = p;     // (1); identica
56         p->next = NULL;      // (2)
57         pLast = p;           // (3)
58     }
59 }
```



Adding a new element p at the beginning of a non empty list

- (Verify that there is at least one element stored in the list by checking that `pFirst != NULL` evaluates to true.)
- Make the next-pointer of the new element point to the (former) first element: `p->next = pFirst`.
- Make the pointer to the first element point to the new element: `pFirst = p`.



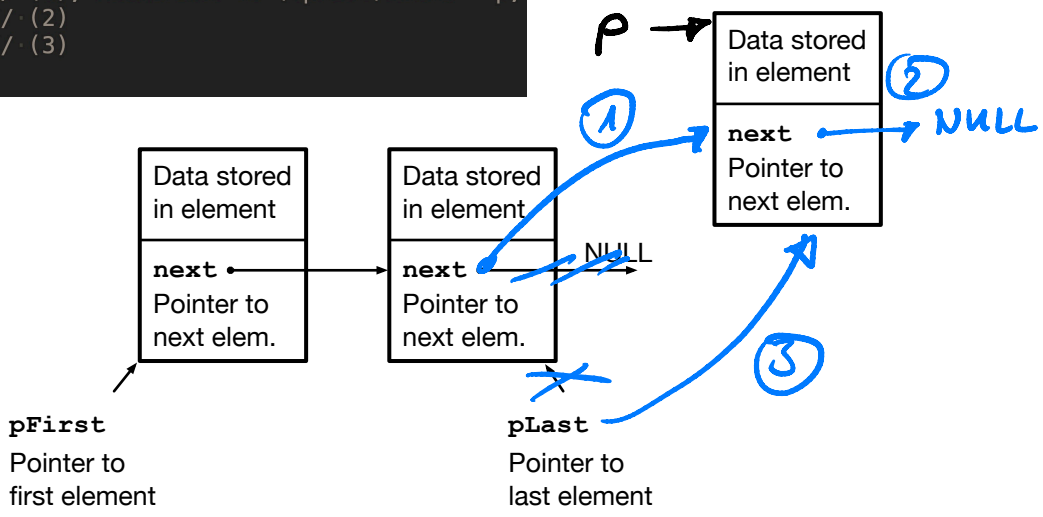

```
Name: Song1
Duration: 1
Name: Song12
Duration: 12
Name: Song123
Duration: 123
Name: Song234
Duration: 234
Name: Song0
Duration: 0

Song234 (03:54)
Song123 (02:03)
Song12 (00:12)
Song1 (00:01)

→  Code
```

```
61 void add_song_at_the_beginning(struct Song *p)
62 {
63     // if the list is empty
64     if(pFirst == NULL)
65     {
66         pFirst = p;       // (1)
67         pLast = p;        // (2)
68         p->next = NULL; // (3)
69     }
70     // if the list is not empty
71     else
72     {
73         p->next = pFirst;    // (1)
74         pFirst = p;          // (2)
75     }
76 }
```

Adding a new element p at the end of a non empty list

- (Verify that there is at least one element stored in the list by checking that `pFirst != NULL` evaluates to true.)
- Make the next-pointer of the (former) last element point to the new element: `pLast->next = p`.
- Make the next-pointer of the new element point to `NULL`: `p->next = NULL`.
- Update the pointer to the last element to point to the new element: `pLast = p`.

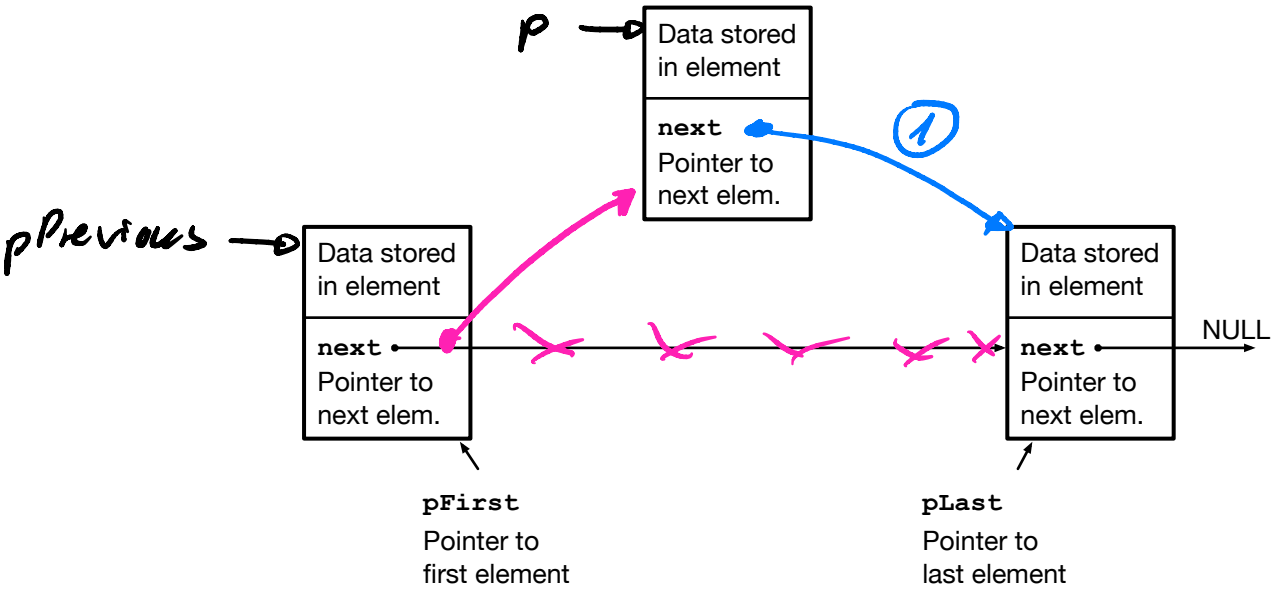
```
// if the list is not empty
else
{
    pLast->next = p;     // (1); identical to (*pLast).next = p;
    p->next = NULL;      // (2)
    pLast = p;           // (3)
}
```




```
43 void add_song_at_the_end(struct Song *p)
44 {
45     // if the list is empty
46     if(pFirst == NULL)
47     {
48         pFirst = p;      // (1)
49         pLast = p;       // (2)
50         p->next = NULL; // (3)
51     }
52     // if the list is not empty
53     else
54     {
55         pLast->next = p;     // (1); identical to (*pLast).next = p;
56         p->next = NULL;      // (2)
57         pLast = p;           // (3)
58     }
59 }
```

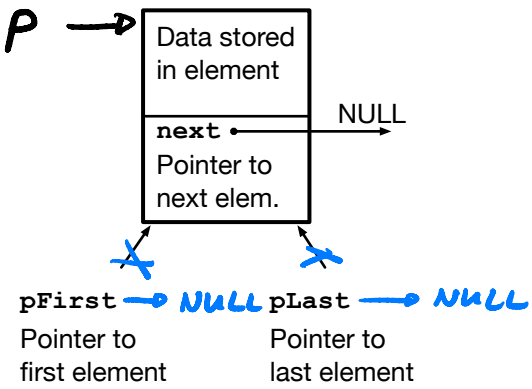
Adding a new element p in the middle, after an element pPrevious

- (Check that there are at least two elements in the list by validating that the pointer to the first and the pointer to the last element differ, i.e. `pFirst != pLast` evaluates to true.)
- (Validate that you do not add the element at the beginning of the list by ensuring that `pPrevious != pFirst` evaluates to true.)
- (Check that you do not append the element at the end of the list by validating that `pPrevious != pLast` evaluates to true.)
- Make the next-pointer of the new element p point to its successor: `p->next = pPrevious->next;`
- Make the next-pointer of the predecessor (pPrevious) point to the new element: `pPrevious->next = p;`



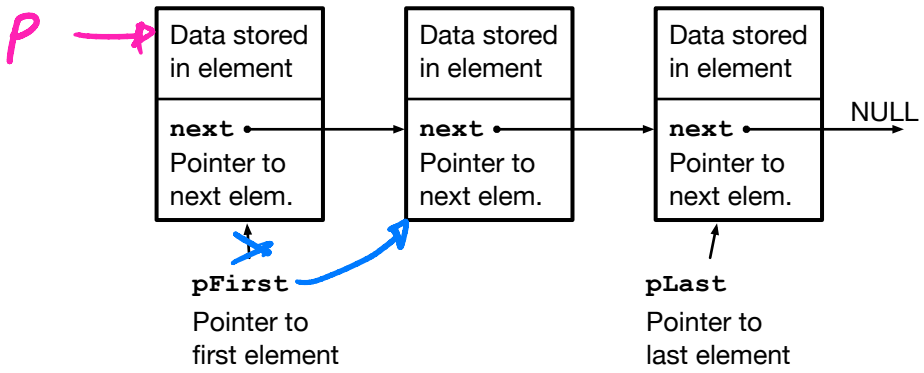
Removing the only element from a list

- (Verify that there is only one element in the list by checking `pFirst == pLast` equals to true.)
- Save a pointer to the element to remove: `p = pFirst`.
- Assign `NULL` to the pointer to the first and the pointer to the last element: `pFirst = pLast = NULL`.
- Return `p`



Removing the first element from a list of at least two elements

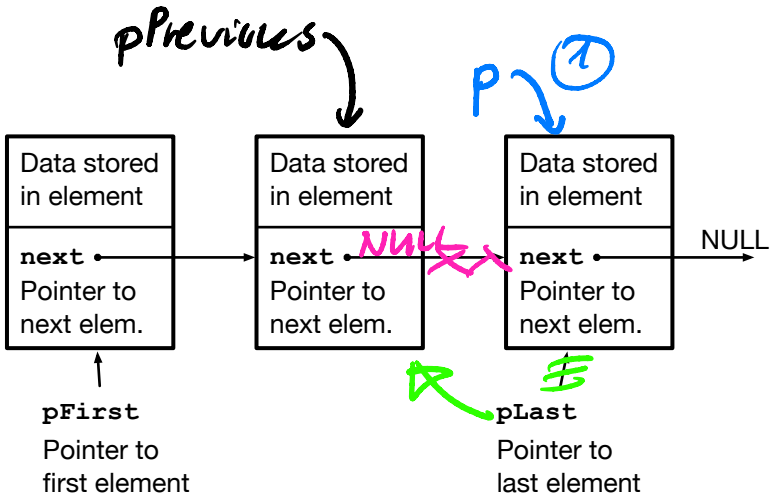
- (Verify that there is more than one element in the list by checking that `pFirst != pLast` equals to true.)
- Save a pointer to the first element: `p = pFirst`.
- Make the pointer to the first element point to the (former) second element: `pFirst = pFirst->next`
- Return `p`

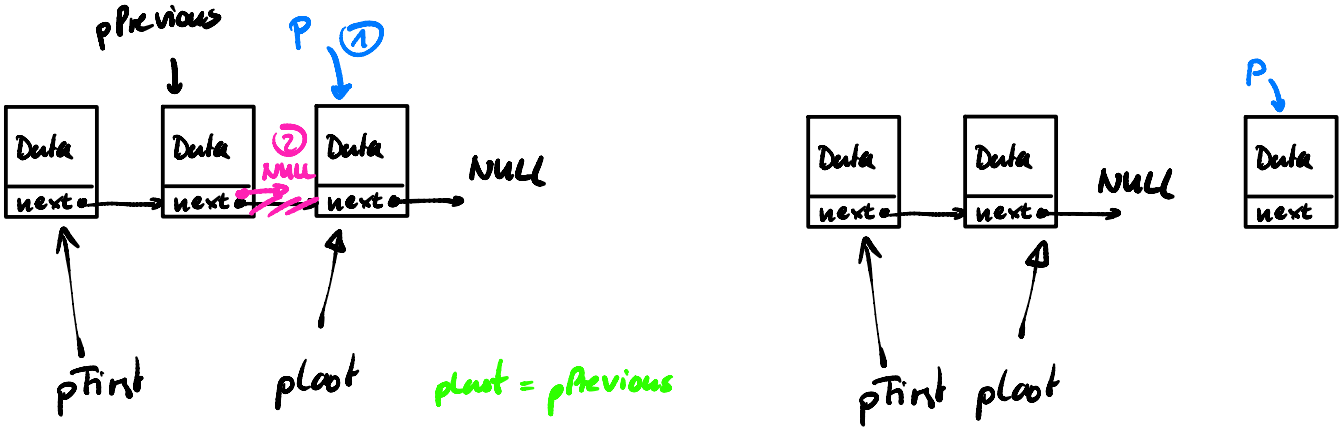


Removing the last element from a list of at least two elements

In order to remove the last element from the list, you have to modify the next-pointer of its predecessor. Therefore, when iterating a singly linked list from the beginning to the end, you would check at element *i* if you have to remove element *i+1*. Therefore, we assume to have a pointer *pPrevious* to the last but one element in the list.

- Save a pointer to the element to remove: `p = pPrevious->next`.
- Make the next-pointer of the last but one element point to NULL: `pPrevious->next = NULL`.
- Update the pointer to the last element: `pLast = pPrevious`.
- Return `p`

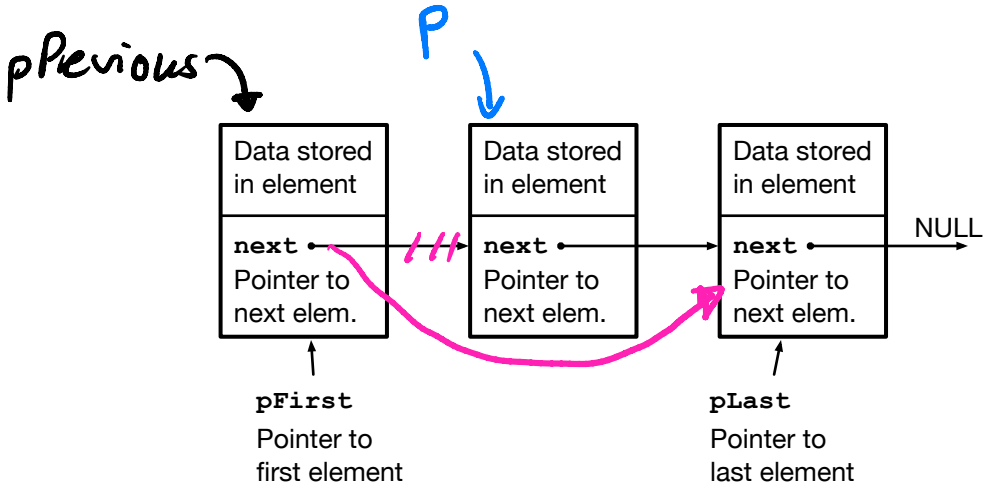




Removing any element between the first and the last element

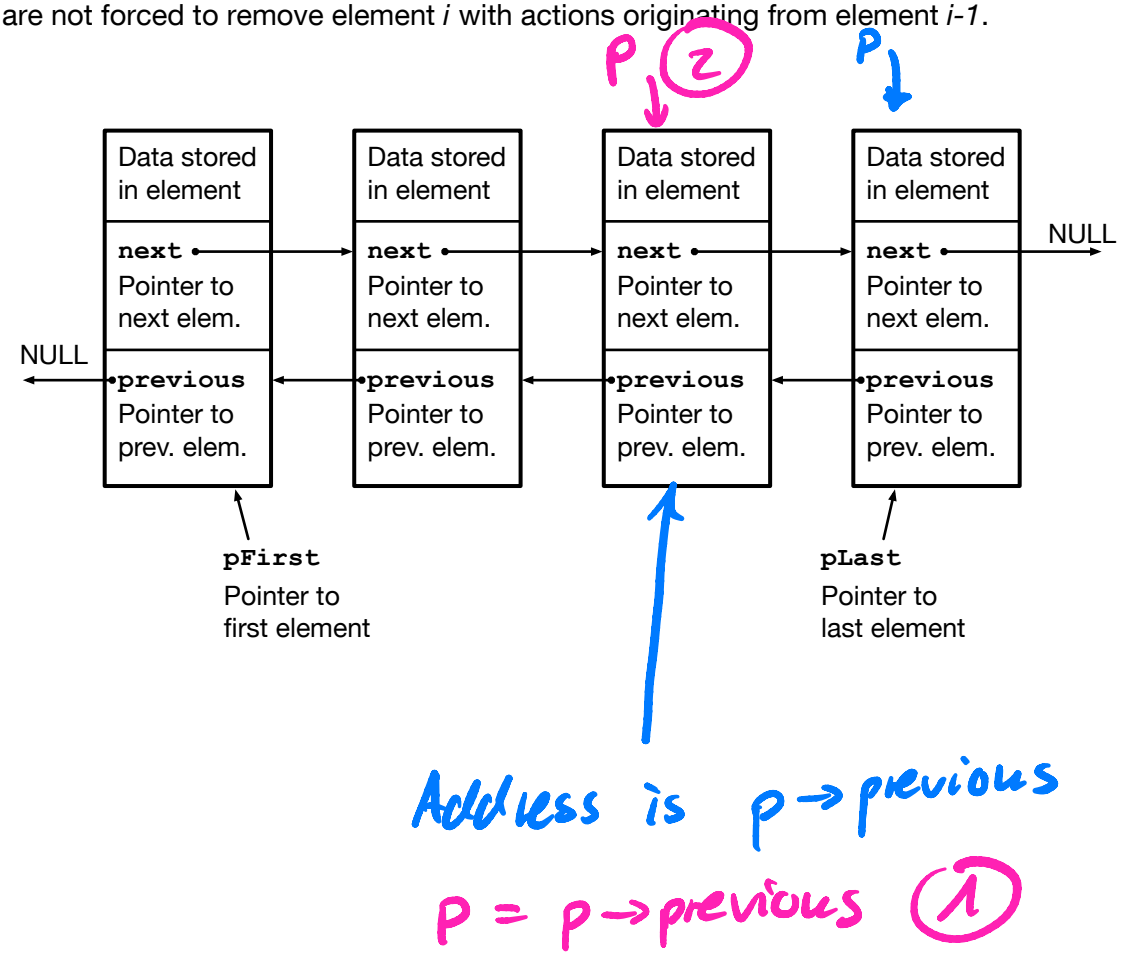
Assume you have to remove the element following pPrevious.

- Save a pointer to the element to remove: `p = pPrevious->next`.
- Make the next-pointer of element pPrevious point to the element after the next: `pPrevious->next = pPrevious->next->next`.
- Return `p`.



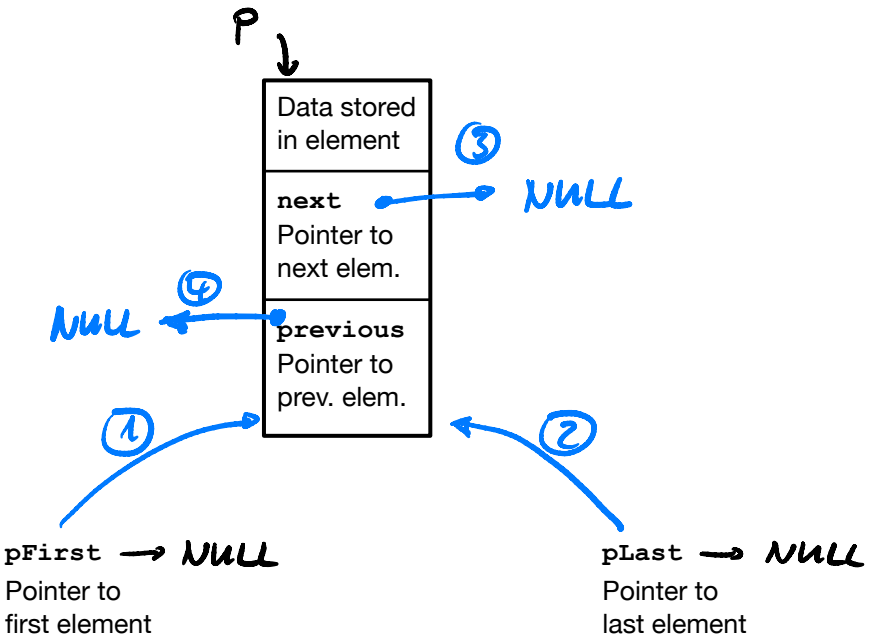
4.2 Doubly linked list

In a doubly linked list, each element does not only store a pointer to the next element, but also a pointer to the previous element. This way, you can step through the list in both ways, forward and backward. Further, you can remove element *i* from position *i*, and you are not forced to remove element *i* with actions originating from element *i-1*.



Adding an element p to an empty list

- (Check if the list is empty by checking if `pFirst` is equal to `NULL`.)
- Make `pFirst` and `pLast` both point to `p`: `pFirst = pLast = p`.
- Make p's pointer to the next and to the previous element point to NULL: `p->next = p->previous = NULL`.

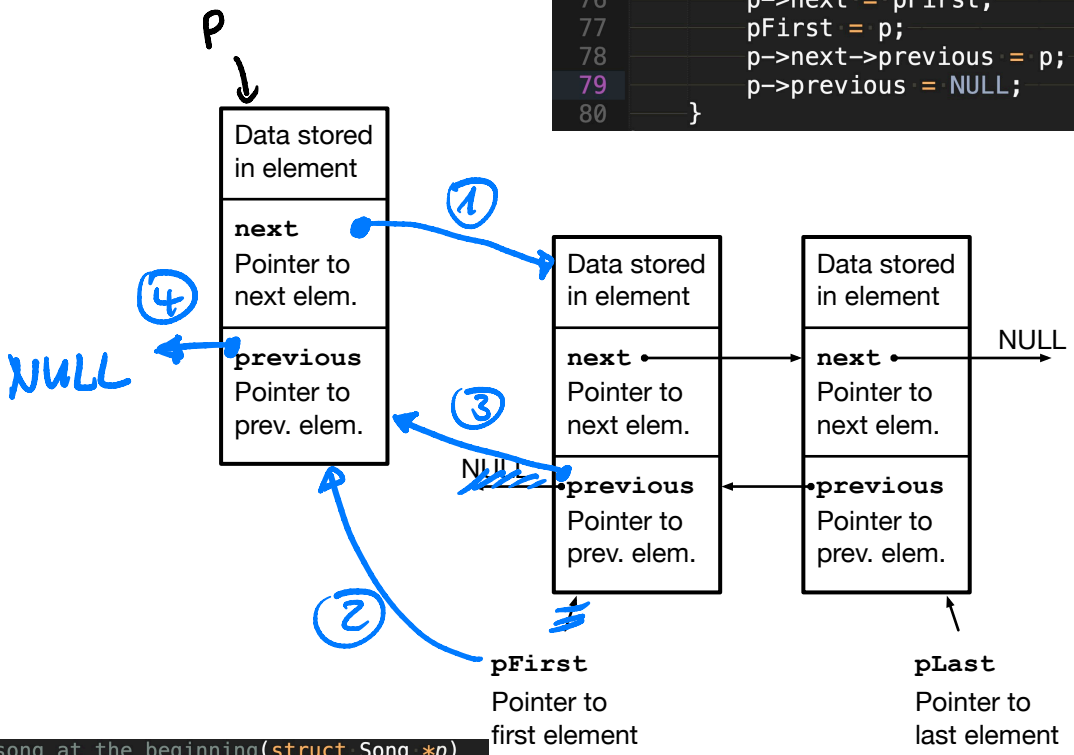



```
64      // if the list is empty
65      if(pFirst == NULL)
66      {
67          pFirst = p;          // (1)
68          pLast = p;           // (2)
69          p->next = NULL;      // (3)
70          p->previous = NULL; // (4)
71
72      }
```

Adding an element p at the beginning of a non empty list

- (Verify that there is at least one element stored in the list by checking that `pFirst != NULL` evaluates to true.)
- Make the next-pointer of the new element point to the (former) first element: `p->next = pFirst`.
- Make the pointer to the first element point to the new element: `pFirst = p`.
- Make the previous-pointer of the former first element point to the new element: `p->next->previous = p`.
- Make the previous-pointer of the new element point to `NULL`: `p->previous = NULL`.

```
73      // if the list is not empty
74      else
75      {
76          p->next = pFirst;          // (1)
77          pFirst = p;                // (2)
78          p->next->previous = p;     // (3)
79          p->previous = NULL;        // (4)
80      }
```

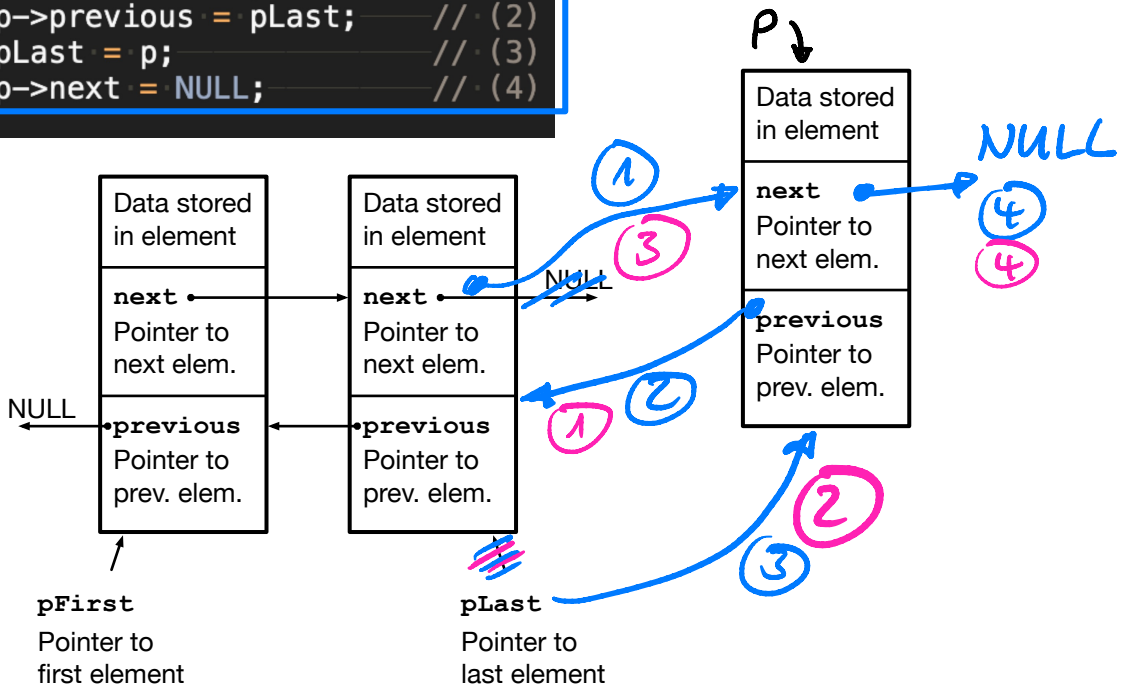



```
62  void add_song_at_the_beginning(struct Song *p)
63  {
64      // if the list is empty
65      if(pFirst == NULL)
66      {
67          pFirst = p;          // (1)
68          pLast = p;           // (2)
69          p->next = NULL;      // (3)
70          p->previous = NULL; // (4)
71
72      }
73      // if the list is not empty
74      else
75      {
76          p->next = pFirst;          // (1)
77          pFirst = p;                // (2)
78          p->next->previous = p;     // (3)
79          p->previous = NULL;        // (4)
80      }
81  }
```

Adding a new element p at the end of a non empty list

- (Verify that there is at least one element stored in the list by checking that `pFirst != NULL` evaluates to true.)
- Make the next-pointer of the (former) last element point to the new element: `pLast->next = p`.
- Make the previous-pointer of the new element point to the (former) last element: `p->previous = pLast`.
- Update the pointer to the last element to point to the new element: `pLast = p`.
- Make the next-pointer of the new element point to `NULL`: `p->next = NULL`.

```
54      // if the list is not empty
55      else
56      {
57          pLast->next = p;          // (1)
58          p->previous = pLast;      // (2)
59          pLast = p;                // (3)
60          p->next = NULL;           // (4)
61      }
```

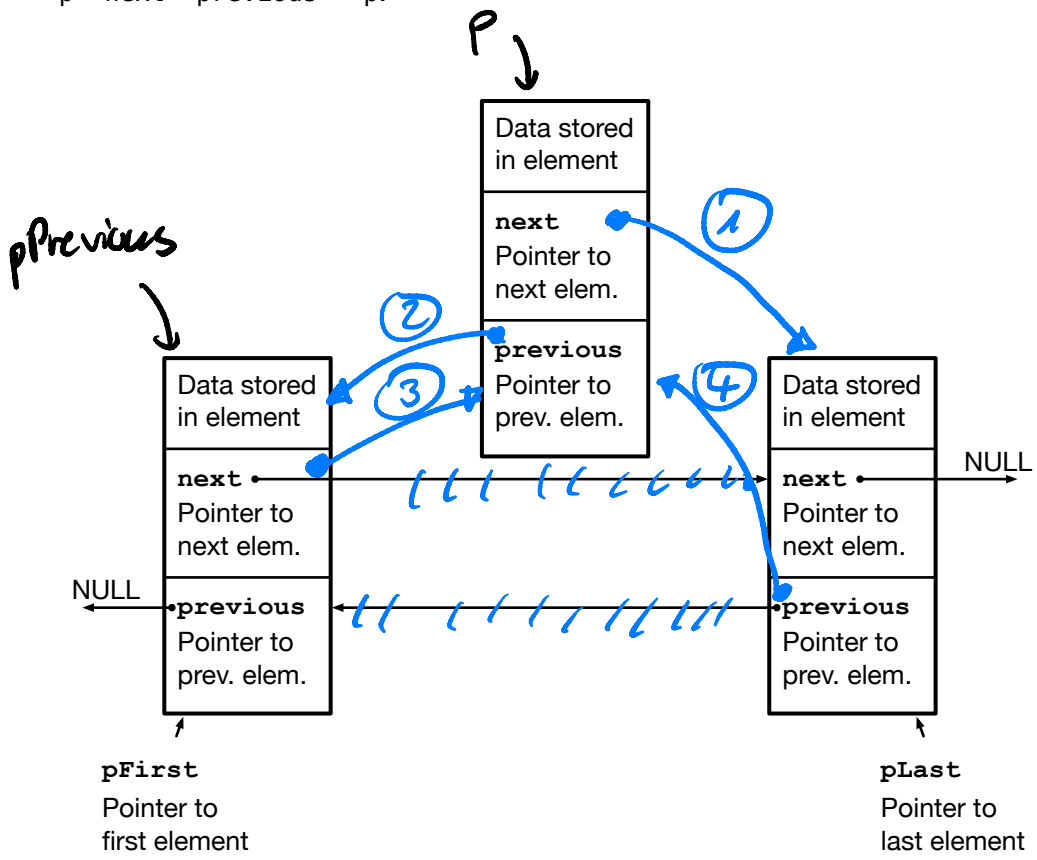



```
44  void add_song_at_the_end(struct Song *p)
45  {
46      // if the list is empty
47      if(pFirst == NULL)
48      {
49          pFirst = p;           // (1)
50          pLast = p;            // (2)
51          p->next = NULL;       // (3)
52          p->previous = NULL;   // (4)
53      }
54      // if the list is not empty
55      else
56      {
57          pLast->next = p;          // (1)
58          p->previous = pLast;      // (2)
59          pLast = p;                // (3)
60          p->next = NULL;           // (4)
61      }
62  }
```

```
55      // if the list is not empty
56      else
57      {
58          p->previous = pLast;      // (1)
59          pLast = p;                // (2)
60          p->previous->next = p;    // (3)
61          p->next = NULL;           // (4)
62      }
```

Adding a new element p in the middle, after an element pPrevious

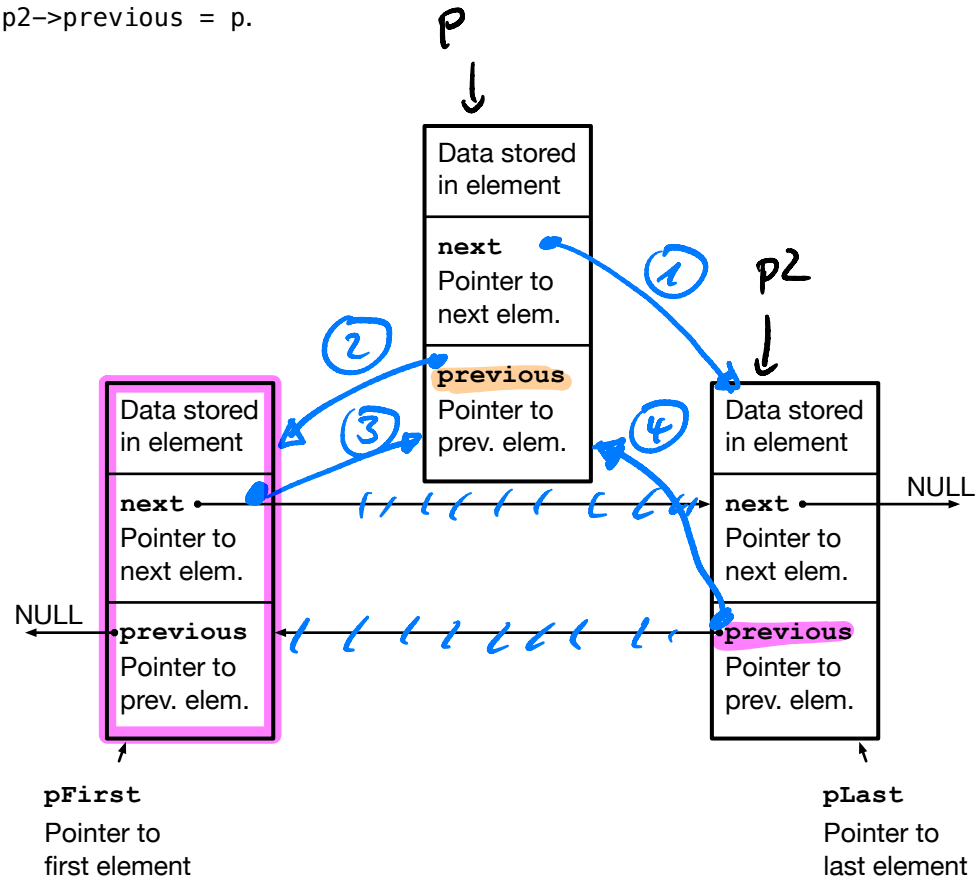
- (Check that there are at least two elements in the list by validating that the pointer to the first and the pointer to the last element differ, i.e. `pFirst != pLast` evaluates to true.)
- (Validate that you do not add the element at the beginning of the list by ensuring that `pPrevious != pFirst` evaluates to true.)
- (Verify that you do not append the element at the end of the list by validating that `pPrevious != pLast` evaluates to true.)
- Make the next-pointer of the new element p point to its successor: `p->next = pPrevious->next`.
- Make the previous-pointer of the new element p point to its predecessor: `p->previous = pPrevious`.
- Make the next-pointer of the predecessor point to the new element: `pPrevious->next = p`.
- Make the previous-pointer of the successor point to the new element: `p->next->previous = p`.




```
149 void add_element_in_the_middle_after_element(struct Song *p, struct Song *pPrevious)
150 {
151     if(pFirst == NULL || pFirst == pLast || pPrevious == pLast) return;
152
153     p->next = pPrevious->next;   // (1)
154     p->previous = pPrevious;     // (2)
155     pPrevious->next = p;         // (3)
156     p->next->previous = p;       // (4)
157 }
```

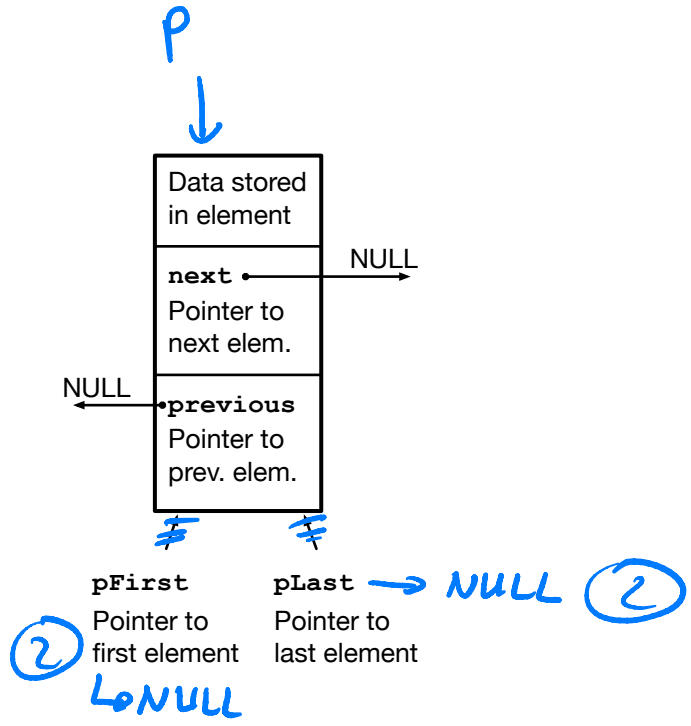
Adding a new element p in the middle, before an element p2

- (Check that there are at least two elements in the list by validating that the pointer to the first and the pointer to the last element differ, i.e. `pFirst != pLast` evaluates to true.)
- (Validate that you do not add the element at the beginning of the list by ensuring that `pPrevious != pFirst` evaluates to true.)
- (Verify that you do not append the element at the end of the list by validating that `pPrevious != pLast` evaluates to true.)
- Make the next-pointer of the new element p point to its successor: `p->next = p2`.
- Make the previous-pointer of the new element p point to its predecessor: `p->previous = p2->previous`.
- Make the next-pointer of the predecessor point to the new element: `p->previous->next = p`.
- Make the previous-pointer of the successor point to the new element: `p2->previous = p`.



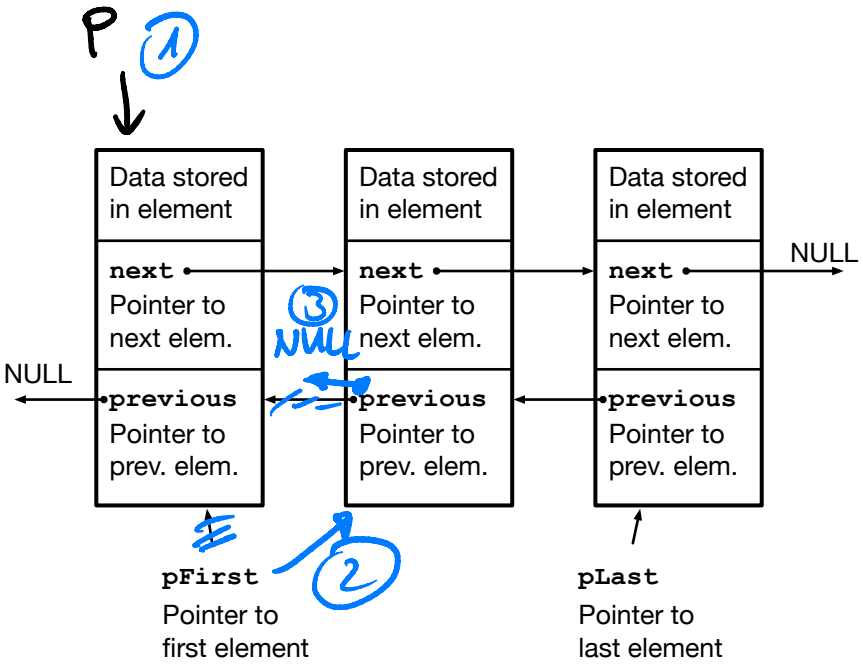
Removing the only element from the list

- (Verify that there is only one element in the list by checking `pFirst == pLast` equals to true.)
- Save a pointer to the element to remove: `p = pFirst`.
- Assign `NULL` to the pointer to the first and the pointer to the last element: `pFirst = pLast = NULL`.
- Return `p`



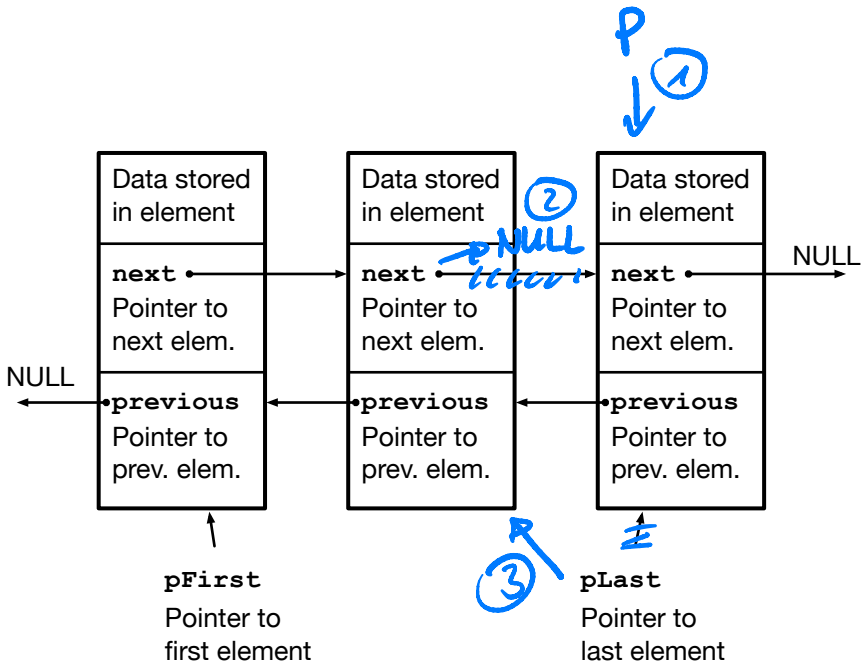
Removing the first element from a list of at least two elements

- (Verify that there is more than one element in the list by checking that `pFirst != pLast` equals to true.)
- Save a pointer to the first element: `p = pFirst`.
- Make the pointer to the first element point to the (former) second element: `pFirst = pFirst->next`
- Make the previous-pointer of the new first element point to `NULL`: `pFirst->previous = NULL`.
- Return `p`



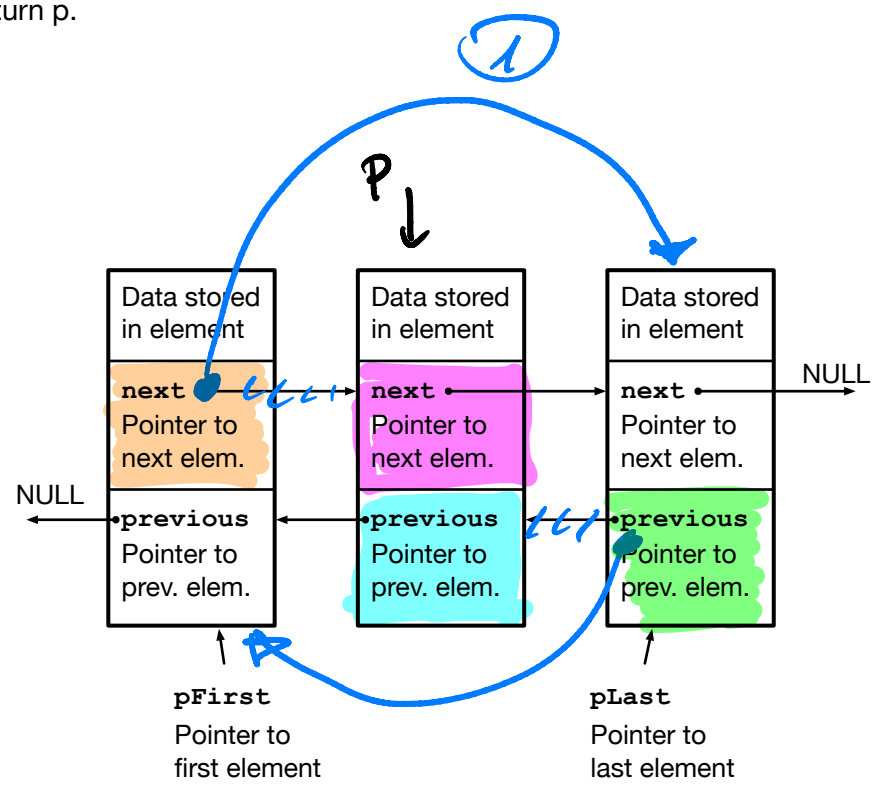
Removing the last element from a list of at least two elements

- Save a pointer to the element to remove: `p = pLast`.
- Make the next-pointer of the last but one element point to NULL: `pLast->previous->next = NULL`.
- Update the pointer to the last element: `pLast = pLast->previous`.
- Return `p`



Removing any element p between the first and the last element

- (Validate that you do not remove the first or last element by checking that both `p != pFirst` and `p != pLast` evaluates to true.)
- Make the next-pointer of the predecessor point to the successor:
  `p->previous->next` = `p->next`.
- Make the previous-pointer of the successor point to the predecessor:
  `p->next->previous` = `p->previous`.
- Return p.

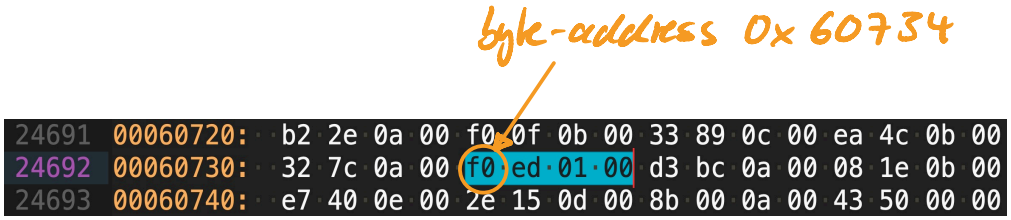


```
→  Code ./p
Position 2000: 395923

Time: 3.000000 µs
```

```
→  Code ./p
Position 50000: 143050

Time: 187.000000 µs
```

```
→  Code ./p
Position 98765: 126448

Time: 443.000000 µs
```

5 Searching



5.1 Searching in an array

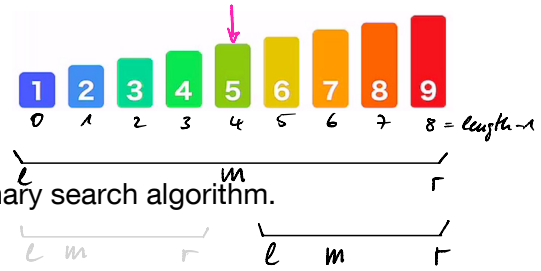
Linear Search

a) Watch the presentation/video of the linear search algorithm.

b) Provide code for function linearsearch that does a linear search on array a. If the function finds the number, it returns its position, or -1 otherwise. Measure the time the algorithm takes to find number 126448 in the file with 100,000 entries.

```c
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include "../datafiles/ints.h"


int linearsearch(int *a, int length, int number)
{
    for(int i = 0; i < length; i++)
        if(a[i] == number) return i;

    return -1;
}


int main()
{
    int n = 100000;
    int *a = malloc(n * sizeof(int));

    fill_array(a, "../datafiles/ints", n);

    clock_t start, end;

    start = clock();

    int position = linearsearch(a, n, 126448); // do the sorting

    end = clock();

    // a ? b : c     if(a) b; else c
    printf("Position %d: %d\n\n", position,
           position >= 0 ? a[position] : -1);
    printf("Time: %lf µs\n\n",
           (((double) (end - start)) / CLOCKS_PER_SEC) * 1000000);
    printf("\n");
}
```

Binary Search



a) Watch the presentation of the binary search algorithm.

b) Provide code to implement the binarysearch function.

```c
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "../datafiles/ints.h"


int binary_search(int *a, int length, int number)
{

int binary_search(int *a, int length, int number)
{
    int l, r, m;    // left, right, middle

    l = 0;
    r = length - 1;

    while(l <= r)
    {
        m = (l + r) / 2;

        // found it
        if(number == a[m]) return m;

        // if number has not been at position m
        if(a[m] < number) l = m + 1;
        else r = m - 1;
    }

    return -1; // we did not find the number
}

}

void quicksort(int *a, int , int );

int main()
{
    int l = 100000;
    int a[l];

    clock_t start, end;

```

```
→  Code ./p
Number found at position 98765: 987452

Time: 2.000000 µs

→  Code
```

```
→  Code ./p
Number found at position 2000: 19534

Time: 4.000000 µs
```

```
→  Code ./p
Number found at position 50000: 498686

Time: 3.000000 µs
```

```
→  Code ./p
Number found at position 98000: 979651

Time: 4.000000 µs
```

```
43      fill_array(a, "../datafiles/ints", l);  // fill array with contents of file
44
45      quicksort(a, 0, l-1);
46
47      start = clock();
48
49      // provide call to function binary_search here
50      // provide variable 'position' to store the return value
51      end = clock();
52
53      if(position < 0) printf("Error - number not found!\n");
54      else printf("Number found at position %d: %d\n", position, a[position]);
55      printf("\nTime: %lf µs\n\n", (((double) (end - start)) / CLOCKS_PER_SEC) *
        1000000);
56  }
57
```

c) Read data file with 100,000 ints to the array and sort it applying the quicksort algorithm. Then, do a binary search for number 987452. Provide the time it takes to search the number.

```
→  Code ./p
Number found at position 98765: 987452

Time: 2.000000 µs

→  Code
```

5.2 Searching in a linked list

a) Provide the missing C code for the data structure to store an integer value in a doubly linked list.

```c
struct Integer
{
    // provide your code here




};
```

```c
struct Integer
{
    int i;
    struct Integer *pNext;
    struct Integer *pPrevious;
};
```

b) Provide – on file scope – the declaration of two pointers `pFirst` and `pLast` that store the pointer to the first and the last element of the doubly linked list.

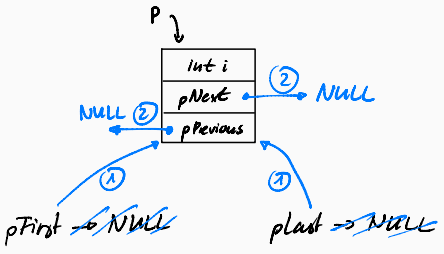
```c

// provide delcaration here
```

```c
// provide delcaration here
struct Integer *pFirst, *pLast;
```

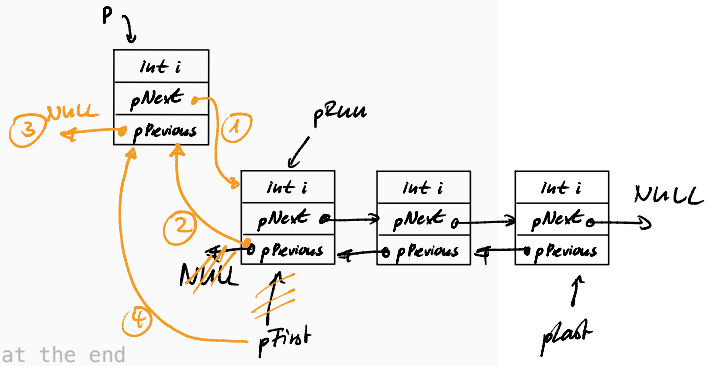
c) Provide C code for function *add_element_sorted*. The function inserts the provided parameter into the doubly linked list such that each element is only followed by elements with a larger number.

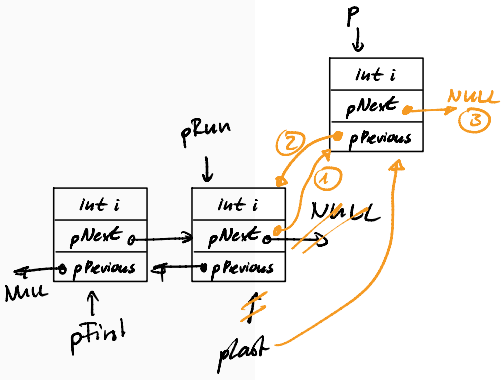
```c
void add_element_sorted(int i)
{
    struct Integer *p, *pRun;

```

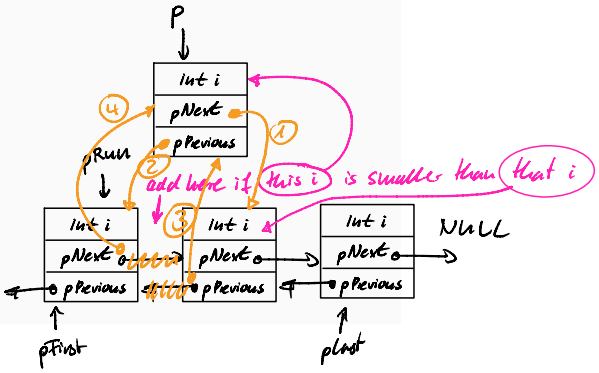
```c
    // allocate the structure
    p = malloc( sizeof(struct Integer) );
    p->i = i;

    // if the list is empty
    if(pFirst == NULL)
    {
        pFirst = pLast = p;             // (1)
        p->pNext = p->pPrevious = NULL; // (2)
    }
```



```
15
16      // if the list is not empty
17      for(pRun = pFirst; pRun ; pRun = pRun->pNext)
18      {
19          // if we have to add the new element just before the
20          // first element
21          if(pRun == pFirst && p->i < pRun->i)
22          {
23              p->pNext = pRun;          // (1)
24              pRun->pPrevious = p;      // (2)
25              p->pPrevious = NULL;      // (3)
26              pFirst = p;               // (4)
27              return;
28          }
29
30          // if we have to add the new element at the end
31          if(pRun->pNext == NULL)
32          {
33              pRun->pNext = p;          // (1)
34              p->pPrevious = pRun;      // (2)
35              p->pNext = NULL;          // (3)
36              pLast = p;                // (4)
37              return;
38          }
39
40          // if we have to add the new element after element pRun,
41          // between pRun and another element
42
43          if(p->i < pRun->pNext->i)
44          {
45              p->pNext = pRun->pNext;       // (1)
46              p->pPrevious = pRun;          // (2)
47              p->pNext->pPrevious = p;      // (3)
48              pRun->pNext = p;              // (4)
49              return;
50          }
51      }
52  }
```







Function `linearsearch` searches the doubly linked list for the provided paramter and returns its position if it finds it. Otherwise it returns -1.

d) Provide the missing C code to implement the function.

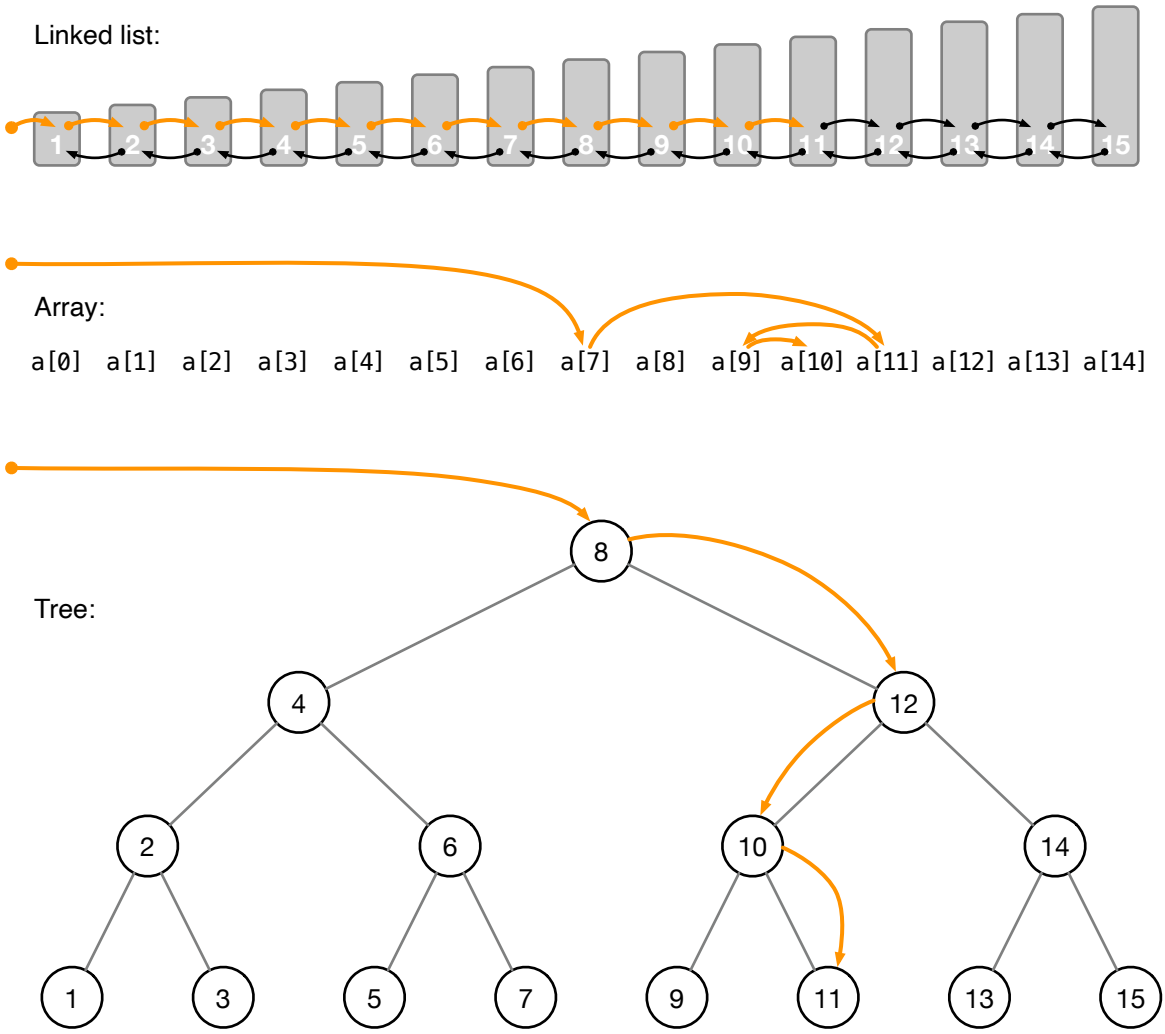
```c
int linearsearch(int i)
{
    // provide code for this function here
    // return -1 if the number is not found

    struct Integer *p;
    int count = 0;

    for(p = pFirst; p; p = p->pNext, count++)
    {
        if(i == p->i) return count;
    }

    return -1;

}
```

The main function reads 100,000 integers from a file and adds each element by calling function `add_element_sorted`.

e) Provide C code for the main function search for number 126488 with function `linearsearch` and store the position in variable `position`. Provide the execution time.

```c
int main()
{
    clock_t start, end;
    int n = 100000;
    int *a = malloc( n * sizeof(int) );

    int number = 126448; // search for 126448

    fill_array(a, "../datafiles/ints", n);  // fill array with file

    // create the list
    for(int i = 0; i < n; i++) add_element_sorted(a[i]);

    start = clock();

    // provide call to function linearsearch here

    int position = linearsearch(number);

    end = clock();

    if(position < 0) printf("Error - number not found!\n");
    else printf("Number %d found at position %d in the sorted list.\n",
        number, position);
    printf("\nTime: %lf µs\n\n",
        (((double) (end - start)) / CLOCKS_PER_SEC) * 1000000);
}
```

```
→  Code ./p
Number 126448 found at position 12689 in the sorted list.

Time: 311.000000 µs
```


Linked list:
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Array:
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11] a[12] a[13] a[14]
Tree:
8
4
12
2
6
10
14
1
3
5
7
9
11
13
15

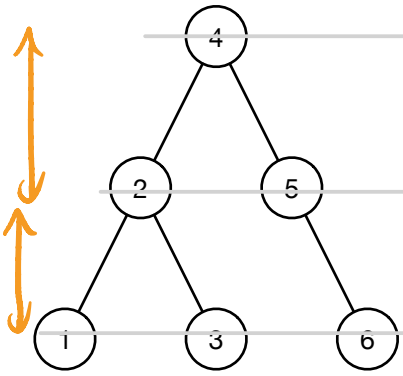
5.3 Binary Search Tree

Binary Search Tree Property:

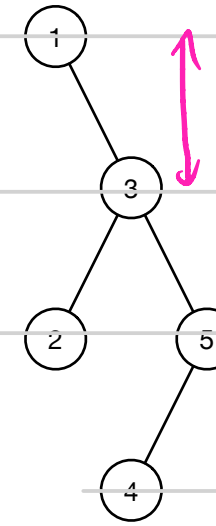
- Let x be a node in a binary search tree.
- If y is a node in the left subtree of x, then y.key < x.key.
- If y is a node in the right subtree of x, then x.key < y.key

```
→  Code ./p
Key: 01
node: 0x600003c911e0
parent: 0x0
left: 0x0
right: 0x600003c91200

Key: 02
node: 0x600003c91240
parent: 0x600003c91200
left: 0x0
right: 0x0

Key: 03
node: 0x600003c91200
parent: 0x600003c911e0
left: 0x600003c91240
right: 0x600003c91220

Key: 04
node: 0x600003c91260
parent: 0x600003c91220
left: 0x0
right: 0x0

Key: 05
node: 0x600003c91220
parent: 0x600003c91200
left: 0x600003c91260
right: 0x600003c91280

Key: 06
node: 0x600003c91280
parent: 0x600003c91220
left: 0x0
right: 0x0
```

4, 2, 5, 6, 1, 3



1, 3, 5, 2, 4, 6

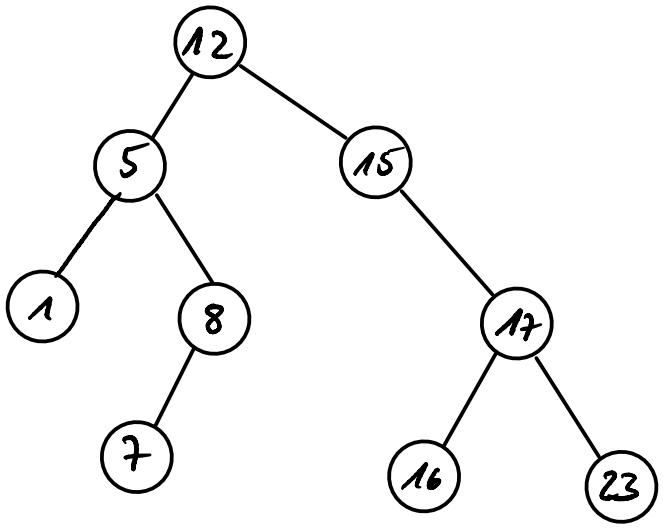


1, 2, 3, 4, 5, 6

The height of a binary serach tree depends on the order of inserting elements. In the examples, the heigts are 2, 3, and 5.

a) Provide the binary search tree for the sequence 12, 5, 8, 15, 1, 17, 16, 23, 7.

12, 5, 8, 15, 1, 17, 16, 23, 7



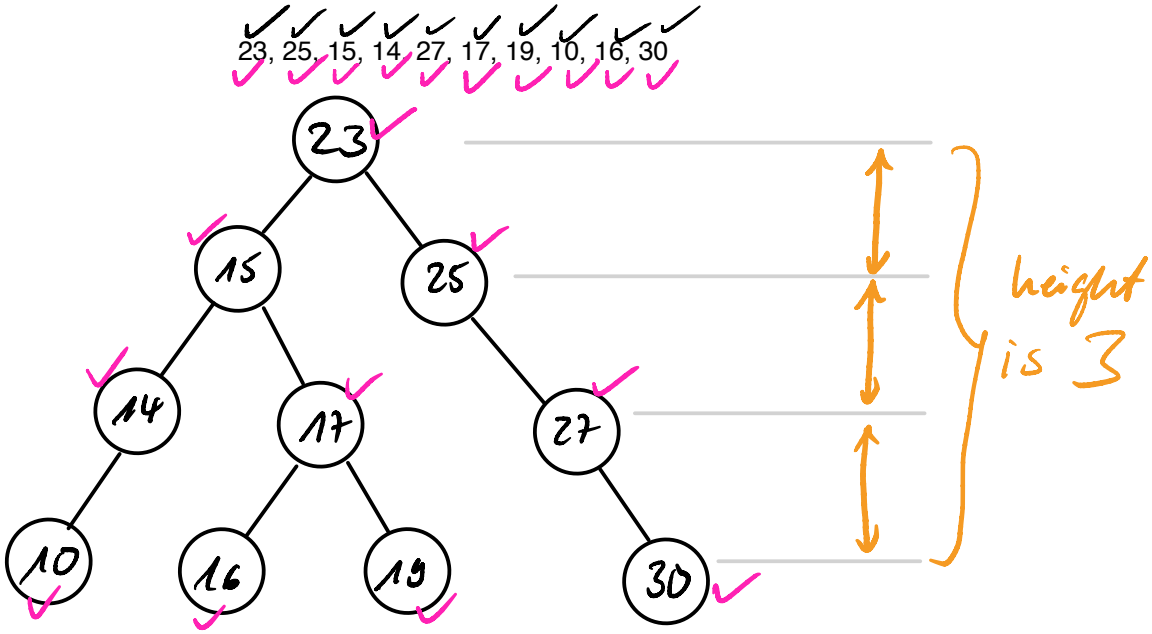
b) Provide a different sequence that results in an identical binary search tree.

12, 15, 17, 5, 8, 1, 7, 16, 23

c) Provide the height of the binary search tree.

3

d) Provide the binary search tree for the sequence 23, 25, 15, 14, 27, 17, 19, 10, 16, 30.



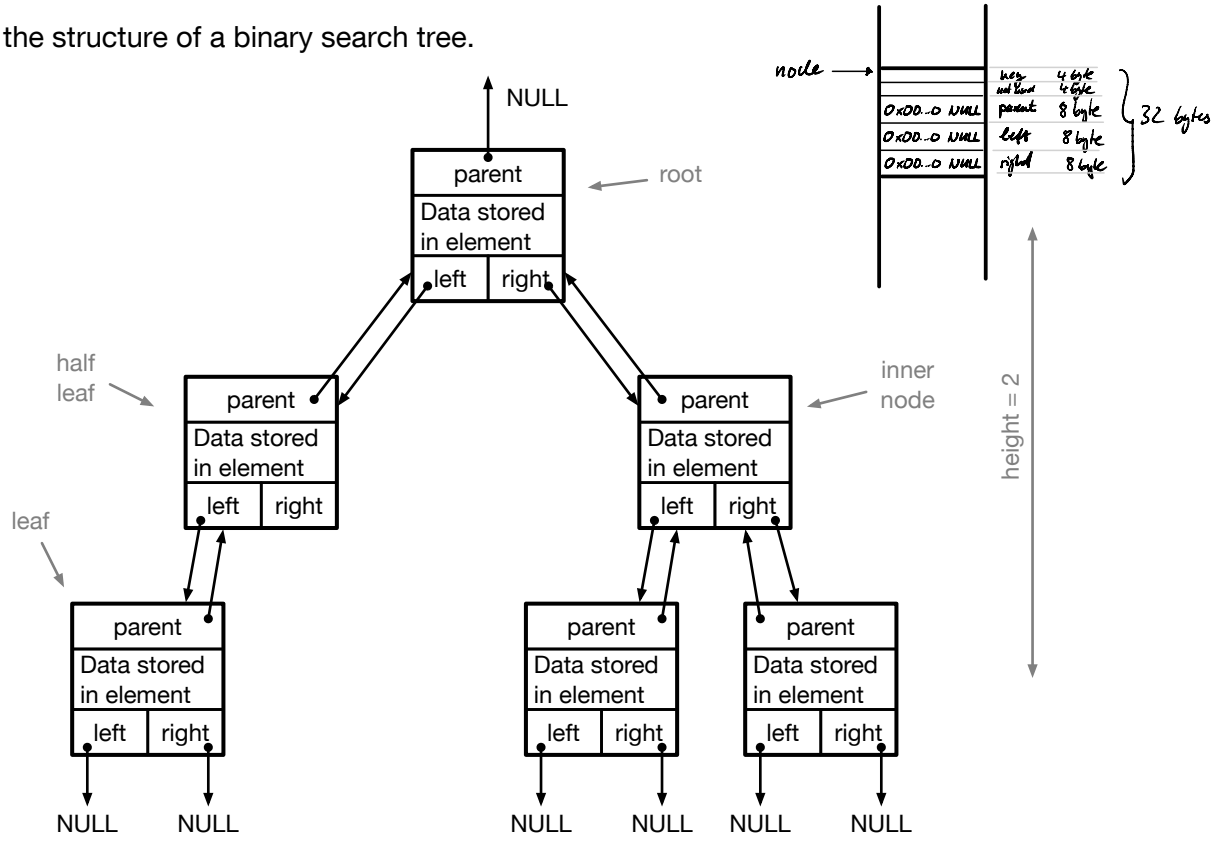
e) Provide a different sequence that results in an identical binary search tree.

23, 15, 25, 27, 14, 17, 30, 16, 19, 10

f) Provide the height of the binary search tree.

3

See the structure of a binary search tree.



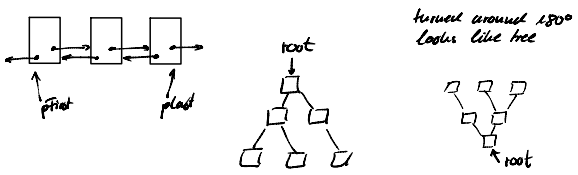
g) Provide a node structure that stores a single integer variable *key*.

```
struct Node
{
    int key;

    struct Node *parent;
    struct Node *left, *right;
};
```

Function `create_node` allocates runtime memory to create a single node, sets all pointers to NULL, initializes the key with the provided value and returns a pointer to the node.

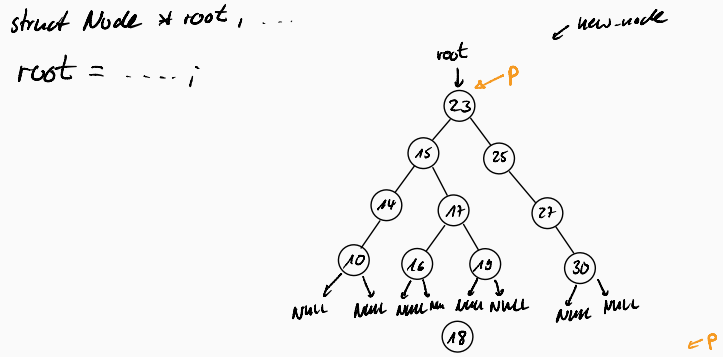
h) Provide C code to implement function `create_node`.

```
struct Node* create_node(int key)
{
    // int a;          // Variable "a" of type "int", i.e. a stores 4 byte fixed point

    // int *b;         // Variable "b" of type "int*", i.e. b
    //                 // stores 8 byte address (of an integer)

    struct Node *node;  // Variable "node" of type "struct Node *",
                        // i.e. node stores 8 byte address
                        // (of a structure of type "struct Node")

    node = malloc( sizeof(struct Node) );

    node->key = key;

    node->parent = NULL;
    node->left = NULL;
    node->right = NULL;

    return node;
}
```



i) Provide C code to implement function `insert_node_into_tree` that inserts node `new_node` into the (sub-)tree provided by `root`.

```
1  void insert_node_into_tree(struct Node **root, struct Node *new_node)
2  {
3
4
5
6
7
8
9
10
11
12
```




```c
void insert_node_into_tree(struct Node **root, struct Node *new_node)
{
    struct Node *p, *parent;

    // if the tree is empty, then
    // insert the first (= root) element
    if( *root == NULL )
    {
        *root = new_node;
        return;
    }

    // being here only the tree is not empty
    p = *root;

    while(p)
    {
        parent = p;

        if(new_node->key < p->key)   // go down left
            p = p->left;

        else                         // go down right
            p = p->right;
    }

    // now we are at the leaf (p == NULL)
    // parent points to the last node that
    // has been available
    new_node->parent = parent;

    // if we add at the left
    if(new_node->key < parent->key)
        parent->left = new_node;

    // add new_node at the right
    else
        parent->right = new_node;
}
```

There are three ways of printing the contents of a binary search tree.

- *Inorder traversing* prints the nodes in sorted order by applying a recursive algorithm
- *Preorder traversing* prints the value stored in a node before printing its subtree nodes.
- *Postorder traversing* prints the subtrees before printing their parent

j) Provide C code to implement function `print_keys_inorder_traversing`. Parameter `node` is the root of the tree to print.

```c
void print_keys_inorder_traverseing(struct Node *node)
{

    if(node)
    {
        print_keys_inorder_traverseing(node->left);

        printf("Key: %02d\nnode: %p\nparent: %p\nleft: %p\nright: %p\n\n",
               node->key, node, node->parent, node->left, node->right);

        print_keys_inorder_traverseing(node->right);

    }

}
```

k) Implement a recursive function `find_node_with_key` that takes a node and a key as parameter and returns the node matching the key. If no node is found, the function returns `NULL`.

```c
struct Node * find_node_with_key(struct Node * node, int key)
{
    if(node == NULL || node->key == key) return node;


    // go down left
    if(key < node->key) return find_node_with_key(node->left, key);

    // go down right
    else return find_node_with_key(node->right, key);
}
```

l) Provide the missing C code of the main function as indicated by the comments.

```c
int main(int argc, char *argv[])
{
    struct Node *node, *root = NULL;

    clock_t start, end;
    int i;
    int n = 100000;
    // numbers: 395923, 143050, 126448
    int number = 126448;

    int *a = malloc( n * sizeof(int) );

    // create the tree
    fill_array(a, "../datafiles/ints", n);
    for(i = 0; i < n; i++)
    {
        node = create_node(a[i]);
        insert_node_into_tree(&root, node);
    }

    // print the tree for inspection
    // print_keys_inorder_traverseing(root);

    start = clock();

    node = find_node_with_key(root, number);

    end = clock();

    printf("\nSearch for: %d\n", number);
    printf("node found: %p\n", node);
    printf("Time: %lf µs\n\n",
        (((double) (end - start)) / CLOCKS_PER_SEC) * 1000000);


    return 0;
}
```

```
Search for: 987452
node found: 0x600001b69400
Time: 2.000000 µs
```

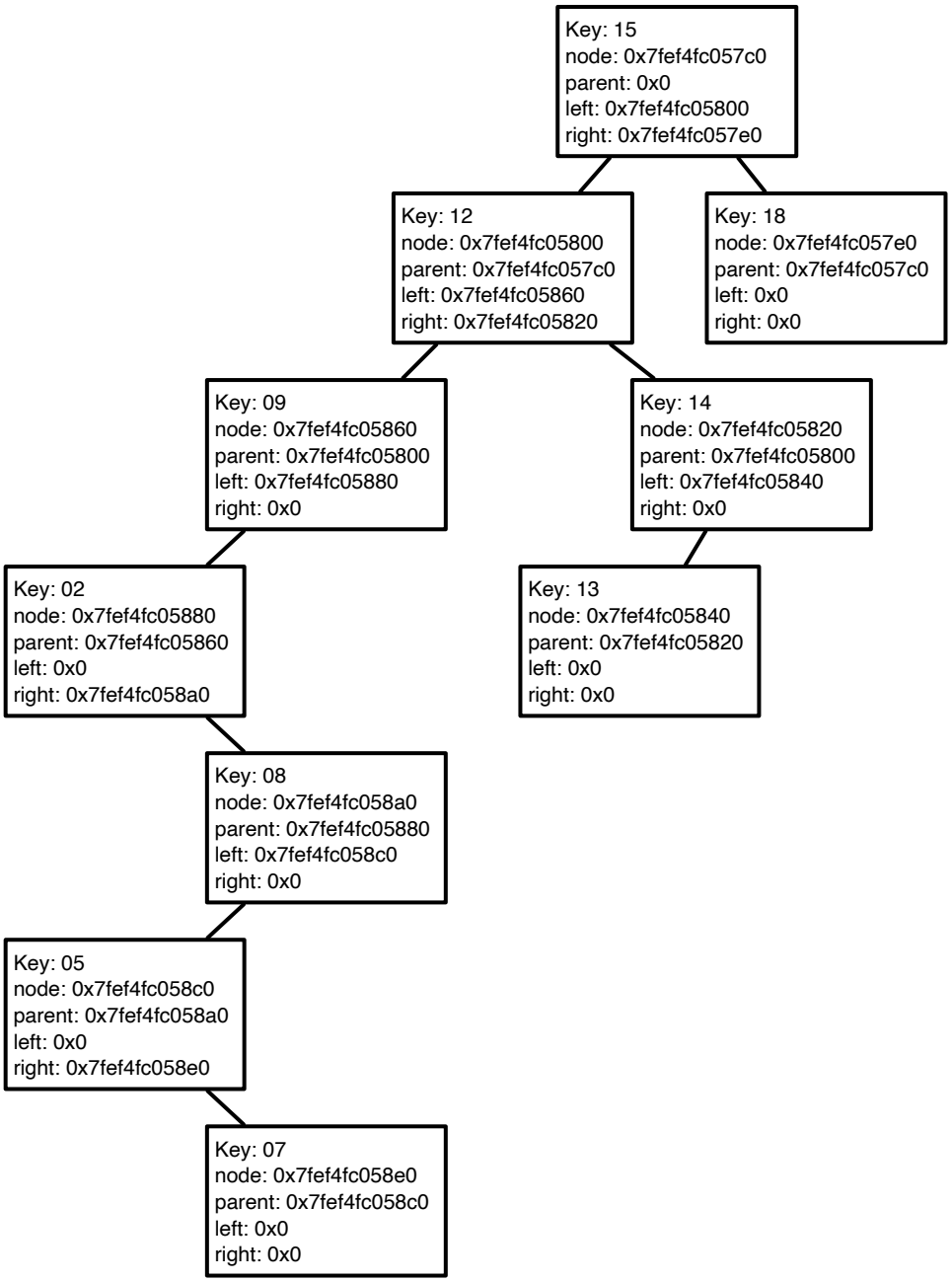
```
Search for: 395923
node found: 0x6000038a6ae0
Time: 3.000000 µs
```

```
Search for: 143050
node found: 0x6000025ed7c0
Time: 2.000000 µs
```

```
Search for: 126448
node found: 0x600002036060
Time: 2.000000 µs
```

See an example of a binary search tree. The tree structure has been determined from the values provided by `print_keys_inorder_traverseing(root)`.

int keys[] = {15, 18, 12, 14, 13, 9, 2, 8, 5, 7};

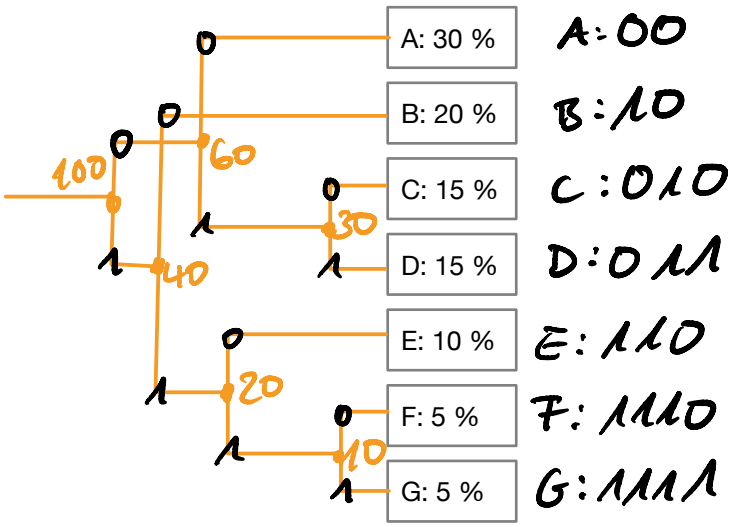


6 Data compression

6.1 Huffman Coding

Please watch the presentation about Huffman Coding.

a) Provide the Huffman tree and character encoding for the following character histogram.



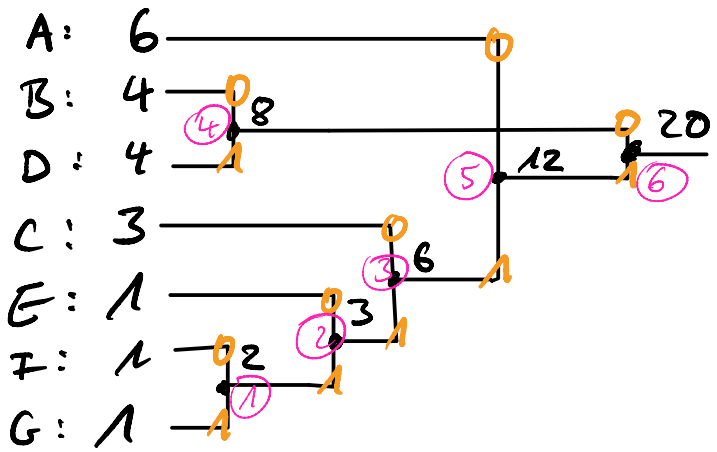
See the following sequence of characters: AABCBDEABDFBDACAACGD

b) Provide the frequency for each character.

A: 6
B: 4
C: 3
D: 4
E: 1
F: 1
G: 1

c) Provide a huffman tree for the sequence of characters.

A:
B:
C:
D:
E:
F:
G:



A: 10
B: 00
C: 110
D: 01
E: 1110
F: 11110
G: 11111

d) Provide the coding for each character. Arrange 0 and 1 such that 0 is on top and 1 at the bottom of a branch.

see c)

A: 10
B: 00
C: 110
D: 01
E: 1110
F: 11110
G: 11111

e) Encode AABCBDEABFBDACAACGD applying Huffman encoding and provide the bit pattern. Provide spaces between each character for better reading.

A A B C B D E A B F B D A C A A C G D

10 10 00 110 00 01 1110 10 00 01 11110 00 01 10 110 10 10 110 11111 01

f) How many bits does it take to encode sequence in Huffman coding?

14 · 2 Bits + 3 · 3 Bits + 1 · 4 Bits + 2 · 5 Bits

28 + 9 + 4 + 10

= 51

g) How many bits would it take to encode string `AABCBDEABDFBDACAACGD` in ASCII?

20 chars @ 8 Bit => 160 Bits

h) How many bits would it take if you encode it with an minimum, but constant amount of bits?

7 different chars => 3 bits

3 · 20 = 60 bits

i) Assume the following encoding and provide the character string for the provided bit pattern.

- A: 11
- B: 00
- C: 01
- D: 101
- E: 1001
- F: 1000

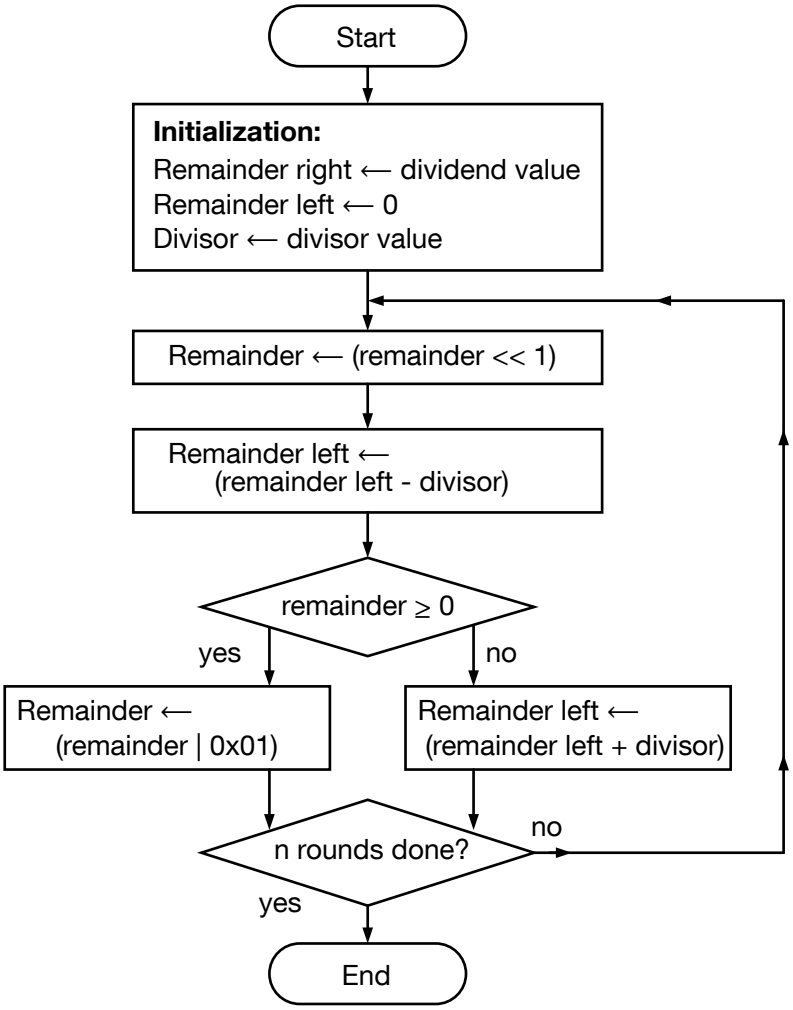
11|01|00|01|01|101|1001|1000|11|11|11|1001|101|00|101|00|00|01|01|11|11

A C B C C D E ≠ A A A E D B D B B C C A A

7 Project

7.1 Division-Algorithm

In this project work, you implement a division algorithm similar to the multiplication algorithm discussed in the lecture. Given the following Division algorithm.



In an initialization step, the dividend is placed into the right part of the remainder register; the left part of the remainder register is initialized with 0. The divisor is placed into the divisor register. Please note that the remainder register is a left-shift register.

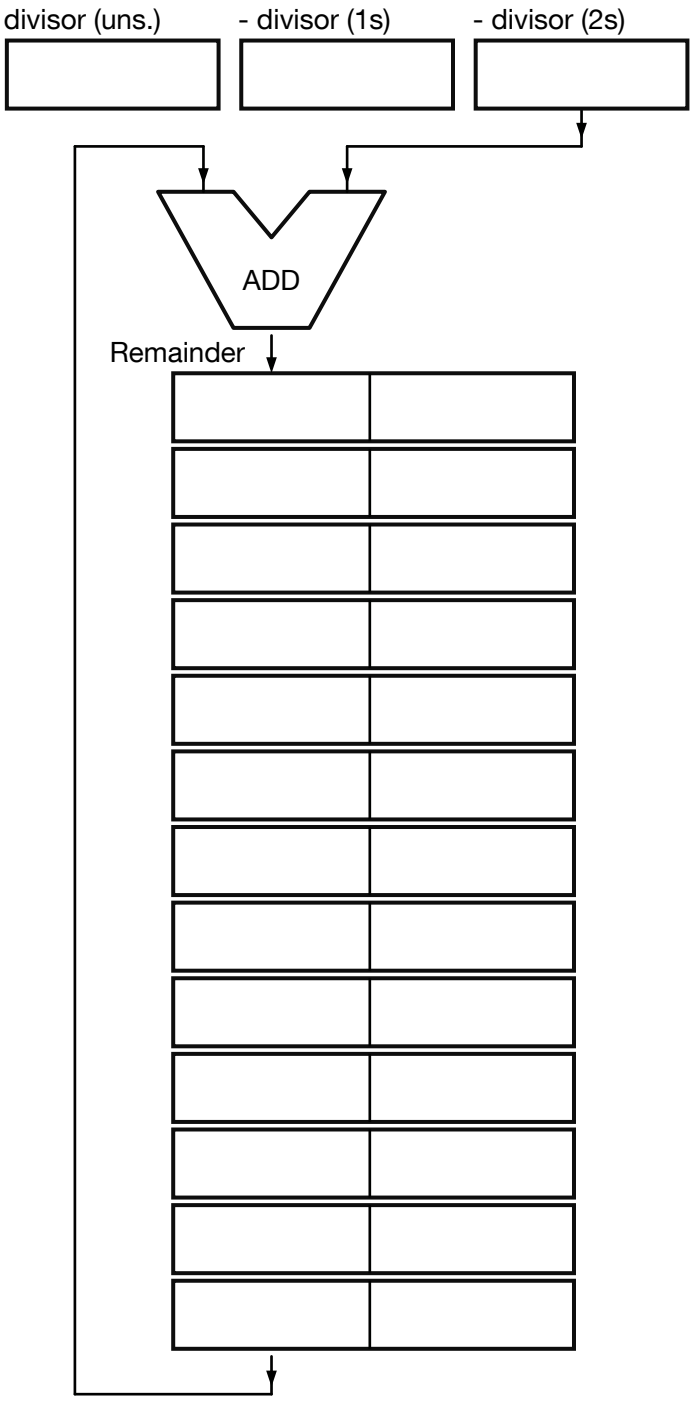
- On a rising edge on input *clk* with signal $\overline{\text{init}}/{<<}1$ set to 0, the remainder register takes over the input, stores it and provides it at its output.
- On a rising edge on input *clk* with signal $\overline{\text{init}}/{<<}1$ equal to 1, the remainder register shifts the internal value, re-stores it and provides it at its output.

Then, the algorithm iteratively repeats the following steps (n rounds for a word size of n bit):

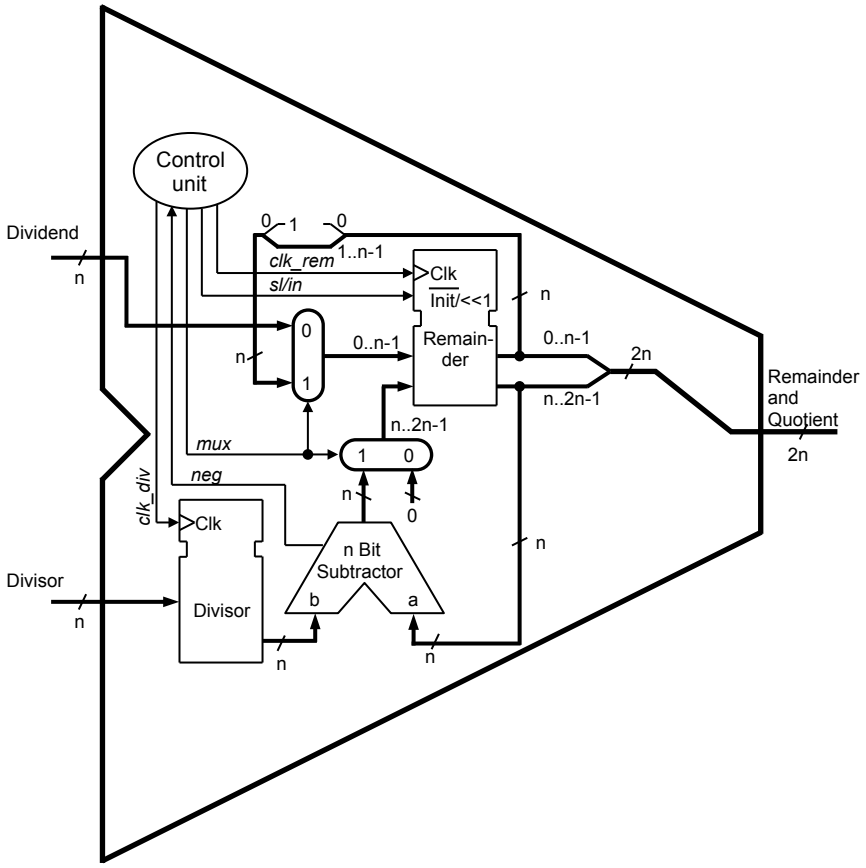
- The remainder is shifted to the left by 1.
- The left part of the remainder register now holds the partial dividend.
- The divisor is then subtracted from the left part of the remainder
    - If the remainder is negative, i.e. if the divisor did not fit the partial dividend, then the left part of the remainder is restored to be positive again.
    - If the remainder is positive, i.e. if the divisor did fit the partial dividend, then the LSB of the remainder register is set to 1.

After n rounds, the right part of the remainder register stores the result of the division (quotient); the left part of the remainder register holds the remainder.

a) Perform division 6/2 = 3 remainder 0. In order to perform the subtraction to check whether or not the divisor fits the partial dividend, add the negative divisor (2's complement) to the remainder. After n = 4 rounds, the remainder register should store bits 0000 0011, i.e. result 3 remainder 0.



The algorithm should now be implemented with the following circuit.

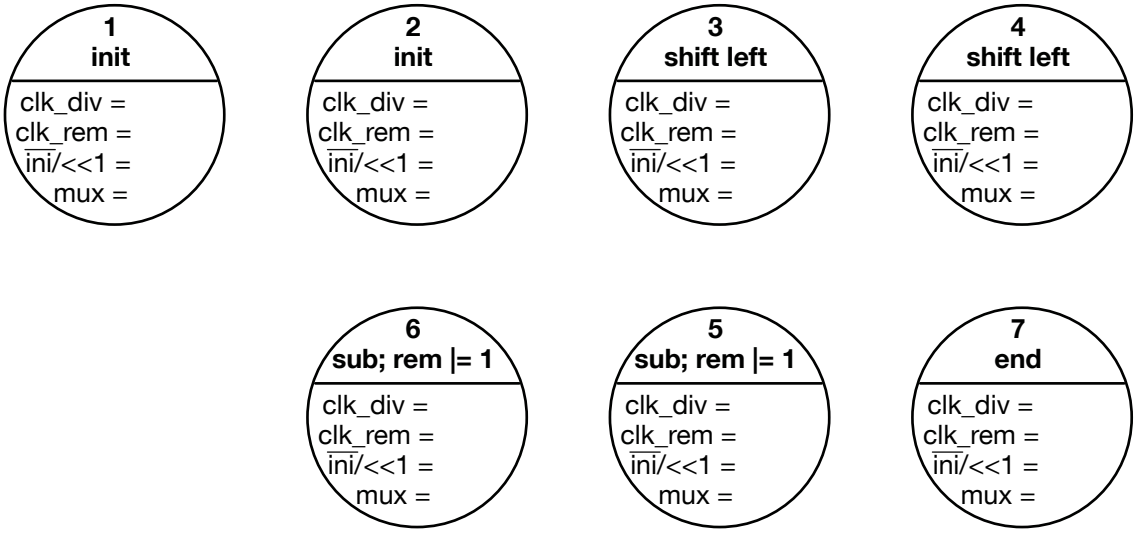


To check whether or not the divisor fits the partial dividend, the neg value of the subtractor provides

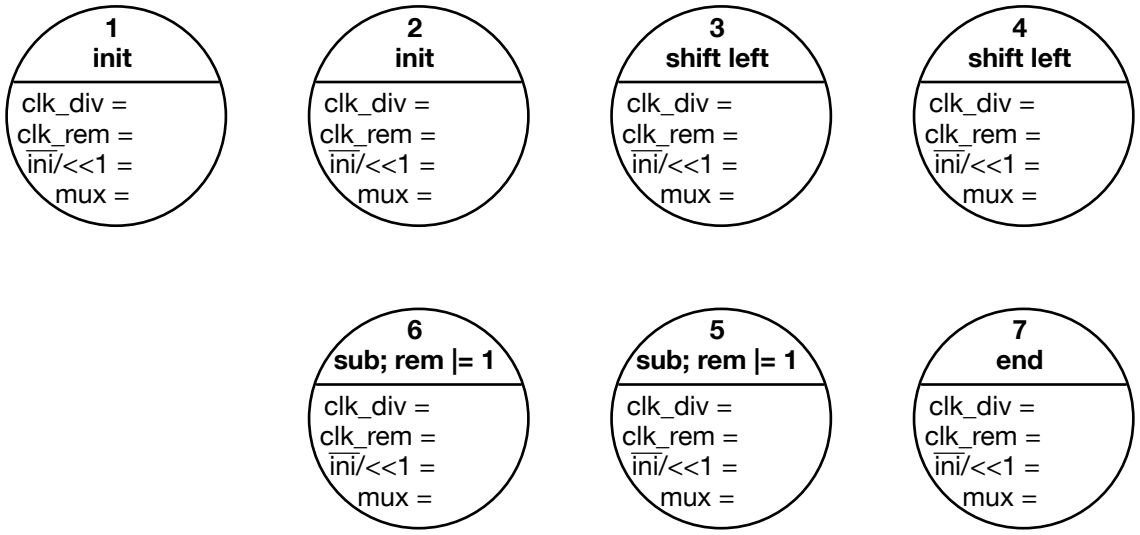
- 0 to the control unit, if partial dividend $\geq$ divisor, and
- 1 to the control unit, if partial dividend < divisor.

Note that the result of the subtraction does not have to be taken over by the remainder register in order to check if the divisor fits the partial dividend.
However, if the divisor fits the partial dividend, taking over the result from the subtractor in the upper part of the remainder register automatically sets the LSB of the lower part of the remainder register.

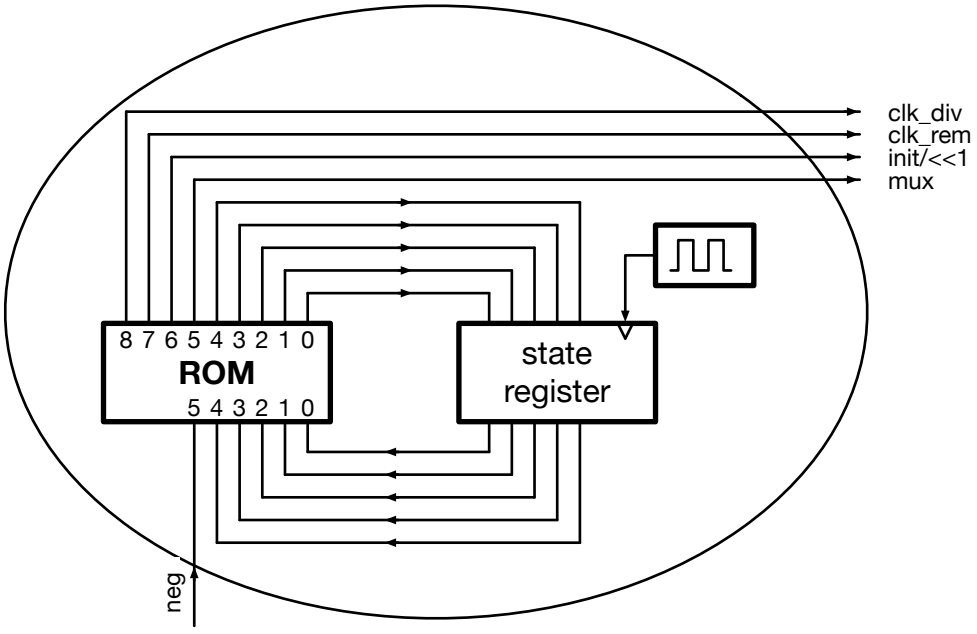
b) Provide transitions and output values to the Moore machine in order to make it control the division circuit appropriately.



See the following Moore machine.



The Moore machine will be implemented with the following circuit:



c) Provide the contents of the ROM memory in order to implement states 1 and 2 of the finite state machine.

d) Provide the contents of the ROM memory in order to implement state 3 of the finite state machine.

e) Provide the contents of the ROM memory in order to implement state 4 of the finite state machine.

f) Provide the contents of the ROM memory in order to implement state 5 of the finite state machine.

g) Provide the contents of the ROM memory in order to implement state 7 of the finite state machine.

7.2 The insertionsort algorithm

a) Watch the presentation. Then, formulate the insertionsort algorithm as a C program.

```c
#include <stdio.h>
#include <string.h>
#include <time.h>

// https://www.desmos.com/calculator/

const int amount = 100000;

void fill_array(int *a)
{
    char filename[100] = "./files/ints_";
    char filename2[100];
    sprintf(filename2, "%d", amount);
    strcat(filename, filename2);

    printf("%s\n", filename);

    FILE *file = fopen(filename, "rb");

    if(fread(a, sizeof(int), amount, file) != amount)
    {
        printf("ERROR - could not read all data!\n");
    }
}


int main()
{
```

```
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73 }
```

b) Run the algorithm with n = 10, 100, 1000, 10000, and 100000 numbers and measure execution time. Compare n and runtime in a table.

c) Provide a graph that plots execution time over n. Look for an appropriate tool or web framework to perform such a plot, e.g. https://www.desmos.com/calculator/.

d) Estimate how the maximum runtime dependes on n.