Gør tanke til handling
VIA University College

Indsæt ikon

Today

- Introduction to CAO and Laurits
- Numbers repetition
- Exercise in number
- Boolean algebra
- Exercise in Boolean algebra
- Introduction to hardware
- Exercises in hardware.

About me

- Laurits Ivar Anesen
- 2 kid

Electronic engineering from Aarhus University – specialty in embedded programming and signal processing – in relation to sound.

Purpose of CAO1 course

Understand how a computer works

Industry relevance

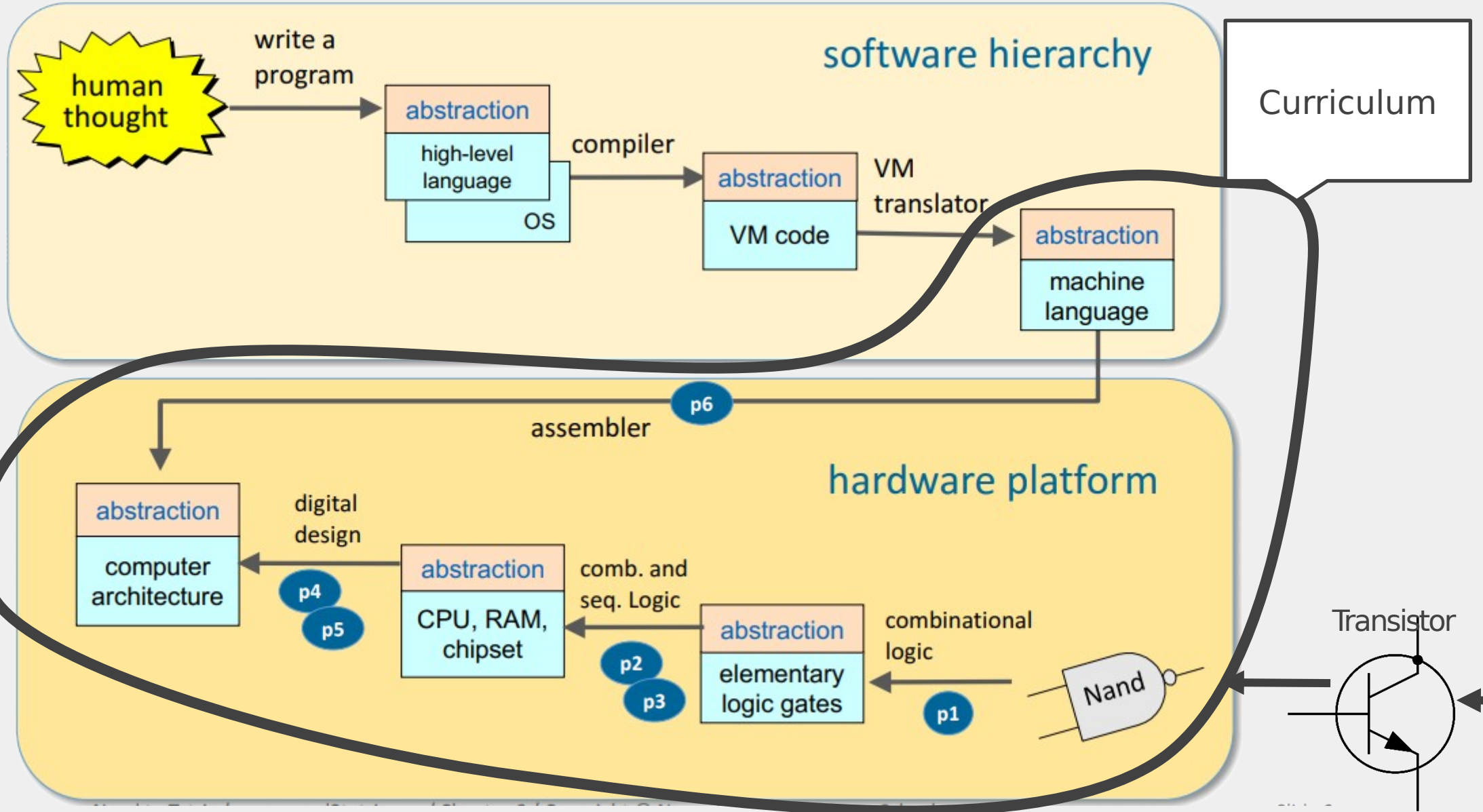
- Embedded Systems / Microcontrollers
- For efficient coding, Computer architecture and assembler language are relevant.

Integral part of Software Technology Engineering

- Boolean logic / algebra
- Foundation for Real Time Programming and interfacing Electronics

Couse description

This course


human thought
write a program
software hierarchy
abstraction
high-level language
OS
compiler
abstraction
VM code
VM translator
abstraction
machine language
Curriculum
p6
assembler
hardware platform
abstraction
computer architecture
digital design
p4
p5
abstraction
CPU, RAM, chipset
comb. and seq. Logic
p2
p3
abstraction
elementary logic gates
combinational logic
p1
Nand
Transistor
Physics

Your commitment:

To achieve experiences I expect:

- Students interest and motivation
- Study, read, and do the exercises
- Ask questions, ask, ask!
- Attendance in class
- Hand in 2 mandatory assignments (9. March and 13. April)
- Plan the 5 ECTS course = 12 hours work/week.

Gør tanke til handling
VIA University College

Lecture 1

Boolean Logic and Boolean arithmetic and breadboard

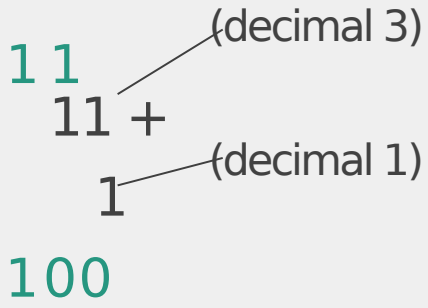
Decimal number

- Humans couldn’t count to more than 10. Therefor we invented the decimal numbers

```
 1 1
   88 +
   13
 101
```

Binary number

- Transistor can’t count to more than 2 (there is voltage or no voltage). Therefor we invented the binary numbers



Decimal Numbers

Decimal	2	0	5
Position value			
Result			

Binary numbers

Binary number:	1	0	1	1	0	1	0	1
Position value (in dec):								
Convertion to dec:								

181

Hex-numbers

HEX number:	A	8	F	2
Position value (in dec):				
Convertion to dec:				

Number conversion

Exercise, 10 min.

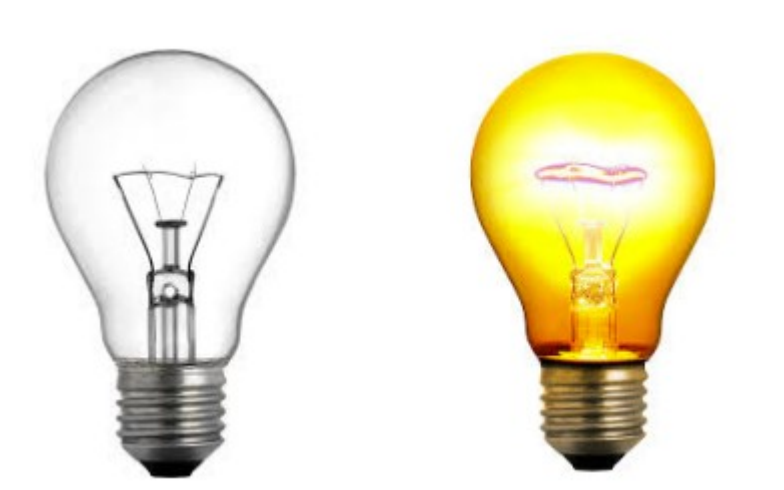
Write the results in decimal:

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Exercises 30 min

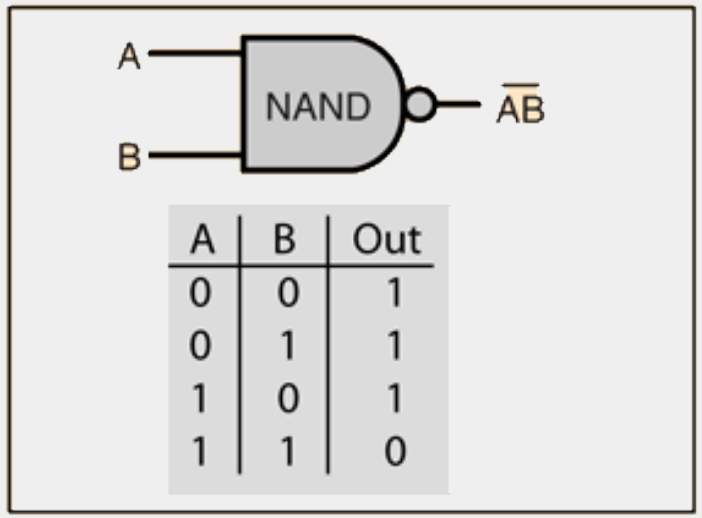
- Do exercise 1 (1.1-1.6) in the uploaded exercises.

Boolean values

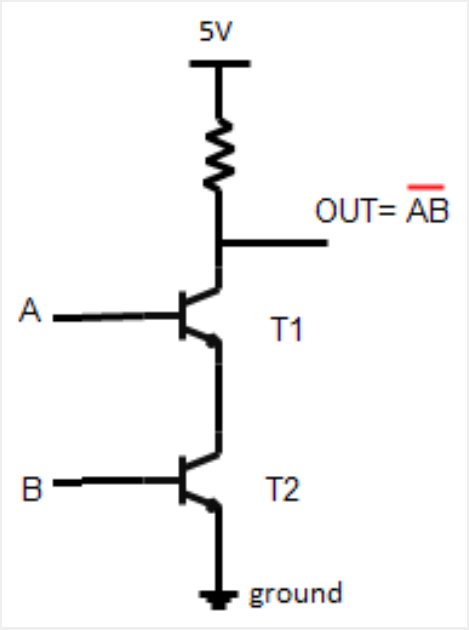


Transistor to NAND

- A transistor is an electrical contact



A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0



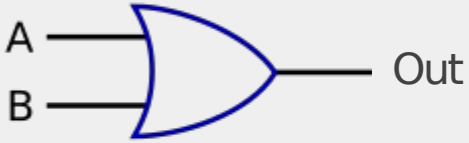
Boolean operations

AND

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

$A * B$

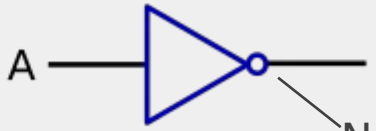
OR



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

$A + B$

NOT



Notice the little circle

A	Out
0	1
1	0

$\bar{A}$

Boolean operations

NAND (Not And)

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

$$\overline{A*B}$$

NOR (Not OR)

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

$$\overline{A+B}$$

XOR

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

$$\overline{A}*B+\overline{B}*A$$

Example: A Boolean function of two variables

Below, the function is displayed in a table, which shows the value of for all combinations of values of the inputs and :

		1	
		0	
		1	
		0	

“helping”-column – this is not strictly necessary, but it assists the calculation

Exercise 3 min

Draw the truth table for the following Boolean expression:

Example: A Boolean function of 3 variables

The function is displayed in the table below, which shows the value of for all combinations of the input values , and :

“helping”-columns – these are not strictly necessary, but they assist the calculation of

Do exercise 2 in the uploaded exercises.

You have 30 minutes to get as far as possible.

Ill introduce the hardware and the breadboard.

Use the rest of the time on the uploaded exercises.

- Draw circuit, truthtable, boolean expression.

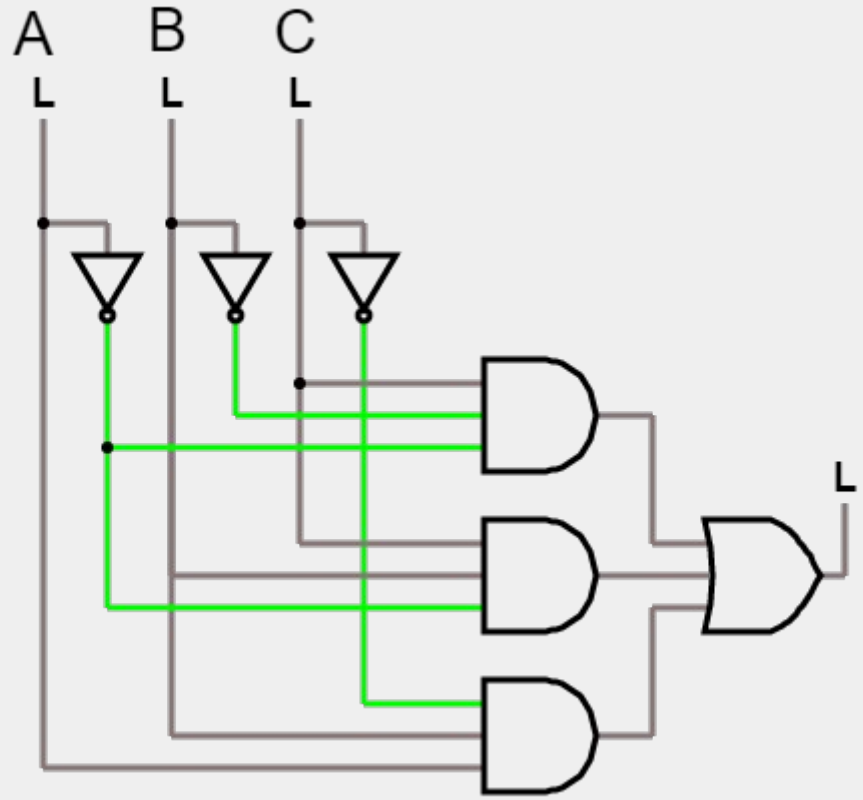
- Agenda:
- Boolean identities
- ALU
- Half adder
- Full adder
- Mux/ demux
- Exercises
- 2. compliment.

Truth table to Function

- Describe the 1’s
- Simplify the expression

$$\overline{A}\,\overline{B}\,C+\overline{A}\,BC+AB\,\overline{C}$$

A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	**0**



FALDSTAD

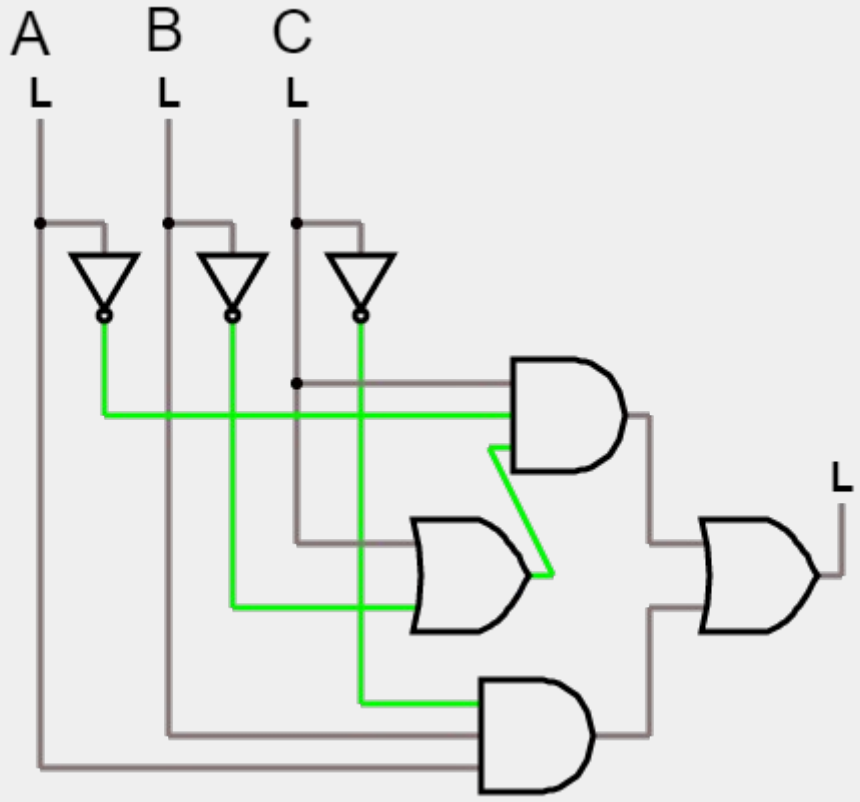
Truth table to Function

- Describe the 1’s
- Simplify the expression

A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	**0**

$$\overline{A}\,\overline{B}\,C+\overline{A}\,BC+AB\,\overline{C}$$

$$\overline{A}\,C\left(\overline{B}+B\right)+AB\,\overline{C}$$



FALDSTAD

Truth table to Function

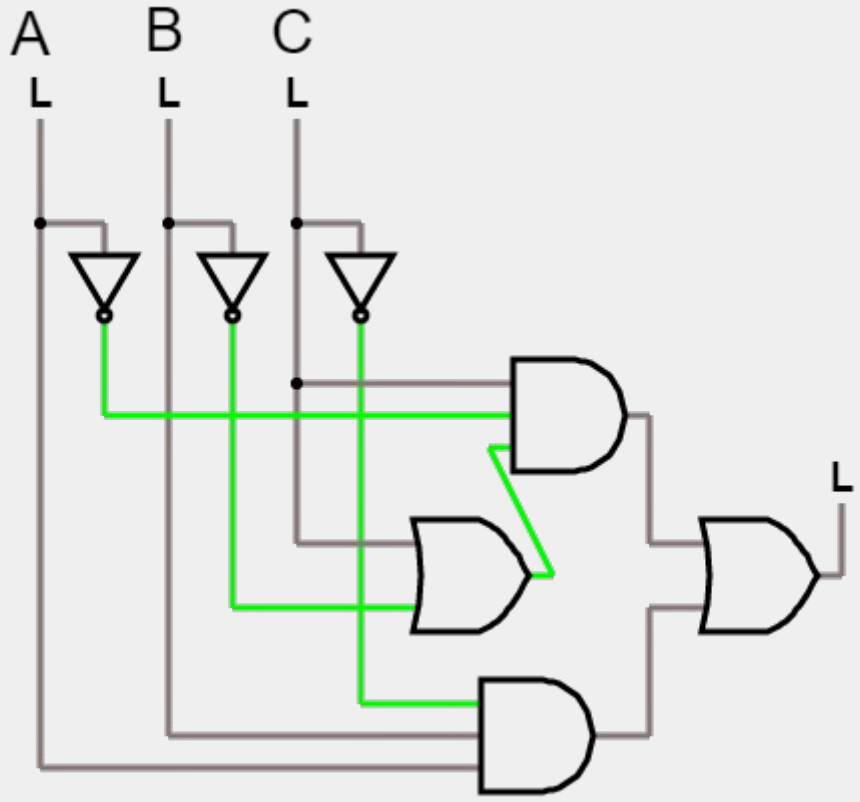
- Describe the 1's
- Simplify the expression

A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	**0**

$$\overline{A}\,\overline{B}\,C+\overline{A}\,BC+AB\,\overline{C}$$

$$\overline{A}\,C(\overline{B}+B)+AB\,\overline{C}$$

$$\overline{A}\,C(1)+AB\,\overline{C}$$



FALDSTAD

Truth table to Function

- Describe the 1's
- Simplify the expression

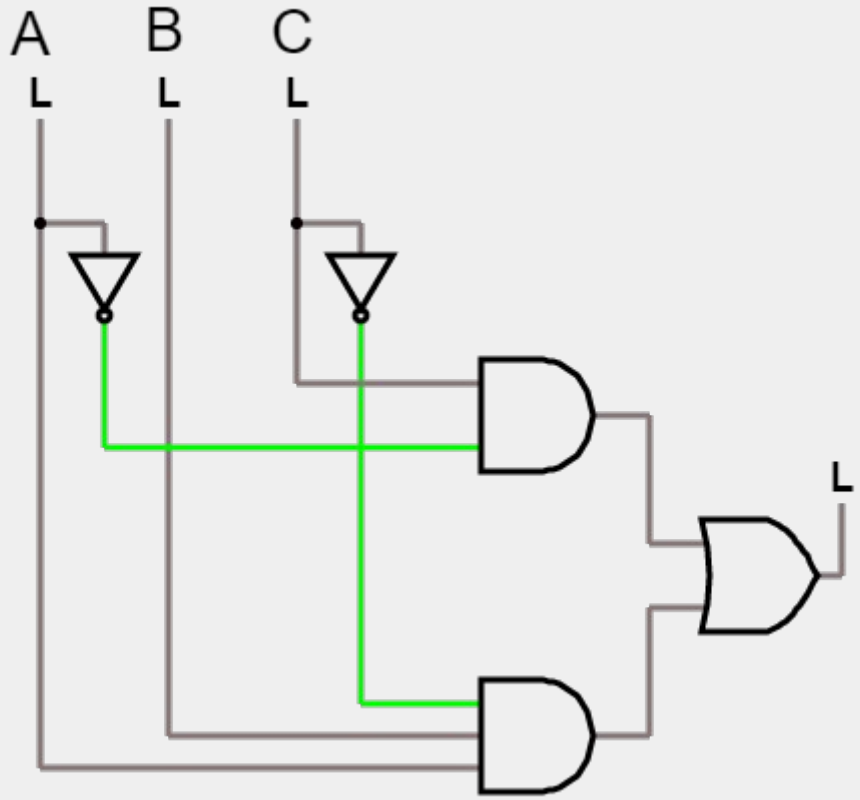
A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	**0**

$$\overline{A}\,\overline{B}\,C+\overline{A}\,BC+AB\,\overline{C}$$

$$\overline{A}\,C(\overline{B}+B)+AB\,\overline{C}$$

$$\overline{A}\,C(1)+AB\,\overline{C}$$

$$\overline{A}\,C+AB\,\overline{C}$$



FALDSTAD

Repetition

- Let me do the exercise from last week:
- Results is a+bc
- circuit

Commutative laws

$$XY = YX$$
$$X + Y = Y + X$$

Associative laws

$$(XY)Z = X(YZ)$$
$$(X + Y) + Z = X + (Y + Z)$$

Distributive laws

$$X(Y + Z) = XY + XZ$$
$$X + (YZ) = (X + Y)(X + Z)$$

De Morgan laws

$$\overline{XY} = \overline{X} + \overline{Y}$$
$$\overline{X + Y} = \overline{X}\,\overline{Y}$$
$$X + \overline{X}Y = X + Y$$

A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Exercise
10 minutes

Reduce it

Draw its truth table

Draw the circuit.

Commutative laws

$$XY = YX$$

$$X + Y = Y + X$$

Associative laws

$$(XY)Z = X(YZ)$$

$$(X + Y) + Z = X + (Y + Z)$$

Distributive laws

$$X(Y + Z) = XY + XZ$$

$$X + (YZ) = (X + Y)(X + Z)$$

De Morgan laws

$$\overline{XY} = \overline{X} + \overline{Y}$$

$$\overline{X + Y} = \overline{X}\,\overline{Y}$$

$$X + \overline{X}Y = X + Y$$

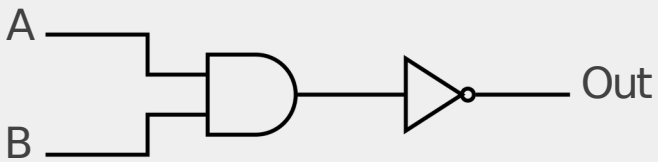
Is there a difference?

$\overline{AB}$

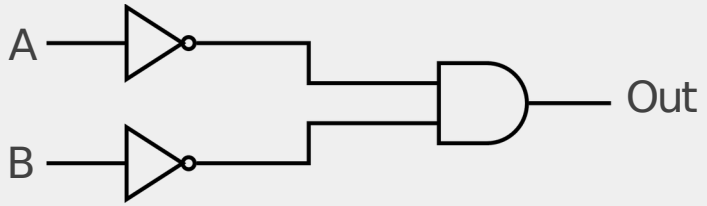
$\bar{A}\bar{B}$

Is there a difference?

$\overline{AB}$

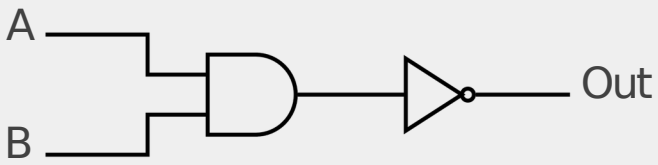


$\bar{A}\bar{B}$



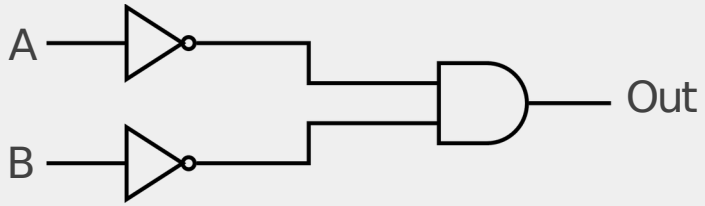
Is there a difference?

$\overline{AB}$



A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

$\bar{A}\bar{B}$



A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

Exercise 10 min.

Reduce the Boolean function above.

Draw its truth table.

Draw the circuit in the simulator falstad.

Commutative laws

$$XY = YX$$

$$X + Y = Y + X$$

Associative laws

$$(XY)Z = X(YZ)$$

$$(X + Y) + Z = X + (Y + Z)$$

Distributive laws

$$X(Y + Z) = XY + XZ$$

$$X + (YZ) = (X + Y)(X + Z)$$

De Morgan laws

$$\overline{XY} = \overline{X} + \overline{Y}$$

$$\overline{X + Y} = \overline{X}\,\overline{Y}$$

$$X + \overline{X}Y = X + Y$$

Exercise
10 min.

- Describe the 1’s
- Simplify the expression
- Draw the circuit in the simulator.

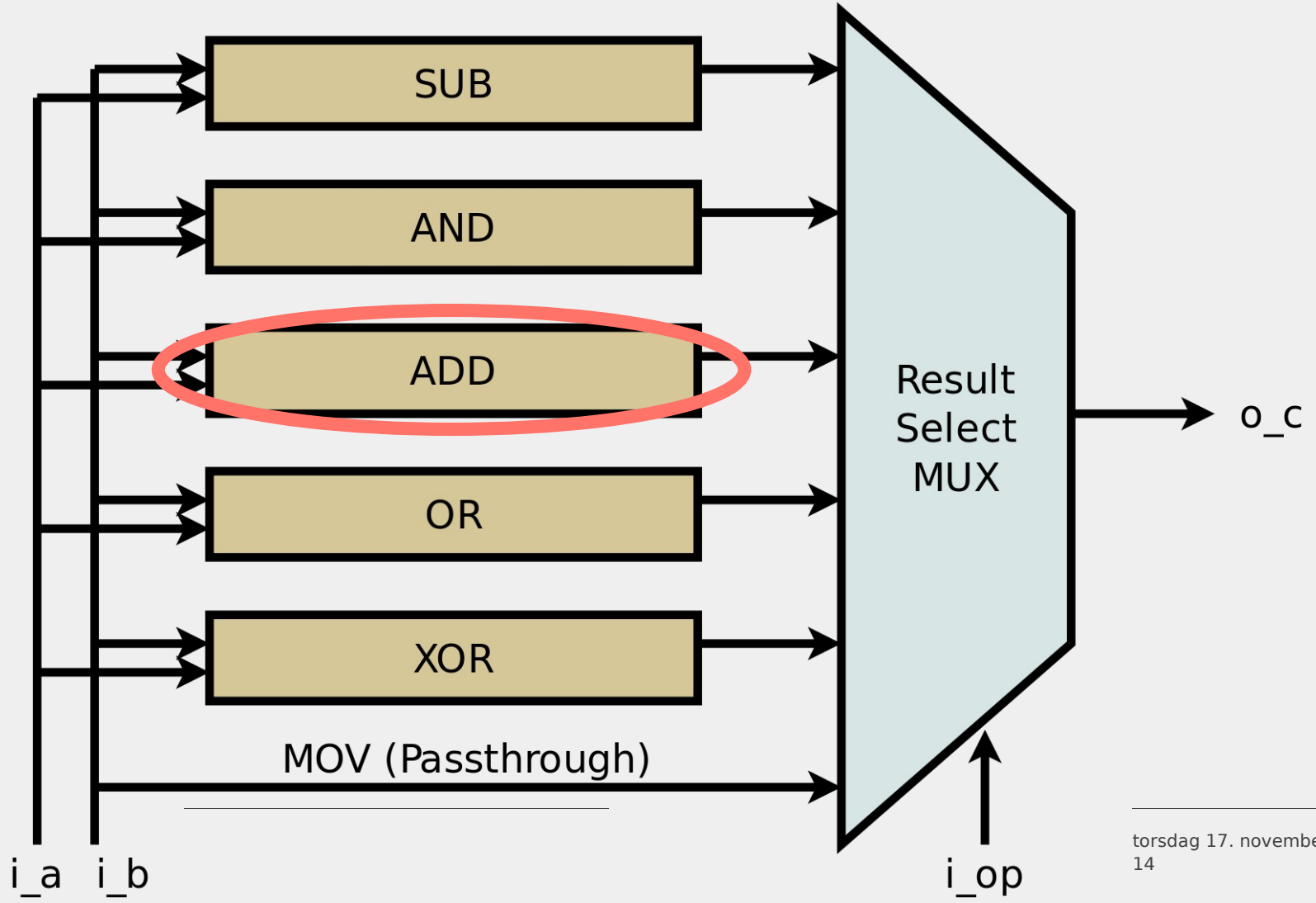
A	B	C	Out
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	**0**

Gør tanke til handling
VIA University College

Lecture 2

ALU

The Arithmetic Logical Unit (ALU)



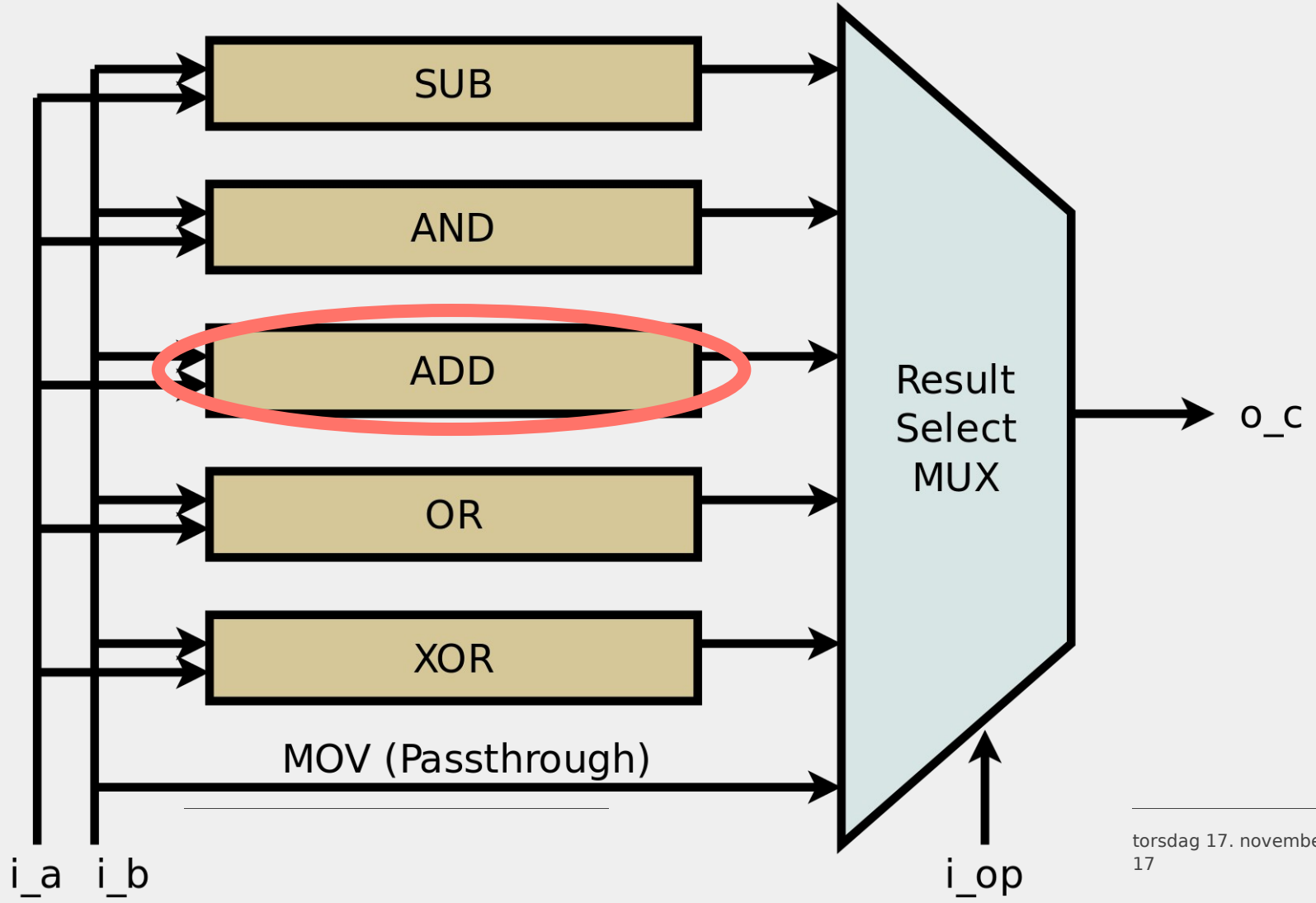
Halfadder:

https://forms.gle/DT2xrCtsbtNEK1cS9

Full adder:

https://forms.gle/wFnvMPG9Dmu27bT

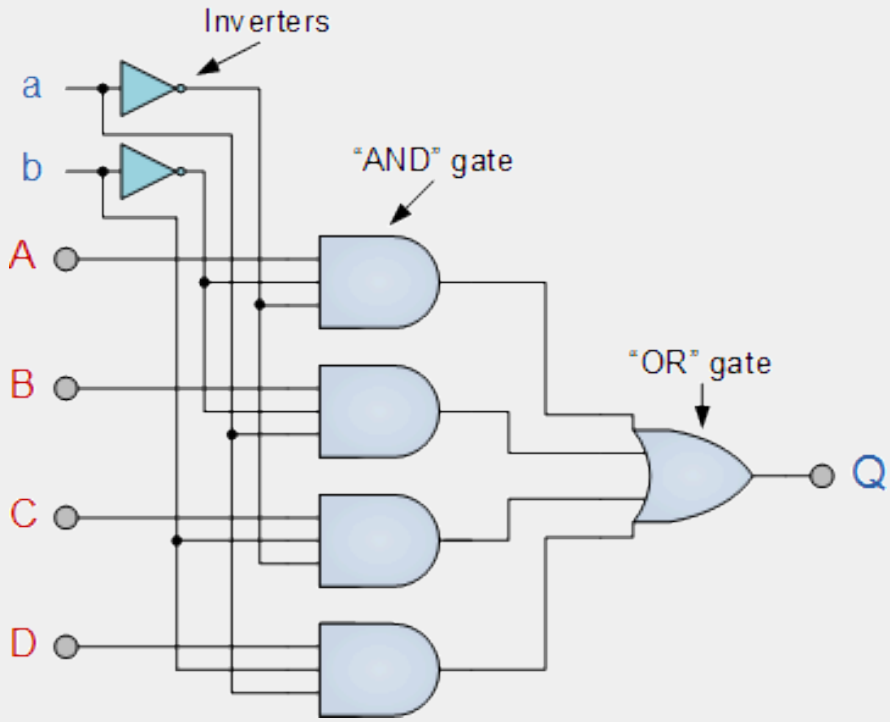
The Arithmetic Logical Unit (ALU)



MUX
https://forms.gle/YzDgtS9T4KLsW5ml

Multiplexer

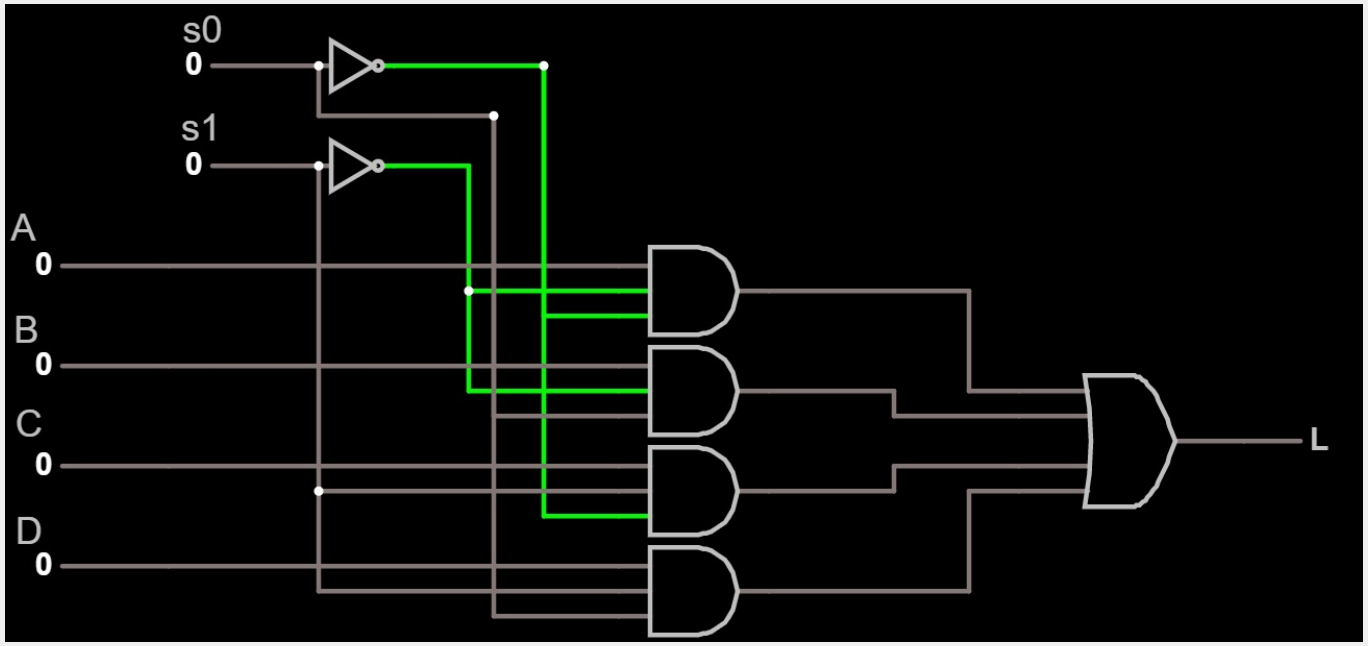
- Used for selecting 1 of multiple inputs
- Has Multiple inputs
- And some select bits



- FALSTAD

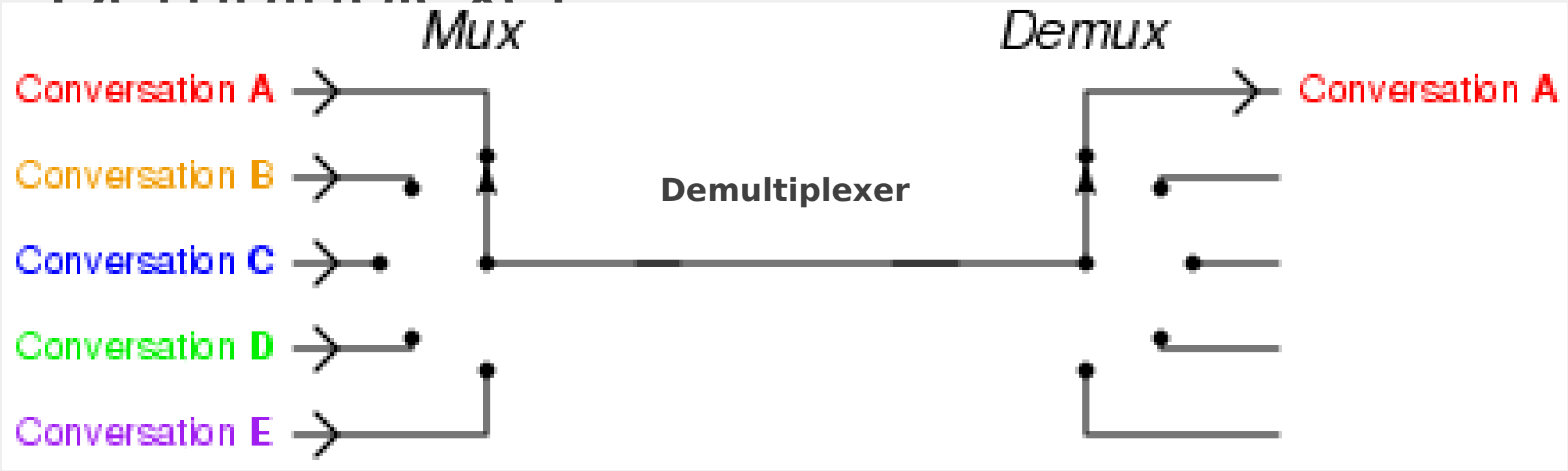
Exercise: 6 min

- Link
- Whats the output when the bitpattern is given as below:



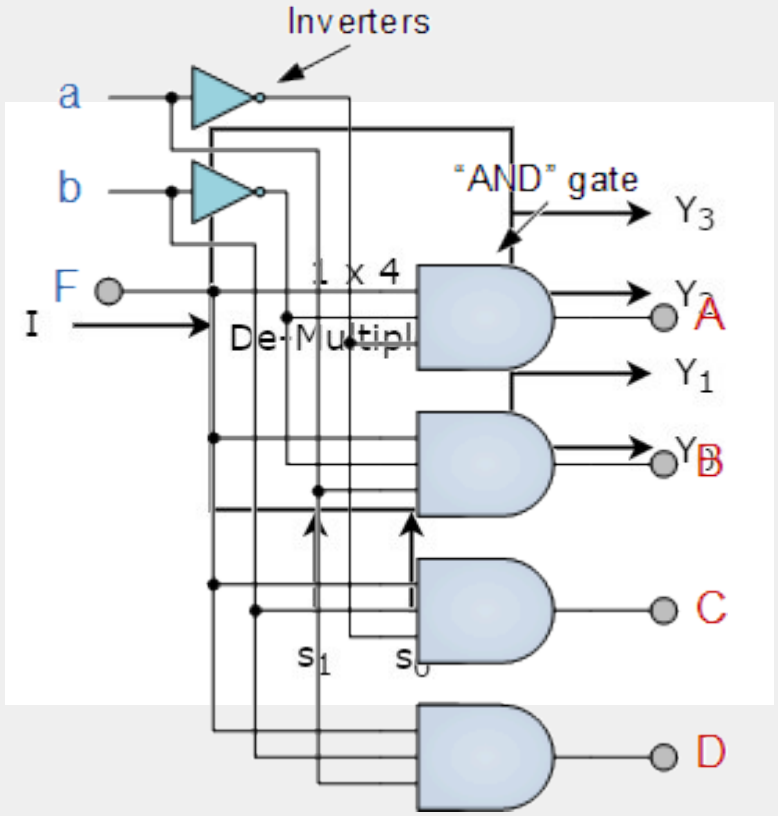
A	B	C	D	s0	S1	Out
0	1	1	1	0	0	?
1	0	1	1	1	0	?
0	0	0	1	1	1	?

MUX and DEMUX
Multiplexer
Demultiplexer



Demultiplexer

- Use to send 1 input to a specific output
- 1 input
- Some select bit



- [FALDSTAD]

Exercises until 11.15

- Do exercises 1, 2 and 3.

2. compliment

- Using a bit for sign is a bad idea.
  - Hard to do math
  - Hard to check if something is 0
  - Using a bit for sign is a bad idea.
- 1. compliment arithmetic
  - Invert the negative number
  - Add the Carry
- 2. compliment arithmetic
  - Invert the negative number, and add 1
  - Remove the Carry.

Uint_4	Uint_4	int	1. com	2. com
0000	0	-8	----	1000
0001	1	-7	1000	1001
0010	2	-6	1001	1010
0011	3	-5	1010	1011
0100	4	-4	1011	1100
0101	5	-3	1100	1101
0110	6	-2	1101	1110
0111	7	-1	1110	1111
1000	8	0	0000/1111	0000
1001	9	1	0001	0001
1010	10	2	0010	0010
1011	11	3	0011	0011
1100	12	4	0100	0100
1101	13	5	0101	0101
1110	14	6	0110	0110
1111	15	7	0111	0111

2. complement

https://forms.gle/j9SSNBabxYbHsdaR8

Gør tanke til handling
VIA University College

Indsæt ikon

Last week

- 2. compliment
- 15-11

Last week 2. compliment

- Using a bit for sign is a bad idea.
  - Hard to do math
  - Hard to check if something is 0
  - Using a bit for sign is a bad idea.
- 1. compliment arithmetic
  - Invert the negative number
  - Add the Carry
- 2. compliment arithmetic
  - Invert the negative number, and add 1
  - Remove the Carry.

Uint_4	Uint_4	int	1. com	2. com
0000	0	-8	----	1000
0001	1	-7	1000	1001
0010	2	-6	1001	1010
0011	3	-5	1010	1011
0100	4	-4	1011	1100
0101	5	-3	1100	1101
0110	6	-2	1101	1110
0111	7	-1	1110	1111
1000	8	0	0000/1111	0000
1001	9	1	0001	0001
1010	10	2	0010	0010
1011	11	3	0011	0011
1100	12	4	0100	0100
1101	13	5	0101	0101
1110	14	6	0110	0110
1111	15	7	0111	0111

- Lets test it in the simulator.

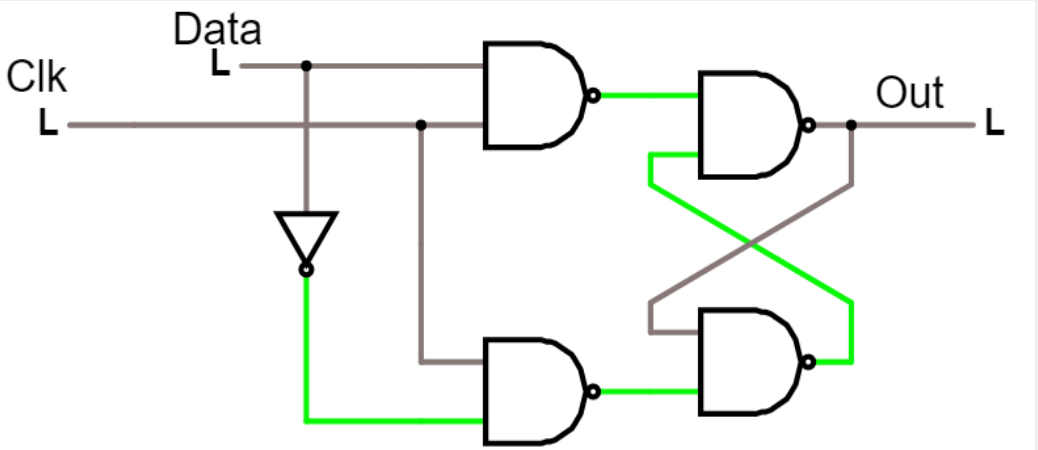
```
int main(void)
{
    /* Replace with your application code */
    int8_t test=-127;
    uint8_t test_u;
    test++;
    test++;
    test_u=test;

    while (1)
    {
    }
}
```

Gør tanke til handling
VIA University College

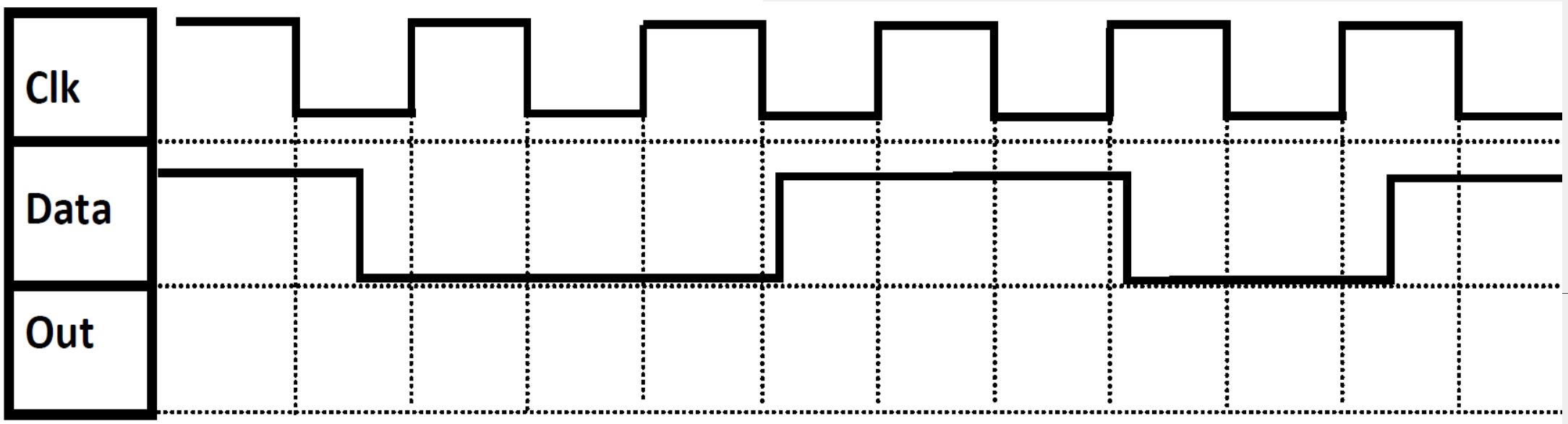
Lecture 3

Sequential logic

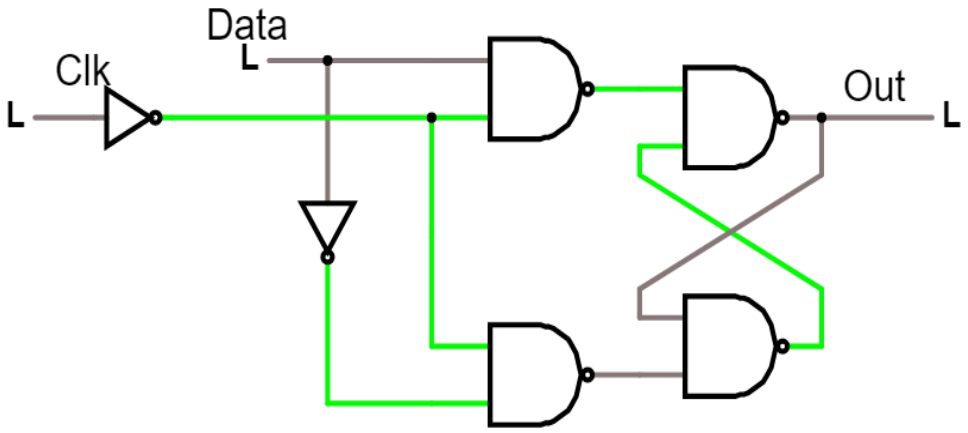
High level triggered circuit
Exercise – Draw the Out – 5 minutes



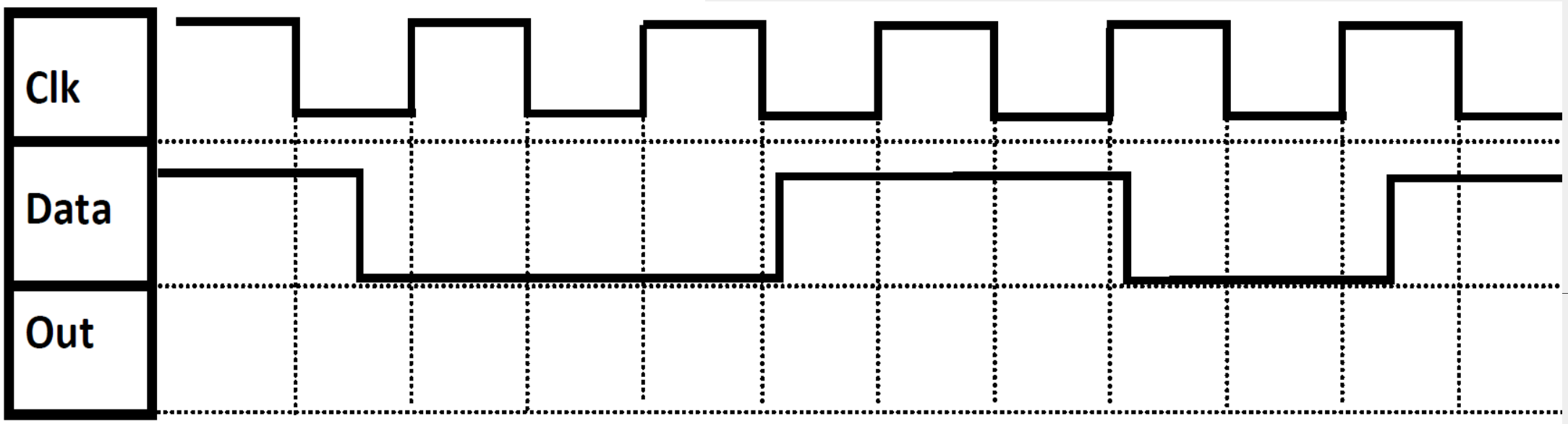
Simulation



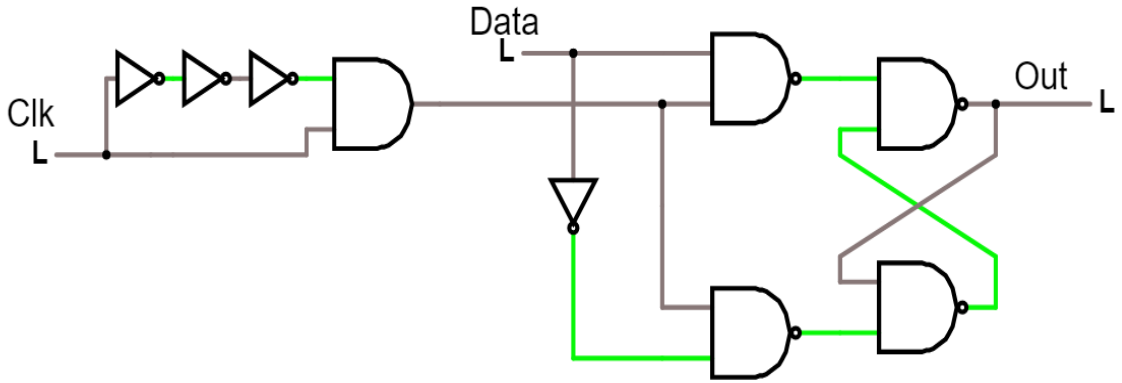
Low level triggered circuit
Exercise – Draw the Out – 5 minutes



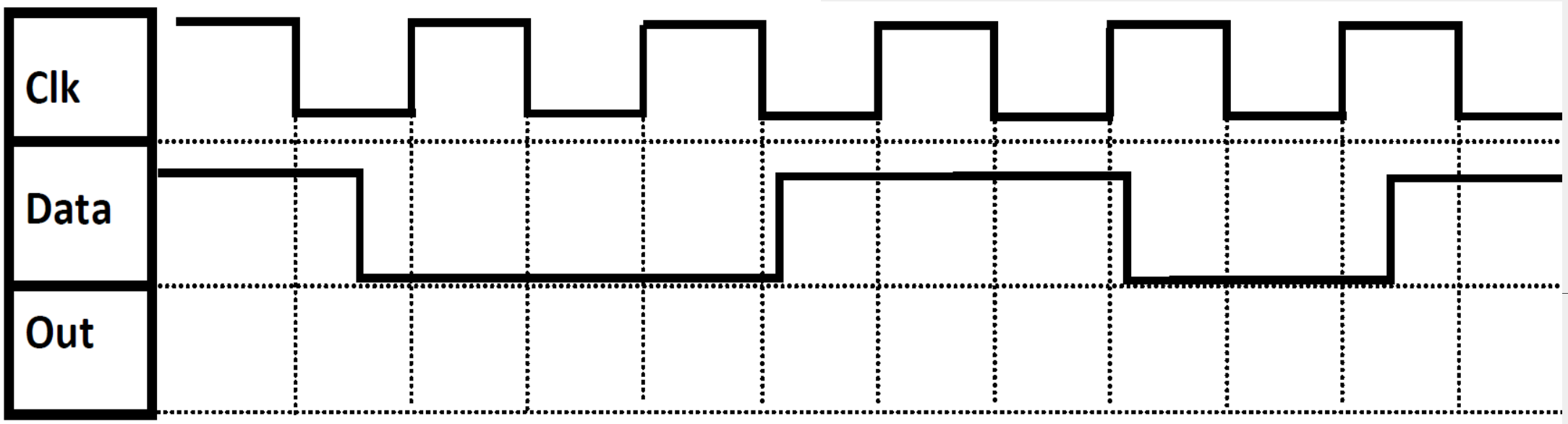
Simulation



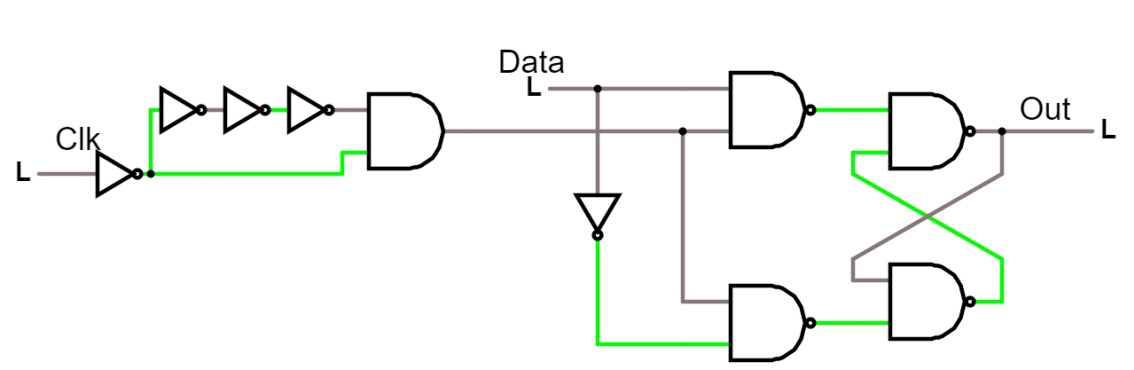
Rising edge triggered circuit Exercise – Draw the Out – 5 minutes



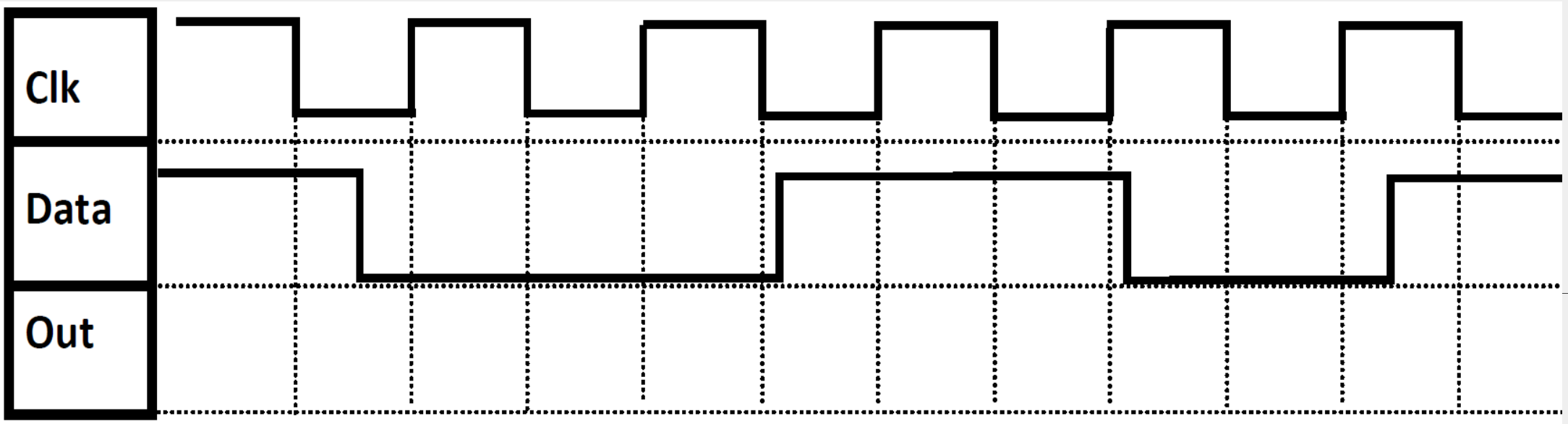
Simulation



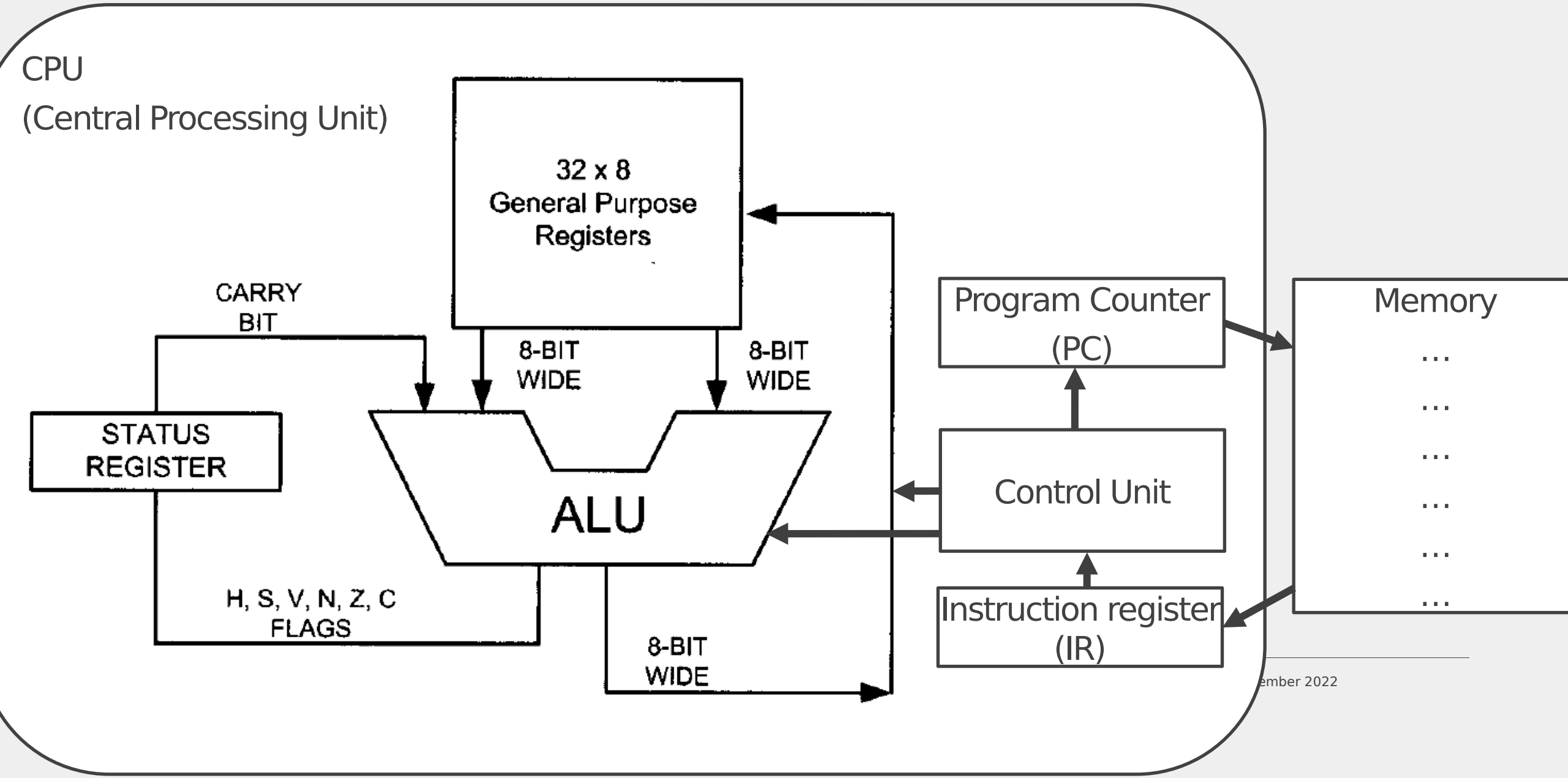
Falling edge triggered circuit
Exercise – Draw the Out – 5 minutes



Simulation



Exercise until 11.15 – Then Ill go through how the cpu works..



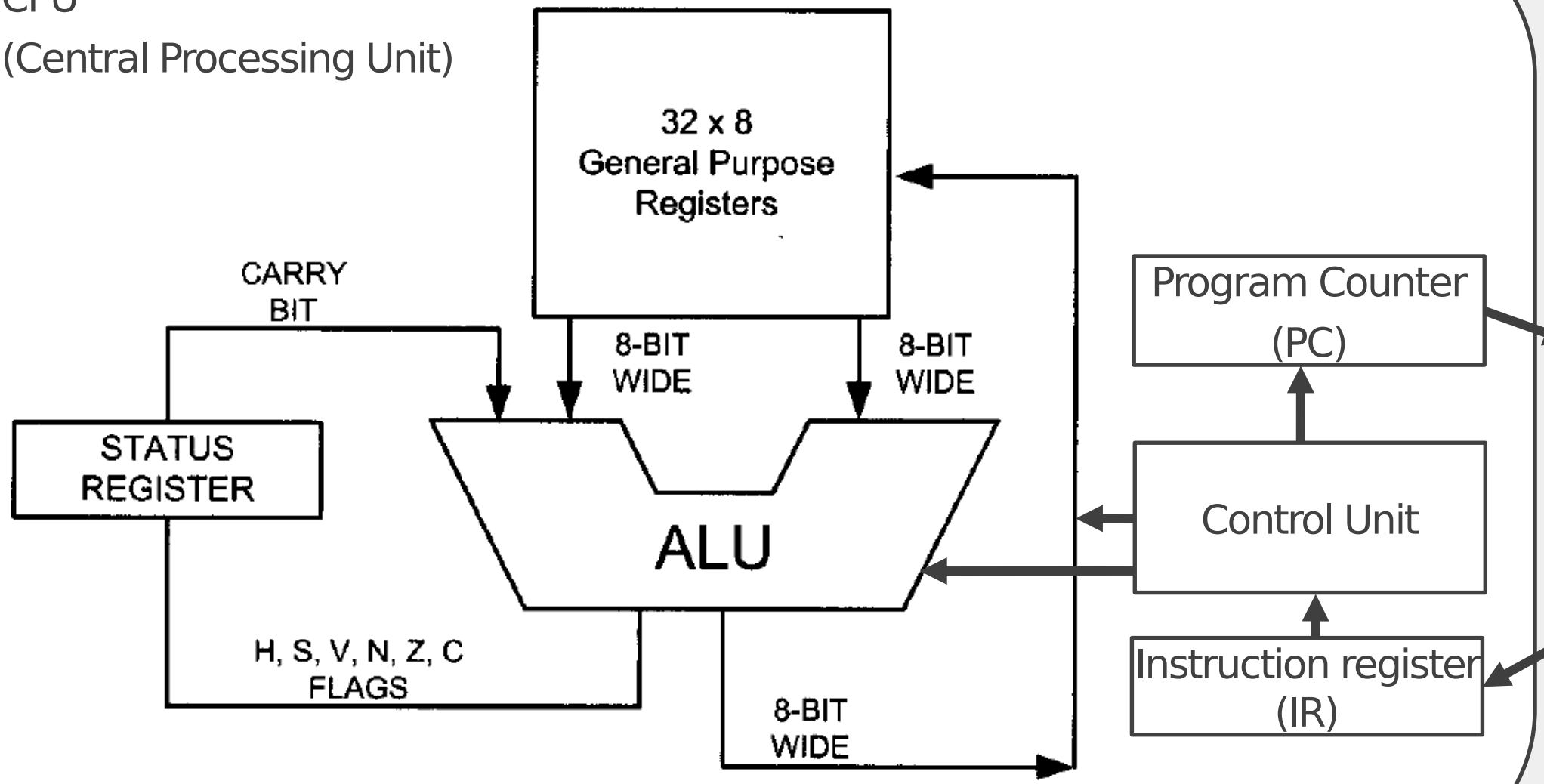
- IR contains:

0010 0000 0000 000

(look at page 3

CPU

(Central Processing Unit)



Program Counter
(PC)

Memory

...

...

...

...

...

...

Control Unit

Instruction register
(IR)

http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

Exercise 1:

- R0=00001111
- R1=11111000

IR contains:

0010 0000 0000 0001

What happens? (look at page 35)

Exercise 2:

- R0=00001111
- R1=11111000

IR contains:

0010 1000 0000 0001

What happens? (look at page 132)

Gør tanke til handling
VIA University College

Lecture 4

CPU

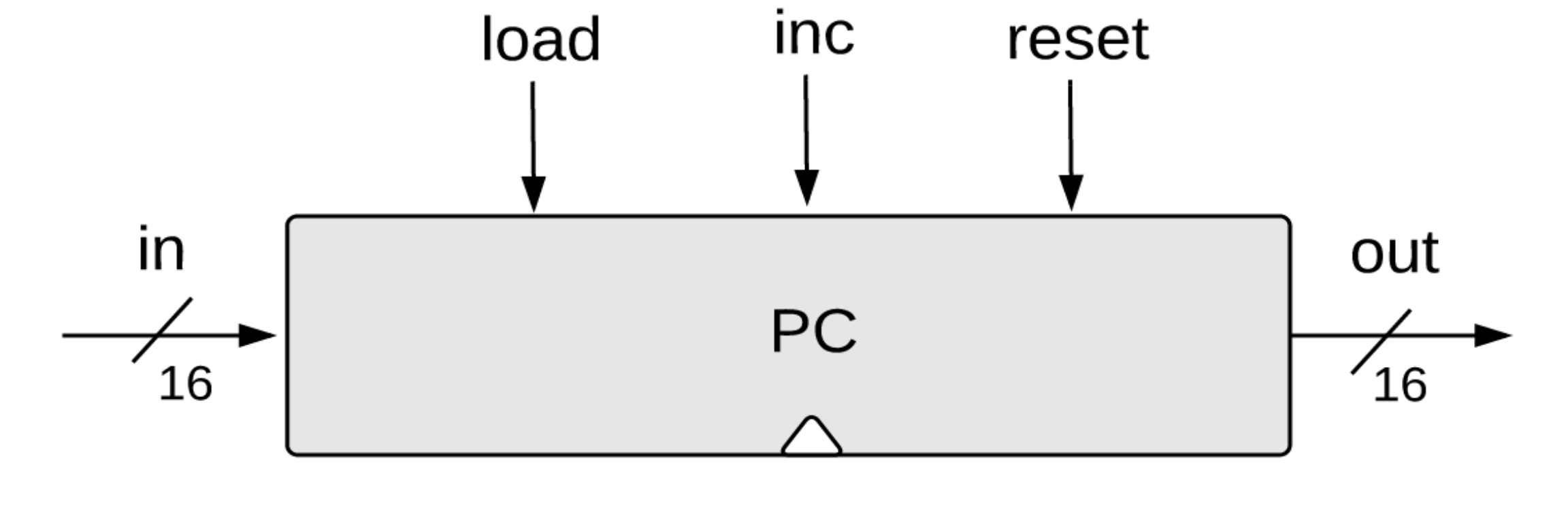
Today you will learn

- ALU
- Registers
- Program counter
- Control unit

VIA University College

Program Counter / Instruction counter

- The computer must keep track of which instruction should be fetched and executed next



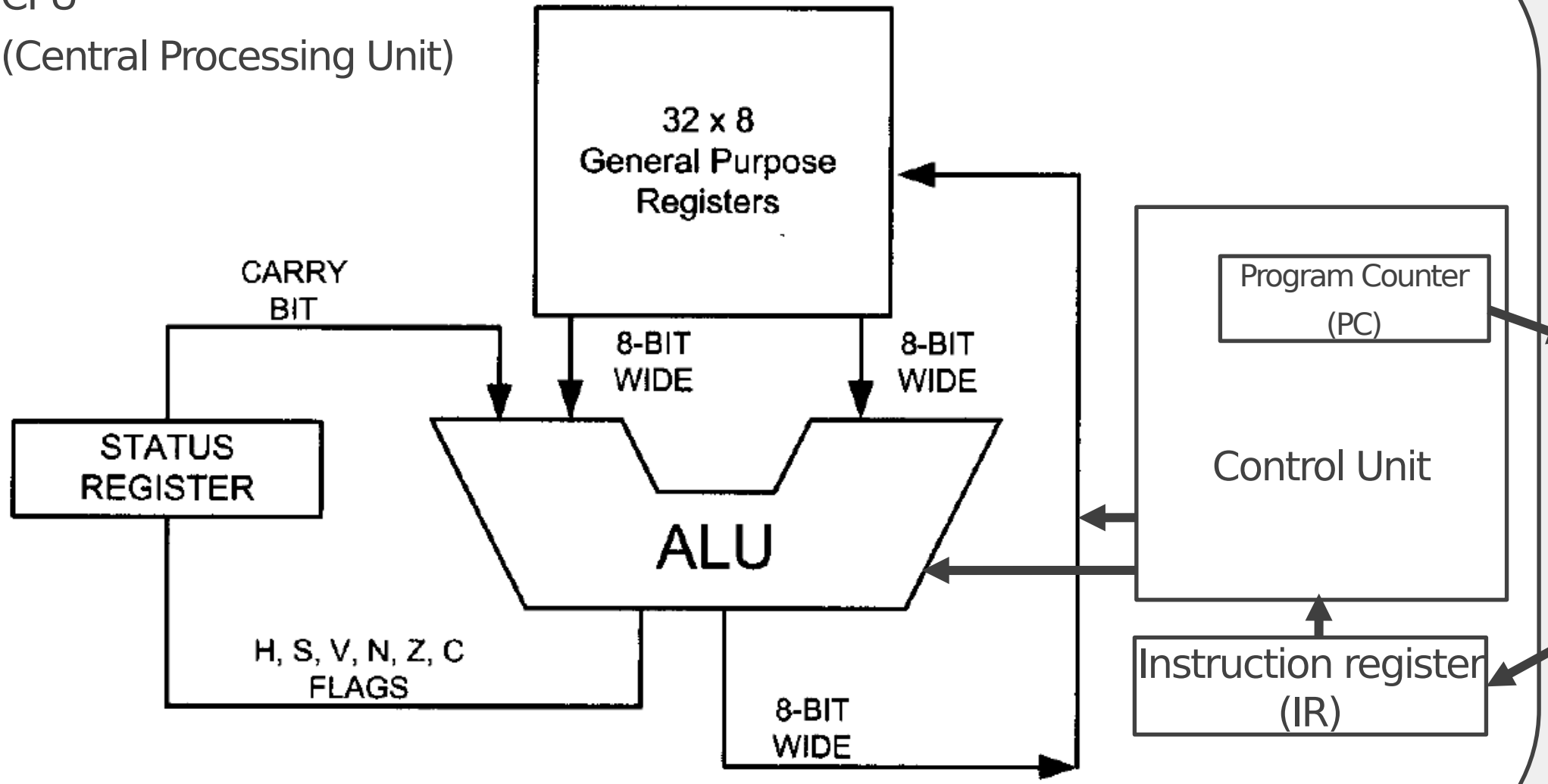
- IR contains:

0010 0000 0000 000

(look at page 3

CPU

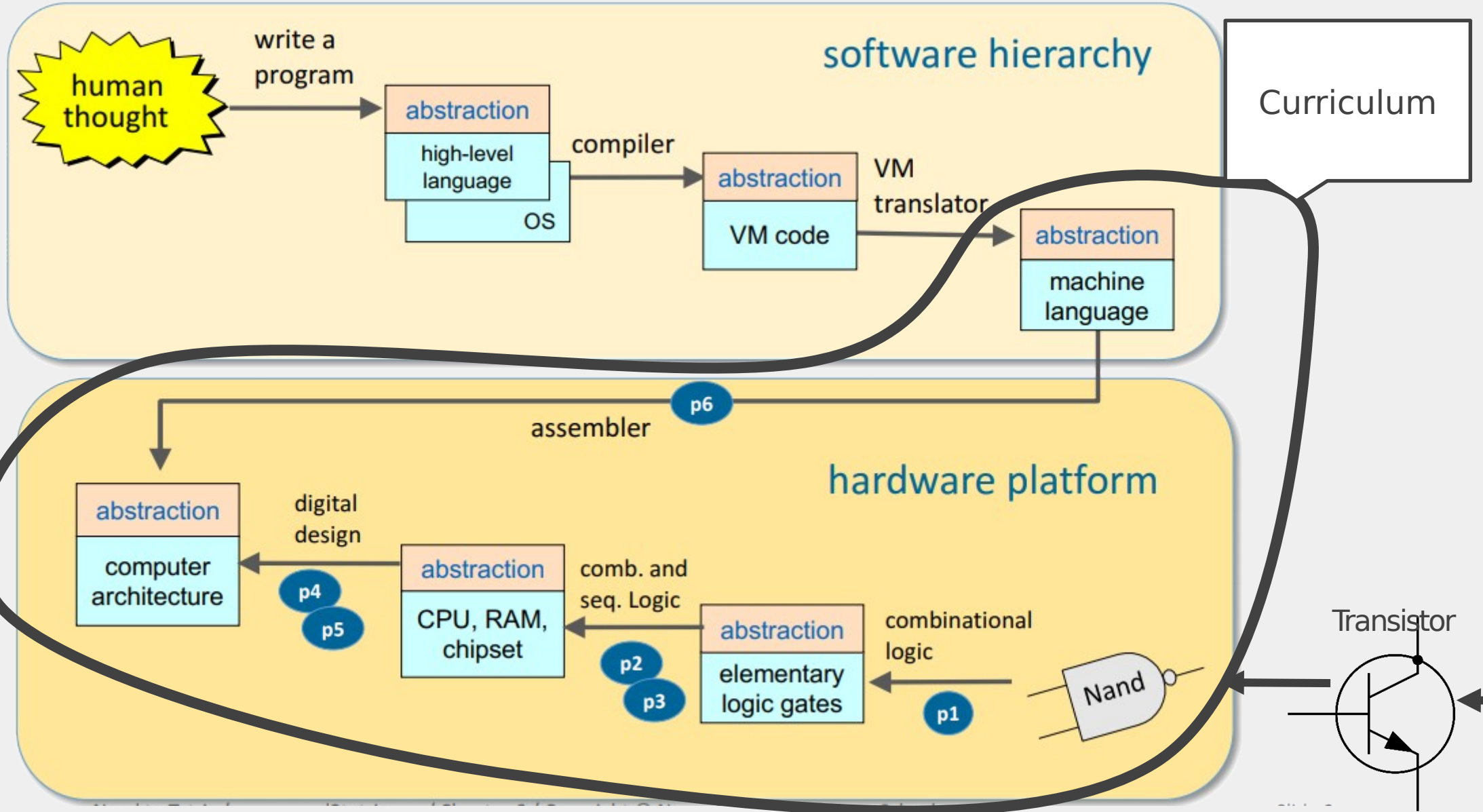
(Central Processing Unit)



CPU cycle

– Fetch	– Fetch the next instruction from memory into the instruction register – Change the PC (Program Counter) to point on the following instruction
– Decode	– Determine the type of instruction just fetched – If the instruction uses a word in memory, determine where it is – Fetch the word, if needed, into the CPU registers
– Execute	– Execute the instruction
	– Go to step 1 and execute the new instruction acoording to the PC

This course


human thought
write a program
software hierarchy
abstraction
high-level language
OS
compiler
abstraction
VM code
VM translator
abstraction
machine language
Curriculum
p6
assembler
hardware platform
abstraction
computer architecture
digital design
p4
p5
abstraction
CPU, RAM, chipset
comb. and seq. Logic
p2
p3
abstraction
elementary logic gates
combinational logic
p1
Nand
Transistor
Physics

Instruction set of the AVR

- Limited operations of the ALU
- Therefor only a limited number of instructions
- http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf
- Example: the code 1110 0001 0010 1111 is fetched by the CPU. What does it mean?
- R18=31

Exercise – 15 min

the code 1110 0100 0011 0001 is fetched by the CPU. What does it mean?

HINT: look at page 115 in

http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

Exercise – 15 min

R16 (register 16) contains the value 39 and R17 contains the value 41.

the code 0000 1111 0001 0000 is fetched by the CPU. What happens?

HINT: look at page 32 in

http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

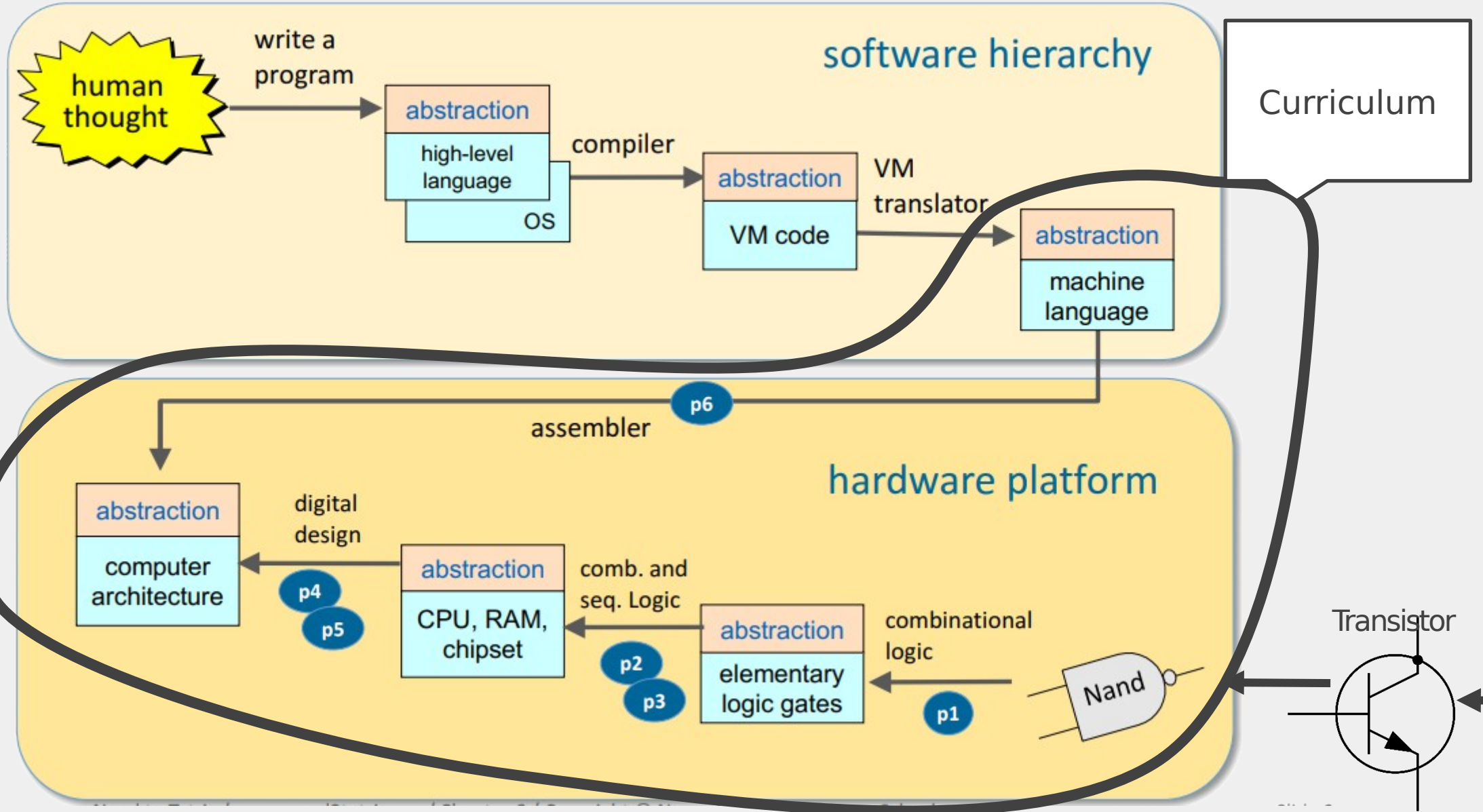
Lets do some more exercises:

- Take a break
- Then do the Lecture 4 exercises.
- I will not answer questions about the handin until later today.
- Ill continue at xxxx

The goal

- To see, that everything comes down to bits.

This course


human thought
write a program
software hierarchy
abstraction
high-level language
OS
compiler
abstraction
VM code
VM translator
abstraction
machine language
Curriculum
p6
assembler
hardware platform
abstraction
computer architecture
digital design
p4
p5
abstraction
CPU, RAM, chipset
comb. and seq. Logic
p2
p3
abstraction
elementary logic gates
combinational logic
p1
Nand
Transistor
Physics

Microprocessor vs. Microcontroller

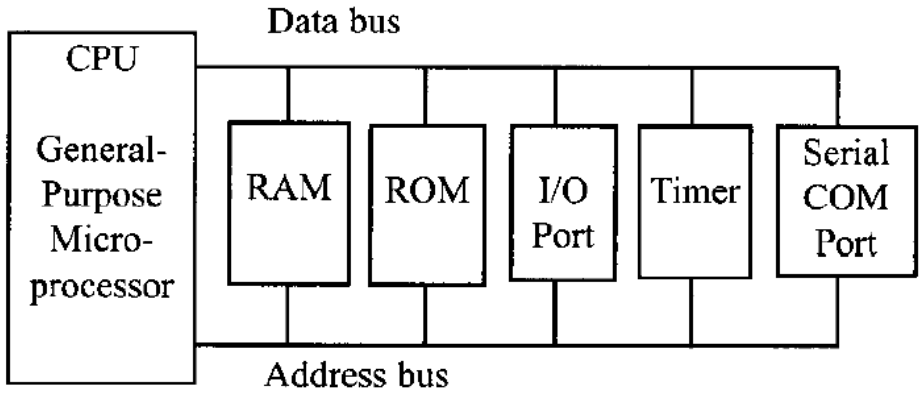
Microprocessor vs. Microcontroller

General purpose computer
- Highly configurable hw
- Many applications
- Different purposes
- Advanced user interface
- Human to human communication
- RAM is for both data and program (Harward architecture)

Embedded computer
- Fixed hw configuration
- Single application
- One purpose
- Simple user interface
- Machine to machine communication
- RAM is for data (von Neumann architecture)

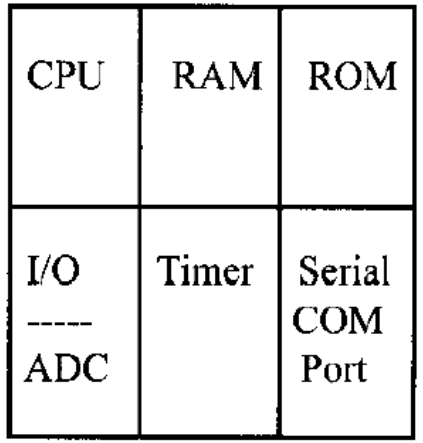
Microprocessor vs. Microcontroller

General purpose computer
- Highly configurable hw
- Many applications
- Different purposes
- Advanced user interface
- Human to human communication
- RAM is for both data and program (Harward architecture)



(a) General-Purpose Microprocessor System

Embedded computer
- Fixed hw configuration
- Single application
- One purpose
- Simple user interface
- Machine to machine communication
- RAM is for data (von Neumann architecture)



(b) Microcontroller

Mandatory Handin 1

- Feel free to hand in in groups (1 to 5 people), but its important that all group members understand all of the exercises.
- Deadline is 17th of March at 5 AM (Thursday early morning!)
- You can only upload ONCE on itslearning.
- If you don’t Pass you will have to redo it
- If there are mistakes in the assignment you will get feedback
- I’ll recommend you begin with Task 5., as you might need help.
- PDF format.

Gør tanke til handling
VIA University College

Indsæt ikon

Handin

Gør tanke til handling
VIA University College

Lecture 5

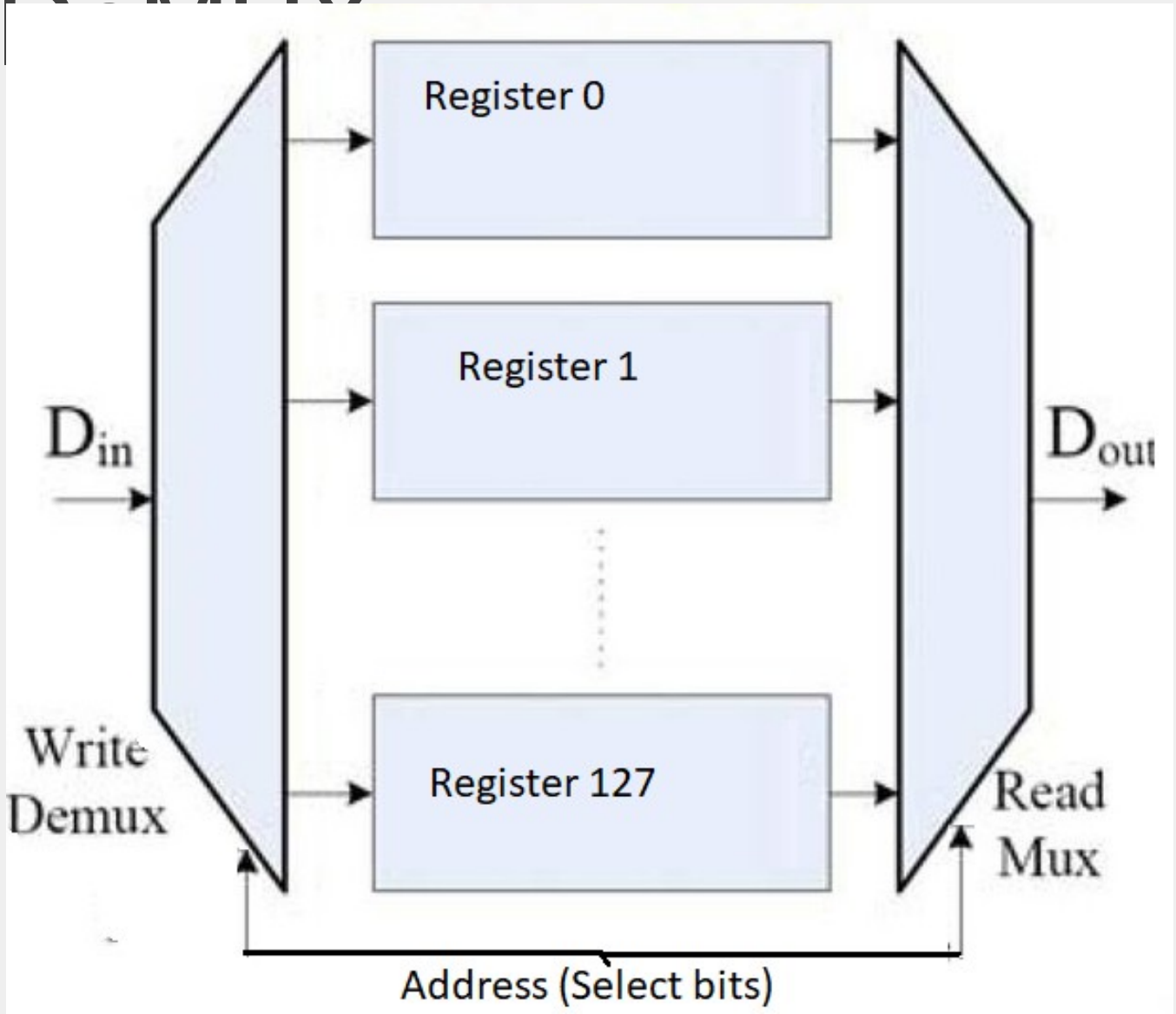
Machine language

Today you will learn

- Memory
- Assembly language
- Ill introduce microchipstudio and the simulator.Lecture

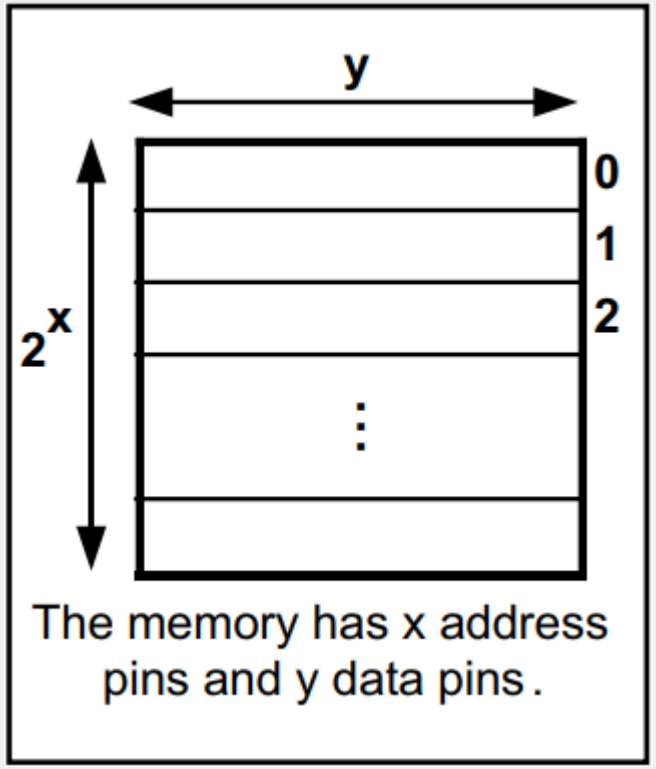
VIA University College

Picture of RAM including MUX and DeMUX



Memory

- A memory chip contains locations, where x is the number of address pins.
- Each location contains y bits, where y is the number of data pins on the chip
- The entire chip will contain × y bits, where x is the number of address pins and y is the number of data pins on the chip.



Example 1

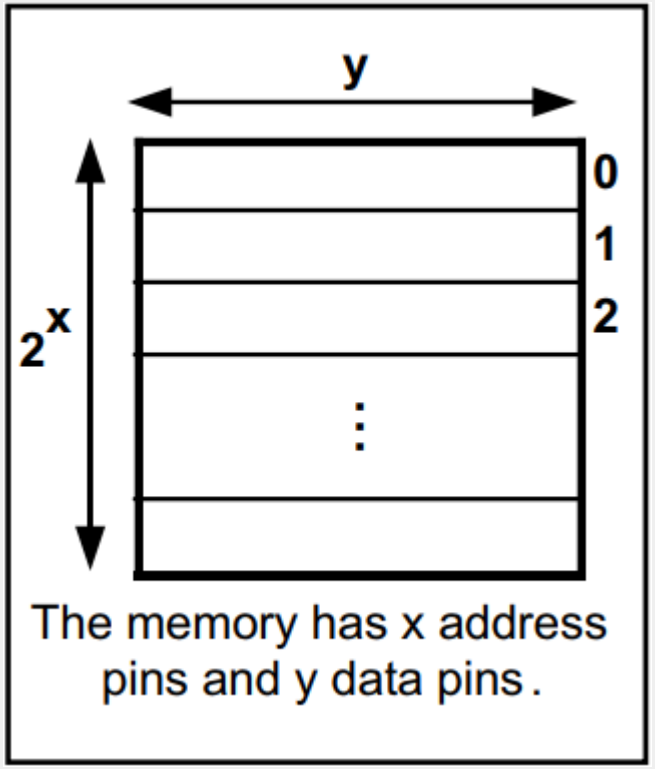
A given memory chip has 12 address pins and 8 data pins.

a) How many Adresses does it have?

b) Whats the total capacity?

a)

b)



Capacity:
What’s the difference?

8Kb vs 8KB

Capacity:
What’s the difference?

8Kb vs 8KB

8Kbit 8Kbyte

Capacity:
What’s the difference?

8kB vs 8KB

Capacity:
What’s the difference?

8kB	vs	8KB
8kilobyte		8kibibyte [KiB]

Capacity:
What’s the difference?

8kB	vs	8KB
8kilobyte		8kibibyte [KiB]
8000byte		byte

Multiples of bytes

V · T · E

Decimal			Binary				
Value	Metric		Value	IEC		JEDEC	
1000	kB	kilobyte	1024	KiB	**kibibyte**	KB	kilobyte
1000^2	MB	megabyte	1024^2	MiB	mebibyte	MB	megabyte
1000^3	GB	gigabyte	1024^3	GiB	gibibyte	GB	gigabyte
1000^4	TB	terabyte	1024^4	TiB	tebibyte		–
1000^5	PB	petabyte	1024^5	PiB	pebibyte		–
1000^6	EB	exabyte	1024^6	EiB	exbibyte		–
1000^7	ZB	zettabyte	1024^7	ZiB	zebibyte		–
1000^8	YB	yottabyte	1024^8	YiB	yobibyte		–

Orders of magnitude of data

Table is copied from Wikipedia

Example 2

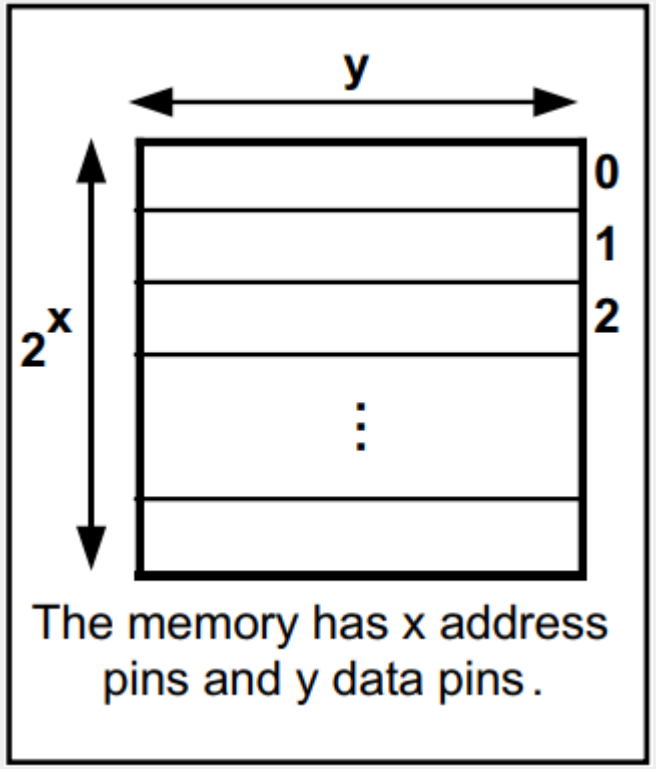
A memory chip with the total capacity of 512Kib, has 8 data pins

(a) How many addresses does it have?

(b) How many address pins does it have?

a)

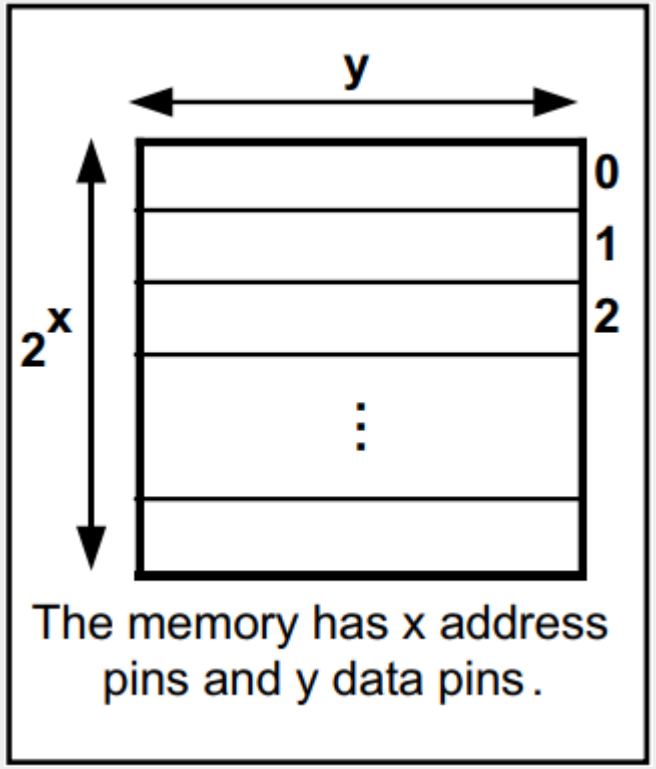
b) As , 16 address pins is necessary ()



Exercise 10 minutes

A given memory chip has 14 address pins and 8 data pins.

a) How many Adresses does it have?
b) What's the total capacity in bits?
c) What's the total capacity in bytes?
d) What's the total capacity in kb?
e) What's the total capacity in Kb?
f) What's the total capacity in kB?
g) What's the total capacity in KB?



A given memory chip has 14 address pins and 8 data pins.

a) How many Adresses does it have?

b) What's the total capacity in bits?

c) What's the total capacity in bytes?

d) What's the total capacity in kb?

e) What's the total capacity in Kb?

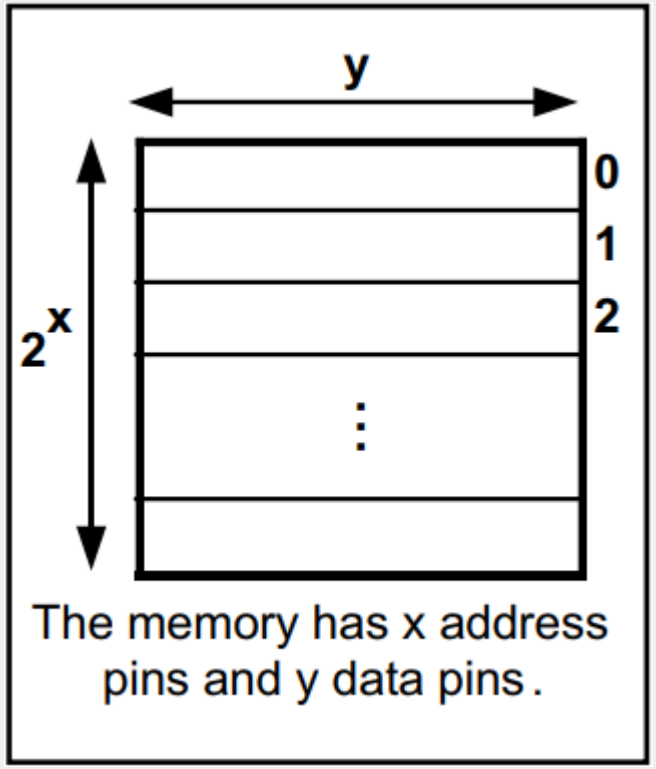
f) What's the total capacity in kB?

g) What's the total capacity in KB?

Exercise 8 min

A given memory chip has 12 address pins and 16 data pins.

a) How many Adresses does it have?
b) What's the total capacity in bits?
c) What's the total capacity in bytes?
d) What's the total capacity in KB?



Instruction set

6. ADD – Add without Carry

6.1. Description

Adds two registers without the C Flag and places the result in the destination register Rd.

Operation:

(i) (i) Rd ← Rd + Rr

	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

0000	11rd	dddd	rrrr

6.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	⇔	⇔	⇔	⇔	⇔	⇔

R16 = 200

R17 = 200

ADD R16, R17

Assembly Language

- It is hard to write machinecode
- Assembly language is a low level programming language
- This language is so low level, that you can only do exactly what the CPU can do.

Assembler

- The **assembler** translate assembly code to machine language
- A bit similar to a compiler, but way more simple.

Example Assembly

Instruction nr.	Assembly	Machine code	Explanation
1	**LDI** R17, 64 ;		
2	**LDI** R18, 130 ;		
3	**ADD** R17, R18 ;		

Instructions set

Example Assembly - ANSWER

Instruction nr.	Assembly	Machine code	Explanation
1	**LDI** R17, 64 ;	1110 0100 0001 0000 0x E 4 1 0	load 64 into register17
2	**LDI** R18, 130 ;	1110 1000 0010 0010 0x E 8 2 2	load 130 into register 18
3	**ADD** R17, R18 ;	0000 1111 0001 0010 0X 0 F 1 2	add r17 and r18 and place the result in r17

The Assembler is platform specific!

- This instruction set wont work on your laptop
- Or any other platform

Exercise 1 (1.1 - 1.3) and 2 (2.1 to 2.8)

60 minutes.

Then I will introduce microchip studio

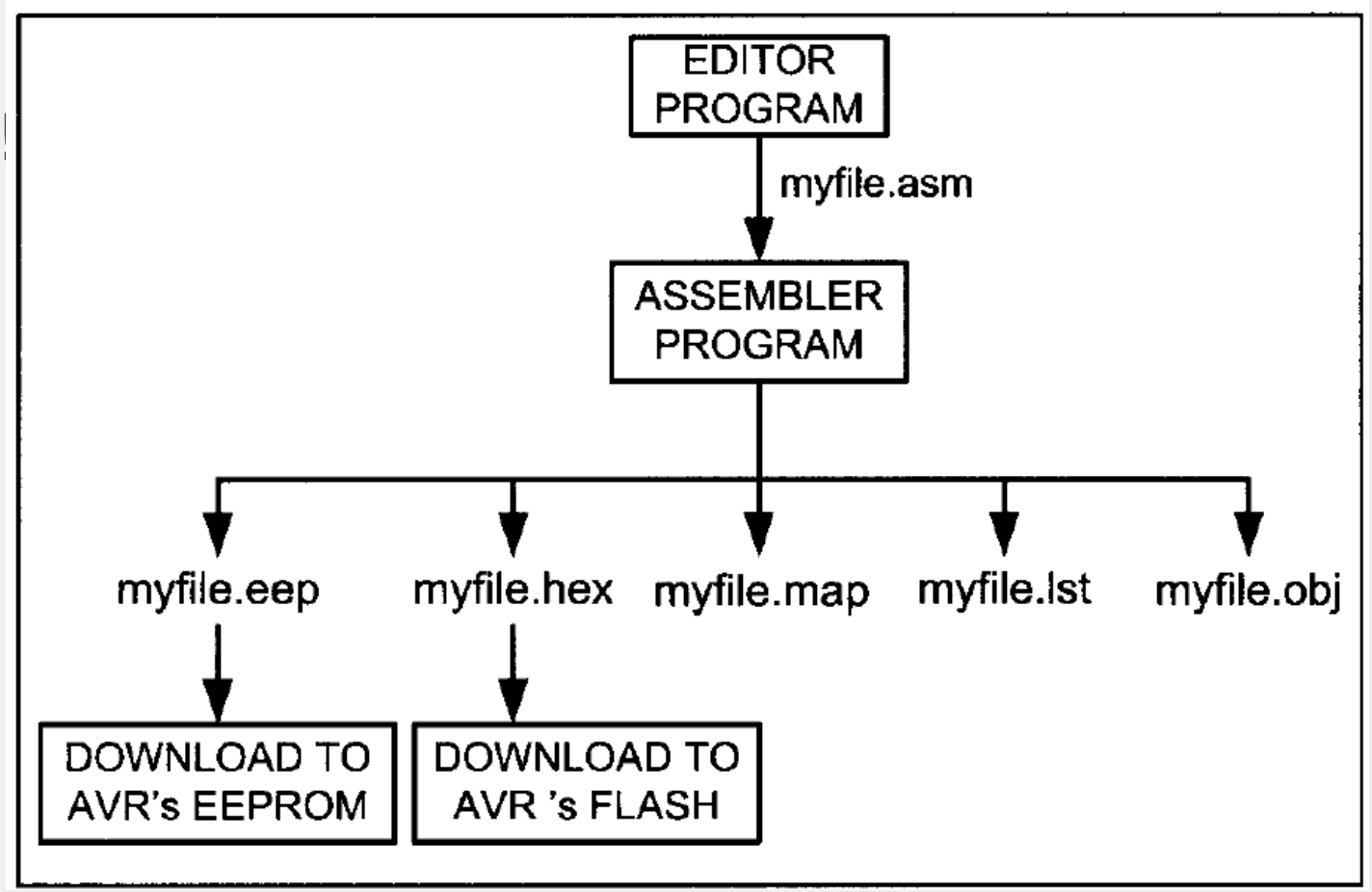
And then do the rest of the exercises.

Example Assembly – ANSWER
Lets look in the program memory.

Instruction nr.	Assembly	Machine code	Explanation
1	LDI R17, 64 ;	1110 0100 0001 0000 0x E 4 1 0	load 64 into register17
2	LDI R18, 130 ;	1110 1000 0010 0010 0x E 8 2 2	load 130 into register 18
3	ADD R17, R18 ;	0000 1111 0001 0010 0X 0 F 1 2	add r17 and r18 and place the result in r17

Assembly Language

! Use the instruction set
m2560def.inc
.lss



Example Assembly – ANSWER
Lets look in the program memory.

```
LDI r16, 0xAA
LDI R17, 0xBB
ldi R18, 0xCC
```

Gør tanke til handling
VIA University College

Lecture 6

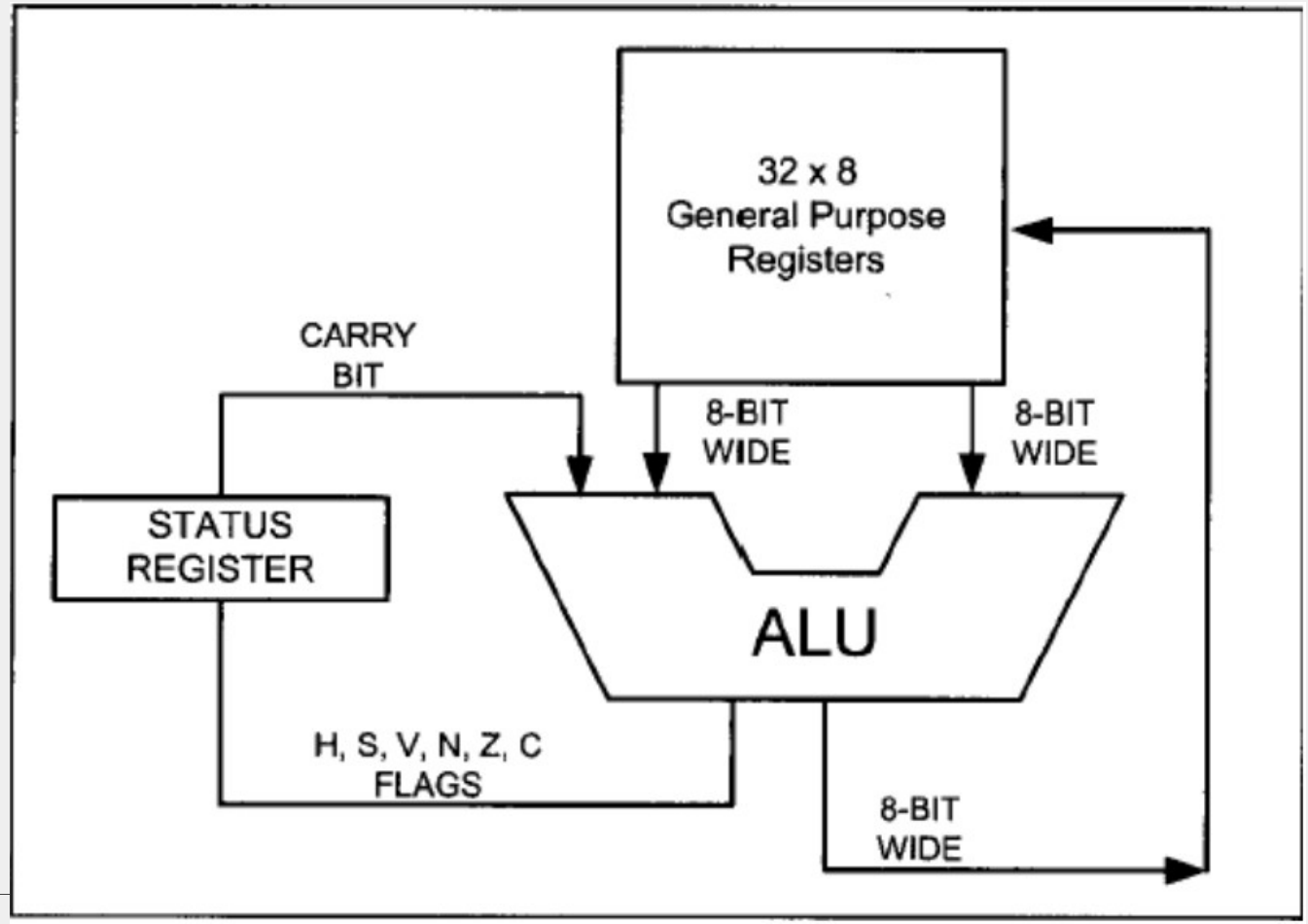
AVR introduction

Today you will learn

VIA University College

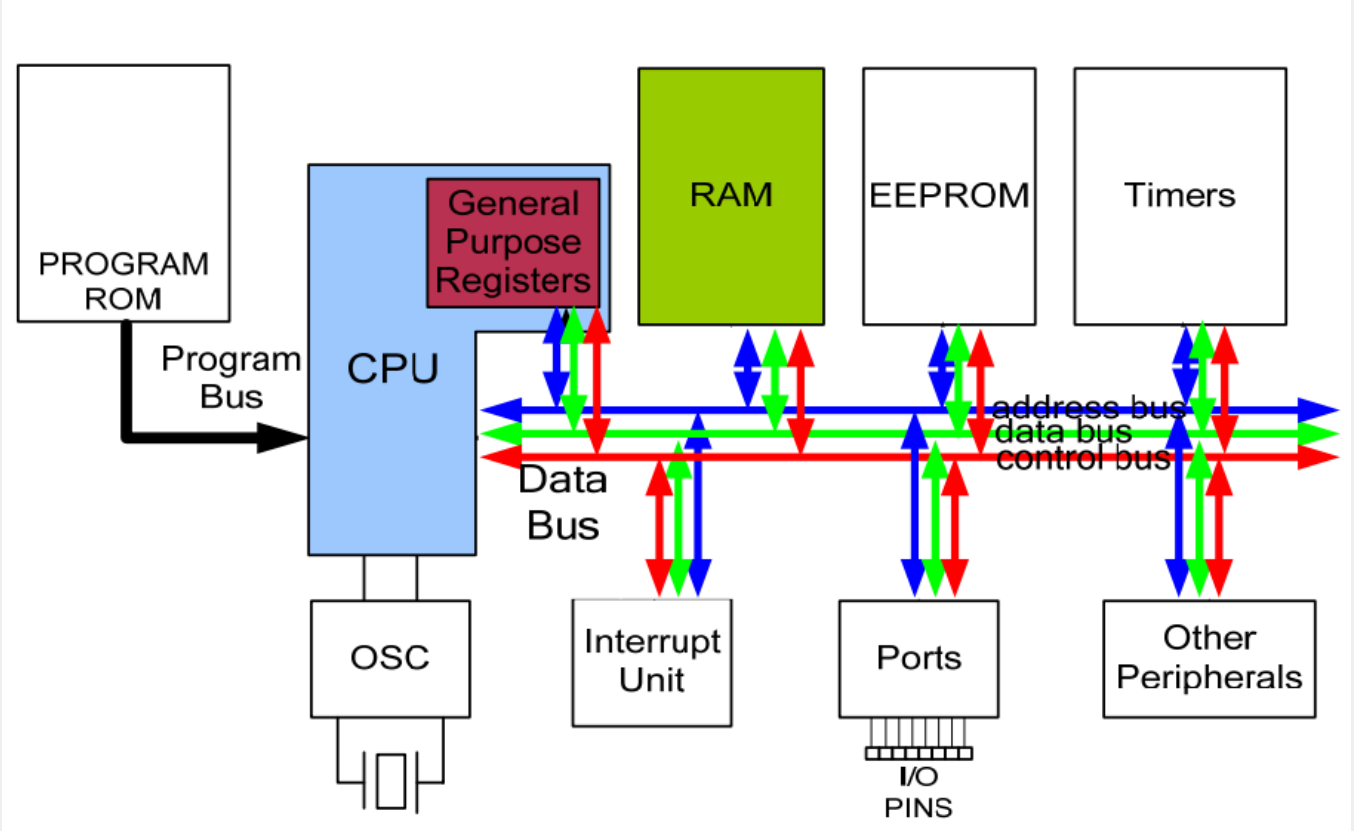
General purpose registers

- Data is loaded in here before it can be used.



General purpose registers

```
LDI R20, 38;
LDI R23, 100;
ADD R20, R23;
```



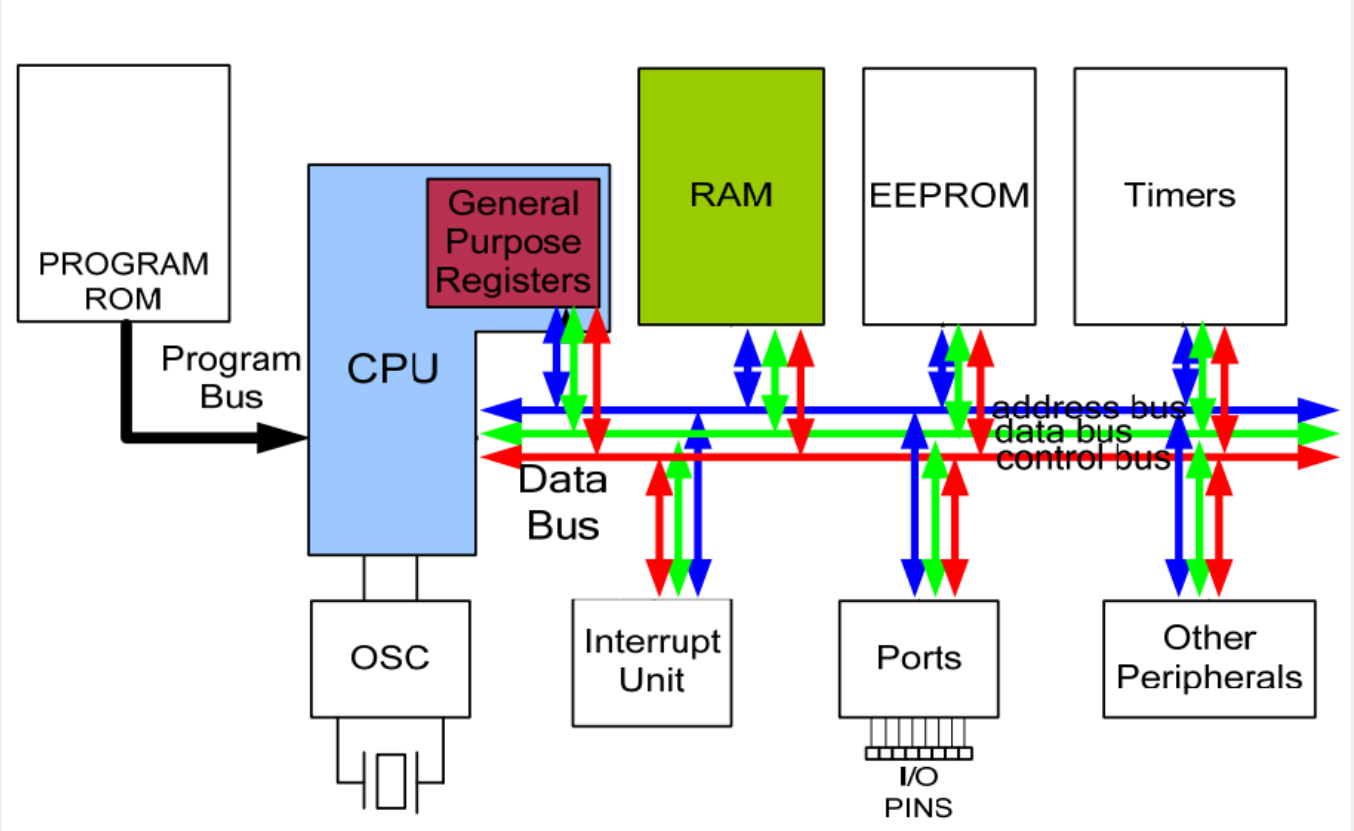
Data memory – Store to data Space

- You can load data to the RAM:

You want to Load the value 100 into the first address of the SRAM

```
LDI R23, 100;
STS 0x200, R23;
```

OR `STS SRAM_START, R23`



Data memory – Load Direct from data Space

LDS R20, 0x0200

Example, where we first STS then LDS the same address.

121. STS – Store Direct to Data Space

121.1. Description

Stores one byte from a Register to the data space. For parts with SRAM, the data space consists of the Register File, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the Register File only. The EEPROM has a separate address space.

A 16-bit address must be supplied. Memory access is limited to the current data segment of 64KB. The STS instruction uses the RAMPD Register to access memory above 64KB. To access another data segment in devices with more than 64KB data space, the RAMPD in register in the I/O area has to be changed.

This instruction is not available in all devices. Refer to the device specific instruction set summary.

Operation:

(i) (k) ← Rr

	Syntax:	Operands:	Program Counter:
(i)	STS k,Rr	0 ≤ r ≤ 31, 0 ≤ k ≤ 65535	PC ← PC + 2

32-bit Opcode:

1001	001d	dddd	0000
kkkk	kkkk	kkkk	kkkk

121.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Example:

```
lds r2,$FF00 ; Load r2 with the contents of data space location $FF00
add r2,r1 ; add r1 to r2
sts $FF00,r2 ; Write back
```

Words	2 (4 bytes)
Cycles	2

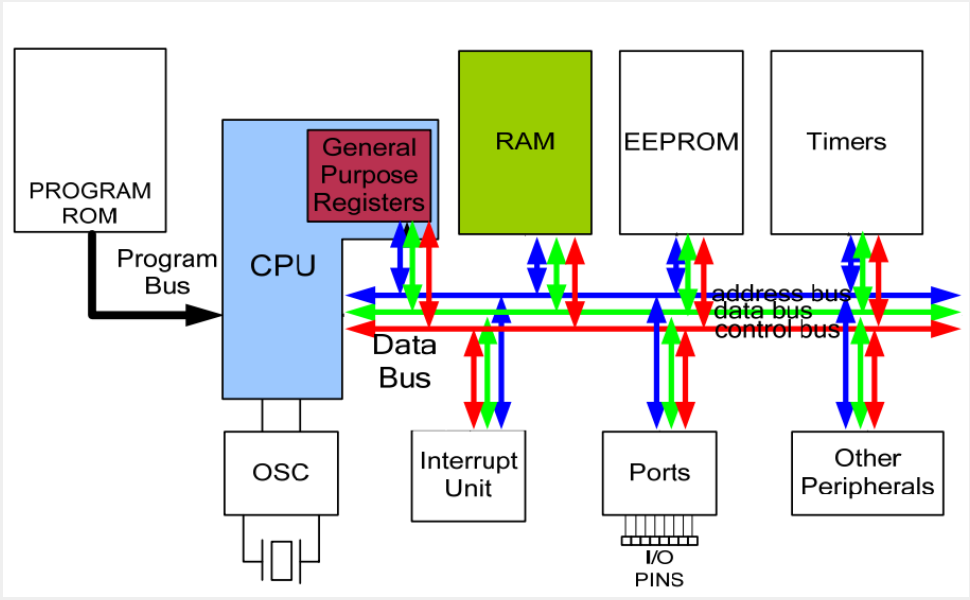
Exercise 10 minutes

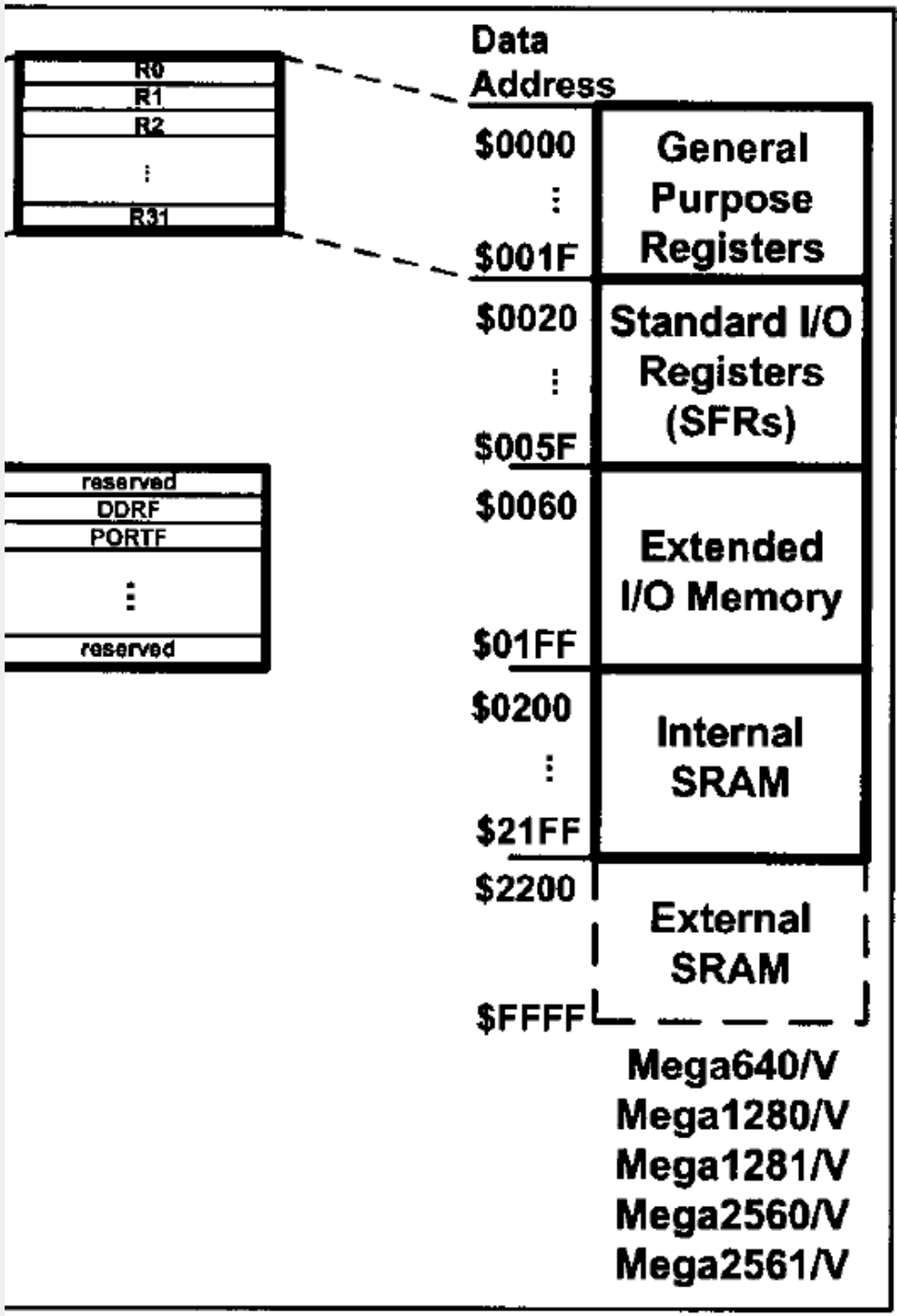
- Add two numbers together
- Then store the results in memory address 0x0201, using STS
- Check that the number is there, in the debugger.

- Then read the value from the RAM using LDS

Input Output

- We can Access these using STS and LDS
- However In and Out can also be used. These are faster, and the address is a bit different.

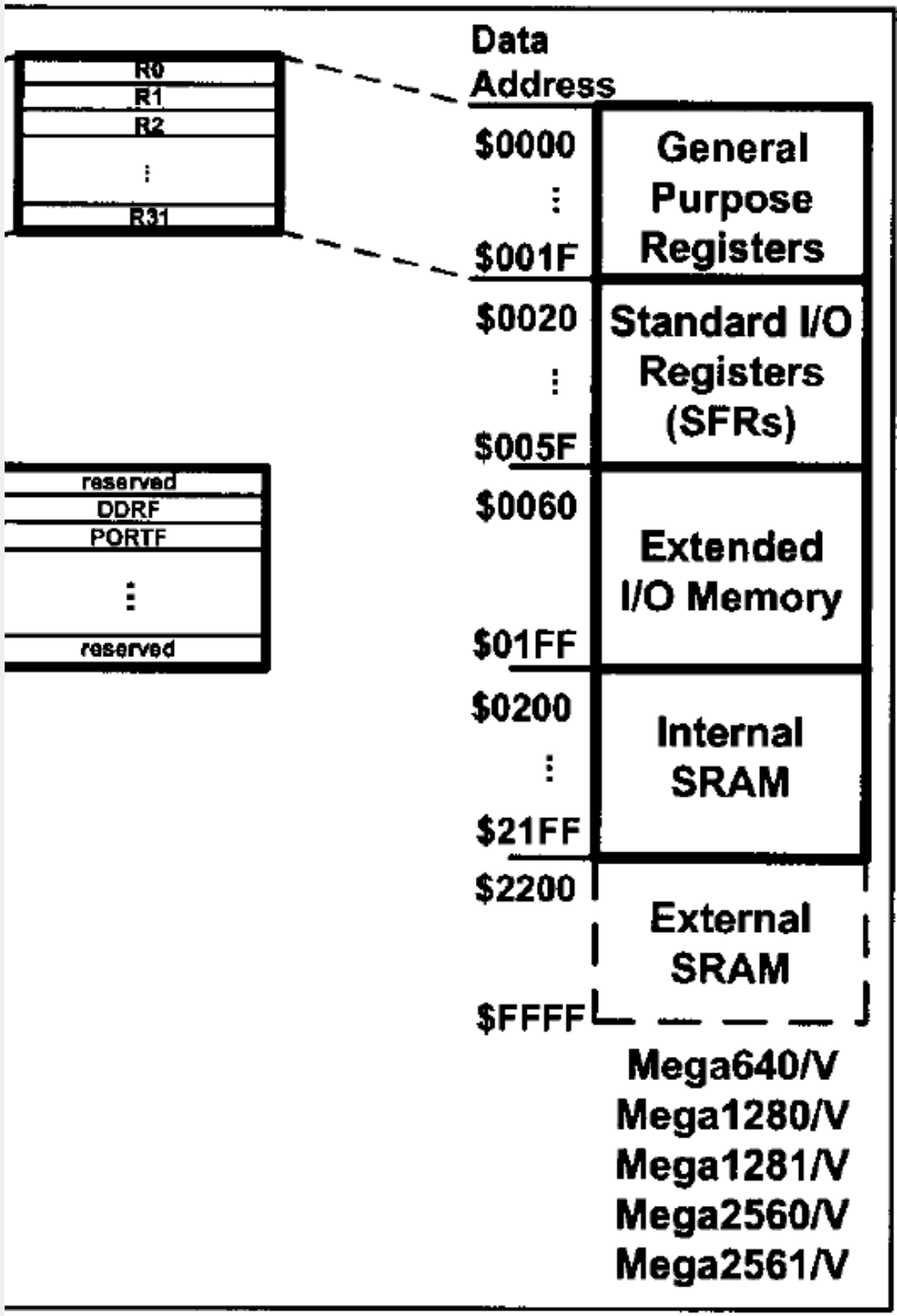




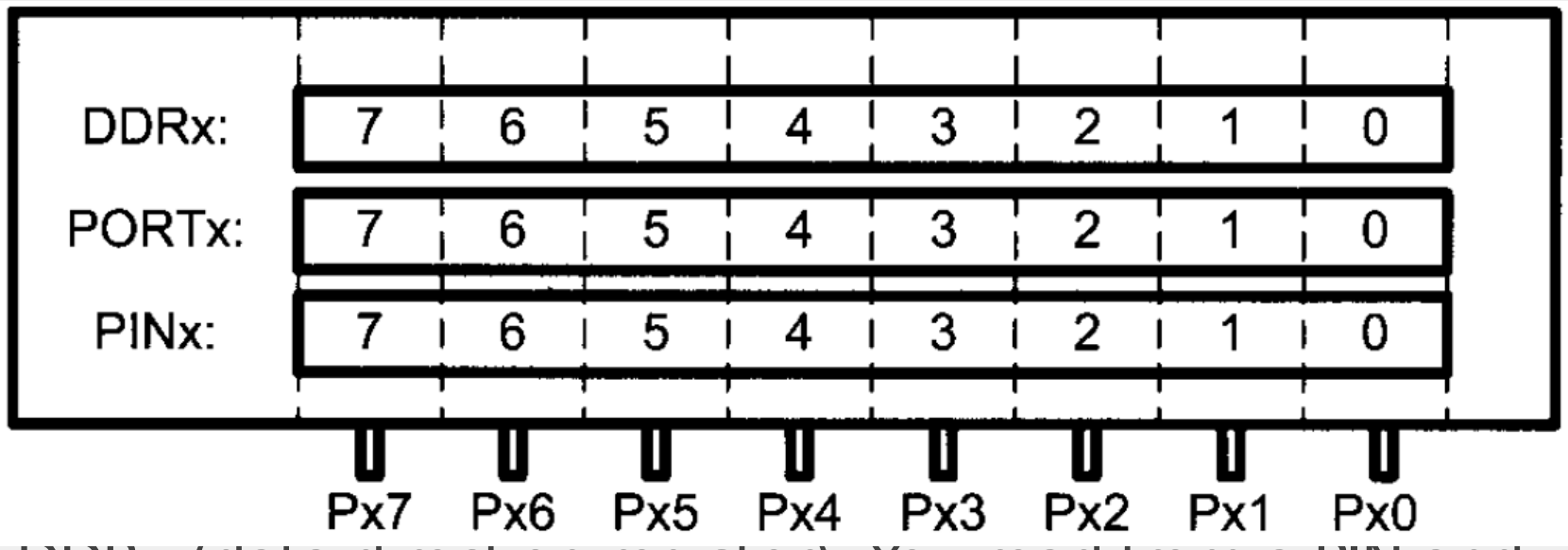
Input Output

- We can Access these using STS and LDS
- However In and Out can also be used. These are faster, and the address is a bit different.

```
ldi r16, 0xaa
sts 0x25, r16

ldi r16, 0xbb
out 0x05, r16

ldi r16, 0xcc
out PORTB, r16
```



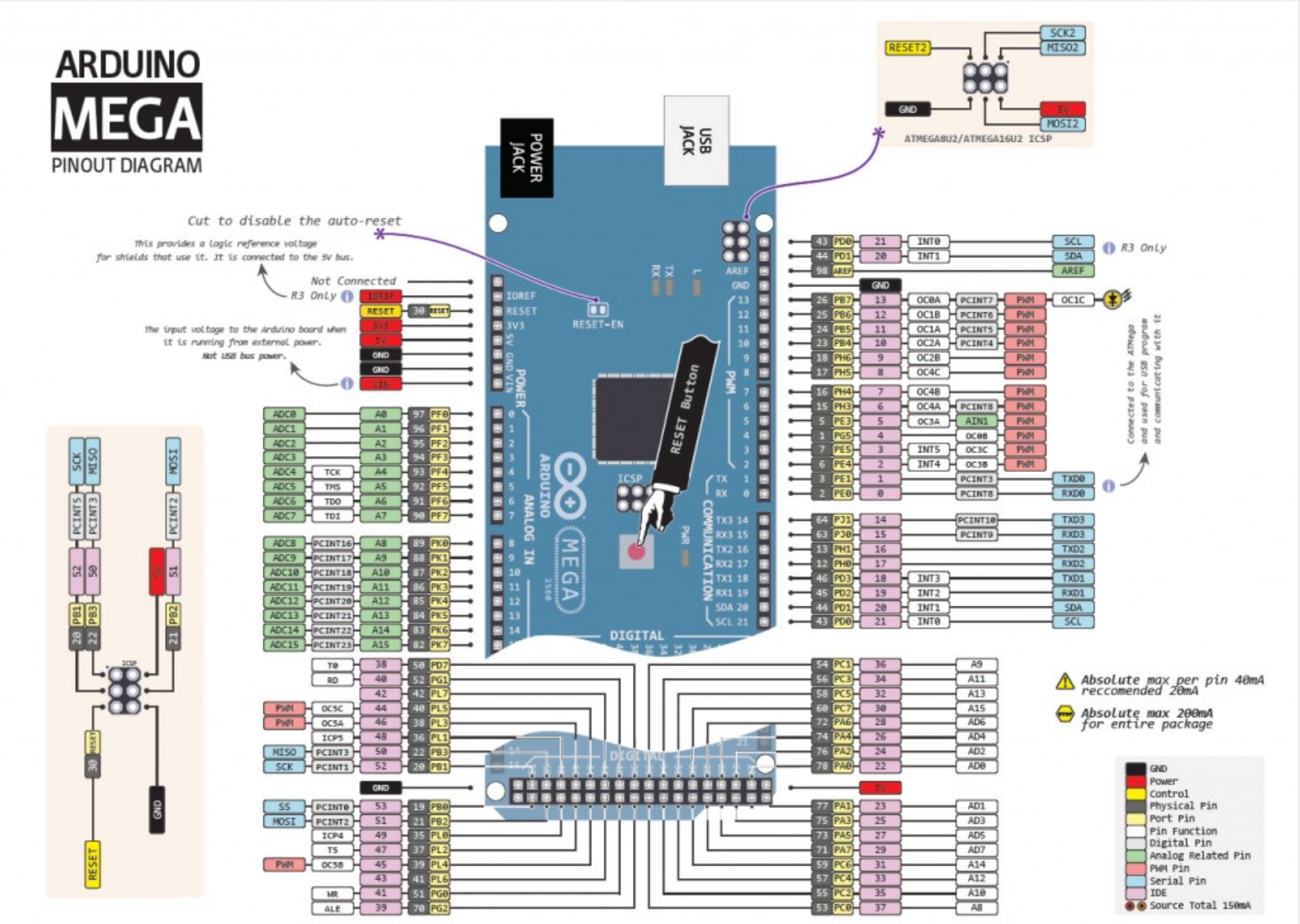
3 registers for each Port/Pin



DDRx (data direction register). You read from a PIN and write to a PORT.

e.g. DDRA = 0b11110000 -> configure PA7, PA6, PA5 and PA4 to be outputs.

Do exercise 1 and 2 for 60 minutes



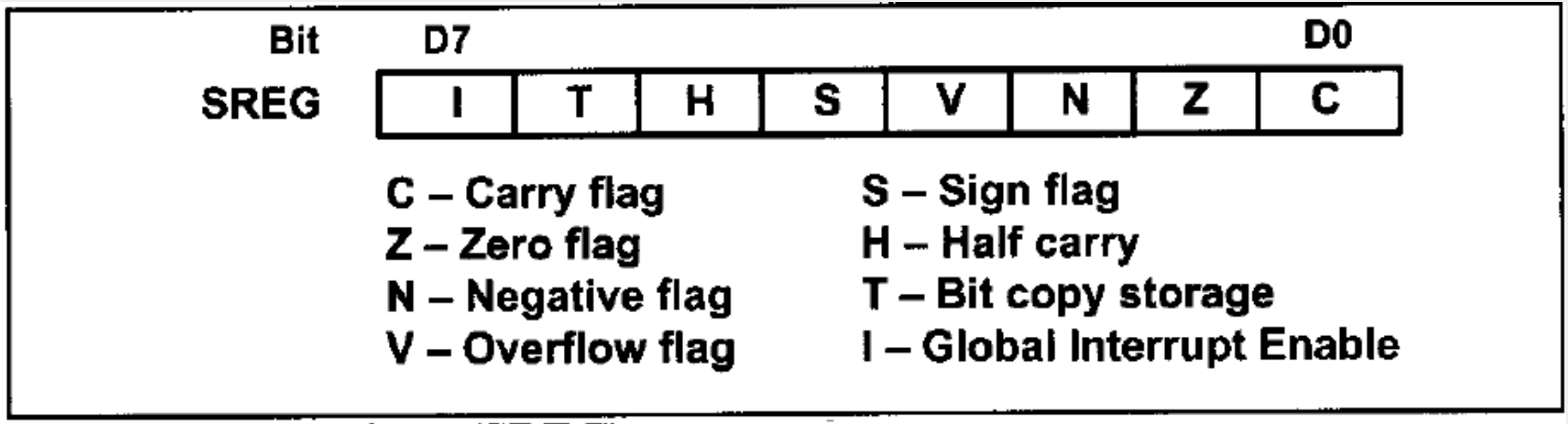
MOV instruction

- Live Demo

```
ldi r16, 0xaa
mov r17,r16
```

Status register

- Shows operations change the status register



Bits of Status Register (SREG)

ADD example

- ldi r21, 255;
- ldi r20,2;
- add r20,r21;

Debug ->Windows ->
Processor Status
(need to be in debug mode!)

Table 2-4: Instructions That Affect Flag Bits

Instruction	C	Z	N	V	S	H
ADD	X	X	X	X	X	X
ADC	X	X	X	X	X	X
ADIW	X	X	X	X	X	
AND		X	X	X	X	
ANDI		X	X	X	X	
CBR		X	X	X	X	
CLR		X	X	X	X	
COM	X	X	X	X	X	
DEC		X	X	X	X	
EOR		X	X	X	X	
FMUL	X	X				
INC		X	X	X	X	
LSL	X	X	X	X		X
LSR	X	X	X	X		
OR		X	X	X	X	
ORI		X	X	X	X	
ROL	X	X	X	X		X
ROR	X	X	X	X		
SEN			1			
SEZ		1				
SUB	X	X	X	X	X	X
SUBI	X	X	X	X	X	X
TST		X	X	X	X	

Note: X can be 0 or 1. (See Chapter 5 for how to use these instructions.)

JMP instruction

Jmp 0 ;

Better to:

Jmp label;

66. JMP – Jump

66.1. Description

Jump to an address within the entire 4M (words) Program memory. See also RJMP.

This instruction is not available in all devices. Refer to the device specific instruction set summary.

Operation:

(i) PC ← k

	Syntax:	Operands:	Program Counter:	Stack:
(i)	JMP k	0 ≤ k < 4M	PC ← k	Unchanged

32-bit Opcode:

1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

66.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Example:

```
 mov r1,r0 ; Copy r0 to r1
 jmp farplc ; Unconditional jump
 ...
farplc: nop ; Jump destination (do nothing)
```

Words	2 (4 bytes)
Cycles	3

Dataformats

- Hex, bin, decimal, ascii table

- 0x6b
- $6B
- 0b0110 1011
- 107
- ‘k’

Exercises

```
: Beginning of file
jmp 3
jmp 0xEA0A
nop
```

Gør tanke til handling
VIA University College

Indsæt ikon

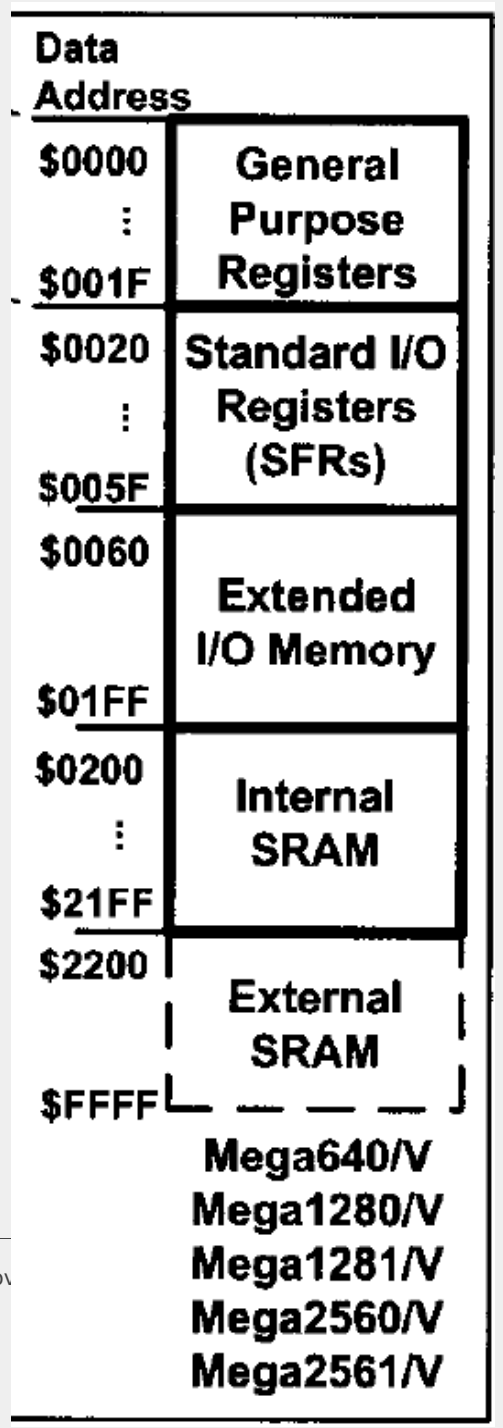
Last week:

2.3 Connect at least 4 LEDs to ports of your own choosing, and turn 2 of them On and 2 of them Off.

What is wrong with "STS PORTA, R23"? What does it do?

3.7 Whats the C and Z flag after the following code has been executed. Afterward, test it using the simulator.

```
LDI R20, 0x9F
LDI R21, 0x61
ADD R20, r21
```



Gør tanke til handling
VIA University College

Lecture 7

Branch, Call and time delay Loop

At the end of today you will be able to create a program that makes a led blink.

Today you will learn

VIA University College

Stack motivation

- Needs to store a lot of temporary data.
- When calling a function, the return address needs to be saved somewhere
- A function parameter

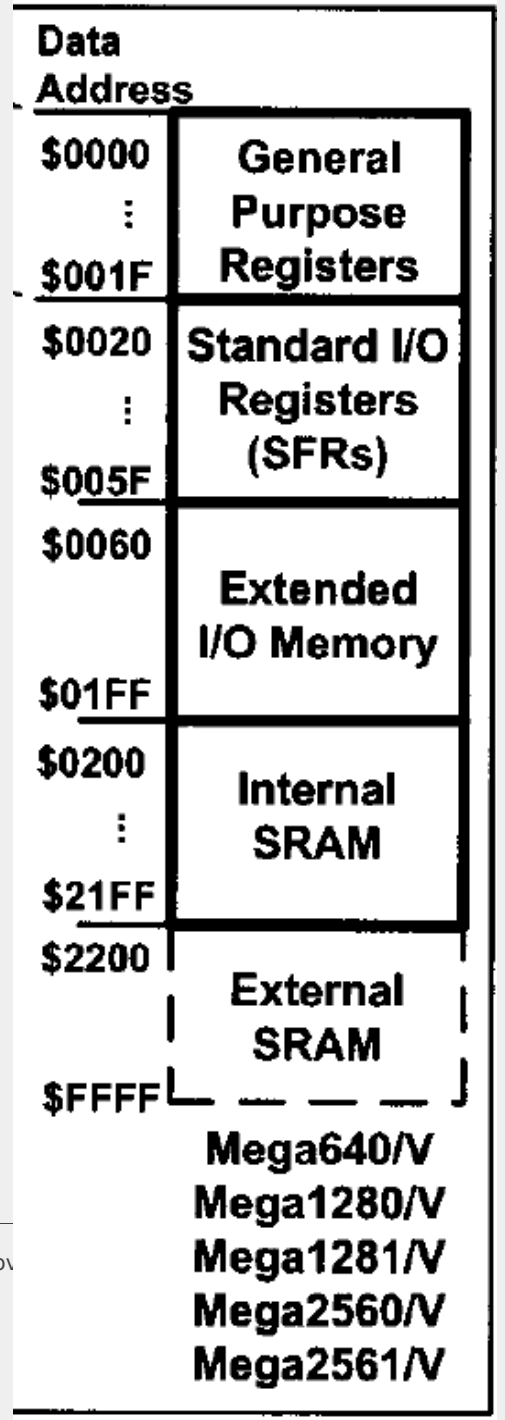
The Stack

- The stack is a section of the RAM which are used to store information temporarily.
- A stack pointer (a register is located in the CPU) is used to point on the top of the stack

More about stack

- The stack can be anywhere in the RAM (where ever the SP is pointing)
- It makes sense to use the Last address of the RAM.
- 8KB RAM -> 8*1024 = 8192=0x2000
- The first RAM address has the address 0x0200.
- So the last address is 0x2200-1 = 0x21FF
- SPL = 0xFF
- SPH = 0x21

-1 since the first address is 0



Stack example

- Show the SP in the Processor Status.

More about stack

PUSH – Push Register on Stack

Description

This instruction stores the contents of register Rr on the STACK. The Stack Pointer is post-decremented by 1 after the PUSH.

This instruction is not available in all devices. Refer to the device specific instruction set summary.

Operation:

(i) STACK ← Rr

	Syntax:	Operands:	Program Counter:	Stack:
(i)	PUSH Rr	0 ≤ r ≤ 31	PC ← PC + 1	SP ← SP - 1

16-bit Opcode:

1001	001d	dddd	1111

POP – Pop Register from Stack

Description

This instruction loads register Rd with a byte from the STACK. The Stack Pointer is pre-incremented by 1 before the POP.

This instruction is not available in all devices. Refer to the device specific instruction set summary.

Operation:

(i) Rd ← STACK

	Syntax:	Operands:	Program Counter:	Stack:
(i)	POP Rd	0 ≤ d ≤ 31	PC ← PC + 1	SP ← SP + 1

16-bit Opcode:

1001	000d	dddd	1111

Push and pop example

- Push and Pop while looking in memory

Exercise 7 min.

1. And put on line numbers by:

Tools->Options->Text Editor->All Language: Settings: Line numbers

2. Switch the values of R16 and R17 by using the stack (use PUSH and POP)

Call instruction

- Can call a subroutine, Which is a code sequence that might be repeated often.
- When a subroutine is called, the program address is saved on the stack.
- After the subroutine is finished the program counter should go back to where it came from.

36. CALL – Long Call to a Subroutine

36.1. Description

Calls to a subroutine within the entire Program memory. The return address (to the instruction after the CALL) will be stored onto the Stack. (See also RCALL). The Stack Pointer uses a post-decrement scheme during CALL.

This instruction is not available in all devices. Refer to the device specific instruction set summary.

Operation:

(i) PC ← k Devices with 16-bit PC, 128KB Program memory maximum.

(ii) PC ← k Devices with 22-bit PC, 8MB Program memory maximum.

	Syntax:	Operands:	Program Counter:	Stack:
(i)	CALL k	0 ≤ k < 64K	PC ← k	STACK ← PC+2 SP ← SP-2, (2 bytes, 16 bits)
(ii)	CALL k	0 ≤ k < 4M	PC ← k	STACK ← PC+2 SP ← SP-3 (3 bytes, 22 bits)

32-bit Opcode:

1001	010k	kkkk	111k
kkkk	kkkk	kkkk	kkkk

RET

RET – Return from Subroutine

Description

Returns from subroutine. The return address is loaded from the STACK. The Stack Pointer uses a pre-increment scheme during RET.

Operation:

	Operation:	Comment:
(i)	PC(15:0) ← STACK	Devices with 16-bit PC, 128KB Program memory maximum.
(ii)	PC(21:0) ← STACK	Devices with 22-bit PC, 8MB Program memory maximum.

	Syntax:	Operands:	Program Counter:	Stack:
(i)	RET	None	See Operation	SP ← SP + 2, (2 bytes, 16 bits)
(ii)	RET	None	See Operation	SP ← SP + 3, (3 bytes, 22 bits)

Call and RET example

- Live code, example..

Exercise 10 min

- Try and call a subroutine using CALL and RET
- Notice the value saved on the stack.
- Notice the stack pointer while stepping through the code

- (use F11 instead of F10 to step into the subroutine)

Branch

- BRNE, Branch if Not Equal

Uses the Zero flag in the Status register

Calculate how many address it can cover.

27. BRNE – Branch if Not Equal

27.1. Description

Conditional relative branch. Tests the Zero Flag (Z) and branches relatively to PC if Z is cleared. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if, the unsigned or signed binary number represented in Rd was not equal to the unsigned or signed binary number represented in Rr. This instruction branches relatively to PC in either direction (PC - 63 ≤ destination ≤ PC + 64). Parameter k is the offset from PC and is represented in two's complement form. (Equivalent to instruction BRBC 1,k.)

Operation:

(i) If Rd ≠ Rr (Z = 0) then PC ← PC + k + 1, else PC ← PC + 1

	Syntax:	Operands:	Program Counter:
(i)	BRNE k	-64 ≤ k ≤ +6	PC ← PC + k + 1 PC ← PC + 1, if condition is false

16-bit Opcode:

1111	01kk	kkkk	k001

27.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Example:

```
 eor r27,r27 ; Clear r27
loop: inc r27 ; Increase r27
 ...
 cpi r27,5 ; Compare r27 to 5
 brne loop ; Branch if r27<>5
 nop ; Loop exit (do nothing)
```

Words	1 (2 bytes)
Cycles	1 if condition is false 2 if condition is true

Branch – coding example.

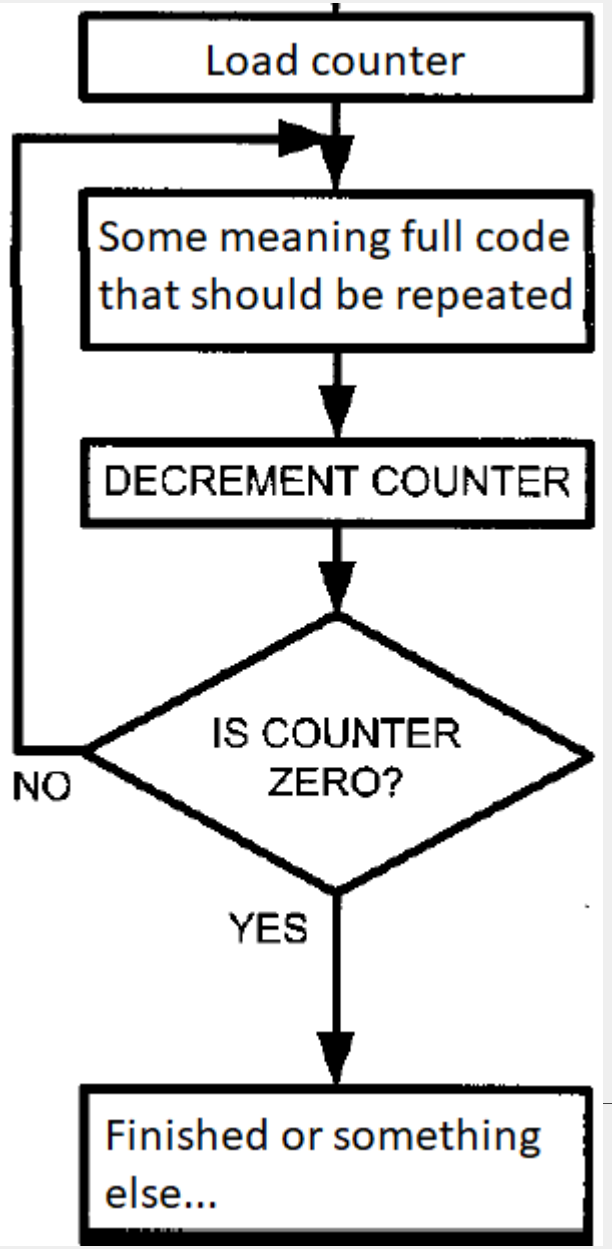
```
-   ldi r16, 3
-   dec r16
-   brne 1 ; if (r16) then jump
-   nop
-   nop
```

Use Branch to create a loo

- Ldi r18, 10

Start:

- nop; some very meaning full code, that should be repeated 10 times
- Dec r18; decrement register 18
- Brne 1 or brne Start



Exercises 1, 2 and 3 – 60 minutes.

Pipelining

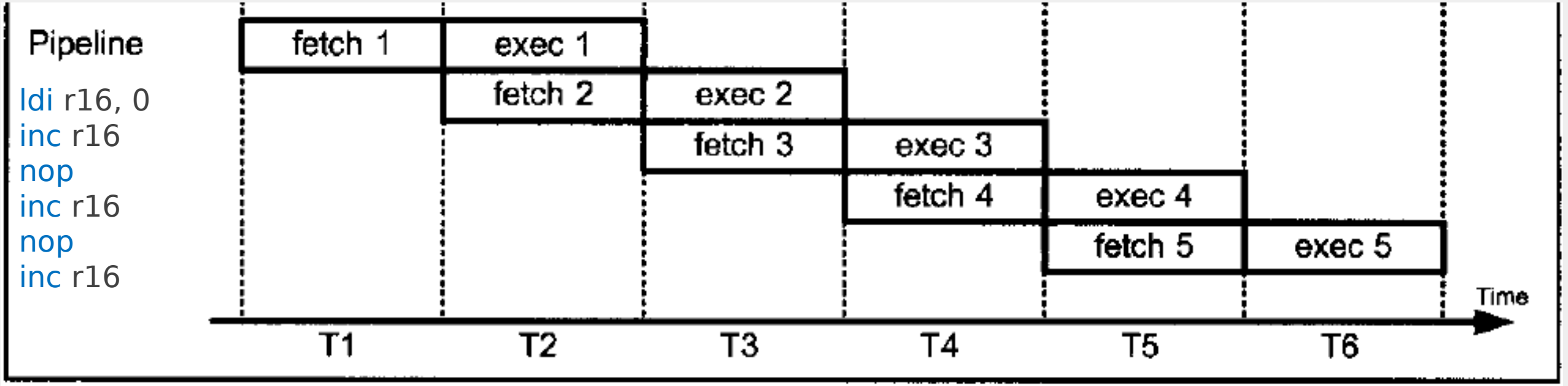


Figure 3-12. Pipeline vs. Non-pipeline

Branch Penalty

- What if a condition is met so the PC changes?
- Since we already fetched the next instruction this need to be flushed.

```
ldi r16, 1
add r16,r16
Brne here
nop
nop

here:
nop
```

- Look at cycle counter.

Instruction cycle time

Instruction	Instruction Cycles
LDI	1
DEC	1
OUT	1
ADD	1
NOP	1
JMP	3
CALL	4
BRNE	1/2

Delay

- Delay of 10ms?
- ATMEGA 2560 16MHz
- So we could use 160000 NOP
- We could make a loop
- Loop inside a loop

Example: How long time does the following code take to execute?

```
ldi r16, 2
loop:
dec r16
brne loop
```

Example: How long time does the following code take to execute?

```
ldi r16, 200
loop:
dec r16
brne loop
```

Example: loop in a loop.

```
ldi r17, 85
loop2:
ldi r16, 200
loop1:
nop
dec r16
brne loop1
dec r17
brne loop2
```

Exercise 7

- Do the rest of the exercises.

Make the LED blink with 1Hz

Lecture 8

I/O ports, arithmetic and logic instructions

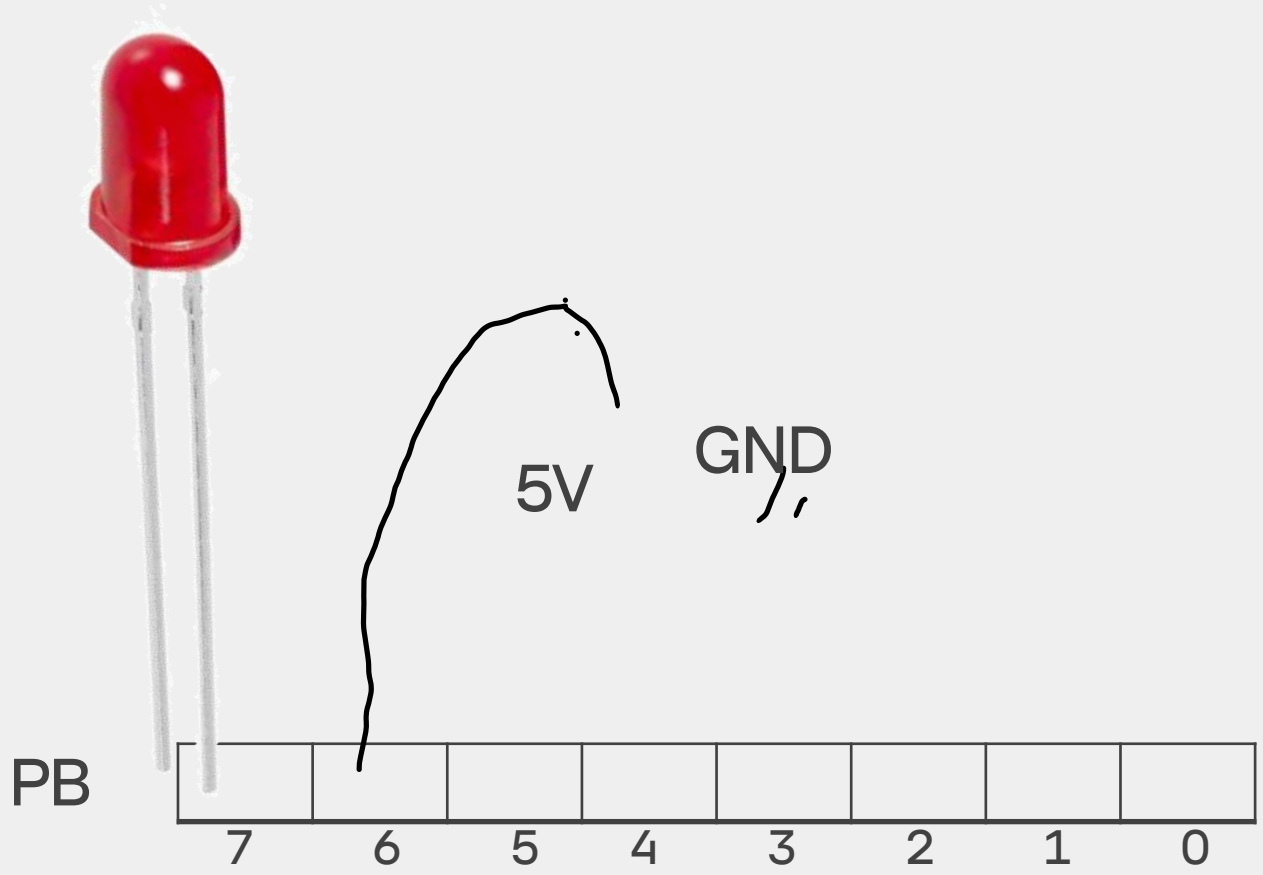
Promise: After this lecture, you can make a program, that turns on/off a LED, when pressing a button

Today you will learn

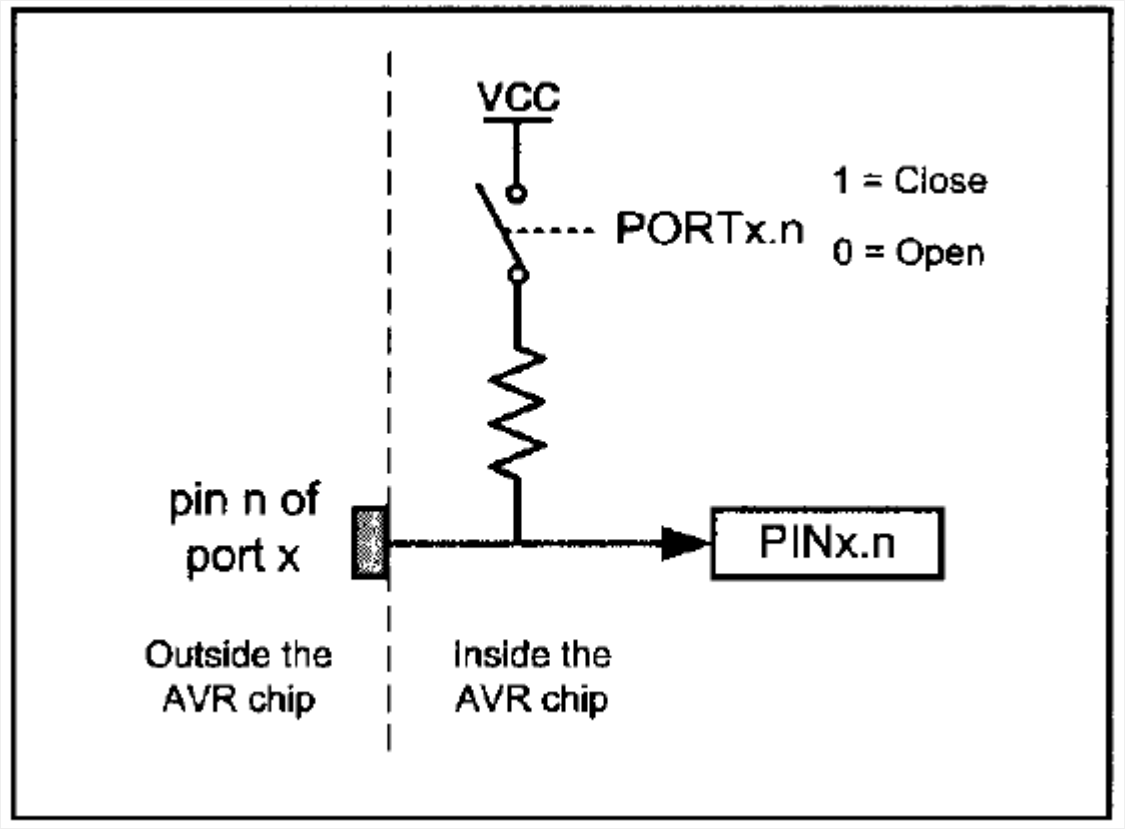
- I/O Port programming
- Arithmetic instructions
- Signed number concept and arithmetic's

INPUT, example without pullup resistor.

```
-   ldi r16, 0b10000000
-   OUT DDRB, r16
-   ever:
-   IN R16, PINB
-   ROL R16
-   OUT PORTB, R16
-   jmp ever
```



Port register while in Read-mode



Exercise 30 min.

- Play around with SBIC and SBIS in the simulator, to see it works as you expect.
- Create a program that turns the LED on and OFF dependent on whether a switch is pressed or not.

Today you will learn

- I/O Port programming
- Arithmetic instructions
- Signed number concept and arithmetic's

Calculations

- ldi r16, 11
- ldi r17, 10
- sub r17, r16
- Add r16, r17
- nop

2. compliment.

- Two's complement is the most common method of representing signed integers on computers
- Compared to other systems for representing signed numbers (*e.g.*, ones' complement), two's complement has the advantage that the fundamental arithmetic operations of addition, subtraction, and multiplication are identical to those for unsigned binary numbers

Exercise 12 minutes.

- Do a multiplication. Send the result to PORTA (Most significant byte) and PORTB (Least significant byte)

- Make a division. Use the script from the previous slide.

NEG

84. NEG – Two's Complement

84.1. Description

Replaces the contents of register Rd with its two's complement; the value $80 is left unchanged.

Operation:

(i) Rd ← $00 - Rd

Syntax:	Operands:	Program Counter:
(i) NEG Rd	0 ≤ d ≤ 31	PC ← PC + 1

16-bit Opcode:

1001	010d	dddd	0001

84.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	⇔	⇔	⇔	⇔	⇔	⇔

Hand in

- Consider starting with exercise 7 as you might need help.
- Group handins
- Name on all pages
- only PDF format.

Gør tanke til handling
VIA University College

Lecture 9

VIA Calling Convention

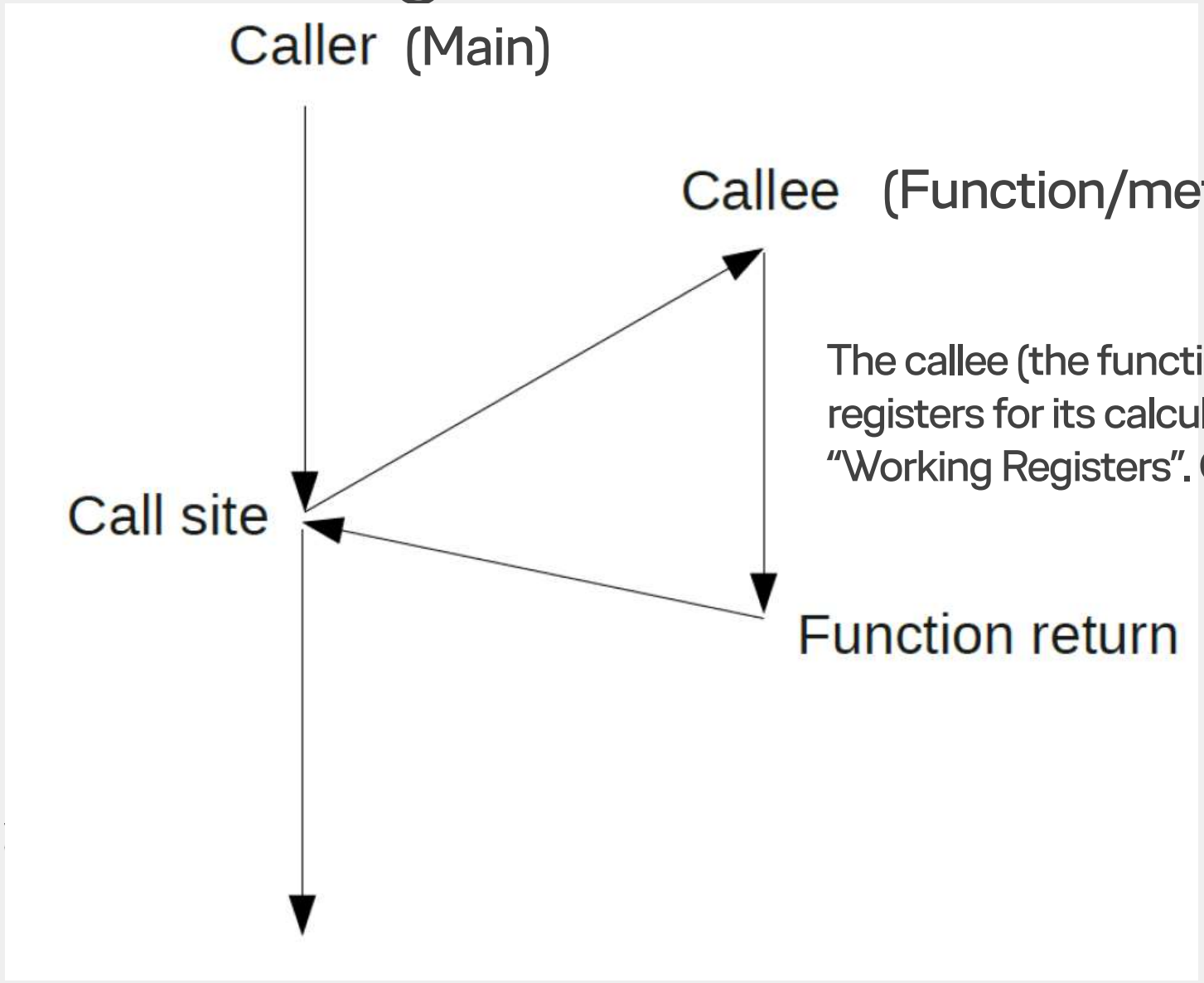
Today you will learn

- VIA Calling convention
  - Call Setup
  - Call Site
  - Saving Working registers
  - Retrieving input values
  - Implementing the Function body
  - Saving the output value
  - Restoring the working registers
  - Return from the function
  - Retrieving the output value

VIA Calling convention

- A way of calling a function (or method), with out disturbing/changing any registers.

VIA Calling convention



The callee (the function/method) needs some registers for its calculations. These are called "Working Registers". Could be R18, R19 and R21

VIA Calling convention

- The problem is, that we need some of the registers to make the calculations in the function/method

- Solution: Use The Stack! Remember the stack is big (actually all the SRAM)!

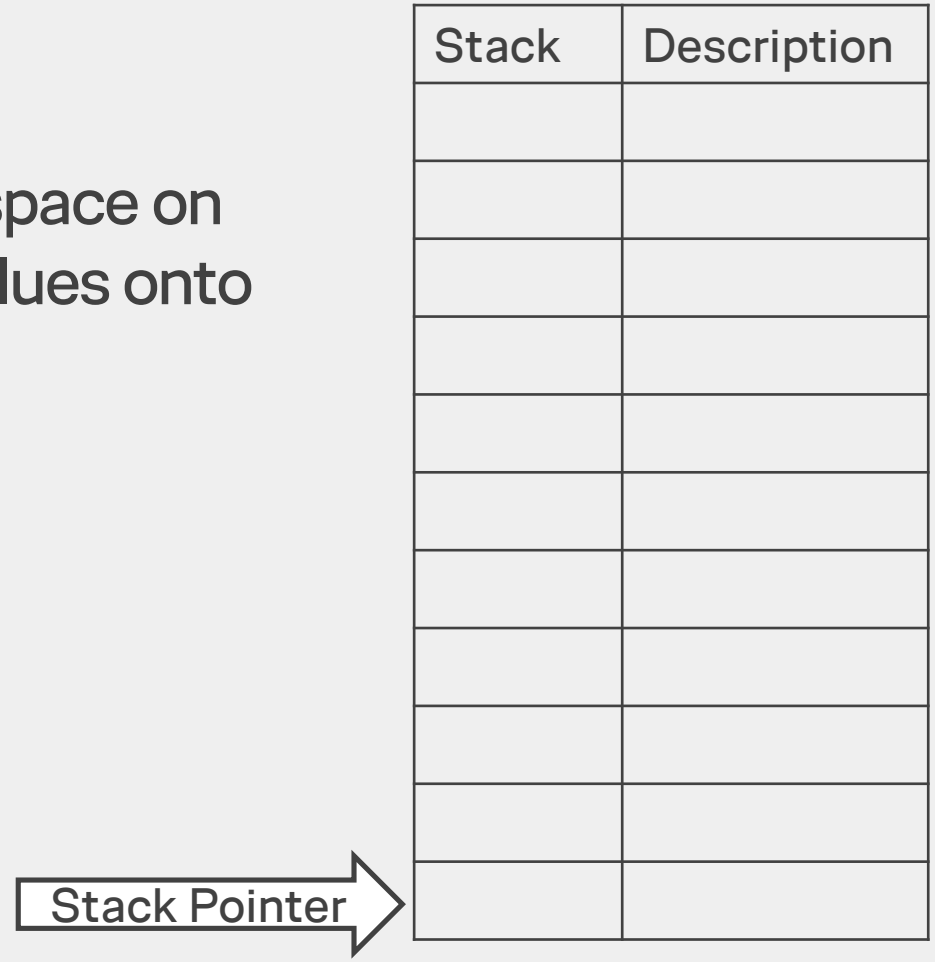
VIA Calling convention

1. **Call setup.** The caller uses the PUSH instruction to allocate space for the returnvalue and to push the input values onto the stack.
2. **Call site.** Using the CALL instruction the caller will jump to the entry point of the callee.
3. **Saving working registers.** The callee pushes all its working registers onto the stack
4. **Retrieving input values.** The callee grabs the input values and load it into the working registers
5. **Implementing the function body.** Code now follows in the callee to do the actual work.
6. **Saving output value.** The result of the function is saved on the stack
7. **Restoring working registers.** The working registers are popped from the stack so the contain their original values.
8. **Return from the function.**
9. **Retrieving output value.** Back at the caller the output value is popped from the stack

1. Call Setup

- The caller uses the PUSH instruction to allocate space on the stack for the output, and to push the input values onto the stack.

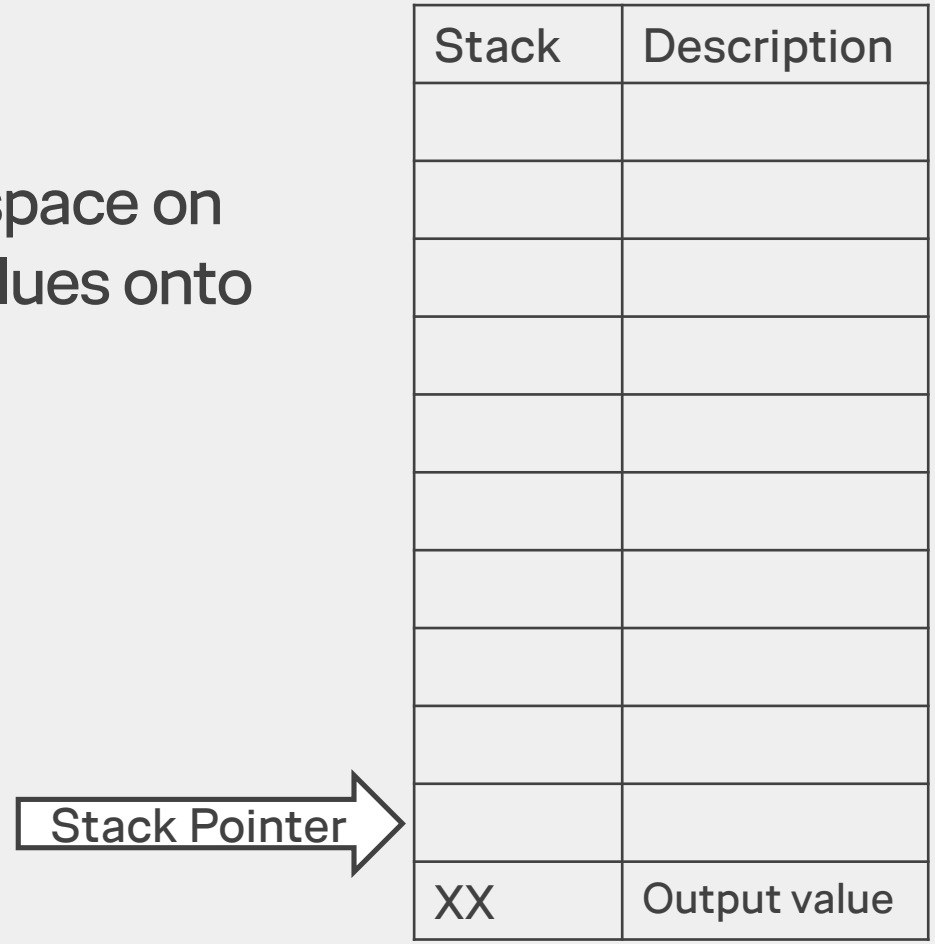
```
PUSH R16
LDI R16, 100
PUSH R16
```



1. Call Setup

- The caller uses the PUSH instruction to allocate space on the stack for the output, and to push the input values onto the stack.

```
PUSH R16
LDI R16, 100
PUSH R16
```



1. Call Setup

- The caller uses the PUSH instruction to allocate space on the stack for the output, and to push the input values onto the stack.

```
PUSH R16
LDI R16, 100
PUSH R16
```

Stack	Description
100	Input value
XX	Output value

2. Call Site

- Using the CALL instruction the caller will jump to the entry point of the callee. The CALL instruction will push the return address onto the stack (three bytes).

```
CALL delay_and_invert
```

Stack	Description
100	Input value
XX	Output value

2. Call Site

- Using the CALL instruction the caller will jump to the entry point of the callee. The CALL instruction will push the return address onto the stack (three bytes).

```
CALL delay_and_invert
```

Stack	Description
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
XX	Output value

3. Saving working registers

- As the very first thing the callee pushes all its working registers onto the stack using the PUSH instruction.

```
delay_and_invert:
PUSH R16
PUSH R17
PUSH R26
PUSH R27
PUSH R18
```

Stack	Description
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
XX	Output value

3. Saving working registers

- As the very first thing the callee pushes all its working registers onto the stack using the PUSH instruction.

```
delay_and_invert:
PUSH R16
PUSH R17
PUSH R26
PUSH R27
PUSH R18
```

Stack Pointer

Stack	Description
R18	Original values of the working registers.
R27	
R26	
R17	
R16	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
XX	Output value

4. Retrieving input values

- The callee uses the X-pointer (register R26, R27) to calculate a pointer to the input value as it is located on the stack.

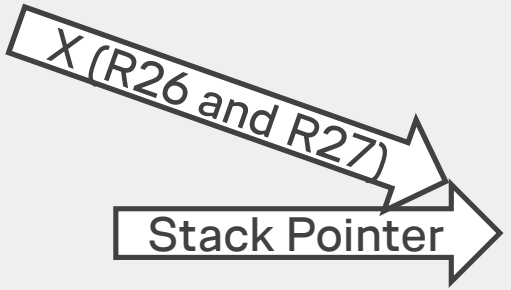
```
IN R26, SPL
IN R27, SPH
ADIW R26, 10
LD R18, -X
```

Stack Pointer

Stack	Description
R18	Original values of the working registers.
R27	
R26	
R17	
R16	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
XX	Output value

4. Retrieving input values

- The callee uses the X-pointer (register R26, R27) to calculate a pointer to the input value as it is located on the stack.



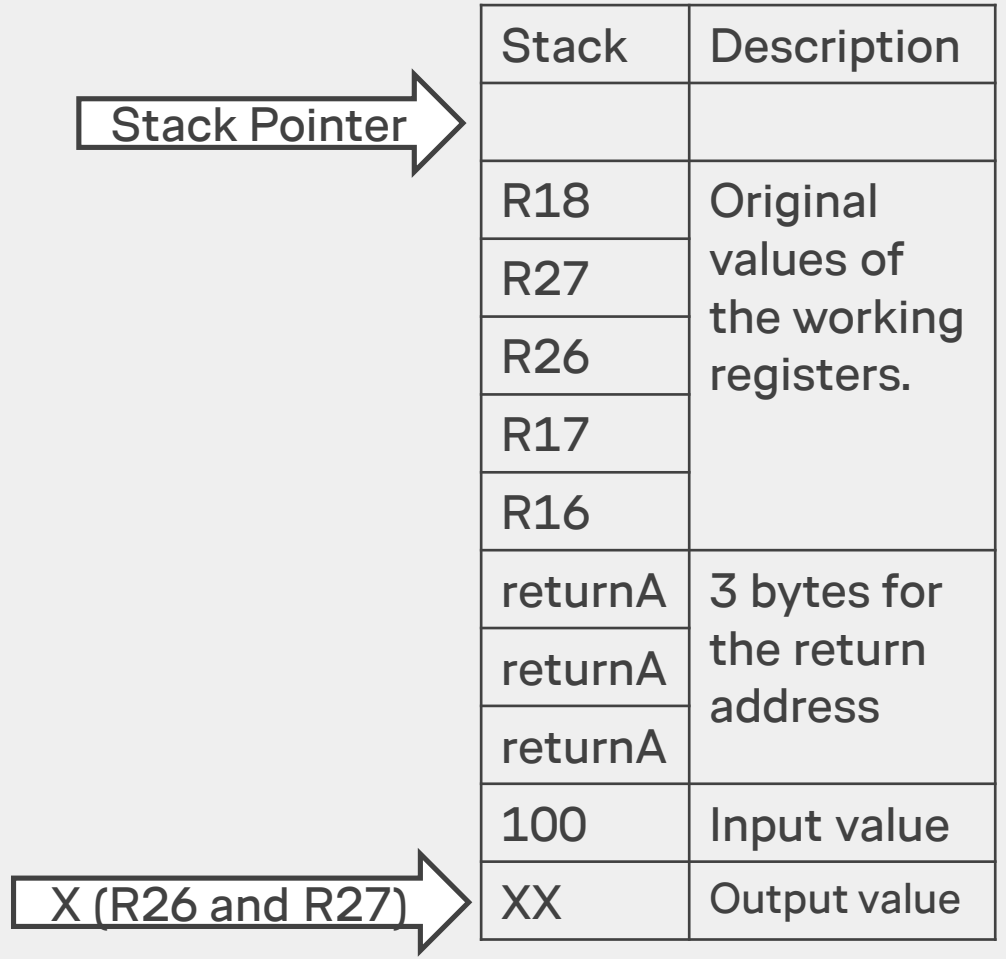
Stack	Description
R18	Original values of the working registers.
R27	
R26	
R17	
R16	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
XX	Output value

```
IN R26, SPL
IN R27, SPH
ADIW R26, 10
LD R18, -X
```

4. Retrieving input values

- The callee uses the X-pointer (register R26, R27) to calculate a pointer to the input value as it is located on the stack.
- Using the ADIW instruction, the number of working registers, the three bytes of return address and the number of input values to the function can be added to the SP to get a pointer to the first input value.

```
IN R26, SPL
IN R27, SPH
ADIW R26, 10
LD R18, -X
```

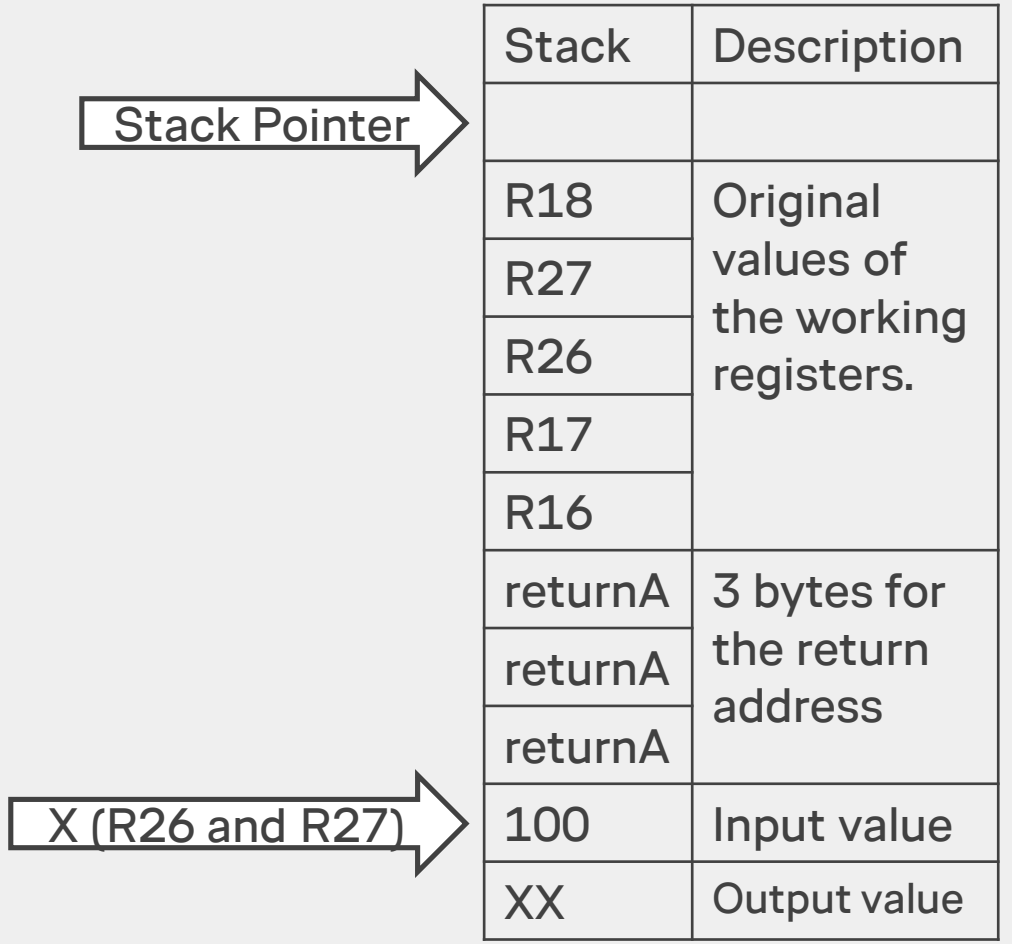


4. Retrieving input values

- The callee uses the X-pointer (register R26, R27) to calculate a pointer to the input value as it is located on the stack.
- Using the ADIW instruction, the number of working registers, the three bytes of return address and the number of input values to the function can be added to the SP to get a pointer to the first input value.
- The input value is retrieved

```
IN R26, SPL
IN R27, SPH
ADIW R26, 10
LD R18, -X
```

R18	100

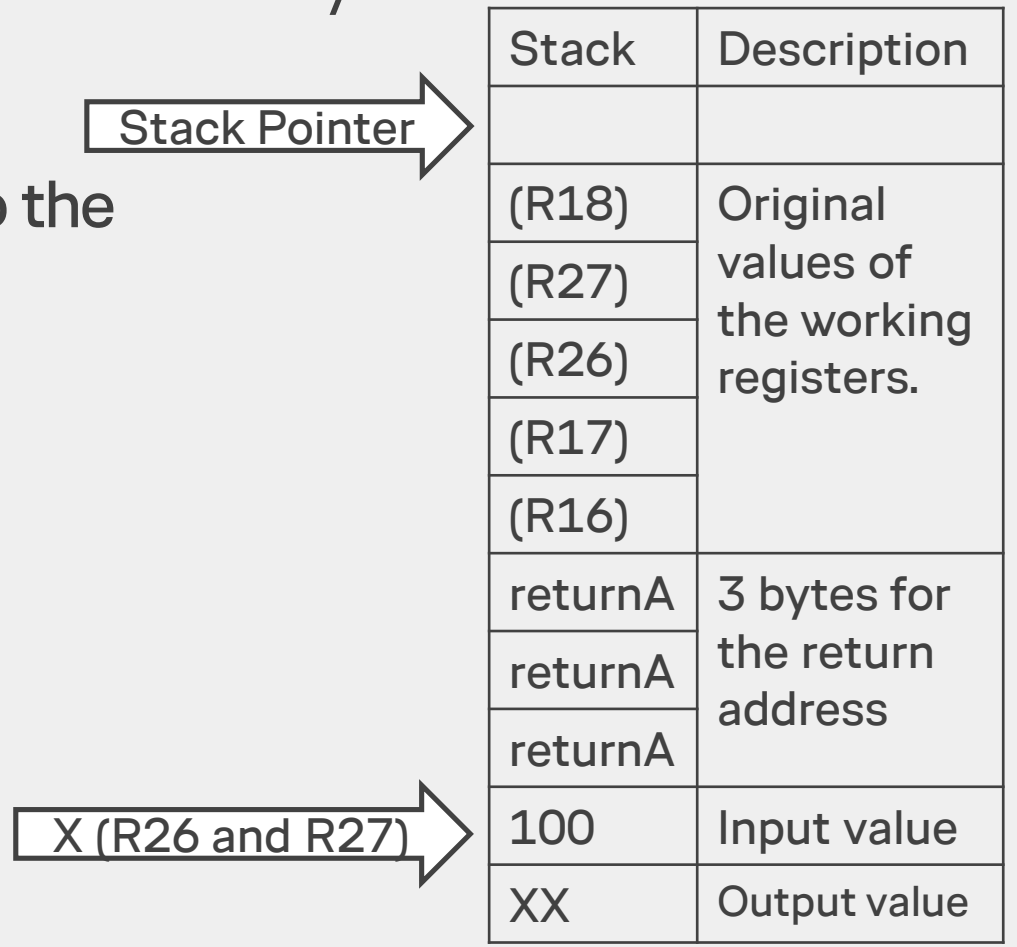


5. Implementing the function body

```
_10msdelay:
LDI R17, 160
loop2:
LDI R16, 199
loop1:
NOP
NOP
DEC R16
BRNE loop1
NOP
NOP
DEC r17
BRNE loop2

DEC r18
BRNE _10msdelay

IN r18, PORTB
COM R18
```

Code now follows in the callee to do the actual function work. In this case, a delay and a reading of portb, and inverting of somebits

R18	100

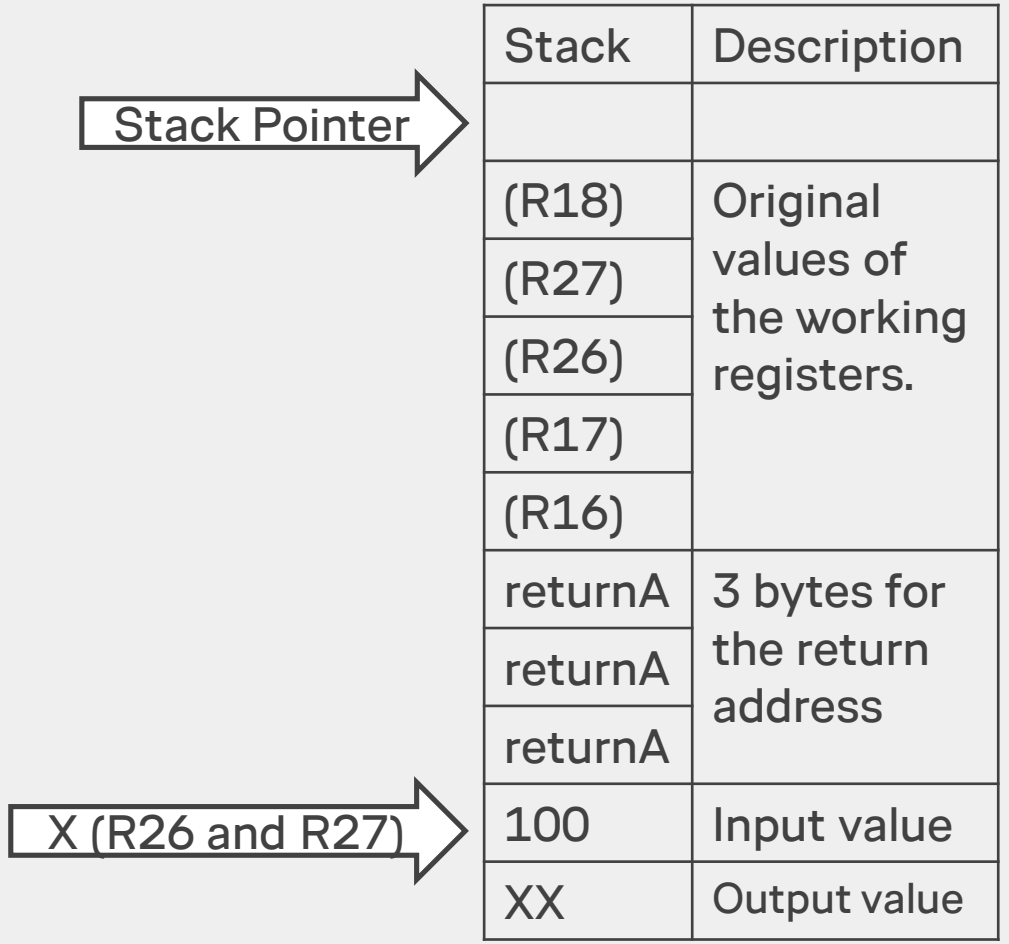


Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
XX	Output value

6. Saving the output value

- Saving output value. The result from the calculations performed in the function body is now saved on the stack at the location of the placeholder for the return value.

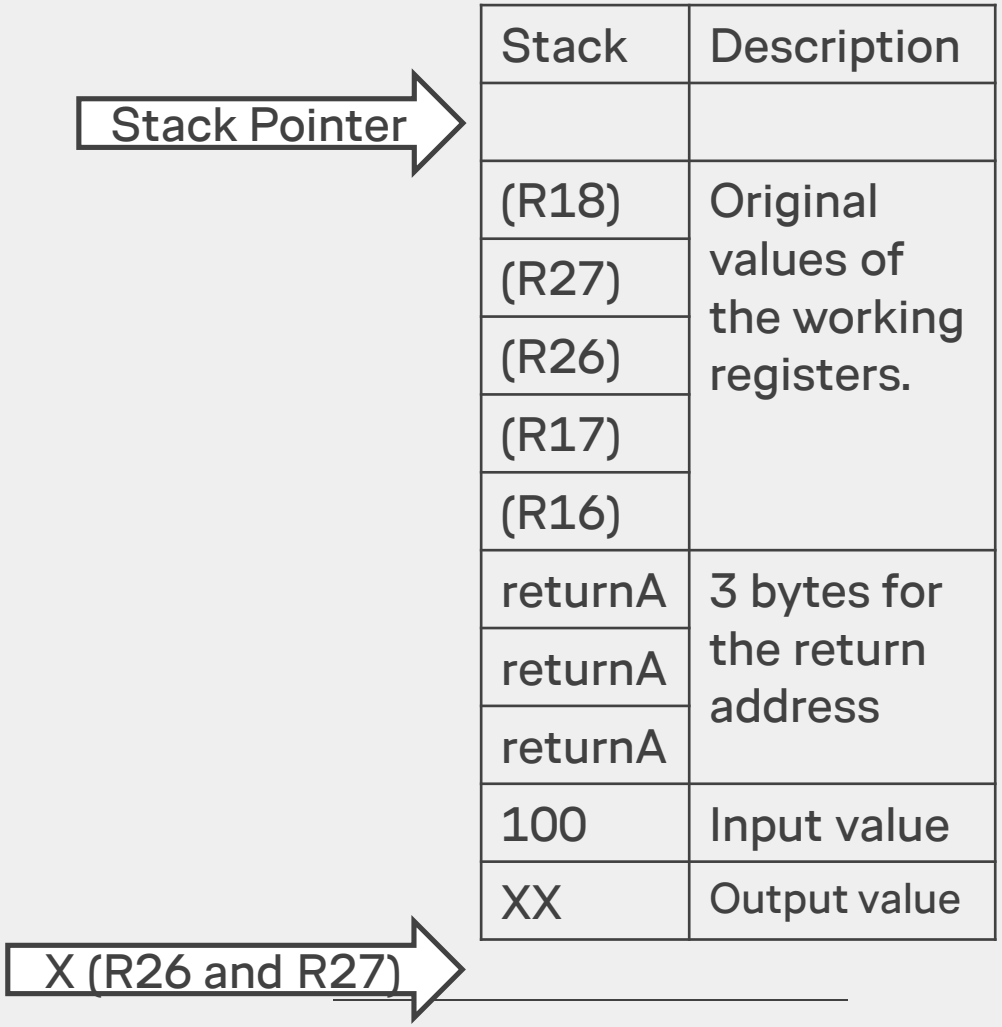
```
ADIW r26, 2
ST -X,R18
```



6. Saving the output value

- Saving output value. The result from the calculations performed in the function body is now saved on the stack at the location of the placeholder for the return value.

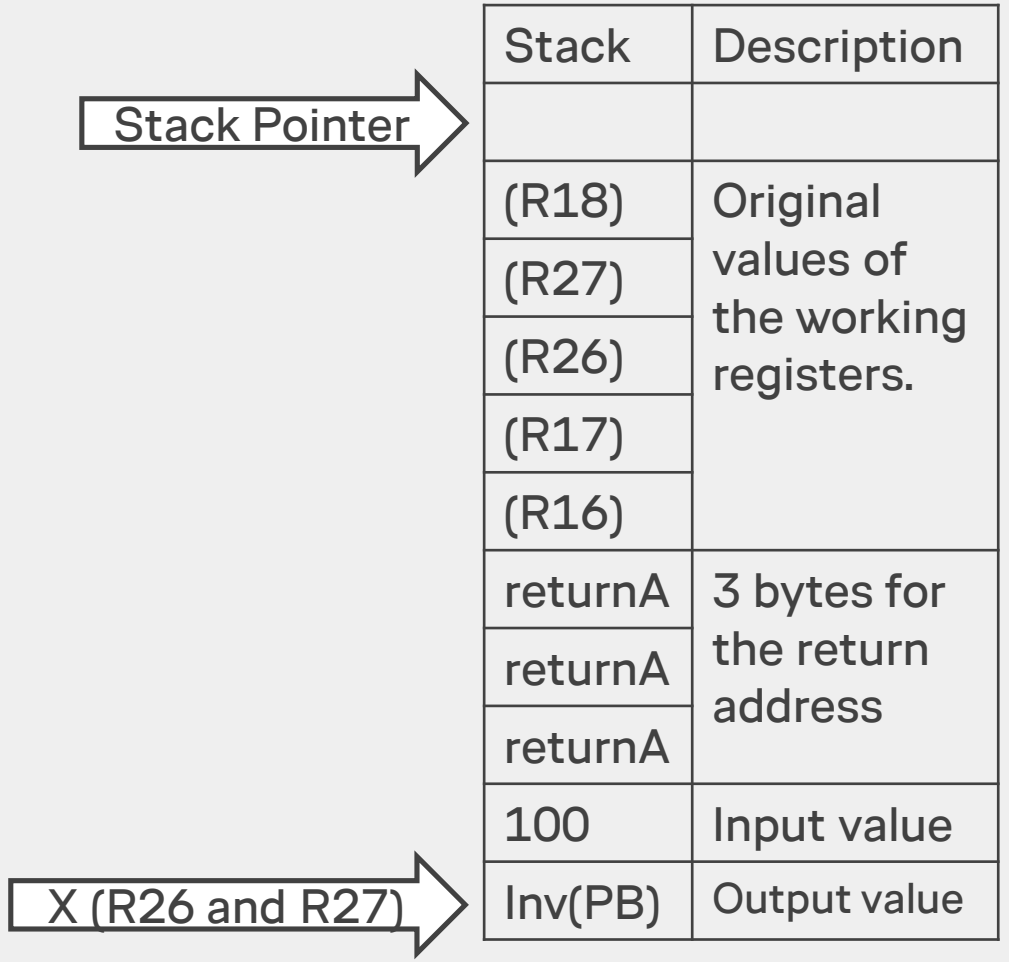
```
ADIW r26, 2
ST -X,R18
```



6. Saving the output value

- Saving output value. The result from the calculations performed in the function body is now saved on the stack at the location of the placeholder for the return value.

```
ADIW r26, 2
ST -X,R18
```



7. Restoring working registers

- Just before the return the working registers are popped from the stack using the POP instruction.

```
POP  R18
POP  R27
POP  R26
POP  R17
POP  R16
```


Stack Pointer

X (R26 and R27)

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

7. Restoring working registers

- Just before the return the working registers are popped from the stack using the POP instruction.

```
POP  R18
POP  R27
POP  R26
POP  R17
POP  R16
```

R18	Original value

Stack Pointer

X (R26 and R27)

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

7. Restoring working registers

- Just before the return the working registers are popped from the stack using the POP instruction.

```
POP  R18
POP  R27
POP  R26
POP  R17
POP  R16
```

R18	Original value
R27	Original value

Stack Pointer

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

7. Restoring working registers

- Just before the return the working registers are popped from the stack using the POP instruction.

```
POP  R18
POP  R27
POP  R26
POP  R17
POP  R16
```

R18	Original value
R27	Original value
R26	Original value

Stack Pointer

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

7. Restoring working registers

- Just before the return the working registers are popped from the stack using the POP instruction.

```
POP  R18
POP  R27
POP  R26
POP  R17
POP  R16
```

R18	Original value
R27	Original value
R26	Original value
R17	Original value

Stack Pointer

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

7. Restoring working registers

- Just before the return the working registers are popped from the stack using the POP instruction.

```
POP  R18
POP  R27
POP  R26
POP  R17
POP  R16
```

R18	Original value
R27	Original value
R26	Original value
R17	Original value
R16	Original value

Stack Pointer

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

8. Return from the function

- The RET instruction will return from the function to the caller.

```
RET
```

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

8. Return from the function

- The RET instruction will return from the function to the caller.

```
RET
```

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

9. Retrieving the output value

- Back at the caller the output value is popped from the stack using the POP instruction. The number of POP instructions to execute must be the same as the number of PUSH instructions during Call Setup.

```
POP  R16
POP  R16
```

Stack Pointer

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

9. Retrieving the output value

- The first POP instruction was the input value

```
POP  R16
POP  R16
```

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

Stack Pointer

9. Retrieving the output value

- The second POP instruction was the return value which should be saved/used

```
POP  R16
POP  R16
```

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

Stack Pointer

```
POP R16
POP R16
OUT PORTB, R16
JMP main_loop
```

Stack	Description
(R18)	Original values of the working registers.
(R27)	
(R26)	
(R17)	
(R16)	
returnA	3 bytes for the return address
returnA	
returnA	
100	Input value
Inv(PB)	Output value

Stack Pointer

Gør tanke til handling
VIA University College

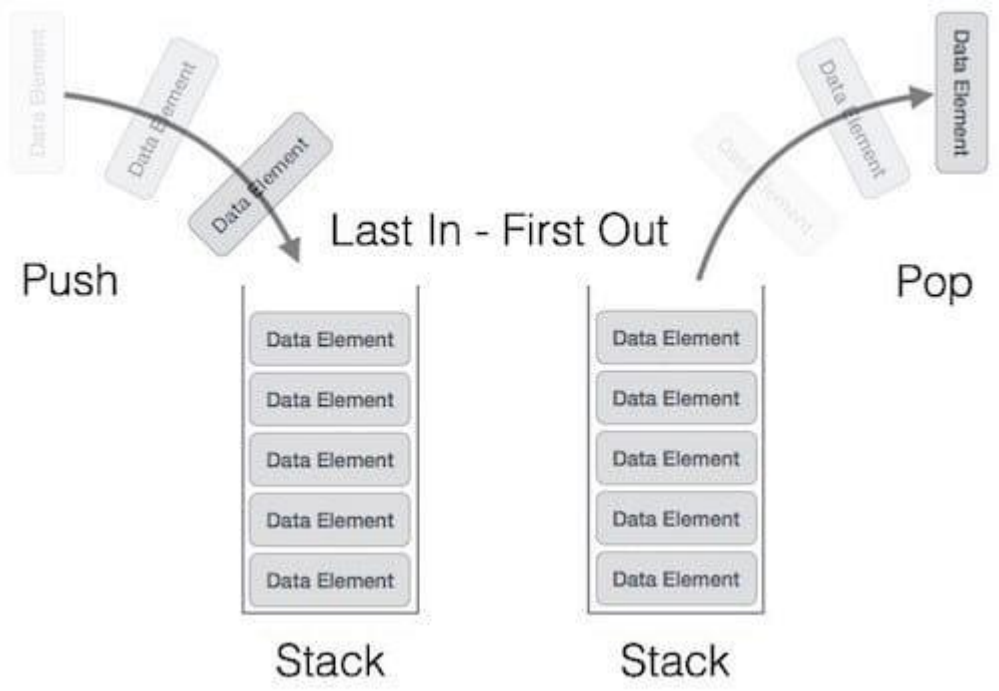
Lecture 9

VIA Calling Convention

Recall

- Stack
- Push
- Pop
- Call
- Ret
- X-pointer
- Exercise
- **VIA calling convention!!!**

Stack - LIFO

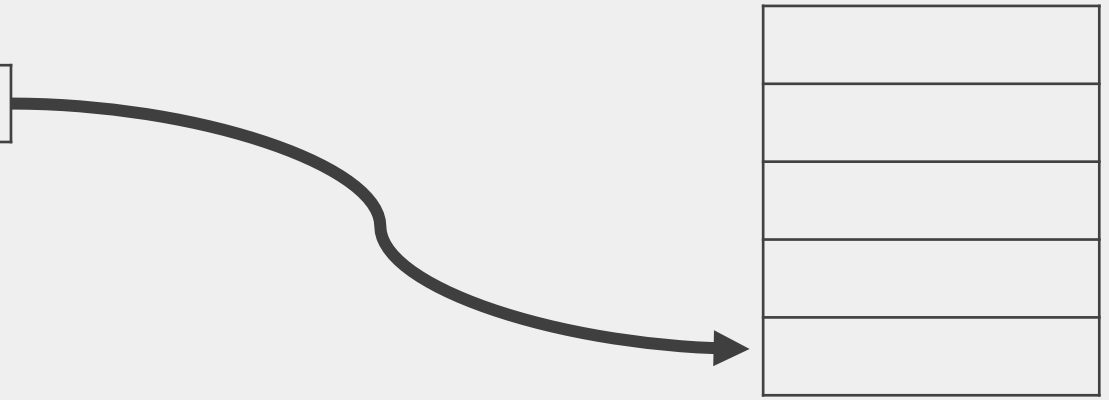

Data Element
Data Element
Data Element
Last In - First Out
Push
Pop
Data Element
Data Element
Data Element
Data Element
Data Element
Stack
Data Element
Data Element
Data Element
Data Element
Data Element
Stack

Stack graphic

```
Ldi r16, 16
Ldi r17, 17
Ldi r18, 18

Push r16
Push r17
Push r18

Pop r19
Pop r20
Push r16
```

R16	0
R17	0
R18	0
R19	0
R20	0

Stack Pointer	0x21FF



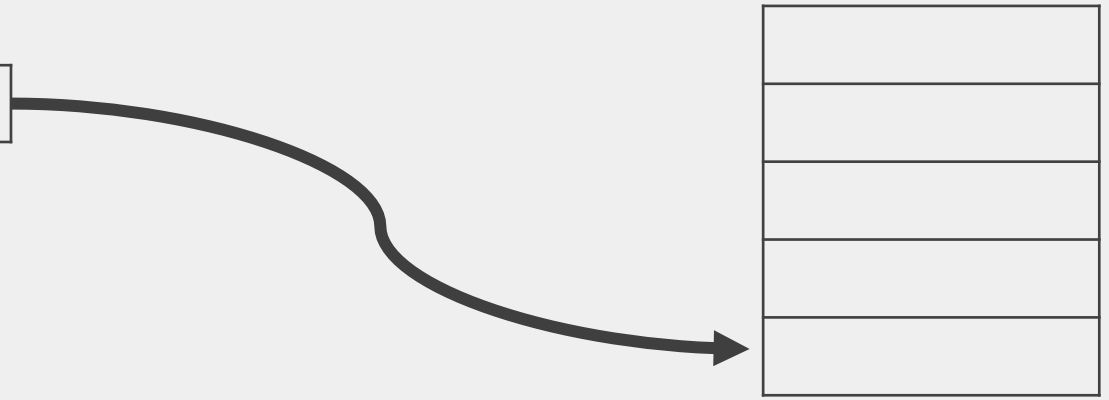
Stack graphic

IRAM

```
Ldi r16, 16
Ldi r17, 17
Ldi r18, 18

Push r16
Push r17
Push r18

Pop r19
Pop r20
Push r16
```

R16	16
R17	0
R18	0
R19	0
R20	0

Stack Pointer	0x21FF



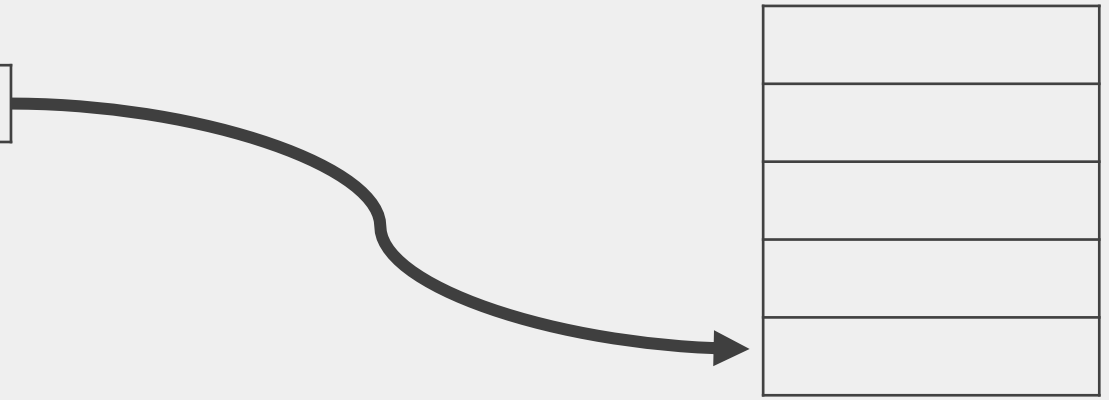
Stack graphic

IRAM

```
Ldi r16, 16
Ldi r17, 17
Ldi r18, 18

Push r16
Push r17
Push r18

Pop r19
Pop r20
Push r16
```

R16	16
R17	17
R18	0
R19	0
R20	0

Stack Pointer	0x21FF



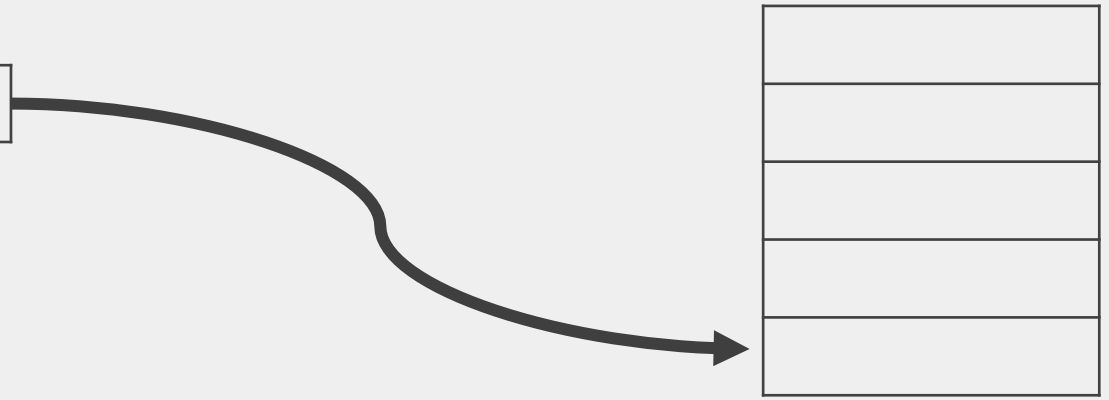
Stack graphic

IRAM

```
Ldi r16, 16
Ldi r17, 17
Ldi r18, 18

Push r16
Push r17
Push r18

Pop r19
Pop r20
Push r16
```

R16	16
R17	17
R18	18
R19	0
R20	0

Stack Pointer	0x21FF

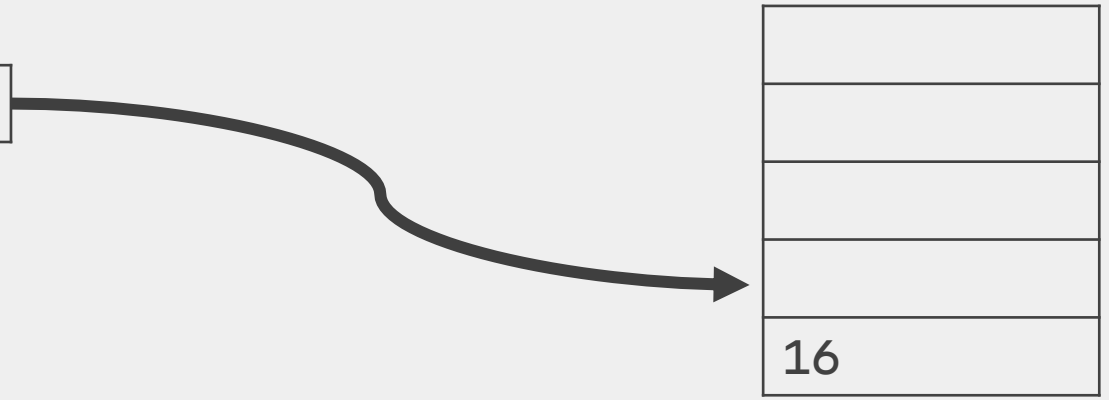


Stack graphic

```
Ldi r16, 16
Ldi r17, 17
Ldi r18, 18

Push r16
Push r17
Push r18

Pop r19
Pop r20
Push r16
```

R16	16
R17	17
R18	18
R19	0
R20	0

Stack Pointer	0x21FE



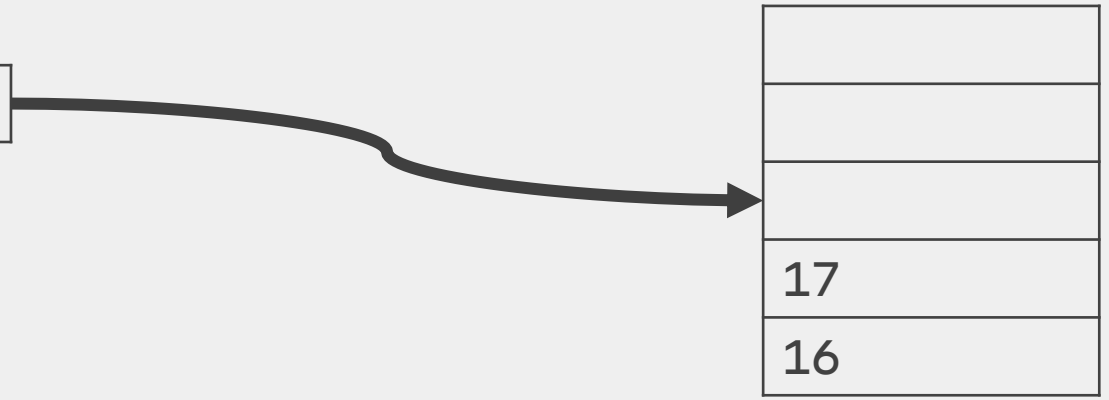
Stack graphic

```
Ldi r16, 16
Ldi r17, 17
Ldi r18, 18

Push r16
Push r17
Push r18

Pop r19
Pop r20
Push r16
```

R16	16
R17	17
R18	18
R19	0
R20	0

IRAM

Stack Pointer	0x21FD

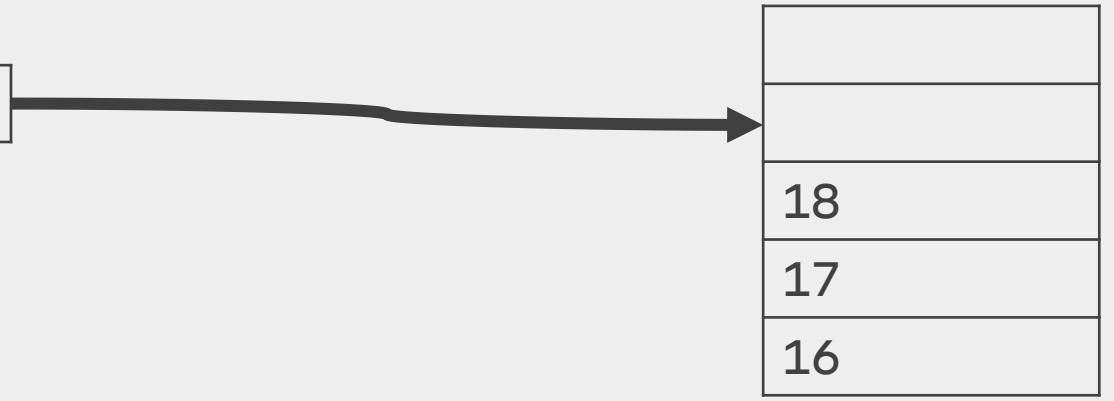


Stack graphic

```
Ldi r16, 16
Ldi r17, 17
Ldi r18, 18

Push r16
Push r17
Push r18

Pop r19
Pop r20
Push r16
```

R16	16
R17	17
R18	18
R19	0
R20	0

Stack Pointer	0x21FC

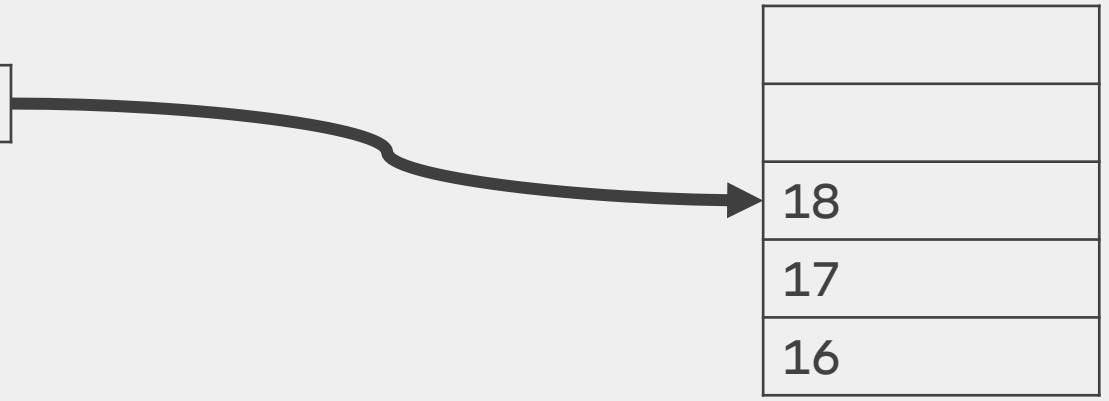


Stack graphic

```
Ldi r16, 16
Ldi r17, 17
Ldi r18, 18

Push r16
Push r17
Push r18

Pop r19
Pop r20
Push r16
```

R16	16
R17	17
R18	18
R19	18
R20	0

Stack Pointer	0x21FD

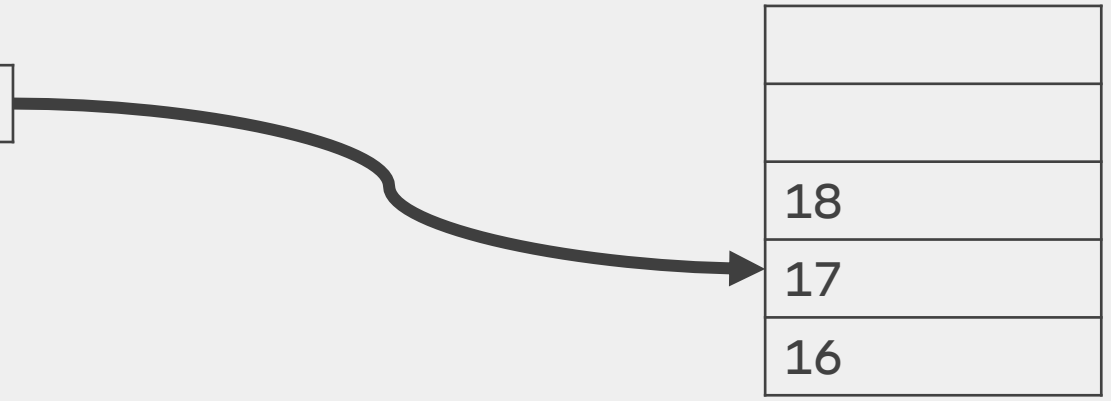


Stack graphic

```
Ldi r16, 16
Ldi r17, 17
Ldi r18, 18

Push r16
Push r17
Push r18

Pop r19
Pop r20
Push r16
```

R16	16
R17	17
R18	18
R19	18
R20	17

Stack Pointer	0x21FC

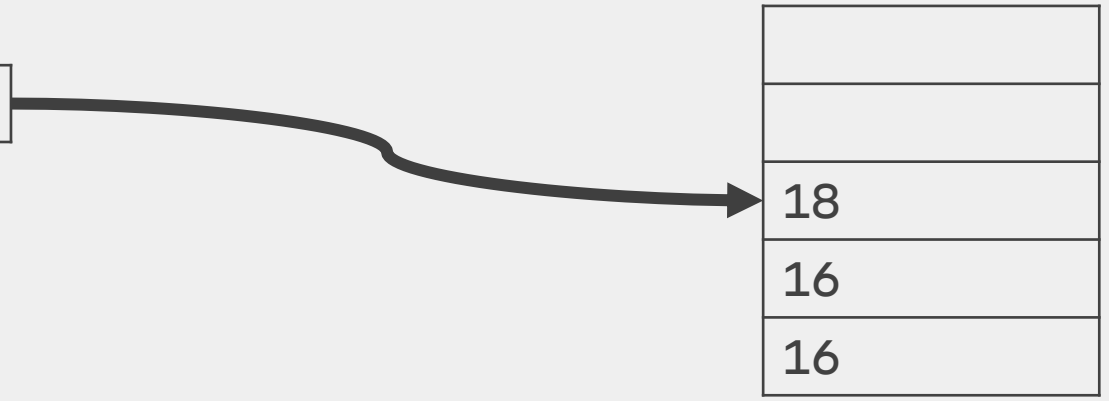


Stack graphic

```
Ldi r16, 16
Ldi r17, 17
Ldi r18, 18

Push r16
Push r17
Push r18

Pop r19
Pop r20
Push r16
```

R16	16
R17	17
R18	18
R19	18
R20	17

Stack Pointer	0x21FD



Lets do the same in the simulator.

Recall

- Stack
- Push
- Pop
- Call
- Ret
- X-pointer
- Exercise
- **VIA calling convention!!!**

Call and Ret - graphic

Call Func

Call and Ret - graphic

Call Func

Nop

R16	0
R17	0

Func:

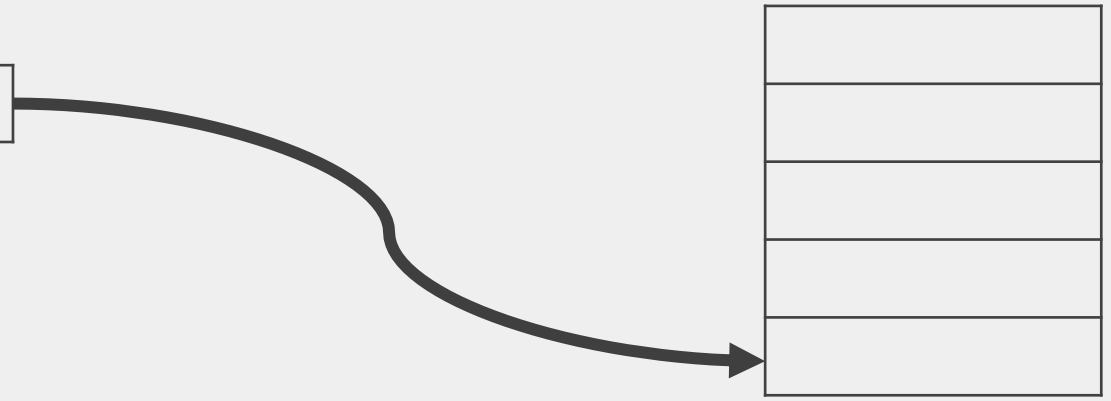
Ldi r16, 16

Push r16

Pop r17

Ret

Stack Pointer	0x21FF



Call and Ret - graphic

Call Func

Nop

R16	0
R17	0

Func:

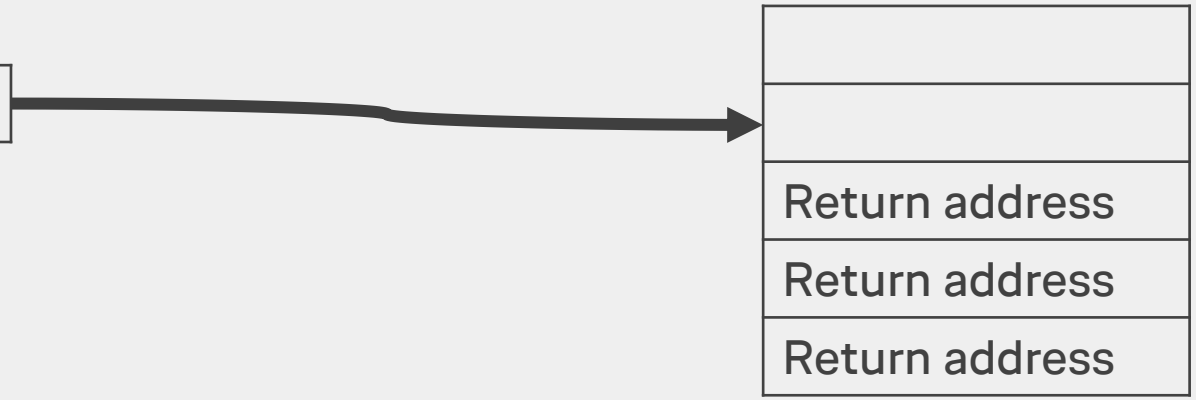
Ldi r16, 16

Push r16

Pop r17

Ret

Stack Pointer	0x21FC

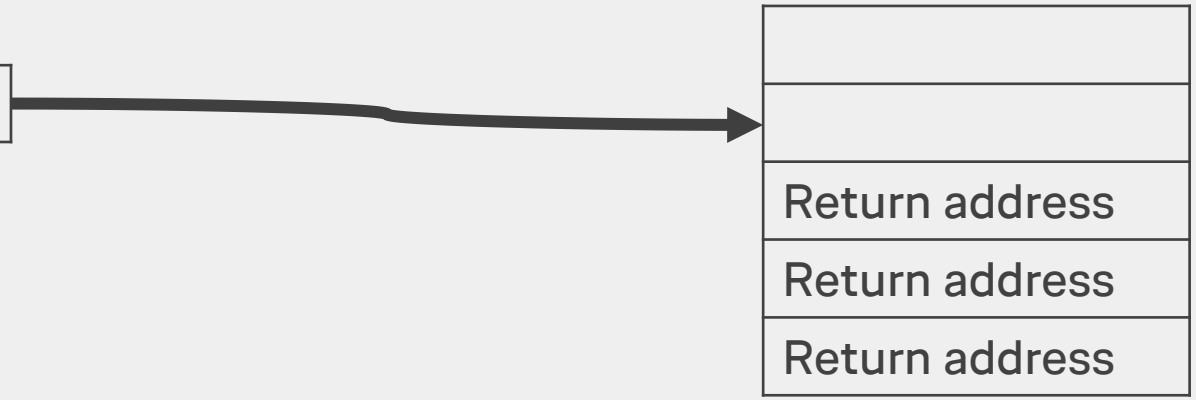


Call and Ret - graphic

```
Call Func
Nop

Func:
Ldi r16, 16
Push r16
Pop r17
Ret
```

R16	16
R17	0

Stack Pointer	0x21FC



Call and Ret - graphic

IRAM

Call Func

Nop

R16	16
R17	0

Func:

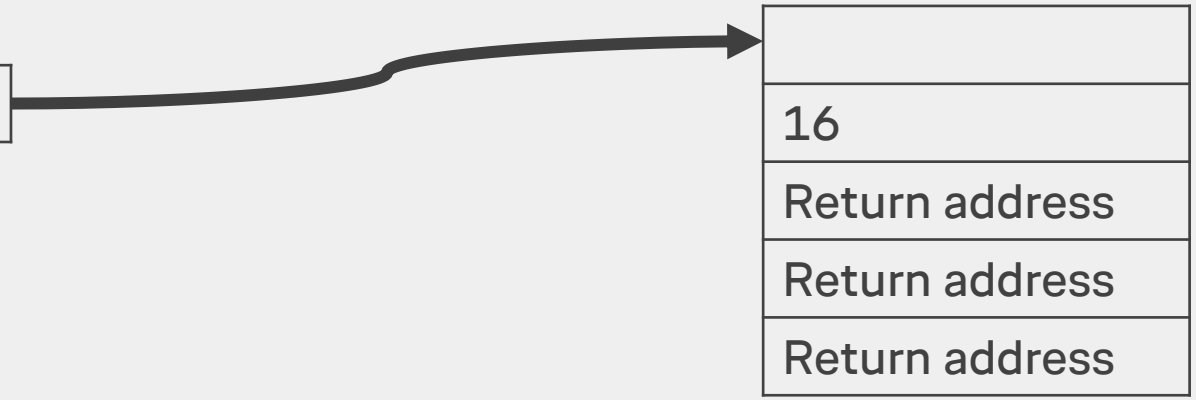
Ldi r16, 16

Push r16

Pop r17

Ret

Stack Pointer	0x21FB



Call and Ret - graphic

Call Func

Nop

R16	16
R17	16

Func:

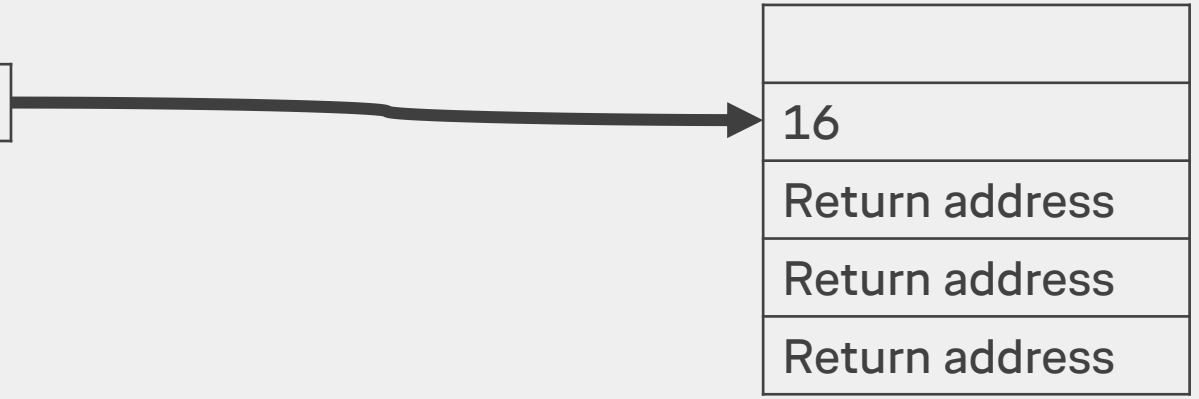
Ldi r16, 16

Push r16

Pop r17

Ret

Stack Pointer	0x21FC



Call and Ret - graphic

Call Func

Nop

R16	16
R17	16

Func:

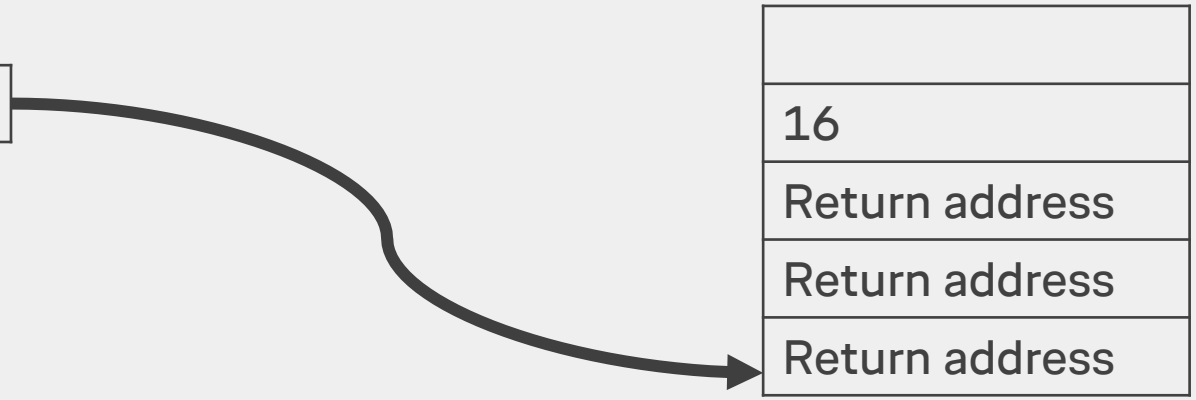
Ldi r16, 16

Push r16

Pop r17

Ret

Stack Pointer	0x21FF



Lets try it in the simulator

Recall

- Stack
- Push
- Pop
- Call
- Ret
- X-pointer
  - LD
  - ST
- Exercise
- **VIA calling convention!!!**

X-pointer

- X-pointer = R26 and R27
- Lets check in the simulator.

X-pointer

LD – Load Indirect from Data Space to Register using Index X

LD R16, X

ST – Store Indirect From Register to Data Space using Index X

ST X, R16

When a function has an input parameter: Push the function parameter on the stack before calling the function.

- **Show how this is done in the simulator.**

Exercise 20 min

- Push an input parameter on the stack and access it in the func (like was done before)
- Return a parameter from the function by using the X-pointer, and access it from the main()

Floating point - motivation

- Promise: show excel.

Floating-Point Numbers

- Floating-Point Numbers
- IEEE 754 Floating-Point Standard
- Floating-Point Addition and Subtraction

The World is Not Just Integers

The World is Not Just Integers

- Programming languages support numbers with fraction
  - Called floating-point numbers
  - Examples:

    3.14159265… (π)

    2.71828… (e)

    0.000000001 or 1.0×10^{-9} (seconds in a nanosecond)

    86,400,000,000,000 or 8.64×10^{13} (nanoseconds in a day)

    last number is a large integer that cannot fit in a 32-bit integer

The World is Not Just Integers

- Programming languages support numbers with fraction
  - Called floating-point numbers
  - Examples:

    3.14159265… (π)

    2.71828… (e)

    0.000000001 or 1.0×10^{-9} (seconds in a nanosecond)

    86,400,000,000,000 or 8.64×10^{13} (nanoseconds in a day)

    last number is a large integer that cannot fit in a 32-bit integer
- We use a scientific notation to represent
  - Very small numbers (e.g. 1.0×10^{-9})
  - Very large numbers (e.g. 8.64×10^{13})
  - Scientific notation: $\pm d.f_1f_2f_3f_4 \ldots \times 10^{\pm e_1e_2e_3}$

Floating-Point Numbers

- Examples of floating-point numbers in base 10 …
  - 5.341×10^{3} , 0.05341×10^{5} , -2.013×10^{-1} , -201.3×10^{-3}

Floating-Point Numbers

- Examples of floating-point numbers in base 10 …
  - 5.341×10^{3} , 0.05341×10^{5} , -2.013×10^{-1} , -201.3×10^{-3}

decimal point

- Examples of floating-point numbers in base 2 …
  - 1.00101×2^{23} , 0.0100101×2^{25} , -1.101101×2^{-3} , -1101.101×2^{-6}

binary point

  - Exponents are kept in decimal for clarity
  - The binary number $(1101.101)_2 = 2^3+2^2+2^0+2^{-1}+2^{-3} = 13.625$

Floating-Point Numbers

- Examples of floating-point numbers in base 10 …
  - 5.341×10^{3} , 0.05341×10^{5} , -2.013×10^{-1} , -201.3×10^{-3}

    decimal point
- Examples of floating-point numbers in base 2 …
  - 1.00101×2^{23} , 0.0100101×2^{25} , -1.101101×2^{-3} , -1101.101×2^{-6}

    binary point
  - Exponents are kept in decimal for clarity
  - The binary number $(1101.101)_2 = 2^3+2^2+2^0+2^{-1}+2^{-3} = 13.625$
- Floating-point numbers should be normalized
  - Exactly one non-zero digit should appear before the point
    - In a decimal number, this digit can be from 1 to 9
    - In a binary number, this digit should be 1
  - Normalized FP Numbers: 5.341×10^{3} and -1.101101×2^{-3}
  - NOT Normalized: 0.05341×10^{5} and -1101.101×2^{-6}

Normalization examples:

Decimal examples

- $782.3 * 10^3 = 7.823 \cdot 10^5$
- $0.000586 \cdot 10^0 = 5.86 \cdot 10^{-4}$

Binary examples:

- $1010.1 * 2^{-5} = 1.01101 \cdot 2^{-2}$
- $0.011 * 2^5 = 1.1$

Stop Here!

- Do exercise 1 and 2

Floating-Point Representation

- A floating-point number is represented by the triple
  - *S* is the Sign bit (0 is positive and 1 is negative)
    - Representation is called sign and magnitude
  - *E* is the Exponent field (signed)
    - Very large numbers have large positive exponents
    - Very small close-to-zero numbers have negative exponents
    - More bits in exponent field increases range of values
  - *F* is the Fraction field (fraction after binary point)
    - More bits in fraction field improves the precision of FP numbers

Value of a floating-point number = $(-1)^{S} \times \text{val}(F) \times 2^{\text{val}(E)}$

Next . . .

- Floating-Point Numbers
- IEEE 754 Floating-Point Standard
- Floating-Point Addition and Subtraction

IEEE 754 Floating-Point Standard

- Found in virtually every computer invented since 1980
  - Simplified porting of floating-point numbers
  - Unified the development of floating-point algorithms
  - Increased the accuracy of floating-point numbers

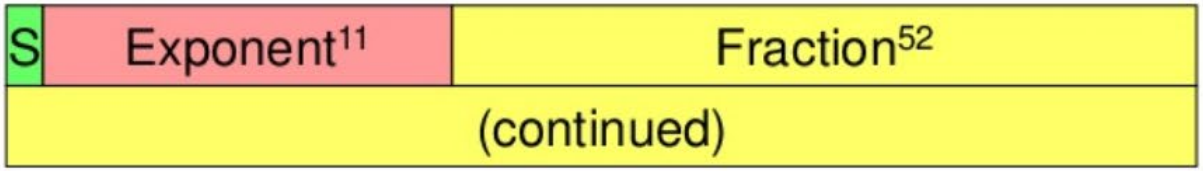
IEEE 754 Floating-Point Standard

- Found in virtually every computer invented since 1980
  - Simplified porting of floating-point numbers
  - Unified the development of floating-point algorithms
  - Increased the accuracy of floating-point numbers
- Single Precision Floating Point Numbers (32 bits)
  - 1-bit sign + 8-bit exponent + 23-bit fraction

IEEE 754 Floating-Point Standard

- Found in virtually every computer invented since 1980
  - Simplified porting of floating-point numbers
  - Unified the development of floating-point algorithms
  - Increased the accuracy of floating-point numbers
- Single Precision Floating Point Numbers (32 bits)
  - 1-bit sign + 8-bit exponent + 23-bit fraction

- Double Precision Floating Point Numbers (64 bits)
  - 1-bit sign + 11-bit exponent + 52-bit fraction



Normalized Floating Point Numbers

- For a normalized floating point number (*S*, *E*, *F*)

Normalized Floating Point Numbers

- For a normalized floating point number (S, E, F)

- Significand is equal to $(1.F)_2 = (1.f_1f_2f_3f_4\ldots)_2$
  - IEEE 754 assumes hidden 1. (not stored) for normalized numbers
  - Significand is 1 bit longer than fraction

Normalized Floating Point Numbers

- For a normalized floating point number (S, E, F)

- Significand is equal to $(1.F)_2 = (1.f_1f_2f_3f_4\ldots)_2$
  - IEEE 754 assumes hidden 1. (not stored) for normalized numbers
  - Significand is 1 bit longer than fraction
- Value of a Normalized Floating Point Number is

$$(-1)^S \times (1.F)_2 \times 2^{\text{val}(E)}$$

$$(-1)^S \times (1.f_1f_2f_3f_4\ldots)_2 \times 2^{\text{val}(E)}$$

$$(-1)^S \times (1 + f_1\times2^{-1} + f_2\times2^{-2} + f_3\times2^{-3} + f_4\times2^{-4}\ldots)_2 \times 2^{\text{val}(E)}$$

$(-1)^S$ is 1 when S is 0 (positive), and –1 when S is 1 (negative)

Biased Exponent Representation

- How to represent a signed exponent? Choices are …
  - Sign + magnitude representation for the exponent
  - Two's complement representation
  - Biased representation

Biased Exponent Representation

- How to represent a signed exponent? Choices are …
  - Sign + magnitude representation for the exponent
  - Two's complement representation
  - Biased representation
- IEEE 754 uses biased representation for the exponent
  - Value of exponent = val(E) = E – Bias (Bias is a constant)

Biased Exponent Representation

- How to represent a signed exponent? Choices are …
  - Sign + magnitude representation for the exponent
  - Two's complement representation
  - Biased representation
- IEEE 754 uses biased representation for the exponent
  - Value of exponent = val(E) = E – Bias (Bias is a constant)
- Recall that exponent field is 8 bits for single precision
  - E can be in the range 0 to 255
  - E = 0 and E = 255 are reserved for special use (discussed later)
  - E = 1 to 254 are used for normalized floating point numbers
  - Bias = 127 (half of 254), val(E) = E – 127
  - val(E=1) = –126, val(E=127) = 0, val(E=254) = 127

Biased Exponent - Cont'd

- For double precision, exponent field is 11 bits
  - E can be in the range 0 to 2047
  - $E = 0$ and $E = 2047$ are reserved for special use
  - $E = 1$ to 2046 are used for normalized floating point numbers
  - Bias = 1023 (half of 2046), val(E) = $E - 1023$
  - val(E=1) = –1022, val(E=1023) = 0, val(E=2046) = 1023

Biased Exponent - Cont'd

- For double precision, exponent field is 11 bits
  - E can be in the range 0 to 2047
  - $E = 0$ and $E = 2047$ are reserved for special use
  - $E = 1$ to 2046 are used for normalized floating point numbers
  - Bias = 1023 (half of 2046), val(E) = $E - 1023$
  - val(E=1) = –1022, val(E=1023) = 0, val(E=2046) = 1023
- Value of a Normalized Floating Point Number is

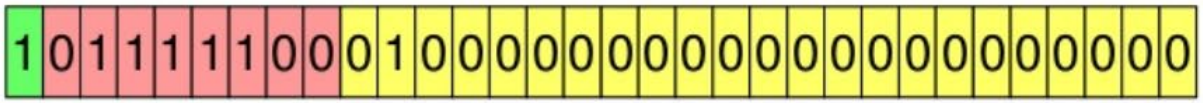
$$(-1)^S \times (1.F)_2 \times 2^{E - \text{Bias}}$$

$$(-1)^S \times (1.f_1 f_2 f_3 f_4 \ldots)_2 \times 2^{E - \text{Bias}}$$

$$(-1)^S \times (1 + f_1 \times 2^{-1} + f_2 \times 2^{-2} + f_3 \times 2^{-3} + f_4 \times 2^{-4} \ldots)_2 \times 2^{E - \text{Bias}}$$

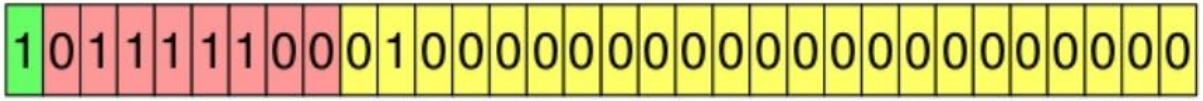
Examples of Single Precision Float

- What is the decimal value of this Single Precision float?



Examples of Single Precision Float

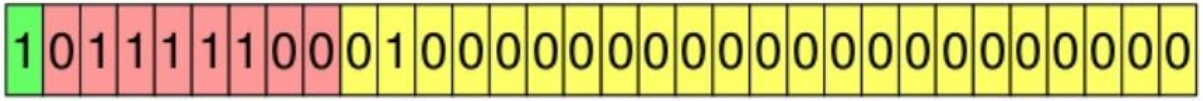
- What is the decimal value of this Single Precision float?



- Solution:
  - Sign = 1 is negative

Examples of Single Precision Float

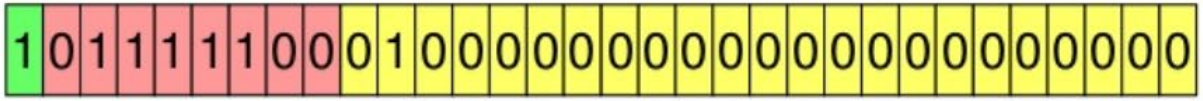
- What is the decimal value of this Single Precision float?



- Solution:
  - Sign = 1 is negative
  - Exponent = $(01111100)_2 = 124$, $E - \text{bias} = 124 - 127 = -3$

Examples of Single Precision Float

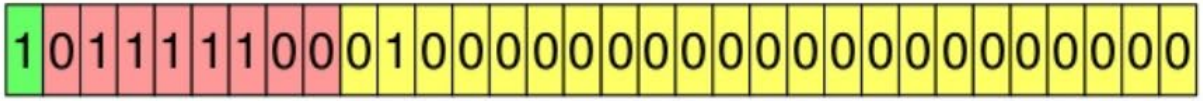
- What is the decimal value of this Single Precision float?



- Solution:
  - Sign = 1 is negative
  - Exponent = $(01111100)_2 = 124$, $E - \text{bias} = 124 - 127 = -3$
  - Significand = $(1.0100 \ldots 0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit)

Examples of Single Precision Float

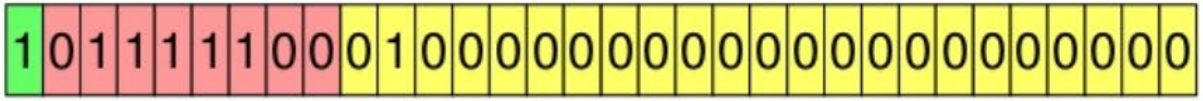
- What is the decimal value of this Single Precision float?



- Solution:
  - Sign = 1 is negative
  - Exponent = $(01111100)_2 = 124$, $E - \text{bias} = 124 - 127 = -3$
  - Significand = $(1.0100 \ldots 0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit)
  - Value in decimal = $-1.25 \times 2^{-3} = -0.15625$

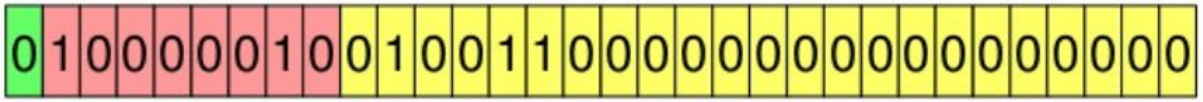
Examples of Single Precision Float

- What is the decimal value of this Single Precision float?



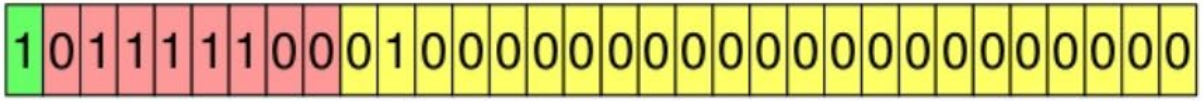
- Solution:
  - Sign = 1 is negative
  - Exponent = $(01111100)_2 = 124$, $E - \text{bias} = 124 - 127 = -3$
  - Significand = $(1.0100 \ldots 0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit)
  - Value in decimal = $-1.25 \times 2^{-3} = -0.15625$

- What is the decimal value of?



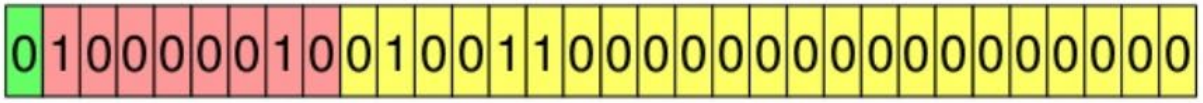
Examples of Single Precision Float

- What is the decimal value of this Single Precision float?



- Solution:
  - Sign = 1 is negative
  - Exponent = $(01111100)_2 = 124$, $E - \text{bias} = 124 - 127 = -3$
  - Significand = $(1.0100 \ldots 0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit)
  - Value in decimal = $-1.25 \times 2^{-3} = -0.15625$

- What is the decimal value of?



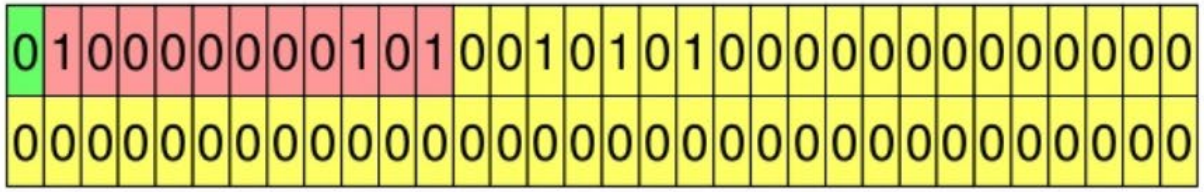
- Solution:

implicit

  - Value in decimal = $+(1.01001100 \ldots 0)_2 \times 2^{130-127} =$ $(1.01001100 \ldots 0)_2 \times 2^3 = (1010.01100 \ldots 0)_2 = 10.375$

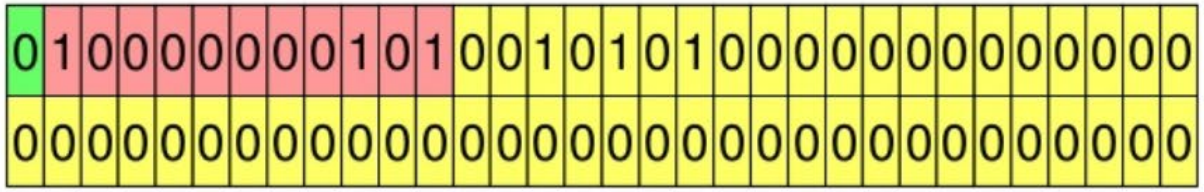
Examples of Double Precision Float

- What is the decimal value of this Double Precision float ?



Examples of Double Precision Float

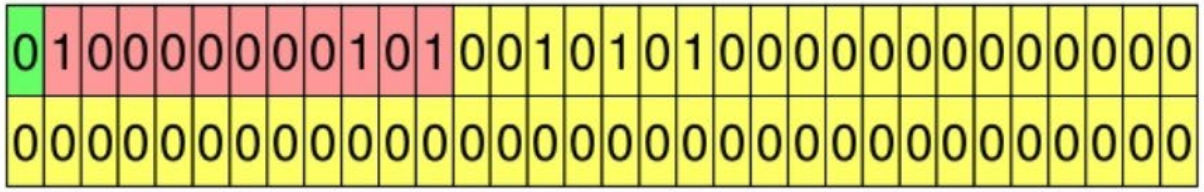
- What is the decimal value of this Double Precision float ?



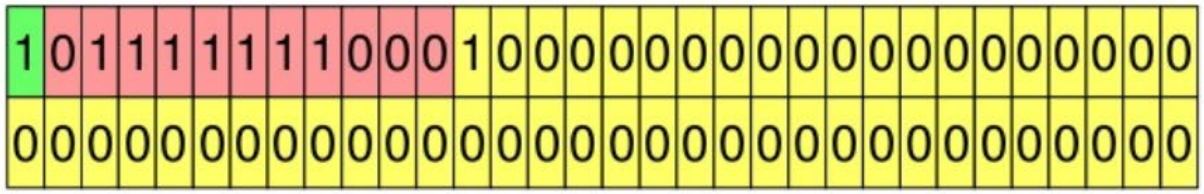
- Solution:
  - Value of exponent = $(10000000101)_2$ – Bias = 1029 – 1023 = 6
  - Value of double float = $(1.00101010 \ldots 0)_2 \times 2^6$ (1. is implicit) = $(1001010.10 \ldots 0)_2$ = 74.5

Examples of Double Precision Float

- What is the decimal value of this Double Precision float ?



- Solution:
  - Value of exponent = $(10000000101)_2$ – Bias = 1029 – 1023 = 6
  - Value of double float = $(1.00101010 \ldots 0)_2 \times 2^6$ (1. is implicit) = $(1001010.10 \ldots 0)_2$ = 74.5
- What is the decimal value of ?



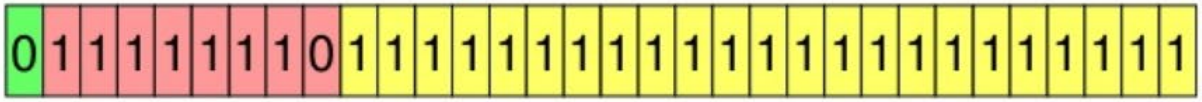
- Do it yourself! (answer should be $-1.5 \times 2^{-7} = -0.01171875$)

Largest Normalized Float

- What is the Largest normalized float?

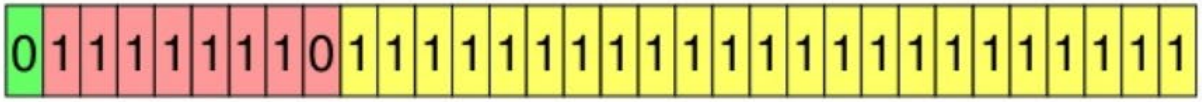
Largest Normalized Float

- What is the Largest normalized float?
- Solution for Single Precision:



Largest Normalized Float

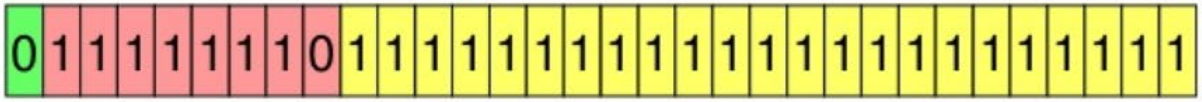
- What is the Largest normalized float?
- Solution for Single Precision:



  - Exponent – bias = 254 – 127 = 127 (largest exponent for SP)
  - Significand = $(1.111 \ldots 1)_2$ = almost 2
  - Value in decimal $\approx 2 \times 2^{127} \approx 2^{128} \approx 3.4028 \ldots \times 10^{38}$

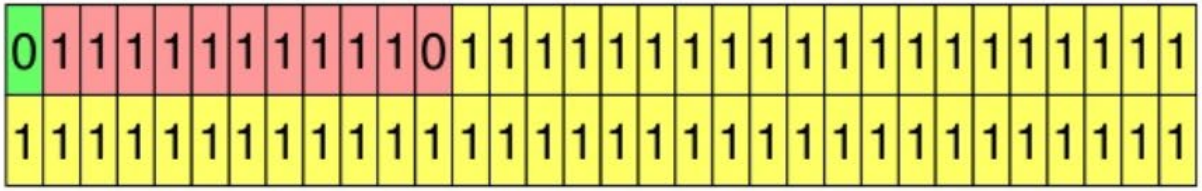
Largest Normalized Float

- What is the Largest normalized float?
- Solution for Single Precision:



  - Exponent – bias = 254 – 127 = 127 (largest exponent for SP)
  - Significand = $(1.111 \ldots 1)_2$ = almost 2
  - Value in decimal $\approx 2 \times 2^{127} \approx 2^{128} \approx 3.4028 \ldots \times 10^{38}$

- Solution for Double Precision:



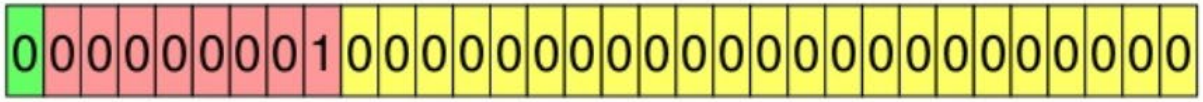
  - Value in decimal $\approx 2 \times 2^{1023} \approx 2^{1024} \approx 1.79769 \ldots \times 10^{308}$

Smallest Normalized Float

- What is the smallest (in absolute value) normalized float?

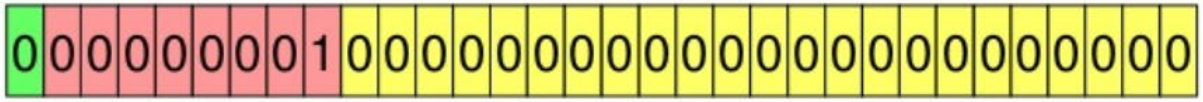
Smallest Normalized Float

- What is the smallest (in absolute value) normalized float?
- Solution for Single Precision:



Smallest Normalized Float

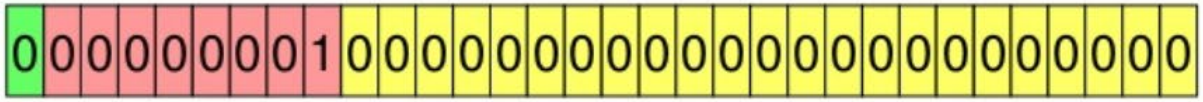
- What is the smallest (in absolute value) normalized float?
- Solution for Single Precision:



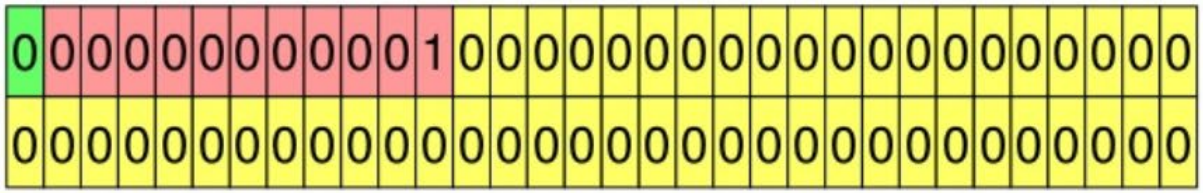
  - Exponent – bias = 1 – 127 = –126 (smallest exponent for SP)
  - Significand = $(1.000 \ldots 0)_2 = 1$
  - Value in decimal = $1 \times 2^{-126} = 1.17549 \ldots \times 10^{-38}$

Smallest Normalized Float

- What is the smallest (in absolute value) normalized float?
- Solution for Single Precision:



  - Exponent – bias = 1 – 127 = –126 (smallest exponent for SP)
  - Significand = $(1.000 \ldots 0)_2 = 1$
  - Value in decimal = $1 \times 2^{-126} = 1.17549 \ldots \times 10^{-38}$
- Solution for Double Precision:



  - Value in decimal = $1 \times 2^{-1022} = 2.22507 \ldots \times 10^{-308}$

Zero, Infinity, and NaN

- Zero
  - Exponent field $E = 0$ and fraction $F = 0$
  - +0 and –0 are possible according to sign bit S

Zero, Infinity, and NaN

- Zero
  - Exponent field $E = 0$ and fraction $F = 0$
  - +0 and –0 are possible according to sign bit S
- Infinity
  - Infinity is a special value represented with maximum E and $F = 0$
    - For single precision with 8-bit exponent: maximum $E = 255$
    - For double precision with 11-bit exponent: maximum $E = 2047$
  - Infinity can result from overflow or division by zero
  - $+\infty$ and $-\infty$ are possible according to sign bit S

Zero, Infinity, and NaN

- Zero
  - Exponent field $E = 0$ and fraction $F = 0$
  - +0 and –0 are possible according to sign bit S
- Infinity
  - Infinity is a special value represented with maximum E and $F = 0$
    - For single precision with 8-bit exponent: maximum $E = 255$
    - For double precision with 11-bit exponent: maximum $E = 2047$
  - Infinity can result from overflow or division by zero
  - $+\infty$ and $-\infty$ are possible according to sign bit S
- NaN (Not a Number)
  - NaN is a special value represented with maximum E and $F \neq 0$
  - Result from exceptional situations, such as 0/0 or sqrt(negative)
  - Operation on a NaN results is NaN: Op(X, NaN) = NaN

Denormalized Numbers

- IEEE standard uses denormalized numbers to …
  - Fill the gap between 0 and the smallest normalized float
  - Provide gradual underflow to zero

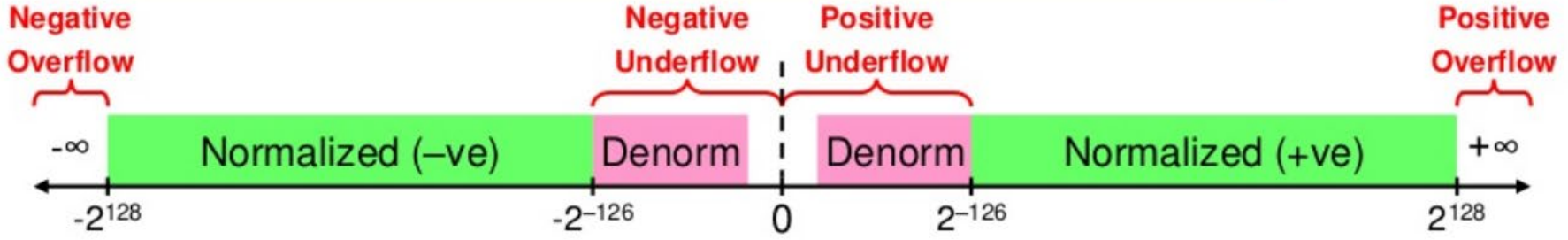
Denormalized Numbers

- IEEE standard uses denormalized numbers to …
  - Fill the gap between 0 and the smallest normalized float
  - Provide gradual underflow to zero
- Denormalized: exponent field E is 0 and fraction $F \neq 0$
  - Implicit 1. before the fraction now becomes 0. (not normalized)

Denormalized Numbers

- IEEE standard uses denormalized numbers to …
  - Fill the gap between 0 and the smallest normalized float
  - Provide gradual underflow to zero
- Denormalized: exponent field E is 0 and fraction $F \neq 0$
  - Implicit 1. before the fraction now becomes 0. (not normalized)
- Value of denormalized number (S, 0, F)

Single precision:	$(-1)^S \times (0.F)_2 \times 2^{-126}$
Double precision:	$(-1)^S \times (0.F)_2 \times 2^{-1022}$



Denormalized Numbers

Summary of IEEE 754 Encoding

Single-Precision	Exponent = 8	Fraction = 23	Value
Normalized Number	1 to 254	Anything	$\pm (1.F)_2 \times 2^{E-127}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-126}$
Zero	0	0	± 0
Infinity	255	0	$\pm \infty$
NaN	255	nonzero	NaN

Double-Precision	Exponent = 11	Fraction = 52	Value
Normalized Number	1 to 2046	Anything	$\pm (1.F)_2 \times 2^{E-1023}$
Denormalized Number	0	nonzero	$\pm (0.F)_2 \times 2^{-1022}$
Zero	0	0	± 0
Infinity	2047	0	$\pm \infty$
NaN	2047	nonzero	NaN

Stop here!

- Do exercise 3.1 to 3.8

Next . . .

- Floating-Point Numbers
- IEEE 754 Floating-Point Standard
- Floating-Point Addition and Subtraction

Floating Point Addition Example

- Consider Adding (Single-Precision Floating-Point):

  $+\ 1.11100100000000000000010_2 \times 2^4$

  $+\ 1.10000000000000110000101_2 \times 2^2$

- Cannot add significands … Why?

Floating Point Addition Example

- Consider Adding (Single-Precision Floating-Point):

  $+\ 1.11100100000000000000010_2 \times 2^4$

  $+\ 1.10000000000000110000101_2 \times 2^2$

- Cannot add significands … Why?
  - Because exponents are not equal
- How to make exponents equal?

Floating Point Addition Example

- Consider Adding (Single-Precision Floating-Point):

  $+\ 1.11100100000000000000010_2 \times 2^4$

  $+\ 1.10000000000000110000101_2 \times 2^2$

- Cannot add significands … Why?
  - Because exponents are not equal
- How to make exponents equal?
  - Shift the significand of the lesser exponent right
  - Difference between the two exponents = 4 – 2 = 2
  - So, shift right second number by 2 bits and increment exponent

$1.10000000000000110000101_2 \times 2^2$

$= 0.01100000000000001100001\ 01_2 \times 2^4$

Floating-Point Addition – cont'd

- Now, ADD the Significands:

$$+\ 1.11100100000000000000010 \times 2^{4}$$

$$+\ 1.10000000000000110000101 \times 2^{2}$$

$$+\ 1.11100100000000000000010 \times 2^{4}$$

$$+\ 0.01100000000000001100001\ 01 \times 2^{4}\ \text{(shift right)}$$

$$+10.01000100000000001100011\ 01 \times 2^{4}\ \text{(result)}$$

- Addition produces a carry bit, result is NOT normalized
- Normalize Result (shift right and increment exponent):

$$+\ 10.01000100000000001100011\ 01 \times 2^{4}$$

$$=\ +\ 1.00100010000000000110001\ 101 \times 2^{5}$$

Rounding

- Single-precision requires only 23 fraction bits
- However, Normalized result can contain additional bits

$1.00100010000000000110001 \mid 101 \times 2^5$

Round Bit: R = 1 *Sticky Bit:* S = 1

- Two extra bits are needed for rounding
  - Round bit: appears just after the normalized result
  - Sticky bit: appears after the round bit (OR of all additional bits)
- Since **RS = 11**, increment fraction to round to nearest

$$
\begin{array}{r}
1.00100010000000000110001 \times 2^5 \\
+1 \\
\hline
1.00100010000000000110010 \times 2^5 \text{ (Rounded)}
\end{array}
$$

Floating-Point Subtraction Example

- Sometimes, addition is converted into subtraction
  - If the sign bits of the operands are different
- Consider Adding:

```
+ 1.00000000101100010001101 × 2^-6
- 1.00000000000000010011010 × 2^-1
-----------------------------------
+ 0.00001000000001011000100 01101 × 2^-1 (shift right 5 bits)
- 1.00000000000000010011010       × 2^-1
-----------------------------------
0 0.00001000000001011000100 01101 × 2^-1
1 0.11111111111111101100110       × 2^-1 (2's complement)
-----------------------------------
1 1.00001000000001000101010 01101 × 2^-1 (ADD)
-----------------------------------
- 0.11110111111110111010101 10011 × 2^-1 (2's complement)
```

- 2's complement of result is required if result is negative

Rounding to Nearest Even

- Normalized result has the form: 1. f_1 f_2 … f_l **R S**
  - The round bit **R** appears after the last fraction bit f_l
  - The sticky bit **S** is the OR of all remaining additional bits
- Round to Nearest Even: default rounding mode
- Four cases for **RS**:
  - **RS = 00** → Result is Exact, no need for rounding
  - **RS = 01** → **Truncate** result by discarding **RS**
  - **RS = 11** → **Increment** result: ADD 1 to last fraction bit
  - **RS = 10** → Tie Case (either truncate or increment result)

Finished!

- Do the rest of the exercises
- https://www.h-schmidt.net/FloatConverter/IEEE754.html

Finished!

- Do the rest of the exercises
- https://www.h-schmidt.net/FloatConverter/IEEE754.html