UNIVERSITY OF TWENTE.

Pearls of Computer Science

Module 1.1, Study *Technical Computer Science*
Academic year 2016/2017

DATE
September 5, 2016
PARTIAL EDITION

Chapter 1: Pearl 000 — Computer Architecture

MODULE COORDINATOR
Maurice van Keulen

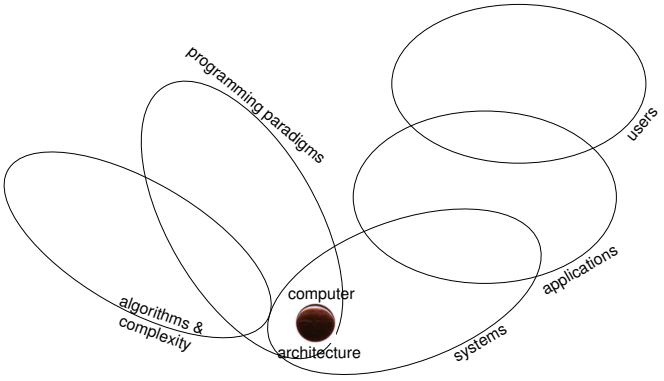
DESIGN TEAM
Maurice van Keulen
Arend Rensink
Pieter-Tjerk de Boer

INVOLVED TEACHERS
Harry Aarts
Rieks op den Akker
Pieter-Tjerk de Boer
Ansgar Fehnker
Marco Gerards
Celine Heijnen
Dirk Heylen
Maurice van Keulen
Frans de Kogel
André Kokkeler
Jan Kuper
Arend Rensink
Andreas Peter
Mannes Poel
Klaas Sikkel
Karen Slotman

FORMER TEACHERS
Pascal van Eck
Anne Remke
Dolf Trieschnigg

TEACHING ASSISTANTS
Kevin Alberts
Dex Bleeker
Tim Blok
Jeroen Bos
René Boschma
Dennis Cai
Sharbel Chmoun
Twan Coenraad
Frans van Dijk
Vincent Dunning
Thijs van Essen
Paula Felix
Noah Goldsmid
Aron van Harten
Joran Honig
Wim Kamerman
Lindsay Kempen
Etienne Khan
Dirk Koelewijn
Mark Kok
Jan-Jaap Korpershoek
Yoep Kortekaas
Remco de Man
Sander Meinderts
Ömer Sakar
Jasper Sustronk
Wouter Timmermans
Jeroen Weener

The Black Pearl is a real ship
Jack Sparrow — The Curse of the Black Pearl



Week 1

Pearl 000: Computer Architecture

1.1 Introduction

The first pearl of computer science that we will study, is the computer hardware itself. We will focus on two aspects: how can one calculate usefully with only zeros and ones, and how can a machine be built which can be "programmed" to do all kinds of calculations?

1.1.1 Global description of the pearl assignment

This week's pearl assignment is programming a simple small computer (the Arduino), in machine language (we'll see later what that means), such that a LED will flash and a buzzer will make sound. You work on this assignment in groups of two students.

To put things in context: programming in machine language is useful to get a good idea of how a computer works, but, as you will notice, it is quite complicated and error-prone. Therefore, in practice, programs are usually written in a so-called high-level language, and translated automatically (by another computer program!) into machine language. Next week, we'll meet one such high-level language, namely PYTHON.

1.1.2 Study material and tools

This week's study material can be found on Blackboard:

- A document about Boolean logic.
- A document about number representation.
- A document about computer architecture, which is mostly intended as background material; the lecture + slides + your own experience in the pearl assignment should suffice.

Also the slides of both lectures will be put on Blackboard.

Besides that, there are reference documents about the Arduino and the processor inside it:

- A document containing its instruction set.
- A very big document about the processor; in principle you don't need it, but it's there just in case.

U	Ma			Di			Wo			Do			Vr		
1	M	lec	T∞	P	lec	T∞	M	self	A∞	P	lec	T∞	M	self	N
2	M	lec	T∞	P	lec	T∞	M	self	A∞	P	lec	T∞	M	self	N
3	P	lec	T∞	P	self	N	M	tut	T∞	P	ass2	A∞	P	ass2	A∞
4	F	lec	T∞	P	self	N	M	tut	T∞	P	ass2	A∞	P	ass2	A∞
6	P	lec	T∞	P	prac	A∞	P	ass2	A∞	M	tut	T∞	P	tst	T∞
7	P	lec	T∞	P	prac	A∞	P	ass2	A∞	M	tut	T∞	P	tst	T∞
8	P	tut	T∞	P	ass2	A∞	P	self	N	M	self	N	P	ass2	A∞
9	P	tut	T∞	P	ass2	A∞	P	self	N	M	self	N	P	ass2	A∞

Abbr	Form
ass	Assignment
prac	Practical
lec	Lecture
pfb	Peer feedback
pj	Project
qa	Q&A
self	Self study
tst	Test
tut	Tutorial

Abbr.	Part
P	Pearl
F	Final project
S	Academic skills
M	Math

Abbr	Supervision
T	Teacher
A	Teaching assistent
N	None

Ma 1–2 Math.

3–4 Lecture. The first lecture gives an overview of the module, both about contents, what is Computer Science and what are you going to learn in this module, and about organisation, what do you have to do, when and who is going to help your with that. The first part of the lecture provides an overview of all of the contents of this module guide. The second part introduces the module's final project.

6–7 Lecture about *binary logic and calculation*. Computers calculate using bits, i.e., zeros and ones. Before we can study (tomorrow) how a computer works, we first have a look at how to calculate with zeros and ones. One aspect of this is the so-called Boolean algebra: a set of rules that are logical if 0 means false and 1 means true. Another aspect is how to represent numbers bigger than 1 and smaller than 0 by a row of bits.

8–9 "Werkcollege" about *binary logic and calculation*; the exercises are in Section 1.3.1

Di 1–2 Lecture about *computer architecture*. In this lecture we study the parts that just about every computer is made of, and how those parts work together: memory, registers, arithmetic and logic unit, input and output. As an illustration, we use a very simple computer, the so-called Arduino. In this week's pearl assignment, you'll work on this Arduino in practice.

3–4 Self study.

6–7 Start of the Arduino assignment. Each group of two students can borrow one Arduino from us for this week's assignment. Before starting the main assignment, we'll first do some introductory assignments to get acquainted with the Arduino. The assigments are in Sections 1.3.2 and 1.3.3.

8–9 Work on the assignment.

Wo 1–4 Math.

6–7 Work on the assignment.

8–9 Self study, both about binary computation and about computer architecture, if you didn't do this yet. Continue to finish the assignments about binary logic and calculation from Monday.

Do 1–2 Practice test. During the first hour, we do a practice test, and in the second hour we'll discuss the answers. On the one hand this serves as extra practice material, and on the other hand it gives you an idea of how well you understand the subject already. This test is about all topics of this pearl, and it is representative for the real test on Friday. Example questions can be found in Section 1.4.

3–4 Work on the assignment.

6–9 Math.

Vr 1–2 Selfstudy for math.

3–4 Selfstudy for the test / work on the assignment.

6–7 Test.

8–9 Finish the assignment.

Furthermore, we use the following tools to program the Arduino:

- A text editor of your choice (e.g., Notepad).
- A program which loads your Arduino program via a USB cable to the Arduino (`upload.bat`).
- A program called "Putty" in which you can see what the Arduino sends back; this is useful to find errors in your program.
- A program which translates machine languages commands ("mnemonics") into hexadecimal numbers (avra and `asmupload.bat`).

The latter three are in a zip file which you can find on Blackboard.

1.1.3 Obligatory sessions and deliverables

Compulsory elements of this week are:

- Participation in the Academic Skills-session.
- Participation in the test.
- Handing in the pearl assignment, and demonstrating it to the teaching assistent.

1.2 Academic Skills

There is no session for Academic Skills this week, but you need to do a preparatory assignment:

- LA1: Read chapters 1 and 2 of the book
  Self assessment (short form on Blackboard)
  Deadline: Next week Tuesday 13 sep 3+4 hour (bring your form to the session)

More information on the Academic Skills sessions, the assignments, and the study material can be found on Blackboard $\rightarrow$ Week 1.

1.3 Description of the sessions and lab assignments

1.3.1 Tutorial exercises

☞ **1.1** Use Boolean algebra to prove the following formulas:

(a) $1(1+1) = 1$

(b) $\overline{(\overline{A}+\overline{B})} = AB$

(c) $\overline{A}B + A\overline{A} + BB = B$

□

☞ **1.2** Use the result from (b) in the previous exercise to sketch how to build an AND gate from just NOR gates. □

☞ **1.3** Give a realisation of $F = (A \cdot B) + C$ in which only AND gates and inverters are used. Use Boolean algebra to show that your idea is indeed correct. □

☞ **1.4** Do the following number conversions:

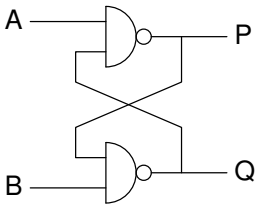
(a) 001 unsigned binary to decimal
(b) 110 unsigned binary to decimal
(c) 110 2-complements binary to decimal
(d) 15 decimal to unsigned binary

(e) -5 decimal to 5-digit 2-complements binary
(f) -5 decimal to 5-digit 1-complements binary
(g) 1D hexadecimal to decimal
(h) 100 decimal to hexadecimal
(i) E9 hexadecimal to binary
(j) 010101100111 binary to hexadecimal

□

☞ **1.5** Consider adding two numbers of one bit each. This results in a two-bit number. Make a truth-table for this, a logical schematic diagram in which you can use any desired type of gate, and a diagram in which you use only NAND gates. □

☞ **1.6** Make a truth table for the following schematic diagram.
Hint: you'll notice that for some combination of input signals, this circuit behaves in a fundamentally different way than the previous circuits; try to describe what this behaviour is.



□

1.3.2 Introductory Arduino assignments

In this pearl, you will be writing small machine language programs for the Arduino. As we've seen in the lecture, such a program consists of a pattern of zeros and ones in the memory of the computer (the Arduino in our case). In the very first computers, one could put those zeros and ones directly into the memory of the computer using switches, but that's not possible with the Arduino. We therefore use a tool. We write the zeros and ones hexadecimally into a text file on a normal PC or laptop, and then upload these to the Arduino using an auxiliary program (available on Blackboard). For that hexadecimal text file, of which you'll find an example below at the first assignment, the following rules hold:

- Each Arduino instruction consists of 16 bits, so 4 hexadecimal digits.
- You can put multiple Arduino instructions on a single line, separated by spaces and/or tabs.
- If on a line there is a semicolon (`;`), everything after that is ignored. This is very handy to write text after the hexadecimal code which is readable for humans (called "comments"), like we also do in the example.
- The first hexadecimal number will end up at adress 0 in the Arduino's memory, the second on address 1, and so on. If you want to deviate from this (but that is surely not needed for the introductory assignments), you can write a hexadecimal number preceeded by an exclamation mark (`!`) in the text file: that number will then be used as the starting address for writing the following numbers (without exclamation mark) to the Arduino's memory.

To find out which instructions the Arduino has and what their encoding is, there is a document called "instruction set" on Blackboard.

As an example and starting point, we give you the following program:

```
e200    ; ldi r16,$20     Register 16 now contains 0010 0000

b904    ; out 4,r16       We write r16 to the Data Direction Register:
        ;                 it makes the pin to which the LED is connected act as an
        ;                 output pin. This needs to be done only once: the pin
        ;                 remains an output as long as the program is running.

b905    ; out 5,r16       Write that same 0010 0000 to the output register; thus
        ;                 the pin is really set to 1 now, making the LED light up.

cfff    ; rjmp -1         Endless loop, don't do anything anymore.
```

Explanation:

- `ldi` means "load immediate", and the dollar sign (in the comments) is a reminder that the number is written in hexadecimal.
- The `out` instruction sends a byte to a special register that controls external hardware; in our case, this external hardware is the LED. These I/O registers are comparable to the normal registers `r0` through `r31`, but have their own numbering (addresses), and have their own instructions for writing to and reading from them.
- The Arduino processor has a number of Input/Output ports[1]. Each I/O port consists of 8 pins (electrical connections) of the processor. Each of those pins corresponds to one bit in a byte that can be written to that I/O port, or read from it. Each pin can be used either as an input (e.g., to read the status of a switch or sensor), or as an output (to drive an LED, buzzer, etc.). Since a pin cannot be used as both an input and an output at the same time, we need to tell the Arduino which pins are to be used as inputs and which as outputs, as will be explained below.
- The first I/O register we use, is officially called (in the processor's documentation) DDRB: "Data Direction Register for port B", and has address 4. The bits in this register determine whether that pin of the processor will be used as an input (in case of 0) or as an output (in case of 1). Typically, the bits in this register would be set somewhere at the beginning of the program, and then remain untouched in the rest of the program.
  In our example program, the binary value `0010 0000` is written to this register, using the `out 4,...` instruction. It will switch one pin to be an output, namely the pin corresponding to the 3rd bit (counting from left). This is the pin the LED is connected to.
- The second I/O register we use, is officially called PORTB, and has address 5. The bits you write to it determine whether the corresponding pin of the processor will have a low voltage (0 volts, if the bit is 0), or a (comparatively) high voltage (5 volts, if the bit is 1). Note that this only works if that pin had previously been made an output by writing a 1 bit to DDRB, as noted above.
  In our example program, we write the binary value `0010 0000` to this register, setting one pin to the high voltage. This pin is the one connected to the LED, thus lighting it up.
- How do we know which of the 8 bits in DDRB and PORTB is the one connected to the LED? For the Arduino, I just told you in the above. In general, one would have to look into the Arduino documentation, or follow the wires under the paint on the Arduino board to find out which pin of the chip is connected to which external hardware.
- `rjmp` means "relative jump": a jump to an address that is calculated with respect to the address where we would otherwise (i.e., without the jump) be. So –1 means jumping back by 1 step, while normally (i.e., without a jump) we'd do 1 step forward (because we normally execute the instructions one by one), so the net result is doing the same instruction again. And again. And again...
  Besides these relative jumps, the processor als has "absolute" jumps, in which the address to which to jump is coded directly, not relative to the current address.

[1] In Dutch, the word "poort" is used both for this kind of I/O ports, and for logic gates (AND, OR, NAND etc.). In English, the words are different.

Instructions for use of the Arduino tools on MS-Windows:

- Unpack the ZIP file from Blackboard; this creates a new directory (called "`module 1.1 week 1`") with some contents. That new directory will henceforth be called your working directory. It is also in this directory that you should put the programs (hex files) you write yourself.
- Install `arduino-1.0.5-windows.exe`
- Install `WinAVR-20100110-install.exe`, but **watch out** while doing this, since by default this installer messes up the PATH variables. Therefore, proceed as follows:
  - During the installation of WinAVR:
    - Remember the directory where WinAVR was installed, the default is: `C:\WinAVR-20100110`
    - On the screen "Choose Components" **de**select "Add Directories to PATH (Recommended)". (This option will break it.).
  - After the installation:
    - Start SystemPropertiesAdvanced.exe (For example using [Win]+R).
    - Click "Environment Variables..", the button in the bottom right corner.
    - In the list below "System variables" find the variable "PATH" and click "Edit..."
    - Now add `C:\WinAVR-20100110\bin;C:\WinAVR-20100110\utils\bin;` if you installed WinAVR in `C:\WinAVR-20100110`; otherwise you will have to replace `C:\WinAVR-20100110` with the installation directory. ! Beware to not break the path yourself !
- Connect the Arduino.
  - Windows 7/8/10: Open Start > Apparaten en printers (or Devices and Printers). Look for "Arduino (COMnr)" and remember the COM port number.
  - Windows Vista and older: Open "Apparaatbeheer" (right mouse button > Deze Computer > Eigenschappen > Apparaatbeheer). (or This Computer > System Properties > Device manager) Look for "Arduino (COMnr)" and remember the COM port number.
- Open a command prompt (cmd.exe) in the working directory. In case you're unfamiliar with this, there are several ways to do this:
  - In Windows 8/10:
    Go to your working directory in Explorer, and choose "Open command prompt".
  - Or, in Windows 7:
    Go to your working directory in Explorer, keep Shift pressed while right-clicking on the folder, and choose "Open command window here".
  - Or, just start `cmd.exe`; then, at the command prompt, use the `cd` command to navigate to your working directory.
- To upload hex files to the Arduino:
  Given a hex file `demo.txt` and the Arduino connected to COM13, type at the prompt:
  `upload com13 demo.txt`.

Instructions for use of the Arduino tools on MacOS:

- There's a file called `pearl000-for-MacOS.zip` on Blackboard, containing both some needed scripts and a README file explaining how to install and use this.

Brief instructions for those who want to use (Ubuntu) Linux:

- Install avrdude and cutecom: `apt-get install avrdude cutecom`
- Add your own account to the group which is allowed access to the COM ports: `adduser` *your-username-here* `cout`
- Get `hex2hex.c` from the Windows zip file and compile it: `make hex2hex`
- Make your own equivalent of the `upload.bat` you find in the zip file.

☞ **1.7** Type the above program in Notepad (or another editor), and save it as `prog7.txt`.[2] Upload the program to the Arduino, and check that it works.
Look up the three instructions from the above example program in the instruction set document (available on Blackboard), compute their hexadecimal codes, and compare those to the codes in the program (this may seem useless now, but you'll need it later). Hint: skip the first 15 pages of the instruction set document; pages 16 through 157 describe all instructions in detail, in alphabetical order of their mnemonics. □

☞ **1.8** Change the program such that it switches the LED off; save this to a file called `prog8.txt`, and check that it works.
Note: for all programs, please always write a comment after every instruction with its mnemonic! This is not only handy for yourself, but also for the teachers and teaching assistants. □

☞ **1.9** Make a program called `prog9.txt` which does the following: 47 times `nop`, then switch the LED on, then another 47 times `nop` then switch the LED off, and then jump back go the first `nop`. Calculate how quickly the LED will be switched on and off, knowing that the Arduino runs at 16 million clock cycles per second; what do you expect to see from this with the naked eye?
Note: simply put the 47 `nop`s as 47 `nop`s one after the other in your program; only later we'll do this more efficiently by programming a loop. □

When programming the Arduino, it would be handy to peek into the chip to see the contents of the registers, especially if it doesn't do what you expect it to do. Some of the earliest computers indeed had lamps (LEDs hadn't been invented yet) on their front panel showing the state of all the bits in their registers, but the Arduino is too small for that. Therefore, we (the teachers) have made a small program which you can use to send the contents of `r16` back to your laptop via the USB cable. To use this, put the following code in your Arduino program:

```
940e 0200    ; send r16 to laptop
```

Instructions for viewing the data sent back to the laptop by the Arduino:

- Upload the program containing the above hex codes to the Arduino.
- Open Putty.exe. Choose the "Serial" connection type and the right COM port.
- Click "Open". The Arduino will restart and as soon as the above line of your program is reached, the content from `r16` will be sent to the laptop and displayed by Putty (in binary).
- Close Putty again before you upload a new program to the Arduino, otherwise the upload is disturbed (since Putty and the upload tool try to receive from the Arduino simultaneously).
- For Linux users: use cutecom instead of putty.

☞ **1.10** Try this: write a program called `prog10.txt` which writes the number 3B hex into r16 and sends it to your laptop. Verify that the bits your laptop shows indeed match 3B hex. □

☞ **1.11** Experimentally verify that the `subi` instruction indeed uses 2-complement notation for negative numbers. □

By now, you're probably fed up with translating the instructions into hexadecimal codes by hand, and having to count precisely where the `rjmp` goes. It's good to have seen once how this works, but it is not a handy way to write large programs.

[2] It would be more logical to let the filename and in `.hex` or something like that, but Notepad has the nasty habbit of adding another `.txt` if you try that.

That's why, already quite early in the history of computing, so-called *assemblers* have been created: computer programs which can read the mnemonics and convert them into the corresponding binary or hexadecimal codes. From now on, you're allowed to use the assembler which you can find on Blackboard. As an example, we give the example program from page 22, but now in assembly (that's what the assembler's language is called) format:

```
.device atmega168      ; what type of processor do we use?

.equ DDRB = 4
.equ PORTB = 5

   ldi r16,$20         ; register 16 now contains 0010 0000
   out DDRB,r16        ; write this to the data-direction-register
   out PORTB,r16       ; and to the port itself, which switches the LED on

again:                 ; this is not an instruction, but a label, as
                       ; indicated by the colon at the end

   rjmp again          ; infinite loop: jump back to the label
```

Note how the assembler makes our life easier:

- we no longer need to lookup the hexadecimal codes for the instructions;
- we can give names to numbers, as was done in the example for `DDRB` and `PORTB`, which makes mistakes less likely and makes the program more readable;
- jumps are computed automatically: we can give names to places in the program, like `again` in the example, and use that name in the jump instruction.
  Note: always use labels; do not try to use numbers behind `rjmp`, since the assembler will try to interpret such numbers as absolute addresses and convert them to relative offsets for you.

And this is just the start. When using an assembler like this, there is still a one-to-one correspondence between what you type and the individual instructions given to the processor. The next step up is the use of a "higher-level programming language", in which that correspondence is no longer there; one example of such a language is Python, which we will use next week. Computer scientists and engineers have created increasingly more advanced tools to allow describing what a program should do at increasingly high levels of abstraction (and thus easier and with fewer errors). However, it is good to keep in mind that, with whatever tools you create programs, in the end they will be converted to machine language instructions, because those are the only ones that the computer can really process.

Instructies for using the assembler tool:

- To upload assembly programs to the Arduino:
  Given the assembly file `demo.txt` and the Arduino connected to COM13, type at the command prompt: `asmupload com13 demo.txt`.
- Reading a register from the Arduino now works differently:
  - Instead of the hex codes given previously, put the following line into your assembly program at the point where you want to send `r16` to the laptop: `call sendr16tolaptop`
  - At the **eind** of your asm file, insert the following line: `.include "rs232link.inc"`

Brief instructions for (Ubuntu) Linux:

- Install avra: `apt-get install avra`
- Make your own equivalent of the `asmupload.bat` contained in the zip file for Windows.

☞ **1.12** Put the above program into a file called `prog12.txt`, upload it via the assembler to the Arduino, and observe that it works. Then convert also your program from assignment 1.9 to assembly and try it. □

1.3.3 Main Arduino assignments

☞ **1.13** Add so much delay to your previous LED program, that the LED flashes at a rate which is visible for humans. Note that you can't do this simply by adding nops, because the Arduino doesn't have enough memory for so many nops (millions!). So you'll have to think of something else.
Hint: try to build and test parts of your solution, instead of hoping to get it all correct at once.
Let a TA sign off your work. □

Besides the LED, you can also connect a buzzer to the Arduino. Do this by inserting its black wire into the connection marked "GND" and the red wire into the connection marked "12". The buzzer won't buzz by itself though. Your program should make pin 12 an output (like you already made the LED pin an output); this pin corresponds to bit 0001 0000 in the DDRB- and PORTB-registers. Next, you have to switch this pin between 0 and 1 at a frequency which corresponds to an audible tone (e.g., about 1 kHz).

☞ **1.14** Set yourself an aim w.r.t. the use of the LED and/or the buzzer, e.g. letting the LED flash in a morse-code pattern, make the buzzer make a siren-sound or even play a Beethoven symphony. Discuss your plan with the teaching assistant, and realise it.
Hint: again, don't try to write a working program for your final goal all at once, but build in steps, where you test each step separately (with LED and buzzer, and/or by sending data back to your laptop).
Let a TA sign off your work. □

Bonus points

In this pearl, two half bonus points can be earned.

Half bonus point: music

One half bonus point can be earned from assignment 1.14 by having the Arduino play a *clearly recognizable melody* of at least 12 tones, and making good use of the `call` instruction (look up for yourself what this does). Whether the melody is recognizable, is to be judged by the teaching assistant.

Half bonus point: multiplication

The other half bonus point can be earned by writing a program which multiplies two 4-bit (unsigned) numbers, resulting in an 8-bit unsigned number, *without* using the processor's multiply instructions (`mul`). Also, it is not allowed to simply do a repeated addition, because that would be inefficient for large numbers. Think of you how you learned to do multiplications of decimal numbers in primary school, and translate that to binary numbers. You may want to use the processor's *rotate* and/or *shift* instructions. The program should expect the two numbers to be multiplied in registers `r17` and `r18`, and deliver the result in `r16`, so it can be sent to your laptop for testing.

Note that in this bonus assignment, the two aspects of this pearl (binary calculation and machinecode programming) meet.

Your bonus depends on how short your program is: the shortest program submitted will get the entire half point, longer programs get a proportionally smaller bonus. Hint: it's possible to do this in less than 10 instructions!

Let a TA sign off your work.

Handing in

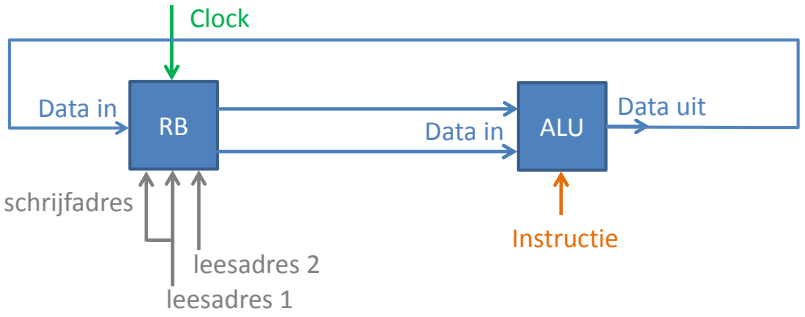
You can hand in your work via Blackboard. Hand in your assembly files for assignment 1.14, and if applicable, also the assembly file for the second bonus question. Each file should have sufficient explanations

("comments") to make clear to the teachers how your program works. If you hand in multiple files, please do *not* put them together in a `.rar` or `.zip` archive.

1.4 Example test questions

The test will be both about binary arithmetic and logic, and about computer architecture.
Example questions about binary arithmetic and logic can be found in Section 1.3.1. Example questions about computer architecture are given below.

1.



[Translation: uit = out, schrijfadres = write address, leesadres = read address, instructie = instruction]

The above simple processor has two instructions: 0 = '+' (add) and 1 = '*' (multiply).

Give the program for this processor to do the following computation: R2 = (R1+R2)*(R0+R1)

	read adress 1 / write address	read address 2	instruction
Time slot 0			
Time slot 1			
Time slot 2			
Time slot 3			
Time slot 4			
...			

2. Given the following AVR program ("BRNE" means "BRanch if Not Equal", "MUL" means MULtiply, "DEC" means "DECrement (subtract 1)" and "SUB" means "Subtrac t"):

```
LDI  R16, $03
LDI  R17, $03
LDI  R18, $02
LDI  R20, $01
MUL  R17, R18
DEC  R16
MOV  R19, R16
SUB  R19, R20
BRNE -5
```

Give a table showing how the contents of the registers changes while this program is executed; you can choose to use either hexadecimal or decimal numbers, but do show which of the two you choose.

How many clock cycles are needed for the execution of this program?