IN1805II Project Manual

Thomas Schaap & Boaz Pat-El
Original version by Sander Koning & Bas van der Doorn

April 7, 2009

Abstract

This manual contains the project assignment, reference material, and some hints to students for the Operating System Concepts project, course code IN1805II. The reference material includes a short description of the 6502 microprocessor used in the project.

The information in this manual has been kept as complete as possible, but no guarantee can be given that all information needed is readily available. The assistants are happy to help you out, but try and find a solution for your problem yourselves first.

Contents

1 Introductory notes

1.1 Code

In this project, you will have to hand in various things at your project assistant. A very important part of your deliverables is code. Writing good and legible code is mandatory in all projects, and since the code for this project is to be in x86 ("Intel") assembly, this holds even stronger. Code alone does not suffice, extensive and clear comments are required as well. As the IN1705II manual already mentioned, goals and functionality of your code should be understandable by reading only the comments. Apart from that, we also ask for correct and elegant code. Clear laziness in programming, or code that does not do what it should, will not receive high marks.

Please make sure that your deliverables are as consistent as possible. Think of consistent use of registers and stack, but also of consistent looks: the use of whitespace, the markup and style of your comments, and last but not least the natural language you use in designs, code, and report.

1.2 Designs

Another important part of the deliverables are code designs. Before you start to write your code, have the global idea approved by the assistants. Code for which the design has not been approved may not be accepted, in which case it is your own responsibility to correct this. The amount of detail in which the design needs to be specified, is discussed with the assignment.

1.3 Testing

When writing assembly code, it is sometimes quite hard to know if your code works correct, since dependencies to other, still to be written, code often remain. To partially solve this problem, you can ask your project assistant to test your code by running it through a testsuite. Since one can easily overlook something when reading assembly, this testsuite is also used by the assistants to check your code for correctness. A testsuite is available for every point in the assignment where you need to show your code to an assistant.

Please make sure you stick to the conventions mentioned in section 2.8. Without those conventions it would be near impossible to automatically test your code.

1.4 Progress

Every group is different. Chances are high that some groups finish the assignment well within the allocated time. It is therefore possible to implement one or more additional modules, or expand the existing ones, to try and get a higher mark. On the other hand, groups having problems with finishing in time can obtain one or more precompiled modules, so that they can focus on getting other parts to work. This will of course reduce the final grade.

To keep track of each group, the assistants want to know about your progress on a regular basis. Therefore, each group is to have an internal meeting once a week, of which the agenda and the minutes are project deliverables. In these meetings, you should discuss whether you are on schedule, whether you have any serious problems, and of course, when applicable, what design decisions have been made and why.

Of course, you will also want feedback on your progress. Apart from the assistants being available for (semi-)individual help during the project sessions, there are some plenary feedback sessions in which the most important points of attention will shortly be discussed. See the separate handout for the exact dates.

1.5 Grading

Your grade will consist of a group mark for the final result (product and process), the report and a personal mark based on the peer reviews and the assistants' impressions. See section A.2 for more information.

2 Assignment

2.1 Introduction

Your main goal of this project is to write an emulator for the 6502 microprocessor. This means that your program, which you will write in x86 assembly code, will read and execute machine code originally written for the 6502. The main part of the implementation is the fetch-decode-execute cycle, but you will also need to write debugging subroutines and take care of some other peculiarities you will encounter along the way.

The project is divided in various modules, which will together form the emulator. These modules will have to be implemented partly sequentially, partly in parallel. As a rule of thumb, when parts can be implemented in parallel, do so. See the next sections for detailed instructions.

Since parts of the project are developed in parallel, various people will be working on different, interacting parts of the code at the same time. To make sure that you are able to call each other's code, you will have to agree on things like naming and argument passing before you can start your implementation. Some guidelines for naming important routines are already given in the assignment, but do consider your own design and decide on any extra agreements it requires.

Section 3 describes the 6502 in more detail. You will need to read this section carefully. Make sure you fully understand a certain part of the 6502 architecture before starting to implement that part in your emulator. Discuss with your group members and do not hesitate to ask the assistants. Some things are trivial. Others are definitely not.

To guide you in implementing the emulator modules in a suitable order, the project is divided in various phases. At the end of each phase, you will have a working emulator. Its capabilities are limited in the first phases, though.

In phase I, you will implement instruction fetching and debugging. This is followed by instruction decoding in phase II and execution in phases III and IV. During phase V the report is written. The faster groups also have an opportunity to improve their grades by implementing one or more extras, such as input and output, during phase V.

We will now first give an overview of the modules you are going to implement. After that, the various phases of the actual assignment are discussed in more detail.

2.2 Module listing

As described in the previous section, you will have to implement your project as various modules. To make your and the assistants' lives easier, these modules should be kept in separate files. The various modules to be written are:

- the main program and basics module, `6502.s`, which contains the registers and other globally used data, and the emulator initialization and termination code;
- the program loading module, `readprog.s`, which contains the function to load a program into the emulator;
- the decode module, `decode.s`, which decodes the instruction and calls the required function(s) to execute it;
- the operand fetch module, `operand.s`, which contains functions that fetch an operand according to the required addressing mode;
- the execute module, `execute.s`, which contains the functions to actually execute the various instructions;
- the debug module, `debug.s`, which offers functions to display status information about the current instruction, its operand and the registers.

These modules are mandatory and together correspond, intermixed, to phases I through IV. You will also create the file `main.s` containing a very simple main function.

2.3 Phase I: The fetch-debug cycle

When phase I is complete, your main module should do the following:

- reserve 65536 bytes of main memory;
- create and initialize the A, X, Y, PC, S, IR and P registers;
- define debugging strings for the current address, the current instruction, the P register status and any other data you see fit;
- initialize the memory with zero bytes;
- initialize the program counter (subroutine `initpc`);
- start the fetch cycle (subroutine `fetch`).

See sections 3.1, 3.2 and 3.3 for details on the registers, the memory layout and the stack, respectively. Your `fetch` subroutine should do the following, for now:

- read the instruction at the current address;
- call the `showi` debug routine;
- if the current opcode is 0xDB (instruction STP), return to the main module, otherwise increase the program counter and restart the cycle by reading the next instruction.

Your debug module should initially contain one subroutine:

- `showi`, which displays the current program counter (as a 4-digit hexadecimal number) and opcode (as a 2-digit hexadecimal number), in a legible way.

This `showi` routine should also serve as a safety mechanism: it should bail out of the program (i.e., end the emulator) if an address higher than 0xFFFF or an instruction higher than 0xFF is encountered, since this would mean that your emulator produces values which would be impossible to obtain on the real 6502 —try it yourself: put the value 256 in an 8-bit register. An error message stating the cause of the bailout should be displayed in this case.

Before you start implementing the fetch and debug modules, have the code of your main module and a design in pseudocode for your fetch and debug modules approved by an assistant.

Keep in mind that all symbols (functions, constants, variables) that you want to share across source files, have to be declared global (use the `.global` directive, see section 3.5.3 of the IN1705II manual) before you define them.

You should be abled to use `gcc -o 6502 -m32 *.s`[1] to compile your emulator, receive no warnings or errors, and run the compiled program with `./6502`.

Test your emulator thoroughly, and modify your code if needed. To help you, a routine which loads a very small sample 6502 program is given below. Copy this routine to your `readprog.s` and add a call to it in your main program, before the call to `initpc`.

```
#
# Read program into memory
#
readprog:
   pushl  %ebp               # push base pointer
   movl   %esp,  %ebp        # copy stack pointer
   movl   $0xfffc, %ecx      # FF:FC contains minor byte of
   movb   $0x00, MEM(%ecx)   #    initial PC (here: 00)
   incl   %ecx               # FF:FD contains major byte of
```

[1] -m32 makes the compiler compile 32 bit assembly on 64 bit operating systems.

```
   movb   $0xfe, MEM(%ecx)   #    initial PC (here: FE)

   movl   $0xfe00, %ecx      # start the program at FE:00
   movb   $0xe8, MEM(%ecx)   # put an INX here
   incl   %ecx               # next memory cell
rpli:
   movb   $0xc8, MEM(%ecx)   # put an INY here
   incl   %ecx               # next memory cell
   cmpl   $0xfe05, %ecx      # if we are not at FE:05 yet
   jl     rpli               #    then do so again
   movb   $0x9a, MEM(%ecx)   # put a TXS here
   incl   %ecx               # next memory cell
   movb   $0x98, MEM(%ecx)   # put a TYA here
   incl   %ecx               # next memory cell
   movb   $0xdb, MEM(%ecx)   # put a STP here
   movl   %ebp, %esp         # clear local variables
   popl   %ebp               # restore base pointer
   ret                       # return from subroutine
```

The program consists of 8 instructions, represented by the opcodes 0xE8 0xC8 0xC8 0xC8 0xC8 0x8A 0c98 0xDB. Make sure all instructions are read correctly and that the program stops after having read the STP instruction.

2.4 Phase II: Decoding the instruction

The next step is decoding the opcode which was fetched. Because you will have to determine the actual instruction as well as (for most opcodes) the addressing mode, this requires some smart coding.

Refer to section 3.4 for the instructions that should be available in your emulator.

The real 6502 determines the operation and addressing mode by looking at bit patterns in the opcode. You can do so as well in your emulator (and it is surely a good way to mimic the 6502 internals as closely as possible), but be adivsed that this requires a lot of comparisons and conditional jumps, which make program flow quite complicated. See the website `http://www.llx.com/~nparker/a2/opcodes.html` for details.

Another way of implementing the decoding of the opcode is by using a large jump table (see section 3.2.5 of the IN1705II manual). In this case every opcode has its own handler function in the jump table. Entries for opcodes that have no legal instruction attached would point to an error handling subroutine which takes care of the illegal instruction.

In case of instructions that don't have different addressing modes, this handler function could directly carry out the insrtruction. If more addressing modes are possible, the handler function, on its turn, could contain two function calls: first to a function that reads the operand, then to the function that actually executes the instruction once the operand is known.

For example, take opcode 0x29 (AND using the immediate addressing mode). Say that the correponding entry in the jumptable is called `do_and_immediate`, then the only things this subroutine would do are:

```
do_and_immediate:
    # pointer preservation
    call    fetch_immediate_operand
    call    execute_and
    # pointer restoration
    ret
```

You might prefer somewhat shorter names for your subroutines than shown above, but keep them clearly understandable (did we mention comments?).

Decide upon which method you are going to use to decode opcodes (bit pattern analysis, a single large jump table or another method). Create a naming scheme for the required subroutines. Explain (on paper) how this will work in your fetch-decode-execute cycle, using the actually implemented code for the parts that are already there, and pseudocode for the parts that correspond to this phase. Have this design checked by an assistant.

Implement the calling of the handler functions and add stubs for these to your execute module. These stubs do not have to contain useful code yet, but should just directly return. (Hint: make use of the fallthrough principle.)

Implement the illegal instruction handler and test your emulator. You can do this by modifying the program above. You will probably notice, though, that testing this way becomes tedious. Now would be a good time to take a look at a more useful `readprog` routine, perhaps one that reads a compiled 6502 program from disc and loads that into memory. A 6502 compiler is available on Blackboard. Either way, don't forget to test illegal opcodes.

Then continue your implementation to have any handler functions actually call the subroutines for fetching the operand and executing the instruction. Implement stubs for these that only contain the boilerplate code for saving and returning the x86 stack. You should have 57 stubs in your execute module (one for each instruction) and 12 stubs in your operand fetch module (one fore each addressing mode).

2.5 Phase III: Executing basic instructions

First, extend your debug module with two subroutines:

- `showr`, which displays the contents of the A, X, Y, S and P registers. All registers should be displayed as a 2-digit hexadecimal number;
- `showo`, which displays the effective address of the operand for the current instruction (or the operand itself, in the case of accumulator and immediate addressing modes), as a 4-digit hexadecimal number.

The rest of phase III is divided into two parts.

2.5.1 Part A: Addressing modes

Implement your operand fetch module. Subsection 3.5 gives a detailed description of the various addressing modes. The current reference emulator used by the assistants, uses a global variable containing the x86 address of the operand; the old reference emulator used to return the 6502 address in register `edx`. Whether you find these example conventions useful or not is entirely up to you, but do choose a common and consistent method to return the effective address so that the execute module can make use of it. You could also use the stack, for instance.

Make sure that after fetching an operand, the program counter is set to the next memory byte, so that the next instruction is fetched from the right address. This is especially important for the branching instructions you are going to implement in the next phase.

Test your emulator again and make sure that all addressing modes fetch the correct value. Have an assistant approve your code.

2.5.2 Part B: The first instructions

Now it is time to implement the first part of the execute module. Write the execute routines for all instructions that do not have to do with jumps, subroutines or the decimal mode. That is, all instructions that are listed in table 2 and that have been marked. Make sure you understand every instruction before implementing it in your emulator. An explanation of all instructions is available in section 3.4.

Again, test and then show the working emulator to an assistant.

2.6 Phase IV: Branching, subroutines and decimal mode

Now that you have a basic emulator, it is time to add the interesting stuff.

First, the decimal mode. The 6502 has native support for BCD-coded numbers for addition and subtraction. Add two routines to your execute module, `tobcd` and `frombcd`, which will convert a numer to and from BCD notation, respectively. Decide how the value is to be passed and returned, and document this. Implement the `SED` en `CLD` instructions and extend your implementation of the `ADC` and `SBC` instructions so that they operate with BCD coded numbers when the decimal mode flag is set.

Also implement the other instructions from table 2. These are the comparison, branching, subroutine/jump and interrupt related instructions. Note that the branching instructions all use the relative addressing mode, which requires a little attention. Take care as well with the stack usage by the subroutine related instructions.

At the end of this phase you will have a fully functional emulator. Have an assistant approve it after extensive testing.

2.7 Phase V: Extras and project report

During phase V you will extend your emulator with some form of input and output, and continue in a direction you want to explore. There are many directions to explore. Examples from the past are a small game for the 6502 in graphical mode, more emulated hardware, or a live debugger for the 6502 code running in the emulator. Whatever crazy ideas you have, tell an assistant about them. They might very well work.

Before starting phase V, draw up a plan for the phase. This should include what type of input and output you are going to implement and how you wish to continue. Have an assistant approve the plan before continuing.

After implementing your ideas, test your emulator and extensions, making sure all functionality is correctly implemented. Hand in your well-documented code to an assistant.

During phase V you will also write your final report. Read section A.3 for more information on the report.

2.8 Naming conventions and other requirements

As described in section 1.3 your code will be tested using an automated testsuite. The testsuite puts a number of restrictions on your code, to allow it to be tested. Please make sure your code adheres to the following requirements:

- your emulator starts with the routine `start`, which takes no arguments. This routine is used instead of the usual `main` routine;
- always end the execution of your emulator with a call to `exit`. Provide it with an argument on the stack: 0 if everything went fine, or something else if an error occurred, such as an illegal instruction. The routine `start` should never return;
- your registers are called A, X, Y, PC, S, IR and P, and have all space for exactly one byte, with the exception of PC, which has two bytes of memory;
- your memory array is called MEM;
- values stored in the PC register are little endian, so the least significant byte comes first;
- the file `readprog.s` contains one global subroutine: `readprog`;
- don't use any symbols or names starting with `_00_`.

Since you won't be having the usual main routine, include a file `main.s` with the following content:

```
# main.s
# Jump to the start

.text
.global main

main:
    jmp start
```

2.9 Software tips

When writing and testing your code you are bound to run into problems. Here you will find a few pieces of software that can help you out.

2.9.1 kdbg

Although gdb is present on all linux computers of EWI, many find its use cumbersome or awkward. An alternative is the KDE frontend kdbg, which is also present. When you start it for the first time, be sure to setup your views correctly. Useful views are the stack, the registers, watched expressions, and any others you may want to watch. If you have the breakpoints view active, you can enter the name of one of your procedures and click `Add Breakpoint` to add a breakpoint at the start of that procedure. Using the buttons at the top you can then step through your program or run until the next breakpoint.

2.9.2 64 bit operating systems

If you're working on your own computer you will encounter problems with compiling if you have a 64 bit operating system. For this to work you will need 32 bit libraries installed.

For Debian based systems the following command will install them:
`apt-get install libc6-dev-i386 gcc-multilib`

Other systems have not been encountered yet. If you do, please tell the assistants how you've managed to install the 32 bit libraries, so they can add instructions here.

3 6502 Microprocessor reference

The microprocessor emulated in this project is the 6502, which can be found in for example the Apple II. Because this processor is very suitable for further extension, many manufacturers have made varieties which differ more or less from the basic 6502 and each other. These extensions are not part of the assignment (apart from the `STP` instruction which is included for convenience), the basic instruction set holds enough difficulties to gain insight in the workings of a microprocessor. Of course, groups wanting to do so, are allowed to implement extensions during phase V (after consulting the assistants).

3.1 Registers

The 6502 microprocessor is a full-fledged 8 bit processor. This means that instructions, operands and registers are 8 bits (1 byte) long.

The 6502 has the following registers:

- an 8 bit instruction register (IR). This contains the currently executed instruction;
- an 8 bit accumulator register (A). This register can contain an operand for an instruction or the outcome of arithmetic and logical instructions;
- two 8 bit index registers (X and Y). These can be used as normal registers, but are also used for the indexed addressing modes;
- an 8 bit processor status register (P). This register contains various processor status flags;
- a 16 bit program counter register (PC). This register contains the memory address which is currently used to read an instruction from. At the start of the program, the PC value is read from memory addresses (FF:FC) and (FF:FD);
- an 8 bit stack pointer register (S). This register indicates the next available stack address. At the start of the program, S should be set to 0xFF in order to maximize the available stack space.

The P register contains 8 bits, with bit 0 being the least significant, which are all considered individually:

- bit 0 is the carry flag. This bit is set or reset by instructions that can yield a result with carry (higher than 0xFF, or 0x99 when using BCD notation), and can be explicitly changed by the user;
- bit 1 is the zero flag. This bit is set by certain instructions when the result of ofr example a calculation is zero;
- bit 2 is the IRQ (interrupt request) disable flag. A value of 1 indicates that maskable interrupts are not being handled, a value of 0 indicates that maskable interrupts may safely be handled. Non-maskable interrupts are always handled. This flag can be changed by the user;
- bit 3 is the decimal flag. If this bit is set, values are not represented as normal hexadecimal numbers, but as BCD digits (so a byte value of 0x50 is interpreted as decimal 50 instead of decimal 80). The only allowable values in this mode are decimal 00 through 99. This flag can be changed by the user;
- bit 4 is the break flag. This bit is set if an interrupt has occurred because of a BRK instruction, rather than a hardware interrupt;
- bit 5 is unused;

- bit 6 is the overflow flag. This bit is set or reset by an instruction if the sign bit of its result changed (results more than decimal +127 or less than decimal -128). It can be reset (not set) by the user;
- bit 7 is the negative flag. This bit is set or reset depending on the result being negative or not.

3.2 Memory

Although the 6502 is an 8 bit processor, the PC register is 16 bits. This implies that 2^{16} bytes of memory can be addressed. Indeed, through the use of a 16 bit address bus and a 16 bit program counter, 65536 bytes of memory can be addressed. This memory is divided in 256 pages (0x00 through 0xFF) of 256 bytes (0x00 through 0xFF). The usual notation is (HI:LO), where HI indicates the high byte of the address (the memory page), and LO the low byte of the address (the byte on the indicated memory page).

Some of these pages are freely available for the programmer, others are used for special purposes:

- page 00 is the so-called "scratchpad". This page is freely available for the program. The 6502 has a series of addressing modes operating especially on this zero page, and it is therefore quite useful for data that needs to be referred to often;
- page 01 is the stack space. The stack pointer (S) refers to an address on this page;
- pages 02 through 3F are freely available as RAM;
- pages 40 through 7F are used for I/O;
- page FF, bytes FA through FF, are used for program counter initialisation. Bytes FA and FB contain the low and high address bytes of the interrupt handler routine for non-maskable interrupts, bytes FC and FD contain the low and high address bytes of the initial program counter at startup, and bytes FE and FF contain the low and high address byte of the interrupt handler routine for maskable interrupts.

3.3 Stack

The 6502 has a 256-byte stack space, starting at address (01:FF) and growing downwards to (01:00). The stack pointer (S) points to the first unused byte of the stack, and is thus initialized at 0xFF (the high byte, 0x01, is implied).

After a byte is written to the stack (pushed), the stack pointer is decreased, and when a byte is read from the stack (pulled), the stack pointer is increased to point to the address just read from (which is then the new byte to write to).

Take care that S only stores the lower byte of the stack address. If a byte is pushed while the stack pointer points to (01:00), the next address will be (01:FF) and you will risk overwriting previous data. Similarly for pulling when the contents of S are 0xFF.

3.4 Operations

Table 1 lists all instructions available on the 6502. The cell corresponding to an implemented instruction contains its opcode and —if appropriate —its addressing mode. The addressing modes are abbreviated, see section 3.5 for a list. An empty cell denotes an illegal instruction.

Table 2 contains a listing of all operations, their opcodes and some properties of each operation. Table 3 contains a listing of all operations and the flags in the status register that are affected by them. The instructions are described in further detail below.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	BRK	ORA inX				ORA zp	ASL zp		PHP	ORA imm	ASL acc			ORA abs	ASL abs	
1x	BPL rel	ORA inY				ORA zpX	ASL zpX		CLC	ORA abY				ORA abX	ASL abX	
2x	JSR abs	AND inX			BIT zp	AND zp	ROL zp		PLP	AND imm	ROL acc		BIT abs	AND abs	ROL abs	
3x	BMI rel	AND inY				AND zpX	ROL zpX		SEC	AND abY				AND abX	ROL abX	
4x	RTI	EOR inX				EOR zp	LSR zp		PHA	EOR imm	LSR acc		JMP abs	EOR abs	LSR abs	
5x	BVC rel	EOR inY				EOR zpX	LSR zpX		CLI	EOR abY				EOR abX	LSR abX	
6x	RTS	ADC inX				ADC zp	ROR zp		PLA	ADC imm	ROR acc		JMP ind	ADC abs	ROR abs	
7x	BVS rel	ADC inY				ADC zpX	ROR zpX		SEI	ADC abY				ADC abX	ROR abX	
8x		STA inX			STA inY	STA zp	STX zp		DEY		TXZ		STY abs	STA abs	STX abs	
9x	BCC rel	STA inY			STY zpX	STA zpX	STX zpY		TYA	STA abY	TXS			STA abX		
Ax	LDY imm	LDA inX	LDX imm		LDY zp	LDA zp	LDX zp		TAY	LDA imm	TAX		LDY abs	LDA abs	LDX abs	
Bx	BCS rel	LDA inY			LDY zpX	LDA zpX	LDX zpY		CLV	LDA abY	TSX		LDY abX	LSA abX	LDX abY	
Cx	CPY imm	CMP inX			CPY zp	CMP zp	DEC zp		INY	CMP imm	DEX		CPY abs	ZMP abs	DEC abs	
Dx	BNE rel	CMP inY				CMP zpX	DEC zpX		CLD	CMP abY		STP		CMP abX	DEC abX	
Ex	CPX imm	SBC inX			CPX zp	SBC zp	INC zp		INX	SBC imm	NOP		CPX abs	SBC abs	INC abs	
Fx	BEC rel	SBC inY				SBC zpX	INC zpX		SED	SBC abY				SBC adX	INC abX	

Table 1: Instruction matrix with opcodes

Instr	Addressing mode												
	none	abs	abX	abY	acc	imm	ind	inX	inY	rel	zp	zpX	zpY
ADC *		0x6D	0x7D	0x79		0x69		0x61	0x71		0x65	0x75	
	Add memory to accumulator with carry												
AND *		0x2D	0x3D	0x39		0x29		0x21	0x31		0x25	0x35	
	“AND” memory with accumulator												
ASL *		0x0E	0x1E		0x0A						0x06	0x16	
	Arithmetic shift one bit left												
BCC										0x90			
	Branch on carry clear												
BCS										0xB0			
	Branch on carry set												
BEQ										0xF0			
	Branch on equal												
BIT		0x2C									0x24		
	Bit test												
BMI										0x30			
	Branch on result minus												
BNE										0xD0			
	Branch on not equal												
BPL										0x10			
	Branch on result plus												
BRK	0x00												
	Break instruction												
BVC										0x50			
	Branch on overflow clear												
BVS										0x70			
	Branch on overflow set												
CLC *	0x18												
	Clear carry flag												
CLD	0xD8												
	Clear decimal mode												
CLI	0x58												
	Clear interrupt disable bit												
CLV *	0xB8												
	Clear overflow flag												
CMP		0xCD	0xDD	0xD9		0xC9		0xC1	0xD1		0xC5	0xD5	
	Compare memory and accumulator												
CPX		0xEC				0xE0					0xE4		
	Compare memory and X register												
CPY		0xCC				0xC0					0xC4		
	Compare memory and Y register												
DEC *		0xCE	0xDE								0xC6	0xD6	
	Decrement memory by one												
DEX *	0xCA												
	Decrement X by one												
DEY *	0x88												
	Decrement Y by one												
EOR *		0x4D	0x5D	0x59		0x49		0x41	0x51		0x45	0x55	
	“Exclusive OR” memory with accumulator												

Instr	Addressing mode												
	none	abs	abX	abY	acc	imm	ind	inX	inY	rel	zp	zpX	zpY
INC *		0xEE	0xFE								0xE6	0xF6	
	Increment memory by one												
INX *	0xE8												
	Increment X register by one												
INY *	0xC8												
	Increment Y register by one												
JMP		0x4C					0x6C						
	Jump to new location												
JSR		0x20											
	Jump to subroutine												
LDA *		0xAD	0xBD	0xB9		0xA9		0xA1	0xB1		0xA5	0xB5	
	Load accumulator with memory												
LDX *		0xAE		0xBE		0xA2					0xA6		0xB6
	Load X register with memory												
LDY *		0xAC	0xBC			0xA0					0xA4	0xB4	
	Load Y register with memory												
LSR *		0x4E	0x5E		0x4A						0x46	0x56	
	Logical shift one bit right												
NOP *	0xEA												
	No operation												
ORA *		0x0D	0x1D	0x19		0x09		0x01	0x11		0x05	0x15	
	“OR” memory with accumulator												
PHA *	0x48												
	Push accumulator on stack												
PHP *	0x08												
	Push processor status on stack												
PLA *	0x68												
	Pull accumulator from stack												
PLP *	0x28												
	Pull processor status from stack												
ROL *		0x2E	0x3E		0x2A						0x26	0x36	
	Rotate one bit left												
ROR *		0x6E	0x7E		0x6A						0x66	0x76	
	Rotate one bit right												
RTI	0x40												
	Return from interrupt												
RTS	0x60												
	Return from subroutine												
SBC *		0xED	0xFD	0xF9		0xE9		0xE1	0xF1		0xE5	0xF5	
	Subtract memory from accumulator with borrow												
SEC *	0x38												
	Set carry flag												
SED	0xF8												
	Set decimal mode												
SEI	0x78												
	Set interrupt disable status												
STA *		0x8D	0x9D	0x99				0x81	0x91		0x85	0x95	
	Store accumulator in memory												

Instr	Addressing mode												
	none	abs	abX	abY	acc	imm	ind	inX	inY	rel	zp	zpX	zpY
STP *	0xDB												
	Stop the processor												
STX *		0x8E									0x86		0x96
	Store the X register in memory												
STY *		0x8C									0x84	0x94	
	Store the Y register in memory												
TAX *	0xAA												
	Transfer accumulator to X register												
TAY *	0xA8												
	Transfer accumulator to Y register												
TSX *	0xBA												
	Transfer stack pointer to X register												
TXA *	0x8A												
	Transfer X register to accumulator												
TXS *	0x9A												
	Transfer X register to stack pointer												
TYA *	0x98												
	Transfer Y register to accumulator												

Table 2: Instruction matrix with opcodes per addressing mode

Instr	Affected flags						
	C	Z	I	D	B	O	N
ADC *	×	×				×	×
AND *		×					×
ASL *		×					×
BCC							
BCS							
BEQ							
BIT		×				×	×
BMI							
BNE							
BPL							
BRK							
BVC							
BVS							
CLC *	×						
CLD				×			
CLI			×				
CLV *						×	
CMP	×	×					×
CPX	×	×					×
CPY	×	×					×
DEC *		×					×
DEX *		×					×
DEY *		×					×
EOR *		×					×
INC *		×					×
INX *		×					×
INY *		×					×
JMP							

Instr	Affected flags						
	C	Z	I	D	B	O	N
JSR							
LDA *		×					×
LDX *		×					×
LDY *		×					×
LSR *	×	×					×
NOP *							
ORA *		×					×
PHA *							
PHP *							
PLA *							
PLP *							
ROL *	×	×					×
ROR *	×	×					×
RTI							
RTS							
SBC *	×	×				×	×
SEC *	×						
SED				×			
SEI			×				
STA *							
STP *							
STX *							
STY *							
TAX *		×					×
TAY *		×					×
TSX *		×					×
TXA *		×					×
TXS *							
TYA *		×					×

Table 3: Instruction matrix with affected flags

The next sections discuss the available instructions. A short description of the instruction's behaviour is given, together with any other effects its execution has. If an instruction is said to "affect" a certain flag, that means that it sets or resets the flag according to the description in section 3.1. For the instructions that are not directly straightforward, examples are given.

ADC (Add memory to accumulator with carry)

The `ADC` instruction adds a value from memory, the current accumulator value and the carry, and stores the result in the accumulator. It affects the carry, overflow, negative, and zero flags.

`ADC` can very well handle negative numbers, using 2's complement notation. In that case, you have to pay attention to the negative and overflow flags, but don't need to worry about the carry flag (which will be set, but is not of interest to the user).

The negative flag is simply set to the value of bit 7 of the result. The overflow flag is a little trickier: it indicates that a carry has occurred from the 7 least significant bits into bit 7 (recall that the least significant bit is bit 0). This allows you to check whether the range of a signed 8-bits number (consisting of a sign bit and seven value bits) has been exceeded.

Table 4: Examples for ADC interpreted as unsigned numbers

accumulator	34	00100010	Normal addition with carry. There is no carry in the result, so the answer is correct as it is.
memory	28	00011100	
carry	1	1	
result	63	00111111	
accumulator	128	10000000	Normal addition without carry. There is a carry in the result, so the real result is 257 instead of 1.
memory	129	10000001	
carry	0	0	
result	1	00000001	

Table 5: Examples for ADC interpreted as signed numbers

accumulator	96	01100000	Signed addition. The result is negative, but the set overflow flag indicates the result should be positive, the real answer is therefore 144.
memory	48	00110000	
carry	0	0	
result	-112	10010000	
accumulator	8	00001000	Signed addition. The result in the accumulator has remained positive, and no overflow has occurred, so this result is correct.
memory	-6	1111010	
carry	0	0	
result	2	00000010	
accumulator	3	00000011	Signed addition. No overflow has occurred, so bit 7 (and the negative flag) indicates a negative result.
memory	-7	11111001	
carry	0	0	
result	-4	11111100	
accumulator	-5	11111011	Signed addition. There is no overflow, so the result is negative.
memory	-9	11110111	
carry	1	1	
result	-13	11110011	
accumulator	-96	10100000	Signed addition. The result looks positive, but the overflow flag indicates that the result should be negative, the real answer is therefore -144.
memory	-48	11010000	
carry	0	0	
result	112	01110000	

AND ("AND" memory with accumulator)

The `AND` instruction performs a bitwise AND operation on a value in memory and the accumulator value. The result is stored in the accumulator. The zero and negative flags are affected depending on the new value in the accumulator.

ASL (Arithmetic shift one bit left)

The `ASL` instruction shifts the value in the accumulator or a memory address one bit to the left. The low bit is set to 0, the bit that is shifted out is stored in the carry flag. The negative and zero flags are affected depending on the new value.

Table 6: Example for ASL

old value	10111001	
new value	01110010	and a carry

BCC (Branch on carry clear)

The `BCC` instruction checks if the carry flag is set. If not, then a branch is taken, otherwise execution is resumed normally. See Section 3.5 for more details about how this branch is performed.

BCS (Branch on carry set)

The `BCS` instruction checks if the carry flag is set. If so, then a branch is taken, otherwise execution is resumed normally. See Section 3.5 for more details about how this branch is performed.

BEQ (Branch on equal)

The `BEQ` instruction checks if the zero flag is set. If so, then a branch is taken, otherwise execution is resumed normally. See Section 3.5 for more details about how this branch is performed.

BIT (Bit test)

The `BIT` instruction performs an "AND" on a memory value and the value in the accumulator without affecting either, and sets the zero flag according to the result of the "AND" operation. Furthermore, the negative flag is set equal to bit 7 of the memory value and the overflow flag is set equal to bit 6 of the memory value tested.

BMI (Branch on result minus)

The `BMI` instruction checks if the negative flag is set. If so, then a branch is taken, otherwise execution is resumed normally. See Section 3.5 for more details about how this branch is performed.

BNE (Branch on not equal)

The `BNE` instruction checks if the zero flag is set. If not, then a branch is taken, otherwise execution is resumed normally. See Section 3.5 for more details about how this branch is performed.

BPL (Branch on result plus)

The `BPL` instruction checks if the negative flag is set. If not, then a branch is taken, otherwise execution is resumed normally. See Section 3.5 for more details about how this branch is performed.

BRK (Break instruction)

The `BRK` instruction interrupts the program to run an interrupt sequence subroutine, but only if the disable interrupt flag is zero (this is a maskable interrupt). This is done by storing various information on the stack. First the memory location 2 bytes after the `BRK` instruction (that is, skipping one byte with respect to the instruction) is put onto the stack, lower byte first. Then the current processor status (the contents of the `P` register) is pushed, and finally an extra status byte which only contains a 1 at the position of the `B` (break) flag in the `P` register. The latter byte can be used in the interrupt handling routine to determine whether it has been called by a maskable interrupt or by a `BRK` instruction. Then, the new program counter value is determined by reading addresses (`FF:FE`) (low byte) and (`FF:FF`) (high byte), which contain the address of the interrupt handler.

In table 7 you can see a small snippet of code and the resulting stack directly after the `BRK` instruction has been executed. Note that when the interrupt returns, the `STP` instruction will be

skipped and the JMP instruction will be the first instruction to be executed after the interrupt has been returned.

Table 7: Example for BRK

Address	Byte	Stack after BRK
00:60	BRK	00010000
00:61	STP	P
00:62	JMP	0x00
00:63	[ADL]	0x62
00:64	[ADH]	(rest of the stack)

BVC (Branch on overflow clear)

The BVC instruction checks if the overflow flag is set. If not, then a branch is taken, otherwise execution is resumed normally. See Section 3.5 for more details about how this branch is performed.

BVS (Branch on overflow set)

The BVS instruction checks if the overflow flag is set. If so, then a branch is taken, otherwise execution is resumed normally. See Section 3.5 for more details about how this branch is performed.

CLC (Clear carry flag)

The CLC instruction sets the carry flag of the P register to 0.

CLD (Clear decimal mode)

The CLD instruction sets the decimal mode flag of the P register to 0.

CLI (Clear interrupt disable bit)

The CLI instruction sets the interrupt disable flag of the P register to 0.

CLV (Clear overflow flag)

The CLV instruction sets the overflow flag of the P register to 0.

CMP (Compare memory and accumulator)

The CMP instruction compares a value in memory with the value in the accumulator by subtracting the memory value from the accumulator, without affecting either of them. The zero and negative flags are affected as usual. The carry flag is set if the memory value is less than or equal to the accumulator value, and reset if it is larger.

CPX (Compare memory and X register)

The CPX instruction subtracts a value in memory from the value in the X register, without storing the result. If the memory value is larger than the value in the X register, the carry flag is set to 0, otherwise it is set to 1. The negative flag is set if the result has bit 7 set (otherwise it is reset), and if the result is zero the zero flag is set (otherwise it is reset).

CPY (Compare memory and Y register)

The `CPY` instruction subtracts a value in memory from the value in the `Y` register, without storing the result. If the memory value is larger than the value in the `Y` register, the carry flag is set to 0, otherwise it is set to 1. The negative flag is set if the result has bit 7 set (otherwise it is reset), and if the result is zero the zero flag is set (otherwise it is reset).

DEC (Decrement memory by one)

The `DEC` instruction subtracts 1 from the value at the effective memory address. The zero and negative flags are affected depending on the new value.

DEX (Decrement X by one)

The `DEX` instruction subtracts one from the value in the `X` register. The negative and zero flags are affected depending on the new `X` register value.

DEY (Decrement Y by one)

The `DEY` instruction subtracts one from the value in the `Y` register. The negative and zero flags are affected depending on the new `Y` register value.

EOR ("Exclusive OR" memory with accumulator)

The `EOR` instruction performs a bitwise `XOR` (exclusive-OR) operation on the value in memory and the accumulator value. The result is stored in the accumulator. The zero and negative flags are affected depending on the new value in the accumulator.

INC (Increment memory by one)

The `INC` instruction adds 1 to the value at the effective memory address. The zero and negative flags are affected depending on the new value.

INX (Increment X register by one)

The `INX` instruction adds 1 to the value in the `X` register. The negative and zero flags are affected depending on the new `X` register value.

INY (Increment Y register by one)

The `INY` instruction adds 1 to the value in the `Y` register. The negative and zero flags are affected depending on the new `Y` register value.

JMP (Jump to new location)

The `JMP` instruction performs a jump by reading a new value for the program counter.

JSR (Jump to subroutine)

The `JSR` instruction calls a subroutine. The address of the second byte after the address the `JSR` is at (that is, skipping one byte with respect to the instruction byte), is stored on the stack, lower byte first. The new program counter is read from the two bytes following the instruction, in (low, high) order. For clarity: `JSR ADL ADH` will store a pointer to the byte the `ADH` is at, on the stack, and will jump to (`ADH:ADL`).

LDA (Load accumulator with memory)

The `LDA` instruction reads a byte from memory and stores it in the accumulator. It affects the zero and negative flags.

LDX (Load X register with memory)

The `LDX` instruction stores a value from memory in the `X` register. The zero and negative flags are affected depending on the stored value.

LDY (Load Y register with memory)

The `LDY` instruction stores a value from memory in the `Y` register. The zero and negative flags are affected depending on the stored value.

LSR (Logical shift one bit right)

The `LSR` instruction shifts the value in the accumulator or a memory address one bit to the right. The high bit is set to 0, the bit that is shifted out is stored in the carry flag. The negative flag is reset, the zero flag is affected depending on the new value.

Table 8: Example for LSR

old value	10010011	
new value	01001001	and a carry

NOP (No operation)

The `NOP` instruction does nothing.

ORA ("OR" memory with accumulator)

The `ORA` instruction performs a bitwise `OR` operation on the value in memory and the accumulator value. The result is stored in the accumulator. The zero and negative flags are affected depending on the new value in the accumulator.

PHA (Push accumulator on stack)

The `PHA` instruction stores the value in the accumulator to the current stack location, and decrements the stack pointer.

PHP (Push processor status on stack)

The `PHP` instruction stores the value in the P register to the current stack location, and decrements the stack pointer.

PLA (Pull accumulator from stack)

The `PLA` instruction increments the stack pointer and stores the value located at the new stack address in the accumulator.

PLP (Pull processor status from stack)

The `PLP` instruction increments the stack pointer and stores the value located at the new stack address in the `P` register.

ROL (Rotate one bit left)

The `ROL` instruction shifts the value in the accumulator or a memory address one bit to the left. The low bit is set to the value of the carry flag, and the bit that is shifted out is stored in the carry flag. The negative and zero flags are affected depending on the new value.

Table 9: Example for ROL

old value	10101010	without a carry
new value	01010100	and a carry

ROR (Rotate one bit right)

The `ROR` instruction shifts the value in the accumulator or a memory address one bit to the right. The high bit is set to the value of the carry flag, and the carry flag is set to the old value of the low bit. The negative and zero flags are affected depending on the new value.

Table 10: Example for ROR

old value	10101010	with a carry
new value	11010101	without a carry

RTI (Return from interrupt)

The `RTI` instruction returns program control to normal execution when interrupt handling is complete, by restoring the processor status and the program counter from the stack. Note that this is not symmetrical with `BRK`. Below is some code that can be used as a handler for interrupts, including `BRK` interrupts.

```
interrupt_handler:
    ; Detect a BRK instruction first
    PLA                 ; Load status register
    PHA                 ; Restore onto stack
    AND #$10            ; Isolate B flag
    BNE break_handler   ; If the B flag is set, this was a BRK instruction
                        ; so handle it

    ; Normal interrupt handling goes here
    RTI

break_handler:
    ; Pull the extra byte from the stack
    PLP                 ; Pull one byte from the stack
    ; BRK handling goes here
    RTI
```

RTS (Return from subroutine)

The `RTS` instruction returns from a subroutine call by restoring the program counter from the stack so that the instruction following the invoking `JSR` is executed next.

SBC (Subtract memory from accumulator with borrow)

The `SBC` instruction subtracts the memory and borrow from the accumulator, and stores the result in the accumulator. Borrow is defined as the logical inverse of carry, so if the carry flag is set, there is no borrow to be subtracted.

The 6502 adds the carry to the bit-inverted value read from memory, to gain a proper 2's complement representation of the negative number to be subtracted. Then the negative value is added to the accumulator value. Check for yourself that doing it this way does give the correct value!

The carry flag is affected as borrow, so the carry flag is set when no borrow was needed, and vice versa. Again, check for yourself that a carry means that no borrow was needed. The overflow, negative, and zero flags are affected as usually.

Table 11: Examples for SBC

memory	16	00010000
completent		11101111
carry	0	0
neg. memory	-17	11101111
accumulator	32	00100000
neg. memory	-17	11110000
result	15	00001111

After the addition, the carry flag is set, so no borrow was required. The result of 16 is therefore correct.

memory	6	00000110
complement		11111001
carry	1	1
neg. memory	-6	11111010
accumulator	5	00000101
neg. memory	-6	11111010
result	255	11111111

After the addition, the carry flag is not set, so a borrow was required. The result should be -1 instead of 255.

SEC (Set carry flag)

The `SEC` instruction sets the carry flag of the `P` register to 1.

SED (Set decimal mode)

The `SED` instruction sets the decimal mode flag of the `P` register to 1.

SEI (Set interrupt disable status)

The `SEI` instruction sets the interrupt disable flag of the `P` register to 1.

STA (Store accumulator in memory)

The `STA` instruction stores the value in the accumulator at the effective memory address given as its operand.

STP (Stop processor)

The `STP` instruction halts the processor. *This instruction was not available on the original 6502, but is rather an extension on for example the W65C02S. We have included this instruction here to give you a nice and easy way to terminate execution of the FDE cycle.*

STX (Store the X register in memory)

The `STX` instruction stores the value in the `X` register at the effective memory address given as its operand.

STY (Store the Y register in memory)

The `STY` instruction stores the value in the `Y` register at the effective memory address given as its operand.

TAX (Transfer accumulator to X register)

The `TAX` instruction copies the value in the accumulator to the `X` register. The negative and zero flags are affected depending on the value copied.

TAY (Transfer accumulator to Y register)

The `TAY` instruction copies the value in the accumulator to the `Y` register. The negative and zero flags are affected depending on the value copied.

TSX (Transfer stack pointer to X register)

The `TSX` instruction copies the value of the stack pointer register to the `X` register. The negative and zero flags are affected depending on the value copied.

TXA (Transfer X register to accumulator)

The `TXA` instruction copies the value in the `X` register to the accumulator. The negative and zero flags are affected depending on the value copied.

TXS (Transfer X register to stack pointer)

The `TXS` instruction copies the value of the `X` register to the stack pointer register. No flags are affected.

TYA (Transfer Y register to accumulator)

The `TYA` instruction copies the value in the `Y` register to the accumulator. The negative and zero flags are affected depending on the value.

3.5 Addressing modes

The abbreviations used in table 1 and in the headings of the descriptions below, are:

abs	absolute
abX	absolute, X
abY	absolute, Y
acc	accumulator
imm	immediate
ind	indirect
inX	indirect X (“(zero page, X)”)
inY	indirect Y (“(zero page), Y”)
rel	relative
zp	zero page
zpX	zero page, X
zpY	zero page, Y

Absolute (abs)

Absolute addressing is one of the easiest addressing modes. The instruction is followed by two bytes, `ADL ADH`, which together form the address at which the operand can be found.

Table 12: Absolute addressing

Instruction:	`opcode ADL ADH`
Effective address:	(`ADH : ADL`)

Example: `opcode 0x42 0x09`
The operand is read from (`09:42`).

Absolute, X (abX)

The Absolute, X mode (or "Absolute Indexed with X") extends the Absolute addressing mode by adding the contents of the X register to the address given with the instruction.

Table 13: Absolute, X addressing

Instruction:	`opcode ADL ADH`
Effective address:	(`ADH : ADL`) + `X`

Example: `opcode 0x42 0x09`
Suppose that the `X` register contains `0x30`, then the operand is read from (`09:42`) + `30`, which is (`09:72`).

Absolute, Y (abY)

The Absolute, Y mode (or "Absolute Indexed with Y") is similar to the Absolute, X mode, with the difference that the `Y` register is used instead of the `X` register.

Table 14: Absolute, Y addressing

Instruction:	`opcode ADL ADH`
Effective address:	(`ADH : ADL`) + `Y`

Example: `opcode 0x42 0x09`
Suppose that the `Y` register contains `0xD0`, then the operand is read from (`09:42`) + `D0`, which is (`0A:22`) (note the carry).

Accumulator (acc)

With Accumulator addressing, the operand is read from the accumulator register.

Table 15: Accumulator addressing

Instruction:	`opcode`
Operand:	`accumulator`

Immediate (imm)

With Immediate addressing, the operand is located at the second byte of the instruction.

Table 16: Immediate addressing

Instruction:	`opcode operand`
Operand:	`operand`

Table 17: Indirect addressing

Instruction:	`opcode ADL ADH`
Indirect address (`IA`):	`(ADH : ADL)`
Effective address:	`(byte at (IA + 1) : byte at IA)`

Indirect (ind)

With Indirect addressing mode, the opcode is followed by two bytes that form an indirect address. This address, and the byte following it, contain the effective address, lower byte first.
Example: `opcode 0x42 0x09`

The indirect address is (`09:42`).

Suppose that the byte at (`09:42`) is `0x18` and that the byte at (`09:43`) is `0x27`, then the effective address is (`27:18`).

Indirect X (inX)

In Indirect X (or "Zero Page Indexed Indirect") mode, the instruction is followed by one byte. To this byte, the `X` register is added, discarding a possible carry. This is the lower byte of an address on page zero which contains the effective address.

Table 18: Indirect X addressing

Instruction:	`opcode ADL`
Base address:	`(0 : ADL)`
Indirect address (`IA`):	`(0 : (ADL + X))`
Effective address:	`(byte at (IA + 1) : byte at IA)`

Example: `opcode 0x42`

The base address is (`00:42`).

Suppose that the `X` register contains `0x20`, then the indirect address is (`00:62`). Suppose that the byte at (`00:62`) is `0x18` and that the byte at (`00:63`) is `0x27`, then the effective address is (`27:18`).

Indirect Y (inY)

In Indirect Y (or "Zero Page Indirect Indexed") mode, the instruction is followed by the lower byte of a zero page address. The next byte on the zero page is the higher byte of the base address. To this base address, the contents of the `Y` register are added to form the effective address.

Table 19: Indirect Y addressing

Instruction:	`opcode ADL`
Indirect address (`IA`):	`(0 : ADL)`
Base address (`BA`):	`(byte at (IA + 1) : byte at IA)`
Effective address:	`BA + Y`

Example: `opcode 0x42`

The indirect address is (`00:42`).

Suppose that the byte at (00:42) is 0x18 and that the byte at (00:43) is 0x27, then the base address is (27:18). Suppose that the Y register contains 0xF0, then the effective address is (28:08).

Relative (r)

Used with the branching instructions, the Relative addressing mode takes an offset from the (signed) byte following the instruction, and adds this to the program counter if the test on which the branch depends, results in true.

Table 20: Relative addressing

Instruction:	`opcode offset`
Effective address:	`PC + offset`

Note that the offset is relative to the address following the offset, so if the offset is located at e.g. (04:50), its value will be added to (04:51), the address of the first instruction after it. The 6502 reads the offset, increments the program counter and then performs the test.

(This is not too hard to remember, if you remind yourself that an offset of 0 should just execute the next instruction, which is located at the byte following the offset value.)

Zero Page (zp)

With Zero Page addressing, the byte following the instruction is the address of the operand in page zero.

Table 21: Zero Page addressing

Instruction:	`opcode ADL`
Effective address:	`(0 : ADL)`

Example: `opcode 0x42`

The operand is read from (00:42).

Zero Page, X (zpX)

With Zero Page, X (Zero Page Indexed, X) addressing, the byte following the instruction and the contents of the X register are added to form the address of the operand in page zero.

Table 22: Zero Page, X addressing

Instruction:	`opcode ADL`
Effective address:	`(0 : (ADL + X))`

Example: `opcode 0x42`

Suppose that the X register contains 0x68, then the operand is read from (00:AA).

Zero Page, Y (zpY)

With Zero Page, Y (Zero Page Indexed, Y) addressing, the byte following the instruction and the contents of the Y register are added to form the address of the operand in page zero.

Table 23: Zero Page, Y addressing

Instruction:	`opcode ADL`
Effective address:	`(0 : (ADL + Y))`

Example: `opcode 0x42`

Suppose that the `Y` register contains `0xFF`, then the operand is read from (`00:41`).

A Administrativia

A.1 Deadlines

Although there are no hard deadlines apart from the date for the final delivery of your source and report, we recommend you stick to the schedule, distributed separately from this manual.

A.2 Grading

The final grade (G_{final}) of an individual student is determined from the overall group product and process grades. If needed, this is corrected according to the outcome of the peer reviews, or if the assistants see a defendable need to adjust the mark:

$$G_{final} = 0.7 \times G_{product} + 0.3 \times G_{process} + G_{individual}$$

The product grade, $G_{product}$, is determined on a scale of 1 to 10. Factors that contribute to this grade are:

- (15%) specifications;
- (25%) code correctness;
- (25%) code completeness;
- (15%) coding style;
- (20%) final report.

The process grade, $G_{process}$, is determined on a scale of 1 to 10. This grade includes:

- (20%) agendas and minutes;
- (30%) task division;
- (25%) group collaboration and conflict resolution;
- (25%) time management.

The individual grade, $G_{individual}$, is determined by the outcome of the two peer reviews, and is typically in the range of -1 to +1.

A.3 Report

Your report should at least contain the following parts:

- introduction;
- process and project overview (what did you do and when, which problems did you encounter and how did you solve them, . . .);
- noteworthy details about your implementation;
- reflection on the project (what did you learn, what did you find (not) useful and why, . . .);
- suggestions for next year.

Furthermore, do not forget basic parts like a title page or a table of contents.

Also take note of the possible extra requirements imposed on your report by your "Schriftelijk Rapporteren" teacher.

B References

Most of the information in this document is taken from reference documentation available online via `http://www.6502.org/documents/datasheets/`. Especially Synertek's *SY6500/MCS6500 Microcomputer Family Programming Manual* contains much in-depth information. Note that the various 650x implementations differ on some details, so not all information in every document available there is applicable to this project.

If you find any other sources of useful information with which the manual could be extended or improved, please tell the project assistants about it.

C Acknowledgements

We are thankful to all the people who gave us feedback: Henk Sips and Jan David Mol (former project staff), Denis de Leeuw Duarte and Jonne Zutt (former practical staff), Thomas Schaap (general proofreading and rewriting), Boaz Pat-El (rewriting), and any others we have forgotten.