



PROGRAMMING, STEP BY STEP

D. M. de Boer

Since RB's KIM campaign in August of last year, the number of KIM owners has grown considerably. Readers' reactions show that some of the new KIM owners are already getting the hang of programming quite well. Another group, however, is clearly lagging behind: the people who cannot find the 'beginning'. That is very understandable when you consider that, although the computer is built from electronics, programming is completely new territory for many. In this article RB will therefore try to help this group on their way.

The KIM keyboard

To be able to program the KIM, we must be able to put particular numbers into particular memory locations. Each number has a given meaning for the microcomputer. By putting the right combination of numbers into consecutive memory locations, we can make the microprocessor perform a particular action. To put these numbers into memory, the KIM has a small keypad with digits (0 ... F) and control keys (AD, DA, PC, +, GO, ST and RS). When we press AD (address) once, we can select an address (= a memory location). This address is visible on the 4 left-hand displays. The 2 right-hand displays show the contents of this address. If we want to change the contents of the address we must first press 'DA' (data). From that moment on, digits typed are put into the selected memory location. The display always shows the contents of memory. So there is no need to press '+' or any other key to 'clock in' the information. Every digit pressed automatically ends up in the selected memory location. When typing in a program it would be rather cumbersome to keep pressing AD, entering the address, pressing DA, entering the data, pressing AD again, and so on. That is why the '+' key exists. With this key the address can be incremented by 1, so that a program can be entered easily.

The 'ST' key, a dangerous key

The 'ST' key (STOP key) serves to stop a program that has been started. At the moment the ST key is pressed, the microprocessor finishes the instruction it is working on and then stops the program. The display now shows the address of the next instruction to be executed. Pressing 'GO' continues the program as if nothing had happened. But why then is this key dangerous? Well, pressing 'ST' generates an interrupt, and the microprocessor will jump to the interrupt routine. We can define the location of this routine ourselves, so that it can also jump to programs we have written. If this address has not been defined, the microprocessor will, after 'ST' is pressed, start at some

random location in memory. That may result in the entire memory being wiped. Therefore: make a habit of entering the interrupt vector immediately after switching on the KIM (press 'RS' first to make the display light up). For the stop function (and single step) we enter:

Address	contents
17FA	00
17FB	1C

Once this has been done, you can press 'ST' as often as you like.

The 'GO', 'RS' and 'PC' keys

There are now 3 keys left that have not been discussed. The easiest of the 3 is 'GO'. With this key you can start a program you have written. 'GO' starts the microprocessor from the address shown on the display at that moment. So you must always enter the start address first. (That is, the address where the first instruction is.) The 'RS' key (reset) unconditionally jumps to the monitor program. (The monitor program drives the display and decodes the signals from the keypad.) The function of 'RS' is a bit like that of 'ST'. There is, however, an important difference. With 'ST' a program can be interrupted, and the display shows at which address the microprocessor had got to. With 'RS' a program is also interrupted, but the display now shows the last address that was made visible (usually the start address). We cannot continue the program, because we do not know where the microprocessor had got to. Moreover, certain data from the internal registers of the microprocessor have not been preserved. We can now only restart the program from the beginning. The 'RS' key does, however, have some essential advantages.

For example, we can start the monitor program when the interrupt vector has not yet been defined (e.g. when the KIM has just been switched on). With program errors it can happen that a non-existent instruction is read. The 'ST' key often refuses to work then, because the microprocessor first wants to finish the instruction it is working on. With some non-existent instructions (e.g. 42) the microprocessor never finishes, so 'ST' does not work. Here too the 'RS' key offers a way out. Among more advanced programmers it can happen that a different interrupt vector is entered, giving 'ST' a different function. In that case too we depend on 'RS'. In addition, pressing 'RS' sets the stack pointer correctly and the contents of the direction registers of the inputs/outputs become '00'. We need not worry about these points yet; for the time being we will not use these functions.

The 'PC' key works in combination with 'ST' or single step. (We assume that the correct interrupt vector has been entered, '00' at 17FA and '1C' at 17FB.) We have already said that pressing 'ST' puts on the display the address of the instruction where the microprocessor stopped. We can now enter other addresses to check certain registers or memory locations. If we want the original address back, we simply press 'PC' (program counter), and the correct address appears on the display again. Sometimes it

is handy to be able to recall a particular address quickly. In that case we can also use 'PC'. If, for example, we want address 0123 to appear when 'PC' is pressed, we enter this as follows:

address	contents
00EF	23
00F0	01

As long as we do not use 'ST' and single step (SST), this address will appear every time 'PC' is pressed.

The SST slide switch

When the SST slide switch is 'on', we work in single step. Each press of 'GO' now executes only one instruction. The interrupt vector must have been entered, though (see the section on the 'ST' key). After each instruction we can check how the contents of the various registers and memory locations have changed. Pressing 'PC' brings the address of the next instruction back onto the display, and pressing 'GO' executes that instruction too. (Single step was discussed at great length in RB of February 1978.) Single step is for checking programs and, if the program does not work, for finding the error.

The first instructions

When you look at the large number of instructions with their associated addressing modes, it will certainly seem very confusing at first. People often do not understand which addressing mode should be used. That is why we start with a limited number of instructions, using only 'absolute addressing'. For now the instructions are just:

Instruction	Code
LDA	AD
STA	8D
JMP	4C

The first instruction lets you take a number from any location in memory and put it in register A. (Register A is inside the microprocessor.) The second instruction does the opposite: a number in register A can be put into any location in memory. The third instruction (JMP) is an unconditional jump. This means that the microprocessor does not fetch the next instruction from the next memory location, but from the address given after the JMP ('jump') instruction. So the microprocessor jumps to another part of memory.

With these 3 instructions we can write our first little program. We want a program that puts a number located at address 0000 into address 0001. The program looks like this:

0200 AD 00 00	LDA, abs \$0000
0203 8D 01 00	STA, abs \$0001
0206 4C 22 1C	JMP, abs RST

At address 0200 we see the instruction 'AD', which stands for 'LDA'. We can then ask ourselves: 'Which number has to go into register A?' Directly after the number 'AD' (the

op-code) comes the operand, spread over two memory locations (bytes). In this case it is '0000'. So the microprocessor will put the number located at address '0000' into register 'A'. The second instruction does the reverse: the contents of register A are now stored in memory. Which memory location this is, is again given by the next 2 memory locations. (In the first location the last digits of the address, and in the second location the first digits of the address.) With these two instructions we have already achieved what we wanted: the number at address 0000 has been moved to address 0001. The microprocessor, however, does not know this and stubbornly carries on with the next instruction. Here we find 'JMP'. The microprocessor will now jump to address 1C22 (this address has been given the name RST). Here the monitor program begins, so that the display lights up and we can use the keypad again. When you have typed in the program and start it, it will look as if nothing happens. The program is so short (it executes in 11 μ s) that this is not noticeable. But if you check memory locations 0000 and 0001, you will see that the task has been carried out perfectly.

Another simple example

We will discuss another simple example; now the numbers at addresses 0000 and 0001 must be swapped. If we start the same way as in the previous example (LDA 0000, STA 0001), the contents of memory location 0001 are lost. So we have to add an intermediate step and first put the number at address 0001 somewhere else temporarily. This can be done as follows:

0200 AD 01 00	LDA, abs \$0001
0203 8D 02 00	STA, abs \$0002
0206 AD 00 00	LDA, abs \$0000
0209 8D 01 00	STA, abs \$0001
020C AD 02 00	LDA, abs \$0002
020F 8D 00 00	STA, abs \$0000
0212 4C 22 1C	JMP, abs RST

For this short piece of program we need no fewer than 21 memory locations. This number can be drastically reduced by using 'zero-page addressing'.

Zero page addressing

You may have noticed that the memory locations the program uses all begin with '00' (0000, 0001 and 0002). We also say: all these addresses lie in page 0. The program we wrote runs from address 0200 ... 0214. These addresses therefore lie in page 2. For addresses in page 0 the KIM has a separate addressing mode, namely 'zero page addressing'. This addressing has the same effect as absolute addressing, but can only be used with addresses in page 0. We will now write the same program, but this time using 'zero page addressing'.

0200 A5 01	LDA, Zpage \$01
0202 85 02	STA, Zpage \$02
0204 A5 00	LDA, Zpage \$00

0206 85 01	STA, Zpage \$01
0208 A5 02	LDA, Zpage \$02
020A 85 00	STA, Zpage \$00
020C 4C 22 1C	JMP, abs RST

We see that we now need only 15 memory locations instead of the 21 with absolute addressing in the previous example. The instruction codes are also different (A5 instead of AD, 85 instead of 8D); this is logical, since the microprocessor must know which addressing mode is used. This example shows right away that the program can be shorter if the memory locations it uses lie in page 0. (This short method would not have been possible if we had used, say, addresses 0300, 0301 and 0302.) The program can be shorter still, because the microprocessor contains not only a register A but also a register X. We now use memory location 0002 to store a number temporarily. If we use register X for this, we can again save a few locations.

Register X

Register X, like register A, is a storage location inside the microprocessor. Register X, however, can perform some specific tasks in combination with 'indexed addressing'. That is why we also call it index register X. We will come back to this. For now, 2 new instructions with the addressing modes already discussed.

instruction	abs.	Zpage
LDX	AE	A6
STX	8E	86

If we now do the same example again (swap the numbers at addresses 0000 and 0001), we get:

0200 A6 01	LDX, Zpage \$01
0202 A5 00	LDA, Zpage \$00
0204 85 01	STA, Zpage \$01
0206 86 00	STX, Zpage \$00
0208 4C 22 1C	JMP, abs RST

And we see that only 11 memory locations are now needed, instead of the 21 we needed at first. The time required has also been reduced: working only with register A and 'absolute addressing', this program takes 27 μ s. In the last example it is only 15 μ s. Seemingly irrelevant differences, but bear in mind that with longer programs these differences really do start to matter. (By using the right addressing, more instructions, and thus longer and better programs, fit into the same memory space.)

Names for memory locations

A memory location is often given a name, because a program is then often easier to read. In our example 2 numbers had to be swapped, namely the numbers at address 0000 and address 0001. If we now call these memory locations 'GETAL1' and 'GETAL2' respectively (Dutch for NUMBER1 and NUMBER2), the program looks clearer:

```

0200 A6 01  START      LDX, Zpage GETAL2
0202 A5 00              LDA, Zpage GETAL1
0204 85 01              STA, Zpage GETAL2
0206 86 00              STX, Zpage GETAL1
0208 4C 22 1C          JMP, abs  RST

```

We now see at a glance that first number 2 is put in register X, then number 1 in register A, to be stored straight away at the old location of number 2. Finally number 2, now in register X, is stored at the old location of number 1. (The JMP instruction only serves to indicate that the program has finished.) Particularly in larger programs with many memory locations, names tell us much more than addresses. Memory locations within the program sometimes have a name too. This name is then placed in front of the instruction. In the example above we have given address 0200 the name 'START'. Likewise the monitor program has 'RST' at the same position at address 1C22 (page 16, appendix I, KIM User Manual).

Example 3

In programs it can also happen that a particular memory location and/or register has to be filled with a value that is already known. (In the previous example the numbers to be swapped could be set from outside the program.) So now we want to write a program that puts the values '11' and '05' at address 0000 and address 0001 respectively.

With the instructions discussed so far, the program could look like this:

```

0200 AD 0D 02  START      LDA, abs  ELF
0203 85 00              STA, Zpage GETAL1
0205 AD 0E 02          LDA, abs  VIJF
0208 85 01              STA, Zpage GETAL2
020A 4C 22 1C          JMP, abs  RST
020D 11              ELF
020E 05              VIJF

```

First the number at address 020D is put into register A. The programmer has put 11 here because the program must store this number at address 0000. When you have typed in the program you can start it at address 0200. Seemingly nothing happens, but just check memory locations 0000 and 0001. For this simple task we need no fewer than 15 memory locations. It can be done even shorter, with the third, much-used addressing mode.

Immediate addressing

After absolute and zero page addressing we now come to 'immediate addressing'. With this addressing mode we can fill a register with a fixed value. The program from the previous example now becomes:

```

0200 A9 11  START      LDA, imm. $11
0202 85 00              STA, Zpage GETAL1

```

0204 A9 05	LDA, imm. \$05
0206 85 01	STA, Zpage GETAL2
0208 4C 22 1C	JMP, abs RST

The number of memory locations has been reduced to 11, while the program has become somewhat clearer. The code for 'LDA' is different again, so that the microprocessor knows that 'immediate addressing' is used here. Just to be clear: register A now does not receive the number from memory location \$11, but the number \$11 itself. (\$ means that the number is written in hexadecimal, i.e. in base 16.)

In this article you have become acquainted with the 3 most important addressing modes, namely absolute, zero page and immediate. In the next article we will begin with conditional jumps in the program, and indexed addressing will also be covered.

In the previous part we became acquainted with the three most important addressing modes, namely absolute, zero page and immediate addressing. For clarity, the number of instructions was also limited to 5. In this second part, as announced, we will deal with the conditional jump instructions. 'Indexed addressing' will also be introduced by means of a few examples.

In the previous part we were introduced to the three most important addressing modes, namely absolute, zero page and immediate addressing. For the sake of clarity, the number of instructions was also limited to 5. In this second part, as announced, we will deal with conditional jump instructions. 'Indexed addressing' will also be introduced by means of a few examples.

Brief summary of part 1

- To use the single step or the 'ST' key, the interrupt vector must first be entered ('00' at address 17FA and '1C' at address 17FB).
- The 'ST' key can be used to interrupt a program. The address at which the program is to be continued appears on the display, and can always be recalled by pressing 'PC'.
- Pressing 'GO' starts (or continues) the program from the address shown on the display.
- With 'absolute addressing' we need 3 memory locations per instruction, namely one for the op code and two for the operand.
- If we make sure that the memory locations used by the program lie in page 0 (all addresses beginning with '00'), we can use 'zero page addressing'.
- With 'zero page addressing' only 2 memory locations per instruction are needed, so the same program takes up less memory space.
- To fill registers with already known values we can use 'immediate addressing'. The operand (the second byte of the instruction) then does not give the address of a particular number, but the number itself.
- For now we know the following instructions:

instr.	abs	Zpage	imm
LDA	AD	A5	A9

STA	8D	85	-
LDX	AE	A6	A2
STX	8E	86	-
JMP	4C	-	-

(For the instructions STA, STX and JMP, 'immediate addressing' is by definition not possible.)

- At the end of a program we jump via a 'JMP' to the monitor program, so that the display lights up again.

The status register

An important register in the microprocessor is the status register. With this register we can make the microprocessor perform conditional jumps. Like registers A and X, the status register is 8 bits wide. Of these 8 bits, however, only 7 are used. The status register is not used to store numbers, as registers X and A are; each bit of the status register has its own function. Fig. 1 shows all the internal registers of the 6502 once more. Registers Y, PC and S have not yet been discussed. For the status register the function of each bit is indicated.

But how does the status register actually work? For now we will limit ourselves to bit 1 of the status register, the Z bit. The Z bit is automatically set or reset by the microprocessor when certain instructions are used. On the instruction card, which lists the instructions with their codes, a separate column indicates which instructions affect the bits in the status register. For the limited instruction set we are using for clarity, we now get the following list:

instr.	abs	Z-page	imm	Z-bit
LDA	AD	A5	A9	V
STA	8D	85	-	-
LDX	AE	A6	A2	V
STX	8E	86	-	-
JMP	4C	-	-	-

A dash in the last column means that the Z bit keeps its old value; a V means that the bit becomes '0' or '1' according to the corresponding condition. In our case this means that the Z bit becomes '1' as soon as '00' is put into register X or A, and '0' as soon as any other number is put into X or A.

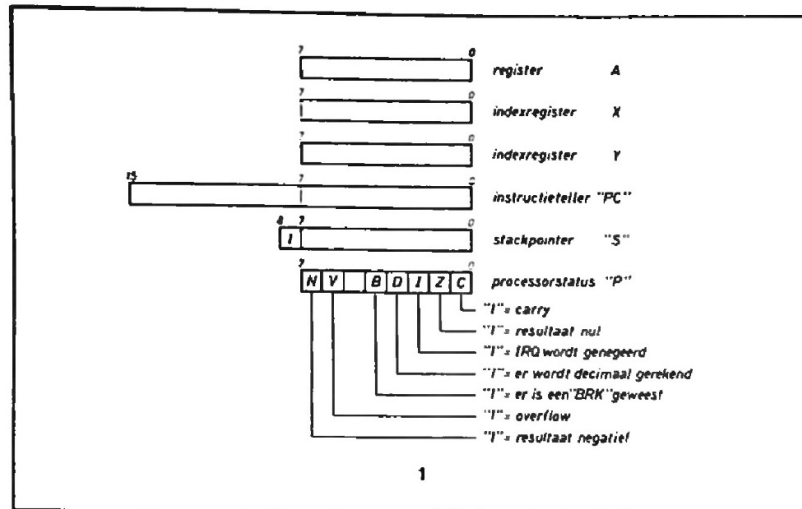


Fig. 1: The internal registers of the 6502. Registers Y, PC and S have not yet been discussed.

Conditional jump instructions

The 'JMP' instruction discussed earlier was always executed. The KIM also has conditional jump instructions, which are or are not executed depending on the state of a particular bit in the status register. For now, again, only the jump instructions relating to the Z bit:

instructie	code	voorwaarde
BNE	D0	Z=0
BEQ	F0	Z=1

If the condition is not met, the instruction is not executed and the microprocessor simply carries on with the next instruction. The conditional jump instructions exist only with 'relative addressing'. So, unlike with 'JMP', we do not give an absolute address for the microprocessor to go to, but a number of places to jump forward or back. We will explain this with an example.

Example

As an example we take a program that fills all memory locations between addresses \$0001 and \$00E0 with '\$EA'. The program looks like this:

0200	A9 EA	LDA, imm	\$EA
0202	85 E0	STA, Zpage	\$E0
0204	CE 03 02	DEC, abs	\$0203
0207	D0 F9	BNE, rel	NEXT
0209	A9 E0	LDA, imm	\$E0
020B	8D 03 02	STA, abs	\$0203
020E	4C 22 1C	JMP, abs	RST

At address \$0204 we see an instruction we have not used before, namely 'DEC' (decrement). This instruction decreases the contents of a particular memory location

by 1. In this example the instruction is written with 'absolute addressing'. The program now runs as follows: the value EA is put via register A at address \$00E0. Next, memory location \$0203 is decreased by 1. As long as this does not result in memory location \$0203 becoming '00', the Z bit in the status register is set to '0' each time. As a result, the conditional jump instruction BNE (condition: Z=0) will be executed each time. In the program we see that we have to jump to memory location 'NEXT'. (How we arrive at F9 we will leave aside for the moment.) At this location (address \$0202) the contents of register A ('\$EA') are again stored in memory, but now at location \$00DF. (After all, the contents of location \$0203 were decreased by 1.) After this the contents of address \$0203 are decreased again.

The whole process repeats until address \$0203 contains the value '\$01'. By then all memory locations between \$01 and \$E0 contain the value '\$EA'. When the contents of \$0203 are now decreased by 1 again, the result of this operation is '00'. As a result, the 'Z' bit in the status register now becomes '1'.

This causes the 'BNE' instruction to be ignored, and we arrive at the last part of the program, from address \$0209. Here we put '\$E0' back at address \$0203 via register A, so that the program is restored.

If we did not do this, we would have to check address \$0203 every time we ran the program. With this program we have achieved what we wanted, but it is not an elegant solution. A program that works in this way can never be stored in a ROM, because the program continually changes itself (address \$0203). Moreover, there is a much nicer and shorter method of achieving the same thing, namely with 'indexed addressing'.

Indexed addressing

With indexed addressing, the contents of the index register are first added to the operand; the sum of the two numbers becomes the address that will be operated on. The codes for indexed addressing are as follows:

instr.	abs.X	Zp, X
LDA	BD	B5
STA	9D	95
LDX	-	-
STX	-	-
DEC	DE	D6
JMP	-	-

As we can see, there are two kinds of indexed addressing: again 'absolute indexed addressing', which can be used throughout the whole memory space, and 'zero page indexed addressing', which can only be used on page 0 and therefore needs only 2 instead of 3 memory locations. We will now write the program from the example again, but this time using 'indexed addressing'. Since the memory locations used lie in page 0, we use 'zero page indexed addressing'.

```
0200 A2 E0          LDX,imm    $E0
```

0202	A9 EA		LDA,imm	\$EA
0204	95 00	NEXT	STA,Zp,X	\$0000
0206	CA		DEX,impl	
0207	D0 FB		BNE,rel	NEXT
0209	4C 22 1C		JMP,abs	RST

Here too we see a new instruction at address \$0206, namely 'DEX'. This instruction decreases the contents of index register X by 1. The difference from 'DEC' is thus that the instruction does not relate to memory but to an internal register of the microprocessor. That is why this instruction can only be written with 'implied addressing'; the address is 'enclosed' in the op code.

The program works as follows: register X receives the value \$E0, then the value \$EA is written to memory via register A (address \$0204). The address where \$EA ends up is calculated by the microprocessor by adding the contents of register X to the operand of the instruction (which is at address \$0205). The result of this calculation is $\$00 + \$E0 = \$E0$, so address \$E0 receives the value '\$EA'. Next, the contents of register X are decreased by 1. As long as the result is not '00', the Z bit in the status register is set to '0' each time and the jump instruction is executed each time. So the contents of register X run through all values between \$E0 and \$01, and memory locations \$E0 . . . \$01 are filled with '\$EA'. The moment the contents of (index) register X go from 01 to 00, the Z bit in the status register is set to 1. The BNE instruction is now skipped and we jump via JMP to the monitor program.

We see that, by using indexed addressing, the same program now occupies only 12 instead of 17 memory locations. Moreover, the program itself is no longer modified, so it could be put in a ROM (Read Only Memory). There is, however, another reason why the latter method is preferable. In the first example we could only easily vary the upper limit (\$E0) of the block of memory into which a fixed value was written; the lower limit could not simply be shifted. This is because the conditional jump is only skipped when the value '00' has been reached, so the lower limit was always at address \$01.

With indexed addressing we can vary both limits without extra instructions. Suppose we now want to fill only the addresses \$0020 . . . \$0030 with a fixed value. We can use the same program again; only the limits have to be changed.

0200	A2 11		LDX,imm	\$11
0202	A9 08		LDA,imm	\$00
0204	95 1F	NEXT	STA,Zp,X	\$1F
0206	CA		DEX,impl	
0207	D0 FB		BNE,rel	NEXT
0209	4C 22 1C		JMP,abs	RST

The initial content of register X is now \$11 instead of \$E0, and the operand of STA is now \$1F instead of \$00. The contents of register X now run through the values \$11 . . . \$01, so that the upper limit lies at $\$1F + \$11 = \$30$ and the lower limit at $\$1F + \$01 = \$20$ (always the

operand + contents of index register X). In this example we have also changed the fixed value with which the memory is filled (\$08 instead of \$EA), because otherwise we could not check whether the program had carried out its task correctly. After all, addresses \$01 . . . \$E0 had already been written with '\$EA' by the previous program. If all is well, after starting the program (the time is still so short that it seems as if nothing happens) memory locations \$0020 . . . \$0030 should be filled with '08'.

Another example

In the next example we want a program that searches from address \$00E0 down to and including address \$0000 for a memory location containing '00', and then stores the address of this memory location in memory location \$00E1. If we start the program after giving, for example, address \$0037 the value '00' beforehand, the contents of \$00E1 will thus be '\$37'. If no '00' can be found, the program will keep searching and the display stays dark.

0200	A2 E1	START	LDX,imm	\$E1
0202	B5 FF	NEXT	LDA,Zp,X	\$FF
0204	F0 05		BEQ,rel	UIT
0206	CA		DEX,impl.	
0207	D0 F9		BNE,rel	NEXT
0209	F0 F5		BEQ,rel	START
020B	CA UIT		DEX,impl	
020C	86 E1		STX,Zpage	\$E1
020E	4C 22 1C		JMP,abs	RST

As you can see, this program is already considerably more complicated than the examples discussed so far. Because the number of conditional jumps has increased, some of the overview has been lost. That is precisely why a programmer uses a flow chart. Fig. 2 shows the flow chart of this program. With this flow chart you immediately have a much better overview of how the program runs. First, register X is filled with the value \$E1. This causes register A to be filled with the contents of address \$E1+\$FF=\$E0 (actually this is \$01E0, but the microprocessor ignores the carry, so that the result of 'zero page indexed addressing' always lies on page 0). If the number that has ended up in register A was not '00', the Z bit will be made 0 and the jump to 'uit' (out) is not made. The value of X is decreased by 1; as long as this is not '00' we go back to 'NEXT'. When register X has reached the value \$01, the 'DEX' instruction (address \$0206) will make register X '00'. This makes Z=1, and now 'BNE' (address \$0207) will be ignored. At address \$0209 we now jump back to 'START'. Because the BNE and BEQ instructions are written one after the other, a jump will always be made (if Z=0 via BNE to 'NEXT', and if Z=1 via BEQ to 'START'). At 'START' the search cycle begins again; however, there was no '00' in the area being searched, so the program keeps running until 'RS' or 'ST' is pressed. If a '00' is found, the microprocessor will jump via the BEQ instruction at address \$0204 to 'uit'. Here we must put the address of the memory location that contained '00' at address \$00E1. Suppose index register X has the value \$81 at that

moment; then the memory location that contained '00' can be calculated by adding \$81 to the operand of 'LDA, Zp, X'. We get $\$81 + \$FF = \$0180$. As said before, the carry into the \$100s is ignored, so the '00' found must have come from address \$80. Apparently it is sufficient to decrease index register X by 1. This is done at address \$020B. Finally, the value of index register X is put into memory location \$E1, and the program is finished.

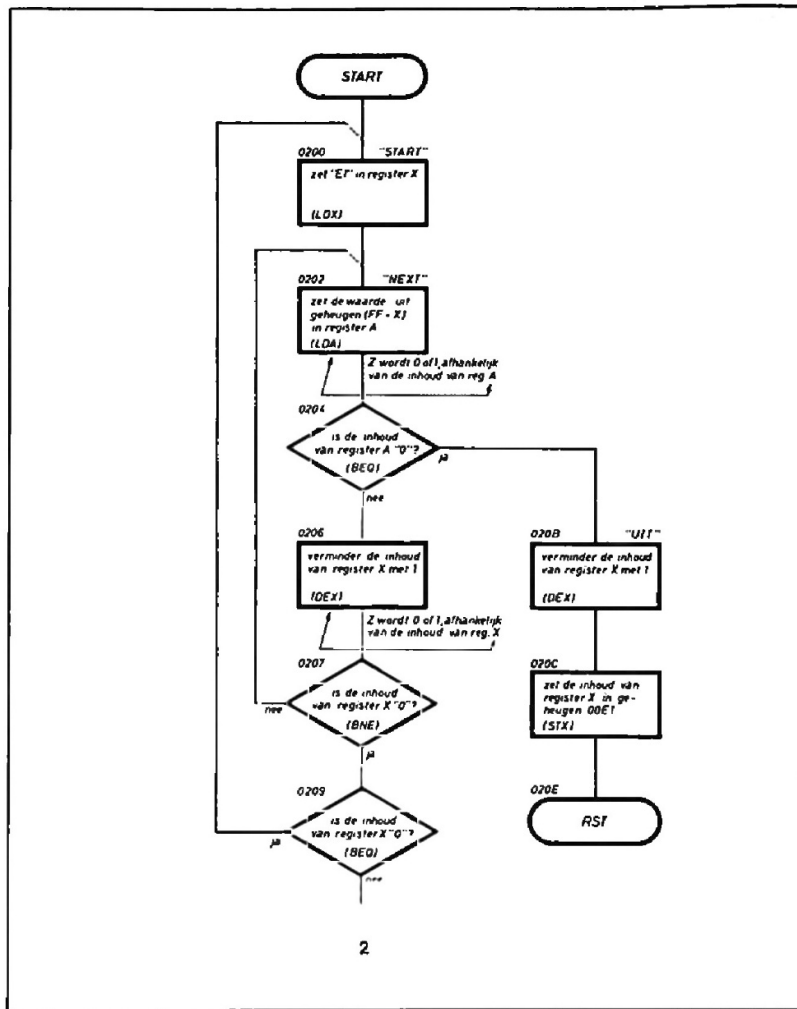


Fig. 2: The flow chart of the last example but one. Because a flow chart gives a much better overview of how a program runs, a flow chart is always made first.

Determining the relative jump

In all the conditional jumps in the previous examples, a relative address was used. That is, only the number of places to jump forward or back is given. There are various methods for determining the correct value of this jump. In theory, the value of the jump is obtained by subtracting the value of the program counter from the address to be jumped to. If we look at address 0204 in the last example, we see the instruction 'BEQ, rel UIT'. So a relative jump to \$020B has to be calculated. At the moment the microprocessor calculates the new address, however, the program counter has already been incremented to the next instruction and is thus already at \$0206. The value of the

jump is then: \$020B-\$0206 = \$5. This method is not very handy, though, because we have to do the subtraction in hexadecimal. Especially when jumping back, the negative hexadecimal value has to be converted to two's complement, which will certainly cause some problems at first. So it is best simply to count the number of steps. Using table 1, the number of steps can then be converted to its hexadecimal equivalent.

Example of determining the relative jump

We will again explain things with a small example. From the last program example we take the instruction 'BNE, rel NEXT' at address \$0207. We take into account that the program counter has already been incremented to the next instruction (address \$0209, 'F0'). We put a pen or pencil on this location and count our way through the memory contents F9, D0, CA, 05, F0, FF, B5 to address \$0202. This was the address we had to go to, and we have counted to 7. Since we have to go back, we now look up -7 in table 1. We find this number in row F0 under column 9, so for the jump (address \$0208) we can fill in \$F9. Another example: address \$0209, also from the last example, the conditional jump 'BEQ, rel START'. Again we put a pen or pencil on the next instruction (i.e. on \$020B, 'CA') and count the places F5, F0, F9, D0, CA, 05, F0, FF, B5, E1, A2. The value 'A2' is at the address we had to jump to. We end up at 11. Since we are again jumping back, we look up -11 in the table and find this number in row F0, column 5. The correct jump value is therefore \$F5. Jumps in the forward direction of the program can also be determined in this way. Again in the last example (at address \$0204) is the instruction 'BEQ, rel UIT'. Again we look at the next instruction ('CA', address \$0206). We then count to 'UIT' (over the numbers D0, F9, F0, F5, CA) and arrive at 5. This time we jump forward, so we look up +5 in the table. We find this number in row 00 under column 5. The hexadecimal equivalent is therefore also \$05. All numbers below 10 give a hexadecimal equivalent equal to the decimal number. To be safe, however, it is best (for now) always to consult the table.

Table 1. Table for converting a positive or negative jump value to its hexadecimal (two's complement) equivalent.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15	00
10	+16	+17	+18	+19	+20	+21	+22	+23	+24	+25	+26	+27	+28	+29	+30	+31	10
20	+32	+33	+34	+35	+36	+37	+38	+39	+40	+41	+42	+43	+44	+45	+46	+47	20
30	+48	+49	+50	+51	+52	+53	+54	+55	+56	+57	+58	+59	+60	+61	+62	+63	30
40	+64	+65	+66	+67	+68	+69	+70	+71	+72	+73	+74	+75	+76	+77	+78	+79	40
50	+80	+81	+82	+83	+84	+85	+86	+87	+88	+89	+90	+91	+92	+93	+94	+95	50
60	+96	+97	+98	+99	+100	+101	+102	+103	+104	+105	+106	+107	+108	+109	+110	+111	60
70	+112	+113	+114	+115	+116	+117	+118	+119	+120	+121	+122	+123	+124	+125	+126	+127	70
80	-128	-127	-126	-125	-124	-123	-122	-121	-120	-119	-118	-117	-116	-115	-114	-113	80
90	-112	-111	-110	-109	-108	-107	-106	-105	-104	-103	-102	-101	-100	-99	-98	-97	90
A0	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81	A0
B0	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65	B0
C0	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49	C0
D0	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33	D0
E0	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17	E0
F0	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	F0
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Checking the jump

If you have used conditional jumps in a program you have written yourself, it is certainly advisable first to check whether the jumps have been calculated correctly. A wrong jump often not only means that the program does not work, but can also destroy the whole program (or part of it). So, certainly at the beginning: always check the relative jumps first. This can be done as follows: first set the SST switch to 'ON', so that the program can be run in single step. Then type in address \$00F1, the address where the status register can be checked and changed. For jump instructions with the condition status bit = 0 (such as BNE) we type in '00'; for jump instructions with the condition status bit = 1 (such as BEQ) we type in '\$FF'. Now we type in the address of the jump instruction and press 'GO'. The jump is now always executed, because we have set the status register correctly beforehand. The display now shows the address to which the microprocessor has jumped. If we have jumped too far back, the jump value must be made larger. If we have jumped too far forward, the jump value must be made smaller. Once all jumps have been checked in this way, and corrected if necessary, you can start the program with peace of mind. Needless to say: an incorrect jump can never damage the KIM; at worst you will have to type the program in again.

Comparing 2 numbers

We have now seen that certain instructions make the Z bit in the status register '1' when the contents of a register or memory location become '00'. Using the state of the Z bit we can then make conditional jumps. With a special instruction it is also possible to make the Z bit become '1' when 2 numbers are equal. The instruction is called 'CMP' (compare). The codes are as follows:

instr.	imm.	abs.	Zpage	Zp,X	abs,X
CMP	C9	CD	C5	D5	DD

If, for example, we write in a program:

CMP,imm \$63

then the Z bit in the status register will become '1' if the contents of register A at that moment are also \$63. With the CMP instruction we can therefore arrange conditional jumps for any value of register A. We can also compare register A with the contents of a particular memory location:

CMP,Zpage \$63

Now the Z bit will only become 1 if the contents of register A are the same as the contents of memory location \$0063. Finally, we give another program example in which 'CMP' is used. It is once again the search program mentioned earlier. This time, however, it does not search for the value '00', but for a number that you can define yourself at address \$00E2. If a location is found containing this number, that location will be shown on the display. If no location with the defined contents is found, the display keeps showing the start address of the program.

0200	A2 E1	START	LDX,imm	\$E1
0202	B5 FF	NEXT	LDA,Zp,X	\$FF

0204	C5 E2		CMP,Zpage	\$E2
0206	F0 05		BEQ,rel	UIT
0208	CA		DEX,impl	
0209	D0 F7		BNE,rel	NEXT
020B	F0 07		BEQ,rel	RST2
020D	CA	UIT	DEX,impl	
020E	86 FA		STX,Zpage	POINTL
0210	A9 00		LDA,imm	\$00
0212	85 FB		STA,Zpage	POINTH
0214	4C 22 1C	RST2	JMP,abs	RST

We will not go further into how this program works, but check for yourself whether you arrive at the same values for the relative jumps (printed in bold). You can also see here the beginning of driving the display. We will come back to this in a later article. We will then also go a little deeper into arithmetic operations, subroutines, the keypad, and inputs and outputs.

The first two parts covered the most important addressing methods and the most common instructions. In this third part we will start on the inputs and outputs of the KIM. The internal registers S and Y also get their turn in this part, and finally we will make a start on subroutines. We will return to this last point in more detail in the next part.

The inputs and outputs of the KIM

On the KIM's application connector (the connector level with the keypad), pins 2 . . . 16 carry the 15 connections the KIM has. The 6502 microprocessor has no separate instruction for these connections. To the microprocessor, the KIM's connections simply look like memory locations. Because the data bus is 8 bits wide, a maximum of 8 connections can be on one memory location. So 2 memory locations are needed for the 15 connections.

When we type in the address of such a memory location, the display shows the state of the inputs (after pressing 'RS', all connections are programmed as inputs). When the inputs are open, we see the value \$FF at address \$1700. By converting this hexadecimal value into its binary equivalent, we can determine the state of each bit. \$FF corresponds to the binary number '11111111', so we see that all (open) inputs are read as a logic '1'. If we make certain inputs of port PA '0', we can see on the display what happens, the best proof that the KIM really is looking at its connections. As an example we now give a program that looks only at connection PA3. As long as this connection is high, the display will stay dark. When this connection goes low, the program will end, causing the display to light up. However, the computer must not react to a change in any of the other inputs. When the microprocessor receives the instruction LDA \$1700, inputs PA0 . . . PA7 are all read at the same time. To make it possible to react only to PA3, we are going to 'mask' the other bits. For this we use the 'AND' instruction.

This instruction gives as its result the logical AND of a number in register A and a number in memory. For clarity we will now write the numbers in binary instead of hexadecimal:

inhoud register A	:	00110101
inhoud geheugen	:	10101100
resultaat in register A	:	00100100

(contents of register A / contents of memory / result in register A)

We see that the normal AND is performed between all the bits. The most important point here is that a '0' in either number always gives a '0' in the result. If we now put the state of the inputs into register A and mask it with \$08, we get:

inhoud register A	:	XXXXXXXX
inhoud geheugen	:	00001000
resultaat register A	:	0000X000

Here an X means that the bit can be either '0' or '1'. In this way we can isolate any bit or any combination of bits. The program now runs as follows:

0200	AD 00 17	START	LDA,abs	PAD
0203	29 08		AND,imm	\$08
0205	D0 F9		BNE,rel	START
0207	4C 22 1C		JMP,abs	RST

At address \$200, register A is filled with a value that depends on the state of inputs PA0 . . . PA7. These inputs are located at address \$1700 (so we cannot use zero page addressing). This address (\$1700) has been given the name PAD, which is short for Port A Data. At address 0203 all bits that are not of interest are forced to '0' in the way described. Note that the binary number 00001000 is now written in the shorter hexadecimal form \$08. As long as bit PA3 is not 0, the result of this AND instruction will not be '00' either. So the Z bit in the status register will keep being set to '0', and the jump to start is made at address \$0205. As soon as PA3 has gone low, the AND instruction will give '00' as its result:

inhoud register A	:	X X X X 0 X X X
inhoud geheugen	:	0 0 0 0 1 0 0 0
resultaat register A	:	0 0 0 0 0 0 0 0

As a result, the Z bit in the status register is set to '1'. The jump instruction (BNE) is therefore not executed, and via 'JMP' we jump back to the monitor program.

Connections PB0 . . . PB5 and PB7

Of port B, connection PB6 is not available; on the KIM this connection is used for the address selection of the 6530. Port B is at address \$1702. This address has been given the name PBD (Port B Data). When you type in this address, you do not see the value \$FF as with PAD, but the value \$3F or \$BF. Written in binary, this is:

10111111 or 00111111

We see that bit PB6, which we cannot use, is read as '0'. Bit PB7 also behaves differently. This is because PB7 has no internal pull-up resistor, so when the pin is left open it has an undefined value. Once switched to output, pin PB7 behaves as an open-collector output and can be connected in parallel with other open-collector outputs.

The connections as outputs

As mentioned earlier, each connection can be either an input or an output. To define a connection as an output, we must put a '1' in the data direction register. The direction

register for connections PA0 . . . PA7 is at address \$1701 and has been given the name PADD (Port A Data Direction). The direction register for connections PB0 . . . PB5 and PB7 is at address \$1703 and has been given the name PBDD (Port B Data Direction). For an overview, here is a small table:

address	name	function
\$1700	PAD	PA connections
\$1701	PADD	PA direction
\$1702	PBD	PB connections
\$1703	PBDD	PB direction

So if we put the value \$03 in PADD, PA0 and PA1 will act as outputs and PA2 . . . PA7 as inputs. (With \$03, bits 0 and 1 are high and the rest low.) When 'RS' is pressed, all outputs are switched back to inputs. You can check this by putting a number at address \$1701. After pressing 'RS' it will always read '00' again.

Continuity buzzer

We will now show an example in which one of the connections is switched to output. To this output (PA0) we connect a loudspeaker as shown in fig. 1. The program makes the loudspeaker produce a tone as soon as PA3 is connected to ground.

0200	A9 01	START	LDA,imm	\$01
0202	8D 01 17		STA,abs	PADD
0205	AD 00 17	LEES	LDA,abs	PAD
0208	29 08		AND,imm	\$08
020A	D0 F9		BNE,rel	LEES
020C	E8	LOOP	INX,impl	
020D	D0 FD		BNE,rel	LOOP
020F	EE 00 17		INC,impl	PAD
0212	4C 05 02		JMP,abs	LEES

We see that as the first step in the program we program connection PA0 as an output and the rest as inputs (\$01 is put in PADD). The actual program begins at address 0205. The first part is the program from the previous example; this part determines whether input PA3 is high or low. As long as PA3 is high (open input), the microprocessor will keep jumping back to 'LEES' (read), address \$0205. As soon as PA3 is low, the jump to 'LEES' is not made (the result of AND \$08 is now 0, so Z becomes '1'). We now arrive at the part of the program that generates the tone. The principle is simple: memory location PAD is incremented by 1, and via JMP we jump back to 'LEES'. As long as PA3 is low, we keep running through the first part (address \$0205 . . . \$020A) and 'PAD' is

continually incremented by '1' (address 020F). Because of this incrementing, PA0 continually flips from 0 to 1 and then from 1 to 0. The other connections are programmed as inputs and do not respond to this increment instruction. (Inputs can only be made high or low from outside.) So a square wave now appears on output PA0. Because the microprocessor only has to execute a few instructions, the increments follow each other very quickly and the frequency of the square wave lies above the limit of hearing. That is why a loop has been included at addresses \$020C . . . \$020E. In this loop the microprocessor will keep incrementing the contents of register X until the contents of this register are '00'. Because of this extra delay, the period of the square wave becomes long enough to produce an audible tone. One difference from previous programs is that this program is executed over and over again. As a result the display goes dark. Only by pressing 'RS' or 'ST' (if you had entered the NMI vector, 00 at \$17FA and \$1C at \$17FB) can you return to the monitor program.

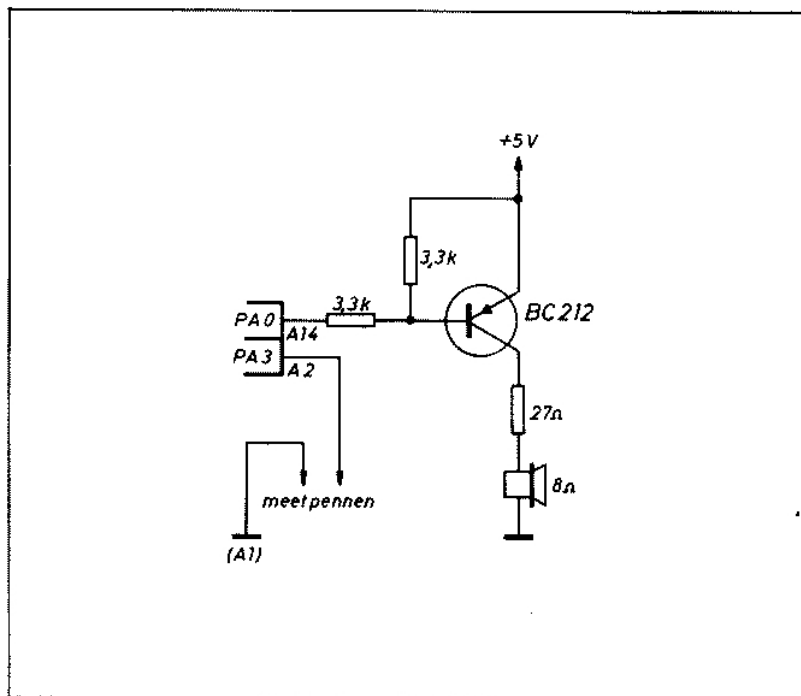


Fig. 1: The connections to be made for the continuity buzzer.

The stack

A completely different subject is the KIM's stack. The stack was dealt with once before, in one of the first articles in RB about the microprocessor. The stack is a pile of registers in which data can be stored temporarily. One advantage is that the memory location in which a particular value is placed does not have to be addressed; a limitation is that the data can only be retrieved in reverse order. For now, the instructions belonging to the stack are:

instruction	code
PHA	48
PLA	68

The instruction 'PHA' (PUSH register A) means: 'put the contents of register A on the stack'. This instruction occupies only one memory location and is therefore attractive. The opposite instruction is 'PLA' (PULL register A). This instruction puts the value that was last placed on the stack into register A. The values placed on the stack ultimately simply end up at certain addresses in RAM; how this works we will deal with later, but first an example in which we use the stack.

Example

In the following example we first put 5 different numbers on the stack and then take these numbers off again. We put the first number taken from the stack at address \$0000; the last number goes to address \$0004. This way we can see clearly in which order the numbers come off the stack.

0200	A9 11	IN	LDA,imm	\$11
0202	48		PHA,imp	
0203	A9 22		LDA,imm	\$22
0205	48		PHA,imp	
0206	A9 33		LDA,imm	\$33
0208	48		PHA,imp	
0209	A9 44		LDA,imm	\$44
020B	48		PHA,imp	
020C	A9 55		LDA,imm	\$55
020E	48		PHA,imp	
020F	68	UIT	PLA,imp	
0210	85 00		STA,Zpage	nul
0212	68		PLA,imp	
0213	85 01		STA,Zpage	een
0215	68		PLA,imp	
0216	85 02		STA,Zpage	twee
0218	68		PLA,imp	
0219	85 03		STA,Zpage	drie
021B	68		PLA,imp	
021C	85 04		STA,Zpage	vier
021E	4C 22 1C		JMP,abs	

After running this program we see that the number put on the stack first is the last one to come off (address \$0000 contains \$55, and address \$0004 contains \$11).

How the stack works

The KIM uses part of the RAM for the stack, namely the address space \$0100 . . . \$01FF. In a special register the microprocessor keeps track of which address has not yet been used. This register has been given the name 'S', short for Stack Pointer. After pressing 'RS' the contents of register 'S' will always be \$1FF, so the first number put on the stack with PHA will end up at address \$01FF. In addition, 'PHA' automatically decreases the stack pointer by 1, to \$1FE. So in our previous example \$11 was put at address \$01FF, \$22 at address \$01FE, \$33 at address \$01FD, \$44 at address \$01FC and finally \$55 at address \$01FB.

If you look at address \$01FB after running the last example, you will also see \$55 there. Addresses \$01FC . . . \$01FF have meanwhile been overwritten by the monitor program, because the monitor program also uses the stack. With every 'PLA' the reverse happens: the stack pointer is increased, and the number at the location indicated by the stack pointer is put into register A. At address \$00F2 we can always see the state of the stack pointer. After pressing 'RS' the contents of \$00F2 will always be \$FF (address \$00F2 is not the stack pointer itself; it is a copy of the stack pointer made by the monitor program). By running the previous example in single step you can follow the whole process. After each PHA (\$48) you see that the stack pointer (which can be checked at address \$00F2) has been decreased by 1, and you see the numbers \$11 . . . \$55 being put on the stack one by one. After each PLA (\$68) you see the reverse happen. Because the monitor program also uses the stack, numbers that have been taken off the stack will immediately be overwritten, giving the impression that numbers taken off the stack really disappear from memory. In reality (when we are not working in single step) a number simply stays in memory after a PLA. A copy of that number from the stack goes into register A, while the stack pointer points to that memory location as the first location where a number can be put again. The stack pointer handles all this flawlessly, however, so we need not go into this matter too deeply.

Subroutines

In programs it often happens that certain pieces of program occur many times. It is then handy to implement such a piece of program as a subroutine. We will again explain this with an example. Earlier in this article we discussed the inputs of the KIM. As an example we will now make a program that counts how many times a particular input has been '0', i.e. how many times this input has been connected to ground. When the program starts, the input will be '1' (left open). In the loop we now wait until this input goes low. As soon as that happens, a counter must be incremented, so that we can see how many times the input has been low. After this we must wait again until the input goes high, and only then can we jump back to the beginning of the program. To prevent contact bounce from incrementing the counter too often, we also build a delay into the

program. The program (without subroutines) is shown in list 1. We see that an extra column has been added to this list, giving a brief explanation of the program steps. We see a number of new instructions in this program. First 'TAX' (address \$0204); this instruction makes a number that is in register A also appear in register X. We use this instruction here to give register X the initial content '00'. In principle we could also have used LDX,imm \$00, but that instruction occupies 2 memory locations instead of 1.

We are also introduced here to the second index register, namely index register Y. This register has roughly the same capabilities as register X. For some tasks register X has the most attractive instructions; for other tasks register Y is preferable. Here register Y is used to nest 2 loops inside each other. This is because 1 loop does not give enough delay to absorb the contact bounce. Register X always starts a loop with '00' and always ends a loop with '00'. Register Y is given a starting value each time; with this value the delay can be made longer or shorter. The instructions for register Y:

instr.	imm.	abs.	Zpage	Zp,X	abs,X	impl.
LDY	A0	AC	A4	B4	BC	-
DEY	-	-	-	-	-	88

The instruction 'DEY' (decrement reg. Y) decreases the contents of index register Y by one. Because this instruction always relates to register Y, no further address is needed, and the addressing mode is implied.

But back to our example: when we start the program, the display goes dark. We now make input PA0 '0' a number of times by connecting it to ground. We press 'RS', and at address \$0000 we now see how many times input PA0 has been low.

The same example using subroutines

In the program of list 1 we see that the piece of program between addresses \$0205 and \$0212 is almost identical to the piece between \$0216 and \$0223. Only addresses \$020A and \$021B differ slightly. The first time, we have to wait until the input has become '0'. The second time, we have to wait until the input has become '1'. Despite this small difference, we can still implement these pieces of program as a subroutine; we simply postpone the decision whether to jump back on input = 1 or input = 0 until we return from the subroutine. To achieve this we must store the contents of the status register for a moment, because the state of the Z bit in this register indicates whether the input was 0 or 1 (see the previous part). The KIM has a handy instruction for this, namely:

instr.	code
PHP	08
PLP	28

So we can put the contents of the status register on the stack. The delay loops change the state of the Z bit, but after 'PLP' the Z bit once again reflects the input condition. (It got this value after the AND instruction: Z=1 if PA0=0 and vice versa.) List 2 now shows the program using subroutines. In our case the program is four memory locations shorter, and it hardly seems worth the trouble. But you must bear in mind that we have used a short program as an example.

The longer the subroutine, and the more often it is called, the greater the saving in memory space. In addition, a program often becomes somewhat clearer when subroutines are used. This is because a particular subroutine is soon seen as a kind of 'custom-made instruction'. In the next part we will go a little deeper into how subroutines work. The arithmetic operations, keypad and display announced earlier are also still on the programme.

Lijst 1

0200	A9 00	START	LDA,imm	\$00	} register op 0
0202	85 00		STA,Zpage	TELLER	
0204	AA		TAX,impl		
0205	AD 00 17	WACHT1	LDA,abs	PAD	} wacht tot de ingang 0 is geworden
0208	29 01		AND,imm	\$01	
020A	D0 F9		BNE,rel	WACHT1	
020C	A0 40		LDY,imm	\$40	} wacht totdat er geen dender meer is
020E	E8	LOOP1	INX,impl		
020F	D0 FD		BNE,rel	LOOP1	
0211	88		DEY,impl		} hoog de teller op wacht tot de ingang 1 is geworden
0212	D0 FA		BNE,rel	LOOP1	
0214	E6 00		INC,Zpage	TELLER	
0216	AD 00 17	WACHT2	LDA,abs	PAD	} wacht totdat er geen dender meer is
0219	29 01		AND,imm	\$01	
021B	F0 F9		BEQ,rel	WACHT2	
021D	A0 40		LDY,imm	\$40	} wacht totdat er geen dender meer is
021F	E8	LOOP2	INX,impl		
0220	D0 FD		BNE,rel	LOOP2	
0222	88		DEY,impl		} ga terug
0223	D0 FA		BNE,rel	LOOP2	
0225	4C 05 02		JMP,abs	WACHT1	

Lijst 2

0200	A9 00	START	LDA,imm	\$00	} register op 0
0202	85 00		STA,Zpage	TELLER	
0204	AA		TAX,impl		
0205	20 14 02	WACHT1	JSR,abs	SUB	} wacht tot de ing. 0 is, en uitgedenderd hoog de teller op wacht tot de ing. 1 is en uitgedenderd, ga terug
0208	D0 FB		BNE,rel	WACHT1	
020A	E6 00		INC,Zpage	TELLER	
020C	20 14 02	WACHT2	JSR,abs	SUB	} is de ingang 0 of 1? zet status weg
020F	F0 FB		BEQ,rel	WACHT2	
0211	4C 05 02		JMP,abs	WACHT1	
0214	AD 00 17	SUB	LDA,abs	PAD	} wacht totdat er geen dender meer is
0217	29 01		AND,imm	\$01	
0219	08		PHP,impl		
021A	A0 40		LDY,imm	\$40	} haal status terug terug naar het hoofdprogramma
021C	E8	LOOP	INX,impl		
021D	D0 FD		BNE,rel	LOOP	
021F	88		DEY,impl		
0220	D0 FA		BNE,rel	LOOP	
0222	28		PLP,impl		
0223	60		RTS,impl		

In the first part of this series the 3 most important addressing methods were explained. The second part dealt mainly with conditional jump instructions. The third part covered the input and output facilities, the stack and subroutines. After a short break in this series (the first three parts appeared in March, April and May of this year), in this 4th part we will look at adding and subtracting hexadecimal and decimal numbers. The 'two's complement' representation of negative numbers will also be covered.

Adding two 8-bit binary numbers

We start with the simplest arithmetic operation, namely adding two 8-bit numbers. The instruction the 6502 has for this is 'ADC'; this instruction adds a number from memory to the contents of the accumulator. The codes for the various addressing methods covered so far:

inst.	imm.	abs.	Zpage
ADC	69	6D	65

Zp,X	abs,X	abs,Y
75	7D	79

In the following example we assume that the two numbers to be added are at addresses \$00 and \$10:

0200	18	START CLC impl.	
0201	D8	CLD impl.	
0202	A5 00	LDA, Zpage	GETAL 1
0204	65 10	ADC, page	GETAL 2
0206	85 20	STA, Zpage	GETAL 3
0208	4C 22 1C	JMP, abs.	RST

If we now put, for example, \$12 at address \$100 and \$28 at address \$10, then (after running the program) \$3A will be at address \$20. This is indeed the hexadecimal sum of the two numbers. We will come back to the 'CLC' instruction (address \$0200) later. With the instructions 'CLD' and 'SED' ('clear decimal mode' and 'set decimal mode' respectively) we have the very handy option of performing an addition in decimal or in hexadecimal. In the program above we used 'CLD', so the addition is performed in hexadecimal. (For the microprocessor this is the 'normal' way.) If we replace the 'CLD' instruction with 'SED', the addition will be performed in decimal:

0200	18	START CLC impl.
0201	F8	SED, impl.

0202	A5 00	LDA, Zpage	GETAL 1
0204	65 10	ADC, page	GETAL 2
0206	85 20	STA, Zpage	GETAL 3
0208	4C 22 1C	JMP, abs.	RST

With 12 at address \$00 and 28 at address \$10, we now get 40 as the result at address \$20. The rest of the program speaks for itself: at address \$0202, GETAL 1 (number 1) is put in the accumulator. At address \$0204 the contents of address \$10 (GETAL 2) are added to the number in the accumulator. The result is also in the accumulator. With 'STA' at address \$0206 the answer is moved to address \$20. Finally we jump back to the monitor program, so that the display lights up and we can use the keys again.

The result does not fit in one memory location

The program above lets us add 2 numbers. But now try, for example, 86 and 95 ('86' at address \$00, '95' at address \$10, and start the program at address 0200). The result is 81, while it should be 181. The cause of this error will be clear: the number 81 is stored in memory as 1000 0001; that is 8 bits, and there is no room left for the hundreds. It would of course be absurd if the microprocessor could not handle larger numbers. So the hundreds are not lost; they are remembered in the 'carry bit' of the status register. This bit becomes '1' if a 'carry' occurs during an addition, and '0' if no carry occurs. Earlier we mentioned that the 'ADC' instruction adds the contents of a memory location to the contents of the accumulator (= register A). However, this was not the whole story: the contents of the carry bit (0 or 1) are also added to register A. This immediately explains the first instruction of the program, since 'CLC' means 'clear carry'. Because the carry bit is added in every addition, we must first set this carry bit to 0.

if we now want the program to show the hundreds as well, we must extend the answer to 16 bits. The units and tens go to address \$20, the hundreds to address \$21:

0200	18	START CLC, impl.	
0201	F8	SED, impl.	
0202	A5 00	LDA, Zpage	GETAL 1
0204	65 10	ADC, Zpage	GETAL 2
0206	85 20	STA, Zpage	GET. E 3
0208	A9 00	LDA, imm.	\$00
020A	69 00	ADC, imm.	\$00
020C	85 21	STA, Zpage	GET. H 3
020E	4C 22 1C	JMP, abs.	RST

If we now put '86' at address \$00 and 95 at address \$10, after running the program address \$21 will contain '01' and address \$20 will contain '81', so the answer is 181. We can of course also add in hexadecimal by replacing the 'SED' instruction at address

\$0201 with 'CLD', i.e. D8 instead of F8. In that case you get as the answer:

```
$86 = 1000 0110
$95 = 1001 0101
      ----- +
      10001 1011 = $11B
```

You can see that hexadecimal numbers are easiest to add by writing them in binary. The computer does indeed give the number \$1B at address \$20 and the number \$01 at address \$21. The \$ sign in front of the numbers indicates that these are hexadecimal numbers, which are not equal in value to the decimal numbers.

The program has remained the same up to address \$0208. At address \$0208 we make register A '00'. With the instruction 'ADC, imm. \$00' the value '00' and the contents of the 'C' bit are added to the contents of register A. The end result is that register A gets the value '01' if the carry bit is '1', and the value '00' if the carry bit is '0'. In this way the hundreds are now in register A. At address 020C these contents are stored at address \$21. Needless to say, once more the difference between 'ADC, Zpage \$00' and 'ADC, imm. \$00': in the first case the contents of memory location '00' are added to the contents of the accumulator (register A); in the second case the number '00' is added to the contents of register A.

Making the result visible

In the examples above the computer always calculates the answers flawlessly. But if we want to see the answer, we always have to type in its address. It would be nicer if the answer appeared on the display straight away. To achieve this we can use a subroutine from the monitor program, namely 'SCANDS'. This subroutine provides all the control signals needed to drive the display. To use this subroutine we must put the units and tens at address \$F9, the hundreds and thousands at address \$FA, and finally the tens of thousands and hundreds of thousands at address \$FB. Once addresses F9 . . . FB have been filled with the correct values, the instruction 'JSR, abs. SCANDS' is enough to make the display work for a few ms. By jumping to this subroutine over and over again, the display stays lit continuously. The program now becomes:

0200	18	START	CLC impl.	
0201	F8		SED, impl.	
0202	A5 00	LDA, Zpage	GETAL 1	
0204	65 10	ADC, Zpage	GETAL 2	
0206	85 F9	STA, Zpage	EENHED	
0208	A9 00	LDA, imm.	\$00	
020A	85 FB	STA, Zpage	TIENDU	
020C	69 00	ADC, imm.	\$00	

020E	85 FA		STA, Zpage	HONDER
0210	20 1F 1F	DIS	JSR, abs.	SCANDS
0213	4C 10 02		JMP, abs.	DIS

We see that the answers are no longer moved to address \$20 but to address \$F9. The hundreds go to \$FA instead of \$21. Let us go through the program once more from the beginning.

Address \$0200. Here the carry bit in the status register is set to '0'. This is necessary because the addition further on in the program always adds the carry bit too. (Other microprocessors often have a separate add instruction that does not include the carry bit.)

Address \$0201. Here the decimal mode bit in the status register is set to '1'. This makes all arithmetic operations be performed in decimal. (Normally: $2 + 8 = A$; now: $2 + 8 = 10$)

Address \$0202. The first number is put in register A, so that the 2nd number can then be added to it.

Address \$0204. The second number, together with the contents of the carry bit, is added to the contents of register A. The result stays in register A. The carry bit had already been set to '0', so register A now contains the sum of numbers 1 and 2. If the result needs more than 8 bits, the carry bit is also set to '1' (i.e. if the result is ≥ 100).

Address \$0206. The answer (without the hundreds) is put at address \$F9, so that it will later be visible on the two right-hand displays. The carry bit is left untouched.

Address \$0208. The contents of register A are made '00'.

Address \$020A. The contents of register A (00) are put at address \$FB. As a result, the 2 left-hand displays will show the digits '00'.

Address \$020C. The fixed value '00' plus the value of the carry bit is added to the contents of register A (still '00'). The result will be '01' if the carry bit was '1', and '00' if the carry bit was '0'.

Address \$020E. Here the contents of register A are put at address \$FA. The result is that the middle two displays will show '00' or '01', depending on the state of the carry bit.

Address \$0210. The contents of addresses \$FB, \$FA and \$F9 are made visible on the display, from left to right, for a few ms.

Address \$0213. Because of the jump to address \$0210, the 'SCANDS' subroutine is repeated endlessly, so that the answer stays on the display continuously. The program can only be interrupted with 'RS' or 'ST'.

Adding larger numbers

It is of course also possible to add numbers of more than 2 digits. By chaining enough memory locations together we can add numbers of 3, 4, 5 etc. digits. There is hardly any limit. (Memory space is the only limiting factor.) If we used all the memory locations of page 0, we could add 2 numbers of 128 bytes each. Each byte can hold 2 digits (decimal or hex), so numbers of 256 digits can easily be added. And we would have used only

1/4k of memory. Of course it is pointless to work with such long numbers. In our example we will limit ourselves to numbers of 6 digits (which fits the display exactly). First let us ask ourselves how you would add such a number yourself:

```
598169
139516
----- +
```

For this addition we would proceed as follows:

```
9 + 6 = 15 -> write down 5, carry 1
1 + 6 + 1 = 8 -> write down 8
1 + 5 = 6 - > write down 6
8 + 9 = 17 - > write down 7, carry 1
1 + 9 + 3 = 13 -> write down 3, carry 1
1 + 5 + 1 = 7 -> write down 7
```

This gives us the result: 737685. You may recognise the way the 'ADC' instruction works. Just as with this instruction, you have each time added the carry from the previous addition to the new sum. The microprocessor, however, always works with 2 digits at a time:

```
0 + 69 + 16 = 85  carry = 0
0 + 81 + 95 = 76  carry = 1
1 + 59 + 13 = 73  carry = 0
```

As you can see, we must again start at the end and always include the carry from the previous addition. We must make sure that the carry is '0' the first time. The program becomes:

0200	18	START CLC, impl.
0201	F8	SED, impl.
0202	A5 00	LDA, Zpage GET 1, 1
0204	65 10	ADC, Zpage GET 2, 1
0206	85 F9	STA, Zpage EENHED
0208	A5 01	LDA, Zpage GET 1, 2
020A	65 11	ADC, Zpage GET 2, 2
020C	85 FA	STA, Zpage HONDER
020E	A5 02	LDA, Zpage GET 1,3
0210	65 12	ADC, Zpage GET 2, 3
0212	85 FB	STA, Zpage TIENDU

0214	20 1F 1F	DIS	JSR, abs.	SCANDS
0217	4C 14 02		JMP, abs	DIS

The numbers to be added are each spread over 3 memory locations. Sticking with the two numbers from the example (598169 + 139516), the numbers must be put into memory as follows:

address contents

00	69 \
01	81 } number 1
02	59 /

10	16 \
11	95 } number 2
12	13 /

So we have chained 3 bytes together. The addition always consists of 3 parts: fetching a part of number 1, adding the corresponding part of number 2, and storing the answer. These 3 steps must be repeated until the whole number has been added. The rest of the program speaks for itself.

Entering the numbers could of course be done much more nicely. We could, for example, write a helper program that shows the numbers as they are keyed in, just like a calculator. It is of course also possible to program a complete calculator. Such programs, however, become rather long and are therefore beyond the scope of this series of articles. Perhaps we will devote a separate article to this in due course.

Subtracting 2 binary numbers

Subtracting 2 binary numbers is in principle no more difficult than adding. Here too there is a separate instruction, namely 'SBC'. With this instruction we subtract the contents of a memory location from the contents of the accumulator. The codes of 'SBC' for the addressing methods discussed so far:

instr.	imm.	abs.	Zpage
SBC	E9	ED	E5

Zp,X	abs,X	abs,X
F5	FD	F9

In the following example we again assume that the two numbers to be subtracted are at addresses \$00 and \$10:

0200	38	START SEC, impl.	
0201	D8	CLD, impl.	
0202	A5 00	LDA, page	GETAL 1
0204	E5 10	SBC, Zpage	GETAL 2
0206	85 20	STA, Zpage	GETAL 3
0208	4C 22 1C	JMP, abs RST	

Because the contents of memory are subtracted from the contents of the accumulator, number 2 will now be subtracted from number 1 (after all, we put number 1 in the accumulator). We now put '0A' at address \$00 and '08' at address \$10. After running the program, address \$20 will contain '02'. This is indeed the difference between \$0A (= 10) and \$08 (= 8).

One more thing about the program: instead of 'CLC' we now see the instruction 'SEC' (set carry). For subtraction the carry bit must be '1'. Why this is so will become clear later on.

Negative numbers

Something we have not yet discussed is negative numbers. There are several ways of representing negative numbers in binary. The most widely used is the 'two's complement' notation. We will limit ourselves to this notation, because it is used in practically all microprocessors. To represent, for example, the number -\$67 in two's complement, we must first write the hexadecimal number in binary:

-\$67 -> 0110 0111

then every bit of the number must be inverted:

0110 0111 -> 1001 1000

As the last step, 1 must be added to the result:

1001 1000 + 1 = 1001 1001

written in hexadecimal this is \$99.

When we work with negative numbers, \$99 **always** means -\$67. This notation has many advantages, for example when we add numbers. Suppose we want to add the number \$69 to the number -\$67.

In the computer, -\$67 is written as \$99:

\$69	01101001
\$99	10011001

```

----- +
100000010

```

So the answer is 0000 0010, which corresponds to \$02. (The carry bit has become '1'.)
 So we see that, thanks to this notation, we can simply add both negative and positive numbers. Even when the answer is negative, the correct number comes out:

```

      $65  01100101
$-67 = $99 10011001
            ----- +
            11111110

```

So this answer should be the two's complement of \$-02, which is indeed correct.
 We are now left with one problem: how can you recognise a negative number? In other words, is the number \$99, for example, really \$99, or is it the two's complement notation of \$-67?

To obtain an unambiguous indication, it has therefore been agreed that bit 7 (in our examples the leftmost bit) determines the sign. If bit 7 = 0 the number is positive, and if bit 7 = 1 the number is negative. The consequence of this is that numbers greater than \$7F can no longer be represented in 1 byte. The largest negative number is \$80. In decimal, the range of 1 byte now runs from -128 . . . +127:

decimal:	hexa	binary:
.	decimal:	
127	7F	01111111
126	7E	01111110
.	.	.
.	.	.
2	02	00000010
1	01	00000001
0	00	00000000
-1	FF	11111111
-2	FE	11111110
.	.	.
.	.	.
-127	81	10000001
-128	80	10000000
		^
		sign bit

The carry bit

We can also regard adding a positive and a negative number as subtracting two numbers. The addition \$69 + \$99 should therefore give the same result as the subtraction \$69 - \$67. After all, the number \$99 is the two's complement notation of \$-67. In the addition \$69 + \$99 the carry bit was set to '1', so in the subtraction \$69 - \$67 the carry bit is also set to '1'. In the addition \$65 + \$99 the carry bit was set to '0', and so in the subtraction \$65 - \$67 the carry bit will also have to be set to '0'. So we see that the carry bit becomes '0' if the result is negative, or (which amounts to the same thing) if we have borrowed from a following byte. As an example we will again subtract two 6-digit numbers:

```
598169
139516
-----
```

If we do the calculation ourselves:

```
9 - 6 = 3
6 - 1 = 5
1 - 5 is not possible, borrow one:
11 - 5 = C (hex!!)
(8 - 1) - 9 is not possible, borrow one:
17 - 9 = E (hex!!)
(9 - 1) - 3 = 5
5 - 1 = 4
```

so the answer is 45EC53.

As you can see, we have done the subtraction in hexadecimal. The computer now calculates as follows:

69 - 19 - 0 = 53 carry bit = 1 so no borrow

81 - 95 - 0 = EC carry bit = 0 so there is a borrow

59 - 13 - 1 = 45 carry bit = 1 so no borrow

'Borrow' is the English borrowing one. We see that if the carry bit is '1', no borrowing is needed, and if the carry bit is '0', a borrow is needed (the result of the middle operation, 'EC', is the two's complement of -14!!).

Back to the computer

The computer program for this subtraction looks like this:

```

0200 38          START SEC, impl.
0201 D8          CLD, impl.
0202 A5 00       LDA, Zpage  GET 1, 1
0204 E5 10       SBC, Zpage  GET 2,1
0206 85 F9       STA, Zpage  EENHED
0208 A5 01       LDA, Zpage  GET 1, 2
020A E5 11       SBC, Zpage  GET 2,2
020C 85 FA       STA, Zpage  HONDER
020E A5 02       LDA, Zpage  GET 1,3
0210 E5 12       SBC, Zpage  GET 2,3
0212 85 FB       STA, Zpage  TIENDU
0214 20 1F 1F    DIS  JSR, abs.  SCANDS
0217 4C 14 02    JMP, abs.  DIS

```

You can see that the program is very similar to the addition program for two 6-digit numbers. The borrow (= carry) is automatically included by the 'SBC' instruction. This means that the first time, the carry must be set to '1' (C = '1' means 'not borrowed'). If you want the computer to perform the calculations above, the numbers must again be entered as follows:

```

address contents
00  69 \
01  81 } number 1
02  59 /

10  16 \
11  95 } number 2
12  13 /

```

If you swap the two numbers, the final result becomes BA13AD, which is the two's complement of -45EC53:

```

BA13AD:  1011 1010 0001 0011 1010 1101
invert:   0100 0101 1110 1100 0101 0010
add 1:    0100 0101 1110 1100 0101 0011
back to hex:  4  5  E  C  5  3
               ^
             sign bit

```

Although the 24-bit number BA13AD occupies 3 bytes in memory, only the very first bit of these 3 consecutive bytes determines the sign.

Decimal subtraction

By setting the 'decimal mode flag' to '1', we can also perform the same operation in decimal. We then get as the answer:

$$598169 - 139516 = 458653$$

(you only need to put 'F8' at address \$0201 instead of 'D8'). When the answer is negative, however, things no longer work out so neatly. The two's complement notation no longer applies. If we subtract the numbers the other way round, we get:

$$139516 - 598169 = 541347$$

We can still find the correct value by taking the 'ten's complement':

number	541347
invert	
(complement to 9)	458652
add 1	458653
answer	458653

You can see that the method used is almost identical, but for the computer it becomes a good deal more difficult. Firstly, it is now no longer so easy to determine whether a number is negative. Secondly, the ten's complement is not easy to find either. (For the microprocessor, the two's complement is a single instruction.) That is why, when calculating in decimal, the sign is often kept separately. A special routine then makes sure that the smaller number is always subtracted from the larger, while another routine makes sure the answer gets the correct sign again. So decimal arithmetic has the advantage that we do not have to convert from hex to decimal and back, but the disadvantage that negative numbers are somewhat harder to handle. As a result, calculations in the hexadecimal system will also run somewhat faster.

Adding two 8-bit numbers, the carry bit, making the result visible, adding larger numbers, decimal or binary. Subtracting two binary numbers, notation of negative numbers (two's complement). The carry bit in subtraction (borrow).

Adding 2 negative numbers

In the previous part we already gave a small program for adding two 8-bit numbers:

0200	18	START	CLC, impl	
0201	D8		CLD, impl	
0202	A5 00	LDA, page	GETAL	1

0204	65 10	ADC, Zpage	GETAL 2
0206	85 20	STA, Zpage	GETAL 3
0208	4C 22 IC	JMP, abs	RST

This program adds a number at memory location \$00 to a number at location \$10. The answer goes to address \$20. If we put \$0F at address \$00 and \$07 at address \$10, then after running the program address \$20 will contain \$16 ($\$0F + \$07 = \$16$). All this was explained in detail in the previous part. We also saw then that we can simply add negative numbers, provided they are written in two's complement. So if we want to add \$-0F and \$-07, we must first convert these numbers to two's complement:

written in binary: 0F -> 00001111
 invert: -> 11110000
 add one: -> 11110001
 back to hex: -> F1

In the same way we find the two's complement \$F9 for \$-07. We now put \$F1 (-0F) at address \$00 and \$F9 (-07) at address \$10. As the answer we get \$EA at address \$20. So this must be the two's complement notation of the answer. Going back to the notation with a minus sign, which is easier for us to read, works the same way as converting to two's complement:

write in binary: EA - > 11101010
 invert: - > 00010101
 add one: - > 00010110
 back to hex: - > 16

So the answer is \$-16, which is indeed correct. You may wonder why we make things so complicated. After all, we could simply add \$0F and \$07 and remember that these are negative numbers. The advantage of two's complement notation is that we can add and subtract positive and negative numbers mixed together, without exception.

Another example

Let us continue with the example above. Suppose we want to add the numbers \$-70 and \$-13. The result should therefore be \$-83. First calculate the two's complement:

\$-70 -> \$90
 \$-13 -> \$ED

If we put these numbers at addresses \$00 and \$10, after running the program we will find the number \$7D at address \$20. Let us convert back:

```

write in binary:    7D -  > 01111101
invert:            -      > 10000010
add one:           -      > 10000011
back to hex:       -      > 83

```

And we see that the answer is indeed correct, or at least it seems so!! If \$7D is indeed taken to be a negative number, we do get \$-83 as the answer. But in the previous part we saw that bit 7 (the leftmost bit) must be '1' if a number is negative and '0' if it is positive. In the number \$7D bit 7 has the value '0', so the microprocessor takes \$7D to be a positive number.

The overflow flag

It will be clear that errors like the one above must not go unnoticed. So what is actually going on? With one byte we can make 256 different bit combinations (2^8). If we are not working with negative numbers, we can therefore represent all numbers between 0 and 255. If we work with both positive and negative numbers, this range lies between -128 and +127, or in hexadecimal between \$-80 and \$7F. With \$-83 we have passed this limit, so the answer was regarded as positive. When this situation occurs, the overflow flag is set in the microprocessor. This gives us the opportunity to give an error message or apply a correction.

Example:

We will now extend the program given earlier with an error message if an overflow occurs.

```

0200 18          START CLC, impl
0201 D8          CLD, impl
0202 A5 00        LDA, page   GETAL 1
0204 65 10        ADC, Zpage  GETAL 2
0206 70 FE        FOUT BVS, rel  FOUT
0208 85 20        STA, Zpage  GETAL 3
020A 4C 22 1C     JMP, abs    RST

```

If we add the numbers \$60 and \$10 (i.e. 2 positive numbers), after running the program address \$20 will contain \$70. Nothing wrong, then. If we add \$60 and \$20, the display will go dark (\$60 at address \$00, \$20 at address \$10, start the program at address \$0200). In this way the program tells us that an overflow has occurred. The program can be stopped again with 'RS'. Before going into how the program works, let us first make clear why an overflow occurred, since this time we added two positive numbers! The expected answer should be \$20 + \$60 = \$80. If we work only with positive numbers, this answer is correct and we need not pay any attention to the overflow flag. But if,

within one program, we distinguish between positive and negative numbers, the number is taken to be negative, because bit 7 has the value '1' (\$80 = 1000 0000). So it comes down to the fact that we have again exceeded the available capacity (\$-80 . . . \$+7F), and so the overflow flag was set. In the program we see that the instruction 'BVS, rel FOUT' has been inserted at addresses \$0206 and \$0207. BVS stands for 'Branch if oVerflow Set', or in Dutch: 'maak een sprong als het overflowbit 1 is' (jump if the overflow bit is 1). The jump value has been filled in so that the program keeps jumping back to this same instruction. (Jump instructions were covered in detail in part 2 of 'programming step by step'.) As a result the program will never jump back to the monitor, and the display stays dark. Of course we could also jump to another address where, for example, a somewhat more elegant error message is produced. (FOUT means 'error'.)

The difference between 'overflow' and 'carry'

In the previous part we went into the carry flag in detail. You may remember that the carry flag was also set when a result no longer fitted in one byte. There are, however, clear differences:

The carry flag is set when the answer of an addition becomes greater than 255. The result then no longer fits in a byte.

Here the carry has the function of 'carrying one' in an addition. In subtraction, too, the carry bit is used as 'borrow' or 'borrowing one'. The carry allows us to chain bytes together into 16, 24, 32 etc. bit numbers. The overflow flag indicates that the sign bit has been corrupted because the answer became greater than \$7F or less than \$-80. If we work only with positive numbers we need not pay attention to the overflow. If we work with positive and negative numbers longer than 1 byte, we only need to watch the overflow on the most significant byte (i.e. the byte containing the sign bit).

Multiplication

We have now come to multiplying two numbers. By far the simplest way to multiply is repeated addition. If, for example, we want to calculate 5 x 6, we reduce this to 5 + 5 + 5 + 5 + 5 + 5 (which is also how you learned to multiply). The program for this way of multiplying is very simple:

0200	F8	START SED, impl	
0201	18	CLC, impl	
0202	A6 00	LDX, Zpage	GETAL 1
0204	A9 00	LDA, imm	\$00
0206	65 10	TELOP ADC, Zpage	GETAL 2
0208	CA	DEX, impl	
0209	D0 FB	BNE, rel	TELOP
020B	85 20	STA, Zpage	GETAL 3
020D	4C 22 1C	JMP, abs	RST

As with all addition programs, we start by indicating whether we want to calculate in decimal or hexadecimal. Next the carry is set to '0', and the actual multiplication takes place. One of the two numbers is put in index register X (address \$0202). The contents of the accumulator are made \$00, and in a loop (addresses \$0206 . . . \$0209) the second number is now added to register A as many times as indicated by the index register (the first number). The answer is then stored at address \$20. (TELOP means 'add up'.)

That completes the program. This way of multiplying does, however, need some comments. The program above can hold an answer of at most 8 bits, which means the answer can never be greater than 99 (if we work in decimal) or FF (if we work in hexadecimal). This capacity can of course be extended to 16, 24, 32 etc. bits. But then we have a very slow multiplication. One pass through the loop takes $3 + 2 + 3 = 8 \mu\text{s}$. When multiplying, for example, 9×9 , the loop therefore takes $9 \times 8 = 72 \mu\text{s}$, since the loop has to be run nine times.

Now imagine that a number of four or five digits had to be multiplied. Firstly, more instructions have to be included in the loop, because more bytes now have to be added. We also can no longer use register X, because it holds at most 8 bits. So one pass through the loop already takes longer, and on top of that we now have to run the loop some 10,000 times. A modest multiplication easily runs into tens of ms! If you are programming a calculator, 10 ms for a multiplication is no problem. For series expansions, calculating graphs, etc., 10 ms is far too slow. So we must look for another method.

Another method of multiplying

When you have to work out a multiplication yourself, you proceed as follows:

```
345
123 x
----
1035
6900
34500
-----
42435
```

What exactly do you do? First you take the top number 3 times, and write the answer in a column that you will add up later. Next you write down a 0 and calculate 2×345 .

Because you had already written down the 0, you were actually calculating 2×3450 . The same applies to the last digit: you now start by writing down two zeros

We now let the computer do exactly the same. As you know, the computer works in the binary system, so we must now multiply in binary. As an example we take two simple

numbers. Larger numbers are handled in exactly the same way.

Suppose:

```
1110 (14)
1001 (9)
----- x
```

Before we start, we must first know the multiplication tables of the binary system. A pleasant bonus is that the tables are very short. The table of 0:

$$0 \times 0 = 0$$

$$1 \times 0 = 0$$

The table of 1:

$$0 \times 1 = 0$$

$$1 \times 1 = 1$$

And there are no more digits. All in all, it comes down to this: for a '0' we do not add the number, and for a '1' we do add it. We would do the multiplication as follows:

```
1110 (= $E = 14)
1001 (= $9 = 9)
----- x
  1110
 00000
000000
1110000
----- +
1111110 (= $7E = 126)
```

The computer works in the same way, except that it skips adding 0000:

```
1110
1001
----- x
  1110
1110000
----- +
1111110
```

Now we only need to work out how to translate this method into 'computer language'. To begin with, we must give the various numbers a name. As we have done before, we call the two numbers to be multiplied 'GETAL 1' and 'GETAL 2' (number 1 and number 2), and the answer 'GETAL 3'. We will leave the number of bits in each number open for now. We have the computer add up the intermediate answers, which we ourselves wrote one below the other, straight away. The addition has to be done anyway, and doing it straight away saves memory space. So the first action must be to set 'GETAL 3' to zero, so that we can later add the intermediate answers to it.

We now follow literally the procedure that we followed ourselves:

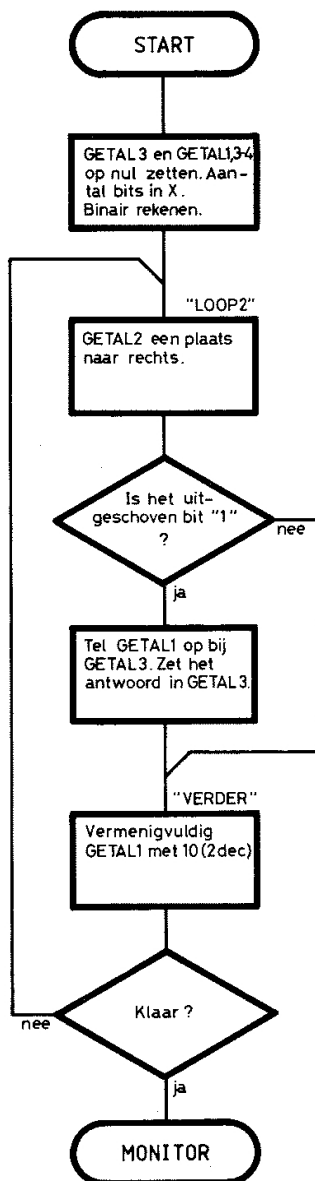
'If bit 0 of number 2 is one, add number 1 to number 3'. Bit 0 of number 2 is indeed 1, so number 3 is now '1110'.

The next instruction will be:

'If bit 1 of number 2 is one, add 10 x number 1 to number 3'.

Because bit 1 of number 1 is '0', nothing will be added to number 3. We must continue in this way until the whole number has been multiplied. You may notice a great similarity between the last two instructions. The only difference is that the first time we look at bit 0 of number 2, the second time at bit 1, and the n-th time at bit n-1.

Moreover, the first time we must add 1 x number 1, the second time 10 x number 1 (10 is a binary number), the third time 100 x number 1, and the n-th time $10^{(n-1)}$ (or in decimal $2^{(n-1)}$) x number 1.



The flowchart

To get a clear picture of the order of the various instructions, we need to make a flowchart. Fig. 1 gives the flowchart of our problem. We must first make GETAL 3 '0'.

This is because we will keep adding GETAL 1 to GETAL 3. The total sum gives the answer. To prevent the old contents of number 3 from being added to the answer, we must start by making GETAL 1 '0'. Two auxiliary registers must also be made '0'; we will come back to this. The loop must be run once for each bit of the number to be multiplied. So we must put the number of bits in reg. X. After each loop register X is decreased by one. As long as the result is not '0' we go back. Because this method only works in binary, the 'decimal mode flag' must be cleared. After this preparatory work (also called 'initializing'), the multiplication starts. First we shift number 2 one place to the right. This puts bit 0 into

the carry bit of the status register. If this bit is '0', we jump over the add instruction. If the bit is 1, 'GETAL 1' is added to the answer. In preparation for the next loop, number 1 is now multiplied by 10 (binary). Multiplying by 2 in the binary system is just as easy as multiplying by 10 in the decimal system. The number simply has to be shifted 1 place to the left:

decimal 123 x 10 = 1230
 binary 1001 x 10 = 10010
 (9 x 2 = 18)

You can see how the program runs in table 1. For each loop, this table shows the size of each number.

Table 1. In this table you can follow exactly what the microprocessor does during each loop.

Loop no.	reg X	GETAL2 becomes	carry bit	GETAL1 was	GETAL3 was	GETAL3 becomes	GETAL1 becomes
1	4	0100	1	1110	00000000	00001110	11100
2	3	0010	0	11100	00001110	00001110	111000
3	2	0001	0	111000	00001110	00001110	1110000
4	1	0000	1	1110000	00001110	01111110	11100000

The program When we come to write the program, we must decide on the number of bits. As an example we take two bytes each for numbers 1 and 2, so that 2 16-bit numbers can be multiplied. The answer (GETAL 3) then occupies at most 4 bytes. Because GETAL 1 is shifted one place to the left each time, GETAL 1 must have as much room as the answer, number 3 (see table 1). The extra room that number 1 needs must also be made '0' before multiplying (these are the auxiliary registers mentioned earlier). For the numbers we use the following memory locations:

number 1,1 00
 number 1,2 01
 number 1,3 02
 number 1,4 03

number 2,1 10
 number 2,2 11
 number 3,1 20
 number 3,2 21
 number 3,3 22
 number 3,4 23

Here we assume that the least significant digit (LSD) is in number X,1. This means that,

for example, with \$3CF5, \$F5 goes into number 1,1 and \$3C into number 1,2. We will now convert the flow diagram, block by block, into program steps.

Initializing

To begin with, number 2,1 2,2 2,3 and 2,4 must be made '00'. In the program we have done this with a loop, because this takes slightly less space than:

```
LDA, imm  $00
STA, Zpage GETAL 3,1
STA, Zpage GETAL 3,2
STA, Zpage GETAL 3,3
STA, Zpage GETAL 3,4
```

Because of the jump that has to be made each time (address \$0207), it does take slightly longer than the 'straightforward' method. But this makes no difference to the total multiplication time. At addresses \$0209 and \$020B number 1,2 and 1,4 are also made '00'. After 'CLD' and 'LDX' the preparatory work is done. (With these instructions we make sure that the addition is done in binary, and that the whole process is repeated for 16 bits (10 hexadecimal).)

LIJST 1				
0200 A2 04	START	LDX, imm	\$04	
0202 A9 00		LDA, imm	\$00	
0204 95 1F	LOOP1	STA, Zp, X	GETAL 3,1-1	
0206 CA		DEX, impl		Diverse geheugenplaatsen '0' maken.
0207 D0 FB		BNE, rel	LOOP1	
0209 85 02		STA, Zpage	GETAL 1,3	
020B 85 03		STA, Zpage	GETAL 1,4	
020D D8		CLD, impl		Binair rekenen
020E A2 10		LDX, imm	\$10	16 bitjes
0210 46 11	LOOP2	LSR, Zpage	GETAL 2,2	GETAL 2 één plaats naar rechts
0212 66 10		ROR, Zpage	GETAL 2,1	
0214 EA EA		NOP, impl		
0216 EA		NOP, impl		9×NOP, zodat eventueel de ROR-instructie gesimuleerd kan worden (zie tekst)
0217 EA EA		NOP, impl		
0219 EA EA		NOP, impl		
021B EA EA		NOP, impl		
021D 90 19		BCC, rel	VERDER	Indien C=1 : optellen
021F 18		CLC, impl		
0220 A5 00		LDA, Zpage	GETAL 1,1	
0222 65 20		ADC, Zpage	GETAL 3,1	
0224 85 20		STA, Zpage	GETAL 3,1	
0226 A5 01		LDA, Zpage	GETAL 1,2	
0228 65 21		ADC, Zpage	GETAL 3,2	
022A 85 21		STA, Zpage	GETAL 3,2	
022C A5 02		LDA, Zpage	GETAL 1,3	
022E 65 22		ADC, Zpage	GETAL 3,3	
0230 85 22		STA, Zpage	GETAL 3,3	
0232 A5 03		LDA, Zpage	GETAL 1,4	
0234 65 23		ADC, Zpage	GETAL 3,4	
0236 85 23		STA, Zpage	GETAL 3,4	
0238 06 00	VERDER	ASL, Zpage	GETAL 1,1	
023A 26 01		ROL, Zpage	GETAL 1,2	
023C 26 02		ROL, Zpage	GETAL 1,3	
023E 26 03		ROL, Zpage	GETAL 1,4	
0240 CA		DEX, impl		Is het laatste bitje geweest?
0241 D0 CD		BNE, rel	LOOP2	
0243 4C 22 1C		JMP, abs	RST	terug naar monitor

Shifting number 2 one place to the right

If GETAL 2 consisted of only one byte, this task would be just one instruction:

LSR, Zpage GETAL2

We have not come across this instruction before. The letters stand for 'Logic Shift Right'. This instruction shifts all bits of the addressed memory location one place to the right. Bit 0 is, as it were, shifted out of the memory location and kept in the carry. Bit 7 shifts to position 6 and a 0 goes into position 7. So:

before shifting:

B7 B6 B5 B4 B3 B2 B1 B0

after shifting:

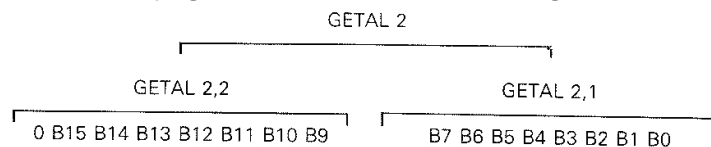
0 B7 B6 B5 B4 B3 B2 B1

B0 is in the carry bit of the status register.

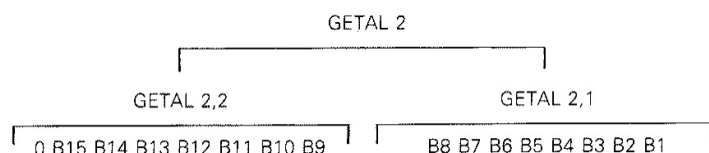
If a number is longer than one byte, the bit shifted out must be shifted into the next byte. So before shifting, GETAL 2 looks like this:



After 'LSR, Zpage GETAL 2,2' the whole thing will look like this:



Bit 8 has not been lost but, as said before, has ended up in the carry bit of the status register. We must now shift GETAL 2,1 one place to the right. This time, however, position 7 must not receive a '0' but the contents of the carry bit (B8). In addition, the bit 0 shifted out must also be kept. This is done by a second new instruction, namely 'ROR', which stands for Rotate Right. With this instruction the contents of the carry bit shift into position 7, and B0 shifts into the carry bit. So the bits 'rotate'. After the instruction 'ROR Zpage GETAL 2,1' the situation looks like this:



As you can see, the whole number has been shifted one position to the right, and B0 is in the carry bit. Unfortunately not all 6502s have the ROR instruction. The older types (before June 1976) lack ROR.

If you have an older type, you can replace

```
LSR, Zpage  GETAL 2,2  
ROR, Zpage  GETAL 2,1
```

with:

```
LDA, imm    $80  
LSR, Zpage  GETAL 2,2  
BCS, rel    $01  
ASL, accu  
LSR, Zpage  GETAL 2,1  
ORA, Zpage  GETAL 2,1  
STA, Zpage  GETAL 2,1
```

Thanks to the NOPs in list 1 (addresses \$0214 . . . \$021C) there is enough room in the program to put these instructions. We will not go into how this piece of program works, because it has nothing to do with multiplication.

Is the bit shifted out a '1'?

In the next block a decision is made. If the bit shifted out of number 2 was a '1', an addition is done; if this bit was '0', no addition is done. Because the bit shifted out is in the carry bit, we can achieve what we want with:

```
BCC, rel (address $021D)
```

This instruction makes a jump (Branch if Carry Clear) if the carry bit is '0'. The size of the jump has been chosen so that we jump exactly over the addition. If the carry bit is '1', the jump is not made, and so the addition is done. (VERDER means 'onward'.)

The 32-bit addition

Addresses \$021F . . . \$0236 contain the addition GETAL 1 + GETAL 3. The answer is again written into GETAL 3. The principle of these 'multi-byte' additions was covered in detail in the previous part of 'step by step'. It is possible to put this long series of instructions in a loop and work with Zpage,X. This would certainly save memory space. As mentioned before, however, a loop takes more time, because register X has to be changed each time and a jump back has to be made each time. But this part of the program is itself inside a loop and is executed 16 times. As a result, the extra time needed for an addition loop will also be multiplied by 16. Where saving memory space is more important than

speed, it is certainly useful to make the addition loop. The counter that keeps track of the number of bits (addresses \$020E and \$0240) must then be implemented with index register Y. The NOP instructions have been included here as a service to those who have a 'ROR-less' microprocessor. The NOP instructions make the microprocessor do nothing, but they do waste time. If this multiplication routine is going to be used seriously, it is advisable to remove the NOPs from this program. The jump size at address \$0241 must then be adjusted.

Multiply number 1 by 2

As noted earlier, this multiplication comes down to shifting GETAL 1 one place to the left. Shifting left is done in the same way as shifting right, except that we now start at the other end:

```
ASL, Zpage  GETAL 1,1  
ROL, Zpage  GETAL 1,2  
ROL, Zpage  GETAL 1,3  
ROL, Zpage  GETAL 1,4
```

Again we see 2 new instructions, namely:

ASL (Arithmetic Shift Left)
ROL (Rotate Left)

They work the same way as LSR and ROR, only in the other direction. Here the number to be shifted is 4 bytes long, so two extra ROL instructions are needed.

Done?

In this block index register X is decreased by one. As long as the result is not '0', we go back to 'LOOP2'. As soon as the result is '0', we go back to the monitor program. The instructions used speak for themselves.