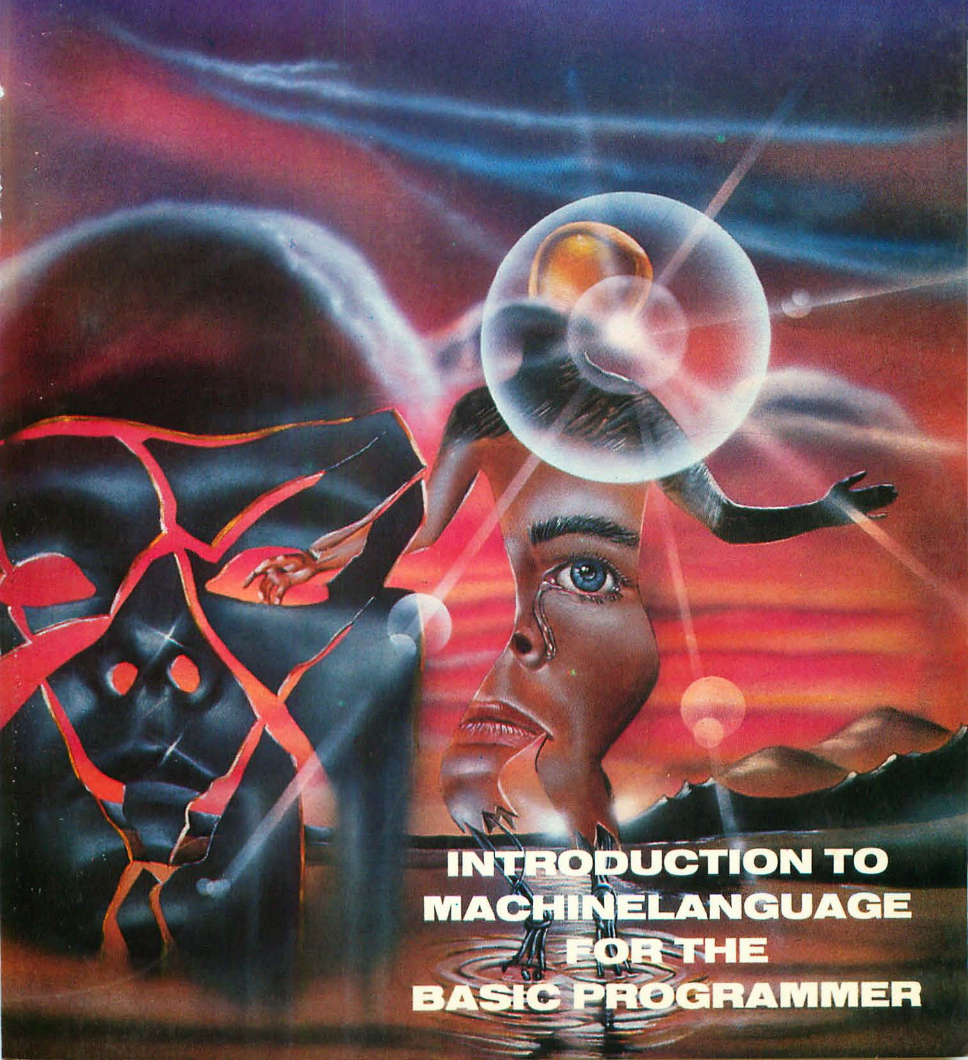
SAM D. ROBERTS

HOW TO PROGRAM YOUR ATARI® in 6502 Machinelanguage



INTRODUCTION TO MACHINELANGUAGE FOR THE BASIC PROGRAMMER

This book is an independent production of Ing. W. HOFACKER GMBH International. It is published as a service to all ATARI personal computer users worldwide.

First Edition
First Printing
October 1982 in the Federal Republic of Germany

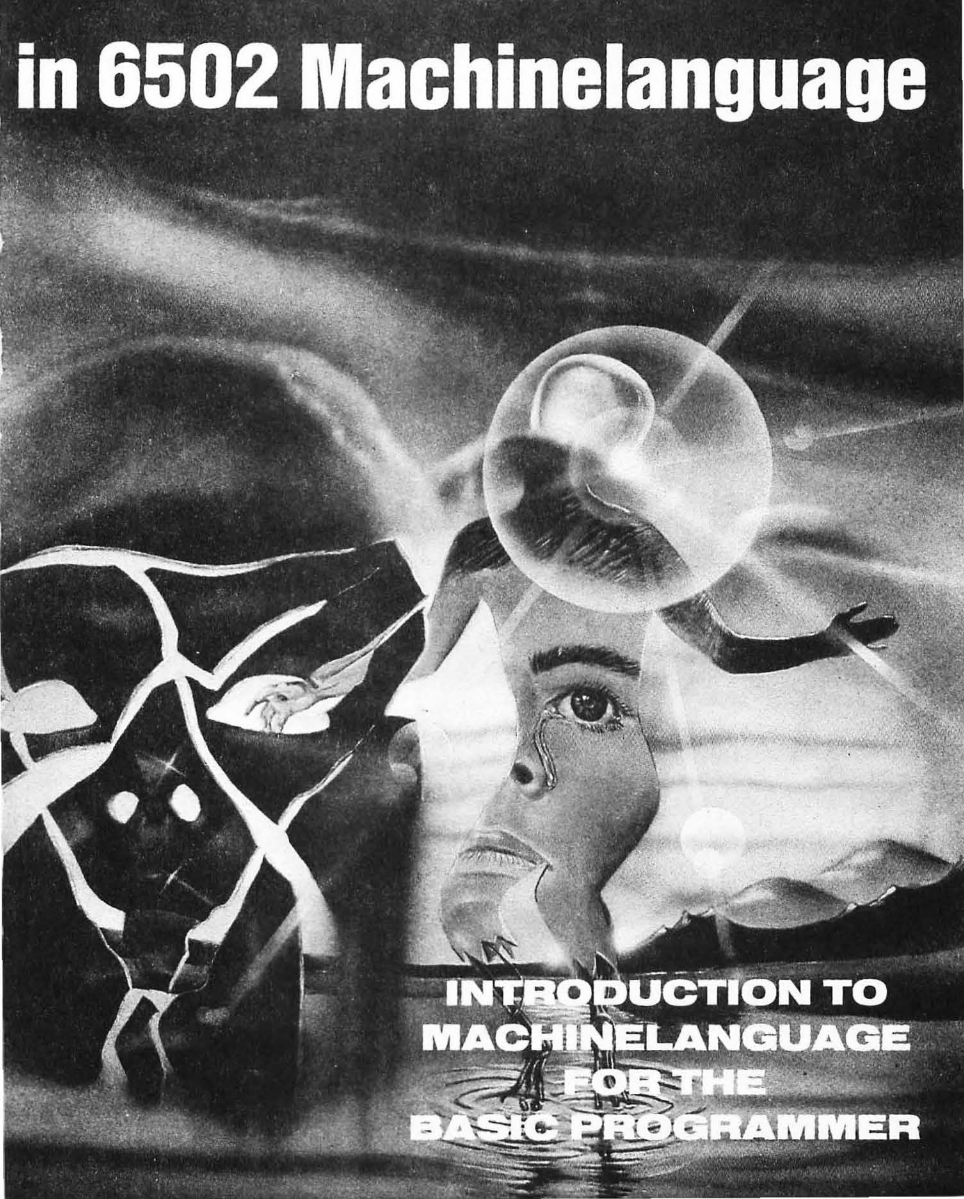
ISBN 3-92 1682-97-5

Publisher:
Ing. W. HOFACKER GmbH, Tegernseerstr. 18, D-8150 Holzkirchen, W.-Germany

US-Distributor:
ELCOMP Publishing, Inc., 53 Redrock Lane, Pomona CA 91766

SAM D. ROBERTS

HOW TO PROGRAM YOUR ATARI® in 6502 Machinelanguage



INTRODUCTION TO MACHINELANGUAGE FOR THE BASIC PROGRAMMER

PREFACE

ATARI Assembly Language Programming
Learning by using

Few features of a home computer confuse the novice computer owner more than software. Many of these new owners have studied the system manuals, they have possibly read articles or even books on microcomputers. Many of them already programmed their ATARI computer in BASIC, FORTH, PILOT or another high level language. After a while, they will find out that the language used is too slow for their needs (animation, sound, graphics, to name just a few applications). They also want to know more about the internal things happening in the computer. They are most likely aware of the ubiquitous 0's and 1's that control the computer. But how do those ubiquitous digits relate to the information displayed on the screen and to the language of the computer. How can they be put to work?

The subject of this book is to teach you how to program your ATARI computer in 6502 machine language. You may use a machine language monitor (like ATMONA-1, Monkey Wrench, the Debugger from the ATARI Editor/Assembler cartridge or the built in monitor from KDOS), to enter and start the programs listed in this book. Later on we will find out that itis too cumbersom to do the assembly by hand. We than use an assembler for our programs and we will learn how to call machine language subroutines from BASIC.

TABLE OF CONTENTS

PART 2

PART 3

PART 4

PART 5

PART 6

PART 7

PART 8

PART 9

PART 10

Machine code examples

INTRODUCTION TO NUMBER SYSTEMS

CHAPTER A

CHAPTER B

1

Part 1

Most people don't realize that BASIC commands like IF or THEN actually are sequences of commands in machine language. This introduction is meant for those who want to leave BASIC and go deeper into their computer.

The 6502 microprocessor and its commands are the subjects of this introduction. Once you understood how this microprocessor works it is not very difficult to learn another one. In this section we will talk about some rudiments.

The first thing you need is the monitor. This is not the television, but the operating system that takes control over the computer after power-up.
The monitor is very important for programming in machine-language. It contains the routines needed most, such as outputs to, and inputs from, a device. To get into the monitor you have to enter a certain command. With the APPLE II the command would be : CALL - 151 (in BASIC), or "M" after power up with OHIO C1P. The AIM 65 is in the monitor automatically after power up. The ATARI 400/800 is in the EDIT-mode, if you use the ASSEMBLER EDITOR cartridge. The samples in this booklet are written for the machine-language monitor ATMONA-1 for ATARI from ELCOMP.

Programs in machine-language work directly in the computers memory. Each command is stored at a certain address. This address is the memory location where the first statement to be executed is stored. To start a machine-language program the startaddress of that progam has to be stored in the progam counter of the microprocessor.

The statements for the microprocessor are one, two, or three bytes long. One byte is eight bits broad and, therefore, one word for a eight bit processor. The first byte contains the operation code. Figure 1 shows the different commands available on the 6502 microprocessor. The left column in that figure shows the mnemonics for the commands (assembler-code). One or two address bytes can follow the operation code. There are several ways for addressing, which will be explained later.

Examples of statements

1.
Load the accumulator with the contents of memory location $1000 ($ means : the following number is hexadecimal).

```
assembler code : LDA $1000
hex-code       : AD 00 10
```

This statement is three bytes long. With the 6502 the addresses are specified with first the lower, then the higher byte.

2.
Compare the contents of the accumulator with the contents of the very next location.

```
assembler code : CMP #$7F
hex-code       : C9 7F
```

This is a two-byte statement. The #-sign means immediate addressing. The operation referes to the memory location which immediately follows the command.

3.
Shift the contents of the accumulator to the left one position.

```
assembler-code : ASL
hex-code       : 0A
```

This is a one-byte statement, no address is needed in this case.

Notes to part 1 :

* monitor
* address
* program counter
* statement
* 1-, 2-, and 3-byte commands

Commands	symb. Code	Operation	Adressing modes													condition codes					
			IMM.	ABS	ABS,X	ABS,Y	Z0	Z0,X	Z0,Y	(IND,X)	(IND),Y	REL	IND	ACCU	IMPL	N	Z	C	1	D	V
Transport	LDA	M → A	A9	AD	BD	B9	A5	B5		A1	B1					X	X	–	–	–	–
	LDX	M → X	A2	AE		BE	A6		B6							X	X	–	–	–	–
	LDY	M → Y	A0	AC	BC		A4	B4								X	X	–	–	–	–
	STA	A → M		8D	9D	99	85	95		81	91					–	–	–	–	–	–
	STX	X → M		8E			86		96							–	–	–	–	–	–
	STY	Y → M		8C			84	94								–	–	–	–	–	–
	TAX	A → X													AA	X	X	–	–	–	–
	TAY	A → Y													A8	X	X	–	–	–	–
	TXA	X → A													8A	X	X	–	–	–	–
	TYA	Y → A													98	X	X	–	–	–	–
	TXS	X → S													9A	–	–	–	–	–	–
	TSX	S → X													BA	X	X	–	–	–	–
	PLA	S+1 → S, Ms → A													68	X	X	–	–	–	–
	PHA	A → Ms, S–1 → S													48	–	–	–	–	–	–
	PLP	S+1 → S, Ms → P													28						
	PHP	P → Ms, S–1 → S													08	–	–	–	–	–	–
arithmetic-	ADC	A+M+C → A	69	6D	7D	79	65	75		61	71					X	X	X	–	–	X
	SBC	A–M–$\overline{C}$ → A	E9	ED	FD	F9	E5	F5		E1	F1					X	X	X	–	–	X
	INC	M+1 → M		EE	FE		E6	F6								X	X	–	–	–	–
	DEC	M–1 → M		CE	DE		C6	D6								X	X	–	–	–	–
	INX	X+1 → X													E8	X	X	–	–	–	–
	DEX	X–1 → X													CA	X	X	–	–	–	–
	INY	Y+1 → Y													C8	X	X	–	–	–	–
	DEY	Y–1 → Y													88	X	X	–	–	–	–
logic-	AND	A ∧ M → A	29	2D	3D	39	25	35		21	31					X	X	–	–	–	–
	ORA	A V M → A	09	0D	1D	19	05	15		01	11					X	X	–	–	–	–
	EOR	A ⊻ M → A	49	4D	5D	59	45	55		41	51					X	X	–	–	–	–
compare-	CMP	A–M	C9	CD	DD	D9	C5	D5		C1	D1					X	X	X	–	–	–
	CPX	X–M	E0	EC			E4									X	X	X	–	–	–
	CPY	Y–M	C0	CC			C4									X	X	X	–	–	–
	BIT	A ∧ M		2C			24									7	X	–	–	–	6
branch-	BCC	BRANCH ON C=0										90				–	–	–	–	–	–
	BCS	BRANCH ON C=1										B0				–	–	–	–	–	–
	BEQ	BRANCH ON Z=1										F0				–	–	–	–	–	–
	BNE	BRANCH ON Z=0										D0				–	–	–	–	–	–
	BMI	BRANCH ON N=1										30				–	–	–	–	–	–
	BPL	BRANCH ON N=0										10				–	–	–	–	–	–
	BVC	BRANCH ON V=0										50				–	–	–	–	–	–
	BVS	BRANCH ON V=1										70				–	–	–	–	–	–
	JMP			4C									6C			–	–	–	–	–	–
	JSR			20												–	–	–	–	–	–
SHIFT-	ASL			0E	1E		06	16						0A		X	X	X	–	–	–
	LSR			4E	5E		46	56						4A		0	X	X	–	–	–
	ROL			2E	3E		26	36						2A		X	X	X	–	–	–
	ROR			6E	7E		66	76						6A		X	X	X	–	–	–
Status- Register	CLC	C=0													18	–	–	0	–	–	–
	CLD	D=0													D8	–	–	–	–	0	–
	CLI	I=0													58	–	–	–	0	–	–
	CLV	V=0													B8	–	–	–	–	–	0
	SEC	C=1													38	–	–	1	–	–	–
	SED	D=1													F8	–	–	–	–	1	–
	SEI	I=1													78	–	–	–	1	–	–
Misc.	NOP	NO OPER													EA	–	–	–	–	–	–
	RTS	RETURN F. SUB													60	–	–	–	–	–	–
	RTI	RETURN F. INT													40						
	BRK	BREAK													00	–	–	–	1	–	–

Table I

READ THIS!

PRTBYT

PROGRAMMING IN MACHINE-LANGUAGE WITH THE MICROPROCESSOR 6502

All examples are written for ATARI 400/800. They work in conjunction with the machine-language monitor ATMONA 1.

The samples use some routines from the ATARI monitor. Two examples are the output of a character to the screen, and the input of a character from the keyboard.

Some programs contain the command JSR PRTBYT. This subroutine calls a routine for output of the contents of the accumulator in the form of two hexadecimal bytes. This routine has to be entered together with the program that calls that routine. PRTBYT starts at address 1000 and is called by the OP-code 20 00 10.

The rest of the programs start at address 600. This is an unused part of memory (page 6) and may be used for short programs or for storage of data. Our examples are short so that they fit in this area.

Here is the routine PRTBYT :

```
1000:    8D 23 10    STA $1023
1003:    4A          LSR
1004:    4A          LSR
1005:    4A          LSR
1006:    4A          LSR
1007:    20 14 10    JSR $1014
100A:    AD 23 10    LDA $1023
100D:    20 14 10    JSR $1014
1010:    AD 23 10    LDA $1023
1013:    60          RTS
1014:    29 0F       AND #$0F
1016:    C9 0A       CMP #$0A
1018:    18          CLC
1019:    30 02       BMI $101D
101B:    69 07       ADC #$07
101D:    69 30       ADC #$30
101F:    4C A4 F6    JMP $F6A4
1022:    00          BRK
```

To enter the above program use the machine-language monitor ATMONA 1.

2

Part 2

2-1 Programming model of the 6502 CPU

By looking at the hardware structure of a microprocessor you get a survey of what statements it can execute. The structure of the 6502 is shown in figure 2-1. There are four eight-bit registers : the accumulator, the X-register, the Y-register, and the status register. The program counter is 16 bit long and can represent addresses from 0 to 65535.

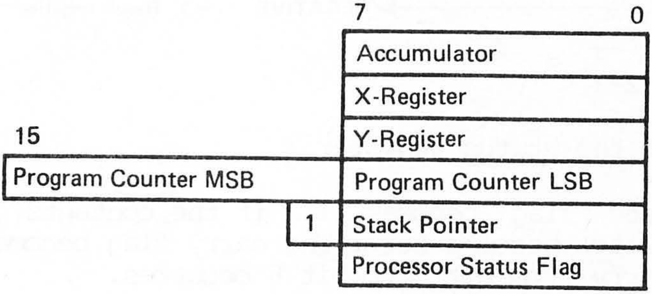


Figure 2-1
programming model of the 6502

Next is a stack pointer. The stack pointer points to a special part of the memory, the stack, at addresses $100 to $1FF. Only eight bits are used for addressing, the ninth bit always is one.

What are all these registers for ?

The main register is the accumulator. This is where all calculations are executed and the results of all calculations are stored. For addressing, one of the index registers may be used. These registers can be used as counters. For example the statement

INX increments the contents of the X-register by one. The index register can also be used to indicate addresses. These features will be used in later sample programs.

The status register indicates the present status of the processor. Each bit marks a result of an operation.

Figure 2-2

bits of the status register

The zero flag becomes 1, if the contents of the accumulator becomes zero. The carry flag becomes 1, if a carry from bit 7 to bit 8 occurres.

The right column of figure 1 shows which operations affect the bits in the status register (X indicates change possible). For example a LDA statement can change bits N and Z; the statement STA can't change any bit of the status register.

The stackpointer points to a free area in the stack.

You can store the contents of the accumulator there with PHA (push accumulator; one byte statement) then the stackpointer will be set to the next memory location. PLA (pull accumulator) sets the pointer back one location. At this time the contents of that location will be transfered to the accumulator.

Note : the top of the stack is address $1FF. The stack builds up to address $100. Another important task of the stack is to hold the current address in case of a jump to a subroutine. At the return from the subroutine this address is transferred back to the program counter. The program counter always holds the address of the command to be executed next. Only jump-instructions change the contents of the program counter.

Figure 2-3 shows all commands available for transferring data between the registers and memory. As you can see the 6502 has no command for transferring data between the registers, or to exchange the contents of X- and Y-register as is possible with other processors.

If you know how to program one processor and wish to program another one, you should study the logical structure, concerning the effects of the commands.

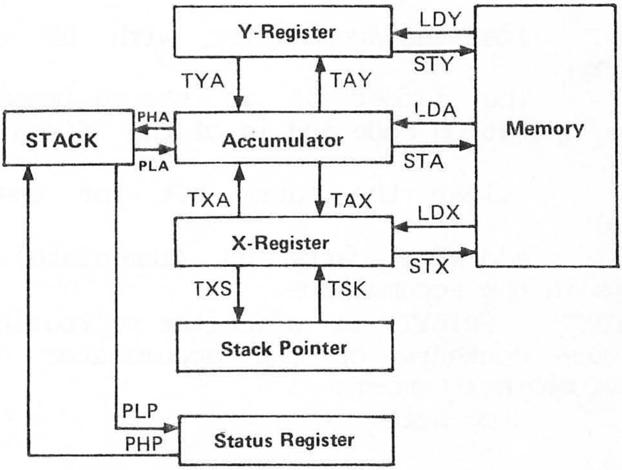


Figure 2-3

Transfer of data between registers and memory

2-2

A first example and the paper-pencil-method.

The addition of two numbers is quite simple in a higher programming language :

```
     10 A=5                   LDA # $05
     20 B=3                   CLC
     30 C=A+B                 ADC # $03
     40 PRINT C               JSR  PRTBYT
     50 END                   BRK
```

To do the same job in machine language it is necessary to answer the following questions first :

Where are the numbers stored ?
Are the numbers of type fixed point or floating point ?
Is there a routine existing in the monitor, which prints the contents of a memory location ?

Here is the program in machine-language :

LDA #$05 load the accumulator with 05 (direct addressing).
The number 05 is stored immediately after the operation code and is of the fixed point type.
CLC clear the carry bit for the next operation
ADC #$03 add with carry 03 (immediate). The result is in the accumulator.
JSR PRTBYT PRTBYT is a monitor subroutine that prints the contents of the accumulator on the screen as two hex-numbers
BRK stop here

Figure 2-4 shows a survey of the memory. On the left side are the addresses in decimal and on the right side they are in hexadecimal form. The addresses from 0 to $400 represent 1k of memory. The addresses from $1000 to $2000 represent 4k. Now we want to translate the program into machine language by using the paper and pencil method. This is the lowest level of programming, but it is useful in learning the programming in machine language.

The first problem is where to start the program. On principle the program can start anywhere in memory. There are however two certain areas which you should not use. First is the zero-page, a very useful area with simplified addressing, second is the stack. (remember that the stack is used by the processor itself !). For these reasons the addresses from 0 to $1FF are not available.

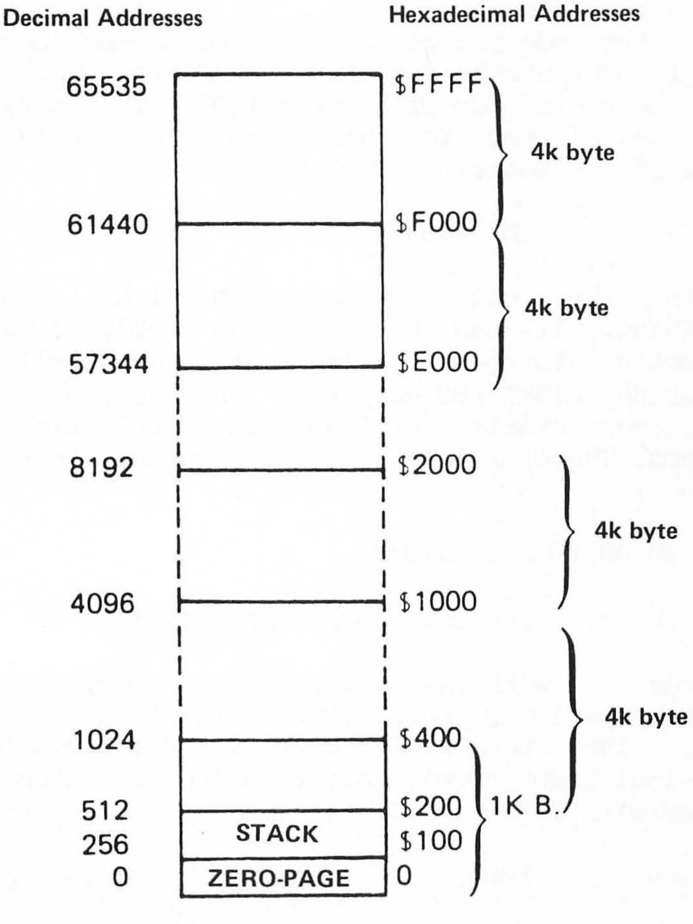


Figure 2.4: Decimal and hexadecimal addressing of a 64 k byte memory

Let's place our program at $600.
Now we can translate the first command. If you look at the table you will find that LDA has the code A9. Adjacent to that the first line looks as follows :

```
$0600 A9 05    LDA #$05
```

A9 is the operation code and 05 is the number which follows immediately. This command is two bytes long. The next line is at $0602.

```
$0602 18       CLC
```

18 is the code for clear carry. It can be found in table 1 under status register statements. The line after that is add with carry (ADC). The carry bit has to be cleared in this case, otherwise the result of the addition could be wrong.

```
$0603 69 03    ADC #$03
```

69 is the code for addition with immediate addressing. It can be found in table 1 under arithmetic statements. The next command calls the subroutine PRTBYT for output to the screen. This subroutine starts at address $1000 with our programs. Therefore the line for output looks as follows :

```
$0605 20 00 10 JSR PRTBYT
```

20 is the code for JSR (JUMP SUBROUTINE).

Remember : with the 6502 processor you first have to enter the lower byte (LSB, least significant byte), then the higher byte of the address (MSB, most significant byte). After which we stop the program with :

```
$0608 00       BRK
```

Most computers jump back into the monitor after they hit a BRK-instruction. The whole program looks like this for the ATARI 400/800 :

```
$0600 A9 05    LDA #$05
$0602 18       CLC
$0603 69 03    ADC #$03
$0605 20 00 10 JSR PRTBYT
$0608 00       BRK
```

Thus a dump of these locations looks as follows :

```
$0600: A9 05 18 69 03 20 00 10
$0608: 00
```

At this point we will not talk about how to enter that program, rather we will discuss different techniques of addressing. Let's assume that there is the same job, but the two numbers are stored in two zero-page locations. The number 5 is stored at location $10 and the number 3 is stored at location $11. Our program would look as follows :

```
$0600  A5  10     LDA $10 ;load the accumulator with
the contents of location $10
$0602 18       CLC       ;clear carry bit
$0603 65 11    ADC $11 ; add contents of location
$11
$0605 20 00 10 JSR PRTBYT ;output
$0608 00       BRK       ;stop
```

A5 is the code for LDA with the contents of a zero-page location.

In the next example we assume, that the numbers are stored anywhere in memory, for example at $200A and at $3005. The program would look as follows :

```
$0600 AD 0A 20 LDA  $200A  ; load the contents of
location $200A
$0603 18       CLC         ;clear carry bit
$0604  6D  05  30  ADC  $3005  ; add the contents of
location $3005
$0607 20 00 10 JSR PRTBYT;output to screen
$060A 00       BRK         ;stop
```

In this case AD is the code for LDA with the contents of an absolute address. The code for ADC the contents of an absolute address is 6D. This last program is two bytes longer than the prior one. If possible, in order to shorten the program, the zero-page should be used for auxiliary cells.

Notes to part 2:

* programming model of the 6502
* CPU register
* zero-page addressing
* absolute addressing

3

Part 3

In part 2 we talked about a program which flows off straight. In this part we will talk about programs which contain branches.

3-1 Programs with branches

There are many programs which contain loops that have to be traveled through until a certain condition becomes complied with. As an example the condition can be whether the contents of a memory location or a register is equal to zero, or whether a number in a register is greater than, or equal to, or smaller than, the contents of a memory location. The bits in the status register are influenced by operations or comparisons (see figure 2-2). Whether branch commands are executed or not, depends on the status of certain bits. An example of this is a delay loop. The contents of the X-register is decremented until it is zero.

Here is the program for that :

```
LDX #$0A  ;load the X-register with A0
M DEX     ;decrement X-register by one
BNE M     ;jump back to M, if not zero
BRK       ;stop program, if X-register=0
```

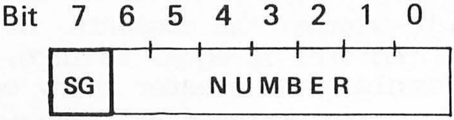
In machine-language it looks as follows :

```
0600 A2 A0   LDX #$A0
0602 CA    M DEX
0603 D0 --   BNE M
0605 00      BRK
```

Location 0604 has been left open. The number of bytes the program has to jump back belongs to there.

The branch commands use the so-called relative addressing. This means the current contents of the program counter becomes increased or decreased by a certain number. The program then continues at the new address. What is the current contents of the program counter ? The program counter of the 6502 always points to the next command; in our example this is the BRK-command at location 0605. To get back to location 0602 we have to decrement the program counter by 3. Therefore the hexadecimal equivalent of -3 has to be stored at location 0604. How are negative numbers displayed ?

Bit 7 is used to determine, whether a number is positive or negative.



If bit 7 is 1, then the number is negative, if bit 7 is zero, then the number is positive.

Positive numbers are :

0 = $00 = %0000 0000
1 = $01 = %0000 0001
2 = $02 = %0000 0010
.

127 = $7F = %0111 1111

Negative numbers are described by the complement on two. To complement a number means to turn around all bits of that number : ones become zeros, zeros become ones. With the complement on two, one is added after that. For example the number -1 :

+1 = %0000 0001 ; the complemented number :
%1111 1110
addition of 1 results in : %1111 1111 = $FF

Negative numbers are :

```
   -1 = $FF = %1111 1111
   -2 = $FE = %1111 1110
   -3 = $FD = %1111 1101
      .

 -128 = $80 = %1000 0000
```

Thus relative branches can range from -128 to +127.

Complete program :

```
0600 A2 A0    LDX #$A0
0602 CA     M DEX
0603 D0 FD    BNE M
0605 00       BRK
```

You also can use the following tables :

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

Table 3-1 Forward branch

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

Table 3-2 Backward branch

Most mistakes happen with the calculation of bytes for relative jumps, when assembling by hand !

3-3 Comparisons

Comparisons always happen between a register (accumulator, X- or Y-register) and a memory location. Bits N (negative), Z (zero), and C (carry) are influenced by comparisons.

Figure 3-3 shows how :

Comparison	N	Z	C
A, X, Y < M	1*	0	0
A, X, Y = M	0	1	1
A, X, Y > M	0*	0	1

* comparison with twos complement

Figure 3-3 Flags with comparisons

If the contents of the accumulator (or X-register, Y-register) is smaller than the contents of a memory location, then the zero flag and the carry flag become 0. For these two flags the numbers can be between 0 and 255. For the N flag the numbers are compared in the twos complement. These numbers can be from -128 to +127.

For example :

The contents of the accumulator is $FD, the contents of a memory location is 00. A comparison A > M (252-00) causes C to become 1 and Z to become 0. Here are different possibilities to branch :

```
A <  M   BCC   LABEL
A <= M   BCC   LABEL
         BEQ   LABEL
A =  M   BEQ   LABEL
A >= M   BCS   LABEL
A >  M   BEQ   NOT LABEL
         BCS   LABEL
```

The following program is a simple example for comparisons and branches. We want to input a character from the keyboard and check whether or not it is a hexadecimal number (0-9, A-F). If the character is hexadecimal, then we want to store it in location INP with address $FF. If not, we want to leave the program ($00 in INP).

For the input we use subroutine GETCHR, which is included in most monitors. This subroutine checks whether or not a key is pressed. If a key is pressed, the program returns from the subroutine with the ASCII character in the accumulator.

Figure 3-4 shows the ASCII characters

LSD	MSD	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SP	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	•	>	N	↑	n	~
F	1111	SI	VS	/	?	O	←	o	DEL

ASCII characters

```
0600:   A9 00       LDA   #$00
0602:   85 FF       STA   $FF
0604:   20 DD F6    JSR   $F6DD
0607:   C9 30       CMP   #$30
0609:   90 13       BCC   $061E
060B:   C9 47       CMP   #$47
060D:   B0 0F       BCS   $061E
060F:   C9 3A       CMP   #$3A
0611:   90 07       BCC   $061A
0613:   C9 41       CMP   #$41
0615:   90 07       BCC   $061E
0617:   18          CLC
0618:   69 09       ADC   #$09
061A:   29 0F       AND   #$0F
061C:   85 FF       STA   $FF
061E:   00          BRK
```

Figure 3-5 program ASCII HEX

Try to assemble the program by hand and calculate the jumps. This is a very good mental exercise. Compare your branch statements with those in the program before you start the program.

Notes to part 3 :

- program branch
- positive and negative numbers
- relative addressing
- comparisons

4

Part 4

In this section we will talk about the use of subroutines. Subroutines are independent parts of programs. They are called by the statement JSR (JUMP SUBROUTINE). With RTS (RETURN FROM SUBROUTINE) you return to the main program.

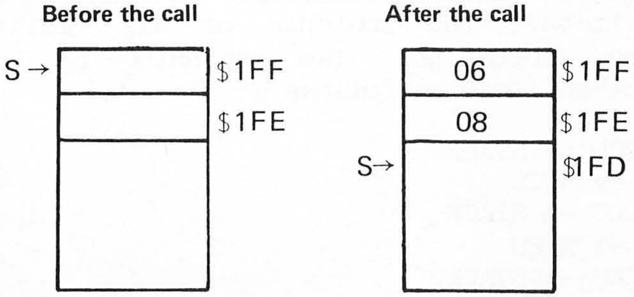
4-1 How to call a subroutine

As an example we use the instruction JSR GETCHAR from the program ASCII HEX. (GETCHAR = $F6DD on the ATARI) The first lines there are :

```
0600:    A9 00        LDA    #$00
0602:    85 FF        STA    $FF
0604:    20 DD F6     JSR    $F6DD
0607:    C9 30        CMP    #$30
```

Location 0604 contains the command for jump to subroutine. With the execution of this statement the address of the command to be executed after that (decremented by one) is stored in the stack.

The stack


Before the call
After the call

The stack is a defined part of memory of 6502 sytems. The TOS (top of stack) is at address $1FF. The stack pointer always points to the next available location in the stack.

It is possible to jump from one subroutine into another one. Figure 4-3 shows the model for that.

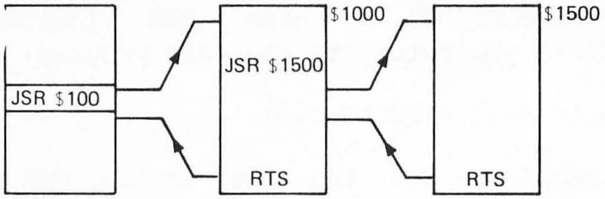


Figure 4-3 nested subroutines

The stack could hold up to 128 return addresses of subroutines at a time, but you will never need that many.

4-2 Saving the contents of registers

Most subroutines change the contents of the registers. If these contents are needed later (after RTS), they have to be saved.
This can be done either in the main program or in the subroutine. If you know what registers are changed by the subroutine, then you can save the contents at an unused location. The easiest way though, is to save the contents of all registes within the subroutine. The beginning of that subroutine then looks as follows :

```
    PHA  ;ACCU -> STACK
    TXA  ;X -> ACCU
    PHA  ;ACCU -> STACK
    TYA  ;Y -> ACCU
    PHA  ;ACCU -> STACK
```

Prior to the RTS command, you have to restore the old contents of the registers. The end of the subroutine will look as follows :

```
   PLA  ;LOAD Y
   TAY  ;
   PLA  ;LOAD X
   TAX  ;
   PLA  ;LOAD ACCU
   RTS  ;JUMP BACK
```

The contents of the registers could also be stored in auxiliary locations instead of the stack.

4-3 Exchange of data between main program and subroutine

There are three ways to exchange data between main program and subroutine.

1. Exchange via the registers. For example most keyboard input routines have the character in the accumulator at the return.

2. Exchange via the stack. This technique is used often when machine language programs are used together with high level languages (for example PASCAL).

3. The main program and the subroutine use a common memory area for the data.

The method you should use depends on the problem to be solved. If the whole program is written by one programmer, then he will use the method he likes best. If more than one programmer works together then they have to arrange the kind of exchange.

Advantages with the use of subroutines :
Longer programs become split into smaller parts. The shorter parts are easier to understand and debugging becomes easier. You can build up a library of subroutines and can use these subroutines later.

4-4 Indirect jumps and indirect jumps to subroutines.

```
SPECL:  LDA     CART            ;CHECK FOR RAM OR CART
        BNE     ENSPEC          ;GO IF NOTHING OR MAYBE RAM
        INC     CART            ;NOW DO RAM CHECK
        LDA     CART            ;IS IT ROM?
        BNE     ENSPEC          ;NO
        LDA     CARTFG          ;YES,
        AND     #$80            ;MASK OFF SPECIAL BIT
        BEQ     ENSPEC          ;BIT SET?
        JMP     (CARTAD)        ;YES, GO RUN CARTRIDGE

        CHECK FOR AMOUNT OF RAM
```

This is an indirect jump.

```
3758  F23F  AD FC BF
3759  F242  D0 12
3760  F244  EE FC BF
3761  F247  AD FC BF
3762  F24A  D0 0A
3763  F24C  AD FD BF
3764  F24F  29 80
3765  F251  F0 03
3766  F253  6C FE BF
3767
3768
3769
3770
```

5

Part 5

5-1 Indexed addressing

Example for indexed addressing :
We have stored data (numbers and letters) at memory locations $1000 - $101F. We now want to transfer this data to another area starting at $2000. This could be done by the following program :

```
   LDA $1000
   STA $2000
   LDA $1001
   STA $2001
   LDA $1002
   STA $2002
     .
     .
     .
   LDA $101F
   STA $201F
```

Please take note!

For DISK systems use $2B00 instead of $1000, in order to avoid overlapping with DOS.

This program is long and tedious. Six bytes are consumed for the transfer of one byte, which means the whole program is 32*6 = 192 bytes long. With indexed addressing this program becomes short and simple. With the statement LDA $1000,X you load the accumulator with the contents of the memory location whose address is the sum of address $1000 and the contents of the X-register.
For example :
If X=1, the contents of location $1001 will be stored in the accumulator;
If X=2, the contents of location $1002 will be stored in the accumulator.

It is also possible to use the Y-register. The statement then would be : LDA $1000,Y.

Here is the program :

```
0600 A2 00        LDX #$00
0602 BD 00 10 M LDA $1000    ;($1000) -> A
0605 9D 00 20     STA $2000,X ;(A) -> $2000
0608 E8           INX
0609 E0 20        CPX #$20    ;(X) = $20 ?
060B D0 F5        BNE M       ;CONTINUE, IF NOT
060D 00           BRK
```

Figure 5-1

First the X-register is loaded with zero. After that the accumulator is loaded : LDA $1000,X then the contents are stored at $2000,X. INX increments the X-register. It is then checked, to see whether all data has been transferred already. We want to transfer the contents of locations $1000 - $101F. The first location that should not be tranfered is $1020. If the contents of the X-register became $20 after INX, the program should stop.

In the comment above $1000 means the address of that location; ($1000) means the contents of that location.

Both index registers are 8 bit long. For that reason it is possible to index from 0 to 255. Thus we can transfer a maximum of 256 bytes with this method. For the transfer of larger areas we have to use a different technique which will be discussed later.

Here is another example :

We want to exchange the contents of locations $1000 with $10FF, $1001 with $10FE, $1002 with $10FD , etc. (figure 5-2).

First we load X with 0 and Y with FF. Then we load the contents of $1000 and store it in the stack. After that we load the contents of $10FF and store it at $1000 and next we store the value in the stack at $10FF. Lastly the Y-register is decremented and the X-register is incremented. The exchange is done when X = $80.

```
0600 A2 00        LDX #$00
0602 A0 FF        LDY #$FF      ;FF -> Y
0604 BD 00 10 M LDA $1000,X   ;($1000+X) -> A
0607 48           PHA           ;(A) -> STACK
0608 B9 00 10     LDA $1000,Y   ;($1000+Y) -> A
060B 9D 00 10     STA $1000,X   ;(A) -> $1000+X
060E 68           PLA           ;(STACK) -> A
060F 99 00 10     STA $1000,Y   ;(A) -> $1000+Y
0612 88           DEY           ;(Y)-1 -> Y
0613 E8           INX           ;(X)+1 -> X
0614 E0 80        CPX #$80      ;READY ?
0616 D0 EC        BNE M
0618 00           BRK
```

Figure 5-2

The effective address with indexed addressing is the sum of the programmed address plus the contents of the index register used. The carry flag is noted with these calculations. (The carry flag will be set, if a carry appears with the calculations). With X = $FF the contents of the accumulator will be stored at $11DF, with the command STA $10E0,X.

The 6502 has two more ways of addressing, which consist of indirect and indexed addressing.
Note : The final address with indirect addressing is not the programmed address, but contents of that address. For example : JMP ($2000) means a jump to $3AFF, if the contents of $2000 and $2001 are $3AFF.

5-2 Indexed indirect addressing

With this kind of addressing the programmed address always is an address of the zero page, with the index register always the X-register. For example LDA ($10,X).
The final address can be calculated by adding the contents of the X-register to $10. The contents of this and the following address is the effective address.

Example :
Contents of locations $0E - $15

(0E) = FF
(0F) = 0F
(10) = 00
(11) = 11
(12) = 2F
(13) = 30
(14) = 00
(15) = 47

If X = 0, then LDA ($10,X) loads the contents of location $1100; if X = 2, then LDA ($10, X) loads the contents of $302F, X = 4 causes the contents of $4700 to be loaded. No attention is payed to a carry occurring during the calculation of the address. For this reason the contents of location $0FFF will be loaded, if X = $FE.

5-3 Indirect indexed addressing

With this kind of addressing the programmed address is in the zero page also. Only register Y can be used as an index register in this case. Example : STA ($10),Y.
To find out the final address, add the contents of locations $10 and $11 to the contents of register Y.
Example :

($20) = 3E
($21) = 2F

If Y = 0, then contents of the accumulator would be stored at location $2F3E.

The last two addressing modes are used mainly as indirect addressing, with X = 0 respectively Y = 0. It then follows that LDA ($10,X) means : load the accumulator with the contents of the memory location, whose address is stored in $10 and $11.
Analogous with the statement LDA ($10),Y if Y = 0.
If the contents of these addresses are changed, you can load the accumulator with the contents of different locations. We will use this technique to do a blocktransfer of not just 256, but 4k byte from $1000 to $2000.

```
0600 A2 00     LDX #$00     ;0 -> X
0602 86 10     STX $10      ;(X) -> LO BYTE START
0604 86 12     STX $12      ;(X) -> LO BYTE DESTINATION
0606 A9 10     LDA #$10     ;$10 -> A
0608 85 11     STA $11      ;(A) -> HI BYTE START
060A A9 20     LDA #$20     ;$20 -> A
060C 85 13     STA $13      ;(A) -> HI BYTE TARGET
060E A1 10   M LDA ($10,X)  ;(($10)) -> A
0610 81 12     STA ($12,X)  ;(A) -> ($12)
0612 E6 10     INC $10      ;($10)+1 -> $10
0614 E6 12     INC $12      ;($12)+1 -> $12
0616 D0 F6     BNE M        ;CONTINUE, IF <> 0
0618 E6 11     INC $11      ;ELSE ($11)+1 -> $11
061A E6 13     INC $13      ;($13)+1 -> $13
061C A5 11     LDA $11
061E C9 20     CMP #$20
0620 D0 EC     BNE M
0622 00        BRK
```

```
0600    A2 00 86 10 86 12 A9 10
0608    85 11 A9 20 85 13 A1 10
0610    81 12 E6 10 E6 12 D0 F6
0618    E6 11 E6 13 A5 11 C9 20
0620    D0 EC 00 00 00 00 00 00
0628    00 00 00 00 00 00 00 00
```

Figure 5 - 3

In this program first the addresses for START ($10, $11) and DESTINATION ($12, $13) are defined. Second we load the accumulator with the contents of $1000 by LDA ($10,X) and store it at $2000 with STA ($12, X). Then we increment $11 and $13 by 1 until we reach the first address not to be moved.

Try the following two programs as an exercise :

1. Program FILL. A part of memory with the start address in $10, $11 and the end address in $12, $13 is to be filled with the hex number, which is stored in $14.

2. Program MOVE. A block of data (start address in $10, $11; end address in $12, $13) should be moved to another area (start address in $14, $15). This block may be at any location, even within the area of the block to be moved itself. This is not possible by the techniques used before.

Notes to part 5 :

* indexed addressing
* indexed indirect addressing
* indirect indexed addressing
* transfer of data within memory

6

Part 6

In this chapter we will talk about the input of data (characters, numbers) into the computer. The data should be entered with the keyboard. All computers with a keyboard are equipped with a subroutine for the input of a character from the keyboard. Most times this routine is called GETCHR. Usually the ASCII code or a similar code (for example ATASCII on the ATARI) is used with these characters. An 'A' in the ASCII code for instance is $41. This coding is used, for example, with the C1P and the PET. The APPLE computer uses $C1 (all normal displayed characters have bit 8 = 1). It follows that you have to be careful if you want to transfer machine language programs from one computer to another one !
With the C1P a check, whether 'A' was pressed looks as follows :

```
JSR GETCHR        (ATARI also)
CMP #$41
```

With the APPLE the same would look as follows :

```
JSR GETCHR
CMP #$C1
```

If the input of data is used very often, then a 'menu' is sometimes used. This technique, that you will know from BASIC, is possible also in machine-language. A text is displayed on the screen and the program waits for an input from the keyboard. It then branches depending on the input. We will show the whole program in a flowchart. A flowchart explains the structure of a program through the use of graphic symbols.

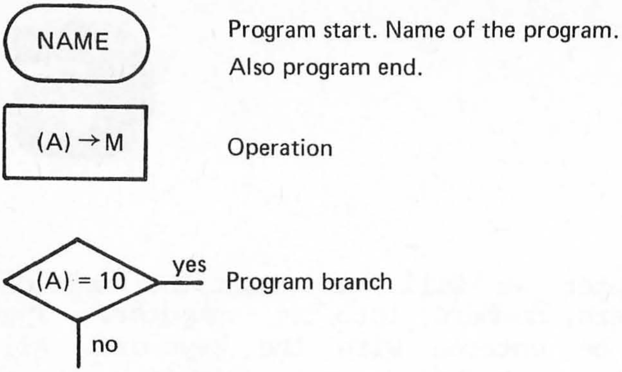


Figure 6-1 elements of a flowchart

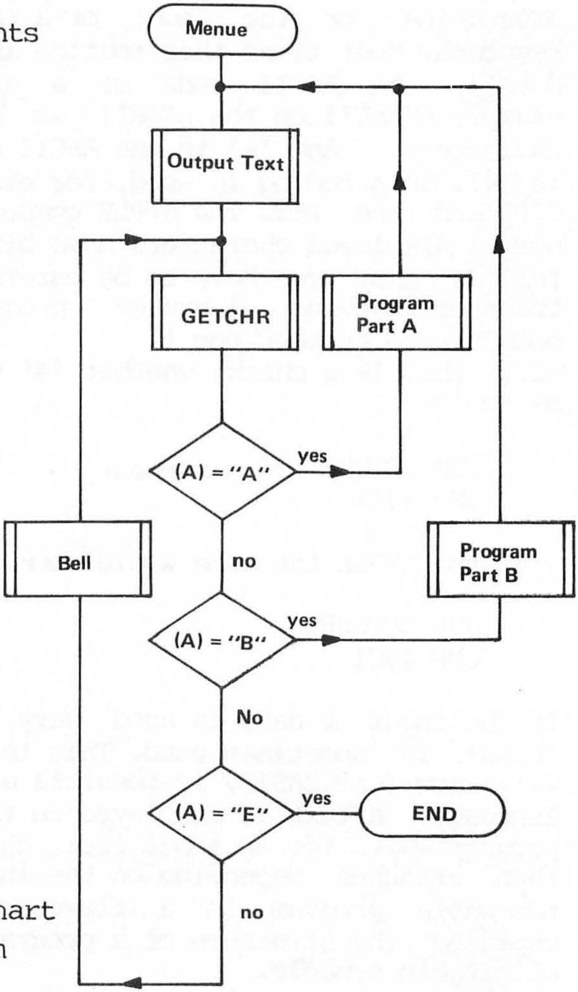


Figure 6-2 Flowchart of a menu program

The flowchart in figure 6-2 shows the structure of our program. The program first prints the text and then waits for a key to be pressed. If A, B, or E has been pressed, the program branches to the matching part. If another key has been pressed, the computer will beep and wait for another input.
This may sound simple to you, but a menu always should consider these two things :
1. The end of the program should be layed down. This means a stop of the program other than with RESET or switching off should be possible.
2. Input errors should be tied up; a warning should appear on the screen or an acustic sign (bell) should mark the error.

Here is the program.
First the screen is cleared, then the text is printed. The text is stored at memory locations starting at $0640 and is printed by the subroutine TXTOUT.
The listing contains a few commands which are not CPU statements. These pseudo statements are for the assembler. We will talk about pseudo opcodes later.

HEX-DUMP of the MENUE-program

```
0600    A97D20A4F6203306    )   $v 3F
0608    A99B20A4F6A90020    ) [ $v)@
0610    DDF6C941D0062064    ]vIAPF d
0618    061890E9C942D006    FXPiIBPF
0620    2073061890DFC945     sFXP_IE
0628    D00100A9FD20A4F6    PA@)   $v
0630    1890D2A99B20A4F6    XPR)[ $v
0638    A240A0062085F360    "@ F Es`
0640    50524F4752414D20    PROGRAM
0648    2841292020505244F   (A)  PRO
0650    4752414D20284229    GRAM (B)
0658    2020454E44452020      ENDE
0660    2845299BA2788A941   (E)["x)A
0668    86FF20A4F6A6FFCA    F  $v& J
0670    D0F460A278A94286    Pt`"x)BF
0678    FF20A4F6A6FFCAD0      $v& JP
0680    F460000000000000    t`@@@@@@
0688    0000000000000000    @@@@@@@@
```

Source Code for the MENUE-program.
Note! This is ATARI Editor/Assembler cartridge syntax

```
0000        10                *=  $600
F385        15 PUTLIN        =      $F385
F6DD        20 GETCHR        =      $F6DD
F6A4        30 EOUTCH        =      $F6A4
0600 A97D   40 MENU           LDA      #$7D
0602 20A4F6 50                JSR      EOUTCH
0605 203306 60 MENU1          JSR      TXTOUT
0608 A99B   70                LDA      #$9B
060A 20A4F6 80                JSR      EOUTCH
060D A900   85                LDA      #$00
060F 20DDF6 90                JSR      GETCHR
0612 C941   0100                CMP      #$41
0614 D006   0110                BNE      MENU2
0616 206406 0120                JSR      A0
0619 18     0130                CLC
061A 90E9   0140                BCC      MENU1
061C C942   0150 MENU2          CMP      #$42
061E D006   0160                BNE      MENU3
0620 207306 0170                JSR      B
0623 18     0180                CLC
0624 90DF   0190                BCC      MENU1
0626 C945   0200 MENU3          CMP      #$45
0628 D001   0210                BNE      MENU4
062A 00     0220                BRK
062B A9FD   0230 MENU4          LDA      #$FD
062D 20A4F6 0240                JSR      EOUTCH
0630 18     0250                CLC
0631 90D2   0260                BCC      MENU1
            0270 ;
0633 A99B   0275 TXTOUT         LDA      #$9B
0635 20A4F6 0276                JSR      EOUTCH
0638 A240   0280                LDX      #$40
063A A006   0290                LDY      #$06
063C 2085F3 0320                JSR      PUTLIN
063F 60     0330                RTS
0640        0340                *= $0640
0640 50     0350                .BYTE"PROGRAM (A) "
0641 52
0642 4F
0643 47
```

```
0644 52
0645 41
0646 4D
0647 20
0648 28
0649 41
064A 29
064B 20
064C 20
064D 50     0360               .BYTE"PROGRAM  (B)"
064E 52
064F 4F
0650 47
0651 52
0652 41
0653 4D
0654 20
0655 28
0656 42
0657 29
0658 20
0659 20
065A 45     0370               .BYTE"ENDE   (E)"
065B 4E
065C 44
065D 45
065E 20
065F 20
0660 28
0661 45
0662 29
0663 9B     0380               .BYTE$9B
0664 A278   0390 A0            LDX     #120
0666 A941   0400 AA            LDA     #$41
0668 86FF   0405               STX     $FF
066A 20A4F6 0410               JSR     EOUTCH
066D A6FF   0415               LDX     $FF
066F CA     0420               DEX
0670 D0F4   0430               BNE     AA
0672 60     0440               RTS
0673 A278   0450 B             LDX     #120
0675 A942   0460 BB            LDA     #$42
0677 86FF   0465               STX     $FF
```

```
0679 20A4F6 0470                   JSR        EOUTCH
067C A6FF   0475                   LDX        $FF
067E CA     0480                   DEX
067F D0F4   0490                   BNE        BB
0681 60     0500                   RTS
0682        0510                   .END
```

Figure 6-3 A menu program

Notes to part 6:
* input of text
* logic flowchart
* elements of a logic flowchart

Differences between the ATARI Editor/Assembler Cartrigde and ATAS-1 and ATMAS-1

To explain the difference of some mnemonics of the ATARI Editor/Assembler cartridge and the Editor/Assembler and ATMAS -1 from ELCOMP Publishing we will show you the program in ATMAS or ATAS syntax as follows:

Instead of the Asterik the ATAS uses the pseudo op-code ORG (see first line).

Another difference is that the ATAS is screen oriented (no line numbers needed). Instead of the equal sign ATAS uses EQU.

Additionally ATAS allows you the pseudo op-code EPZ: Equal Zero.

There is also a difference in using the mnemonics regarding storage of strings within the program.

ATARI		ELCOMP
– BYTE "STRING"	=	ASC " STRING"
– BYTE $	=	DFB $ (Insertion of a byte)
– WORD	=	DFW (Insertion of a word Lower byte, higher byte)

The end of string marker of the ATARI 800/400 output routine is hex 9B.

In the listing you can see, how this command is used in the two assemblers:

ATARI Assembler: –.BYTE $9B
ATMAS from ELCOMP – DFB $9B

Depending on what Editor/Assembler from ELCOMP you use, the stringoutput is handled as follows:

1. ATAS 32K and ATAS 48K Cassette Version

```
   LDX # TEXT
   LDY # TEXT/256
TEXT ASC " STRING"
   DFB$9B
```

2. ATMAS 48K

```
   LDX # TEXT:L
   LDY # TEXT:H
TEXT ASC "STRING"
   DFB $9B
```

There is also a difference between other assemblers and the ATAS-1 or ATMAS-1 in the mnemonic code for shift and relocate commands for the accumulator.
(ASL A = ASL) = 0A
(LSR A = LSR) = 4A
ROL A = ROL = 2A
ROR A = ROR = 6A

Menu program from page 34 in ATAS syntax

```
                      ORG $0600
             PUTLIN   EQU $F385
             GETCHR   EQU $F6DD
             EOUTCH   EQU $F6A4
0600: A97D   MENU     LDA #$7D
0602: 20A4F6          JSR EOUTCH
0605: 203306 MENU1    JSR TXTOUT
0608: A99B            LDA #$9B
060A: 20A4F6          JSR EOUTCH
060D: A900            LDA #$00
060F: 20DDF6          JSR GETCHR
0612: C941            CMP #$41
0614: D006            BNE MENU2
0616: 206406          JSR A0
0619: 18              CLC
061A: 90E9            BCC MENU1
061C: C942   MENU2    CMP #$42
061E: D006            BNE MENU3
0620: 207306          JSR B
0623: 18              CLC
0624: 90DF            BCC MENU1
0626: C945   MENU3    CMP #$45
0628: D001            BNE MENU4
062A: 00              BRK
062B: A9FD   MENU4    LDA #$FD
062D: 20A4F6          JSR EOUTCH
0630: 18              CLC
0631: 90D2            BCC MENU1
0633: A99B   TXTOUT   LDA #$9B
```

```
0635: 20A4F6           JSR EOUTCH
0638: A240             LDX #TEXT:L
063A: A006             LDY #TEXT:H
063C: 2085F3           JSR PUTLIN
063F: 60               RTS
0640: 50524F TEXT      ASC "PROGRAM (A)  "
0643: 475241
0646: 4D2028
0649: 412920
064C: 20
064D: 50524F           ASC "PROGRAM (B)  "
0650: 475241
0653: 4D2028
0656: 422920
0659: 20
065A: 454E44           ASC "ENDE  (E)"
065D: 452020
0660: 284529
0663: 9B               DFB $9B
0664: A278   A0        LDX #120
0666: A941   AA        LDA #$41
0668: 86FF             STX $FF
066A: 20A4F6           JSR EOUTCH
066D: A6FF             LDX $FF
066F: CA               DEX
0670: D0F4             BNE AA
0672: 60               RTS
0673: A278   B         LDX #120
0675: A942   BB        LDA #$42
0677: 86FF             STX $FF
0679: 20A4F6           JSR EOUTCH
067C: A6FF             LDX $FF
067E: CA               DEX
067F: D0F4             BNE BB
0681: 60               RTS

PHYSICAL ENDADDRESS: $0682
```

```
*** NO WARNINGS

PUTLIN              $F385
EOUTCH              $F6A4
MENU1               $0605
MENU3               $0626
TXTOUT              $0633
A0                  $0664
B                   $0673
GETCHR              $F6DD
MENU                $0600     UNUSED
MENU2               $061C
MENU4               $062B
TEXT                $0640
AA                  $0666
BB                  $0675
```

```
0600    A97D20A4F6203306    )  $v 3F
0608    A99B20A4F6A90020    )Ä $v)§
0610    DDF6C941D0062064    üvIAPF d
0618    061890E9C942D006    FXPiIBPF
0620    2073061890DFC945     sFXP_IE
0628    D00100A9FD20A4F6    PA§)  $v
0630    1890D2A99B20A4F6    XPR)Ä $v
0638    A240A0062085F360    "§ F Es"
0640    50524F4752414D20    PROGRAM
0648    2841292020505224F   (A)  PRO
0650    4752414D20284229    GRAM (B)
0658    2020454E44452020      ENDE
0660    2845299BA278A941    (E)Ä"x)A
0668    86FF20A4F6A6FFCA    F  $v& J
0670    D0F460A278A94286    Pt""x)BF
0678    FF20A4F6A6FFCAD0      $v& JP
0680    F460                      t"
```

Part 7

This chapter deals with the input of numbers.

7-1 Input of a hex number

For the input we use subroutine GETCHR. Subroutine PACK then checks the input (0 - 9, A - F). If the character is not a hex number, then the program leaves the input mode, having the ASCII character in the accumulator. The following figure shows the logic flowchart of PACK.

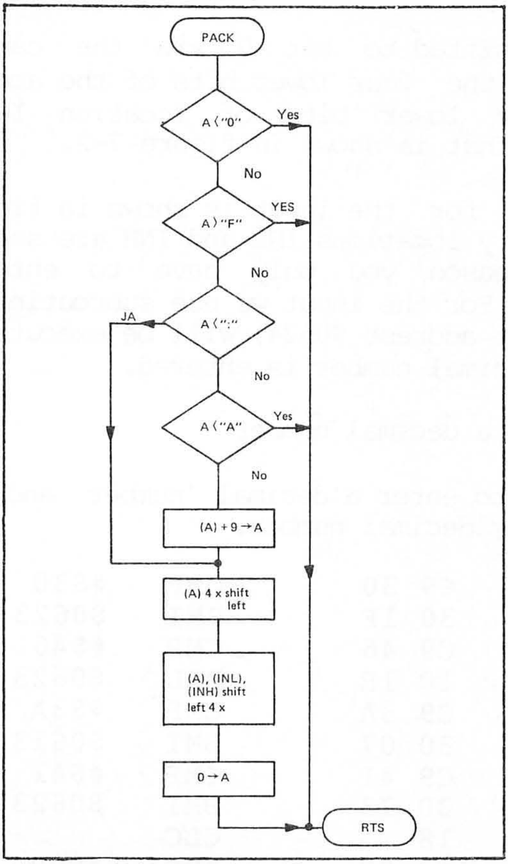


Figure 7-1 Logic flowchart of PACK

The ASCII character has to be in the accumulator, when the subroutine is entered. First the character is compared to 0, then to F. If it is smaller than 0 or greater than F, it is not a hexadecimal number. For the other characters between 0 and F, two other comparisons are to be made. If the character is smaller than ':', then it is a number between 0 and 9. If it is not smaller than A, then it is a number between A and F. In this case 9 will be added to the number. 'A' is $41. With the addition of 9 the lower four bits then represent a 10. By shifting the contents of the accumulator to the left four times this number gets into the four higher bits. Next the contents of the accumulator and locations INL and INH are shifted left by ROL (four times).

Bit 7 gets shifted to bit 0 via the carry bit. After that the four lower bits of the accumulator are the four lower bits of location INL. The program for that is shown in figure 7-2.

The program for the input is shown in figure 7-3. The two memory locations INL and INH are set to 0. For this reason you only have to enter 4F for number 004F. For the input we use subroutine GETCHR. GETWD (start address $0624) will be executed, until a non-hexadecimal number is entered.

7-2 Input of a decimal number

Now we want to enter a decimal number and convert it into a hexadecimal number.

```
0600:   C9 30       CMP   #$30
0602:   30 1F       BMI   $0623
0604:   C9 46       CMP   #$46
0606:   10 1B       BPL   $0623
0608:   C9 3A       CMP   #$3A
060A:   30 07       BMI   $0613
060C:   C9 41       CMP   #$41
060E:   30 13       BMI   $0623
0610:   18          CLC
0611:   69 09       ADC   #$09
```

```
0613:   0A              ASL
0614:   0A              ASL
0615:   0A              ASL
0616:   0A              ASL
0617:   A0 04           LDY    #$04
0619:   2A              ROL
061A:   26 80           ROL    $80
061C:   26 81           ROL    $81
061E:   88              DEY
061F:   D0 F8           BNE    $0619
0621:   A9 00           LDA    #$00
0623:   60              RTS
```

Figure 7-2 PACK

```
0624:   A9 00           LDA    #$00
0626:   85 80           STA    $80
0628:   85 81           STA    $81
062A:   20 DD F6        JSR    $F6DD
062D:   20 00 06        JSR    $0600
0630:   D0 09           BNE    $063B
0632:   A5 80           LDA    $80
0634:   29 0F           AND    #$0F
0636:   20 00 10        JSR    $1000
0639:   10 EF           BPL    $062A
063B:   60              RTS
063C:   00              BRK
```

Figure 7-3 Input of a hex number

HEX-Dump from both programs (Fig. 7-2 and 7-3)

```
0600    C9 30 30 1F C9 46 10 1B
0608    C9 3A 30 07 C9 41 30 13
0610    18 69 09 0A 0A 0A 0A A0
0618    04 2A 26 80 26 81 88 D0
0620    F8 A9 00 60 A9 00 85 80
0628    85 81 20 DD F6 20 00 06
0630    D0 09 A5 80 29 0F 20 00
0638    10 10 EF 60 00 00 00 00
```

The character entered is checked to see if it is a digit, inclusive, 0 through 9. The content of the input buffer is then multiplied by 10 and the new number is added.

Since the 6502 CPU doesn't have a command for multiplication we have to do that another way. One way would be to add the number 10 times. We however, use a different technique. A shift left command corresponds with a multiplication by two.

```
Example :    6 = %00000110
                 %00001100 = 12
```

The number is stored and shifted left two times, which means a multiplication by 4. Next the original number is added so that we now have five times the original number. The final step in multiplying by 10 consists of one more shift left. The program to do this is shown in figure 7-4.

Input of a decimal number

```
0600    A9 00 85 80 85 81 20 DD
0608    F6 20 A4 F6 C9 30 30 3B
0610    C9 39 10 37 29 0F 20 24
0618    06 18 65 80 85 80 90 02
0620    E6 81 90 E2 85 82 A5 80
0628    85 83 A5 81 85 84 26 80
0630    26 81 26 80 26 81 A5 80
0638    18 65 83 85 80 A5 81 65
0640    84 26 80 26 81 B0 03 A5
0648    82 60 00 A9 9B 20 A4 F6
0650    A5 81 20 00 10 A5 80 20
0658    00 10 00 00 00 00 00 00

0600:   A9 00          LDA   #$00
0602:   85 80          STA   $80
0604:   85 81          STA   $81
0606:   20 DD F6       JSR   $F6DD
0609:   20 A4 F6       JSR   $F6A4
060C:   C9 30          CMP   #$30
060E:   30 3B          BMI   $064B
```

```
0610:   C9 39        CMP   #$39
0612:   10 37        BPL   $064B
0614:   29 0F        AND   #$0F
0616:   20 24 06     JSR   $0624
0619:   18           CLC
061A:   65 80        ADC   $80
061C:   85 80        STA   $80
061E:   90 02        BCC   $0622
0620:   E6 81        INC   $81
0622:   90 E2        BCC   $0606
0624:   85 82        STA   $82
0626:   A5 80        LDA   $80
0628:   85 83        STA   $83
062A:   A5 81        LDA   $81
062C:   85 84        STA   $84
062E:   26 80        ROL   $80
0630:   26 81        ROL   $81
0632:   26 80        ROL   $80
0634:   26 81        ROL   $81
0636:   A5 80        LDA   $80
0638:   18           CLC
0639:   65 83        ADC   $83
063B:   85 80        STA   $80
063D:   A5 81        LDA   $81
063F:   65 84        ADC   $84
0641:   26 80        ROL   $80
0643:   26 81        ROL   $81
0645:   B0 03        BCS   $064A
0647:   A5 82        LDA   $82
0649:   60           RTS
064A:   00           BRK
064B:   A9 9B        LDA   #$9B
064D:   20 A4 F6     JSR   $F6A4
0650:   A5 81        LDA   $81
0652:   20 00 10     JSR   $1000
0655:   A5 80        LDA   $80
0657:   20 00 10     JSR   $1000
065A:   00           BRK
```

Figure 7-4 : Input of a decimal number

The program PACK (figure 7-2) uses a loop four times with ROL, ROL INL, ROL INH. This corresponds with a multiplication by 16, which is necessary with the input of hexadecimal numbers.

Notes to part 7 :

* input of a hexadecimal number
* input of a decimal number
* multiplication by 10

8

Part 8

When you program in machine language you will use an assembler most times. An assembler is a program, which translates the mnemonic code into machine code. For example it will translate LDA #$05 into the two bytes A9 05.

An assembler also allows you to use symbolic names. If the name PORTA appears in a program, the assembler has to write in the address previously defined for PORTA. It also has to take notice of labels.
For example :

```
LDA PORTA
BNE M1
LDA PORTB
M1 STA HFZ
```

The assembler automatically calculates the number of bytes from BNE M1 to the label M1.

Assemblers usually consist of two parts. The first part is a text editor for entering the source-code.

There are text editors, where the source-code has to be entered with line numbers, while others don't require them. With most assemblers, labels have to start with a letter and have to be in the first position. Commands have to be in the second position. Labels and names usually can be up to six characters long.

After the source code has been entered, the assembler translates it into machine-code. To do that it needs additional information, so-called pseudo-commands. These pseudo-commands only affect the assembler, not the program itself.

Unfortunately these commands are different on most assemblers, but most assemblers use the following pseudo-commands :

1. ORG

The command ORG (ORIGIN) defines the start address of the machine-code.

```
ORG $2000
```

means, that the code of the first line translated will start at location $2000.

This address also is the base address for the program starting there. All absolute addresses refer to that address. An ORG command always has to be at the beginning of the assembler text, but it is possible to change it within the text.

Example :

```
     ORG $2000
     <TEXT 1>
     ORG $500
     <TEXT 2>
```

The code of text 1 starts at address $2000. The code of text 2 starts at address $500. The machine code is often called the object code.

2. OBJ

The command OBJ allows you to store the machine-code at a different location in memory.

Example :

```
     ORG $3000
     OBJ $2500
```

or on the ATMAS:

```
ORG $3000,$A800
      ↑      ↑
Logical address   physical address
```

The program will be translated with all absolute addresses referring to $3000, but the machine-code will be stored at addresses starting at $2500. If you want to start the program later, you first have to move it to $3000 with a blocktransfer.

3. END

The command END shows the assembler that the text to be translated ends here.

4. EQU

With this command a certain address gets a symbolic name.

Example : PORTA EQU $C0C0

The symbolic name PORTA corresponds with the address $C0C0.
In this case PORTA is used as a label and, by that, has to be in the first position in the text.

Some assemblers need an extra command for addresses from the zero-page.

```
     HFZ EPZ $10
```

The name HFZ corresponds with address $10 of the zero-page.
Some assemblers use the equal sign (=) instead of EQU.

5. HEX

With command HEX you can store hexadecimal numbers within a program.

Example :

```
     DATA HEX 00AFFC05
```

The numbers 00 AF FC 05 are stored in four consecutive locations starting at the symbolic address DATA.

6. ASC

If you want to store text within a program, you can use command ASC.

Example : TEXT ASC "THIS IS A TEXT"

The text between the quotation marks is stored in ASCII code at address TEXT.

Some assemblers use the command BYT.

BYT 0045AF corresponds with HEX 0045AF.

BYT "TEXT" corresponds with ASC "TEXT".

For more information on the different pseudo commands please check with the manual for the assembler.

It is possible to do calculations in the address section. The following program portion shows a pseudo instruction :

```
DATA HEX 00AFFC05
```

The command LDA DATA will load 00, LDA DATA+2 will load FC.

Be careful, if you use address calculation with relative jumps.

```
BNE *+2
```

The above example causes the program to jump two bytes, but not two lines in the text.
With some assemblers the * is a pseudo command, or a pseudo address. It tells you the present value in the program counter.

Example :

```
LDA HFZ
BNE *+2
LDA #$FF
STA HFZ
```

If the contents of HFZ is different from zero, then the command LDA #$FF is jumped.
Some assemblers allow all four basic arithmetic operations, but in most cases addition and subtraction will be enough.

The following is offered to the reader as a programming hint :

When in the program there is line : H EQU $2F

then LDA H means, load the accumulator with the contents of $2F, but LDA #H means, load the accumulator with $2F.

Notes to part 8 :

* pseudo commands
* address calculations

NOTES

9

Part 9

In this, the last chapter we will discuss some helpful suggestions and short cuts.

There are some programs, where you want the program to determine, where in memory it is located. This becomes necessary with programs which contain absolute addresses, but can run at any location in memory. With the APPLE for example, this trick is used to determine into which slot a peripheral board is plugged. Since there is no command which enables you to read the program counter, we use the following trick :

The program contains a JSR-command right to a RTS in the monitor. The present address is thereby written to the stack. You have to take into consideration, however, that the lower byte of the address is lowered by one. Figure 9-1 shows the stack pointer before, during, and after the jump to the subroutine.

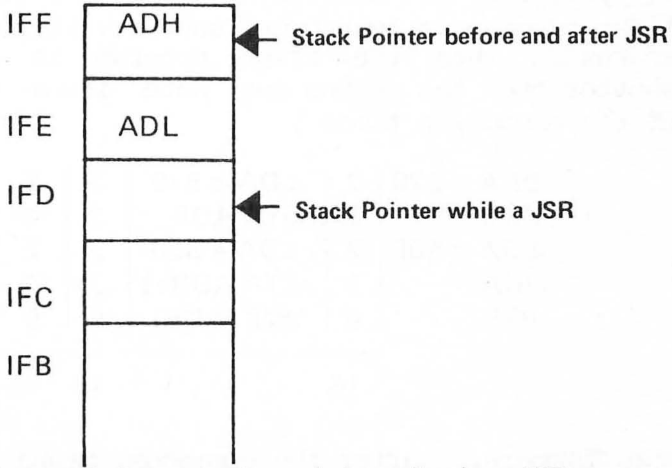


Figure 9-1 : stack pointer during JSR

After the return to the main program you can bring the contents of the stack pointer to register X with TSX. Then you can access address ADH as shown in figure 2.

You also can program another way, with an indirect jump JMP (ADR) as follows :
Let's assume, that the indirect jump should go to $2010. This can be done with the following program

```
     LDA #$20
     PHA
     LDA #$0F
     PHA
     RTS
```

You can find this technique in the operating system of ATARI. Usually an indirect jump is programmed the following way :

```
     LDA #$10
     STA ADR
     LDA #$20
     STA ADR+1
     JMP (ADR)
```

If you use an address in the zero page, then the first program is four bytes shorter. If you use any address, then the first program is six bytes shorter than the second one. Here is a comparison of the execution times :

LDA #$20	2	LDA #$10	2	2
PHA	3	STA ADR	3	4
LDA #$0F	2	LDA #$20	2	2
PHA	3	STA ADR+1	3	4
RTS	6	JMP (ADR)	5	5
	16		15	16

The numbers, after the commands, means the number of machine cycles required for this command. For the second program, the first column is an address in the zero page. The second column is for any address. You can find the number of cycles for the single commands in the reference card of the 6502 microprocessor.

Usually one doesn't think much about execution time, exept with loops which occure frequently.
To that a comparison of two program parts for relocation of data. Only the part which is different is compared. The rest is the same with both programs.

1st program

```
     LDA (FROM,X)   6
     STA (TO,X)     6
     INC FROM       5
     BNE M          2 (+1)
     INC FROM+1     5
     M INC TO       5
     BNE M1         2 (+1)
     INC TO+1       5
     M1            -------
                   36
```

The program needs 36 cycles, if no branches are executed. If a branch is executed, then one more cycle is used.

2nd program

```
     MEM LDA FROM   4
     STA TO         4
     INC MEM+1      5
     BNE M          2 (+1)
     INC MEM+2      5
     M INC MEM+4    5
     BNE M1         2 (+1)
     INC MEM+5      5
     M1            -------
                   32
```

The second program requires four cycles less, but it is a program that changes itself. Location MEM+1 contains the lower byte and location MEM+2 contains the higher byte of the command LDA FROM. This program does not work in ROM, it has to be in RAM. The savings of 4 cycles, which corresponds with 4 microseconds if the clock frequency is 1 megahertz, doesn't look great, but it accumulates with the transfer of large quantities of data.

If, in a subroutine, there is a call of another subroutine immediately before the RTS command, then you can save seven cycles, if you replace the JSR command by a JMP command,
rather than :

```
     JSR TO
     RTS
```

use just :

```
     JMP TO
```

The RTS command in subroutine TO brings you back to the same location as the RTS after JSR TO.

The processor 6502 has an indirect jump : JMP (ADR), but no indirect jump to a subroutine : JSR (ADR).
This is needed, if you want to jump to different subroutines, depending upon conditions, similar to the ON...GOTO instruction in BASIC.
If the program is in RAM, then you could use a self-modifying program, which changes the address after JSR. If the program is in ROM, then you can use the following trick.
Somewhere in memory there is a command

```
JMP1 JMP(ADR)  6C XX XX.
```

Instead of XX XX you write in the address of the subroutine to be executed. You call the subroutine with

```
     JSR JMP1
```

The RTS command in the subroutine brings you back to the command following JSR JMP1.

10

Some examples in Machine Code

Some examples in Machine Code

The following short programs are examples for programming in assembler language. With the first three programs, the equivalent BASIC program is also listed.

The first program prints one row of character C at the top of the screen.
The second program fills the screen with the character entered.
The third program prints the character entered enlarged.
It is a very nice exercise to print four big letters one beside the other.
With the fourth program you can play with two color-registers. Type B to change the background, type F to change the foreground. In each subroutine you may change the luminescence by pressing L. R will restore the old colors.

One row of char C

```
100 PRINT CHR$(125)
105 POKE 84,0
110 POKE 85,0
120 POKE 86,0
130 FOR I=0 TO 39
140 PRINT "C";
150 NEXT I
```

A screen full of characters

```
100 DIM A$(1)
110 INPUT A$
120 PRINT CHR$(125)
130 POKE 84,0
140 POKE 85,0
150 POKE 86,0
160 FOR I=0 TO 39
170 PRINT A$;
180 NEXT I
190 N=PEEK(84)
200 IF N<23 THEN POKE 85,0:GOTO 160
```

A large character

```
100 CS=57344
110 DIM A$(1)
120 INPUT A$
130 A=ASC(A$)
140 A=(A-32)*8+CS
145 PRINT CHR$(125)
150 POKE 84,5
160 POKE 85,10
170 POKE 86,0
180 FOR I=A TO A+7
190 Z=PEEK(I)
200 FOR S=1 TO 8
210 Z=Z*2
220 IF Z<255 THEN PRINT " ";:GOTO 230
222 Z=Z-256
225 PRINT A$;
230 NEXT S
235 PRINT
240 POKE 85,10
250 NEXT I
```

```
             * MACHINE CODE EXAMPLES

             * PRINTS ONE ROW OF CHAR C

             OUTCH    EQU $F6A4  * ACCU TO SCREEN
             INCH     EQU $F6E2  * KEYBOARD TO ACCU
             CV       EPZ $54    * CURSOR VERTICAL
             CH       EPZ $55    * CURSOR HORICONTAL
             AUX      EPZ $F0    * AUXILIARY

                      ORG $A800
A800: 4C0DA8          JMP START

A803: A97D   CLEAR    LDA #$7D   * ERASES SCREEN
A805: 4CA4F6          JMP OUTCH

A808: A99B   CR       LDA #$9B   * CARRIAGE RETURN
A80A: 4CA4F6          JMP OUTCH

A80D: 2003A8 START    JSR CLEAR
A810: A900            LDA #00
A812: 8554            STA CV     * SET CURSOR TO
A814: 8555            STA CH     * THE UPPER LEFT
A816: 8556            STA CH+1   * CORNER
A818: A228            LDX #40    * SET COUNTER
A81A: 86F0   S1       STX AUX    * SAVE X-REG
A81C: A943            LDA 'C'    * CHAR C INTO ACCU
A81E: 20A4F6          JSR OUTCH
A821: A6F0            LDX AUX    * GET X-REG
A823: CA              DEX        * DO IT UNTIL X-REG
A824: D0F4            BNE S1     * IS ZERO. THEN
A826: 20E2F6          JSR INCH   * WAIT FOR KEYPRESS
A829: 00              BRK

PHYSICAL ENDADDRESS: $A82A

*** NO WARNINGS
```

```
             * MACHINE CODE EXAMPLES

             * A SCREEN FULL OF CHARACTERS

             OUTCH    EQU $F6A4  * ACCU TO SCREEN
             INCH     EQU $F6E2  * KEYBOARD TO ACCU
             CV       EPZ $54    * CURSOR VERTICAL
             CH       EPZ $55    * CURSOR HORICONTAL
             AUX      EPZ $F0    * AUXILIARY

                      ORG $A800
A800: 4C0DA8          JMP START

A803: A97D   CLEAR    LDA #$7D   * ERASES SCREEN
A805: 4CA4F6          JMP OUTCH

A808: A99B   CR       LDA #$9B   * CARRIAGE RETURN
A80A: 4CA4F6          JMP OUTCH

A80D: 2003A8 START    JSR CLEAR
A810: 20E2F6          JSR INCH   * GET ONE CHARACTER
A813: 85F1            STA AUX+1
A815: A900            LDA #00
A817: 8554            STA CV
A819: 8556            STA CH+1
A81B: A900   S0       LDA #00    * CURSOR TO START
A81D: 8555            STA CH     * OF LINE
A81F: A228            LDX #40    * SET COUNTER
A821: 86F0   S1       STX AUX    * SAVE X-REG
A823: A5F1            LDA AUX+1  * CHAR  INTO ACCU
A825: 20A4F6          JSR OUTCH
A828: A6F0            LDX AUX    * GET X-REG
A82A: CA              DEX        * DO IT UNTIL X-REG
A82B: D0F4            BNE S1     * IS ZERO. THEN
A82D: A554            LDA CV     * CV IS INCREMENTED
A82F: C917            CMP #23    * AUTOMATICALLY
A831: D0E8            BNE S0     * SCREEN FULL ?
A833: 20E2F6          JSR INCH
A836: 2003A8          JSR CLEAR
A839: 00              BRK

PHYSICAL ENDADDRESS: $A83A

*** NO WARNINGS
```

```
             * MACHINE CODE EXAMPLES

             * A BIG CHARACTER

             OUTCH    EQU $F6A4  * ACCU TO SCREEN
             INCH     EQU $F6E2  * KEYBOARD TO ACCU
             CV       EPZ $54    * CURSOR VERTICAL
             CH       EPZ $55    * CURSOR HORICONTAL
             AUX      EPZ $F8    * AUXILIARY
             ADRL     EPZ AUX+2  * CHAR SET LOW BYTE
             ADRH     EPZ AUX+3  * CHAR SET HIGH BYTE
             CHAR     EPZ AUX+4

                      ORG $A800
A800: 4C0DA8          JMP START

A803: A97D   CLEAR    LDA #$7D   * ERASES SCREEN
A805: 4CA4F6          JMP OUTCH

A808: A99B   CR       LDA #$9B   * CARRIAGE RETURN
A80A: 4CA4F6          JMP OUTCH

A80D: 2003A8 START    JSR CLEAR
A810: A900            LDA #00    * SET STARTING
A812: 85FA            STA ADRL   * ADDRESS OF CHA-
A814: A9E0            LDA #$E0   * RACTER SET
A816: 85FB            STA ADRH
A818: 20E2F6          JSR INCH   * GET ONE CHARACTER
A81B: 85FC            STA CHAR
A81D: 38              SEC        * CALCULATE ADDRESS
A81E: E920            SBC #$20   * #-$20
A820: 85F8            STA AUX
A822: A900            LDA #00
A824: 85F9            STA AUX+1
A826: 18              CLC
A827: A203            LDX #03
A829: 06F8   S0       ASL AUX    * MULTIPLY BY 8
A82B: 26F9            ROL AUX+1
A82D: CA              DEX
A82E: D0F9            BNE S0
A830: 18              CLC        * ADD STARTING
A831: A5F8            LDA AUX    * ADDRESS
A833: 65FA            ADC ADRL
A835: 85FA            STA ADRL
A837: A5F9            LDA AUX+1
A839: 65FB            ADC ADRH
A83B: 85FB            STA ADRH

A83D: A90A            LDA #10    * PRINT CHARACTER
A83F: 8555            STA CH     * UPPER LEFT CORNER
```

```
A841: A905             LDA #05     * AT CV=5 CH=10
A843: 8554             STA CV
A845: A000   W0        LDY #00     * GET BIT PATTERN
A847: B1FA             LDA (ADRL),Y
A849: 85F8             STA AUX
A84B: A208             LDX #08
A84D: 86F9   W01       STX AUX+1
A84F: A920             LDA #$20    * IF THERE IS A ONE
A851: 06F8             ASL AUX     * PRINT CHARACTER
A853: 9002             BCC W1      * OTHERWISE A BLANK
A855: A5FC             LDA CHAR
A857: 20A4F6 W1        JSR OUTCH
A85A: A6F9             LDX AUX+1
A85C: CA               DEX
A85D: D0EE             BNE W01
A85F: 2008A8           JSR CR      * GET NEXT BIT PATTERN
A862: A90A             LDA #10
A864: 8555             STA CH
A866: A554             LDA CV
A868: C90D             CMP #13
A86A: F008             BEQ W2
A86C: E6FA             INC ADRL
A86E: D0D5             BNE W0
A870: E6FB             INC ADRH
A872: D0D1             BNE W0
A874: 20E2F6 W2        JSR INCH
A877: 2003A8           JSR CLEAR
A87A: 00               BRK

PHYSICAL ENDADDRESS: $A87B

*** NO WARNINGS
```

```
              * MACHINE CODE EXAMPLES

              * SETTING THE COLOR REGISTERS

              INCH    EQU $F6E2
              OUTCH   EQU $F6A4
              COLOR   EQU $2C4
              AUX     EPZ $F8

                      ORG $A800
A800: 4C0EA8          JMP START

A803: A204   COLSAV LDX #04     * SAVE COLOR REG
A805: BDC402 C1     LDA COLOR,X
A808: 95F8          STA AUX,X
A80A: CA            DEX
A80B: 10F8          BPL C1
A80D: 60            RTS

A80E: 2003A8 START  JSR COLSAV
A811: 20E2F6 S0     JSR INCH
A814: C942          CMP 'B'     * CHANGE BACKGROUND ?
A816: D003          BNE S1
A818: 202CA8        JSR BCOLOR
A81B: C946   S1     CMP 'F'     * CHANGE FOREGROUND ?
A81D: D003          BNE S2
A81F: 2048A8        JSR FCOLOR
A822: C952   S2     CMP 'R'     * RESTORE OLD COLORS ?
A824: D003          BNE S3
A826: 4C64A8        JMP RCOLOR
A829: 18     S3     CLC
A82A: 90E5          BCC S0

A82C: ADC802 BCOLOR LDA COLOR+4 * ADD ONE TO
A82F: 18            CLC         * COLOR REG
A830: 6910          ADC #%00010000
A832: 8DC802        STA COLOR+4
A835: 20E2F6 B1     JSR INCH
A838: C94C          CMP 'L'     * CHANGE LUMINESCANCE
A83A: D00B          BNE B9
A83C: ADC802        LDA COLOR+4
A83F: 18            CLC
A840: 6902          ADC #$02
A842: 8DC802        STA COLOR+4
A845: D0EE          BNE B1
A847: 60     B9     RTS
A848: ADC602 FCOLOR LDA COLOR+2 * SAME AS BCOLOR
A84B: 18            CLC         * EXCEPT COLOR REG
A84C: 6910          ADC #%00010000
A84E: 8DC602        STA COLOR+2
```

```
A851: 20E2F6 F1          JSR INCH
A854: C94C               CMP 'L'
A856: D00B               BNE F9
A858: ADC602             LDA COLOR+2
A85B: 18                 CLC
A85C: 6902               ADC #$02
A85E: 8DC602             STA COLOR+2
A861: D0EE               BNE F1
A863: 60     F9          RTS
A864: A204   RCOLOR      LDX #04     * RESTORE OLD COLORS
A866: B5F8   R1          LDA AUX,X
A868: 9DC402             STA COLOR,X
A86B: CA                 DEX
A86C: 10F8               BPL R1
A86E: 00                 BRK
```

RELOCATOR

RELOCATOR for the ATARI 400/800

This relocator for the ATARI 400/800 was developed using the ATARI Editor/Assembler cartridge.

Before you start the relocator at 32CF hex you must enter the start address, the end address as well as the destination address of the program to be relocated.

Please check your program for tables and text before relocating, because the relocator may think that this is opcode and change some bytes.

Memory location		**Lable**	**Remarks**
93 hex		RFLAG	0 = Relocate, I = Blocktransfer
81 hex 82 hex		TEST1	LSB Lower MSB address of available memory
83 hex 84 hex		TEST2	LSB Upper address MSB of available memory
85 hex 86 hex	LSB MSB	START	Starting address of the program to be relocated
87 hex 88 hex	CSB MSB	STOP	Endaddress of the program to be relocated
89 hex	LSB		New starting address of relocated program.

This is the assembly text for the ATARI Editor/Assembler cartridge.

Type: ASM,#P:

while in the editor.

```
                0 ;*****************
               20 ;*               *
               30 ;*               *
               40 ;*               *
               50 ;*   PROGRAMM    *
               60 ;*   RELOCATOR   *
               70 ;*               *
               80 ;*               *
               90 ;*****************
0000           95               *=   $700
0000           0100 RFLAG       =    $0
0001           0110 TEST1       =    $1
0003           0120 TEST2       =    $3
0005           0130 START       =    $5
0007           0140 STOP        =    $7
0009           0150 BEG         =    $9
000B           0160 OPTR        =    $B
000D           0170 TEMP2       =    $D
000F           0180 NPTR        =    $F
0011           0190 TEMP1       =    $11
               0200 ;
0700           0210             *=   $2000
2000 A205      0220 BEGIN       LDX  #$5
2002 B505      0230 S10         LDA  START,X
2004 950B      0240             STA  OPTR,X
2006 CA        0250             DEX
2007 10F9      0260             BPL  S10
2009 E8        0270             INX
200A A500      0280 MOVE        LDA  RFLAG
200C F006      0290             BEQ  MO1
200E 204E20    0300             JSR  MOV1
2011 4C5F20    0310             JMP  DONE
2014 A10B      0320 MO1         LDA  (OPTR,X)
2016 A8        0330             TAY
2017 D006      0340             BNE  MO2
2019 205220    0350             JSR  SKIP
201C 4C5F20    0360             JMP  DONE
201F 204E20    0370 MO2         JSR  MOV1
               0380 T
2022 C920      0390             CMP  #$20
2024 D003      0400             BNE  BYTE1
2026 4C7920    0410             JMP  BYTE3
               0420 ;TEST FOR 1 BYTE INTRUCTION
```

```
2029 98     0430 BYTE1   TYA
202A 299F   0440         AND  #$9F
202C F031   0450         BEQ  DONE
202E 98     0460         TYA
202F 291D   0470         AND  #$1D
2031 C908   0480         CMP  #$8
2033 F02A   0490         BEQ  DONE
2035 C918   0500         CMP  #$18
2037 F026   0510         BEQ  DONE
            0520 ;TEST FOR 3 BYTE  INSTRUCTON
            0530 ;
2039 98     0540         TYA
203A 291C   0550         AND  #$1C
203C C91C   0560         CMP  #$1C
203E F039   0570         BEQ  BYTE3
2040 C918   0580         CMP  #$18
2042 F035   0590         BEQ  BYTE3
2044 C90C   0600         CMP  #$0C
2046 F031   0610         BEQ  BYTE3
            0620 ;
            0630 ;REMAINING 2 BYTE INSTRUCTIONS
            0640 ;
2048 204E20 0650         JSR  MOV1
204B 4C5F20 0660         JMP  DONE
            0670 ;MOVE 1 BYTE
            0680 ;
204E A10B   0690 MOV1    LDA  (OPTR,X)
2050 810F   0700         STA  (NPTR,X)
2052 20D920 0710 SKIP    JSR  IOPTR
2055 20E020 0720         JSR  INPTR
2058 60     0730         RTS
            0740 ;
            0750 ;MOVE 2BYTES
            0760 ;
2059 204E20 0770 MOV2    JSR  MOV1
205C 204E20 0780         JSR  MOV1
205F A50B   0790 DONE    LDA  OPTR
2061 8511   0800         STA  TEMP1
2063 A50C   0810         LDA  OPTR+1
2065 8512   0820         STA  TEMP1+1
2067 A507   0830         LDA  STOP
2069 850D   0840         STA  TEMP2
206B A508   0850         LDA  STOP+1
```

```
206D 850E   0860           STA  TEMP2+1
206F 20CE20 0870           JSR  TEST
2072 9096   0880           BCC  MOVE
2074 F094   0890           BEQ  MOVE
2076 00     0900           BRK
2077 EA     0910           NOP
2078 EA     0920           NOP
            0930 ;
            0940 ;3BYYTE INSTRUCTIONS
            0950 ;
            0960 ;
2079 A10B   0970 BYTE3     LDA  (OPTR,X)
207B 8511   0980           STA  TEMP1
207D 20D920 0990           JSR  IOPTR
2080 A10B   1000           LDA  (OPTR,X)
2082 8512   1010           STA  TEMP1+1
2084 20E720 1020           JSR  DOPTR
2087 A501   1030           LDA  TEST1
2089 850D   1040           STA  TEMP2
208B A502   1050           LDA  TEST1+1
208D 850E   1060           STA  TEMP2+1
208F 20CE20 1070           JSR  TEST
2092 F002   1080           BEQ  B10
2094 90C3   1090           BCC  MOV2
2096 A503   1100 B10       LDA  TEST2
2098 850D   1110           STA  TEMP2
209A A504   1120           LDA  TEST2+1
209C 850E   1130           STA  TEMP2+1
209E 20CE20 1140           JSR  TEST
20A1 F002   1150           BEQ  B20
20A3 B0B4   1160           BCS  MOV2
            1170 ;
            1180 ;ADRESS RECOMPUTATION
            1190 ;
20A5 38     1200 B20       SEC
20A6 A10B   1210           LDA  (OPTR,X)
20A8 E505   1220           SBC  START
20AA 850D   1230           STA  TEMP2
20AC 20D920 1240           JSR  IOPTR
20AF A10B   1250           LDA  (OPTR,X)
20B1 E506   1260           SBC  START+1
20B3 850E   1270           STA  TEMP2+1
20B5 20D920 1280           JSR  IOPTR
```

```
20B8 18     1290          CLC
20B9 A50D   1300          LDA  TEMP2
20BB 6509   1310          ADC  BEG
20BD 810F   1320          STA  (NPTR,X)
20BF 20E020 1330          JSR  INPTR
20C2 A50E   1340          LDA  TEMP2+1
20C4 650A   1350          ADC  BEG+1
20C6 810F   1360          STA  (NPTR,X)
20C8 20E020 1370          JSR  INPTR
20CB 4C5F20 1380          JMP  DONE
            1390 ;
            1400 ;TEST COMPARES 2 ADRESSES
            1410 ;
20CE A512   1420 TEST     LDA  TEMP1+1
20D0 C50E   1430          CMP  TEMP2+1
20D2 D004   1440          BNE  T10
20D4 A511   1450          LDA  TEMP1
20D6 C50D   1460          CMP  TEMP2
20D8 60     1470 T10      RTS
            1480 ;
            1490 ;INCREMENT OLD POINTER
            1500 ;
20D9 E60B   1510 IOPTR    INC  OPTR
20DB D002   1520          BNE  INC10
20DD E60C   1530          INC  OPTR+1
20DF 60     1540 INC10    RTS
            1550 ;
            1560 ;INCREMENT NEW POINTER
            1570 ;
20E0 E60F   1580 INPTR    INC  NPTR
20E2 D002   1590          BNE  INC20
20E4 E610   1600          INC  NPTR+1
20E6 60     1610 INC20    RTS
            1620 ;
            1630 ;DECREMENT OLD POINTER
            1640 ;
20E7 C60B   1650 DOPTR    DEC  OPTR
20E9 A50B   1660          LDA  OPTR
20EB C9FF   1670          CMP  #$FF
20ED D002   1680          BNE  D10
20EF C60C   1690          DEC  OPTR+1
20F1 60     1700 D10      RTS
            1710 END
```

You can enter this object-code with the ATMONA-1 from ELCOMP:

```
32CF      A2 05 B5 05 95 0B CA 10
32D7      F9 E8 A5 00 F0 06 20 1D
32DF      33 4C 2E 33 A1 0B A8 D0
32E7      06 20 21 33 4C 2E 33 20
32EF      1D 33 C9 20 D0 03 4C 48
32F7      33 98 29 9F F0 31 98 29
32FF      1D C9 08 F0 2A C9 18 F0
3307      26 98 29 1C C9 1C F0 39
330F      C9 18 F0 35 C9 0C F0 31
3317      20 1D 33 4C 2E 33 A1 0B
331F      81 0F 20 A8 33 20 AF 33
3327      60 20 1D 33 20 1D 33 A5
332F      0B 85 11 A5 0C 85 12 A5
3337      07 85 0D A5 08 85 0E 20
333F      9D 33 90 96 F0 94 00 EA
3347      EA A1 0B 85 11 20 A8 33
334F      A1 0B 85 12 20 B6 33 A5
3357      01 85 0D A5 02 85 0E 20
335F      9D 33 F0 02 90 C3 A5 03
3367      85 0D A5 04 85 0E 20 9D
336F      33 F0 02 B0 B4 38 A1 0B
3377      E5 05 85 0D 20 A8 33 A1
337F      0B E5 06 85 0E 20 A8 33
3387      18 A5 0D 65 09 81 0F 20
338F      AF 33 A5 0E 65 0A 81 0F
3397      20 AF 33 4C 2E 33 A5 12
339F      C5 0E D0 04 A5 11 C5 0D
33A7      60 E6 0B D0 02 E6 0C 60
33AF      E6 0F D0 02 E6 10 60 C6
33B7      0B A5 0B C9 FF D0 02 C6
33BF      0C 60 00 00 00 00 00 00
33C7      00 00 00 00 00 00 00 00
33CF      00 00 00 00 00 00 00 00
33D7      00 00 00 00 00 00 00 00
```

Reverse Video

REVERSE VIDEO
You can enter this program using the ATMONA-1. Start the program with the GOTO command

GOTO 600

A part of the screen is displayed in reverse. If you type GOTO 600 the screen will be switched back to normal operation. Instead of RTS you can also use the BRK command.

```
                                ORG $0600
0600: 68                        PLA
0601: A559                      LDA $59
0603: 85D5                      STA $D5
0605: A900                      LDA #$00
0607: 85D4                      STA $D4
0609: A603                      LDX $03
060B: A458                      LDY $58
060D: B1D4        LOOP          LDA ($D4),Y
060F: 4980                      EOR #$80
0611: 91D4                      STA ($D4),Y
0613: C8                        INY
0614: D0F7                      BNE LOOP
0616: E6D5                      INC $D5
0618: CA                        DEX
0619: 10F2                      BPL LOOP
061B: 60                        RTS

PHYSICAL ENDADDRESS: $061C

*** NO WARNINGS

LOOP                 $060D
```

```
0600    68 A5 59 85 D5 A9 00 85
0608    D4 A6 03 A4 58 B1 D4 49
0610    80 91 D4 C8 D0 F7 E6 D5
0618    CA 10 F2 60
```

ASC II Output

ASCII Output

This is a sample program, which can be typed in using the Editor/Assembler cartridge or the ATMAS-1 (ATAS) from ELCOMP Publishing, Inc.

a) Using ATAS (ATMAS-1)
   CTRL-I = TAB = 9 Blanks (column for commands)
   Start all lables at the beginning of the line.

```
                             ORG $0600
                 EOUTCH      EQU $F6A4
0600: A900       START       LDA #$00
0602: 85D4                   STA $D4
0604: A5D4       REP         LDA $D4
0606: 85D4                   STA $D4
0608: A5D4                   LDA $D4
060A: 20A4F6                 JSR EOUTCH
060D: E6D4                   INC $D4
060F: D0F3                   BNE REP
0611: 00                     BRK

PHYSICAL ENDADDRESS: $0612

*** NO WARNINGS

EOUTCH              $F6A4
REP                 $0604
START               $0600     UNUSED
```

How to enter this program using the EDITOR from ATAS or ATMAS-1?

Start your Editor/Assembler and type

CTRL-I

To set a TAB for

OUT LNP1

which allows you to assemble to the printer later.

Then define your label EOUTCH, the starting address of the screen output routine in the operating system. EOUTCH has to be written at the beginning of the line. EQU is a pseudo opcode and has to be preceded by a CTRL-I.

It is convenient to mark the START of the program with the label "START".

To type in the mnemonic, set the TAB with CTRL-I.

Hexdump of ASCII output:

```
0600     A9 00 85 D4 A5 D4 85 D4
0608     A5 D4 20 A4 F6 E6 D4 D0
0610     F3
```

The ASCII output program in ATARI Editor/Assembler syntax.

```
05   *=$0600
10 START LDA #$00;START WITH ZERO
20  STA $D4
30 REP LDA $D4
40      STA $D4;SAVE
60  STA $D4;SAVE
70  LDA $D4
80      LDA $D4;GET CHARACTER
90  JSR $F6A4;PRINT
0100  INC $D4;CHECK
0110  BNE REP
0000          05          *=   $0600
0600 A900     10 START    LDA  #$00    ;START WITH ZERO
0602 85D4     20          STA  $D4
0604 A5D4     30 REP      LDA  $D4
0606 85D4     40          STA  $D4     ;SAVE
0608 85D4     60          STA  $D4     ;SAVE
060A A5D4     70          LDA  $D4
060C A5D4     80          LDA  $D4     ;GET CHARACTER
060E 20A4F6 90            JSR  $F6A4   ;PRINT
0611 E6D4     0100        INC  $D4     ;CHECK
0613 D0EF     0110        BNE  REP
```

RANDOM

Number Generator

RANDOM Number Generator
Randomness is required for many games like dice-games, maze-games etc. The program is based on a pseudo random shift register approach. Two bytes are used as a shift register. (RNDM and RNDM+1). At least one of the locations RNDM or RNDM+1 has to be non-zero. We have chosen the zero page location $95 and $96. Before starting the program, use the monitor to set one of these locations to a non-zero value.

After assembly you can start the program from the monitor with the GOTO 600 command.
The following program prints only one random number before it hits the BRK command. (If called from BASIC this BRK has to be replaced by an RTS command.

```
                    ORG $0600
       EOUTCH       EQU $F6A4
       RNDM         EPZ $95
0600: A508   RANDOM LDA $08              ;SET ITERATIONS
0602: 48     R1     PHA                  ;SAVE COUNTER
0603: A595          LDA RNDM             ;GET BYTE
0605: 2A            ROL
0606: 4595          EOR RNDM             ;XOR BITS 13 & 14
0608: 2A            ROL
0609: 2A            ROL
060A: 2696          ROL RNDM+1           ;SHIFT BYTE
060C: 2695          ROL RNDM             ;SHIFT 2. BYTE
060E: 68            PLA                  ;GET COUNTER
060F: 18            CLC
0610: 69FF          ADC #$FF             ;DECREMENT
0612: D0EE          BNE R1               ;IF NOT DONE DO AGAIN
0614: A595          LDA RNDM             ;GET RANDOM BYTE
0616: 20A4F6        JSR EOUTCH           ;PRINT
0619: 00            BRK

PHYSICAL ENDADDRESS: $061A

*** NO WARNINGS

EOUTCH          $F6A4              RNDM           $95
RANDOM          $0600    UNUSED    R1             $0602
```

```
0600      A5 08 48 A5 95 2A 45 95
0608      2A 2A 26 96 26 95 68 18
0610      69 FF D0 EE A5 95 20 A4
0618      F6 00
```

The following program is also a random number generator, but it will print 10 random numbers on the screen rather than one.

Note! If you count less than 10 random characters then one character was a control character, for example CARRIAGE RETURN.

```
                                  ORG $0600
                   EOUTCH         EQU $F6A4
                   RNDM           EPZ $95
                   COUNTER        EPZ $98
0600:  A900                       LDA #0
0602:  8598                       STA COUNTER
0604:  A508        RANDOM         LDA $08          ;SET ITERATIONS
0606:  48          R1             PHA              ;SAVE COUNTER
0607:  A595                       LDA RNDM         ;GET BYTE
0609:  2A                         ROL
060A:  4595                       EOR RNDM         ;XOR BITS 13 & 14
060C:  2A                         ROL
060D:  2A                         ROL
060E:  2696                       ROL RNDM+1       ;SHIFT BYTE
0610:  2695                       ROL RNDM         ;SHIFT 2. BYTE
0612:  68                         PLA              ;GET COUNTER
0613:  18                         CLC
0614:  69FF                       ADC #$FF         ;DECREMENT
0616:  D0EE                       BNE R1           ;IF NOT DONE DO AGAIN
0618:  A595                       LDA RNDM         ;GET RANDOM BYTE
061A:  20A4F6                     JSR EOUTCH       ;PRINT
061D:  E698                       INC COUNTER
061F:  A90A                       LDA #$0A
0621:  C598                       CMP COUNTER
0623:  D0DF                       BNE RANDOM
0625:  00                         BRK

PHYSICAL ENDADDRESS: $0626

*** NO WARNINGS

EOUTCH              $F6A4                    RNDM              $95
COUNTER             $98                      RANDOM            $0604
R1                  $0606
```

```
0600      A9 00 85 98 A5 08 48 A5
0608      95 2A 45 95 2A 2A 26 96
0610      26 95 68 18 69 FF D0 EE
0618      A5 95 20 A4 F6 E6 98 A9
0620      0A C5 98 D0 DF 00
```

NOTES

Accessing Machine Language Programs from BASIC

Accessing Machine Language Programs from BASIC
The BASIC programmer often wants to speed up a program. The best to do that, is to link a machine language subroutine to BASIC. Therefore the machine language code has to be placed in a protected area (save from BASIC). From BASIC a machine language subroutine can be called by the statement

10 A = USR(X) : X is the starting address of the machine language subroutine in decimal

Let us now use the Reverse Video program to demonstrate the technique.

```
                                  ORG $0600
     0600:  68                    PLA
     0601:  A559                  LDA $59
     0603:  85D5                  STA $D5
     0605:  A900                  LDA #$00
     0607:  85D4                  STA $D4
     0609:  A603                  LDX $03
     060B:  A458                  LDY $58
     060D:  B1D4       LOOP       LDA ($D4),Y
     060F:  4980                  EOR #$80
     0611:  91D4                  STA ($D4),Y
     0613:  C8                    INY
     0614:  D0F7                  BNE LOOP
     0616:  E6D5                  INC $D5
     0618:  CA                    DEX
     0619:  10F2                  BPL LOOP
     061B:  60                    RTS

     PHYSICAL ENDADDRESS: $061C
              *** NO WARNINGS
```

First we have to translate the machine code from hex into decimal. 68 = 104 dec, A5 = 165 dec. .... etc.

600 hex = 1536 dec. = Start of our program.

Then we use the following BASIC program to poke the code into memory starting at location 1536 dec.

```
10 DATA 104,165,89,133,213,169,0
20 DATA 133,212,166,3,164,88,177
30 DATA 212,73,128,145,212,200
40 DATA 208,247,230,213,202,16
50 DATA 242,96
60 FOR I=1 TO 28
70 READ A
80 POKE (1535+I),A
90 NEXT I
100 END
200 B=USR(1536)
```

To call the machine language subroutine from BASIC you type in GOTO 200. Never forget to terminate your machine language program with a RTS (60 hex = 96 dec.) for RETURN from subroutine, because BASIC uses a JSR (jump subroutine) to get to the machine language program.

Number systems

A

CHAPTER A : NUMBER SYSTEMS

In this chapter we will develop some straightforward mathematics, based on daily experience, which will make it much simpler to model the internal workings of microcomputers.

Decimal numbers
Quantity
Binary Numbers, BITS, and BYTES
Hexadecimal Numbers

DECIMAL NUMBERS, AND THE CONCEPT OF QUANTITY...

Western culture has adopted the ten arabic symbols: 0,1,2,3,4,5,6,7,8, and 9 to represent various quantities. Many other symbols are available to describe a particular quantity. For example, 'three' may be symbolized as three, 3, trois (French), III (Roman Numerals), etc.

With the exception of the Roman Numerals, the above examples refer to the DECIMAL, or BASE-TEN number system which we use daily. The base-ten system is charaterized by the ten symbols which are available to use in constructing symbolic representations of various quantities. For large (multi-digit) numbers, we combine several symbols, and assign each symbol a multiplier based upon it's position within the series of symbols. For example, we represent the number of eggs in a carton with the symbols '12'. The symbol on the far right side is in what we call the 'unit' position. The next symbol to the left is in what we call the 'tens' position, and represents the number of complete groups of ten eggs. The total number of eggs is equal to ten times the number in the tens position, plus one times the number in the unit's position. Were there another symbol to the left, that symbol would be multiplied by ten, and then ten again. (i.e. multiplied by one-hundred). Were there a symbol still further to the left, then that symbol would be accompanied by yet another multiplication by ten. (i.e. multiplied by one-thousand).

Summarizing, the base-ten (or decimal) number system is characterized by:

1). A basic set of TEN symbols (0-9).
2). Each digit positioned left of the unit position are accompanied by a multiplier, and that multiplier increases by a factor of TEN for every additional digit postion to the left.
3). Decimal numbers are NOT the only method of representing a quantity.

We will now explore some number systems commonly used in
association with computer systems. (They are harder for us, but
easier for the computer!).

BINARY NUMBERS...

Generally, computers do not deal directly with the symbols of the decimal number system. The computer is made up of combinations of circuits capable of presenting only two basic symbols (as opposed to ten). Logic circuits inside the computer represent one symbol with a high level voltage (often about five volts), and the other symbol with a low level voltage (often about zero volts).These states are often described with the symbols 'high' or '1' for the high voltage level, and the symbols 'low' or '0' for the low voltage level. Multiple digit binary numbers can therefore be represented by multiple wires, with each wire at either a '1' or a '0' voltage level. By drawing a parallel to the base-ten number system, we may define this to be a BASE-TWO (or BINARY) number system, summarized by the following characteristics:

1). A basic set of TWO symbols (1,2).
2). Each digit positioned left of the unit position are accompanied by a multiplier, and that multiplier increases by a factor of TWO for every additional digit postion to the left.

Significance of digit position, decimal numbers versus binary numbers:

DECIMAL	(10000'S)	(1000'S)	(100'S)	(10'S)	(1'S)
BINARY	(16'S)	(8'S)	(4'S)	(2'S)	(1'S)

Some examples of binary numbers follow.

TRIAL QUANTITY	BASE-2 (BINARY)	EXPLANATION OF BINARY
NONE	0	0 IN UNIT'S PLACE
ONE	1	1 IN UNIT'S PLACE
TWO	10	2 TIMES ONE IN TWO'S PLACE, PLUS ONE IN UNIT'S PLACE.
THREE	11	2 TIMES ONE IN TWO'S PLACE, PLUS ONE IN UNIT'S PLACE.
FOUR	100	2 TIMES 2 TIMES ONE IN FOUR'S PLACE, PLUS TWO TIMES ZERO IN TWO'S PLACE, PLUS ZERO IN UNIT'S PLACE.
FIVE	101	AS ABOVE, BUT ONE IN UNITS PLACE.

THIRTEEN	1101	AS ABOVE, BUT ADD 2 TIMES 2 TIMES 2 TIMES ONE IN THE EIGHT'S PLACE.

Note that in the decimal system, symbol position was used to represent multipliers of 1, 10, 100, 1000, 10000, etc. In the binary number system, symbol position is used to indicate multipliers of 1, 2, 4, 8, 16, 32, 64, 128, 256, etc.

Using the above multipliers, you should be able to convert the following binary numbers (left column) into the decimal numbers in the righthand column.

BINARY NUMBER SYMBOL	DECIMAL NUMBER SYMBOL
110	6
101000	40
1000000	64
111111	63
111110	62
111101	61
1111111	127

There is no real trick to reading binary numbers. If you desire to get the numbers into decimal form, then there is no avoiding the process of multiplying the appropriate digits by 1, 2, 4, 8, 16, etc., and adding up the results.

One digit of a binary number, or one wire in the computer, can represent only one of two possible states. Thus one digit certainly does not contain a great abundance of information. It is therefore appropriate that we refer to one digit of a binary number as a BIT. A bit may be either a one or a zero. Carrying this madness one more step, we refer to a group of 8 BITS (an 8 digit binary number) as a BYTE.

It is important to note that the binary number system is simply an alternative way to write a number, just as Roman Numerals provide an alternative way to write a number. In all cases, a given SYMBOL represents a QUANTITY, and the method we choose to write it is of secondary importance.

Hexadecimal Numbers

HEXADECIMAL NUMBERS...

The preceeding discussion of binary numbers demonstrated that binary symbols for large quantities become very cumbersome, due to the very large number of digits which must be used. This is the natural consequence of having only two possible symbols per digit. In the decimal number system, we had ten symbols available, and large quantities could be represented with relatively few digits. Ideally, we need a number system which provides us with a large number of symbols, while retaining a simple relationship to the on/off world of individual wires within the computer.

Note that a four bit number (four digit binary number) may represent any quantity from zero (0000) to fifteen (1111), for a total of sixteen possible combinations. Now suppose we assign a SINGLE letter or number to each of these combinations, as shown in the righthand column of the table below.

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

Don't be taken aback by the use of letter symbols to represent numbers. After all, we are making the rules here, and if we wish to use the symbol 'D' to represent a quantity of thirteen, then so be it.

The above sixteen symbols (0-9, and A-F) are the sixteen basic symbols of the HEXADECIMAL (or BASE-SIXTEEN!) number system. For multiple digit numbers, we once again start with the UNITS position. But now, each time we move one digit position to the left, we add a multiplication by sixteen.

DECIMAL	BINARY	HEXADECIMAL	EXPLANATION
15	1111	F	15 IN UNIT'S PLACE.
16	1 0000	10	1 IN 16'S PLACE.
17	1 0001	11	1 IN 16'S PLACE, PLUS 1 IN UNIT'S PLACE.
42	10 1010	2A	2 IN 16'S PLACE, PLUS 10 IN UNIT'S PLACE.
255	1111 1111	FF	15 IN 16'S PLACE, PLUS 15 IN UNIT'S PLACE.
256	1 0000 0000	100	1 IN 256'S PLACE, PLUS ZERO IN 16'S PLACE, PLUS ZERO IN UNIT'S PLACE.
769	11 0000 0001	301	THREE IN 256'S PLACE, PLUS ZERO IN 16'S PLACE, PLUS 1 IN UNIT'S PLACE.
783	11 0000 1111	30F	THREE IN 256'S PLACE, PLUS ZERO IN 16'S PLACE, PLUS 15 IN UNIT'S PLACE.

The HEXADECIMAL (BASE-SIXTEEN) number system may be summarized by the following charateristics:

1). A basic set of 16 symbols (0-9,A-F).
2). Each digit positioned left of the unit position is accompanied by a multiplier, and that multiplier increases by a factor of sixteen for every additional digit positio to the left. (i.e. Multipliers of 1,16,256,4096, etc. are used).

Note that binary representations may be very easily converted to hexadecimal representations via the following steps:

1). Group the binary number into groups of four bits, starting with the unit's position, and proceeding right to left.
2). Write the hexadecimal symbol for
2). Substitute the appropriate hexadecimal symbol for each four-bit group from the original number.
3). Simply reverse this process to convert hexadecimal numbers into binary numbers, four bits at a time.

Hexadecimal numbers provide an extremely compact means of expressing multiple-bit binary numbers.

When reading a multiple digit number, it is not always immediately clear whether it is a binary, decimal, or hexadecimal representation. The symbol '1101' might be interpreted as a binary number (thirteen), a decimal number (one-thousand one-hundred and one), or as a hexadecimal number (four-thousand three-hundred and fifty-three = 1 X 4096 + 1 X 256 + 0 X 16 + 1 X 1). The number '1301' is clearly not a binary representation (it contains a '3'), but it could be interpreted as either a decimal or hexadecimal number.

In those instances when binary numbers are used, the writer usually calls attention to this fact, either by using a subscript '2', or by enclosing the notation 'binary' in the text of his discussion. Hexadecimal numbers are often distinguished from decimal numbers by preceding the hexadecimal number with a dollar sign, or by suffixing the hexadecimal number with a capital H. (i.e. $43C7, $7FFF, $4020, 1AD7H, F371H, 9564H). The dollar sign convention is the one adopted by most users of computers based on the 6502 microprocessor chip,including Ohio Scientific Instruments, and is the convention used in this book.

CHAPTER A PROBLEMS...

1). Convert the following binary numbers into decimal representations.

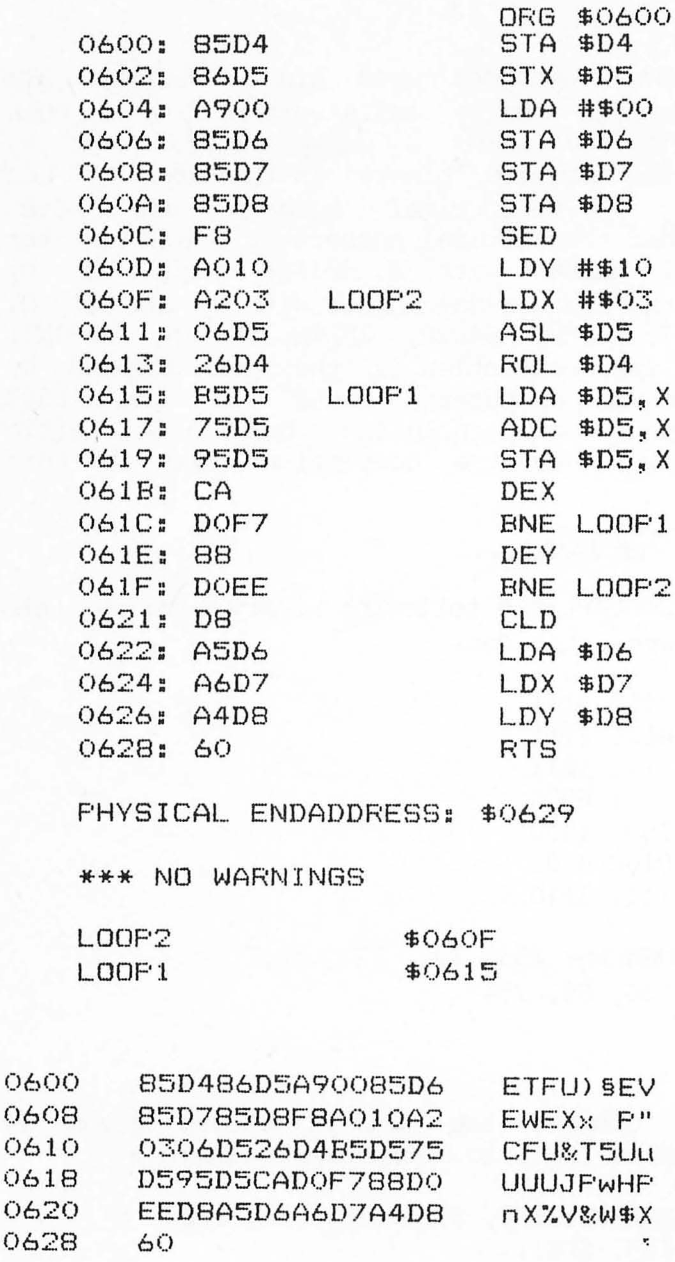
```
1111 1111
0111 1111
 111 1111
   1 0000
1000 1000
0100 0101
1111 1110
```

(ANSWERS: 255, 127, 127, 16,
136, 69, 254).

2). Convert the binary numbers given in problem number (1) into hexadecimal numbers.

(ANSWERS: $FF, $7F, $7F, $10, $88,
$45, $FE).

Here is a subroutine in machine-language for conversion of hexadecimal to decimal numbers:



```
                                    ORG $0600
    0600: 85D4                      STA $D4
    0602: 86D5                      STX $D5
    0604: A900                      LDA #$00
    0606: 85D6                      STA $D6
    0608: 85D7                      STA $D7
    060A: 85D8                      STA $D8
    060C: F8                        SED
    060D: A010                      LDY #$10
    060F: A203       LOOP2          LDX #$03
    0611: 06D5                      ASL $D5
    0613: 26D4                      ROL $D4
    0615: B5D5       LOOP1          LDA $D5,X
    0617: 75D5                      ADC $D5,X
    0619: 95D5                      STA $D5,X
    061B: CA                        DEX
    061C: D0F7                      BNE LOOP1
    061E: 88                        DEY
    061F: D0EE                      BNE LOOP2
    0621: D8                        CLD
    0622: A5D6                      LDA $D6
    0624: A6D7                      LDX $D7
    0626: A4D8                      LDY $D8
    0628: 60                        RTS

    PHYSICAL ENDADDRESS: $0629

    *** NO WARNINGS

    LOOP2                 $060F
    LOOP1                 $0615


0600    85D486D5A90085D6    ETFU)§EV
0608    85D785D8F8A010A2    EWEXx P"
0610    0306D526D4B5D575    CFU&T5Uu
0618    D595D5CAD0F788D0    UUUJPwHP
0620    EED8A5D6A6D7A4D8    nX%V&W$X
0628    60                         `
```

The hexadecimal number has to be in the accumulator (higher byte) and in the X-register (lower byte) when you jump to the subroutine.
Example:
We want to convert 101F hex into a decimal number.

This can be done as follows:

```
A9 10     LDA #$10
A21F      LDX #$1F
20 00 06  JSR $0600
00        BRK
```

If ATMONA-1 hits a break BRK, it displays the contents of the registers. The decimal number is in the X-register and in the Y-register.
101F hex = 4127 dec.

NOTES

Digital Concepts

B

CHAPTER TWO: DIGITAL CONCEPTS

In this chapter we present an overview of digital logic concepts, and the kinds of electronic devices used to accomplish logical operations and data storage within your computer.

LOGIC IN PROGRAMMING AND COMPUTER HARDWARE

> "...a computer is like a brain, a dumb brain, it doesn't do anything unless you program it first, and then it just follows your instructions one after another..."
>
> -reaction of ten-year-old to computers.

People program computers to perform sequences of logical operations. A computer program consists of a sequence of instructions for the computer. Often we wish the computer to decide between alternative courses of action, based upon some information which is external to the program. For example, a computer might be programmed to control the signal lights at a railway crossing. Sensor switches would be placed some distance down the railway, such that they can detect an oncoming train. The computer program might read something like:

```
 1. START HERE
 2. CHECK TO SEE IF A TRAIN IS COMING
 3. IF A TRAIN IS COMING, THEN SKIP
    AHEAD TO LINE 5 OF THE INSTRUCTIONS
 4. GO BACK TO STEP 2 OF THE INSTRUCTIONS
 5. CHECK TO SEE IF THE SAFETY BARRIER
    IS LOWERED
 6. IF THE SAFETY BARRIER IS UP, THEN
    LOWER IT
 7. CHECK TO SEE IF THE TRAIN IS STILL HERE
 8. IF THE TRAIN IS STILL HERE, OR, IF
    ANOTHER TRAIN IS COMING, THEN GO BACK
    TO STEP 7 OF THE INSTRUCTIONS
 9. RAISE THE SAFETY BARRIER
10. GO BACK TO STEP 2 OF THE INSTRUCTIONS
```

The above PROGRAM acts upon the DATA (or information) supplied by the train sensor switch. Another example would be the word-processor program upon which this manuscript is being typed. That program decides which letter to code into computer memory, based upon which one of the keyboard switches are pressed by the typist. Each of these examples also has means provided to output some result to the real world. In the case of the railway crossing, the computer has control of the position of the safety barrier, and uses that barrier to inform people of it's decision regarding the presence or absence of oncoming trains. The word processor program has control of a CRT (picture tube) upon which it displays the text input by the typist. It also outputs this text to computer memory, from whence the typist may command that it be recalled, corrected, and output to a printer. In summary, the computer executes a SEQUENCE of LOGICAL instructions upon some source of DATA input (switches, keyboards, memory, etc.), and produces some consistant OUTPUT as a result. In the remainder of this chapter, we will examine some of the fundamental electronic hardware used to accomplish logical operations within the computer.

LOGIC OPERATIONS AND LOGIC GATES...

Consider the following statements:

If (A is true) Then (Z is true)
If (A is false) Then (Z is False)

We shall assume A, Z, etc. are all either true or false, with nothing in-between being possible. With the above two statements, we have completely defined the condition of the OUTPUT Z, for all possible conditions of the input A. Suppose that we wish to model statements such as the above two, using electronic circuits. Let us define:

1. TRUE is to be represented by any
   voltage in the range from
   +2 volts to +5 volts.
   (i.e. HIGH).
2. FALSE is to be represented by any
   voltage in the range from
   0 volts to +1/2 volt.
   (i.e. LOW).

Now consider a short piece of plain copper wire, the left end labeled "INPUT--A", and the right end labeled "OUTPUT--Z." This piece of wire will certainly model our original logical statements, as re-written:

1. If (A is HIGH) then (Z is HIGH). Certainly, if we connect a 'HIGH' voltage input to point A, then the wire will carry this same high voltage to the output at point Z.

2. If (A is LOW) then (Z is LOW). Once again, the input from A is carried directly to the output at Z.

There is almost always another way to accomplish any given task, and the above example is no exeception. There are electronic circuits other than our piece of wire which we could connect from A to Z, and obtain the same result. The need for these should become apparent as we continue.

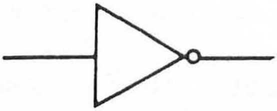
Consider the statements:

1. If (A is true), then (Z is false)
2. If (A is false), then (Z is true)
   (i.e. Z is always the opposite of A).

We cannot model this more complicated situation with only a piece of wire. We must use a readily available electronic circuit called a "NOT-gate", or "INVERTER." These devices are manufactured by many firms in many different forms. For the time being, it is perfectly sufficient to imagine a small box with two wires sticking out. One wire is our familiar input A, and the other wire is our output Z. If we put a high level on the input of an inverter, then we will get a low level at the output. A low level on the input yields a high level at the output. Forcing some signal INTO the output pin is forbidden, but the output of one inverter could certainly control the input to a second inverter. Clearly the output of inverter #2 would be exactly the same as the input to inverter #1. (This is a combination which could replace the copper wire in our earlier example).

There is a standard symbol used to represent an inverter. It is shown below in Figure 2.1.

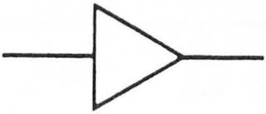
<<<<<<<<<<<<<<FIGURE 2.1>>>>>>>>>>>>>>>>
<<<<<<<<<<LOGIC INVERTER SYMBOL>>>>>>>>>>



<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>

There is a standard symbol used to represent a circuit which behaves as our copper wire did. This symbol represents a logic circuit whose single output duplicates it's single input. It is shown below in Figure 2.2. Note the absence of the "bubble" at the output, as compared with the inverter in figure 2.1. The bubble symbolizes the inversion process.

<<<<<<<<<<<<<<<FIGURE 2.2>>>>>>>>>>>>>>>>
<<<<<<<<<LOGIC BUFFER SYMBOL>>>>>>>>>>>>>?



<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>>

In certain situations we desire to connect the inputs of a number of different logic gates too the output of a single logic gate. If this number becomes too large the output of an ordinary gate might become overloaded. To prevent this we could connect the single output involved to the inputs of a pair of identical logic buffers. We could then distribute the large number of logic gate inputs between the two buffer outputs. Each buffer would have to drive only half the total number of inputs, and would not overload. More or larger buffers could be used if nessesary.

Consider the following statement:

If (A is true) OR (B is true), then (Z is true). (Otherwise Z is false).

This describes a single output (Z) controlled by two inputs (A and B). It is convenient to examine the possible outputs at Z, for all possible input combinations, through the use of a "truth table." A truth table for the current example is shown below in Figure 2.3. Note that a '1' is used to represent a 'true' condition, and that our electronic circuits would represent this with the 'high' voltage level.

TRUTH TABLE
Z = (A OR B)

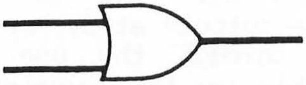
INPUT A	INPUT B	OUTPUT Z
0	0	0
0	1	1
1	0	1
1	1	1

FIGURE 2.3

In figure 2.3 we have described the operation of a "two-input OR-gate." This logical building block may be thought of as a box with THREE wires protruding. The three wires are inputs A, B, and output Z. Such circuits are readily available, and your microcomputer contains many, many of them. Note that we might also create a "Three-input OR-gate," which might have three inputs A, B, C, and output Z. In this case, output Z would become 'true' if any one OR more of the inputs became 'true.'

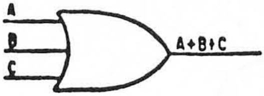
The logical symbol for a two-input OR-gate is shown in Figure 2.4, together with the symbol for a 3-input OR.

<<<<<<<<<<<<FIGURE 2.4A>>>>>>>>>>>>>>>
<<<<<<<<2-INPUT OR GATE SYMBOL>>>>>>>>>



>>

<<<<<<<<<<<<FIGURE 2.4B>>>>>>>>>>>>>>>>>
<<<<<3-INPUT OR GATE SYMBOL>>>>>>>>>>>>



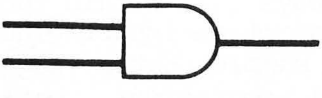
<<<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>

In the last example, we described how a logical output was based upon the truth of one OR another input. Frequently we wish to base some output upon the simultaneous truth of two inputs. For example:

> If (a train is coming) AND (the safety barrier is up), then (lower the safety barrier).
>
> If (A is true) AND (B is true) then (Z is true).

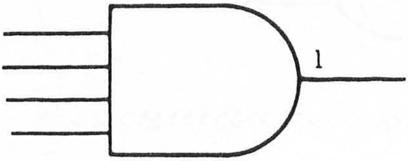
As in the case of the OR gate, we could just as easily base the truth of an output upon the simultaneous truth of three (or many more) inputs. Once again, the AND-gate is a readily available electronic circuit, supplied with two or more inputs as desired. The standard logic symbols for both two and three input AND-gates are shown below in Figure 2.5.

<<<<<<<<<<<<<<FIGURE 2.5A>>>>>>>>>>>>>>>>>
<<<<<<<<SYMBOL FOR 2-INPUT AND GATE>>>>>>>
<<<<<<<<<<<<<<<TITLES>>>>>>>>>>>>>>>>>>>>>



<<<<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<FIGURE 2.5B>>>>>>>>>>>>>>>>>
<<<<<<<<<SYMBOL FOR 4-INPUT AND GATE>>>>>>>
<<<<<<<<<<<<<<<<TITLES>>>>>>>>>>>>>>>>>>>>>



In summary, we have presented three principle types of logic gates. These are the AND, OR, and NOT gates. Each of these gates is readily available, usually packaged as several gates within a single plastic or ceramic cube, with input and output wires protruding in neat rows. In addition to the input and output wires, each package has at least two wires which must be connected to a source of power in order to operate it's internal circuitry. In the very common "Transistor-Transistor-Logic" (or "TTL") family which we describe, the inputs recognize voltages above 2 volts as a "true" or "1." The inputs recognize voltages below about 1/2 volt as "false" or "0." The voltages in the "no man's land" between 1/2 volt and 2 volts are illegal, and result in unpredictable performance of the gate circuit. Furthermore, voltages less than 0 (negative voltages), and voltages greater than 5 volts are excessive, and will damage the inputs. When a gate senses that it should send it's output high (or true), it will force the output to some voltage in the legal region between 2 and 5 volts. Otherwise the gate holds the output false, with a voltage between 0 and about 1/2 volt. Note that the output levels of a gate will always fall within the legal, recognizable voltage areas of an input. Thus it is possible to chain these simple gates together to perform complex logical operations built upon combinations of OR's, AND's, and NOT's acting upon some initial input(s).

Problem: Given four logic inputs A, B, C, and D, which are available on four wires within a computer, design a circuit which will set one logic output true if and only if ABCD=1010. (i.e. A=1, B=0, etc.).

Solution: Let's call our final output 'Z'. We wish to build a circuit such that:

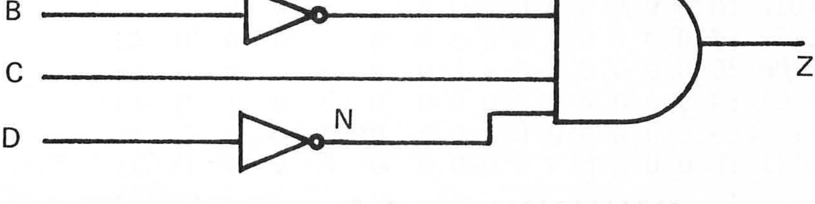
```
IF (A IS TRUE ), AND
   (B IS FALSE), AND
   (C IS TRUE ), AND
   (D IS FALSE), THEN (Z IS TRUE)
```

The B and D terms make it impossible to solve this problem with only a four-input AND-gate. However, if we put inverters on B and D then we might define two new signals:

M=NOT-B (i.e. M is the inverse of B).
N=NOT-D
We use these signals to write:

```
IF (A IS TRUE ), AND
   (M IS TRUE ), AND
   (C IS TRUE ), AND
   (N IS TRUE ), THEN (Z IS TRUE)
```

Our design uses two inverters to derive M and N from B and D respectively. M, N, A, and C are then combined with a four-input AND-gate. This combination is shown in Figure 2.6.



<<<<<<<<<<<<<FIGURE 2.6>>>>>>>>>>>>>>>>>
<<COMBINATIONAL LOGIC EXAMPLE SKETCH>>>>

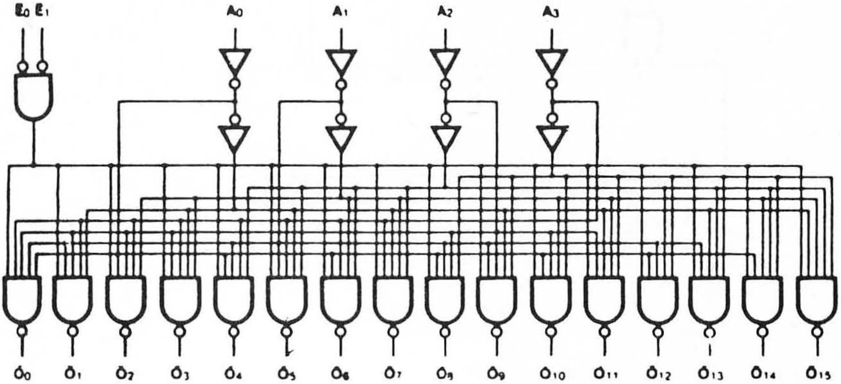
Figure 2.6 is an example of a decoder circuit. The circuit decodes a complex input, and generates a particular output for one possible state of the input. If we regard the four-bit input ABCD as a four bit binary number, then our decoder circuit decodes a count of ten. (Binary 1010). Recall that a four-bit binary number has sixteen possible combinations, zero thru fifteen. It is perfectly possible to design a decoder with four input lines, and sixteen outputs. Each output would represent exactly one of the sixteen possible combinations of the four-bit binary input. Since the input must, of course, be in one and only one of these possible states, it follows that one and only one of the output pins will be true at any one time. Figure 2.7 contains a truth table for such a circuit. Figure 2.8 contains a circuit diagram. The inputs are labeled ABCD, and the sixteen outputs are labeled Y0 thru Y15.

TRUTH TABLE: 4-INPUT 16-OUTPUT DECODER

INPUT	OUTPUTS Y-															
ABCD	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0001	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0010	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0011	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0100	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0101	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0110	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0111	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1000	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1001	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1010	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1011	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1100	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1101	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

FIGURE 2.7

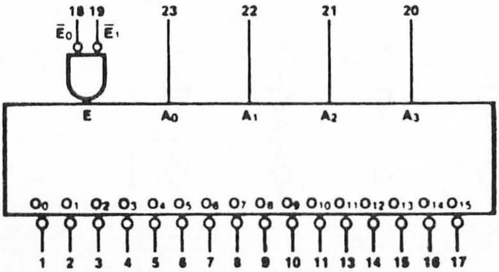
<<<<<<<<<<<<<<<FIGURE 2.8>>>>>>>>>>>>>>>
<<<CIRCUIT DIAGRAM. 4 TO 16 DECODER>>>>>>
<<<<<<<<<<POSITIVE LOGIC OUTPUTS>>>>>>>>>>



<<<<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>

Decoders such as the one shown in Figure 2.8 are available within a single package. Such a package measures about 2/3 inch wide, 2-1/2 inches long, and 1/8 inch high. There are 24 pins extending from the package. These connections consist of the 4 main inputs, 16 outputs, 2 power supply connections, and 2 "enable" inputs. Both of the enable inputs must be true, else NONE of the outputs will go true, irrespective of the state of the 4 main inputs. Smaller packages are available which function as 3-to-8 decoders and 2-to-4 decoders. The outputs of these devices are often inverted by comparison with the decoder example above. (i.e. The one and only selected output will be "low", and all others will be "high"). Figure 2.9 shows a sketch of a typical TTL integrated circuit containing a few logic gates.

<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>
<<<<<<<<<FIGURE 2.9>>>>>>>>>>>>>>>>
<<<<<TTL PACKAGE SKETCH>>>>>>>>>>>>
<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>>>



V_{CC} = Pin 24
GND = Pin 12

DECODERS AND MEMORY...

Decoders are important to the operation of the memory arrays in your computer. Memory consists of a large number of locations wherein the computer may store or recall either "1's", or "0's", as needed. In "8-bit" computers, these locations are grouped into sets of 8-bit BYTES as mentioned in chapter one. Each byte has a unique "ADDRESS", often compared to a post office box number.

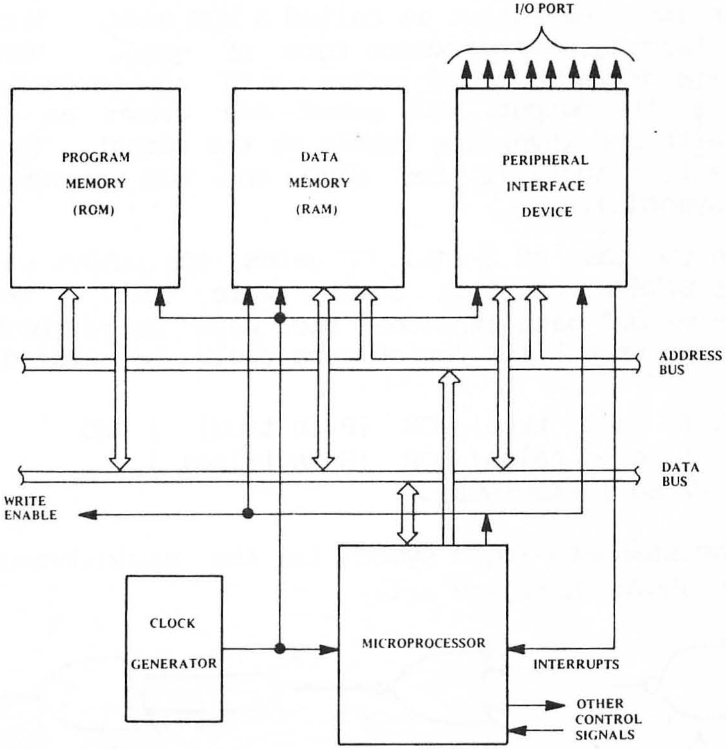
The computer's central processing unit (CPU) accesses a particular byte via the following process.

1. CPU sets a READ/WRITE control line to the proper state (high or low) to indicate a read memory or write to memory operation.

2. CPU outputs the unique address of the byte in question. The address is output in binary form onto a set of wires called "the ADDRESS BUS." Most small microcomputers use a sixteen wire address bus.

There are 65536 possible combinations of the sixteen address lines, meaning that the CPU is capable of distinguishing and controlling 65536 bytes of information. (Or 8 X 65536 = 524288 bits). a 16-to-65536 decoder. Most of this decoding is accomplished inside the memory integrated circuits, so it is not nessesary to imagine an integrated circuit with over 65000 pins protruding! In the case of a read operation, this decoder allows the 8 bits contained in a single location to be output to the CPU via a set of 8 wires called "the DATA BUS." In the case of a write operation, data passes FROM the CPU INTO the 8 bits of memory indicated by the address bus.

<<<<<FIGURE 2.10 CPU BUS SYSTEM>>>>>



<<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>>>>

NAND, NOR, AND EXCLUSIVE-OR GATES...

Consider the effect of adding an inverter to the output of an AND gate. If we call the two inputs A and B, and the final output Z, then we might describe the resulting logic function as:

If (A is true) AND (B is true),
Then (Z is FALSE).

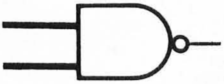
We call this logic function a "NAND GATE". We might write Z = A NAND B in this case. If we added yet another inverter, we would be back to a simple AND function. It turns out that it is easier to make NAND gates than AND gates. For this reason NAND gates are cheaper and more common.

As in the case of the NAND gate, an OR gate with an inverted output is called a NOR gate. Once again, this is a very common form of gate. NAND gates are drawn as AND gates with an inversion bubble at the output. NOR gates are drawn as OR gates with and inversion bubble at the output. (See Figures 2.11 and 2.12 for NAND and NOR standard logic symbols).

In the case of 2-input OR gates, the output was true if EITHER or BOTH inputs were true. The "exclusive-OR" gate excludes the case where BOTH inputs are true. Its performance could be stated:

If ((A is true) OR (B is true)) AND
((A is false) OR (B is false)),
Then (Z IS TRUE).

The standard logic symbol for the exclusive-OR gate is shown in Figure 2.13.

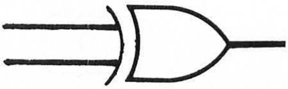


NAND

Fig. 2. 11

NOR

FIG. 2.12



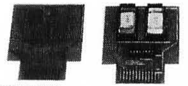
EXCLUSIV OR

Fig. 2. 13


GAMES
ATARI


Microcomputer
Hardware
Handbook


the custom apple
& OTHER MYSTERIES



HOFACKER

HOFACKER

HOFACKER

HOFACKER

HOFACKER

HOFACKER

HOFACKER

HOFACKER