Appendix 1: 6502 Instruction Set

It is convenient to divide up the 56 instructions into four groups, depending upon how many of the bits in their op codes are fixed.

Group 1: Five Bits Fixed

Instructions in this group have fixed (f) and variable (v) bits as follows

fffvvvff

There are two subgroups to consider.

Group 1A: 8 Addressing Modes

These modes are

vvv	*Mode*
000	Indexed indirect (see appendix 3)
001	Zero page
010	Immediate (not STA)
011	Absolute
100	Indirect indexed
101	Zero page, indexed X
110	Absolute, indexed Y
111	Absolute, indexed X

The instructions in this group are

ADC, AND, CMP, EOR, LDA, ORA, SBC, STA

Group 1B: 5 Addressing Modes

The modes are

vvv	*Mode*
000	Immediate (not ASL, LSR, ROL, ROR)
001	Zero page
010	Accumulator (not LDX, LDY)
011	Absolute
101	Zero page, indexed X (indexed Y in LDX)
111	Absolute, indexed X (indexed Y in LDX)

The instructions in this group are

ASL, LDX, LDY, LSR, ROL, ROR

Group 2: Six Bits Fixed

There are two subgroups.

Group 2A

These have fixed (f) and variable (v) bits as follows

fffvvfff

The addressing modes are

vvv	*Mode*
00	Zero page
01	Absolute
10	Zero page, indexed X (indexed Y in STX)
11	Absolute, indexed X (not STX, STY)

The instructions in this group are

DEC, INC, STX, STY

Group 2B

These have fixed (f) and variable (v) bits as follows

ffffvvff

The addressing modes are

vv	*Mode*
00	Immediate
01	Zero page
11	Absolute

The instructions in this group are

CPX, CPY

Group 3: Seven Bits Fixed

There are two subgroups.

Group 3A

This has a fixed (f) and variable (v) bit pattern of

ffffvfff

The modes are

v	*Mode*
0	Zero page
1	Absolute

The only instruction in this group is BIT.

Group 3B

This has a fixed (f) and variable (v) bit pattern of

ffvfffff

The modes are

v	*Mode*
0	Absolute
1	Indirect

The only instruction in this group is JMP.

Group 4: All Bits Fixed

These are the implied and relative addressing mode instructions

BCC, BCS, BEQ, BMI, BNE, BPL, BRK, BVC, BVS, CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, TYA. Also the absolute mode for JSR.

As an example of this, consider LDA. This is in group 1A, and the fixed bits are 101vvv01. Taking each set of values for vvv in turn we arrive at the op codes A1, A5, A9, AD, B1, B5, B9 and BD.

In the detailed summary that follows, the fixed bytes will be given for each mnemonic, and then each addressing mode will have attached its own variable bits. This information is useful if one wishes to construct an assembler or disassembler, for example.

Abbreviations in Table A1.1

*	Plus 1 cycle if page boundary crossed
†	Plus 1 cycle if branch occurs; plus 2 cycles if branch crosses into another page
n	Number of bytes comprising the op code and operand
t	Number of machine cycles needed to complete instruction
v	A variable bit in the op code
M	An arbitrary memory location (that is, an address)
(M)	The contents of M
$\underline{M_6}$	The contents of bit 6 of M
$\overline{M}$	The one's complement of (M)
r	A signed byte (that is, &00 to &7F is +0 to +127; &80 to &FF is −128 to −1)
LOOP	An arbitrary label (that is, an address)
N	The negative flag
Z	The zero flag
C	The carry flag
V	The overflow flag
I	The interrupt disable flag
D	The decimal mode flag
B	The break flag
A	The accumulator
X,Y	The index registers
P	The processor status register
S	The stack pointer
PC	Program counter (containing the address of the *first byte of the instruction*)
→	Copy to memory location or register
↑	Copy to stack (that is, push)
↓	Transfer from stack (that is, pull)
∨	OR
∧	AND
∀	Exclusive-OR
⊕	Signed addition (that is, second byte is treated as a signed byte)
ᵥ	Flag is affected by instruction
—	Flag is not affected by instruction

Table A.1.1 Alphabetical summary of instruction set

ADC

Description of ADC M

Add the contents of M to the accumulator, together with any carry bit. Store the result in the Accumulator and any carry in the carry flag.

Symbolic operation of ADC M

$A + (M) + C \rightarrow A$

$Carry \rightarrow C$

Flags affected

N	Z	C	V
✓	✓	✓	✓

Fixed bit pattern

011VVV01

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	∩	69	2	65	2	6D	3	75	2	7D	3	79	3	71	2			61	2
t	VVV	2	010	3	001	4	011	4	101	4*	111	4*	110	5*	100			6	000

AND

Description of AND M

Perform the logical AND operation bit by bit on the corresponding bits of (M) and the accumulator, leaving the results in the accumulator

Symbolic operation of AND M

$A \wedge (M) \rightarrow A$

∧	0	1
0	0	0
1	0	1

Flags affected

N	Z	C	V
✓	✓	–	–

Fixed bit pattern

001 VVV01

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	∩	29	2	25	2	2D	3	35	2	3D	3	39	3	31	2			21	2
t	VVV	2	010	3	001	4	011	4	101	4*	111	4*	110	5*	100			6	000

ASL

Description of ASL M

Move the contents of M left one bit : bit 7 goes into carry, Zero goes into bit 0.
The result is in M.

Symbolic operation of ASL M

Flags affected

N	Z	C	V
✓	✓	✓	–

Fixed bit pattern

000VVV10

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	∩			06	2	0E	3	16	2	1E	3					0A	1		
t	VVV			5	001	6	011	6	101	7	111					2	010		

BCC

Description of BCC Loop	Symbolic operation of BCC r	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
If C=0, branch to the instruction labelled LOOP	If C = 0: PC + 2 ⊕ r → PC If C = 1: no operation	Relative	90	2	2†	–	–	–	–

BCS

Description of BCS Loop	Symbolic operation of BCS r	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
If C = 1, branch to the instruction labelled LOOP	If C = 1: PC + 2 ⊕ r → PC If C = 0 no operation	Relative	B0	2	2†	–	–	–	–

BEQ

Description of BEQ Loop	Symbolic operation of BEQ r	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
If Z = 1, branch to the instruction labelled LOOP	If Z = 1: PC + 2 ⊕ r → PC If Z = 0 no operation	Relative	F0	2	2†	–	–	–	–

BIT

Description of BIT M	Symbolic operation of BIT M	Flags affected N	Z	C	V	Fixed bit pattern
The logical AND of M and A is performed, the result not being stored. Z is set to 1 if the result is zero, otherwise Z is zero. Finally bits 6 and 7 of M are copied to V and N. A is unchanged.	$\overline{A \wedge (M)} \rightarrow Z$ $M_6 \rightarrow V$ $M_7 \rightarrow N$	M7	✓	–	M6	0010V100

		Immediate		Zero Page		Absolute		Zero page X	Absolute X		Absolute Y	(Indirect) Y		Accumulator	(Indirect X)	
OP CODE	n			24	2	2C	3									
t	v			3	0	4	1									

BMI

Description of BMI LOOP	Symbolic operation of BMI r	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
If N=1, branch to the instruction labelled LOOP	If N=1: PC+2⊕r → PC If N=0: no operation	Relative	30	2	2†	–	–	–	–

BNE

Description of BNE LOOP	Symbolic operation of BNE r	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
If Z=0, branch to the instruction labelled LOOP	If Z=0: PC+2⊕r → PC If Z=1: no operation	Relative	D0	2	2†	–	–	–	–

BPL

Description of BPL LOOP	Symbolic operation of BPL r	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
If N=0, branch to the instruction labelled LOOP	If N=0: PC+2⊕r → PC If N=1: no operation	Relative	10	2	2†	–	–	–	–

BRK

Description of BRK	Symbolic operation of BRK	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
A software interrupt	↑ PC+2 S-2→S ↑ P S-1→S (FFFF ; FFFE)→PC	Implied	00	1	7	–	–	–	–

B is set to 1 before P is pushed onto the stack.

I is set to 1 after P is pushed onto the stack.

BVC

Description of BVC LOOP	Symbolic operation of BVC r	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
If V = 0, branch to the instruction labelled LOOP	If V = 0 : PC + 2 ⊕ r → PC If V = 1 : no operation	Relative	50	2	2†	–	–	–	–

BVS

Description of BVS LOOP	Symbolic operation of BVS r	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
If V = 1, branch to the instruction labelled LOOP	If V = 1 : PC + 2 ⊕ r → PC If V = 0 : no operation	Relative	70	2	2†	–	–	–	–

CLC

Description of CLC	Symbolic operation of CLC	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The carry flag is Set to Zero (cleared)	0 → C	Implied	18	1	2	–	–	0	–

CLD

Description of CLD	Symbolic operation of CLD	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The decimal flag is Set to 0 (cleared). All arithmatic will now be standard binary	0 → D	Implied	D8	1	2	–	–	–	–

D is set to Zero

CLI

Description of CLI	Symbolic operation of CLI	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The interrupt mask bit is set to Zero, so enabling interupts.	0 → I	Implied	58	1	2	–	–	–	–

I is set to 0

CLV

Description of CLV	Symbolic operation of CLV	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The overflow flag is set to Zero (cleared)	0 → V	Implied	B8	1	2	–	–	–	0

CMP

Description of CMP M	Symbolic operation of CMP M	Flags affected N	Z	C	V	Fixed bit pattern
(M) is subtracted from A, but the result is not stored and A is unchanged. Z is 1 on equality, C is 1 if (M) does not exceed A, N is 1 if bit 7 of the result is 1.	A – (M) → result (not stored) if A < (M), Z=0, C=0 if A = (M), Z=1, C=1 if A > (M), Z=0, C=1 } N = result7	✓	✓	✓	–	110VVV01

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	^	C9	2	C5	2	CD	3	D5	2	DD	3	D9	3	D1	2			C1	2
t	vvv	2	010	3	001	4	011	4	101	4*	111	4*	110	5*	100			6	000

CPX

Description of CPX M	Symbolic operation of CPX M	Flags affected N	Z	C	V	Fixed bit pattern
(M) is subtracted from X, but the result is not stored and X is unchanged. Z is 1 on equality, C is 1 if (M) does not exceed X. N is 1 if bit 7 of the result is 1.	X – (M) → result (not stored) if X < (M), Z=0, C=0 if X = (M), Z=1, C=1 if X > (M), Z=0, C=1 } N = result7	✓	✓	✓	–	1110VV00

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	^	E0	2	E4	2	EC	3												
t	vv	2	00	3	01	4	11												

CPY

Description of CPY M	Symbolic operation of CPY M	Flags affected				Fixed bit pattern
(M) is subtracted from Y but the result is not stored and Y is unchanged. Z is 1 on equality, C is 1 if (M) does not exceed X. N is 1 if bit 7 of the result is 1.	Y−(M) → result (not stored) if Y<(M), Z=0, C=0 if Y=(M), Z=1, C=1 if Y>(M), Z=0, C=1 } N = result 7	N	Z	C	V	1100VV00
		✓	✓	✓	–	

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	n	C0	2	C4	2	CC	3												
t	vv	2	00	3	01	4	11												

DEC

Description of DEC M	Symbolic operation of DEC M	Flags affected				Fixed bit pattern
The contents of M is decreased by 1. (If (M) is zero it becomes & FF.) The result is stored in M.	(M) − 1 → M	N	Z	C	V	110VV110
		✓	✓	–	–	

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	n			C6	2	CE	3	D6	2	DE	3								
t	vv			5	00	6	01	6	10	7	11								

DEX

Description of DEX	Symbolic operation of DEX	Addressing mode	Opcode	n	t	Flags affected			
The contents of X is reduced by 1. (If X is zero it becomes & FF.)	X − 1 → X	Implied	CA	1	2	N	Z	C	V
						✓	✓	–	–

DEY

Description of DEY	Symbolic operation of DEY	Addressing mode	Opcode	n	t	Flags affected			
The contents of Y is reduced by 1. (If Y is zero it becomes & FF.)	Y − 1 → Y	Implied	88	1	2	N	Z	C	V
						✓	✓	–	–

EOR

Description of EOR M	Symbolic operation of EOR M	Flags affected				Fixed bit pattern
Perform the exclusive-OR operation bit by bit on the corresponding bits of (M) and the accumulator, leaving the result in the accumulator.	A ⊕ (M) → A	N	Z	C	V	010VVV01
		✓	✓	–	–	

⊕	0	1
0	0	1
1	1	0

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	n	49	2	45	2	4D	3	55	2	5D	3	59	3	51	3			41	2
t	vvv	2	010	3	001	4	011	4	101	4*	111	4*	110	5*	100			6	000

INC

Description of INC M	Symbolic operation of INC M	Flags affected				Fixed bit pattern
The contents of M is increased by 1. (If (M) is &FF it becomes zero.) The result is stored in M.	(M)+1 → M	N	Z	C	V	111VV110
		✓	✓	–	–	

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	n			E6	2	EE	3	F6	2	FE	3								
t	vv			5	00	6	01	6	10	7	11								

INX

Description of INX	Symbolic operation of INX	Addressing mode	Opcode	n	t	Flags affected			
The contents of X is increased by 1. (If X is &FF it becomes zero).	X+1 → X	Implied	E8	1	2	N	Z	C	V
						✓	✓	–	–

INY

Description of INY	Symbolic operation of INY	Addressing mode	Opcode	n	t	Flags affected			
The contents of Y is increased by 1. (If Y is &FF it becomes zero).	Y+1 → Y	Implied	C8	1	2	N	Z	C	V
						✓	✓	–	–

JMP

Description of JMP LOOP	Symbolic operation of JMP LOOP	Flags affected				Fixed bit pattern
		N	Z	C	V	
The address represented by the label LOOP is loaded into the program counter causing a jump to occur to the instruction at that address.	LOOP → PC	–	–	–	–	01V011OO

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)		Indirect	
OP CODE	n					4C	3													6C	3
t	v					3	0													5	1

JSR

Description of JSR LOOP	Symbolic operation of JSR LOOP	Addressing mode	Opcode	n	t	Flags affected			
						N	Z	C	V
The program counter plus 2 (the address minus one of the instruction following the JSR) is saved on the stack. The address represented by LOOP is loaded into the program counter.	PC + 2 ↓ LOOP → PC	Absolute	20	3	6	–	–	–	–

LDA

Description of LDA M	Symbolic operation of LDA M	Flags affected				Fixed bit pattern
		N	Z	C	V	
The contents of M is copied into the accumulator.	(M) → A	✓	✓	–	–	101VVV01

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	n	A9	2	A5	2	AD	3	B5	2	BD	3	B9	3	B1	2			A1	2
t	vvv	2	010	3	001	4	011	4	101	4*	111	4*	110	5*	100			6	000

LDX

Description of LDX M	Symbolic operation of LDX M	Flags affected				Fixed bit pattern
The contents of M is copied into X.	(M) → X	N	Z	C	V	101VVV10
		✓	✓	—	—	

		Immediate		Zero Page		Absolute		Zero page Y		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	∩	A2	2	A6	2	AE	3	B6	2			BE	3						
t	vvv	2	000	3	001	4	011	4	101			4*	111						

LDY

Description of LDY M	Symbolic operation of LDY M	Flags affected				Fixed bit pattern
The contents of M is copied into Y.	(M) → Y	N	Z	C	V	101VVV00
		✓	✓	—	—	

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	∩	A0	2	A4	2	AC	3	B4	2	BC	3								
t	vvv	2	000	3	001	4	011	4	101	4*	111								

LSR

Description of LSR M	Symbolic operation of LSR M	Flags affected				Fixed bit pattern
Move the contents of M right one bit: bit 0 goes into carry. Zero goes into bit 7. The result is in M.	$M_7\ M_6\ M_5\ M_4\ M_3\ M_2\ M_1\ M_0$ → 0; → C	N	Z	C	V	010VVV10
		0	✓	✓	—	

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	∩			46	2	4E	3	56	2	5E	3					4A	1		
t	vvv			5	001	6	011	6	101	7	111					2	010		

NOP

Description of NOP	Symbolic operation of NOP	Addressing mode	Opcode	n	t	Flags affected			
Does Nothing for 2 cycles.	—	Implied	EA	1	2	N	Z	C	V
						—	—	—	—

ORA.

Description of ORA M	Symbolic operation of ORA M	Flags affected				Fixed bit pattern
Perform the inclusive-OR operation bit by bit on the corresponding bits of (M) and the accumulator, leaving the result in the accumulator.	A V (M) → A	N	Z	C	V	000VVV01
		✓	✓	–	–	

V	0	1
0	0	1
1	1	1

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	n	05	2	05	2	0D	3	15	2	1D	3	19	3	11	2			01	2
t	vvv	2	010	3	001	4	011	4	101	4*	111	4*	110	5*	100			6	000

PHA

Description of PHA	Symbolic operation of PHA	Addressing mode	Opcode	n	t	Flags affected			
The contents of the accumulator is copied to the stack, and the stack pointer is decreased by one.	↓A S−1→S	Implied	48	1	3	N	Z	C	V
						–	–	–	–

PHP

Description of PHP	Symbolic operation of PHP	Addressing mode	Opcode	n	t	Flags affected			
The contents of the status register is copied to the stack, and the stack pointer is decreased by 1.	↓P S−1→S	Implied	08	1	3	N	Z	C	V
						–	–	–	–

PLA

Description of PLA	Symbolic operation of PLA	Addressing mode	Opcode	n	t	Flags affected			
The contents of the accumulator is filled by the last byte pushed on the stack, and the stack pointer is increased by one.	↑A S+1→S	Implied	68	1	4	N	Z	C	V
						✓	✓	–	–

PLP

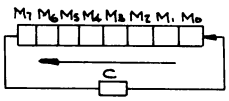
Description of PLP	Symbolic operation of PLP	Addressing mode	Opcode	n	t	Flags affected
The status register is filled with the last byte pushed onto the stack, and the stack pointer is increased by one.	↓ P S+1→S	Implied	28	1	4	N Z C V ✓ ✓ ✓ ✓ B.D and I are also affected.

ROL

Description of ROL M

Move the contents of M left one bit: bit 7 goes into carry after the present contents of carry has gone into bit 0. The result is in M.

Symbolic operation of ROL M



Flags affected

N	Z	C	V
✓	✓	✓	—

Fixed bit pattern: 001VVV10

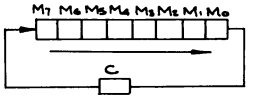
		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	^			26	2	2E	3	36	2	3E	3					2A	1		
t	vvv			5	001	6	011	6	101	7	111					2	010		

ROR

Description of ROR M

Move the contents of M right one bit: bit 0 goes into carry after the present contents of carry has gone into bit 7. The result is in M.

Symbolic operation of ROR M



Flags affected

N	Z	C	V
✓	✓	✓	—

Fixed bit pattern: 011VVV10

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	^			66	2	6E	3	76	2	7E	3					6A	1		
t	vvv			5	001	6	011	6	101	7	111					2	010		

RTI

Description of RTI	Symbolic operation of RTI	Addressing mode	Opcode	n	t	Flags affected
Return to main program after an interrupt has been serviced. The status register and program counter are restored from the stack, and the stack pointer is adjusted.	↓ P S+1→S ↓ PC S+2→S	Implied.	40	1	6	N Z C V ✓ ✓ ✓ ✓ B.D and I are also affected.

RTS

Description of RTS	Symbolic operation of RTS	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
Return to calling program from a subroutine. The return is to the instruction following the call (ie. following JSR). The program counter is restored from the stack and incremented by 1. The stack pointer is adjusted.	↓ PC S + 2 → S PC + 1 → PC	Implied	60	1	6	–	–	–	–

SBC

Description of SBC M	Symbolic operation of SBC M	Flags affected N	Z	C	V	Fixed bit pattern
Subtract the contents of M together with any borrow from the accumulator. The result is left in the accumulator and any borrow in the carry flag.	A − (M) − $\bar{C}$ → A $\overline{\text{Borrow}}$ → C	✓	✓	✓	✓	111vvv01

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	^	E9	2	E5	2	ED	3	F5	2	FD	3	F9	3	F1	2			E1	2
t	vvv	2	010	3	001	4	011	4	101	4*	111	4*	110	5*	100			6	000

SEC

Description of SEC	Symbolic operation of SEC	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The carry flag is set to one	1 → C	Implied	38	1	2	–	–	1	–

SED

Description of SED	Symbolic operation of SED	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The decimal flag is set to one. All arithmetic is now in BCD (ie. ADC and SBC operate according to BCD)	1 → D	Implied	F8	1	2	–	–	–	–

D is set to 1.

SEI

Description of SEI	Symbolic operation of SEI	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The interrupt mask bit is set to 1, so disabling interrupts.	1 → I	Implied	78	1	2	–	–	–	–

I is set to 1.

STA

Description of STA M	Symbolic operation of STA M	Flags affected				Fixed bit pattern
		N	Z	C	V	
The contents of the accumulator is put into the location M.	A → M	–	–	–	–	100VVV01

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	n			85	2	8D	3	95	2	9D	3	99	3	91	2			81	2
t	VVV			3	001	4	011	4	101	5	111	5	110	6	100			6	000

STX

Description of STX M	Symbolic operation of STX M	Flags affected				Fixed bit pattern
		N	Z	C	V	
The contents of X is put into the location M.	X → M	–	–	–	–	100VV110

		Immediate		Zero Page		Absolute		Zero page Y		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	n			86	2	8E	3	96	2										
t	VV			3	00	4	01	4	10										

STY

Description of STY M	Symbolic operation of STY M	Flags affected				Fixed bit pattern
		N	Z	C	V	
The contents of Y is put into the location M.	Y → M	–	–	–	–	100VV100

		Immediate		Zero Page		Absolute		Zero page X		Absolute X		Absolute Y		(Indirect) Y		Accumulator		(Indirect X)	
OP CODE	n			84	2	8C	3	94	2										
t	VV			3	00	4	01	4	10										

TAX

Description of TAX	Symbolic operation of TAX	Addressing mode	Opcode	n	t	Flags affected			
						N	Z	C	V
The contents of the accumulator is copied into the X register.	A → X	Implied	AA	1	2	✓	✓	–	–

TAY

Description of TAY	Symbolic operation of TAY	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The contents of the accumulator is copied into the Y register.	A → Y	Implied	A8	1	2	✓	✓	–	–

TSX

Description of TSX	Symbolic operation of TSX	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The contents of the stack pointer is copied into the X register	S → X	Implied	BA	1	2	✓	✓	–	–

TXA

Description of TXA	Symbolic operation of TXA	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The contents of the x register is copied to the accumulator	X → A	Implied	8A	1	2	✓	✓	–	–

TXS

Description of TXS	Symbolic operation of TXS	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The contents of the x register is copied to the stack pointer	X → S	Implied	9A	1	2	–	–	–	–

TYA

Description of TYA	Symbolic operation of TYA	Addressing mode	Opcode	n	t	Flags affected N	Z	C	V
The contents of the Y register is copied to the accumulator.	Y → A	Implied	98	1	2	✓	✓	–	–

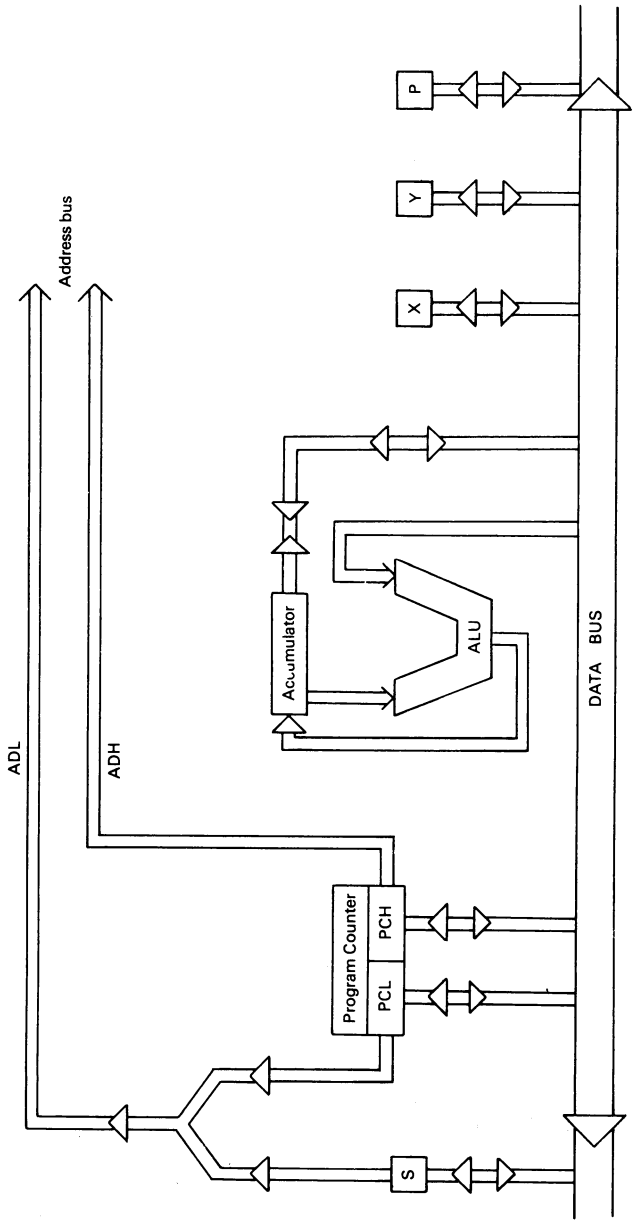
Table A1.2 Instruction set in numerical order of opcodes

In this table the following abbreviations are used:

ZP	Zero page addressing mode
Abs	Absolute addressing mode
Imm	Immediate addressing mode
ZP,X ZP,Y	Zero page indexed addressing mode
Abs,X Abs,Y	Absolute indexed addressing mode
(Ind),Y	Indirect indexed addressing mode
(Ind,X)	Indexed indirect addressing mode
(Ind)	Indirect addressing mode
A	Accumulator addressing mode
LSN	Least significant nybble (for example, A in &EA)
MSN	Most significant nybble (for example, E in &EA)
—	Reserved for future expansion

LSN → / MSN ↓	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E
0	BRK	ORA (Ind,X)	—	—	—	ORA ZP	ASL ZP	—	PHP	ORA Imm	ASL A	—	—	ORA Abs	ASL Abs
1	BPL	ORA (Ind),Y	—	—	—	ORA ZP,X	ASL ZP,X	—	CLC	ORA Abs,Y	—	—	—	ORA Abs,X	ASL Abs,X
2	JSR	AND (Ind,X)	—	—	BIT ZP	AND ZP	ROL ZP	—	PLP	AND Imm	ROL A	—	BIT Abs	AND Abs	ROL Abs
3	BMI	AND (Ind),Y	—	—	—	AND ZP,X	ROL ZP,X	—	SEC	AND Abs,Y	—	—	—	AND Abs,X	ROL Abs,X
4	RTI	EOR (Ind,X)	—	—	—	EOR ZP	LSR ZP	—	PHA	EOR Imm	LSR A	—	JMP Abs	EOR Abs	LSR Abs
5	BVC	EOR (Ind),Y	—	—	—	EOR ZP,X	LSR ZP,X	—	CLI	EOR Abs,Y	—	—	—	EOR Abs,X	LSR Abs,X
6	RTS	ADC (Ind,X)	—	—	—	ADC ZP	ROR ZP	—	PLA	ADC Imm	ROR A	—	JMP (Ind)	ADC Abs	ROR Abs
7	BVS	ADC (Ind),Y	—	—	—	ADC ZP,X	ROR ZP,X	—	SEI	ADC Abs,Y	—	—	—	ADC Abs,X	ROR Abs,X
8	—	STA (Ind,X)	—	—	STY ZP	STA ZP	STX ZP	—	DEY	—	TXA	—	STY Abs	STA Abs	STX Abs
9	BCC	STA (Ind),Y	—	—	STY ZP,X	STA ZP,X	STX ZP,Y	—	TYA	STA Abs,Y	TXS	—	—	STA Abs,X	—
A	LDY Imm	LDA (Ind,X)	LDX Imm	—	LDY ZP	LDA ZP	LDX ZP	—	TAY	LDA Imm	TAX	—	LDY Abs	LDA Abs	LDX Abs
B	BCS	LDA (Ind),Y	—	—	LDY ZP,X	LDA ZP,X	LDX ZP,Y	—	CLV	LDA Abs,Y	TSX	—	LDY Abs,X	LDA Abs,X	LDX Abs,Y
C	CPY Imm	CMP (Ind,X)	—	—	CPY ZP	CMP ZP	DEC ZP	—	INY	CMP Imm	DEX	—	CPY Abs	CMP Abs	DEC Abs
D	BNE	CMP (Ind),Y	—	—	—	CMP ZP,X	DEC ZP,X	—	CLD	CMP Abs,Y	—	—	—	CMP Abs,X	DEC Abs,X
E	CPX Imm	SBC (Ind,X)	—	—	CPX ZP	SBC ZP	INC ZP	—	INX	SBC Imm	NOP	—	CPX Abs	SBC Abs	INC Abs
F	BEQ	SBC (Ind),Y	—	—	—	SBC ZP,X	INC ZP,X	—	SED	SBC Abs,Y	—	—	—	SBC Abs,X	INC Abs,X

Appendix 2: Full Block Diagram of 6502 Architecture



Internal address pathways for the program counter are omitted.

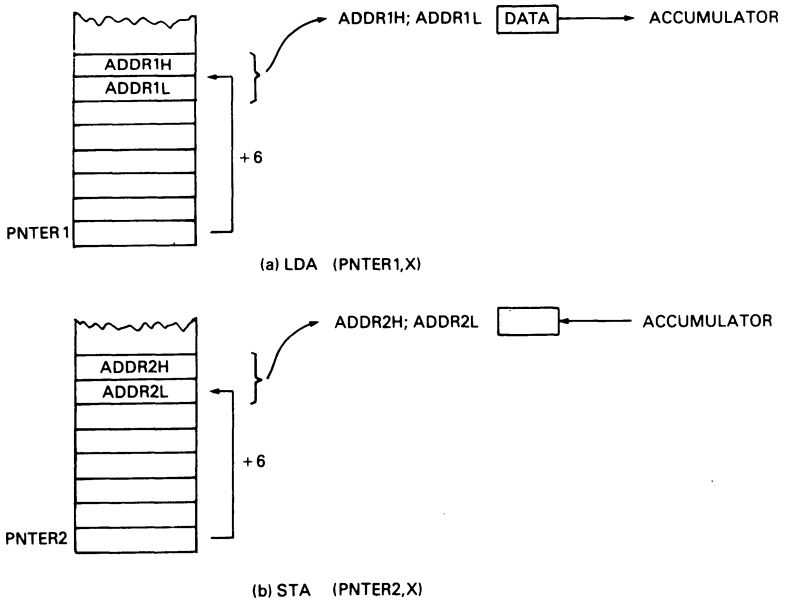
Appendix 3: Indexed Indirect Addressing

There is one more 6502 addressing mode which we have not covered in the book: *indexed indirect addressing*. This omission is quite deliberate, for you are not likely to want to use this mode of addressing in your work on the BBC Micro. The designers of the 6502 included indexed indirect for a very specific purpose: *multiple peripheral programming*.

In this appendix we shall first describe briefly the operation of the indexed indirect mode, and then describe the sort of application for which it is suitable.

Consider a list of pointers stored consecutively in memory. Then the contents of the location to which any pointer is referencing can be loaded into the accumulator by writing LDA (PNTER,X), where PNTER is the base address of the pointers and X is a suitable even number. Similarly STA (PNTER,X) stores a copy of the accumulator in the location pointed to by (PNTER,X). Only the X register can be used for this purpose, just as only the Y register can be used for indirect indexed addressing. Again, like indirect indexed addressing, indexed indirect requires the location PNTER *to be in zero page.*

The diagram below illustrates the pair of instructions LDA (PNTER1,X): STA (PNTER2,X), where X equals 6.



6, the contents of X, is added to the base address PNTER1 to give the location containing the low byte of the address, the content of which is ADDR1L. The next location will always contain the high byte of the address, in (a) ADDR1H. Hence the contents of the address ADDR1H; ADDR1L are put into the accumulator. Similarly, in (b) the accumulator is put into ADDR2H; ADDR2L.

Now, the indexed indirect addressing mode cannot be usefully used to access strings in memory by referring to a list of pointers to those strings, such as we constructed in section 7.6. The reason for this is that each character of the string could then be accessed only by using ordinary indexed addressing, and we cannot combine both modes in the same instruction. That is, we cannot write LDA (BASE,X),Y (it is unfortunate that we cannot do this—it would be a most powerful combination). Because most lists of pointers in the sort of assembly programs that we are likely to write will be used to access a base address from which we will index, indexed indirect is not of much use to us. The methods used in section 7.6 are still the best ones to use in these cases.

However, it is worth understanding the sort of applications where indexed indirect is useful—the sort of application, indeed, for which the addressing mode was designed. Consider a whole series of peripheral devices, say teletypes, each of which will be serviced by one 6502 microprocessor. Each teletype is connected up to its own specific VIA port. Each teletype is sending a message to the microprocessor, which will be terminated by a carriage return. As each character of the message is ready to be processed, the VIA to which the teletype is connected will interrupt the 6502. At this stage, the microprocessor will enter an interrupt service routine which will interrogate in turn the status registers of the VIAs to see which teletype has sent a character. Since teletypes are very slow, the order of polling is of no consequence: any multiple interrupt would be dealt with quite transparently to the user of any of the teletypes.

Each teletype has a small section of memory reserved for it which acts as a buffer. We shall assume here that the buffer is never overfilled. Five sets of pointers are required for this system, the order for the set of tables being arbitrary, although the order for each table must be the same.

(a) TABLE1	The addresses of the input register for the incoming character from the teletype
(b) TABLE2	The addresses of the buffers for the teletypes
(c) TABLE3	The addresses of output to the teletypes for acknowledgement
(d) STATUS	Addresses of the status registers for the teletypes
(e) COPYTABLE2	A copy of TABLE2

(a)–(d) must be in zero page, (e) can be anywhere. The function of TABLE3 needs to be explained. When a key is pressed on a teletype, the

action of the print-hammer is caused not directly by this key but by the computer reflecting the key. In this way, an instant verification is performed to confirm that the correct data has been received.

Here, now, is the simplified code for performing this polling sequence (we simplify by ignoring parity checking, among other things)

```
 1                LDX   #2 * (NUMBER - 1)
 2 LOOP1          LDA   (STATUS,X)
 3                BMI   STOREDATA
 4                DEX
 5                DEX
 6                BPL   LOOP1
 7                BMI   OUT
 8 STOREDATA      LDA   (TABLE1,X)
 9                STA   (TABLE3,X)
10                CMP   #&0D
11                BEQ   ENDSTRING
12                STA   (TABLE2,X)
13                INC   TABLE2,X
14                BNE   OUT
15                INC   TABLE2+1,X
16                BNE   OUT
17 ENDSTRING      LDA   COPYTABLE2,X
18                STA   TABLE2,X
19                LDA   COPYTABLE2+1,X
20                STA   TABLE2+1,X
21                JSR   ANALYSE
22 OUT                  .
                        .
                        .
                        .
                        .
```

In line 1, NUMBER is the number of teletypes connected. The beauty of this program is that it will work for up to 32 teletypes, with no change whatsoever being required in the program ($32 \times 2 \times 4 = 256$, the limit of zero page).

In lines 2 to 7 we examine the status register of each VIA in turn, beginning with the one at the top of the STATUS list. If bit 7 is 1, this indicates that the teletype to which this VIA is attached has sent a character, and we go to line 8 to process it. Otherwise, we decrease X by 2, and look at the next status register down the list. If we happen to go through the entire list without finding the source of the interrupt (a 'phantom' interrupt) we go to some suitable exit code at line 22 onwards

Assuming that we find the appropriate teletype, we then load the contents of the input and store it in the buffer (lines 8 and 9). Notice how indexed indirect allows us to recover the appropriate set of pointers by using just one index value (this is why the tables must be arranged in the same order, of course). If we have reached the end of the message (lines 9 and 10) we go to perform some analysis in lines 17–21. Lines 17–20 reset the pointers in TABLE2, which have been altered in lines 13–15, and line 21 jumps to a subroutine which performs some analysis on the basis of the message (and which will output a line feed, when the analysis is complete). During this analysis, interrupts will be enabled so that further input can be received. One function of the ANALYSIS routine will be to deal with the case where 2 or more teletypes have messages to be analysed simultaneously, using some time-sharing principle which need not concern us here.

If the end of the message is not yet reached, the current character will be stored in the buffer (line 12) and then the address of the buffer will be incremented by one to point to the next free space. This is slower and more cumbrous than the indirect indexed method used in section 7.6, but it is suitable in this case since teletypes are relatively slow anyway, and relatively few increments are required (at least compared to the sorting requirements of section 7.6).

The size of this program is very small considering the complex task it performs, and this is due entirely to the use of the indexed indirect mode. The overall speed of processing is very favourable too, and this is why the designers of the 6502 included this addressing mode. Unfortunately, we are unlikely to be able to profit from it on the BBC Micro. Probably the only time that we are likely to use it, is in the case where we want a simple indirect mode and the Y register is not available. In this case, using indexed indirect with X equal to zero will suffice, since LDA (BASE),Y and LDA (BASE,X) give identical results when X and Y are both zero.

Appendix 4: Floating-point Representation

In this book we have considered only the integer (or fixed point) representation of numbers. The discussion of the *floating-point* representation has been outside our scope. However, for the sake of completeness, we will here discuss this representation, although we will not be considering how arithmetic may be performed upon such numbers.

The number four in base two is 100; if we divide by two we obtain 10, or more suggestively 10.0; divide by two again and we get 1.0, which is one, of course. Now it would seem reasonable to write the result of dividing by two again as 0.1, by two yet again at 0.01, and by two still again as 0.001; and so on. Hence 0.1 is ½, 0.01 is ¼, 0.001 is ⅛; and so on. This is *bicimal* representation, the direct counterpart to the base ten decimal; and we refer to the point as the *bicimal point.*

Any decimal can be written in bicimal; and any bicimal in decimal. For example, 0.75 is 0.11 in bicimal; and 0.0101 in bicimal is 0.3125. Now, fractions that can be written as terminating decimals may give recurring bicimals. For example, ⅕ is $0.\dot{0}01\dot{1}$ in bicimal. However, any fraction that terminates in bicimal will terminate in decimal, because all such fractions will have denominators of a power of two, all of which terminate in decimal (just keep halving 0.5 until you get there). It follows that there may be a loss of precision in translating from decimal to bicimal if we cannot use the recurrence notation (the dots over the relevant repeating digits). Moreover, bicimal takes up many more places than decimal, so we may have to round to get our decimal into a fixed number of bicimal places. Hence, we see that a possible error can be introduced in translating from decimal to bicimal (and vice versa if the number of significant figures allocated to decimal output is fixed). This must be borne in mind when dealing with floating-point numbers (in assembler *or* in BASIC), for in certain circumstances these rounding errors can compound considerably, resulting in significant errors.

When storing bicimal numbers in a computer it is convenient to write them first in a *normalised form.* So, we write 11011.01011 as 0.1101101011×2^5, and 0.00010101011 as 0.10101011×2^{-3}, for example. The convention is to move the bicimal point until the most significant digit is the one just after the point: that is, move it right or left until all digits to the left of the point are zero and the first digit to the right of the point is one. The

power of two attached to this adjusted number reflects the number of moves that the bicimal point has had to make. Applying this power to the adjusted number will set the bicimal point back to its correct place (5 places rightwards in the first case, that is, $0.\overbrace{11011}01011 \rightarrow 11011.01011$ as required, and 3 places leftwards in the second). Thus the point is allowed to *float* across so that a normalised form is achieved, and so we call the representation the *floating-point representation.* Whole numbers can also be represented in this way, of course. For example, 110001011 is 0.110001011×2^9.

The BBC Micro and most other microcomputers use 5 bytes to represent such numbers. The least significant byte represents the power, or as it is usually called, the *exponent.* The next four bytes represent the number, or as it is usually called, the *mantissa.* The exponent is in two's complement form with one difference: the sign bit is reversed. Hence an exponent of &90 represents &10 or 16, while &70 represents −&10 or −16. The reason for this is connected with the representation for zero. Clearly zero gives a zero mantissa (which cannot be normalised since there is no one). It is logical to have the minimum exponent associated with this, which is the maximum negative exponent. This is reasonable since a maximum negative exponent is associated with the smallest number that can be represented for any given mantissa. Without the change in the sign bit this would give &80 00 00 00 00; with the change it gives &00 00 00 00 00, which is much more sensible.

Apart from zero, *all* mantissas will have their most significant bits as one. We can therefore *assume* that the most significant bit is one, and use the actual bit in this position to reflect the sign of the number: 0 is positive, 1 is negative.

Figure A4.1 shows the format for a floating-point number: notice that the byte on the extreme left (carrying the exponent) is the *lowest* in memory of the five. Moreover, in the next four, the most significant byte is *lowest* in memory, and the least significant, *highest* in memory. This is in distinction to integer (fixed point) numbers, where the most significant

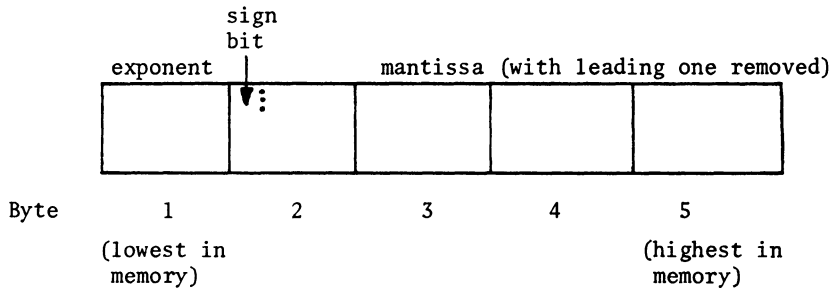


Figure A4.1: Floating point storage in the BBC Micro

byte is the highest in memory. The convention with integers is chosen to fit in with the 6502 convention; with floating-point, there are standard routines for arithmetic and there is no gain in using the specific convention of the 6502 microprocessor.

The largest numerical values are, on the positive side FF 7F FF FF FF and on the negative FF FF FF FF FF (that is, $\pm 1.70141183 \times 10^{38}$ to 9sf, the limit of precision in the BBC Micro). The smallest numerical values (apart from zero) are 00 00 00 00 01 and 00 80 00 00 01 (that is, $\pm 1.46936794 \times 10^{-39}$ to 9sf).

In order to acquaint yourself with this representation, load the monitor (listing 10.6) and use CALLS%,X, with X set at various values. For example

```
X = 2500: CALLS%,X
```

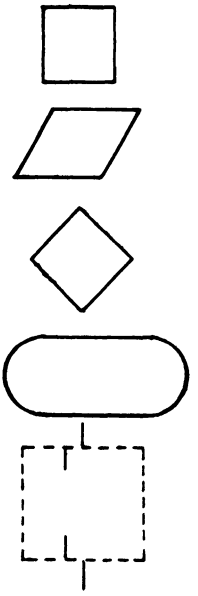
Now type M 0600 and the address pointed to should be &25E4. M 25E4 gives 8C 1C 40 00 00, which is $\&0.9C4 \times 2^{12}$ (that is, $0.1001\ 1100\ 0100 \times 2^{12}$). Now moving the hexadecimal point across each hex digit is equivalent to multiplying by 2^4, so we obtain &9C4, which is indeed 2500.

Again, X = −2500: CALLS%,X, gives 8C 9C 40 00 00, that is, $-\&0.9C4 \times 2^{12}$ or −&9C4, as required. Remember, the sign bit of the second byte is the sign of the whole number.

Finally, X = 0.3175: CALLS%,X gives 7F 20 00 00 00, which is $\&0.A0 \times 2^{-1}$, that is, 0.0101, as required.

Try more yourself—you will soon become very familiar with this form of storage.

Appendix 5: Flowchart Symbols and Conventions Used in This Book



Assignment of values

Input/output

Decision

Start or stop

Structured flowchart symbol:
(i) FOR..... NEXT
(ii) REPEAT..... UNTIL
(iii) REPEAT WHILE

Appendix 6: Further Uses of OPT

You may wish to use your BBC Micro as a development machine to produce machine code programs which you want to put on EPROMs. To do this you will need an EPROM programmer add-on. But this will not be enough: it is likely that you will want your programs on EPROM to occupy space above &8000, the usual ROM and EPROM locations. However, you have no RAM in this area on the BBC Micro, so how can you do it?

The answer is to use OPT4 to OPT7. If you do this you can make use of *two* assembly location variables: P% is still the program counter, but now O% is also available, and this indicates where the code will *physically* go in your BBC Micro. Hence you can set P% above &8000 with O% at a usual RAM location.

As an example, consider listing 4.2 again. Change

20 to 20 P%=&9000:O%=START

and

30 to 30 [OPT7 (the equivalent of OPT3)

and RUN the program.

Notice that it assembles using addresses from &9000 onwards

```
9000 LDA NUM
9002 .BACK etc.
```

However, now type in

```
FOR I%=0 TO 8: PRINT~START?I%:NEXT
```

and note that you get the machine code. Hence, while the assembler uses *addresses* beginning from P%, it assembles *into* the address contained in O%. This is exactly what we require for EPROM programming.

Appendix 7: The User Port

The user port on a model B is part of a 6552 Versatile Interface Adapter (VIA). This is a fairly complex input/output chip, which provides two ports, handshaking, interrupts, two timers and a serial register. It is quite possible to write a book on the applications of this chip, and there is not space here to do this, However, information on the VIA in accessible and digestible form is not easy to come by and, in this appendix, a brief but full account is given of its workings. This should allow you to do most of what you want with the user port.

A7.1 Overview of the 6522 VIA

(This section may be read quickly on first reading, and returned to later.)

The VIA which is used to create the centronics printer port and the user port is memory mapped into the locations &FE60 to &FE6F. Table A7.1 shows the purpose of each location. Since *port A* is used exclusively for the printer, we shall not consider it or its associated control lines in any detail (though if you require up to 10mA of buffered 'sink' current, port A could be used without modification to the circuits).

Table A7.1 User port and printer VIA

Address	*Function*
FE60	Port B
FE61	Port A, with handshaking (printer)
FE62	Data direction register for port B
FE63	Data direction register for port A (printer)
FE64	Timer1 counter, low byte
FE65	Timer1 counter, high byte
FE66	Timer1 latch, low byte
FE67	Timer1 latch, high byte
FE68	Timer2 counter, low byte
FE69	Timer2 counter, high byte
FE6A	Serial shift register
FE6B	Auxiliary control register
FE6C	Peripheral control register
FE6D	Interrupt flag register
FE6E	Interrupt enable register
FE6F	Port A, no handshaking (printer)

Port B at &FE60 has each of its bits connected to a corresponding pin on the user port. There are also two *control lines*, CB1 and CB2, which are connected to the other two pins on the user port: your User Guide contains the relevant circuit diagram. Each bit of the port can either be used for output or for input: the *data direction register* at &FE62 controls this.

The 6522 contains two timers, *timer1* being more complex than *timer2*, and both are available for use. The VIA supports interrupts on both these timers, and also on each of the control lines (CB1 and CB2 for the user port). There is also a *serial register* which can output and accept bits one at a time and shift them accordingly, and this can also generate interrupts. In the BBC computer *all* IRQ interrupts vector first through an address contained in &0204 and &0205 (that is, JMP (&0204) is performed). Interrupts from the user port will not be dealt with by this service routine: instead, the routine is exited with a JMP (&206). Hence, one can write one's own servicing routine for the user port by putting the address of this routine in &206 and &207 (that is, at IRQ2, compare section 9.3). So the user can write his own interrupt service routines in connection with port B, the serial register and the timers. The *interrupt enable register* (&FE6E) controls which interrupts are allowed: the *interrupt flag register* (&FE6D) displays which items are calling for an interrupt (regardless of whether they are enabled), and bit 7 of this register is one if an interrupt is asked for and enabled. Tables A7.2 and A7.3 show these registers.

Table A7.2 Interrupt enable register

7	6	5	4	3	2	1	0
Set or clear control bit	T1	T2	CB1	CB2	SR	CA1	CA2

1 = interrupt enabled; 0 = interrupt disabled (bits 0 to 6)
1 = writing a one sets that bit to 1
0 = writing a one sets that bit to 0
} (bit 7)

Table A7.3 Interrupt flag register

7	6	5	4	3	2	1	0
IRQ	T1	T2	CB1	CB2	SR	CA1	CA2

Bit 7 is 1 if any of bits 0–6 are set to 1 in both this register and the interrupt enable register.

Bits 0–6 are set and cleared by the following operations:

Bit	*Set by*	*Cleared by**
0	Active transition on CA2	Reading or writing &FE61
1	Active transition on CA1	Reading or writing &FE61
2	Completion of 8 shifts (not in free-running mode)	Reading or writing &FE6A
3	Active transition of CB2	Reading or writing &FE60
4	Active transition of CB1	Reading or writing &FE60
5	Time-out of Timer2	Reading &FE68 or writing &FE69
6	Time-out of Timer1	Reading &FE64 or writing &FE65

*Note that interrupt flags can also be cleared by writing 1 into the bit position.

There are two control registers: the *peripheral control register* is concerned with the operation of the four control lines (CA1, CA2, CB1, CB2); the top nybble controls CB1 and CB2, the bottom CA1 and CA2 (which are reserved for the printer port). The *auxiliary control register* determines how the input ports, serial register and the timers behave. Tables A7.4 and A7.5 give the details.

Table A7.4 Peripheral control register (bits 4–7 only; bits 0–3 are identical in function but are for port A)

Bit 4:
- 0 Active transition on CB1 is high to low
- 1 Active transition on CB1 is low to high

} Bit 4 of interrupt flag register is set on active transition

Bits 5–7:
- 000 CB2 handshake input mode
- 001 CB2 independent input mode

} Active transition on CB2 is high to low

- 010 CB2 handshake input mode
- 011 CB2 independent input mode

} Active transition on CB2 is low to high

} Bit 3 of interrupt flag register is set on active transition

- 100 CB2 handshake output mode
- 101 CB2 pulse output mode

} Active transition on CB2 is high to low

- 110 Constant low output on CB2
- 111 Constant high output on CB2

Table A7.5 Auxiliary control register

Bit 0:	0	Disable input latch on port A	
	1	Enable input latch on port A	
Bit 1:	0	Disable input latch on port B	
	1	Enable input latch on port B	
Bits 2–4:	000	Disable shift register	All output on CB2
	001	Shift in at timer2 rate	
	010	Shift in at machine clock rate	
	011	Shift in at external clock rate on CB1	
	100	Free-running output at timer2 rate	
	101	Shift out at timer2 rate	All input on CB2
	110	Shift out at machine clock rate	
	111	Shift out at external clock rate on CB1	
Bit 5:	0	Decrement timer2 in single-interval mode using machine clock	
	1	Decrement timer2 on external pulses via bit 6 of port B	
Bit 6:	0	Single-interval mode on timer1	
	1	Free-running mode on timer1	
Bit 7:	0	Disable output via bit 7 of port B — timer1 only	
	1	Enable output via bit 7 of port B — timer1 only	

A7.2 Configuring the 6522 for input/output

Each bit of each port of a VIA can be programmed to act as an input source or an output source. The data direction registers at &FE62 and &FE63 are used to specify this. A *one* in the relevant position in the data direction register specifies *output* for the corresponding bit in the port. A *zero* in the data direction register specifies *input* for the corresponding bit in the port. Using zero for input is a safeguard, for momentary power failures, faults, resets etc. usually zeroise memory locations, and random output is far more dangerous than random input.

Location &FE63 contains &FF, since all the bits on port A are to be outputs to the printer and this location should not usually be touched by the programmer. Location &FE62 controls the user port and is at the disposal of the programmer. Here are some examples:

(a) All bits inputs:
```
LDA  #0
STA  &FE62
```

(b) Bottom nybble outputs, top nybble inputs:
```
LDA  #&0F
STA  &FE62
```

(c) Odd bits outputs, even bits inputs:
```
LDA  #&AA
STA  &FE62
```

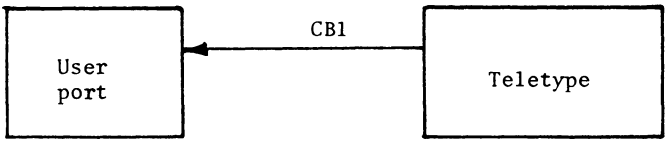
(d) Bit 7 output, rest inputs:
```
LDA  #&80
STA  &FE62
```

You can read the contents of port B even if one or more of its bits are designated as outputs; that is, LDA &FE60 will always give a valid reflection of the contents of &FE60. This is not true of port A, however, where the bits can be validly read only if they are designated inputs — fortunately, port B and not port A is the user port (port B is also a more powerful driver, and with suitable circuits, can drive solenoids etc.).

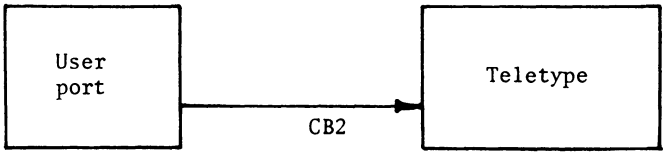
A7.3 Handshaking

Suppose we wish to send data to a teletype. The teletype has a parallel buffer which can store 8 bits, we shall assume. So we configure the user ports to be all output (that is, LDA #&FF: STA &FE62). We deposit the byte we wish to send into port B (STA &FE60, assuming that the byte is in the accumulator), wait until the teletype has processed the byte, and then send the next one. But how do we know when the teletype is ready? And how does it know when we are ready to send the next byte? The answer lies in the concept of *handshaking*.

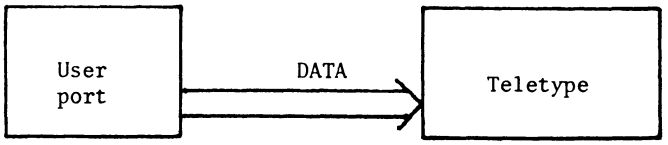
Port B has two control lines, CB1 and CB2. CB1 is *always* an *input*, and so will be used to transmit the signal from the teletype: CB2 can be an input or an output, and in this case we will use it as an ouput (how we specify this will become clear in a moment).



(a) The teletype is ready to receive data.



(b) The microprocessor deposits a byte in Port B, and signals to the teletype (the handshake).



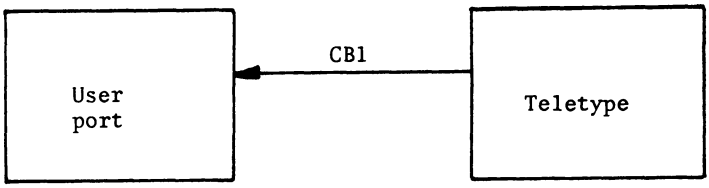
(c) The data is transmitted and processed (for example, a character is printed)

Figure A7.1: An output handshake

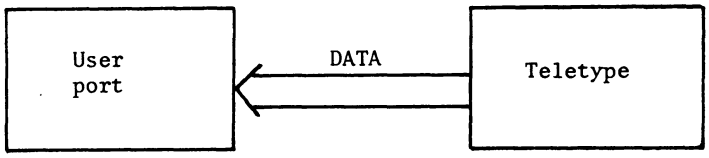
Refer to figure A7.1. The idea is that, when the teletype is ready to receive a byte, it sends a signal on CB1 to the user port (A7.1a). The microprocessor now deposits the byte in port B and sends a signal on CB2 to the teletype indicating that output is now valid (A7.1b). The teletype reads and processes the byte (A7.1c), sends a signal on CB1 asking for the next byte (A7.1a), and so the process continues.

This method of establishing connections between the user port and the teletype (or any other peripheral) is called handshaking: the teletype extends its 'hand' (signal on CB1), and the user port extends its 'hand' in recognition (signal on CB2). Now the information can pass between them. Sometimes CB1 and CB2 are referred to as *strobes* in this context; a strobe is simply an input and/or output line that indicates the availability of data to be transferred or the occurrence of a successful transfer. A strobe is usually a short pulse, one or two cycles long.

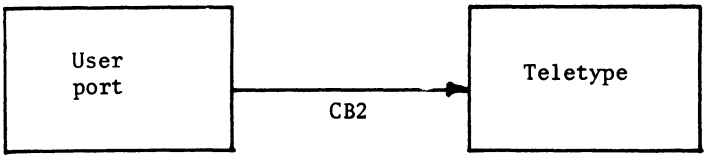
Figure A7.2 shows the same idea when port B is used for input (perhaps again from a teletype). When data is ready to be sent, the teletype sends a signal on CB1 indicating that data is ready (A7.2a). The user port responds by reading in the data (A7.2b), and sending a signal on CB2 indicating that the data has been successfully read (A7.2c) — this is the handshake. Again the process continues, with the teletype signalling the next byte is ready for transmission (A7.2a).



(a) The teletype signals that it is ready to send data



(b) The data is transmitted and perhaps processed



(c) The microprocessor signals that data has been successfully received (the handshake)

Figure A7.2: An input handshake

Notice that for output the handshake takes place *before* the data is sent, but that for input the handshake takes place *after* the data has been read. This is because the handshake is always finalised by the microprocessor.

Now the user port needs to provide the following facilities for these handshaking activities (we focus here only on port B — port A has almost identical features, but it is reserved for the printer in the BBC Computer):

(a) To designate CB2 as an input or output line.

(b) To fix the *levels* of input of CB1 and CB2 if relevant (that is, whether a signal is to be interpreted as high to low voltage — a falling edge, *negative transition* — or vice versa — a rising edge, *positive transition*).

These and other functions are the purpose of the top nybble of the peripheral control register (&FE6C): Table A7.4 shows the 8 possible configurations of bits 5–7, which control CB2, and the two configurations of bit 4 which control CB1.

You will notice in this table reference to the interrupt flag register. Bit 3 of this register will be set to 1 if there is an active transition (as defined by the peripheral control register) on CB2 and bit 4 if there is an active transition on CB1. Table A7.3 shows the entire register, and we will consider other flags later.

The difference in table A7.4 between the handshaking and independent input modes of CB2 lies in this: in the handshaking mode, reading or writing to port B will clear bit 3 of the interrupt flag register automatically (to make way for the next handshaking operation), whereas in the independent mode one can read or write to port B without the interrupt flag being cleared (this is useful if CB2 is being used for a purpose unconnected with what occurs on port B).

However, in our applications here we are interested in the output modes of CB2. The handshake output mode sets CB2 low when data is written into port B by the microprocessor, and sets it high again on an active transition of CB1. The pulse output mode sets CB2 low for one clock cycle following a write to port B (a brief strobe). The last two constant modes are useful if we wish to provide output signals directly under software control, independently of what occurs at port B.

Now let us consider our output application again. We begin with LDA #&FF: STA &FE62, to create an output port at B. We will assume that active transition of CB1 is negative. Thus we write

```
LDA  #&80
ORA  &FE6C
STA  &FE6C
```

to configure CB1 and CB2 as required (ORA then STA, so as to preserve the information for port A). The handshaking sequence is now

```
1       BEGIN   LDA     #&10
2       WAIT    BIT     &FE6D
3               BEQ     WAIT
4               LDA     OUTPUT
5               STA     &FE60
```

Repeating this, with suitable changes in line 4, is all that is needed to output as much data as required. Lines 1–3 wait until CB1 goes active, signalled by bit 4 of the interrupt flag register being set. At this stage, CB2 is automatically high (a feature of handshake output mode). Line 5 sets CB2 low and also clears the CB1 interrupt flag, all automatically. We can now return to line 1 to wait for the next CB1 signal. The signal on CB2, negative transition, has automatically occurred at line 5, and no doubt the teletype will respond in due course.

Consider now the input function. We set port B to input, and configure CB1 and CB2 as before. Lines 1–3 of the handshake are as before. We then write

```
4       LDA     &FE60
5       STA     &FE60
6       STA     OUTPUT
```

We have to write the data back to port B in line 5 in order to activate the CB2 line: in handshake output mode CB2 is only activated on a write (port A does not have this limitation, however, CA2 being activated by a read or write).

One final point in this section. To guard against changes in input before the input port is read — this is especially important if the microprocessor is doing much more than just the wait sequence in lines 1–3 — it is important to hold the input stable. To achieve this, the VIA is provided with *latches* which can protect input from corruption by changes on the input lines. To set the latch on port B we write a one into bit one of the auxiliary control register (at &FE6B) before entering the handshake (this need be done only once). Hence we write LDA #&02: STA &FE6B. Other functions of the auxiliary control register will be considered shortly.

A7.4 Interrupts on CB1 and CB2

If we consider the input application above, it is clear that it is rather wasteful of processor time to wait until the teletype sends its next byte.

Few teletypes work faster than about 30 characters per second, so the microprocessor could be doing other things most of the time. One strategy is to inspect the CB1 flag every 1/50 second — we will see how this is done in the next section. Another is to obtain an interrupt on CB1.

This is easily done: the interrupt enable register at &FE6E is the relevant register — see Table A7.2. We need to set bit 4 to 1 to enable interrupts on CB1. Once this bit is set to 1, an interrupt will be generated as soon as bit 4 of the interrupt flag register is set. In this case bit 7 of the flag register will also be set — this is used by the interrupt servicing routine when it is polling the potential causes of interrupt (see section 9.3). In the case of the BBC machine, if this bit is set, a test is made to see if bit 1 of the enable and flag registers are also set. If they are, the printer has caused the interrupt — if not, the user port service routine will be entered by a JMP (&0206).

Since bit 4 of the interrupt flag register may already be one, it is essential to clear it before enabling the interrupt. This can be done either by writing 1 to bit 4 of the flag register (that is, LDA #&10: STA &FE6D), or by reading port B (that is, LDA &FE60). Clearly the latter is slightly quicker, but not if taken together with the enable for we can neatly write LDA #&90: STA &FE6D: STA &FE6E, as we will see below (writing one into bit 7 of the flag register does nothing).

If your interrupt routine is going at location &0D01 onwards, you will write ?&0206 = 1: ?&0207 = &0D. Now all interrupts on CB1 (and on CB2, T1, T2 and SR) will go to a routine at &0D01. The routine will consist of lines 4–6, with 6 suitably amended and expanded if necessary.

Every time an interrupt is routed to &206 and &207 the first thing that *must* be done in one's servicing routine is to save X, Y and the original contents of the accumulator (stored at &FC). Thus, your routine must *always* begin with

```
TXA
PHA
TYA
PHA
LDA   &FC
PHA
```

and end with

```
PLA
STA   &FC
PLA
TAY
PLA
TAX
JMP   &DE89          (the original contents IRQ2)
```

We take the precaution of saving &FC in case a subroutine is called within the interrupt routine which alters &FC. Note, though, that we do not need to save P: this is done automatically by the 6502 microprocessor itself on interrupt.

The interrupt enable register can be altered only by writing ones into the relevant bit positions: *writing zeros has no effect at all.* To enable CB1 interrupts we must write a one into bit 4 with bit 7 *equal to one*: to disable CB1 interrupts we write a one into bit 4 with bit 7 *equal to zero*. Thus, to enable CB1: LDA #&90: STA &FE6E; and to disable: LDA #&10: STA &FE6E. These operations will only affect the CB1 enable — all other bits will be unaffected.

A7.5 Using the programmed timers

(Note that small inaccuracies in the 6522 timer operation are ignored in this section).

There are two timers in the 6522, and both are at the programmer's disposal. Timer2 is the easier, and we shall consider this first.

Timer 2 has two uses: it can generate a single time interval or it can count pulses input to bit 6 of port B. Bit 5 of the auxiliary control register determines which (see Table A7.5). The timer's counter consists of two bytes: the low byte at &FE68 and the high at &FE69. Always load the low byte first: loading the high byte clears the interrupt flag and starts the timing operation.

Suppose we wish to create an interval of 10,000 (&2710) clock cycles and then generate an interrupt. Here is the coding

```
0   LDA  #&DF  }
1   AND  &FE6B }   Set bit 5 to zero
2   STA  &FE6B }
3   LDA  #&A0  }
4   STA  &FE6D }   Clear T2 interrupt flag
5   STA  &FE6E     Enable T2 interrupts
6   LDA  #&10  }
7   STA  &FE68 }   Load low byte with &10
8   LDA  #&27  }
9   STA  &FE69 }   High byte with &27 and start
                   the countdown
```

If desired, this configuration can be done from BASIC using the query (?) operator.

In the service routine at &D01, just before JMP &DE89 is encountered, the statement LDA &FE68 must appear: this clears the T2 interrupt flag.

Note that a clock cycle here is one-half the 2 MHz machine cycle; for example, 10,000 cycles is 1/100th of a second. This is because the clocking

is done by the phase 2 clock which times memory operations, and this runs at 1 MHz.

The pulse counting mode is used to access an external clock or to synchronise with a set of external events.Changing lines 0 and 1 above to LDA #&20: ORA &FE6B will count 10,000 pulses incoming on bit 6 of port B.

Timer1 has more interesting applications. Instead of generating just one time interval it can generate a whole series of intervals. Bit 6 of the auxiliary control register determines this (Table A7.5). When the high byte is loaded countdown will begin: if bit 6 of the auxiliary control register is 1, at the end of countdown the counter will be reloaded with the original contents of the counters which are stored in latches (at &FE66 and &FE67), and countdown will begin again.

Consider the teletype input again. We can generate an interrupt every 1/50th of a second by loading timer1 with 20,000 (&4E20) in continuous mode

```
0   LDA  #&40  }
1   ORA  &FE6B }  Set bit 6 to one
2   STA  &FE6B }
3   LDA  #&C0  }  Clear T1 interrupt flag
4   STA  &FE6D }
5   STA  &FE6E    Enable T1 interrupts
6   LDA  #&20  }  Load counter and latch with low byte
7   STA  &FE64 }
8   LDA  #&4E  }  Load counter and latch with high
9   STA  &FE65 }  byte and start count
```

Again in the service routine at &D01 include LDA &FE64 to clear the T1 interrupt flag. And again, BASIC can be used for the configuration if desired.

It is possible to alter the contents of the latches while the countdown is proceeding without affecting it in its present run. On the next run, however, new contents will be loaded. This is particularly useful with the other feature of timer1, the generation of pulses out of bit 7 of port B (clearly this bit must be configured as output). In the continuous mode (bit 6 = 1), the level on bit 7 of port B will begin low, then at time-out will go high, then at time-out again low, etc. Hence it is possible to create complicated waveforms, if the contents of the latches are also changed.

If we want to generate considerably longer delays, we can build this into our interrupt service routine. Let lines 0–9 be as before, but suppose we require an operation to occur every 1 second. To do this, we will require to reserve one location for the service routine, say &8F. The idea is that we load it with 50 and decrement it every interrupt. When it reaches zero we

perform the required operation. However, as we saw in section 9.3, events are much easier to use for this purpose, and are to be preferred.

A7.6 The Shift Register

The 6522 has a shift register (at &FE6A) which will input or output bits one at a time under timed control. The timing can either be provided by an external clock, the internal machine clock or else timer2. Bits 2, 3 and 4 of the auxiliary control register determine which (Table A7.5).

If the teletype in our previous examples lacked a parallel buffer, it would be possible to input and output bits synchronously using timer2 and the shift register. To achieve handshaking, it will be necessary to choose the mode that disables the timer each time, and to provide a suitable set of interrupts. The software turns out to be quite tricky,and it is better to use a UART (such as the 6551 ACIA which does not need an external clock): this will sort out all the parity and framing errors, and provide the stop and start bits. This can generate its own interrupts, but the circuitry might be easier if connections are made to the user port, utilising the VIA's interrupts instead (or by using the expansion bus). Of course, for a 300 baud teletype the computer's own RS423 port can be used, and handshaking here is very simple to implement.

There are really only two main uses for the serial registers as far as we are concerned. One is to provide a source of memory clock pulses, or to receive them from another computer, so as to achieve synchronisation between the computers. Use 010 or 110 at bits 2–4 of the auxiliary control register to achieve this. The other is to output a variety of square waves independently of microprocessor control. This allows frequencies from about 2 Hz (loading the shift register with &0F and timer2 with &FFFF) to 500 KHz (loading the shift register with &55 and timer2 with 1). 0100 at bits 2–5 of the auxiliary control register is the configuration in this case.

Appendix 8: Differences between BASIC 1 and BASIC 2

The new BASIC (BASIC 2) has two major additional features as far as assembly programming is concerned. (Type REPORT to find out which version of BASIC you have. ©1982 implies BASIC 2.) The first is the use of the EQU functions, discussed in chapter 6 and later chapters. The other is the extended use of OPT, described in appendix 6.

This book assumes that BASIC 2 is installed, but includes alternative methods for BASIC 1 where appropriate. In chapter 6 all the addresses given are for BASIC 2. The table below gives the BASIC 1 equivalents:

BASIC 2	*BASIC 1*
BC05	BC20
BC0D	BC28
BC0F	BC2A

Answers to Exercises

Exercise 1.1

2. (i) The contents need to change.
(ii) The address must be permanently available, even when power is off.

3. There is a limit to the number of pins economically available in a package. Once 40 pins became established, it became very difficult to produce other size packages. Another problem is that, until recently, a 16-bit bus would be too slow—however, 16-bit processors are now a reality.

4. Data can go *into* memory and *out of* memory. A memory location is chosen, however, by sending an address *towards* it. No *address* information needs to come *from* it.

5. Because each instruction needs to be translated every time that it is executed in an interpreted language. With compilation the translation occurs just once, prior to execution.

Exercise 2.1

1. (a) Immediate

&0C00	&0C01
A9	0E

(b) Absolute

&0C00	&0C01	&0C02
AD	40	7F

(c) Zero page

&0C00	&0C01
A5	20

(d) Absolute

&0C00	&0C01	&0C02
8D	72	7A

(e) Zero page

&0C00	&0C01
85	00

(f) Immediate

&0C00	&0C01
A9	12

(g) Zero page

&0C00	&0C01
85	02

(h) Zero page

&0C00	&0C01
A5	0E

(i) Absolute

&0C00	&0C01	&0C02
8D	00	04

2. The contents of NUM1 are already in the accumulator.

```
P% =  &0C00
NUM1  =  &70
NUM2  =  &71
LDA   #17
STA   NUM1
STA   NUM2
```

&0C00	&0C01	&0C02	&0C03	&0C04	&0C05
A9	11	85	70	85	71

3.

```
P% =  &0C00
NUM1  =  &70
NUM2  =  &71
NUM3  =  &73
LDA   NUM3
STA   NUM1
LDA   NUM2
STA   NUM3
LDA   NUM1
STA   NUM2
```

Exercise 3.1

```
 10 NUM1=&70:NUM2=&71:NUM3=&72:SUML=&73:SUMH=&74
 20 DIM P% 50
 30 [OPT3
 40 .START
 50 LDA NUM1
 60 CLC
 70 ADC NUM2
 80 STA SUML
 90 LDA #0
100 STA SUMH
110 ADC SUMH
120 STA SUMH
130 LDA SUML
140 CLC
150 ADC NUM3
160 STA SUML
170 LDA #0
180 ADC SUMH
```

```
190 STA SUMH
200 RTS:]
210 REPEAT
220   INPUT"First number to be added",?NUM1
230   INPUT"Second number to be added",?NUM2
240   INPUT"Third number to be added",?NUM3
250   CALLSTART
260   PRINT?NUM1+?NUM2+?NUM3,256*?SUMH+?SUML
270   UNTIL FALSE
```

Exercise 3.2

1.

```
 10 NUM1L=&70:NUM1H=&71:NUM2L=&72:NUM2H=&73:SUM0
    =&74:SUM1=&75:SUM2=&76
 20 DIM P% 50
 30 [OPT3
 40 .START
 50 LDA NUM1L
 60 CLC
 70 ADC NUM2L
 80 STA SUM0
 90 LDA NUM1H
100 ADC NUM2H
110 STA SUM1
120 LDA 30
130 STA SUM2
140 ADC SUM2
150 STA SUM2
160 RTS:]
170 REPEAT
180   INPUT"First number to be added",!NUM1L
190   INPUT"Second number to be added",!NUM2L
200   CALLSTART
210   PRINT256*?NUM1H+?NUM1L+256*?NUM2H+?NUM2L,
      65536*?SUM2+256*?SUM1+?SUM0
220   UNTIL FALSE
```

2.

```
 10 DIM NUM1(3),NUM2(3),RESULT(3)
 20 FOR I%=0 TO 3:NUM1(I%)=&70+I%
 30   NUM2(I%)=&74+I%:RESULT(I%)=&78+I%:NEXTI%
 40 DIM P% 50
 50 [OPT3
 60 .START
 70 LDA NUM1(0)
 80 CLC
 90 ADC NUM2(0)
100 STA RESULT(0)
110 LDA NUM1(1)
120 ADC NUM2(1)
130 STA RESULT(1)
```

```
140 LDA NUM1(2)
150 ADC NUM2(2)
160 STA RESULT(2)
170 LDA NUM1(3)
180 ADC NUM2(3)
190 STA RESULT(3)
200 RTS:]
210 REPEAT
220   INPUT"First number to be added",!NUM1(0)
230   INPUT"Second number to be added",!NUM2(0)
240   CALLSTART
250   PRINT!NUM1(0)+!NUM2(0),!RESULT(0)
260   UNTIL FALSE
```

Symbolic representation is

(NUM1(3); NUM1(2); NUM1(1); NUM1(0)) + (NUM2(3); NUM2(2); NUM2(1); NUM2(0)) → RESULT(3); RESULT(2); RESULT(1); RESULT(0)

Exercise 3.3

1.

```
LDA  NUM1L
SEC
SBC  NUM2L
STA  DIFFL
LDA  NUM1H
SBC  NUM2H
STA  DIFFH
```

A mathematical demonstration of why this works may be helpful to you. In general, for *any* numbers A and B, $A - B = A + B_c - B_c - B$, where B_c is the two's complement of B, $= (A + B_c) - (B_c + B) = A + B_c$, if we ignore the 'carry'.

Now, if A and B are double bytes, so that A = (AH;AL) and B = (BH;BL) then $B_c = (\bar{B}\bar{H};BL_c)$, where $\bar{B}\bar{H}$ is the one's complement of BH. Hence, $A - B = (AH + \bar{B}\bar{H} + C;\ AL + BL_c)$, where C is the carry. Now, $AL + BL_c$ is achieved by the first three lines of the program. If this is positive C will be one, if negative C will be zero. Either way, the second part of the program perform $AH + \overline{BH} + C$ as required.

2.

```
10 DIM NUM1(3),NUM2(3),RESULT(3)
20 FOR I%=0 TO 3:NUM1(I%)=&70+I%
30   NUM2(I%)=&74+I%:RESULT(I%)=&78+I%:NEXTI%
40 DIM P% 50
50 [OPT3
```

```
 60 .START
 70 LDA NUM1(0)
 80 SEC
 90 SBC NUM2(0)
100 STA RESULT(0)
110 LDA NUM1(1)
120 SBC NUM2(1)
130 STA RESULT(1)
140 LDA NUM1(2)
150 SBC NUM2(2)
160 STA RESULT(2)
170 LDA NUM1(3)
180 SBC NUM2(3)
190 STA RESULT(3)
200 RTS:]
210 REPEAT
220   INPUT"First number",!NUM1(0)
230   INPUT"Number to be subtracted",!NUM2(0)
240   CALLSTART
250   PRINT!NUM1(0)-!NUM2(0),!RESULT(0)
260   UNTIL FALSE
```

Exercise 3.4

1. (a) &18 − &EE = &18 + &12 = &2A (= 42)
(b) &AA − &23 = &AA + &DD = 1 &87 (= −121)

Overflow can occur only if the signs of the numbers subtracted are different.

2. -2^{31} to $2^{31} - 1$

3. No adjustments necessary.

IF SUM(3) > 127 THEN RESULT = (((SUM(3) − 255) * 256 + (SUM(2) − 255)) * 256 + (SUM(1) − 255)) * 256 + SUM(0) − 256
ELSE RESULT = ((SUM(3) * 256 + SUM(2)) * 256 + SUM(1)) * 256 + SUM(0)

Exercise 3.5

(a)
```
ORA  #&88
AND  #&EE
```

(b)
```
EOR  #&80
ORA  #&40
AND  #&E0
```

Exercise 4.1

1.

```
                LDA   NUM1
                CLC
                ADC   NUM2
                BEQ   ZERO
                BPL   POSITIVE
ZERO            .
                .
                .
                .
                .
POSITIVE        .
                .
                .
                .
                .
```

2.

```
                LDA   NUM1
                SEC
                SBC   NUM2
                BEQ   ZERO
                LDA   NUM2
                STA   NUM3
                LDA   NUM1
                STA   NUM2
                LDA   NUM3
                STA   NUM1
ZERO            .
                .
                .
                .
                .
```

3.

```
                LDA   NUM1
                CLC
                ADC   NUM2
                BCC   NOCARRY
                LDA   #0
                STA   SUM
                BEQ   OVER        (always branches)
NOCARRY         STA   SUM
OVER            .
                .
                .
```

Congratulate yourself if you got this. Congratulate yourself even more if you got the following more economical version

```
            LDA   NUM1
            CLC
            ADC   NUM2
            STA   SUM
            BCC   NOCARRY
            LDA   #0
            STA   SUM
NOCARRY     .
            .
            .
            .
            .
```

Exercise 4.2

1.

```
            LDA   NUM
            CMP   #15
            BEQ   LESSEQ
            BCC   LESSEQ
            LDA   #0
            STA   NUM
LESSEQ      .
            .
            .
            .
            .
```

2.

```
            LDA   NUM
            BPL   POSITIVE
            CMP   #&F6
            BEQ   EQUAL
            BCS   POSITIVE     (or BPL POSITIVE)
EQUAL       STA   INDIC
            JMP   OVER
POSITIVE    LDA   #0
            STA   NUM
            LDA   #1
            STA   INDIC
OVER        .
            .
            .
            .
            .
```

3.

```
                LDA   NUM1
                CMP   NUM2
                BNE   NTEQUAL1
                LDA   NUM3
                STA   NUM2
                JMP   OVER
NTEQUAL1        LDA   NUM4
                CMP   #16
                BEQ   NTEQUAL2
                BCC   NTEQUAL2
                STA   NUM2
                JMP   OVER
NTEQUAL2        LDA   #0
                STA   NUM2
                STA   NUM4
OVER            .
                .
                .
                .
                .
```

Exercise 4.3

1.

```
                LDA   NUM1L
                CMP   NUM2L
                LDA   NUM1H
                SBC   NUM2H
                BCC   LESS
                .
                .
                .
                .
LESS            .
```

2.

```
                LDA   NUM1L
                CMP   NUM2L
                BNE   NTEQUAL
                CLC
NTEQUAL         LDA   NUM1H
                SBC   NUM2H
                BCC   LESSEQ
                .
                .
                .
                .
LESSEQ          .
```

3.

```
        LDA   NUM1(0)
        CMP   NUM2(0)
        LDA   NUM1(1)
        SBC   NUM2(1)
        LDA   NUM1(2)
        SBC   NUM2(2)
        LDA   NUM1(3)
        SBC   NUM2(3)
        BCS   GTEQUAL
              .
              .
              .
              .
GTEQUAL       .
```

4.

```
        LDA   NUM(0)
        ORA   NUM(1)
        ORA   NUM(2)
        ORA   NUM(3)
        BEQ   ZERO
              .
              .
              .
              .
ZERO          .
```

The accumulator will at the end contain ones in the positions where ones occur in any of the four bytes tested. Hence, only if all four bytes are zero will the accumulator be zero.

Exercise 4.4

(a)

```
        LDA   NUM1L
        SEC
        SBC   NUM2
        STA   NUM1L
        LDA   NUM1H
        SBC   #0
        STA   NUM1H
```

(b)

```
        LDA   NUM1L
        SEC
        SBC   NUM2
        STA   NUM1L
        BCS   NOBORROW
        DEC   NUM1H
```

Three bytes are saved.

```
NOBORROW        .
                .
                .
                .
                .
```

Exercise 4.5

1.

```
                LDA   NUM1
                SEC
                SBC   NUM2
                BCC   LESSEQ
                CMP   NUM3
                BEQ   LESSEQ
                BCS   GREATER
LESSEQ          .
                .
                .
                .
                .
GREATER         .
```

2.

```
                LDA   #0
                STA   DIFF2
                LDA   NUM1L
                SEC
                SBC   NUM2L
                STA   DIFF0
                LDA   NUM1H
                SBC   NUM2H
                STA   DIFF1
                BCS   NTNEG
                DEC   DIFF2
                .
NTNEG           .
                .
```

3.

```
                LDA   #0
                STA   SUMH
                LDA   NUM1
                CLC
                ADC   NUM2
                STA   SUML
```

```
                BVC   NOOVFLOW
                EOR   #&80
NOOVFLOW        BPL   NTNEG
                DEC   SUMH
NTNEG                 .
                      .
                      .
                      .
                      .
```

4.

```
                LDA   #0
                STA   DIFF2
                LDA   NUM1L
                SEC
                SBC   NUM2L
                STA   DIFF0
                LDA   NUM1H
                SBC   NUM2H
                STA   DIFF1
                BVC   NOOVFLOW
                EOR   #&80
NOOVFLOW        BPL   NTNEG
                DEC   DIFF2
                      .
NTNEG                 .
                      .
```

5.

```
                LDA   NUM1L
                SEC
                SBC   NUM2L
                STA   NUM1L
                LDA   NUM1M
                SBC   NUM2H
                STA   NUM1M
                BCS   NOBORROW
                DEC   NUM1H
                BPL   OVERFLOW
NOBORROW              .
                      .
                      .
                      .
                      .
OVERFLOW              .
```

Overflow occurs here if NUM1H goes from &80 to &79, and the N flag will register this. The V flag is not affected by DEC or by INC, since the N flag tells us exactly what we want to know. The OVERFLOW routine here would probably be designed to return some error message.

6.

```
                LDA   #0
                STA   SUMH
                LDA   NUM1L
                CLC
                ADC   NUM2L
                STA   SUML
                LDA   NUM1H
                ADC   NUM2H
                STA   SUMM
                BVC   NOOVFLOW
                EOR   #&80
NOOVFLOW        BPL   NTNEG
                DEC   SUMH
NTNEG           .
                .
                .
                .
                .
```

7.

```
                LDA   NUM1
                SEC
                SBC   NUM2
                BVC   NOOVFLOW
                EOR   #&80
NOOVFLOW        BPL   GTEQUAL
                .
                .
                .
                .
GTEQUAL         .
                .
                .
```

8.

```
                LDA   NUM1
                SEC
                SBC   NUM2
                BVC   NOOVFLOW
                BPL   LESSEQ      (Since overflow occurred, this
                                  branches if the result is negative,
                                  that is, NUM1–NUM2<0
                BMI   OVER        —and this branches if result
                                  positive
NOOVFLOW        BMI   LESSEQ
OVER            CMP   NUM3        Usual comparison between unsigned
                                  numbers.
                BEQ   LESSEQ
```

```
                BCS   GREATER
LESSEQ                .
                      .
                      .
                      .
                      .
GREATER               .
```

9.

```
                LDA   NUM1
                SEC
                SBC   NUM2
                BVC   NOOVFLOW
                BMI   MORE        Positive overflow must exceed
                                  NUM3.
                BPL   LESSEQ      Negative overflow must be less
                                  than NUM3.
NOOVFLOW1       SEC
                SBC   NUM3
                BEQ   LESSEQ
                BVC   NOOVFLOW2
                EOR   #&80
NOOVFLOW2       BPL   MORE
LESSEQ                .
                      .
                      .
                      .
                      .
MORE                  .
                      .
                      .
```

10.

```
                LDA   NUM1L
                CMP   NUM2L
                LDA   NUM1H
                SBC   NUM2H
                BVC   NOOVFLOW
                EOR   #&80
NOOVFLOW        BPL   GTEQUAL
                      .
                      .
                      .
                      .
GTEQUAL               .
```

Exercise 5.1

1. Yes they will. For example, FOR X = −2 TO 50 in (e) will result in just one cycle of the loop since it will be understood as FOR X = 254 TO 50.

We solve such problems by using BPL and BMI instead of BCS and BCC after comparisons (so that, if NUM is signed, it is the second program in (d) which is correct). It is not quite this simple in (f), however, since we can get overflow: FOR X = 126 TO 10 STEP 20 is an example; FOR X = 50 TO 100 STEP 40 is another.

Hence we need a test for overflow also, and the solution to question 3 deals with this.

2. (a)

```
        LDX   NUM2
LOOP    .
        .
        .
        .
        .
        DEX
        CPX   #&FF
        BEQ   OUT
        CPX   NUM1
        BCS   LOOP
OUT     .
        .
        .
        .
        .
```

(b)

```
        LDX   NUM2
LOOP    .
        .
        .
        .
        .
        TXA
        SEC
        SBC   NUM3
        BCC   OUT
        TAX
        CMP   NUM1
        BCS   LOOP
OUT     .
        .
        .
        .
        .
```

3.

```
        LDX   NUM1
LOOP    .
        .
        .
        .
        .
        TXA
        CLC
        ADC   NUM3
        TAX
        BVS   OUT     If overflow, loop must be finished.
        CMP   NUM2
        BMI   LOOP
        BEQ   LOOP
OUT     .
        .
        .
        .
        .
```

It is quicker still to work 'backwards' (although—(NUM3) may turn out to be positive):

```
        LDX   NUM2
LOOP    .
        .
        .
        .
        .
        TAX
        SEC
        SBC   NUM3
        TAX
        BVS   OUT
        CMP   NUM1
        BPL   LOOP
OUT     .
        .
        .
        .
        .
```

Exercise 5.2

1. After LDX #0 put LDA NUMH
   BEQ LOOP2

2. (i) In this case, all that is required is that we create a loop with (NUMH; NUML) + 1 cycles. The most efficient way to do this is

```
            LDX   NUML
LOOP1             .
                  .
                  .
                  .
                  .
            DEX
            CPX   #&FF
            BNE   LOOP1
            LDY   NUMH
            BEQ   OUT
            INX
LOOP2             .
                  .
                  .
                  .
                  .
            DEX
            BNE   LOOP2
            DEY
            BNE   LOOP2
OUT               .
                  .
                  .
                  .
                  .
```

(ii) N cannot be computed in any simple way from (i), since N does not decrease in strict descending order (in LOOP2 &00 precedes &FF). Also the high byte of N is given by Y − 1 in LOOP2.

We require a loop in strict order where N is (Y;X) in LOOP2 and (NUMH;X) in LOOP1

```
        LDX   NUML
LOOP1           .
                .
                .
                .
                .
        DEX
        CPX   #&FF
        BNE   LOOP1
        LDY   NUMH
        BEQ   OUT
        DEY
LOOP2           .
                .
                .
                .
                .
        DEX
        CPX   #&FF
        BNE   LOOP2
        DEY
        CPY   #&FF
        BNE   LOOP2
OUT             .
                .
                .
                .
                .
```

Only (i) is an improvement over the forward loop.

Exercise 5.3

1. (a)

```
TXA
SEC
SBC   M
STA   M
```

(b)

```
        STX   MEMLOC+1
        SEC
MEMLOC  SBC   #0
```

(c)

```
TXA
SEC
SBC   M
TAX
```

(d)

```
        STX   MEMLOC+1
        SEC
MEMLOC  SBC   #0
        TAX
```

```
(e)             TXA
                STY   MEMLOC+1
                CLC
  MEMLOC        ADC   #0
```

```
(f)             TXA
                STY   MEMLOC+1
                SEC
  MEMLOC        SBC   #0
```

```
(g)             STA   MEMLOC+1
  MEMLOC        CPX   #0
```

```
(h)             STY   MEMLOC+1
  MEMLOC        CPX   #0
```

```
(i)             STX   MEMLOC+1
                TAX
  MEMLOC        LDA   #0
```

2.

```
                STA   MEMLOC+1
                TXA
                CLC
                ADC   M
                STA   M
  MEMLOC        LDA   #0
```

3.

```
                STA   MEMLOC+1
                TXA
                SEC
                SBC   M          MEMLOC1   SBC   #0
                TAX
  MEMLOC        LDA   #0         MEMLOC2   LDA   #0
```

```
                STX   MEMLOC
                STA   MEMLOC
                SEC
                TAX
```

4.

```
                STY   MEMLOC1 + 1
                STA   MEMLOC2 + 1
                TXA
                CLC
MEMLOC1         ADC   #0
                TAX
MEMLOC2         LDA   #0
```

Note that the use of the stack makes questions 2, 3 and 4 easier to solve. This is considered further in chapter 9.

Exercise 5.4

1.

```
10 TERML=&70:TERMH=&71:NUM=&72:SUM1=&73:SUM2=&74:
   SUM3=&75:?&76=0
20 DIM START 100
30 FOR I%=0 TO 2 STEP 2:P%=START
40   [OPTI%
50   LDA #0
60   TAX
70   STA SUM2
80   STA SUM3
90   STA TERMH
100  LDA #1
110  STA SUM1
120  STA TERML
130  .LOOP
140  INX
150  CPX NUM
160  BEQ FINISH
170  STX MEMLOC+1
180  CLC
190  LDA TERML
200  .MEMLOC
210  ADC #0 Dummy operand
220  STA TERML
230  BCC NOCARRY
240  CLC
250  INC TERMH
260  .NOCARRY
270  LDA TERML
280  ADC SUM1
290  STA SUM1
300  LDA TERMH
310  ADC SUM2
320  STA SUM2
330  BCC LOOP
340  INC SUM3
```

```
350    BCS LOOP
360    .FINISH
370    RTS:]NEXTI%
380 CLS:REPEAT
390    INPUT"How many terms",?NUM
400    CALLSTART
410    PRINT!SUM1
420    UNTIL FALSE
```

Lines 140–160 stop the loop being entered if (NUM) = 1.
Lines 170–250 compute the (X + 1)th term (Xth term + X).
Lines 270–350 add the (X + 1)th term to the sum for X terms to get the sum for (X + 1) terms.

2.

```
 10 TERML=&70:TERMH=&71:SUM1=&72:SUM2=&73:SUM3=&74:
    STOTAL1=&75:STOTAL2=&76:STOTAL3=&77
 20 DIM START 100
 30 FOR I%=0 TO 2 STEP 2:P%=START
 40   [OPTI%
 50   LDA #0
 60   TAX
 70   STA SUM2
 80   STA SUM3
 90   STA TERMH
100   LDA #1
110   STA SUM1
120   STA TERML
130   .LOOP
140   INX
150   LDA STOTAL1
160   CMP SUM1
170   LDA STOTAL2
180   SBC SUM2
190   LDA STOTAL3
200   SBC SUM3
210   BCC FINISH
220   STX MEMLOC+1
230   CLC
240   LDA TERML
250   .MEMLOC
260   ADC #0 Dummy operand
270   STA TERML
280   BCC NOCARRY
290   CLC
300   INC TERMH
310   .NOCARRY
320   LDA TERML
330   ADC SUM1
```

```
340    STA SUM1
350    LDA TERMH
360    ADC SUM2
370    STA SUM2
380    BCC LOOP
390    INC SUM3
400    BCS LOOP
410    .FINISH
420    RTS:]NEXTI%
430 CLS:REPEAT
440    INPUT"Maximum total",!STOTAL1
450    PRINT((USRSTART AND &0FFFFFFF)MOD &10000) DIV &100
460    UNTIL FALSE
```

As question 1 except that lines 150 and 160 are replaced by 150–210.

3.

```
 10 TERML=&70:TERMH=&71:SUM1=&72:SUM2=&73:SUM3=&74:
    STOTAL1=&75:STOTAL2=&76:STOTAL3=&77
 20 DIM START 100
 30 FOR I%=0 TO 2 STEP 2:P%=START
 40    [OPTI%
 50    LDA #0
 60    TAX
 70    STA SUM2
 80    STA SUM3
 90    STA TERMH
100    LDA #1
110    STA SUM1
120    STA TERML
130    .LOOP
140    INX
150    SEC
160    LDA STOTAL1
170    SBC SUM1
180    STA MEMLOC1+1
190    LDA STOTAL2
200    SBC SUM2
210    STA MEMLOC2+1
220    LDA STOTAL3
230    SBC SUM3
240    BCC FINISH1
250    .MEMLOC1
260    ORA #0 Dummy operand
270    .MEMLOC2
280    ORA #0 Dummy operand
290    BEQ FINISH2
300    SBC SUM2
310    LDA STOTAL3
320    STX MEMLOC+1
330    CLC
340    LDA TERML
```

```
350    .MEMLOC
360    ADC #0 Dummy operand
370    STA TERML
380    BCC NOCARRY
390    CLC
400    INC TERMH
410    .NOCARRY
420    LDA TERML
430    ADC SUM1
440    STA SUM1
450    LDA TERMH
460    ADC SUM2
470    STA SUM2
480    BCC LOOP
490    INC SUM3
500    BCS LOOP
510    .FINISH1
520    DEX
530    .FINISH2
540    RTS:]NEXTI%
550 CLS:REPEAT
560    INPUT"Maximum total",!STOTAL1
570    PRINT((USRSTART AND &OFFFFFFF)MOD &10000) DIV &100
580    UNTIL FALSE
```

As question 1 except that lines 150 and 160 are replaced by 150–310 and 360 by 510–530.

Exercise 6.1

When an early part of the new locations overlaps a later part of the old locations.

Exercise 6.2

1.

```
 10 INPUT"HOW MANY BYTES",NUMBER
 20 DIM ARRAY NUMBER-1:DIM START 50
 30 FORI%=0 TO NUMBER-2 STEP4:!(ARRAY+I%)=RND:NEXTI%
 40 FLAG=&70:TEMP=&71
 50 FORI%=0 TO 2 STEP 2:P%=START
 60    [OPTI%
 70    .BEGIN
 80    LDX #NUMBER-1
 90    LDA #0
100    STA FLAG
110    .LOOP
120    LDA ARRAY,X
130    CMP ARRAY-1,X
140    BCS OVER
150    STA TEMP
```

```
160    LDA ARRAY-1,X
170    STA ARRAY,X
180    LDA TEMP
190    STA ARRAY-1,X
200    LDA #1
210    STA FLAG
220    .OVER
230    DEX
240    BNE LOOP
250    LDA FLAG
260    BNE BEGIN
270    RTS:]NEXTI%
280 CALL START
290  FOR I%=0 TO NUMBER-1: PRINT?(ARRAY+I%),:NEXT
```

(a) By reversing the comparison we gain space and time since only one branch instruction is necessary.

(b) Usually, we put values into storage locations so that we do not need to reassemble the program every time that we change the values concerned. In this case, however, we will want to change the base address of the array (that is, ARRAY) and this will require reassembly. We will see a way round this in the next chapter.

2.

```
 10 INPUT"HOW MANY BYTES",NUMBER
 20 DIM ARRAY NUMBER-1:DIM START 50
 30 FORI%=0 TO NUMBER-2 STEP4:!(ARRAY+I%)=RND:NEXTI%
 40 TEMP=&70
 50 FORI%=0 TO 2 STEP 2:P%=START
 60   [OPTI%
 70   LDY #0
 80   .BEGIN
 90   STY MEMLOC+1
100   LDX #NUMBER-1
110   .LOOP
120   LDA ARRAY,X
130   CMP ARRAY-1,X
140   BCS OVER
150   STA TEMP
160   LDA ARRAY-1,X
170   STA ARRAY,X
180   LDA TEMP
190   STA ARRAY-1,X
200   .OVER
210   DEX
220   .MEMLOC
230   CPX #0 (Dummy operand)
240   BNE LOOP
250   INY
260   CPY #NUMBER-1
```

```
270   BNE BEGIN
280   RTS:]NEXTI%
290 CALL START
300  FOR I%=0 TO NUMBER-1: PRINT?(ARRAY+I%),:NEXT
```

Notice that this is less efficient since we require an extra CPY.

3.

```
 10 INPUT"HOW MANY BYTES",NUMBER
 20 DIM ARRAY NUMBER-1:DIM START 50
 30 FORI%=0 TO NUMBER-2 STEP4:!(ARRAY+I%)=RND:NEXTI%
 40 TEMP=&70
 50 FORI%=0 TO 2 STEP 2:P%=START
 60   [OPTI%
 70   LDY #1
 80   .BEGIN
 90   STY MEMLOC+1
100   .MEMLOC
110   LDX #0 (Dummy operand)
120   .LOOP
130   LDA ARRAY,X
140   CMP ARRAY-1,X
150   BCS OVER
160   STA TEMP
170   LDA ARRAY-1,X
180   STA ARRAY,X
190   LDA TEMP
200   STA ARRAY-1,X
210   .OVER
220   DEX
230   BNE LOOP
240   INY
250   CPY #NUMBER
260   BNE BEGIN
270   RTS:]NEXTI%
280 CALL START
290  FOR I%=0 TO NUMBER-1: PRINT?(ARRAY+I%),:NEXT
```

Exercise 6.3

1. The flowchart is in figure 6.7.

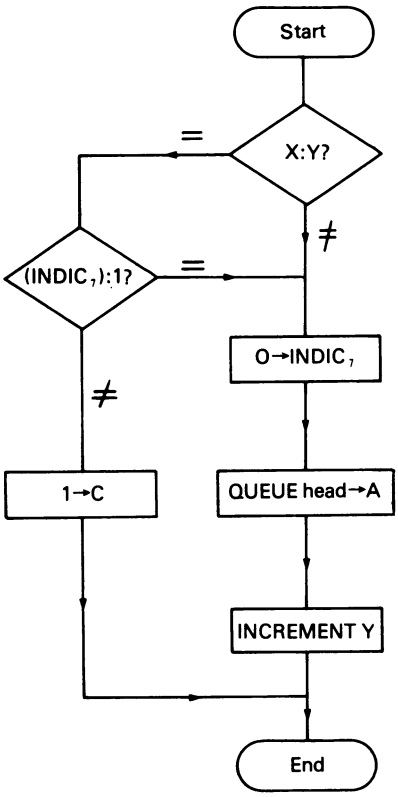


Figure 6.7: Flowchart for withdrawing an item from a queue

```
 10 DIM START 50:VDU14
 20  DATUM=&70:INDIC=&71:DIM BASMEM 255
 30 FOR I%=0 TO 252 STEP4:!(BASMEM+I%)=RND:NEXTI%
 40 DIM TEST 256
 50 X%=0:Y%=0:?INDIC=&80
 60 FOR I%=0 TO 2 STEP2:P%=START
 70   [OPTI%
 80   STX MEMLOC+1
 90   .MEMLOC
100   CPY #0 (Dummy operand)
110   BNE OK
120   LDA INDIC
130   BMI OK
140   SEC
150   RTS
160   .OK
170   LDA #0
180   STA INDIC
190   LDA BASMEM,Y
```

```
200    INY
210    CLC
220     RTS:]NEXTI%
230 FOR I%=0 TO 256
240     !&404=USR(START):X%=?&405:Y%=?&406:
        ?(TEST+I%)=?&404
250    IF (?&407 AND 1) =1 THEN PRINT"ERROR AT
       "STR$(I%+1)"TH WITHDRAWAL":GOTO270
260     NEXTI%
270 PRINT'"     WITHDRAWAL  QUEUE"'
280 FOR I%=0 TO 255
290    PRINT ?(BASMEM+I%),?(TEST+I%):NEXTI%
300 VDU15
```

2.

```
 10 DIM START1  50:DIM START2 50:VDU12
 20  DATUM=&70:INDIC=&71:DIM BASMEM 255
 30 X%=128:Y%=0:?INDIC=&80
 40 FORI%=0 TO 124 STEP4: !(BASMEM+I%)=RND:NEXTI%
 50 FOR I%=0 TO 2 STEP2:P%=START1
 60    [OPTI%
 70    STX MEMLOC+1
 80    .MEMLOC
 90    CPY #0 (Dummy operand)
100    BNE OK
110    LDA INDIC
120    BPL OK
130    SEC
140    RTS
150    .OK
160    LDA #&80
170    STA INDIC
180    LDA DATUM
190    STA BASMEM,X
200    INX
210    CLC
220     RTS:]NEXTI%
230 FOR I%=0 TO 2 STEP2:P%=START2
240    [OPTI%
250    STX MEMLOC+1
260    .MEMLOC
270    CPY #0 (Dummy operand)
280    BNE OK
290    LDA INDIC
300    BMI OK
310    SEC
320    RTS
330    .OK
340    LDA #0
350    STA INDIC
360    LDA BASMEM,Y
370    INY
380    CLC
390     RTS:]NEXTI%
```

```
400 REPEAT
410   INPUT'"DATA",?DATUM
420   IF ?DATUM<128 THEN !&404=USR(START1) ELSE !&404=
      USR(START2)
430   X%=?&405:Y%=?&406
440   IF (?&407 AND 1) =1 THEN PRINT"ERROR"
450   IF X%>Y% THEN LGTH=X%-Y% ELSE IF X%<Y% THEN LGTH=
      256-Y%+X% ELSE IF ?INDIC = 0 THEN LGTH=0 ELSE
      LGTH=256
460   PRINT"HEAD   "?(BASMEM+Y%)
470   PRINT"TAIL   "?(BASMEM+X%-1)
480   PRINT"LENGTH"LGTH
490   UNTIL FALSE
```

Exercise 6.4

Note that all programs should begin in the usual way, defining labels as necessary.

1.

```
50          LDX   #0
60          JSR   OUTPUT2
70          RTS
80          .OUTPUT2
                  .
                  .
                  .
                  .
150         .TEXT:]NEXT
160         $TEXT  =  "This is question one"
```

or

```
150         .TEXT
160         EQUS  "This is question one"
170         EQUB  13:] NEXT
```

2.

```
50          LDX   #0
60          JSR   OUTPUT1
70          RTS
80          .OUTPUT1
                  .
                  .
                  .
                  .
```

```
160             .TEXT:]NEXT
170             $TEXT  =  "This is question two"
```

or

```
160             .TEXT
170             EQUS  "This is question two":] NEXT
```

3.

```
 50             JSR    OSNEWL
 60             RTS:]
```

4.

```
 50             JSR    OSNEWL
 60             LDX    #0
 70             JSR    OUTPUT1
 80             RTS
 90             .OUTPUT1
                .
                .
                .
                .
170             .TEXT:]NEXT
180             $TEXT  =  "This is question four"
```

or

```
170             .TEXT
180             EQUS  "This is question four":]NEXT
```

Exercise 6.5

1.

```
 10 OSWRCH=&FFEE:OSNEWL=&FFE7
 20  DIM START 50
 30 FOR I%=0 TO 2 STEP2
 40   P%=START
 50   [OPTI%
 60   LDA #ASC("?")
 70   JSR OSWRCH
 80   JSR &BC05
 90   LDX #&FF
100   .LOOP1
110   INX
120   LDA &700,X
130   CMP #13
140   BNE LOOP1
150   CPX #0
```

```
160   BEQ NOSTRING
170   .LOOP2
180   DEX
190   LDA &700,X
200   JSR OSWRCH
210   CPX #0
220   BNE LOOP2
230   .NOSTRING
240   JSR OSNEWL
250   JMP START:]NEXTI%
260 CALL START
```

2.

```
 10 OSWRCH=&FFEE:OSASCI=&FFE3
 20  DIM START 50
 30 FOR I%=0 TO 2 STEP2
 40   P%=START
 50   [OPTI%
 60   LDA #ASC("?")
 70   JSR OSWRCH
 80   JSR &BC05
 90   LDA &700
100   CMP #ASC("*")
110   BNE NTSTAR
120   LDA &701
130   CMP #13
140   BEQ FINISH
150   .NTSTAR
160   LDX #&FF
170   .LOOP
180   INX
190   LDA &700,X
200   CMP #ASC(" ")
210   BEQ LOOP
220   JSR OSASCI
230   CMP #13
240   BNE LOOP
250   BEQ START
260   .FINISH
270    RTS:]NEXTI%
280 CALL START
```

3.

```
 10 OSWRCH=&FFEE
 20  DIM START 250
 30 FOR I%=0 TO 2 STEP2
 40   P%=START
 50   [OPTI%
 60   LDA #TEXT MOD 256
 70   STA &37
 80   LDA #TEXT DIV 256
```

```
 90   STA &38
100   LDA #ASC("?")
110   JSR OSWRCH
120   JSR &BCOD
130   LDX #&FF
140   .LOOP1
150   INX
160   LDA TEXT,X
170   CMP #13
180   BNE LOOP1
190   CPX #0
200   BEQ NOSTRING
210   .LOOP2
220   DEX
230   LDA TEXT,X
240   CMP #ASC(" ")
250   BEQ LOOP2
260   INX
270   LDA #13
280   STA TEXT,X
290   .NOSTRING
300   RTS
310   .TEXT:]NEXTI%
320 CALL START
330 PRINT"New length is ";LEN($TEXT)
340 PRINT'"Another?"
350 A$=GET$:IF A$="Y" THEN320 ELSE IF A$="N"
    THEN END ELSE 350
```

Exercise 6.6

(a)

```
 10 DIM NUM1(3),NUM2(3),RESULT(3)
 20 DIM START 50
 30 P%=START
 40 FOR I%=0 TO 3:NUM1(I%)=&70+I%
 50   NUM2(I%)=&74+I%:RESULT(I%)=&78+I%
 60   GOSUB140:NEXTI%
 70 [OPT2
 80 RTS:]
 90 REPEAT
100   INPUT"Numbers to be added",!NUM1(0),!NUM2(0)
110   CALLSTART
120   PRINT!NUM1(0)+!NUM2(0),!RESULT(0)
130   UNTIL FALSE
140 [OPT2:LDA NUM1(I%):]
150 IF I%=0 THEN [OPT2:CLC:]
160 [OPT2:ADC NUM2(I%)
170 STA RESULT(I%):]
180 RETURN
```

(b)

```
 10 DIM NUM1(3),NUM2(3),RESULT(3)
 20 DIM START 50
 30 REPEAT
 40   PRINT"Add or subtract (A/S)?";
 50   REPEAT: A$=GET$
 60     UNTIL A$ ="A" OR A$="S"
 70   P%=START
 80   FOR I%=0 TO 3:NUM1(I%)=&70+I%
 90     NUM2(I%)=&74+I%:RESULT(I%)=&78+I%
100     GOSUB170:NEXTI%
110   [OPT2
120   RTS:]
130   INPUT'"Numbers",!NUM1(0),!NUM2(0)
140   CALLSTART
150   PRINT!NUM1(0)+(A$="S")*!NUM2(0)-(A$="A")
      *!NUM2(0),!RESULT(0)
160   UNTIL FALSE
170 [OPT2:LDA NUM1(I%):]
180 IF I%=0 AND A$="A" THEN [OPT2:CLC:] ELSE IF I%=0
    THEN [OPT2:SEC:]
190 IF A$="A" THEN [OPT2:ADC NUM2(I%):] ELSE [OPT2:
    SBC NUM2(I%):]
200 [OPT2:STA RESULT(I%):]
210 RETURN
```

Exercise 7.1

1. !NEWLOC < !OLDLOC or !NEWLOC ≥ !OLDLOC + !NUML.

2.

```
 10 CLS
 20 NUM=&70:OLDLOC=&72:NEWLOC=&74
 30 DIM START 100
 40 FOR I%=0 TO 2 STEP 2:P%=START
 50   [OPTI%
 60   LDA OLDLOC
 70   CLC
 80   ADC NUM
 90   STA OLDLOC
100   LDA OLDLOC+1
110   ADC NUM+1
120   STA OLDLOC+1
130   DEC OLDLOC+1
140   LDA NEWLOC
150   CLC
160   ADC NUM
170   STA NEWLOC
180   LDA NEWLOC+1
190   ADC NUM+1
200   STA NEWLOC+1
210   DEC NEWLOC+1
```

```
220   LDY #&FF
230   LDX NUM+1
240   BEQ LOLOOP
250   .LOOP1
260   LDA (OLDLOC),Y
270   STA (NEWLOC),Y
280   DEY
290   CPY #&FF
300   BNE LOOP1
310   DEC OLDLOC+1
320   DEC NEWLOC+1
330   DEX
340   BNE LOOP1
350   .LOLOOP
360   LDX NUM
370   BEQ FINISH
380   .LOOP2
390   LDA (OLDLOC),Y
400   STA (NEWLOC),Y
410   DEY
420   DEX
430   BNE LOOP2
440   .FINISH
450   RTS:]NEXTI%
460 INPUT"How many bytes will be moved",!NUM
470 INPUT"Starting address of memory to be moved",A$:
    !OLDLOC=EVAL(A$)
480 INPUT"Starting address of new location",B$:!NEWLOC=
    EVAL(B$)
490 CALL START:PRINT"Memory moved. Checking now."
500 A=EVAL(A$):B=EVAL(B$)
510 FOR I%=0 TO 256*?(NUM+1)+?NUM-1
520   IF ?(A+I%)<>?(B+I%) PRINT "Error at move"I%+1:END
530   NEXTI%
540 PRINT"Check OK":GOTO460
```

This works if !NEWLOC > !OLDLOC or !OLDLOC ≥ !NEWLOC + !NUML.

Listing 7.3 is more efficient.

3. (i) No provision for (NUM + 1) = 0. Put

65 BEQ NOHIGH and 175 .NOHIGH

(ii) In the move of the (NUM) residual bytes, a major fault occurs. For example, if !OLDLOC = &9400, !NEWLOC = &4000 and !NUM = &1040, when the residual bytes are to be moved !OLDLOC is &A400, !NEWLOC is &5000 and X = &40. The first move is &A440 to &5040 and the last &A401 to &5001. &A400 to &5000 has been missed out, and &A440 has been moved erroneously.

This is not correctable without reverting to the method in listing 7.3. The fault is obscured if (NUM) = 0, or if (NUM + 1) = 0 and the correction in (i) is not made.

(iii) The routine fails if !NUM ⩾ 33K or if !OLDLOC − !NEWLOC < 256. Neither of these is correctable.

Exercise 7.2

```
 10 NUMBER=&70:FIRST=&72:SECOND=&74:TEMP=&76:RECLENGTH=
    &77:KEYSTART=&78:KEYEND =&79:BASE=&7A:LOOPCOUNTH=&7C
 20 DIM START 150
 30 FOR I%=0 TO 2 STEP 2:P%=START
 40   [OPTI%
 50   LDA BASE
 60   STA SECOND
 70   LDA BASE+1
 80   STA SECOND+1
 90   LDX #0
100   STX LOOPCOUNTH
110   .BEGIN
120   LDY KEYSTART
130   LDA SECOND+1
140   STA FIRST+1
150   LDA SECOND
160   STA FIRST
170   CLC
180   ADC RECLENGTH
190   STA SECOND
200   BCC LOOP1
210   INC SECOND+1
220   .LOOP1
230   LDA (FIRST),Y
240   CMP (SECOND),Y
250   BCC NEWRECORD
260   BNE SWAP
270   INY
280   CPY KEYEND
290   BCC LOOP1
300   BEQ LOOP1
310   BCS NEWRECORD
320   .SWAP
330   LDY RECLENGTH
340   .LOOP2
350   DEY
360   LDA (FIRST),Y
370   STA TEMP
380   LDA (SECOND),Y
390   STA (FIRST),Y
400   LDA TEMP
410   STA (SECOND),Y
420   CPY #0
430   BNE LOOP2
```

```
440    .NEWRECORD
450    INX
460    BNE NTZERO
470    INC LOOPCOUNTH
480    .NTZERO
490    CPX NUMBER
500    BNE BEGIN
510    LDA LOOPCOUNTH
520    CMP NUMBER+1
530    BNE BEGIN
540    DEC NUMBER
550    BEQ LOWZERO
560    LDA NUMBER
570    CMP #&FF
580    BNE START
590    DEC NUMBER+1
600    BPL START
610    .LOWZERO
620    LDA NUMBER+1
630    BNE START
640    RTS:]NEXT
650   CLS:INPUT"What is the record length",R:
      ?RECLENGTH=R+1
660  INPUT'"What are the limits for the key",
     ?KEYSTART,?KEYEND
670  INPUT'"How many records",N:!NUMBER=N-1
680  DIM B ?(RECLENGTH)*N: !BASE=B
690  PRINT'"Setting up strings now"
700   FOR I%=0 TO N-1:FOR J%= 0 TO R-1:?(B+I%*(R+1)+J%)
      =RND(26)+64:NEXTJ%:?(B+I %*(R+1)+J%)=13 :NEXTI%
710  PRINT"Sorting now.":CALLSTART:PRINT"Checking."
720  FOR I%=0 TO (?RECLENGTH)*(N-2) STEP (?RECLENGTH):
     IF MID$($(B+I%),?(KEYSTART)+1,?KEYEND-?KEYSTART+1)
     >MID$($(B+I%+(?RECLENGTH)),?(KEYSTART)+1,
     ?KEYEND-?KEYSTART+1) THEN PRINT "ERROR
     AT"STR$(I%):END
730    NEXT:PRINT"O.K.":END
```

The reasons for this considerable increase in complication when dealing with more than 256 records will be discussed in section 10.3.

Exercise 7.3

```
10 NUMBER=&70:FIRST=&72:SECOND=&74:TEMP=&76:
   BASE=&77:LOOPCOUNT=&79
20 DIM START 150
30 FOR I%=0 TO 2 STEP 2:P%=START
40   [OPTI%
50   LDA BASE
60   STA SECOND
70   LDA BASE+1
80   STA SECOND+1
90   LDA #0
```

```
100   STA LOOPCOUNT
110   STA LOOPCOUNT+1
120   .BEGIN
130   LDY #0
140   LDA SECOND+1
150   STA FIRST+1
160   LDA SECOND
170   STA FIRST
180   CLC
190   ADC #4
200   STA SECOND
210   BCC NOCARRY
220   INC SECOND+1
230   .NOCARRY
240   LDX #4
250   SEC
260   .LOOP1
270   LDA (SECOND),Y
280   SBC (FIRST),Y
290   INY
300   DEX
310   BNE LOOP1
320   BVC NOOVFLOW
330   EOR #&80
340   .NOOVFLOW
350   EOR #0
360   BPL OVER
370   DEY
380   .LOOP2
390   LDA (FIRST),Y
400   STA TEMP
410   LDA (SECOND),Y
420   STA (FIRST),Y
430   LDA TEMP
440   STA (SECOND),Y
450   DEY
460   BPL LOOP2
470   .OVER
480   INC LOOPCOUNT
490   BNE NTZERO
500   INC LOOPCOUNT+1
510   .NTZERO
520   LDA LOOPCOUNT
530   CMP NUMBER
540   BNE BEGIN
550   LDA LOOPCOUNT+1
560   CMP NUMBER+1
570   BNE BEGIN
580   DEC NUMBER
590   BEQ LOWZERO
600   LDA NUMBER
610   CMP #&FF
620   BNE START
630   DEC NUMBER+1
640   BPL START
650   .LOWZERO
```

```
660   LDA NUMBER+1
670   BNE START
680   RTS:]NEXT
690 CLS:INPUT"How many numbers",N:!NUMBER=N-1:
    DIM B 4*!NUMBER:!BASE=B
700 FOR I%=0 TO N-1:!(B+4*I%)=RND:NEXTI%
710 PRINT"Numbers assigned.  Sorting now":CALLSTART:
    PRINT"Done.  Checking now. "
720 FOR I%=0 TO N-2:IF  !(B+4*I%)>!(B+4+4*I%) THEN
    PRINT"ERROR AT "STR$(I%):END
730   NEXTI% :PRINT"Checking O.K.":END
```

Exercise 7.4

Identical to changes made in exercise 7.2, with LOOP4 and LOOP3 instead of BEGIN and START respectively. The pointer allocation in line 20 will need to be increased also.

Exercise 7.5

1.

```
        CMP   #&61
        BCC   OVER
        CMP   #&7B
        BCS   OVER
        SEC
        SBC   #&20
OVER    JMP   E0A4
```

2.

```
        LDA   #ASC("?")
        JSR   OSWRCH
        JMP   &DEC5
```

Exercise 8.1

1. Box 1 becomes $0 \rightarrow$ RES
$0 \rightarrow$ RES+1
$0 \rightarrow$ RES+2
$0 \rightarrow$ RES+3

Box 2 becomes Shift (MULTER+1; MULTER) right

Box 3 becomes (RES+3; RES+2) + (MULTED+1; MULTED) $\rightarrow$ RES+3; RES+2; RES+1; RES) right

Box 4 becomes Rotate (RES+3; RES+2; RES+1; RES) right

```
 10 MULTER=&70:MULTED=&72:RES=&74
 20 DIM START 50
 30 FOR I%=0 TO 2 STEP2:P%=START
 40   [OPTI%
 50   LDA #0
 60   STA RES
 70   STA RES+1
 80   STA RES+2
 90   STA RES+3
100   LDX #16
110   .LOOP
120   LSR MULTER+1
130   ROR MULTER
140   BCC ZERO
150   LDA RES+2
160   CLC
170   ADC MULTED
180   STA RES+2
190   LDA RES+3
200   ADC MULTED+1
210   STA RES+3
220   .ZERO
230   ROR RES+3
240   ROR RES+2
250   ROR RES+1
260   ROR RES
270   DEX
280   BNE LOOP
290   RTS:]NEXTI%
300 CLS:REPEAT
310    INPUT"Numbers to be multiplied",A,B:!MULTE
       =A:!MULTED=B
320   CALLSTART
330   PRINTA*B,16777216*?(RES+3)+65536*?(RES+2)+
      256*?(RES+1)+?RES
340   UNTIL FALSE
```

The accumulator is used for the multiple precision add in lines 160 to 210 so it cannot be used to store part of the result.

2.

```
 10 MULTER=&70:MULTED=&71:RES=&72
 20 DIM START 150
 30 FOR I%=0 TO 2 STEP2: P%=START
 40   [OPTI%
 50   LDY #0
 60   LDX #0
```

```
 70   LDA MULTER
 80   BPL PLUS1
 90   INX
100   EOR #&FF
110   CLC
120   ADC #1
130   STA MULTER
140   .PLUS1
150   LDA MULTED
160   BPL PLUS2
170   DEX
180   EOR #&FF
190   CLC
200   ADC #1
210   STA MULTED
220   .PLUS2
230   TXA
240   BEQ PLUS
250   LDY #1
260   .PLUS
270   LDA #0
280   STA RES
290   LDX #8
300   .LOOP
310   LSR MULTER
320   BCC ZERO
330   CLC
340   ADC MULTED
350   .ZERO
360   ROR A
370   ROR RES
380   DEX
390   BNE LOOP
400   STA RES+1
410   TYA
420   BEQ ANSPL
430   SEC
440   LDA #0
450   SBC RES
460   STA RES
470   LDA #0
480   SBC RES+1
490   STA RES+1
500   .ANSPL
510   RTS:]NEXTI%
520 CLS:REPEAT
530   INPUT"Numbers to be multiplied",A,B
540    IF A<0 THEN C=256+A ELSE C=A
550    IF B<0 THEN D=256+B ELSE D=B
560    ?MULTER=C:?MULTED=D:CALLSTART
570   PRINTA*B,
580   IF ?(RES+1)>127 THEN PRINT(?(RES+1)-255)*
      256+?RES-256 ELSE PRINT?(RES+1)*256+?RES
590   UNTIL FALSE
```

The details are

50	Flag for sign of result
60	Indicator for each sign
70–130	Check if multiplier negative, and if so increment X and obtain two's complement of multiplier
150–210	Check if multiplied number is negative, and if so decrement X and obtain two's complement of multiplied number
230–250	If X is not zero, result will be negative. Set flag in Y
270–400	As listing 8.3
410–490	If result is to be negative, form two's complement of result by subtracting from zero

3. There are many possible solutions, one of which is to use listing 8.3, but the following is shorter and quicker. It does the computation by evaluating 256 * Y + X − 6 * Y. It is often the case that knowledge of the multiplier can allow quicker routines than the general-purpose listing 8.3, and this is such a case.

```
 10 RES=&70
 20 DIM START 50
 30 FOR I%=0 TO 2 STEP 2
 40   P%=START
 50   [OPTI%
 60   STY MEMLOC+1
 70   STY RES+1
 80   TXA
 90   LDY #6
100   .LOOP
110   SEC
120   .MEMLOC
130   SBC #0 Dummy operand
140   BCS NOBORROW
150   DEC RES+1
160   .NOBORROW
170   DEY
180   BNE LOOP
190   STA RES
200   LDY MEMLOC+1
210   RTS:]NEXTI%
220 CLS:REPEAT
230   INPUT"What are X and Y",X%,Y%
240   CALLSTART
250   PRINT250*Y%+X%,256*?(RES+1)+?RES
260   UNTIL FALSE
```

Lines 70 and 80 compute 256 * Y + X (with X in the accumulator).
Lines 90 to 180 subtract Y six times, giving 250 * Y + X.

It is possible to use TEMP to save Y, instead of the internal location MEMLOC, but while this saves 2 bytes it costs 4 cycles of time (see appendix 1).

Exercise 8.2

```
Put  193          DEC   NUMBER+1
     196          BPL   BACK
```

and change ?NUMBER to !NUMBER in line 400.

Exercise 8.3

1. The REPWHILE loop executes infinitely. Put BEQ MISTAKE at 75, where MISTAKE is some error-handing routine (see listing 8.8).

2.

```
 10 DVID=&70:DVIS=&72:QUOT=&74
 20 DIM START 50
 30 FOR I%=0 TO 2 STEP 2:P%=START
 40   [OPTI%
 50   LDX #0
 60   LDA #0
 70   STA QUOT
 80   STA QUOT+1
 90   LDA DVIS+1
100   BMI LOOP
110   ORA DVIS
120   BEQ FINISH
130   .REPWHILE
140   INX
150   ASL DVIS
160   ROL DVIS+1
170   BPL REPWHILE
180   .LOOP
190   LDA DVID
200   CMP DVIS
210   LDA DVID+1
220   SBC DVIS+1
230   BCC LESS
240   INC QUOT
250   LDA DVID
260   SEC
270   SBC DVIS
280   STA DVID
290   LDA DVID+1
300   SBC DVIS+1
310   STA DVID+1
320   .LESS
330   DEX
340   BMI FINISH
350   LSR DVIS+1
360   ROR DVIS
370   ASL QUOT
380   ROL QUOT+1
```

```
390   JMP LOOP
400   .FINISH
410    RTS:]NEXTI%
420 CLS:REPEAT
430   INPUT"Dividend",DD
440   INPUT"Divisor",DS
450   !DVID=DD:!DVIS=DS
460   CALLSTART
470   PRINTDD DIV DS,DD MOD DS
480   PRINT256*?(QUOT+1)+?QUOT,256*?(DVID+1)+?DVID
490   UNTIL FALSE
```

3. The following code should be inserted after line 390 in the listing in question 2, replacing BEQ or BMI FINISH in lines 120 and 340 by BMI ROUND. Alter lines 460 and 480 accordingly.

```
ROUND    LSR  DVIS+1   \
         ROR  DVIS      |  Divide divisor by two.
         BCC  NOFRAC    |  If there is a half,
         INC  DVIS      |  then round up.
         BNE  NOFRAC    |
         INC  DVIS+1    /
NOFRAC   LDA  DVID      \
         CMP  DVIS      |  Compare remainder (in DVID)
         LDA  DVID+1    |  with half the divisor (in DVIS).
         SBC  DVIS+1    |
         BCC  FINISH    /
         INC  QUOT      \
         BNE  FINISH    |  If remainder is at least half
         INC  QUOT+1    |  the divisor round up.
FINISH   RTS            /
```

4.

```
 10 DVID=&70:DVIS=&71:QUOT=&72
 20 DIM START 150
 30 FOR I%=0 TO 2 STEP 2:P%=START
 40   [OPTI%
 50   LDY #0
 60   LDX #0
 70   LDA DVIS
 80   BPL PLUS1
 90   INX
100   EOR #&FF
110   CLC
120   ADC #1
130   STA DVIS
```

```
140   .PLUS1
150   LDA DVID
160   BPL PLUS2
170   DEX
180   EOR #&FF
190   CLC
200   ADC #1
210   STA DVID
220   .PLUS2
230   TXA
240   BEQ PLUS
250   LDY #1
260   .PLUS
270   LDX #0
280   STX QUOT
290   LDA DVIS
300   BEQ ANSPL
310   .REPWHILE
320   BMI LOOP
330   INX
340   ASL DVIS
350   BPL REPWHILE
360   .LOOP
370   LDA DVID
380   CMP DVIS
390   BCC LESS
400   INC QUOT
410   LDA DVID
420   SEC
430   SBC DVIS
440   STA DVID
450   .LESS
460   DEX
470   BMI ROUND
480   LSR DVIS
490   ASL QUOT
500   JMP LOOP
510   .ROUND
520   LSR DVIS
530   BCC NOFRAC
540   INC DVIS
550   .NOFRAC
560   LDA DVID
570   CMP DVIS
580   BCC SIGN
590   INC QUOT
600   .SIGN
610   TYA
620   BEQ ANSPL
630   SEC
640   LDA #0
650   SBC QUOT
660   STA QUOT
670   .ANSPL
680   RTS:]NEXTI%
690 CLS:REPEAT
```

```
700   INPUT"Dividend",A:IF A<0 THEN C=256+A ELSE C=A
710   INPUT"Divisor",B:IF B<0 THEN D=256+B ELSE D=B
720   ?DVID=C:?DVIS=D:CALLSTART
730   PRINTA/B
740   IF ?QUOT>127 PRINT ?QUOT-256 ELSE PRINT?QUOT
750   UNTIL FALSE
```

The details are

50–250	Virtually identical to first half of solution for question 2, exercise 8.1
270–500	Identical to listing 8.7 (with the addition in question 1)
520–590	Equivalent to question 3 for 8-bit numbers
610–660	Identical to last part of question 2, exercise 8.1

Exercise 8.4

```
 10 DVID=&70:DVIS=&72:QUOTH=&74:OSWRCH=&FFEE
 20 DIM START 100
 30 FOR I%=0 TO 2 STEP 2:P%=START
 40   [OPTI%
 50   LDA DVIS
 60   ORA DVIS+1
 70   BEQ MISTAKE
 80   LDA #0
 90   STA QUOTH
100   LDX #16
110   .LOOP
120   ASL DVID
130   ROL DVID+1
140   ROL A
150   TAY
160   ROL QUOTH
170   CMP DVIS
180   LDA QUOTH
190   SBC DVIS+1
200   BCC LESS
210   TYA
220   SBC DVIS
230   TAY
240   LDA QUOTH
250   SBC DVIS+1
260   STA QUOTH
270   INC DVID
280   .LESS
290   TYA
300   DEX
310   BNE LOOP
320   RTS
330   .MISTAKE
340   LDA #ASC("?")
350   JSR OSWRCH
```

```
360    LDA #7
370    JSR OSWRCH
380    RTS:]NEXTI%
390 CLS:REPEAT
400    INPUT"Dividend",A
410    INPUT"Divisor",B
420    !DVID=A:!DVIS=B
430    !&403=USRSTART
440    PRINTA DIV B,A MOD B
450    PRINT256*?(DVID+1)+?DVID,256*?QUOTH+?&403
460    UNTIL FALSE
```

The details are

50–70	Output error message if (DVIS+1; DVIS) is zero
120–160	Shift left (DVIS+1; DVIS) one bit into (QUOTH; A) saving the accumulator temporarily in Y
170–200	Compare (QUOTH; A) to (DVIS+1; DVIS)
210–270	If quotient not less than divisor, retrieve the accumulator, subtract (DVIS) (the carry must already be set), save the accumulator in Y, subtract (DVIS+1) from (QUOTH) with any borrow, and increment the dividend by one
290–320	Retrieve the accumulator, loop 16 times, and then return
340–380	If a division by zero occurs, output a query sign and a short beep, and return

The program is shorter and quicker.

Exercise 9.1

1.

```
 10 OSWORD=&FFF1:OSWRCH=&FFEE
 20 DIM START 100
 30 !&80=&FFFFFF9C:?&84=&FF
 40 FOR I%=0 TO 3 STEP 3:P%=START
 50    [OPTI%
 60    .SETCLOCK
 70    LDA #4
 80    LDX #&80
 90    LDY #0
100    JSR OSWORD
110    RTS
120    .ENTRY
130    STA &FC
140    PHA:TXA:PHA:TYA:PHA:PHP
150    LDA &FC
160    CMP #5
170    BNE FINISH
```

```
180   LDA #7
190   JSR OSWRCH
200   JSR SETCLOCK
210   .FINISH
220   PLP:PLA:TAY:PLA:TAX:PLA
230   RTS:]NEXT
240 ?&220=ENTRY MOD 256:?&221=ENTRY DIV 256
250 *FX14,5
260 CALL SETCLOCK
```

2.

```
 10 OSWRCH=&FFEE:OSWORD=&FFF1
 20 DIM START 500
 30 FOR I%=0 TO 2 STEP 2
 40   P%=START
 50   [OPT I%
 60   LDA #TEXT MOD 256
 70   STA &80
 80   LDA #TEXT DIV 256
 90   STA &81
100   LDA #19
110   STA &82
120   LDA #&41
130   STA &83
140   LDA #&5A
150   STA &84
160   .BEGIN
170   LDA #ASC("?")
180   JSR OSWRCH
190   LDX #&80
200   LDY #0
210   LDA #0
220   JSR OSWORD
230   BCC CR
240   RTS
250   .CR
260   LDA &80
270   ADC #20 \ Carry already clear
280   STA &80
290   BCC BEGIN
300   INC &81
310   BCS BEGIN
320   .TEXT:]NEXT
330 CALL START
```

Exercise 9.2

1.

```
PHP
PLA
ORA   #&C0
PHA
PLP
```

2.

```
            SEC
            PHP
LOOP1       PLP
            LDA   (SECOND), Y
            SBC   (FIRST), Y
            INY
            PHP
            CPY   #4
            BNE   LOOP1
```

The first is more efficient in memory space, since fewer bytes are used; and more efficient in time in that the loop here contains two excess instructions.

Exercise 9.3

1.

```
            TAY
            TXA
            PHA
            TYA
            TAX
            PLA
```

2. (i)

```
            PHA
            STX   MEMLOC+1
            SEC
MEMLOC      SBC   #0          Dummy
            TAX
            PLA
```

(ii)

```
            PHA
            TXA
            STY   MEMLOC+1
MEMLOC      ADC   #0          Dummy
            TAX
            PLA
```

Exercise 9.4

```
10 COLUMNS=&70:ROWS=&71:COLCOPY=&72:LIMIT=&73:
   BEGINCONTROL=&74:HIBYTE=&75:LOCATION=&76:
   STORE=&78:OSWRCH=&FFEE:OSBYTE=&FFF4
20 FORI%=0 TO 2 STEP 2:P%=&D01:RESTORE
30   [OPTI%
40   LDA #3
50   STA LIMIT
```

```
 60   LDX #0
 70   JSR CONTROL:LDA #2:JSR OSWRCH
 80   LDX #4
 90   LDA #&85
100   JSR OSBYTE
110   STY MEMLOC+1
120   LDA #&84
130   JSR OSBYTE
140   STY HIBYTE
150   .MEMLOC
160   CPY #0 Dummy operand
170   BNE ZEROMODE
180   LDA #7
190   STA LIMIT
200   LDA #3
210   STA BEGINCONTROL
220   LDA #40
230   BNE FOURMODE
240   .ZEROMODE
250   LDA #11
260   STA LIMIT
270   LDA #7
280   STA BEGINCONTROL
290   LDA #80
300   .FOURMODE
310   STA COLUMNS
320   LDA &322
330   STA LOCATION
340   LDA &323
350   STA LOCATION+1
360   LDA #32
370   STA ROWS
380   .BEGIN
390   LDA COLUMNS
400   STA COLCOPY
410   LDX BEGINCONTROL
420   JSR CONTROL
430   .LOOP1
440   LDY #7
450   .LOOP2
460   LDA (LOCATION),Y
470   STA STORE,Y
480   DEY
490   BPL LOOP2
500   LDY #8
510   .LOOP3
520   LDX #7
530   LDA #1
540   JSR OSWRCH
550   .LOOP4
560   ASL STORE,X
570   ROR A
580   DEX
590   BPL LOOP4
600   JSR OSWRCH
610   DEY
```

```
 620    BNE LOOP3
 630    LDA LOCATION
 640    CLC
 650    ADC #8
 660    STA LOCATION
 670    BCC NOCARRY
 680    INC LOCATION+1
 690    .NOCARRY
 700    DEC COLCOPY
 710    BNE LOOP1
 720    LDA #1
 730    JSR OSWRCH
 740    LDA #&0D
 750    JSR OSWRCH
 760    LDA LOCATION+1
 770    BPL OVER
 780    LDA HIBYTE
 790    STA LOCATION+1
 800    .OVER
 810    DEC ROWS
 820    BNE BEGIN
 830    LDA #13
 840    STA LIMIT
 850    LDX #11
 860    JSR CONTROL:LDA #3:JSR OSWRCH
 870    RTS
 880    .CONTROL
 890    LDA #1
 900    JSR OSWRCH
 910    LDA TABLE,X
 920    JSR OSWRCH
 930    INX
 940    CPX LIMIT
 950    BNE CONTROL
 960    RTS
 970    .TABLE:]NEXTI%
 980 FOR I%=1 TO 13
 990    READ ?P%
1000    P%=P%+1:NEXTI%
1010 DATA27,65,8,27,75,64,1,27,76,128,2,27,50
```

The details are

40–130	As listing 8.6
140	Only the high byte of physical screen memory is required to be saved
150–310	As listing 8.6, 160–320
320–350	Store start of actual screen memory (that is, taking into account changes due to scrolling) in LOCATION
360–750	As listing 8.6, 330–720
760–790	If we go beyond &7FFF, the high byte becomes negative, and we replace it by the high byte of the start of physical

screen memory (that is, the start of actual screen memory when there has been no scrolling)
800–1010 As listing 8.6, 730–930

Notice that this program is longer; so if we know that there will be no scrolling (typically so when using the high-resolution graphics), listing 8.6 is the better choice if memory is at a premium.

Index

Details of Cassette

A software cassette is available to accompany this book. This cassette is obtainable through all major bookshops, but in case of difficulty order direct from

Globe Book Services
Houndmills
Brunel Road
Basingstoke
Hampshire RG21 2XS

ISBN 0–333–38267–6

The cost of the cassette is £9.00 (including VAT). This price applies to the United Kingdom.