Assembly Language Programming on the TRS-80 MC-10

Using the Virtual MC-10

and the TASM compiler

Getting Started

- Pre-requisites for the tutorial.
  - Virtual MC-10 version 0.7 or higher
  - Familiarity with MICROCOLOR BASIC 1.0 (this is the basic that comes with the MC-10)
  - Basic knowledge of “hexadecimal numbers”
  - A text editor. This tutorial uses the “Notepad” editor that comes bundled with Microsoft Windows.

Getting Started

- Download the Virtual MC-10
  1. As of 2008, the emulator can be found at http://www.geocities.com/emucompboy.
  2. Find and download the “Virtual MC-10”
  3. Follow any additional installation instructions that come with the emulator
  4. A good book on assembly is helpful, but hopefully not required. I found William Barden Jr.’s “Assembly Language on the TRS-80 Color Computer” to be very helpful.

Hexadecimal Numbers

- Good tutorials exist on the web for hexadecimal numbers.
- The Virtual MC-10 and the bundled TASM compiler use a “$” pre-fix to distinguish a hexadecimal representation of a number from its decimal equivalent.
- The tutorials slowly introduce hexadecimal (and binary) numbers over time.
- The sections involving the video display modes and memory map use hexadecimal numbers almost exclusively.
- The Virtual MC-10 debug windows displays information primarily with hexadecimal numbers, but usually also displays their decimal equivalents.

Other Material

- A good book on assembly is helpful, but hopefully not required.
- William Barden Jr.’s “Assembly Language Programming on the TRS-80 Color Computer” covers the MC-10’s “big brother” (the MC6809E processor found in the CoCo).
- You may also find good information if you search for “6800 assembly tutorial” on the internet. The MC-10 uses a 6801 processor, which has only a few additional instructions.

MC-10 Hardware

- The CPU: (Motorola MC6803)
- A video display chip (Motorola MC6847)
- On board memory shared between the MC6803 and MC6847) totalling 4K
- An input/ouput buffer to control the RS232-C port and cassette port.
- A latch to control the TV's speaker
- An expansion bus accessible by the CPU
- The infamous "rubber chicklet" (a.k.a. "rubber chicken") keyboard.

MC-10 Memory Map

Address	Component
$00-$FF	On-chip RAM
$0100-$03FF	Unused
$4000-$4FFF	4K Video RAM
$5000-$8FFF	16K Expansion
$9000-$BFFF	Video/Sound Control
$C000-$DFFF	Mirror of ROM
$E000-$FFFF	BASIC ROM

On-Chip Memory

Address	Function	Description
$00	Port 1 Data Direction Register	Enables Keyboard (Set to 255)
$01	Port 2 Data Direction Register	Enables Keyboard (Set to 255)
$02	Port 1 Data Register	Keystrobe
$03	Port 2 Data Register	Keystrobe
$04-$07	Unused	
$08	Timer Control and Status	Working!
$09-$0A	16-bit counter	Working!
$0B-$0C	Output Compare Register	Working!
$0D-$0E	Input Capture Register	Not emulated by VMC-10
$0F	Unused	
$10	Rate and Mode Control	Not emulated by VMC-10
$11	Transceiver Control/Status	Not emulated by VMC-10
$12	Receive Data	Not emulated by VMC-10
$13	Transmit Data	Not emulated by VMC-10
$14	RAM Control Register	Not emulated by VMC-10
$15-$1F	Internal Registers	Not emulated by VMC-10
$20-$7F	Unused	
$80-$FF	User RAM	Heavily used by BASIC

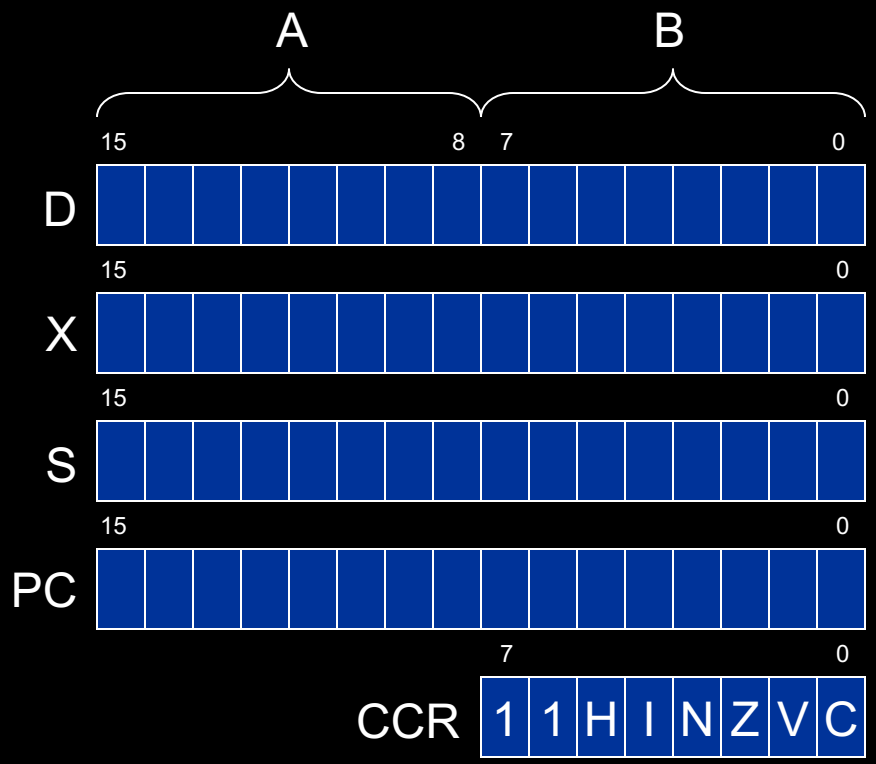
Video/Sound Control

49151 ($BFFF)

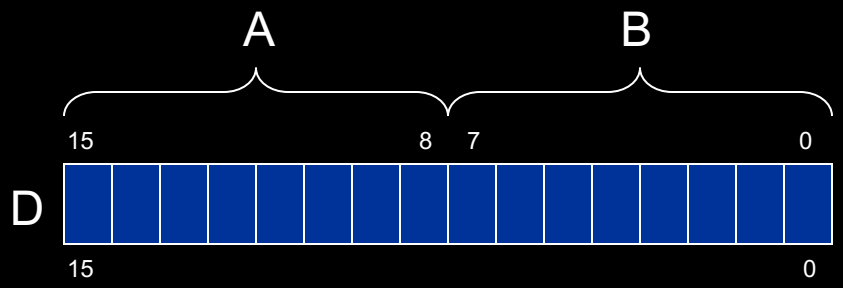
7	6	5	4	3	2	1	0
Sound	CSS	A/G	GM0	GM1	GM2 INT/EXT	N/C	N/C

- Operation is controlled by writing to any memory location between $9000-$BFFF (36864-49151).
- The convention used by MC-10 programmers was to use $BFFF. This would have allowed programs to be backwards-compatible on any future version of the MC-10.
- The speaker is energized / de-energized by setting bit 7 of the control register.
- The remaining bits controls the various graphics mode of the MC6847 chip. (we will cover the graphics modes in a later section)

6803 Registers



Accumulator



Register D is known as the “double accumulator”
It a 16-bit register on which you can perform addition/subtraction.
You can also shift it left or right.

Register D is broken up into two eight bit registers:
•A, the “high“ byte
•B, the “low” byte
You can perform any arithmetic or bitwise operation on the 8-bit registers
Unsigned multiplication of A with B can be performed, the result is stored in-place in the D register. There are no division instructions.

6803 Registers

Register X is a 16-bit register which contains an address you can read from or write to.

You can also add a constant positive 8-bit offset to it and compare it with other 16-bit values.

6803 Registers

Register S is known as the stack pointer.

It is intended to contain an address to a contiguous section of memory known as the stack. The stack can only grow or shrink in size from one end. You can “push” or “pull” information from the top of the stack.

The stack is primarily used to keep track of subroutine addresses, local variables, or temporary storage.

6803 Registers

Register PC is the program counter

It contains the address of the next instruction to be executed.

6803 Registers

Register CCR is the 8-bit condition code register.

The two most significant bits are always set to one.
The other six bits are updated after every instruction to contain status information.

Half-carry - facilitates binary coded decimal arithmetic
It is used by the DAA instruction.

- Interrupt - prevents the timers from redirecting the PC counter to a different code segment
- Negative - indicates that a negative result occurred.
- Zero - indicates that the result of the last instruction was zero.
- oVerflow - indicates that the result could not be fit into an 8-bit signed number.
- Carry - indicates that the result could not be fit into an 8-bit unsigned number.

6803 Instructions

aba	beq	bra	daa	jmp	mul	rol	staa	tst
abx	bne	brn	dec	jsr		rola	stab	tsta
adca	bcc		deca		neg	rolb	std	tstb
adcb	bcs	cba	decb	ldaa	nega	ror	sts	
adda	bhi	clc	des	ldab	negb	rora	stx	wai
addb	bhs	cli	dex	ldd	nop	rorb	suba	
addd	blo	clr		lds		rti	subb	
anda	bls	clra	eora	ldx	oraa	rts	subd	
andb	bge	clrb	eorb	lsl	orab		swi	
asr	bgt	clv		lsla		sba		
asra	ble	cmpa	inc	lslb	psha	sbca	tab	
asrb	blt	cmpb	inca	lsld	pshb	sbcb	tap	
	bmi	cpx	incb	lsr	pshx	sec	tba	
bita	bpl	com	ins	lsra	pula	sei	tpa	
bitb	bvc	coma	inx	lsrb	pulb	sev	tsx	
bsr	bvs	comb		lsrd	pulx		txs	

Addressing Modes

Syntax	Mode	Instruction Size (bytes)
<opcode>	(inherent)	1 byte
<opcode> #value	(immediate)	1 byte + 1 or 2 byte value
<opcode> address	(direct)	1 byte + 1 byte address
<opcode> address	(extended)	1 byte + 2 byte address
<opcode> offset,x	(indexed)	1 byte + 1 byte offset
<branch op> address	(relative)	1 byte + 1 byte offset

Inherent	- does not take an operand
Immediate	- operand as data
Direct	- operand as address to on-chip memory (<256)
Extended	- operand as an address to off-chip memory (>255)
Indexed	- add positive single byte offset to x, use result as address
Relative	- operand as an address up to 128 bytes before or 127 bytes after the next instruction.

Cycle Counting

- Indexed and extended instructions will take the same number of clock cycles.
- Direct addressing instructions take one clock cycle less than either indexed or extended instructions.
- Immediate addressing instructions take one less cycle than direct addressing.

Tutorial #1 – load/store

aba	beq	bra	daa	jmp	mul	rol	staa	tst
abx	bne	brn	dec	jsr		rola	stab	tsta
adca	bcc		deca		neg	rolb	std	tstb
adcb	bcs	cba	decb	ldaa	nega	ror	sts	
adda	bhi	clc	des	ldab	negb	rora	stx	wai
addb	bhs	cli	dex	ldd	nop	rorb	suba	
addd	blo	clr		lds		rti	subb	
anda	bls	clra	eora	ldx	oraa	rts	subd	
andb	bge	clrb	eorb	lsl	orab		swi	
asr	bgt	clv		lsla		sba		
asra	ble	cmpa	inc	lslb	psha	sbca	tab	
asrb	blt	cmpb	inca	lsld	pshb	sbcb	tap	
	bmi	cpx	incb	lsr	pshx	sec	tba	
bita	bpl	com	ins	lsra	pula	sei	tpa	
bitb	bvc	coma	inx	lsrb	pulb	sev	tsx	
bsr	bvs	comb		lsrd	pulx		txs	

Load/Store Opcodes

ldaa - load 8-bit accumulator "a“
ldab - load 8-bit accumulator "b“
ldd - load 16-bit double accumulator "d“
ldx - load16-bit index register "x“
lds - load16-bit index register "x“

staa - store accumulator a
stab - store accumulator d
std - store double accumulator
stx - store x register
sts - store s register

Load/Store Opcodes

ldaa, ldab - can use direct, extended, indexed, or 1 byte immediate.

ldd, ldx, lds - can use direct, extended, indexed, or 2 byte immediate.

staa, stab, std, stx, sts

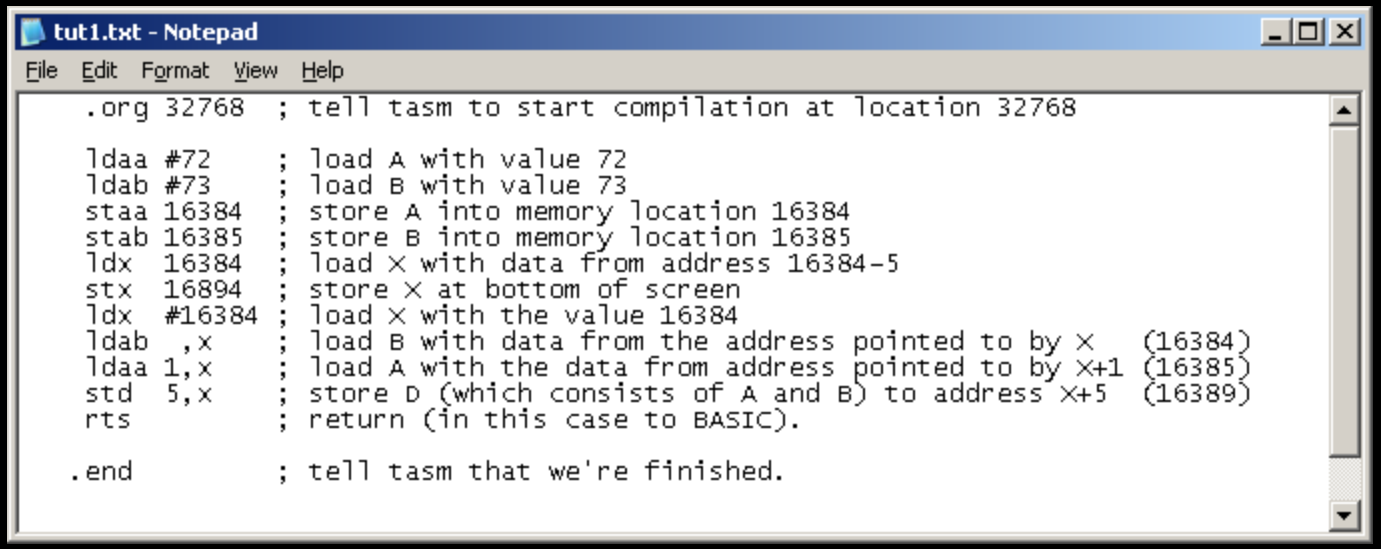
- can use only direct, extended or indexed modes.

Example Program

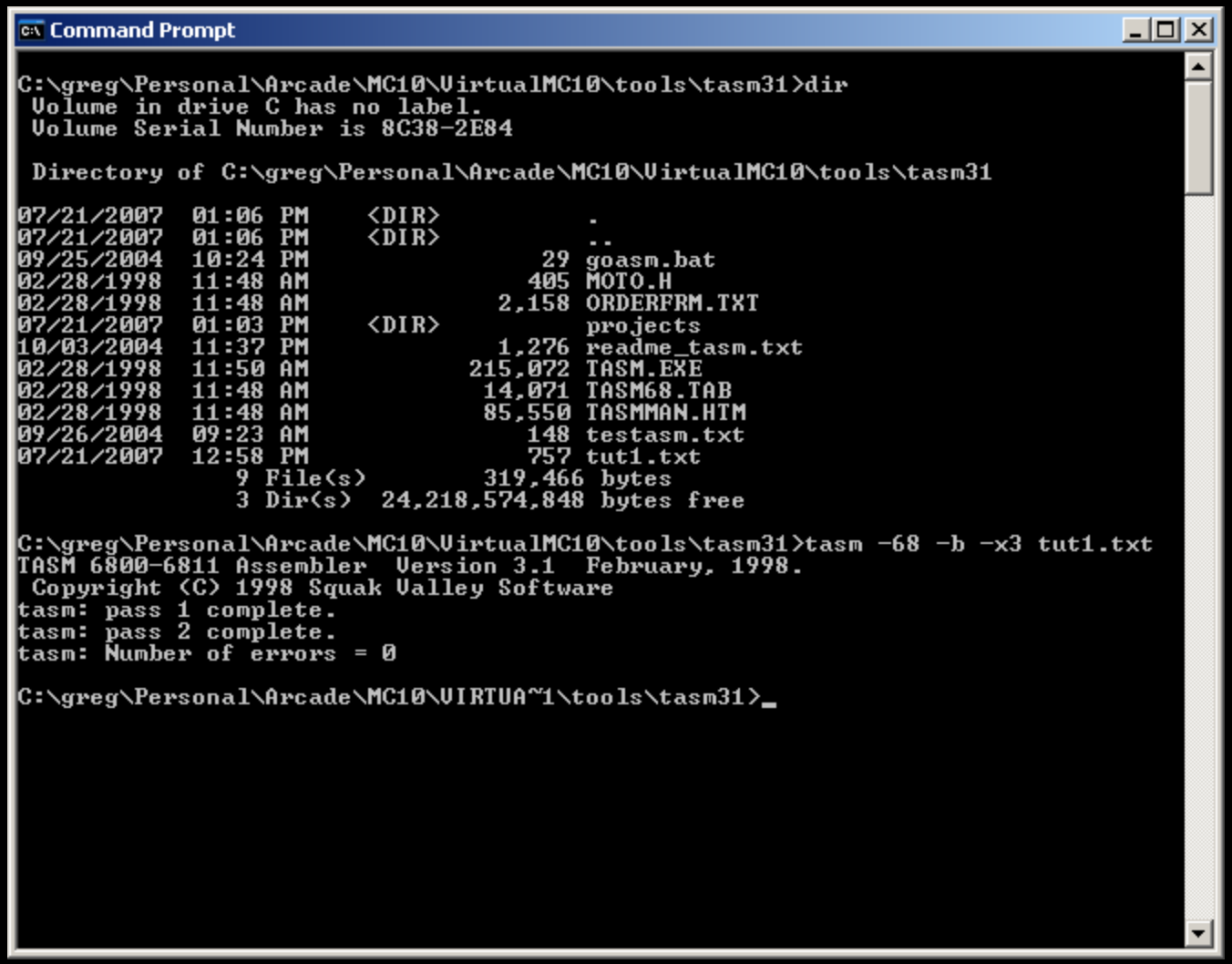
We'll ignore the S, PC, and CCR registers for now, and just focus on the accumulator and index registers…we'll learn more about the stack and condition codes later.

Here's an example – note the spaces at the beginning of each line. The MC-10's text display starts at location $4000 (16384) and ends at $41FF (16895).

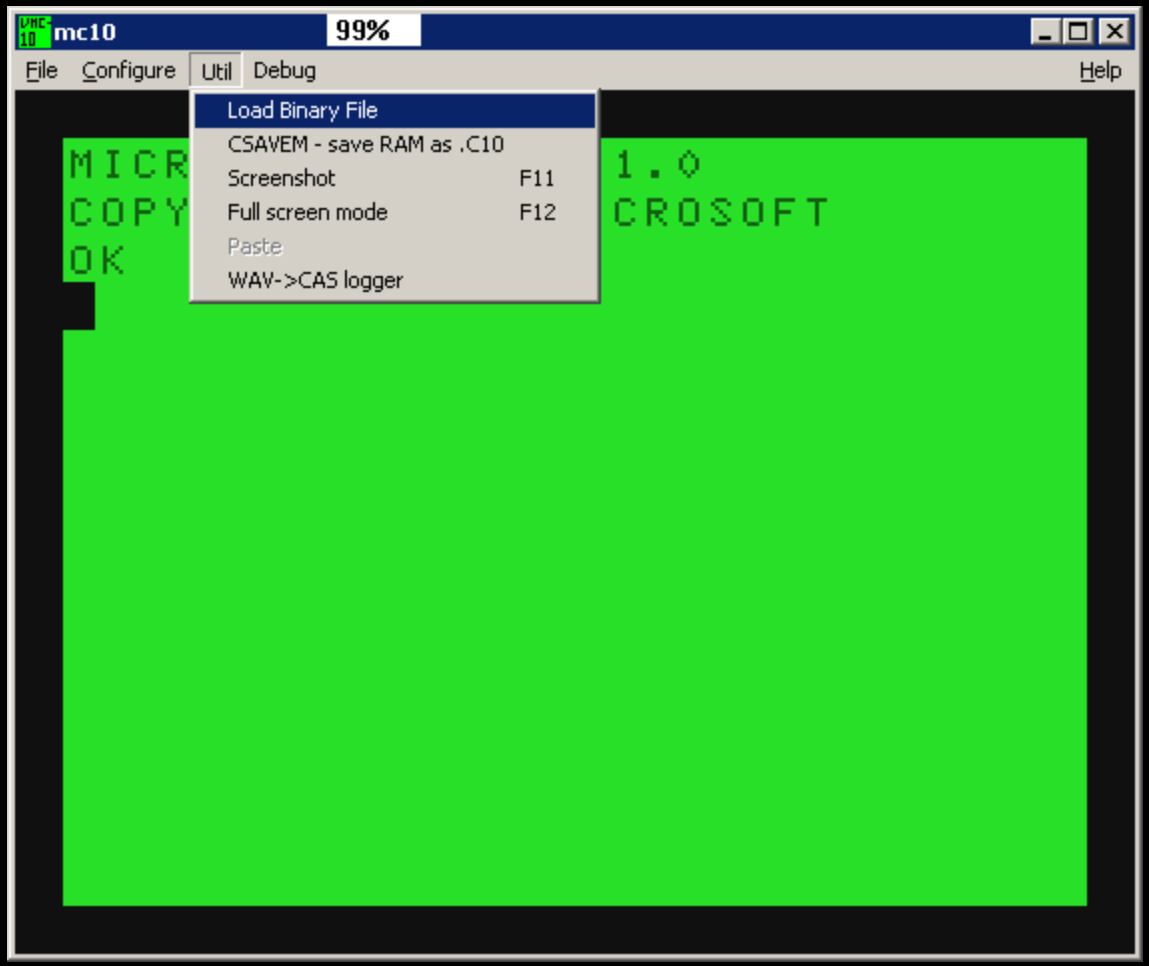
Let's write 72(H) and 73(I) to the screen in various locations:



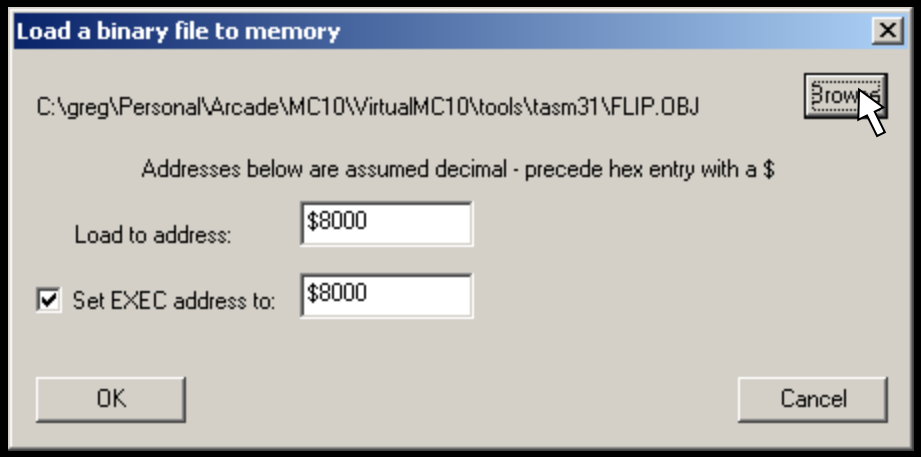
Use TASM to compile



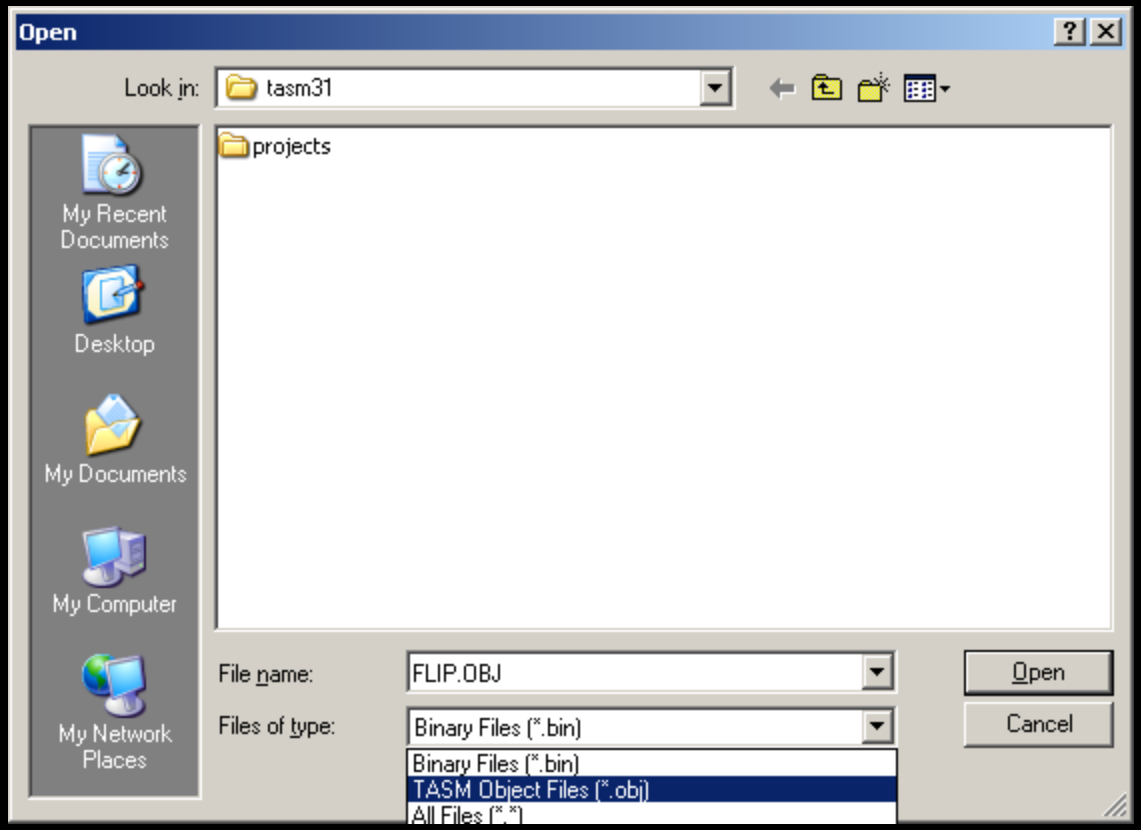
Use Virtual MC-10 to read in



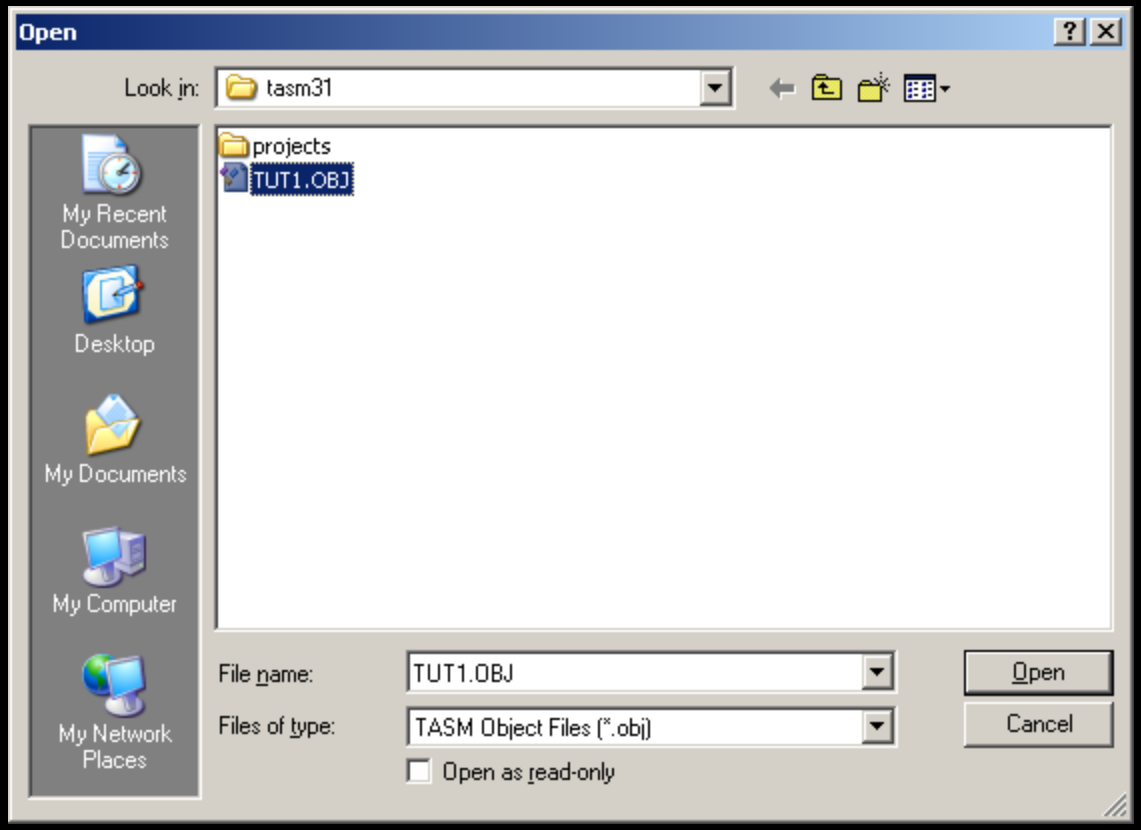
Browse to get your object



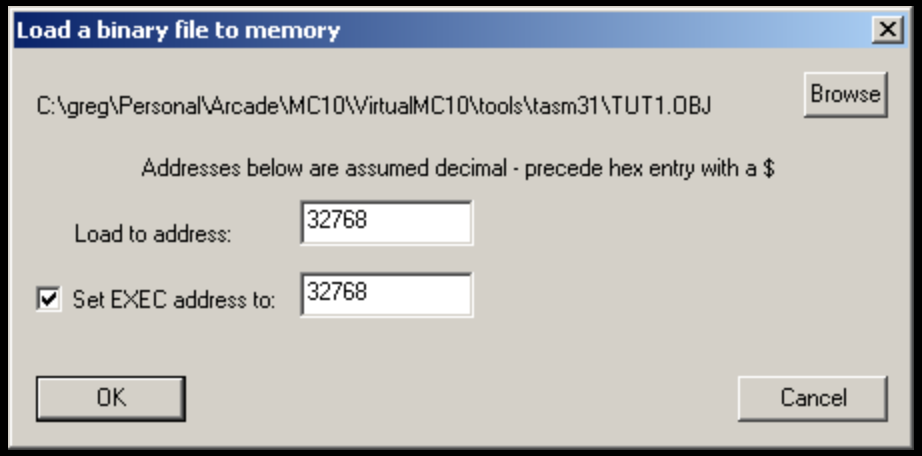
Look for TASM *.obj files



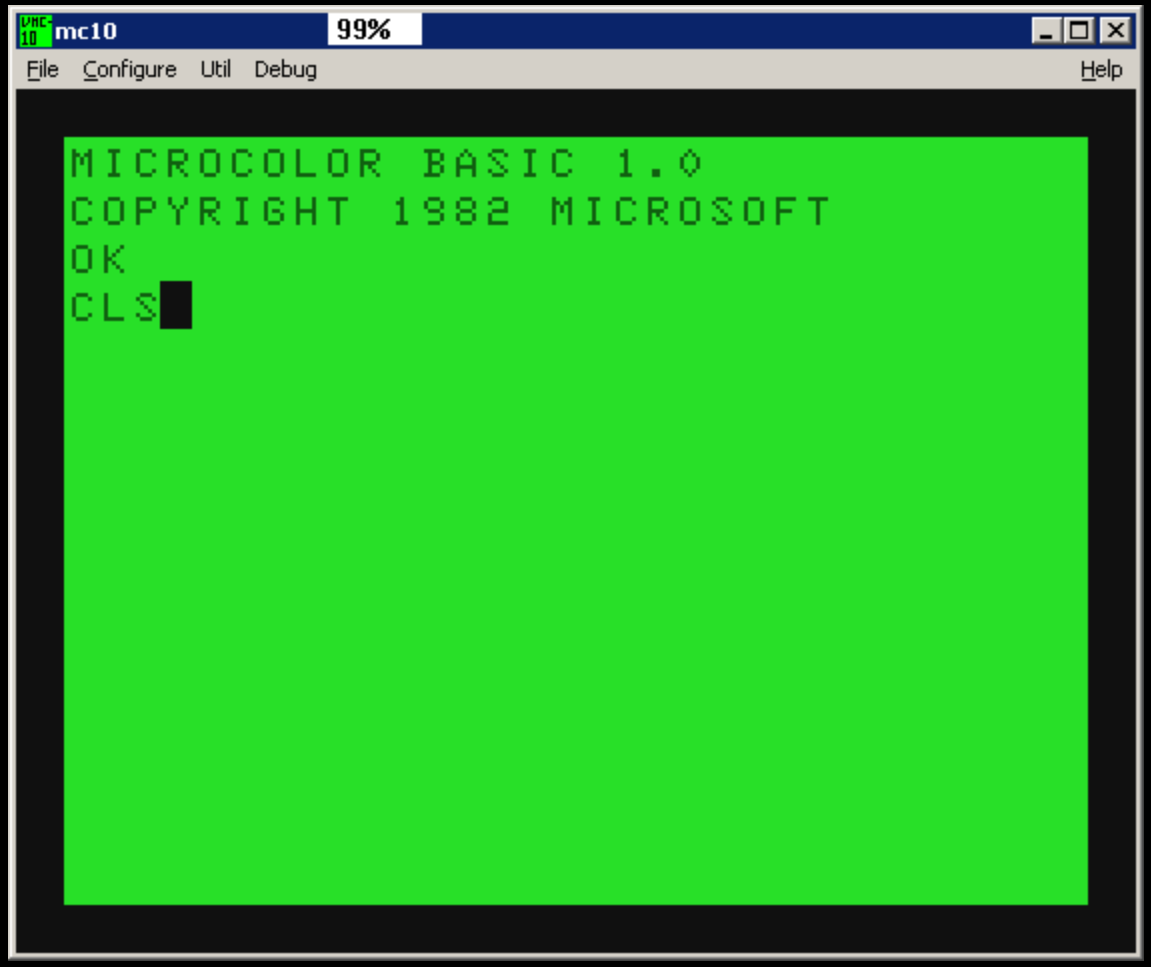
Open the obj file



Now set where it should go

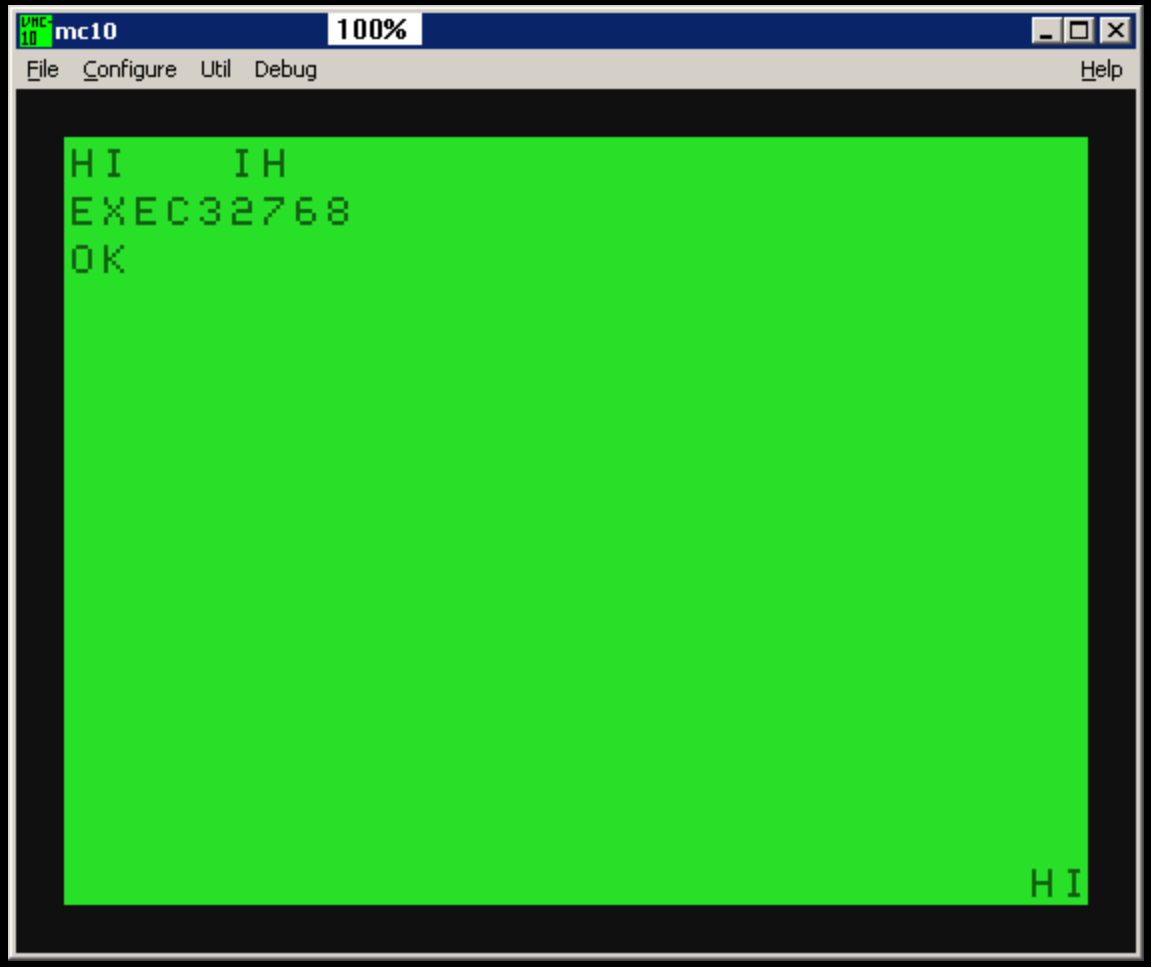


Clear the screen

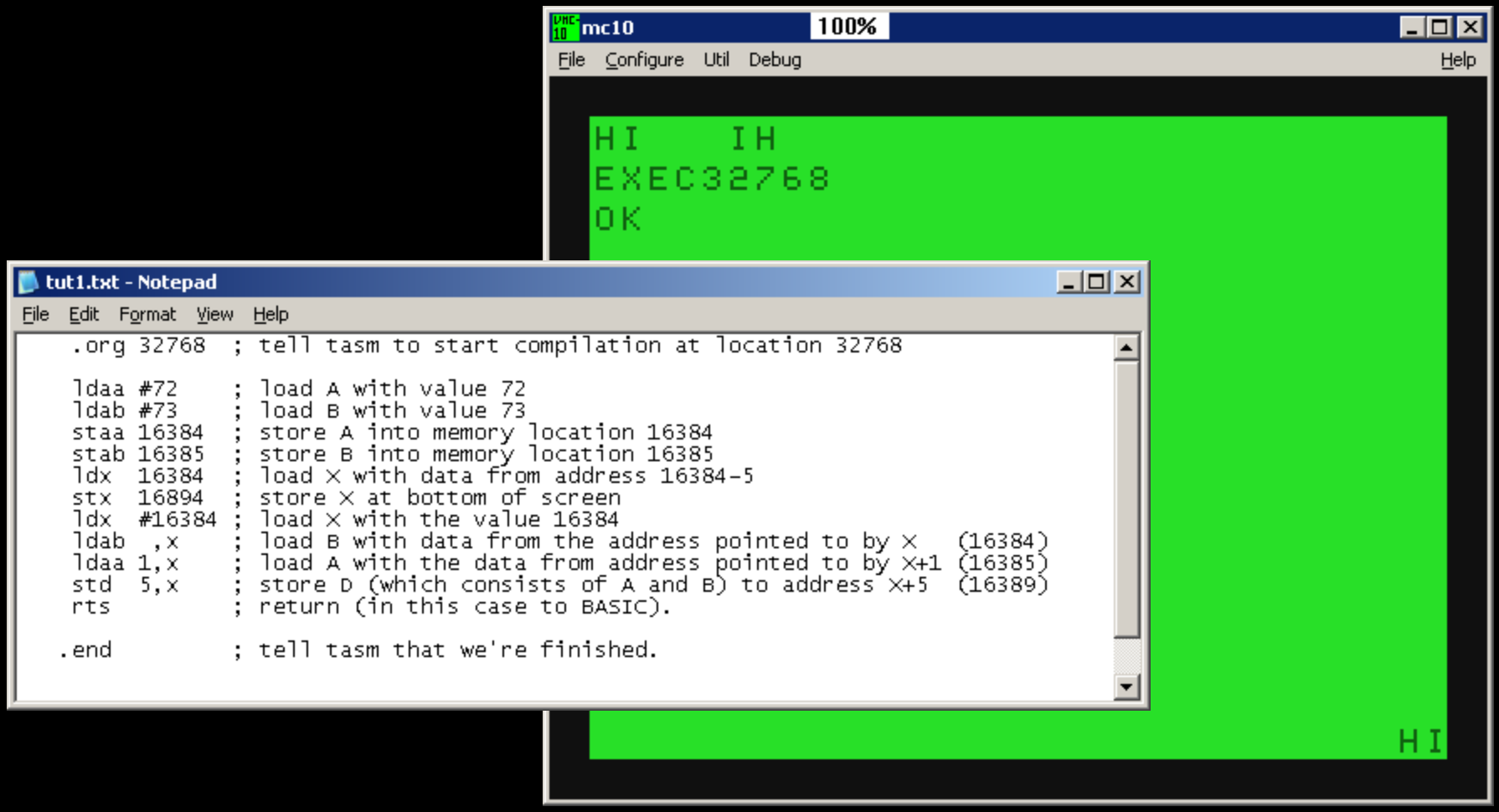


Now run your program

Ta-da!


mc10
100%
File Configure Util Debug Help
HI IH
EXEC32768
OK
HI

Here's the program and its output.



Using the debugger

- The Virtual MC-10's debugger has several windows:
  - Register
  - Memory
  - History
  - Disassembly
  - Breaks
  - LST (program listing)
  - Script
  - Map/Level

Using the debugger

- The Virtual MC-10's debugger has several windows:
  - Register
  - Memory
  - History
  - Disassembly
  - Breaks
  - LST (program listing)
  - Script
  - Map/Level

Tutorial #2
Increment and Decrement

aba	beq	bra	daa	jmp	mul	rol	~~staa~~	tst
abx	bne	brn	dec	jsr		rola	~~stab~~	tsta
adca	bcc		deca		neg	rolb	~~std~~	tstb
adcb	bcs	cba	decb	~~ldaa~~	nega	ror	~~sts~~	
adda	bhi	clc	des	~~ldab~~	negb	rora	~~stx~~	wai
addb	bhs	cli	dex	~~ldd~~	nop	rorb	suba	
addd	blo	clr		~~lds~~		rti	subb	
anda	bls	clra	eora	~~ldx~~	oraa	rts	subd	
andb	bge	clrb	eorb	lsl	orab		swi	
asr	bgt	clv		lsla		sba		
asra	ble	cmpa	inc	lslb	psha	sbca	tab	
asrb	blt	cmpb	inca	lsld	pshb	sbcb	tap	
	bmi	cpx	incb	lsr	pshx	sec	tba	
bita	bpl	com	ins	lsra	pula	sei	tpa	
bitb	bvc	coma	inx	lsrb	pulb	sev	tsx	
bsr	bvs	comb		lsrd	pulx		txs	

Increment instructions

inca – increment register a (a = a+1)

incb – increment register b (b = b+1)

inx – increment register x (x = x+1)

ins – increment register s (s = s+1)

inc address – increment contents of specified 2 byte address

inc offset,x – increment contents of address pointed to by adding X and the single-byte offset

Decrement instructions

deca – decrement register a (a = a - 1)

decb – decrement register b (b = b - 1)

dex – decrement register x (x = x - 1)

des – decrement register s (s = s - 1)

dec address – decrement contents of the specified 2 byte address

dec offset,x – decrement contents of address pointed to by adding x and the single-byte offset

Rollover

- Incrementing a byte that contains the highest value (255) will roll it over to 0.
- Decrementing a byte that contains the lowest value (0) will roll it back to 255.
- Two-byte words (x, s) will “roll over” between the values of 0 and 65535

Example program

tut2.txt - Notepad

File Edit Format View Help

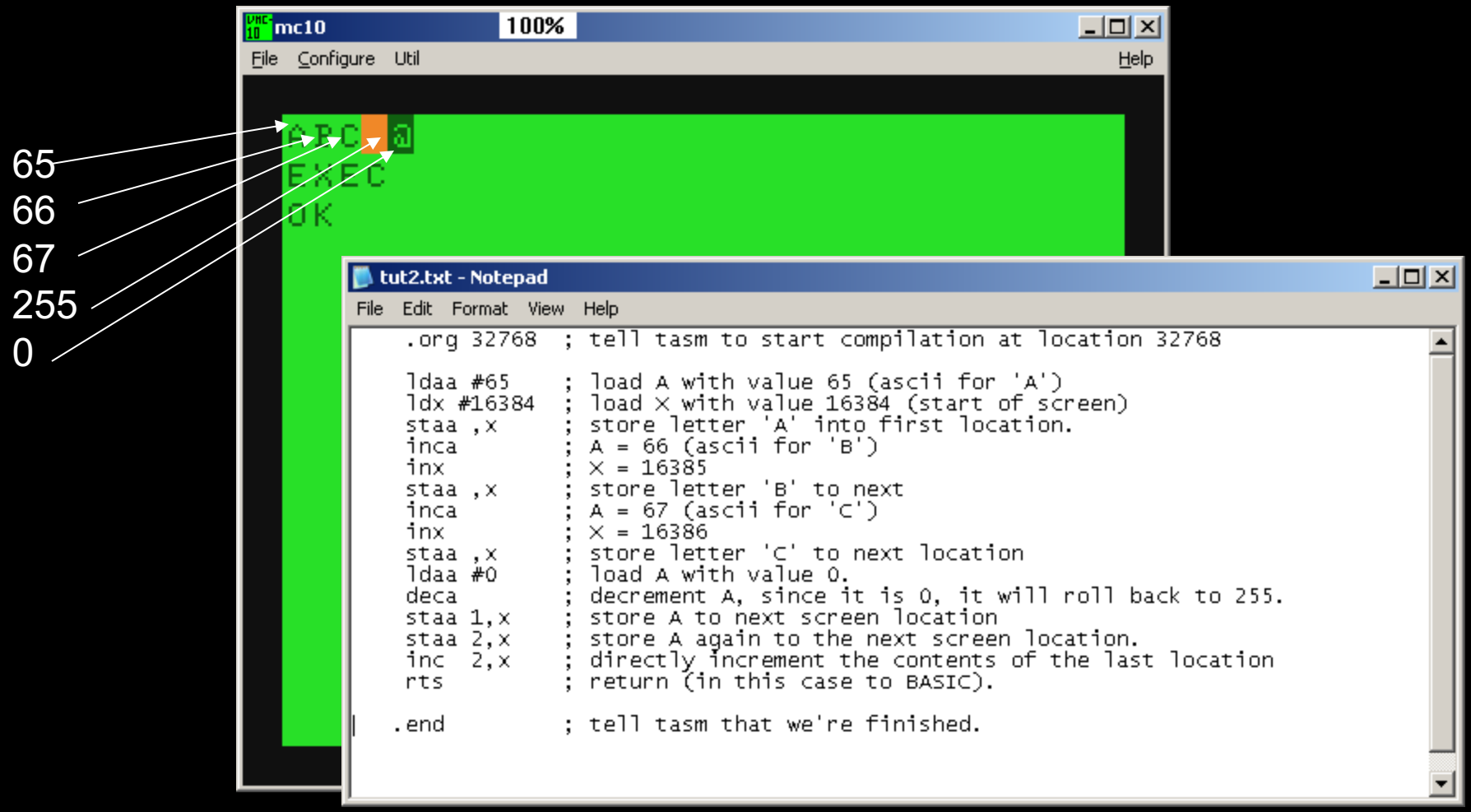
```
    .org 32768  ; tell tasm to start compilation at location 32768

    ldaa #65    ; load A with value 65 (ascii for 'A')
    ldx #16384  ; load X with value 16384 (start of screen)
    staa ,x     ; store letter 'A' into first location.
    inca        ; A = 66 (ascii for 'B')
    inx         ; X = 16385
    staa ,x     ; store letter 'B' to next
    inca        ; A = 67 (ascii for 'C')
    inx         ; X = 16386
    staa ,x     ; store letter 'C' to next location
    ldaa #0     ; load A with value 0.
    deca        ; decrement A, since it is 0, it will roll back to 255.
    staa 1,x    ; store A to next screen location
    staa 2,x    ; store A again to the next screen location.
    inc  2,x    ; directly increment the contents of the last location
    rts         ; return (in this case to BASIC).

    .end        ; tell tasm that we're finished.
```

Command Prompt

```
C:\greg\Personal\Arcade\MC10\VIRTUA~1\tools\tasm31>tasm -68 -b -x3 tut2.txt
TASM 6800-6811 Assembler  Version 3.1  February, 1998.
 Copyright (C) 1998 Squak Valley Software
tasm: pass 1 complete.
tasm: pass 2 complete.
tasm: Number of errors = 0

C:\greg\Personal\Arcade\MC10\VIRTUA~1\tools\tasm31>_
```

Output

Output


65
66
67
255
0
mc10
100%
File Configure Util
Help
ABC@
EXEC
OK
tut2.txt - Notepad
File Edit Format View Help
.org 32768 ; tell tasm to start compilation at location 32768
ldaa #65 ; load A with value 65 (ascii for 'A')
ldx #16384 ; load X with value 16384 (start of screen)
staa ,x ; store letter 'A' into first location.
inca ; A = 66 (ascii for 'B')
inx ; X = 16385
staa ,x ; store letter 'B' to next
inca ; A = 67 (ascii for 'C')
inx ; X = 16386
staa ,x ; store letter 'C' to next location
ldaa #0 ; load A with value 0.
deca ; decrement A, since it is 0, it will roll back to 255.
staa 1,x ; store A to next screen location
staa 2,x ; store A again to the next screen location.
inc 2,x ; directly increment the contents of the last location
rts ; return (in this case to BASIC).
.end ; tell tasm that we're finished.

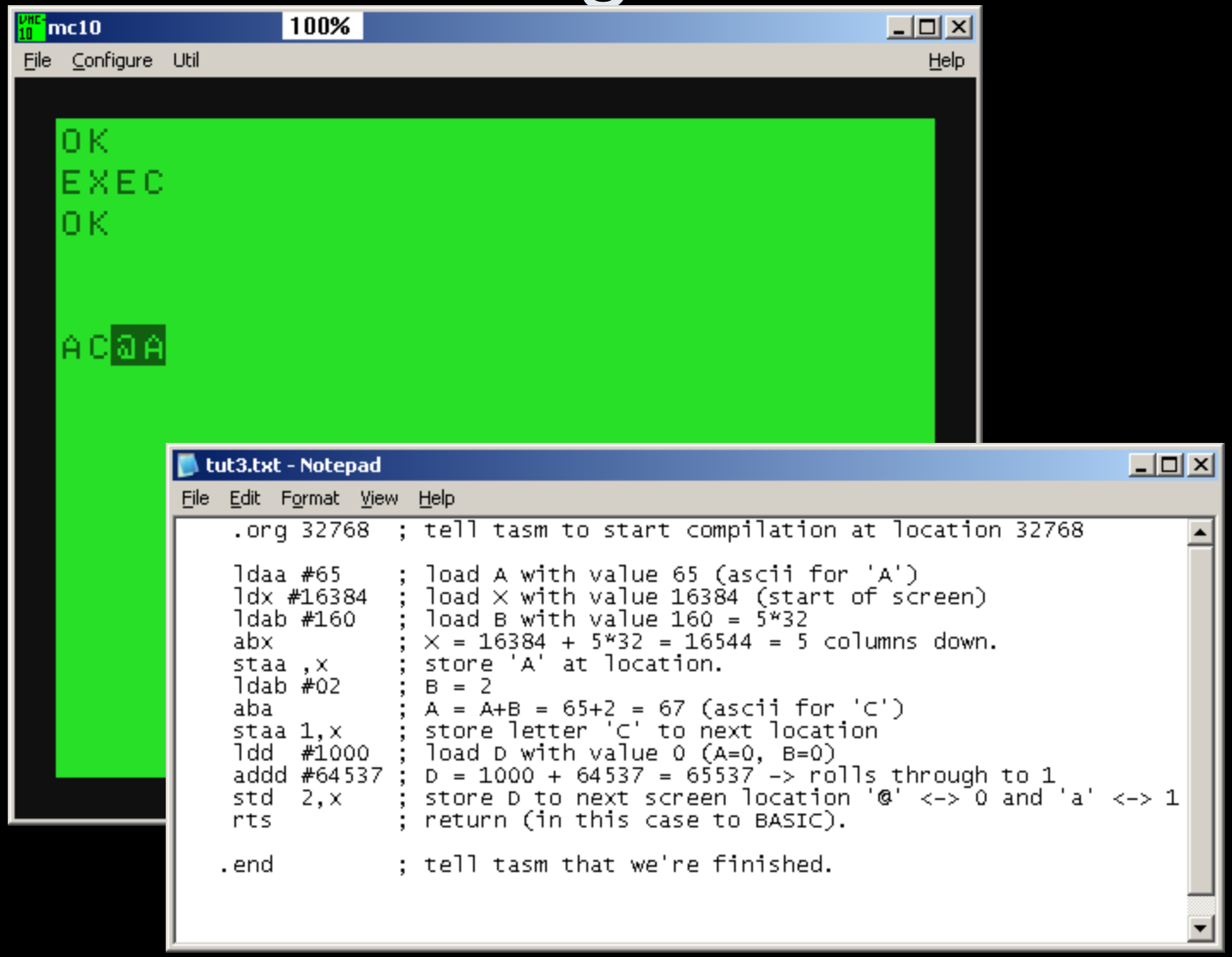
Tutorial #3 and #4 Addition

aba	beq	bra	daa	jmp	mul	rol	~~staa~~	tst
abx	bne	brn	~~dec~~	jsr		rola	~~stab~~	tsta
adca	bcc		~~deca~~		neg	rolb	~~std~~	tstb
adcb	bcs	cba	~~decb~~	~~ldaa~~	nega	ror	~~sts~~	
adda	bhi	clc	~~des~~	~~ldab~~	negb	rora	~~stx~~	wai
addb	bhs	cli	~~dex~~	~~ldd~~	nop	rorb	suba	
addd	blo	clr		~~lds~~		rti	subb	
anda	bls	clra	eora	~~ldx~~	oraa	rts	subd	
andb	bge	clrb	eorb	lsl	orab		swi	
asr	bgt	clv		lsla		sba		
asra	ble	cmpa	~~inc~~	lslb	psha	sbca	tab	
asrb	blt	cmpb	~~inca~~	lsld	pshb	sbcb	tap	
	bmi	cpx	~~incb~~	lsr	pshx	sec	tba	
bita	bpl	com	~~ins~~	lsra	pula	sei	tpa	
bitb	bvc	coma	~~inx~~	lsrb	pulb	sev	tsx	
bsr	bvs	comb		lsrd	pulx		txs	

Unsigned Addition

- Addition shares the same “rollover” properties as incrementing.
- You may add values to the A, B, or D accumulators (adda, addb, addd)
- You can add to the contents of the A register by the contents of the B (aba).
- You can add the B register (unsigned) to the contents of the X register (abx)

Tutorial #3 – unsigned addition

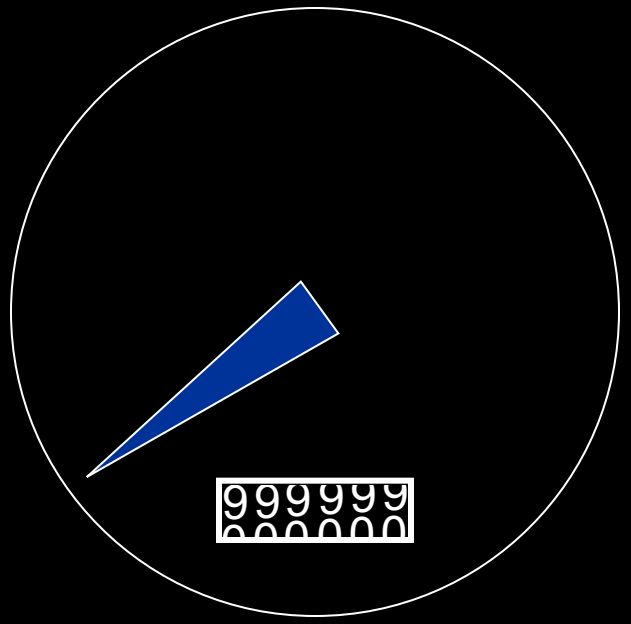


Tutorial #4 Signed Addition

- The “rollover” property can be used to construct negative numbers.
- We will first discuss an odometer which rolls over at 1000000 miles.
- We’ll then apply this to our 8-bit and 16-bit accumulators which rollover at 256 and 65536.

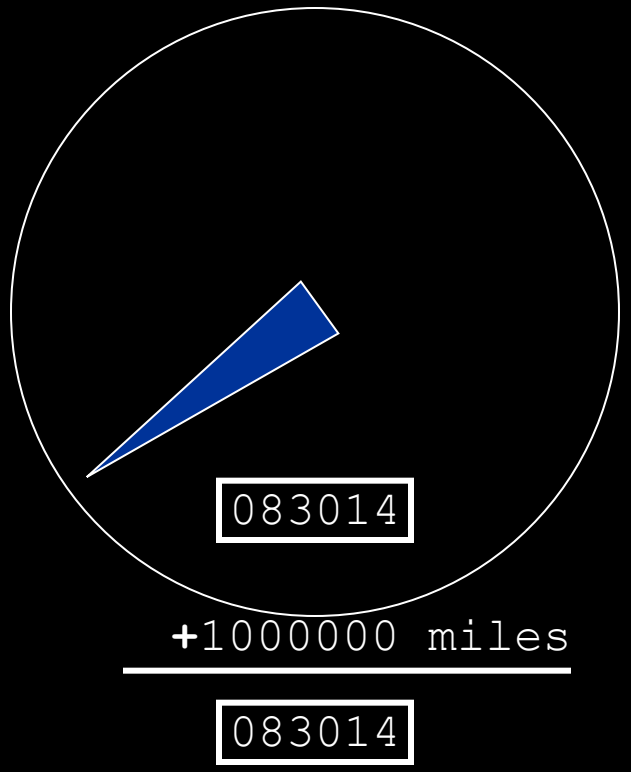
Odometer Example

- Consider an old fashioned odometer that goes from 000000 to 999999.
- If you drive exactly 1000000, 2000000, 3000000 or any multiple of 1000000 miles, the odometer will go back to 000000.



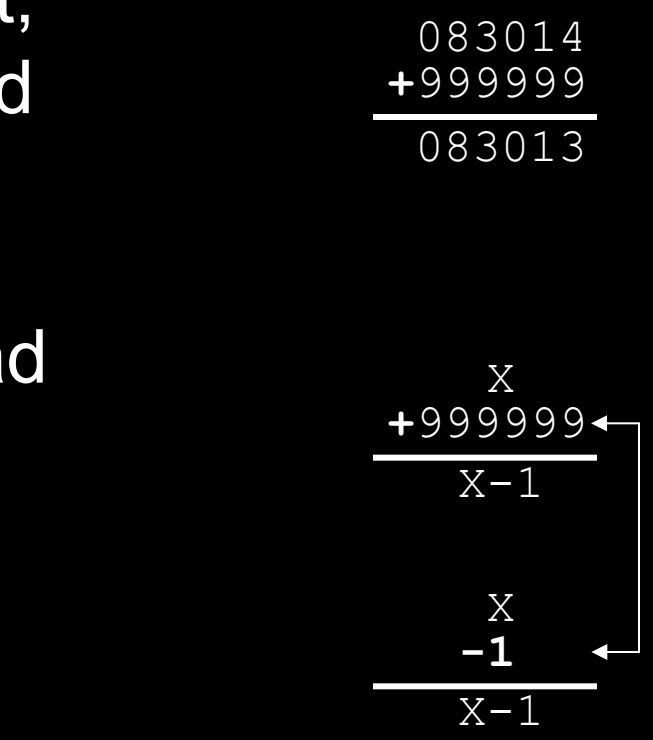
Odometer Example

- If your odometer is at, say, 83014 miles, and you drive exactly 1000000 miles, then the odometer will still read 83014.



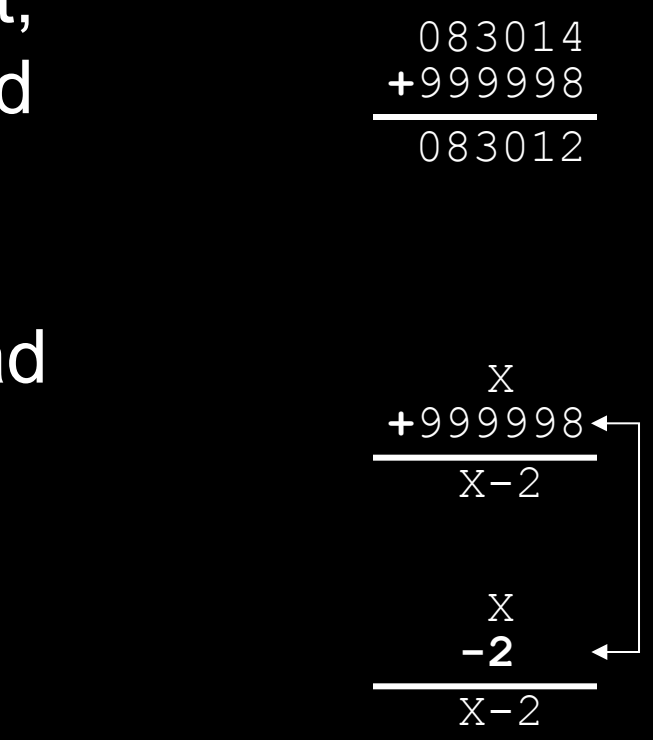
Odometer Example

- If your odometer is at, say, 83014 miles, and you drive exactly 999999 miles, then the odometer will read read 83013.
- Note that adding 999999 is the same as adding -1.



Odometer Example

- If your odometer is at, say, 83014 miles, and you drive exactly 999998 miles, then the odometer will read read 83012.
- Note that adding 999998 is the same as adding -2.

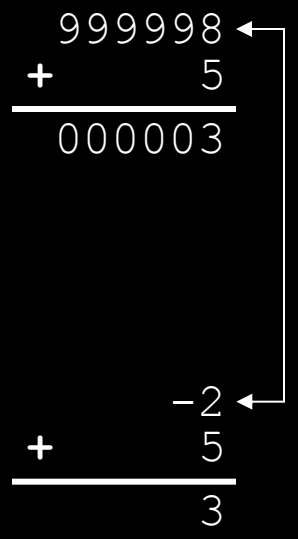


Odometer Example

- This suggests the following relationship:
  - $X = 1000000 - X$
- Thus if your “odometer” only has positive numbers, you can use the “rollover” property to find the equivalent negative number.

Odometer Example

- Let's say your odometer is at 999998 miles. If you drive 5 miles, you'll end up with a reading of 000003 miles.
- Note the equivalency again of 999998 = -2



8-bit Odometer

- A single byte roll-over occurs at 256.
- Thus,
  -X = 256 – X (single-byte)
- By convention, numbers 0, 1, 2, …,127 are considered “positive”
- Numbers 128, 129, 130, …, 253, 254, 255 are considered “negative” and correspond to values -128, -127, -126, …, -3, -2, -1.
- Note that +128 cannot be represented with this convention, but -128 can.

8-bit Negation Instructions

- You can negate the value of either the A or B register (nega, negb)
- You can negate an address

```
ldaa #10   ; A holds 10
nega       ; A holds -10 (246 = 256-10)
neg 16384 ;negate first screen char
neg ,X     ; negate what X points to
```

16-bit Odometer

- A double byte roll-over occurs at 65536.
- Thus,
  -X = 65536 – X (double-byte)
- By convention, numbers 0, 1, 2, …, 32767 are considered “positive”
- Numbers 32768, 32769, 32770, …, 65533, 65534, 65535 are considered “negative” and correspond to values -32768, -32767, -32766, …, -3, -2, -1.
- Note that -32768 is in this set, but that +32768 cannot be.
- Sadly, there is no 16-bit negation instruction

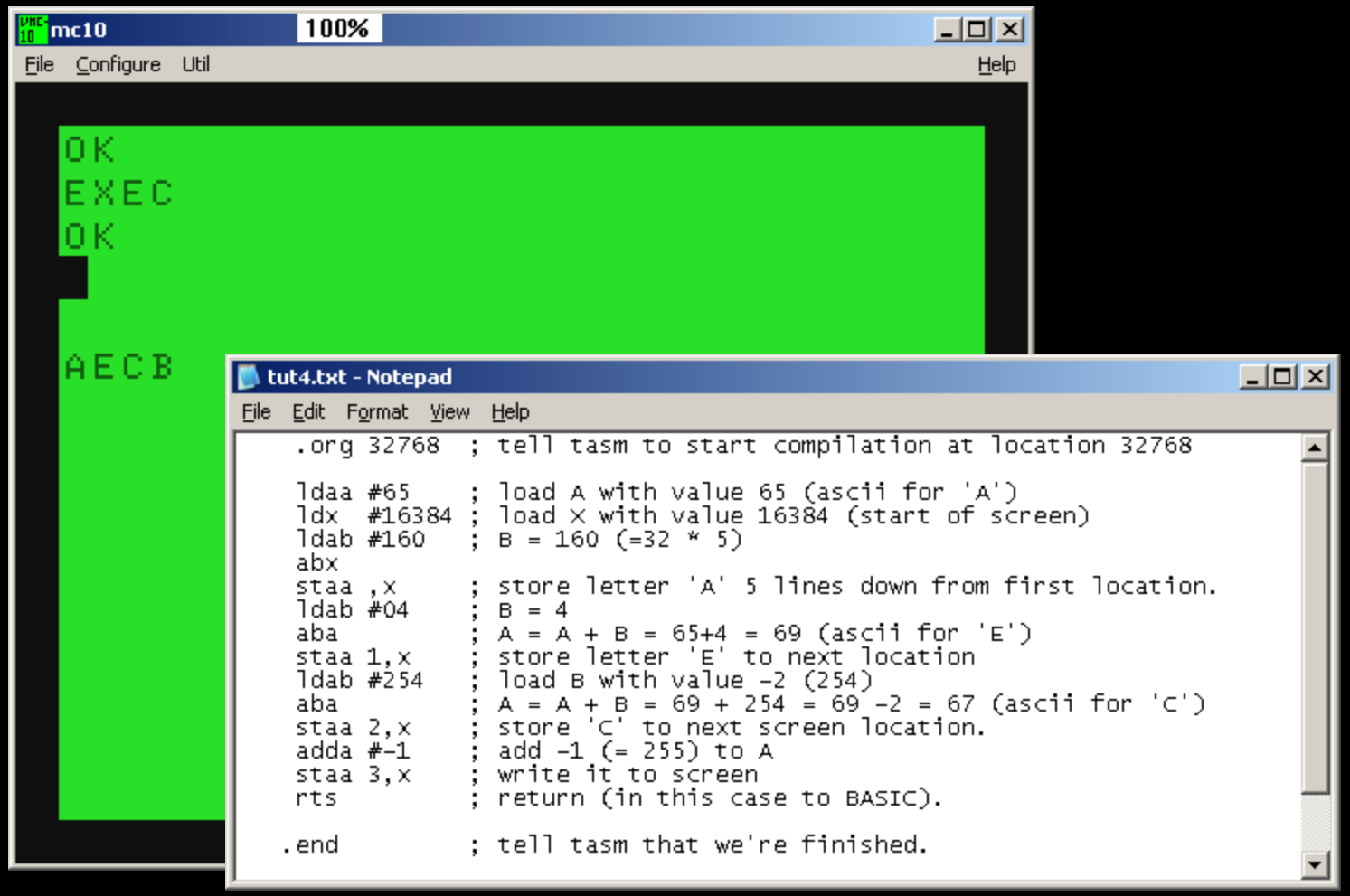
Signed arithmetic

Like driving the car, addition and subtraction with rollover don't really care if you consider your numbers as signed or unsigned. The computer will blindly increment or decrement its "internal odometer" the specified number of times, and leave you with the result, which you can use either as signed or unsigned depending on your needs.

Negative Operands

- The TASM compiler will gladly accept negative numbers to instructions that take immediate operands.
- It will automatically convert them to their ‘unsigned’ equivalents depending on whether the particular instruction expects a 8-bit or 16-bit operand.
    - ldaa #-1 -> ldaa #255
    - addd #-3 -> addd #65533

Example #4 - Signed Addition



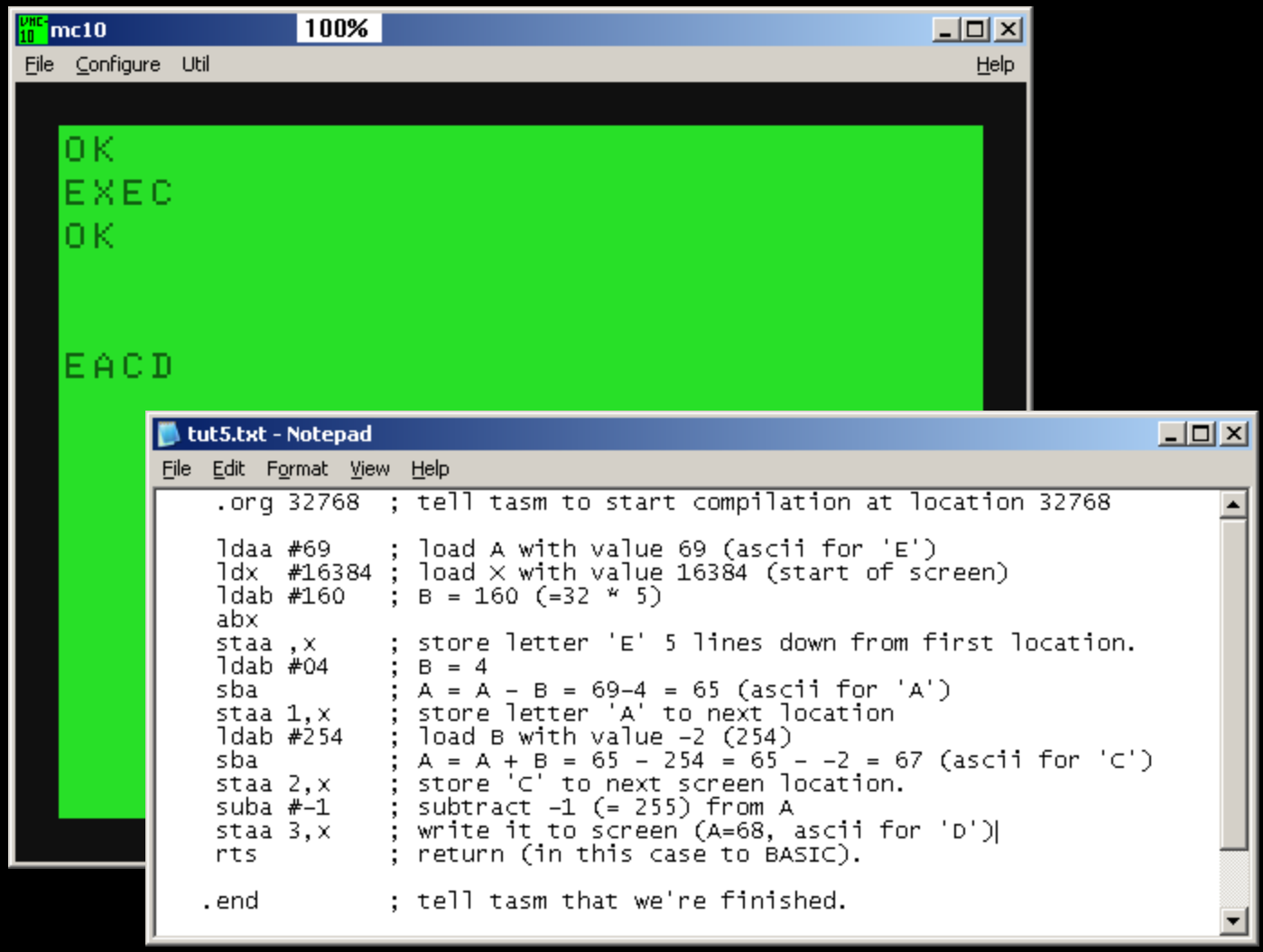
Tutorial #5
Subtraction

~~aba~~	beq	bra	daa	jmp	mul	rol	~~staa~~	tst
~~abx~~	bne	brn	~~dec~~	jsr		rola	~~stab~~	tsta
adca	bcc		~~deca~~		~~neg~~	rolb	~~std~~	tstb
adcb	bcs	cba	~~decb~~	~~ldaa~~	~~nega~~	ror	~~sts~~	
~~adda~~	bhi	clc	~~des~~	~~ldab~~	~~negb~~	rora	~~stx~~	wai
~~addb~~	bhs	cli	~~dex~~	~~ldd~~	nop	rorb	suba	
~~addd~~	blo	clr		~~lds~~		rti	subb	
anda	bls	clra	eora	~~ldx~~	oraa	rts	subd	
andb	bge	clrb	eorb	lsl	orab		swi	
asr	bgt	clv		lsla		sba		
asra	ble	cmpa	~~inc~~	lslb	psha	sbca	tab	
asrb	blt	cmpb	~~inca~~	lsld	pshb	sbcb	tap	
	bmi	cpx	~~incb~~	lsr	pshx	sec	tba	
bita	bpl	com	~~ins~~	lsra	pula	sei	tpa	
bitb	bvc	coma	~~inx~~	lsrb	pulb	sev	tsx	
bsr	bvs	comb		lsrd	pulx		txs	

Subtraction

- Subtraction shares the same “rollover” properties as decrementing.
- You may subtract
  - values from the A, B, or D accumulators (suba, subb, subd)
  - the contents of the A register by the contents of the B. (sba)
- Unfortunately, there is no corresponding sbx instruction that subtracts the B register from the X.

Example #5
Subtraction



Tutorial #6
Branch Instructions

~~aba~~	beq	bra	daa	jmp	mul	rol	~~staa~~	tst
~~abx~~	bne	brn	~~dec~~	jsr		rola	~~stab~~	tsta
adca	bcc		~~deca~~		neg	rolb	~~std~~	tstb
adcb	bcs	cba	~~decb~~	~~ldaa~~	nega	ror	~~sts~~	
~~adda~~	bhi	clc	~~des~~	~~ldab~~	negb	rora	~~stx~~	wai
~~addb~~	bhs	cli	~~dex~~	~~ldd~~	nop	rorb	~~suba~~	
~~addd~~	blo	clr		~~lds~~		rti	~~subb~~	
anda	bls	clra	eora	~~ldx~~	oraa	rts	~~subd~~	
andb	bge	clrb	eorb	lsl	orab		swi	
asr	bgt	clv		lsla		~~sba~~		
asra	ble	cmpa	~~inc~~	lslb	psha	sbca	tab	
asrb	blt	cmpb	~~inca~~	lsld	pshb	sbcb	tap	
	bmi	cpx	~~incb~~	lsr	pshx	sec	tba	
bita	bpl	com	~~ins~~	lsra	pula	sei	tpa	
bitb	bvc	coma	~~inx~~	lsrb	pulb	sev	tsx	
bsr	bvs	comb		lsrd	pulx		txs	

Condition Codes for Addition and Subtraction

- The “Z” bit is set when the result is zero
- The “N” bit is set when the result is “negative”
- The “C” bit is set when the operation rolls through zero.
- The “V” bit is set when the operation rolls through 32768.

Comparison

- Comparison sets condition codes exactly like subtraction, except that the result is discarded and not stored back in the registers.
- You may compare
  - the values from the A or B accumulators (cmpa, cmpb)
  - the contents of the A register to the contents of the B (cba)
  - The value of the X register (cpx)
- Unfortunately, you can’t compare the value of the D register (but you may subtract it).

Branch Instructions

- Usually performed after subtraction or comparing two numbers
- Branches can reach only a small distance from the location of the next instruction (up to 127 bytes ahead or 128 bytes behind)
- Larger branches require an explicit, unconditional, jump to memory (jmp)

Generic Branch Opcodes

- bra – branch always
- brn – branch never*
- beq – branch if equal to zero (Z=1)
- bne – branch if not equal to zero (Z=0)
- bmi – branch if minus (N=1)
- bpl – branch if plus or zero (N=0)
- bcs – branch if carry set (C=1)
- bcc – branch if carry cleared (C=0)
- bvs – branch if overflow set (V=1)
- bvc – branch if overflow cleared (V=0)

*brn is usually used as padding or a timewaster

Unsigned Branch Opcodes (used after subtraction/comparison)

- blo – branch if lower
- bls – branch if lower or same
- bhi – branch if higher
- bhs – branch if higher or same

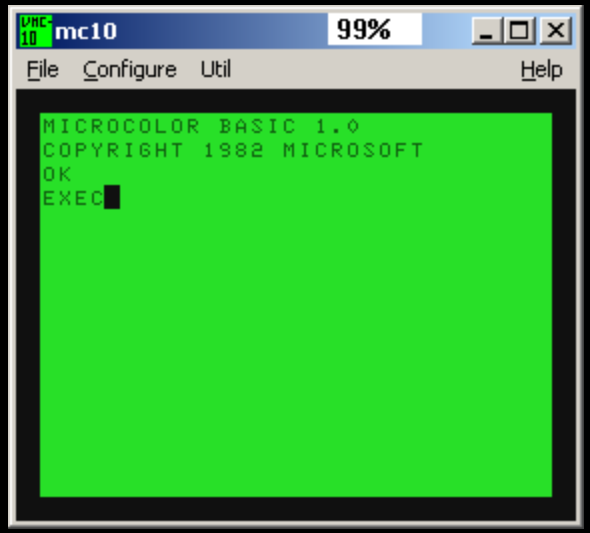
- beq – branch if equal
- bne – branch if not equal

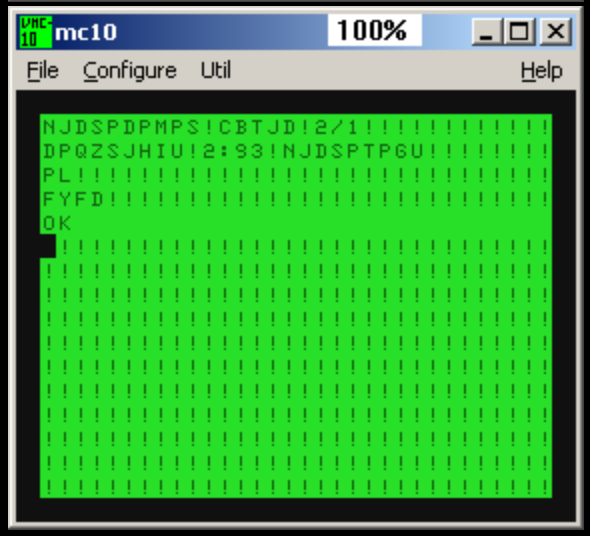
Signed Branch Opcodes (used after subtraction/comparison)

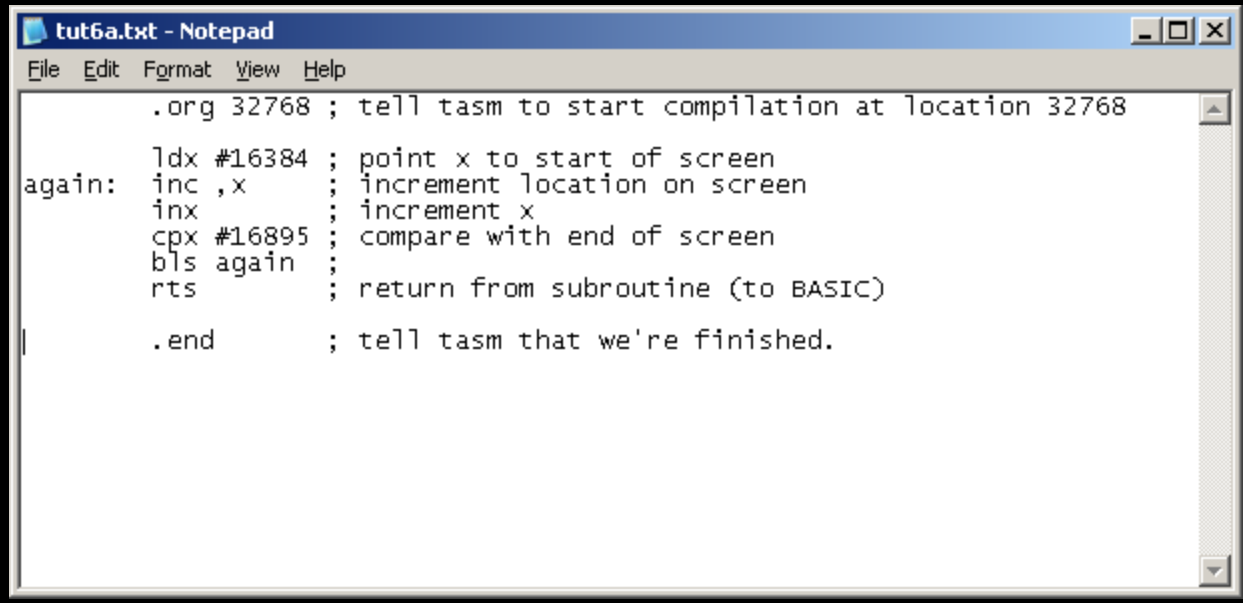
- blt – branch if less than
- ble – branch if less than or equal to
- bgt – branch if greater than
- bge – branch if less than or equal to

- beq – branch if equal
- bne – branch if not equal

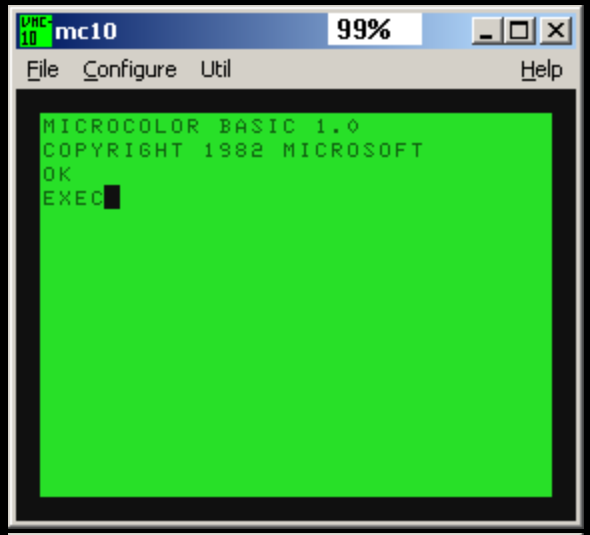
Example 6a
Increment all values on screen







Example 6b
screen bubble sort

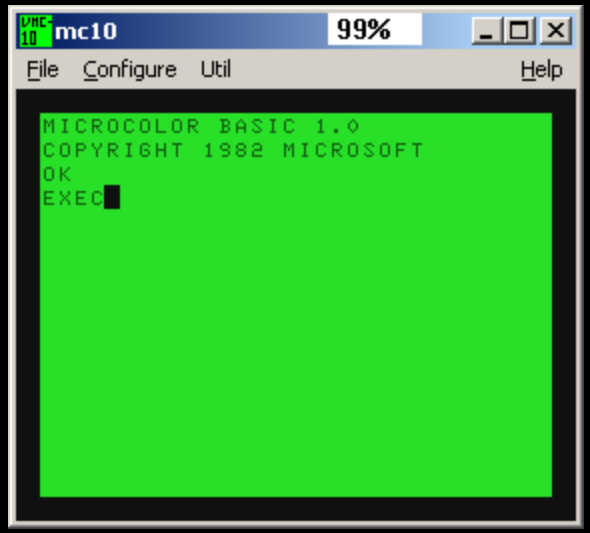


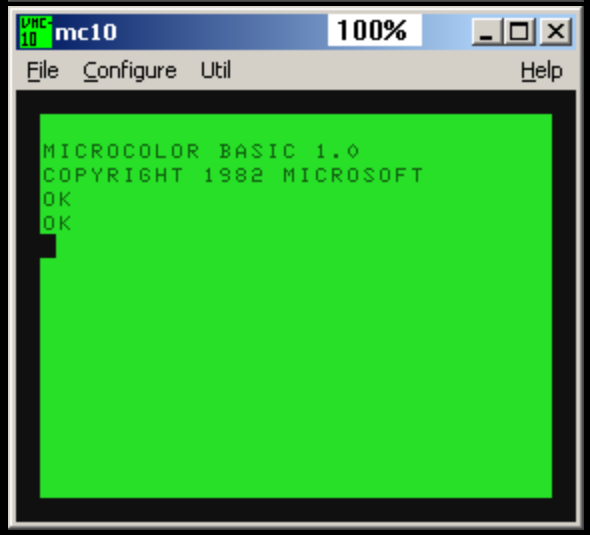
tut6b.txt - Notepad

File Edit Format View Help

```
        .org 32768  ; tell tasm to start compilation at location 32768

again:  ldx  #16384 ; point x to start of screen
        ldab #1     ; set b flag to true
nxtpos: ldaa  ,x    ; get screen data
        cmpa 1,x    ; compare with next screen data
        bls noswap  ; already less than or equal -- don't swap
        ldab 1,x    ; swap contents of x and x+1
        staa 1,x    ;
        stab  ,x    ;
        ldab #0     ; set b to zero if we made a swap
noswap: inx         ; increment screen pointer
        cpx  #16894 ; compare with next to last screen position
        bls  nxtpos ; go to nextpos if not yet reached
        cmpb #0     ; is b zero?
        beq  again  ; yes, try again until we're completely sorted
        rts         ; return from subroutine (to BASIC)

        .end        ; tell tasm that we're finished.
```

Example 6c
Reverse Scroll





tut6c.txt - Notepad

File Edit Format View Help

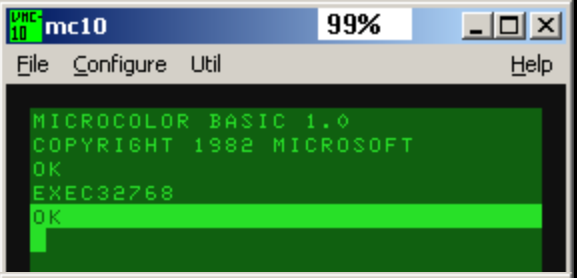
```
        .org 32768  ; tell tasm to start compilation at location 32768

        ldx  #16864 ; point x to one position after row above last cell
next:   dex
        ldaa   ,x   ; get screen data
        staa 32,x   ; store one row down
        cpx  #16384 ; compare with first screen position
        bhi  next   ; go if bigger
        ldaa #96    ; load space character
again   staa ,x     ; store on screen
        inx
        cpx  #16416 ; compare with second row, first column
        blo  again  ; go until we're there
        rts         ; return from subroutine (to BASIC)

        .end        ; tell tasm that we're finished.
```

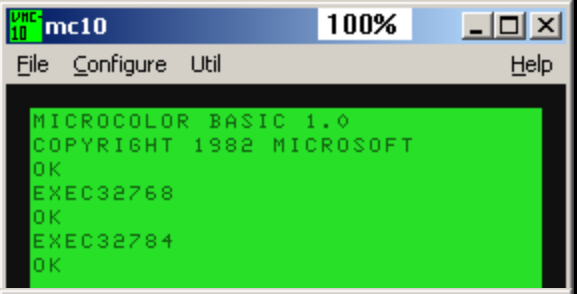
Tutorial #7
Mask Instructions

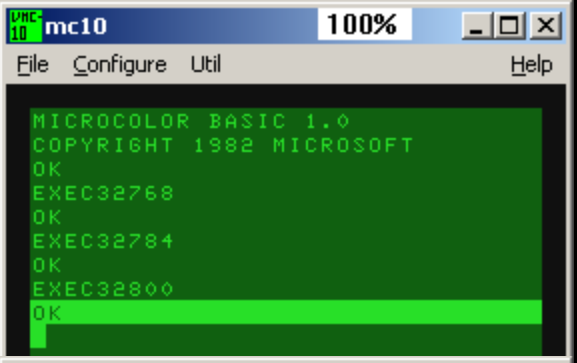
~~aba~~	~~beq~~	~~bra~~	daa	~~jmp~~	mul	rol	~~staa~~	tst
~~abx~~	~~bne~~	~~brn~~	~~dec~~	jsr		rola	~~stab~~	tsta
adca	~~bcc~~		~~deca~~		~~neg~~	rolb	~~std~~	tstb
adcb	~~bcs~~	~~cba~~	~~decb~~	~~ldaa~~	~~nega~~	ror	~~sts~~	
~~adda~~	~~bhi~~	clc	~~des~~	~~ldab~~	~~negb~~	rora	~~stx~~	wai
~~addb~~	~~bhs~~	cli	~~dex~~	~~ldd~~	nop	rorb	~~suba~~	
~~addd~~	~~blo~~	clr		~~lds~~		rti	~~subb~~	
anda	~~bls~~	clra	eora	~~ldx~~	oraa	rts	~~subd~~	
andb	~~bge~~	clrb	eorb	lsl	orab		swi	
asr	~~bgt~~	clv		lsla		~~sba~~		
asra	~~ble~~	~~cmpa~~	~~inc~~	lslb	psha	sbca	tab	
asrb	~~blt~~	~~cmpb~~	~~inca~~	lsld	pshb	sbcb	tap	
	~~bmi~~	~~cpx~~	~~incb~~	lsr	pshx	sec	tba	
bita	~~bpl~~	com	~~ins~~	lsra	pula	sei	tpa	
bitb	~~bvc~~	coma	~~inx~~	lsrb	pulb	sev	tsx	
bsr	~~bvs~~	comb		lsrd	pulx		txs	

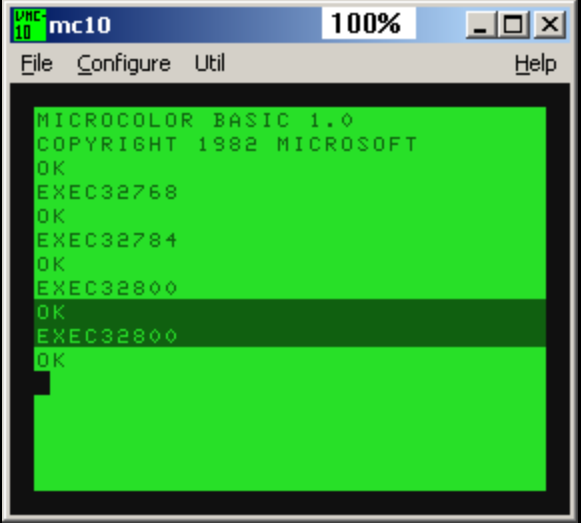
And, Or, Eor

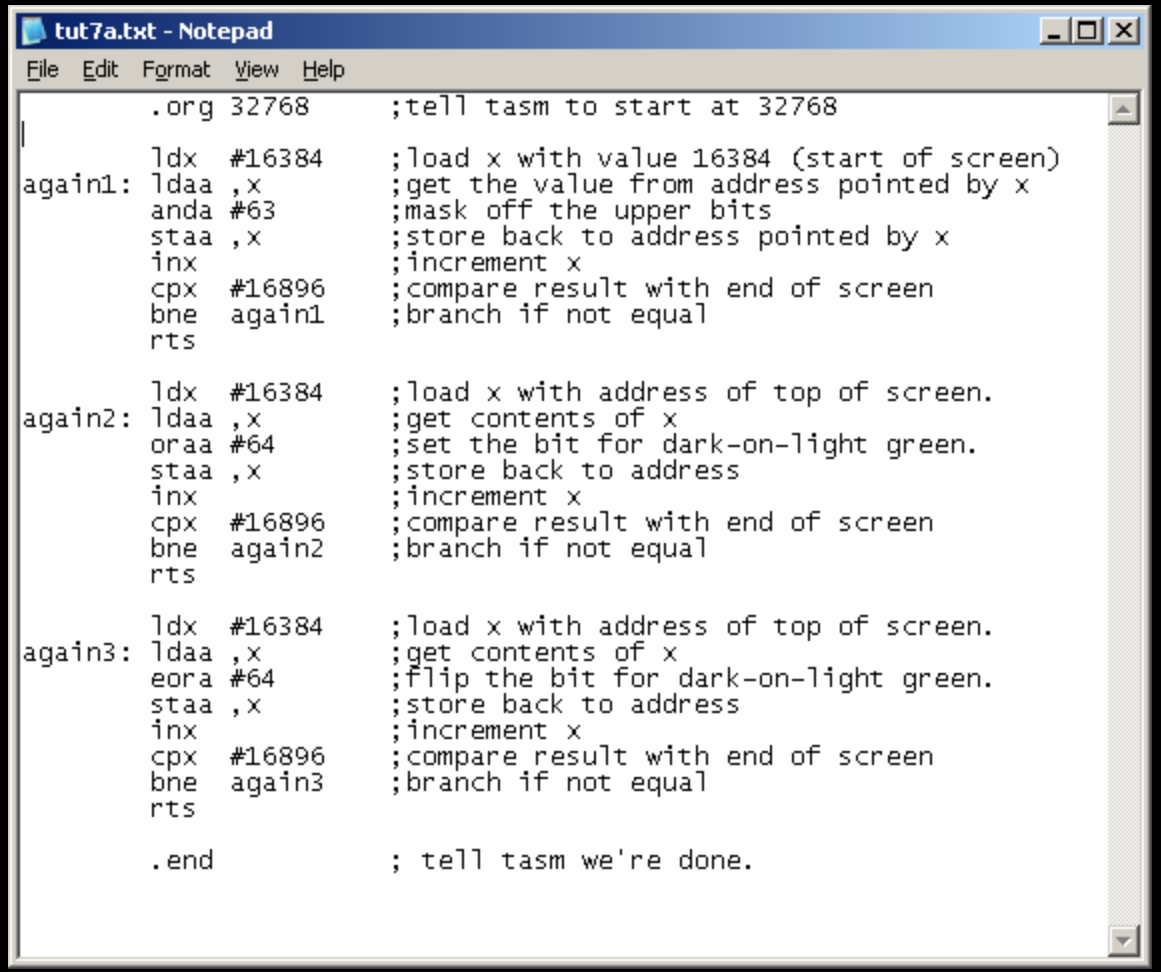
- These are bitwise operations with the usual properties:
  - And with 0 -> 0.
  - Or with 1 -> 1.
  - Eor with 1 -> flip the value.
  - Otherwise, leave value unchanged.
- They work on either the A or B register (anda, andb, oraa, orab, eora, eorb).


mc10
99%
File Configure Util Help
MICROCOLOR BASIC 1.0
COPYRIGHT 1982 MICROSOFT
OK
EXEC32768
OK


mc10
100%
File Configure Util Help
MICROCOLOR BASIC 1.0
COPYRIGHT 1982 MICROSOFT
OK
EXEC32768
OK
EXEC32784
OK

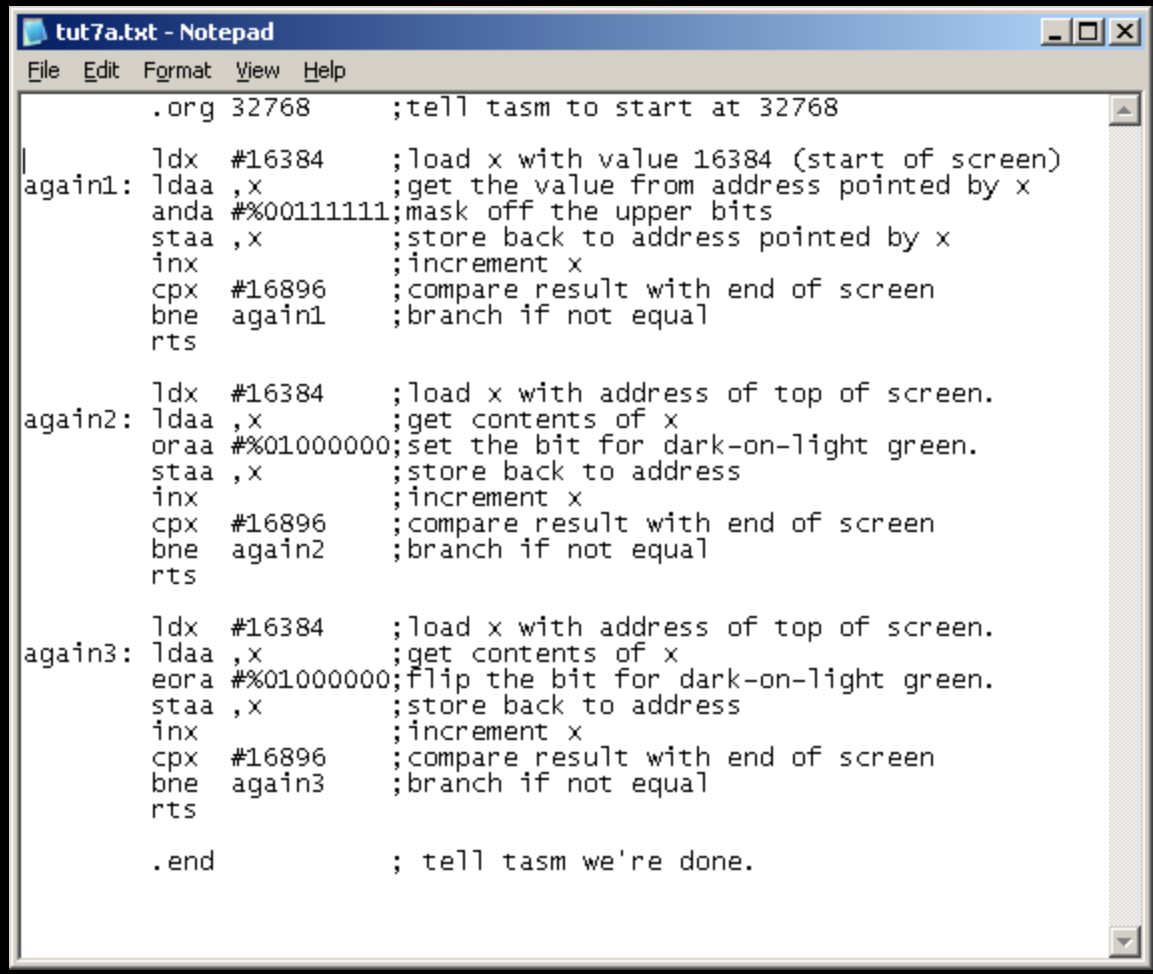

mc10
100%
File Configure Util Help
MICROCOLOR BASIC 1.0
COPYRIGHT 1982 MICROSOFT
OK
EXEC32768
OK
EXEC32784
OK
EXEC32800
OK


mc10
100%
File Configure Util Help
MICROCOLOR BASIC 1.0
COPYRIGHT 1982 MICROSOFT
OK
EXEC32768
OK
EXEC32784
OK
EXEC32800
OK
EXEC32800
OK


tut7a.txt - Notepad
File Edit Format View Help
.org 32768 ;tell tasm to start at 32768
ldx #16384 ;load x with value 16384 (start of screen)
again1: ldaa ,x ;get the value from address pointed by x
anda #63 ;mask off the upper bits
staa ,x ;store back to address pointed by x
inx ;increment x
cpx #16896 ;compare result with end of screen
bne again1 ;branch if not equal
rts
ldx #16384 ;load x with address of top of screen.
again2: ldaa ,x ;get contents of x
oraa #64 ;set the bit for dark-on-light green.
staa ,x ;store back to address
inx ;increment x
cpx #16896 ;compare result with end of screen
bne again2 ;branch if not equal
rts
ldx #16384 ;load x with address of top of screen.
again3: ldaa ,x ;get contents of x
eora #64 ;flip the bit for dark-on-light green.
staa ,x ;store back to address
inx ;increment x
cpx #16896 ;compare result with end of screen
bne again3 ;branch if not equal
rts
.end ; tell tasm we're done.

Binary constants and ‘%’.

- You can type in a binary number directly by using the ‘%’ prefix.



Bit

- Same as “anda” or an “andb” instruction except the result is discarded.
- The condition codes can be used to inspect if certain bits were set.
- Since the result is discarded, you can check the A or B register for various bits without needing to save the value.

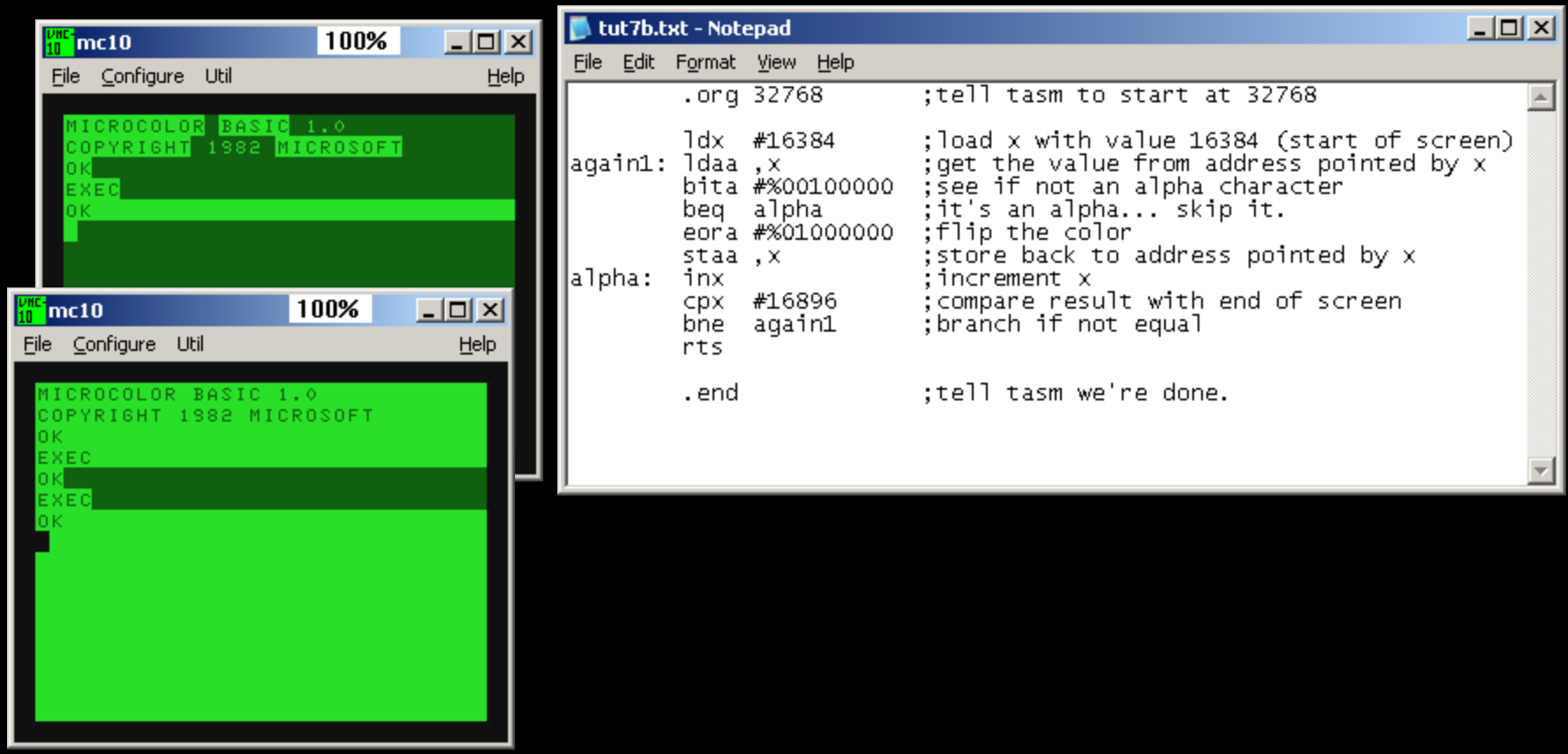
Bit Example

```
ldaa #%00110110    ; % is binary prefix
bita #%01000000    ; see if bit 6 is set.
bne bitwasset     ; go if bit 6 was set.
bita #%10001000    ; set if either bit 7 or 4 is set.
beq bothwerezero  ; go if both bits were zero
```

Tst

- Sets the N and Z condition codes depending on the data.
- You can test the A or B registers (tsta, tstb), or a value in memory (tst)

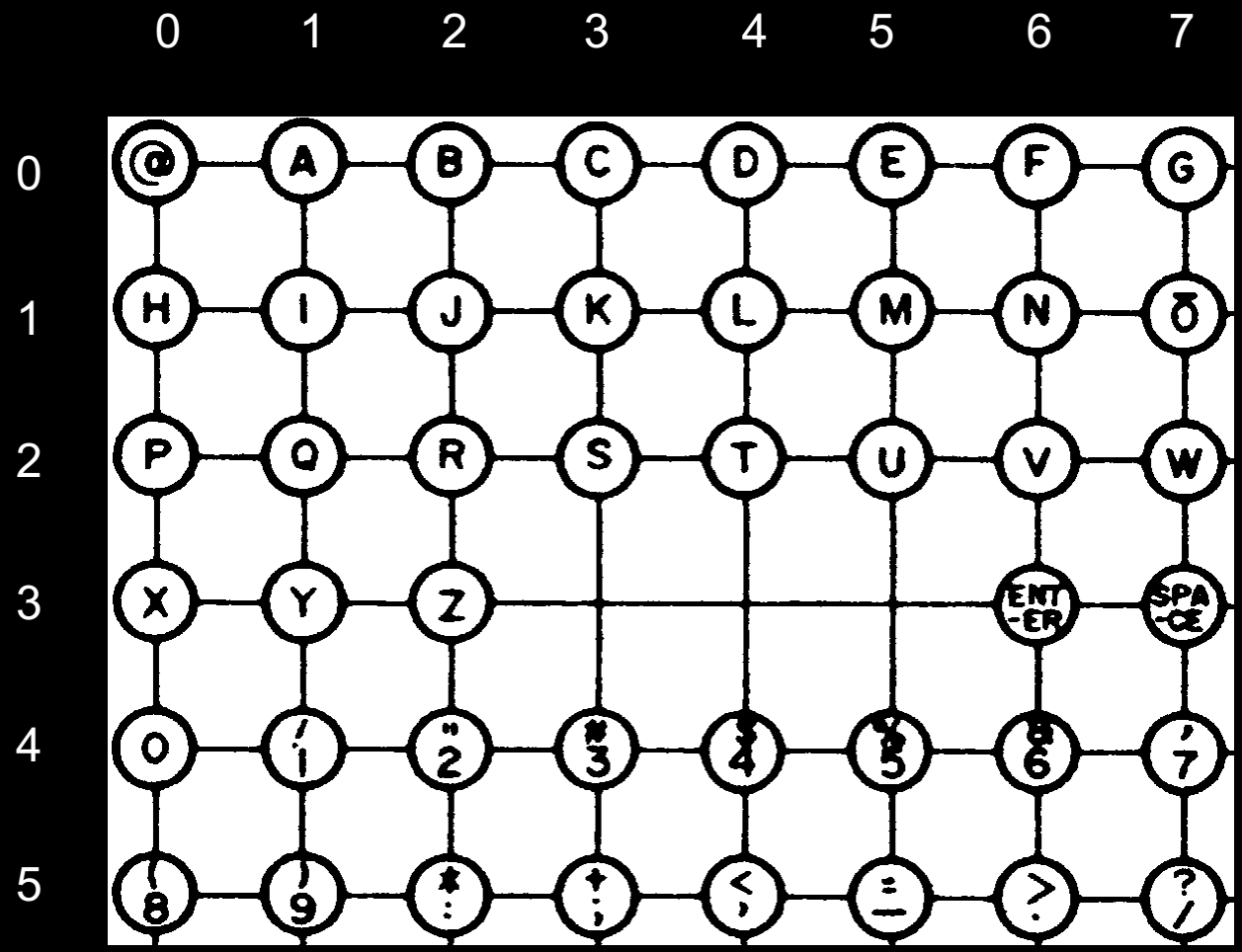
Bit example


mc10
100%
File Configure Util Help
MICROCOLOR BASIC 1.0
COPYRIGHT 1982 MICROSOFT
OK
EXEC
OK
mc10
100%
File Configure Util Help
MICROCOLOR BASIC 1.0
COPYRIGHT 1982 MICROSOFT
OK
EXEC
OK
EXEC
OK
tut7b.txt - Notepad
File Edit Format View Help
.org 32768 ;tell tasm to start at 32768
ldx #16384 ;load x with value 16384 (start of screen)
again1: ldaa ,x ;get the value from address pointed by x
bita #%00100000 ;see if not an alpha character
beq alpha ;it's an alpha... skip it.
eora #%01000000 ;flip the color
staa ,x ;store back to address pointed by x
alpha: inx ;increment x
cpx #16896 ;compare result with end of screen
bne again1 ;branch if not equal
rts
.end ;tell tasm we're done.

Reading the keyboard

- Reading the keyboard involves writing to location 2 and reading from either location 49151 or 3.
- The keys are grouped in sections of 8.
- You'll use logic-0 values to determine which key is pressed.

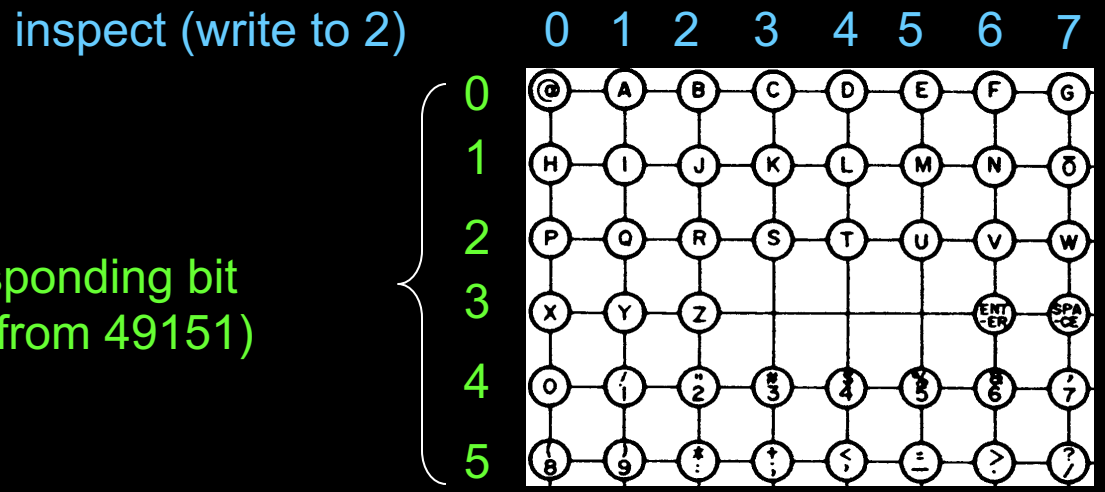
Keys associated with 49151



Example: is “L” pressed?

bits to inspect (write to 2)

corresponding bit
(read from 49151)

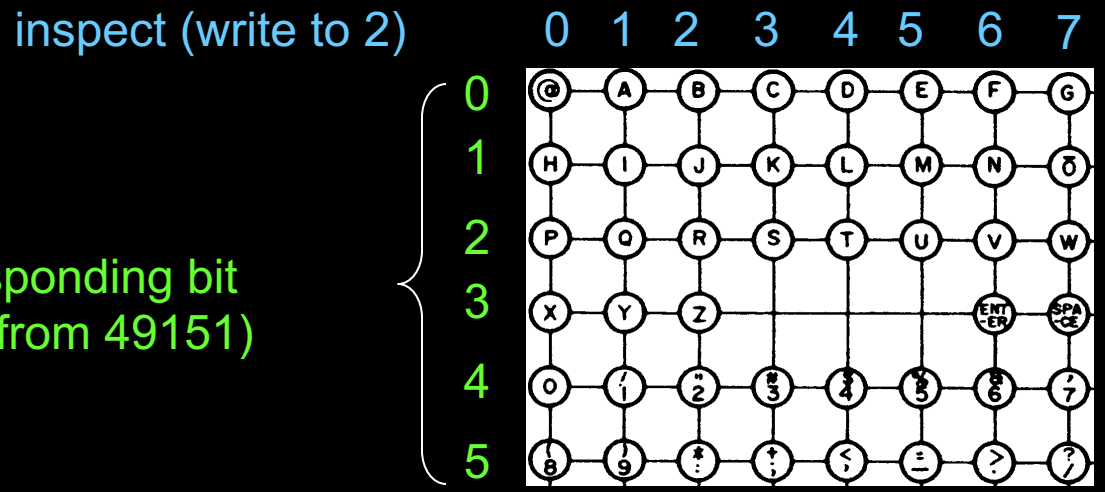



```
ldaa #%11101111 ;bit 4 corresponds to ‘L’
staa 2
ldaa 49151
bita #%00000010
beq  keypressed
```

Example: is “X” pressed?

bits to inspect (write to 2)

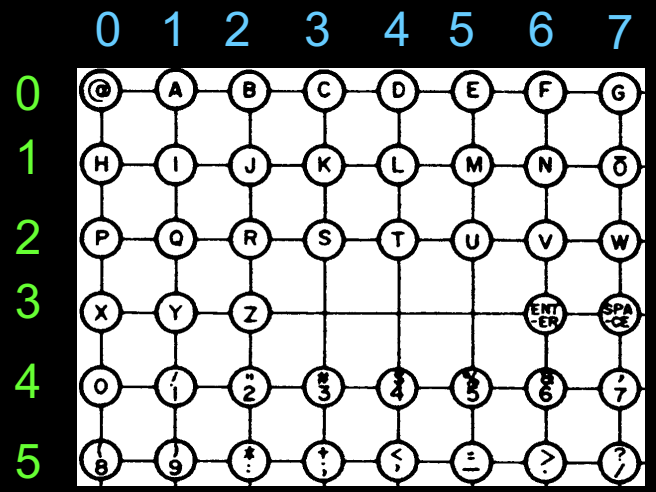
corresponding bit
(read from 49151)



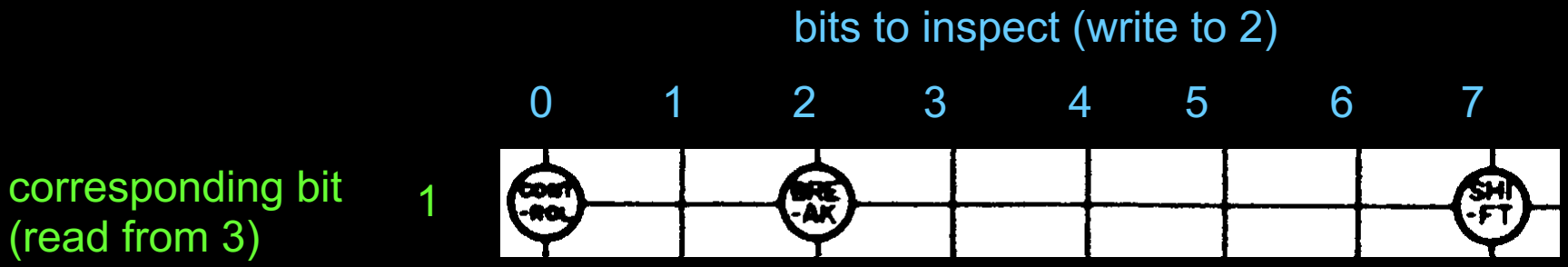

```
ldaa #%11111110 ;bit 0 corresponds to ‘X’
staa 2
ldaa 49151
bita #%00001000
beq  keypressed
```

Example: Direction keys?

```
ldaa #%01111111
staa 2
ldaa 49151
bita #%00000100
beq  keyW
ldaa #%11111011
staa 2
ldaa 49151
bita #%00001000
beq  keyZ
ldaa #%11111101
staa 2
ldaa 49151
bita #%00000001
beq  keyA
ldaa #%11110111
staa 2
ldaa 49151
bita #%00000100
beq  keyS
```



Keys associated with 3




```
;is BREAK pressed?
ldaa #%11111011 ;bit 2 corresponds to ‘break’
staa 2
ldaa 3
bita #%00000010 ;bit 1 cleared for CTL, BRK, and SHIFT
beq  keypressed
```

Tutorial #8
Bit manipulation

~~aba~~	~~beq~~	~~bra~~	daa	~~jmp~~	mul	rol	~~staa~~	~~tst~~
~~abx~~	~~bne~~	~~brn~~	~~dec~~	jsr		rola	~~stab~~	~~tsta~~
adca	~~bcc~~		~~deca~~		~~neg~~	rolb	~~std~~	~~tstb~~
adcb	~~bcs~~	~~cba~~	~~decb~~	~~ldaa~~	~~nega~~	ror	~~sts~~	
~~adda~~	~~bhi~~	clc	~~des~~	~~ldab~~	~~negb~~	rora	~~stx~~	wai
~~addb~~	~~bhs~~	cli	~~dex~~	~~ldd~~	nop	rorb	~~suba~~	
~~addd~~	~~blo~~	clr		~~lds~~		rti	~~subb~~	
anda	~~bls~~	clra	~~eora~~	~~ldx~~	oraa	rts	~~subd~~	
andb	~~bge~~	clrb	~~eorb~~	lsl	orab		swi	
asr	~~bgt~~	clv		lsla		~~sba~~		
asra	~~ble~~	~~cmpa~~	~~inc~~	lslb	psha	sbca	tab	
asrb	~~blt~~	~~cmpb~~	~~inca~~	lsld	pshb	sbcb	tap	
	~~bmi~~	~~cpx~~	~~incb~~	lsr	pshx	sec	tba	
bita	~~bpl~~	com	~~ins~~	lsra	pula	sei	tpa	
bitb	~~bvc~~	coma	~~inx~~	lsrb	pulb	sev	tsx	
bsr	~~bvs~~	comb		lsrd	pulx		txs	

Clear Instructions

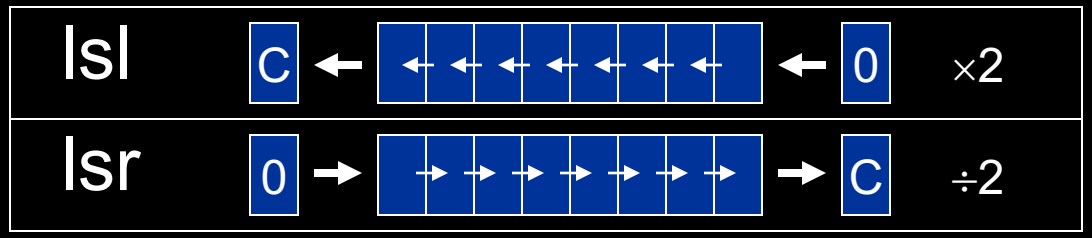
- Equivalent to loading A or B with zero (clra, clrb)
- You may clear a memory address directly through either the extended or indexed modes (clr)

Complement Instructions

- Flips all the bits in either an accumulator (coma, comb) or register or memory address (com)

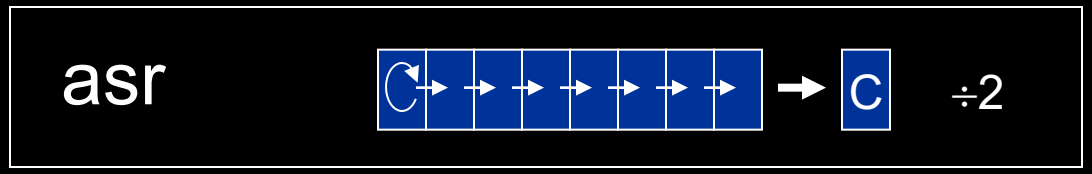
Logical Shifting

- A quick way of multiplying/dividing an unsigned number by 2.
- You can shift the accumulators or memory left/right by 1 bit.
  (lsla, lslb, lsld, lsl; lsra, lsrb, lsrd, lsr)
- Zero is shifted in, and the remaining bit is shifted into the carry.



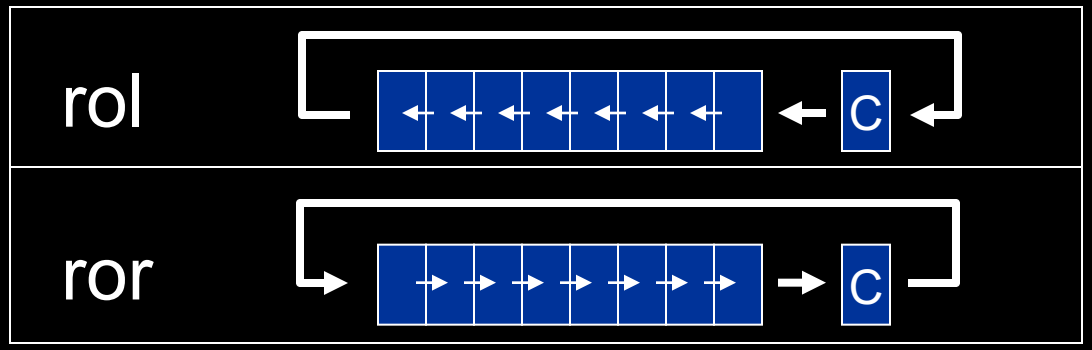
Arithmetic Shift Right

- A quick way of dividing a signed number by 2.
- You can shift the A or B accumulator or memory right by 1 bit. (asra, asrb, asr)
- The “sign” bit is left unchanged, and the remaining bit is shifted into the carry.
- Signed multiplication by 2 can be done using the lsl instructions



Rotating through the carry

- You can shift the A or B accumulator or memory left/right by 1 bit.
  (rola, rolb, rol; rora, rorb, ror)
- The carry is shifted in, and the remaining bit is shifted into the carry.



Keyboard Strobe Example

tut8a.txt - Notepad

File Edit Format View Help

```
temp1   .equ 32766
temp2   .equ 32767
        .org 32768      ;tell tasm to start at 32768

        ldx #16384      ;clear the screen
        ldaa #32
clrscn  staa ,x
        inx
        cpx #16895
        bls clrscn

;First, draw the keypad
        clra                    ;a hold current char
        ldx #16384+12+128       ;start of keypad
drwrow  ldab #8                 ;b holds column count
drwnxt  staa ,x                 ;write char to screen
        inca                    ;get next char
        inx                     ;bump screen pointer
        cmpa #64                ;see if done
        beq  drwent             ;draw the enter key if done
        cmpa #27                ;see if gone past 'Z'
        bne  drmore             ;keep going
        ldx  #16384+12+128+128  ;otherwise, load next row
        ldaa #48                ;set char to '0' (ascii 48)
        bra  drwrow             ;draw next row
drmore  decb                    ;decrement column count
        bne  drwnxt             ;go if not done with column
        ldab #24                ;otherwise, bump screen pointer by 24
        abx
        bra drwrow              ;draw next row
drwent  ldaa #31                ;char for "<-" symbol
        staa 16384+12+128+96+6  ;store in "enter" location

;Now highlight keys as they are pressed
keylite ldx #16384+12+128       ;start of keypad
        ldaa #1                 ;start with bit 0
        staa temp1              ;temp1 = which bit to mask  (0-5)
        staa temp2              ;temp2 = which bit to check (0-7)
nxlite  ldaa temp2
        coma                    ;flip the bits
        staa 2                  ;store in keystrobe
        ldaa 49151              ;get the key group
        ldab ,x                 ;get the current screen char
        bita temp1              ;check the key against the current one to inspect
        beq hilite              ;hilite if pressed.
        andb #%00111111         ;otherwise de-highlight it
        bra mklite
hilite  orab #%01000000         ;set the hilight bit of the char
mklite  stab ,x                 ;write char to screen
        inx                     ;bump screen pointer
        lsl temp2               ;inspect next bit
        bcc nxlite              ;go if not done
        rol temp2               ;store bit back at bit0.
        ldab #24                ;bump screen pointer by 24
        abx
        ldab temp1              ;bump mask to next bit
        lslb
        stab temp1
        cmpb #%01000000         ;see if no more keys left
        bne nxlite

        ldaa #%11111011         ; check break key
        staa 2
        ldaa 3
        bita #%00000010
        bne keylite             ; keep hilighting keys until break pressed.
        rts

        .end
```

Video Modes

- The MC6847 is capable of two major modes of execution
  - Major Mode 1:
    A fully-interchangeable 32x16 character format
  - Major mode 2:
    A dedicated graphics mode

Video Control

49151 ($BFFF)	7	6	5	4	3	2	1	0
	Sound	CSS	$\overline{A}$/G	GM0	GM1	GM2 $\overline{INT}$/EXT	N/C	N/C

- The MC6847 $\overline{INT}$/EXT pin is physically tied to the GM2 pin.

Clearing bit 5 of the control register puts the MC6847 into “Major Mode 1” which can display basic text and limited graphics characters.

Setting bit 5 of the control register puts the MC6847 into “Major Mode 2” which is divided into a two color (resolution graphics) or four color (color graphics) mode.

TABLE 1 — MODE CONTROL LINES (INPUTS)

$\overline{A}$/G	$\overline{A}$/S	$\overline{INT}$/EXT	INV	GM2	GM1	GM0	Alpha/Graphic Mode Select	# of Colors
0	0	0	0	X	X	X	Internal Alphanumerics	2
0	0	0	1	X	X	X	Internal Alphanumerics Inverted	
0	0	1	0	X	X	X	External Alphanumerics	
0	0	1	1	X	X	X	External Alphanumerics Inverted	
0	1	0	X	X	X	X	Semigraphics 4 (SG4)	8
0	1	1	X	X	X	X	Semigraphics 6 (SG6)	8
1	X	X	X	0	0	0	64 × 64 Color Graphics One (CG1)	4
1	X	X	X	0	0	1	128 × 64 Resolution Graphics One (RG1)	2
1	X	X	X	0	1	0	128 × 64 Color Graphics Two (CG2)	4
1	X	X	X	0	1	1	128 × 96 Resolution Graphics Two (RG2)	2
1	X	X	X	1	0	0	128 × 96 Color Graphics Three (CG3)	4
1	X	X	X	1	0	1	128 × 192 Resolution Graphics Three (RG3)	2
1	X	X	X	1	1	0	128 × 192 Color Graphics Six (CG6)	4
1	X	X	X	1	1	1	256 × 192 Resolution Graphics Six (RG6)	2

6847 Major Mode 1

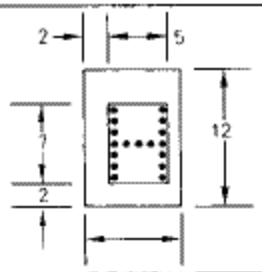
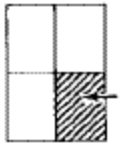
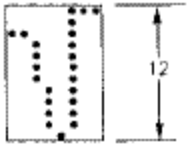
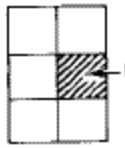
DISPLAY MODES

There are two major display modes in the VDG. Major mode 1 contains four alphanumeric and two limited graphic modes. Major mode 2 contains eight graphic modes. Of these, four are full color graphic and four restricted color graphic modes. The mode selection for the VDG is summarized in Table 2. The mnemonics of these fourteen modes are explained in the following sections.

In major mode 1 the display window is divided into 32 columns by 16 character element rows thus requiring 512 bytes of memory. Each character element is 8 half periods by 12 scan lines in size as shown in Figure 19. The area outside the display window is black.

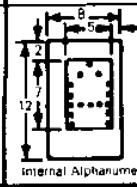
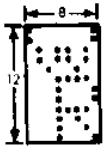
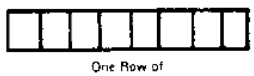
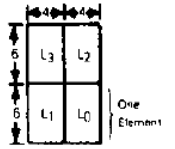
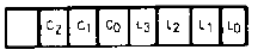
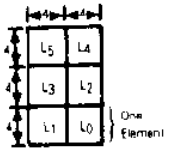
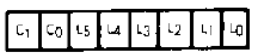
The VDG has a built-in character generator ROM containing the 64 ASCII characters in a 5 × 7 format (see Figure 20). The 5 × 7 character font is positioned two columns to the right and three rows down within the 8 × 12 character element. Six bits of the 8-bit data word are typically used for the internal ASCII character generator. The remaining two bits may be used to implement inverse video, color switching, or external character generator ROM selection on a character-by-character basis. For those who wish to display lower case letters, special characters, or even limited-graphics, an external ROM may be used. If such external ROM is used, all of the 8 × 12 picture elements, or pixels, in the character element can be utilized. Characters may be either green on a dark green background or orange on a dark orange background, depending on the state of the CSS pin. The invert pin can be used to display dark characters on a bright background.

TABLE 2 — SUMMARY OF MAJOR MODES
Major Mode 1 — Alpha Modes

Title	Memory	Display Elements	Colors	Title	Memory	Display Elements	Colors
Alphanumerics (Internal)	512 × 8		2	Semigraphic 4	512 × 8		8
Alphanumerics (External)	512 × 8		2	Semigraphic 6	512 × 8		4

6847 Major Mode 1

- The MC6847 is capable of simultaneously displaying the following *all on the same screen* on a character-by-character basis:
  - 64 Internally or 64 externally generated alpha-numeric characters of *any* of the four color schemes
    - Dark green on light green
    - Light green on dark green
    - Dark orange on light orange
    - Light orange on dark orange
  - Any 2x2 or 2x3 graphics character of *any* one of the eight colors on black
  - (512 alpha-numerics + 1024 graphics characters = 1152 possible characters)
  - [This would correspond to 11 (really 10.5) bits to uniquely specify each character]

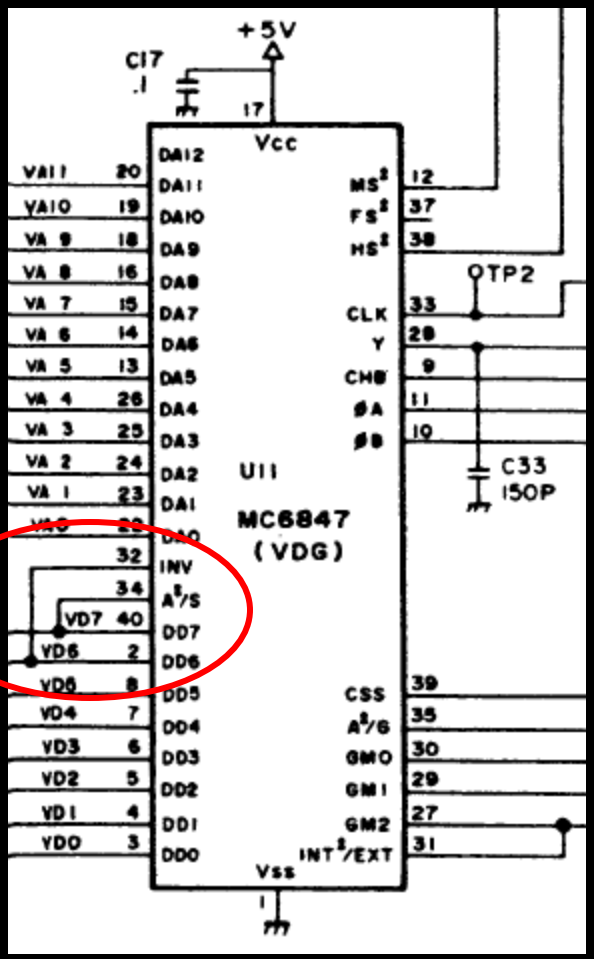
VDG Pins								Color			TV Screen		VDG Data Bus	Comments
MS	G/A	S/A	EXT/INT	GM2	GM1	GM0	CSS	INV	Character Color	Background	Border	Display Mode	Detail		
1	0	0	0	x	x	x	0 1	0 1 0 1	Green Black Orange Black	Black Green Black Orange	Black Black	32 Characters per row 16 Characters in rows	 Internal Alphanumerics		The ALPHANUMERIC INTERNAL mode uses an internal character generator (which contains the following five dot by seven dot characters @ABCDEFGHIJKLMNOPQRSTUVWXYZ [\]↑← SP !"#$%&'()*+,-./0123456789:;<=>? The six bit ASCII code leaves two bits free and these may be externally connected to the mode pins (G/A, S/A, EXT/INT, GM2, GM1, GM0, CSS or INV).
1	0	0	1	x	x	x	0 1	0 1 0 1	Green Black Orange Black	Black Green Black Orange	Black Black	32 Characters per row 16 Characters in rows			The ALPHANUMERIC EXTERNAL mode uses an external character generator as well as a row counter. Thus, custom character fonts or graphic symbol sets with up to 256 different 8 x 12 dot "characters" may be displayed
1	0	1	0	x	x	x	x	x	L2 C2 C1 C0 Color 0 x x x Black 1 0 0 0 Green 1 0 0 1 Yellow 1 0 1 0 Blue 1 0 1 1 Red 1 1 0 0 Buff 1 1 0 1 Cyan 1 1 1 0 Magenta 1 1 1 1 Orange	Black	64 Display elements per row 32 Display elements in rows			The SEMIGRAPHICS FOUR mode uses an internal "course graphics" generator in which a rectangle (eight dots by twelve dots) is divided into four equal parts. The luminance of each part is determined by a corresponding bit on the VDG data bus. The color of illuminated parts is determined by three bits
1	0	1	1	x	x	x	0 1	x	L2 C1 C0 Color 0 x x Black 1 0 0 Green 1 0 1 Yellow 1 1 0 Blue 1 1 1 Red 0 x x Black 1 0 0 Buff 1 0 1 Cyan 1 1 0 Magenta 1 1 1 Orange	Black	64 Display elements per row 48 Display elements in rows			The SEMIGRAPHIC SIX mode is similar to the SEMIGRAPHIC FOUR mode with the following differences. The eight dot by twelve dot rectangle is divided into six equal parts. Color is determined by the two remaining bits
									C1 C0 Color						The COLOR GRAPHICS ONE mode uses a maximum of 1024 bytes of

6847 Major Mode 1

- The MC-10 uses only 8-bits to specify the characters in major mode 1.
  - bit7 of the incoming data character is wired to control the A/S line. This lets you switch between semi-graphics and alpha-numeric characters
  - bit 6 of the incoming data character is set to control the INV line (which is used in alpha-numeric mode to determine if a character's color scheme is inverted)

TABLE 1 — MODE CONTROL LINES (INPUTS)

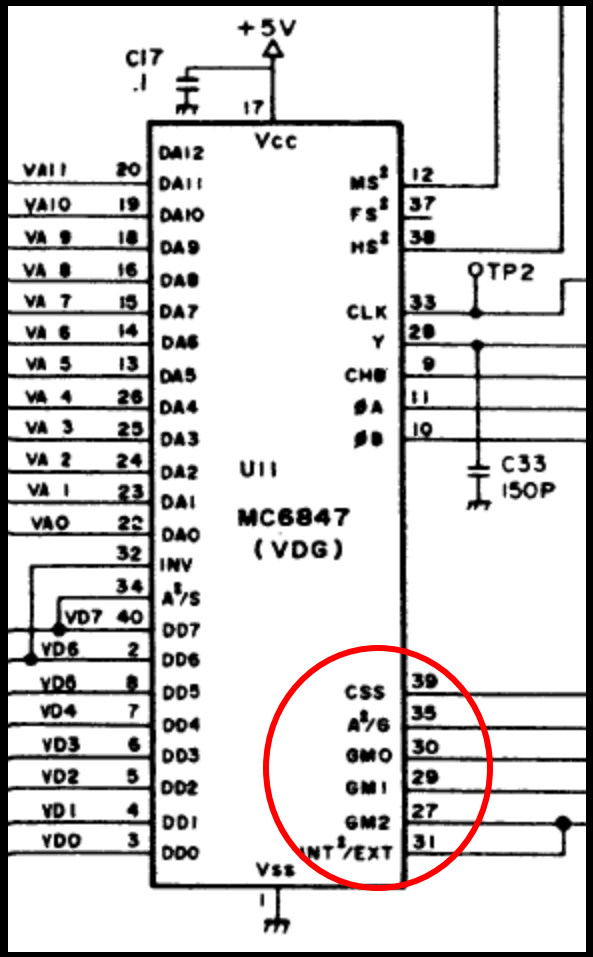
A/G	A/S	INT/EXT	INV	GM2	GM1	GM0	Alpha/Graphic Mode Select	# of Colors
0	0	0	0	X	X	X	Internal Alphanumerics	2
0	0	0	1	X	X	X	Internal Alphanumerics Inverted	
0	0	1	0	X	X	X	External Alphanumerics	
0	0	1	1	X	X	X	External Alphanumerics Inverted	
0	1	0	X	X	X	X	Semigraphics 4 (SG4)	8
0	1	1	X	X	X	X	Semigraphics 6 (SG6)	8
1	X	X	X	0	0	0	64 × 64 Color Graphics One (CG1)	4
1	X	X	X	0	0	1	128 × 64 Resolution Graphics One (RG1)	2
1	X	X	X	0	1	0	128 × 64 Color Graphics Two (CG2)	4
1	X	X	X	0	1	1	128 × 96 Resolution Graphics Two (RG2)	2
1	X	X	X	1	0	0	128 × 96 Color Graphics Three (CG3)	4
1	X	X	X	1	0	1	128 × 192 Resolution Graphics Three (RG3)	2
1	X	X	X	1	1	0	128 × 192 Color Graphics Six (CG6)	4
1	X	X	X	1	1	1	256 × 192 Resolution Graphics Six (RG6)	2



6847 Major Mode 1

The remaining control pins that govern the first major mode (CSS, $\overline{\text{INT}}$/EXT) are governed by writing to the Video Control register and are not selectable by the data character.

- Only text and inverted text of one color can be displayed on the screen with any 2x2 any-colored-character block because the color select pin (CSS) is globally settable.
- Having the CSS restricted to a global setting restricts the color palette of the 2x3 blocks to either green-yellow-blue-red or buff-cyan-magenta-orange.
- Since bit 7 of the incoming data character is tied to the A/S pin, the color selection of the 2x3 blocks are restricted to either red/blue or magenta/orange.
- Since the $\overline{\text{INT}}$/EXT line is also globally settable, the 2x3 character blocks must be used in conjunction with the mode for an external character ROM. When bit7 goes low, the chip tries to use the external character ROM (which is not implemented on the MC-10).



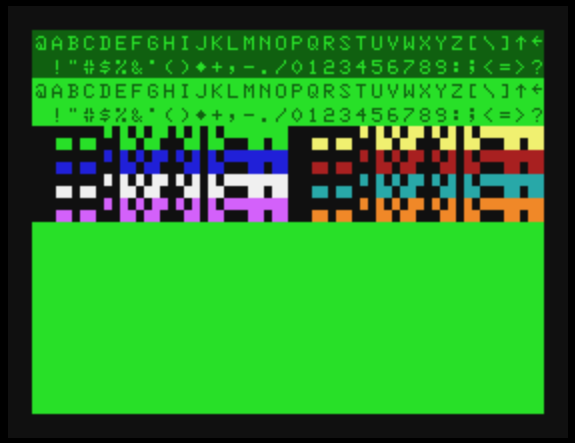
Major Mode 1

store at $BFFF (49151)

SG4 64x32

$4000-$41FF
16384-16895

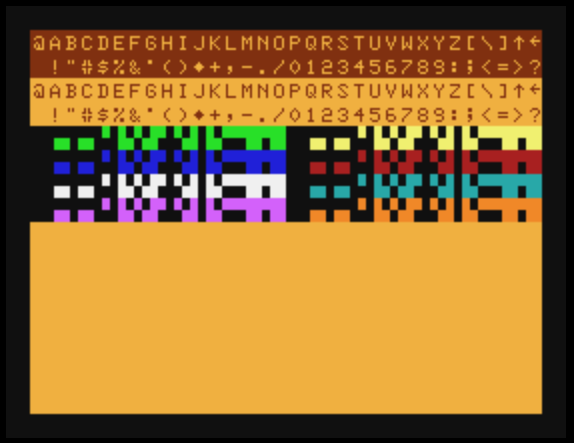
$BFFF←$00
POKE 49151, 0



SG4 64x32

$4000-$41FF
16384-16895

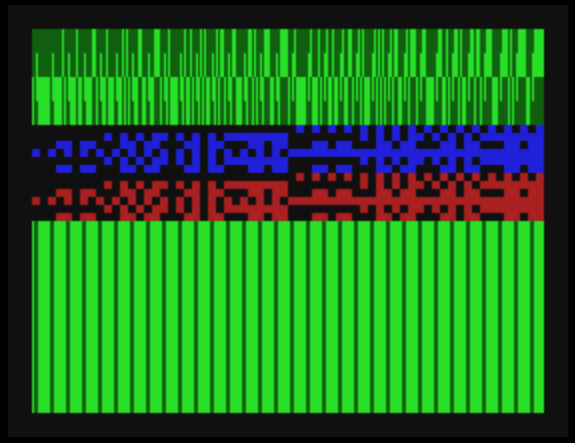
$BFFF←$40
POKE 49151,64



SG6 64x48

$4000-$41FF
16384-16895

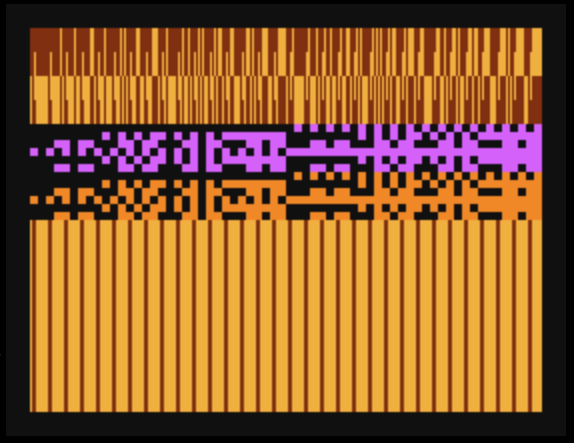
$BFFF←$0C
POKE 49151,12



SG6 64x48

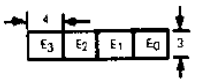
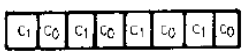
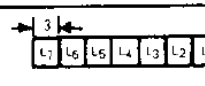
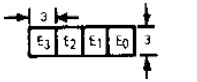
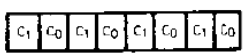
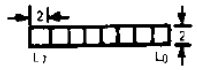
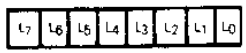
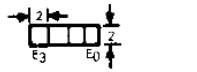
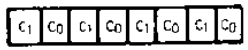
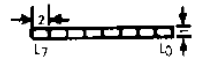
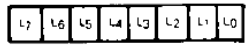
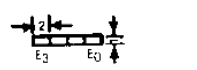
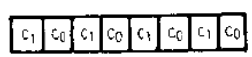
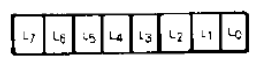
$4000-$41FF
16384-16895

$BFFF←$4C
POKE 49151,74



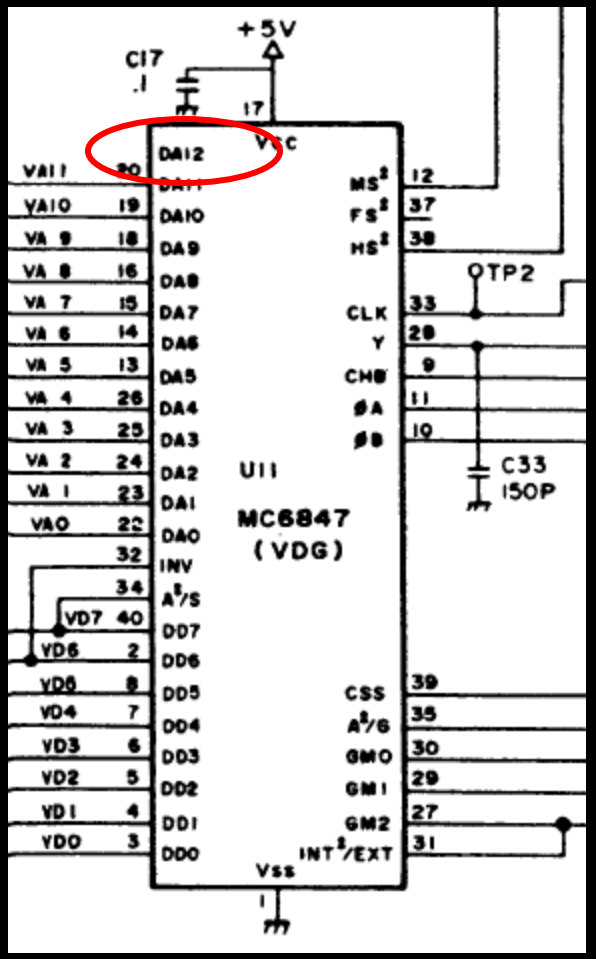
Major Mode 2

- Since the CSS pin is globally settable only one palette at a time is displayed on-screen for the graphics modes.
  - Resolution graphics modes: green/dark green or buff/black
  - Color graphics modes: green-yellow-blue-red or buff-cyan-magenta-orange
- The MC6847 would otherwise allow you to switch between these color schemes horizontally every eight cells (resolution graphics) or four cells (color graphics)

												Orange					
1	1	X	X	0	0	0	0	X		C1 C0 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1	Color Green Yellow Blue Red Buff Cyan Magenta Orange	Green Buff	64 Display elements per row 64 Display elements			The COLOR GRAPHICS ONE mode uses a maximum of 1024 bytes of display RAM in which one pair of bits specifies one picture element.
1	1	X	X	0	0	1	0	X		Lx 0 1 0 1	Color Black Green Black Buff	Green Buff	128 Display elements per row 64 Display elements in rows			The RESOLUTION GRAPHICS ONE mode uses a maximum of 1024 bytes of display RAM in which one bit specifies one picture element.
1	1	X	X	0	1	0	0	X	Same color as Color Graphics One			Green Buff	128 Display elements per row 64 Display elements in rows			The COLOR GRAPHICS TWO mode uses a maximum of 2048 bytes of display RAM in which one pair of bits specifies one picture element.
1	1	X	X	0	1	1	0	X	Same color as Resolution Graphics One			Green Buff	128 Display elements per row 96 Display elements in rows			The RESOLUTION GRAPHICS TWO mode uses a maximum of 1536 bytes of display RAM in which one bit specifies one picture element.
1	1	X	X	1	0	0	0	X	Same color as Color Graphics One			Green Buff	128 Display elements per row 96 Display elements in rows			The COLOR GRAPHICS THREE mode uses a maximum of 3072 bytes of display RAM in which one pair of bytes specifies one picture element.
1	1	X	X	1	0	1	0	X	Same color as Resolution Graphics One			Green Buff	128 Display elements per row 192 Display elements in rows			The RESOLUTION GRAPHICS THREE mode uses a maximum of 3072 bytes of display RAM in which one bit specifies one picture element.
1	1	X	X	1	1	0	0	X	Same color as Color Graphics One			Green Buff	128 Display elements per row 192 Display Elements in rows			The COLOR GRAPHICS SIX mode uses a maximum of 6144 bytes of display RAM in which one pair of bits specifies one picture element.
1	1	X	X	1	1	1	0	X	Same color as Resolution Graphics One			Green Buff	256 Display elements per row 192 Display elements in rows			The RESOLUTION GRAPHICS SIX mode uses a maximum of 6144 bytes of display RAM in which one bit specifies one picture element.

Major Mode 2

- The on-board Video RAM for the MC-10 is 4K in size (12 bit address: $4000-$4FFF).
- Thus only 12 of the 13 address lines were wired to the MC6847. (DA12 is unconnected)
- CG6 and RG6 will paint data from $4000 - $47FF instead of $5000 - $57FF.



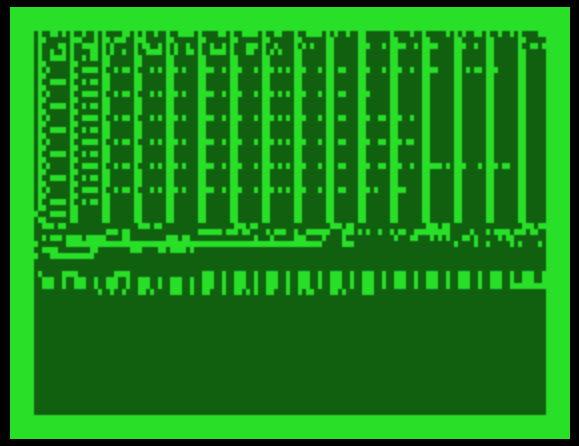
Major Mode 2

store at $BFFF (49151)

RG1 128x64

$4000-$43FF
16384-17407

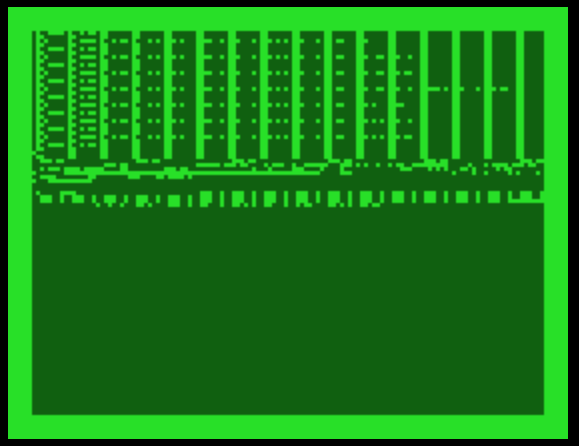
$BFFF←$30
POKE 49151,48



RG2 128x96

$4000-$45FF
16384-17919

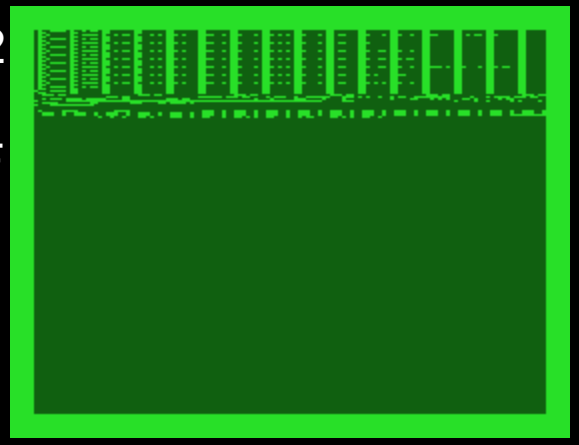
$BFFF←$38
POKE 49151,56



RG3 128x192

$4000-$4BFF
16384-19455

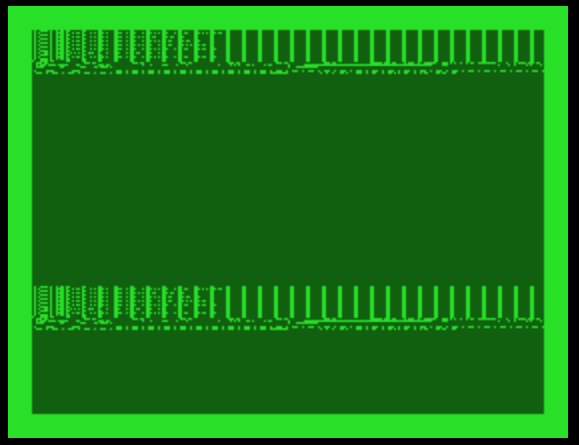
$BFFF←$34
POKE 49151,52



RG6 256x192

$4000-$4FFF
16384-20479

$BFFF←$3C
POKE 49151,60



Major Mode 2

store at $BFFF (49151)

RG1 128x64

$4000-$43FF
16384-17407

$BFFF←$70
POKE 49151,112

RG2 128x96

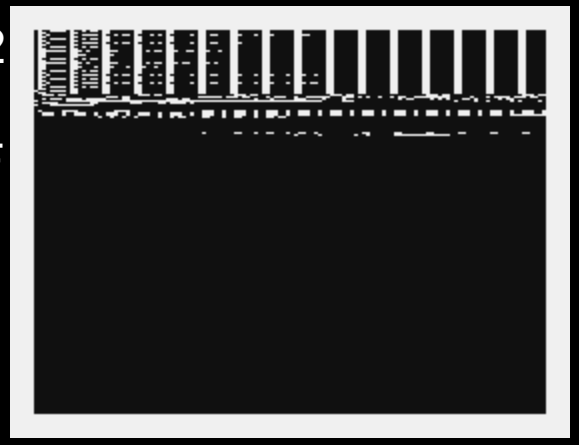
$4000-$45FF
16384-17919

$BFFF←$78
POKE 49151,120

RG3 128x192

$4000-$4BFF
16384-19455

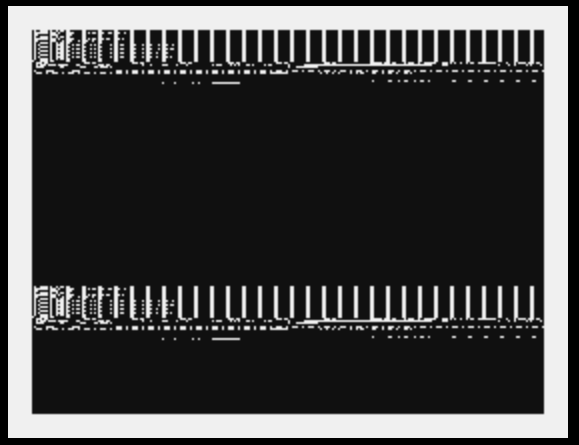
$BFFF←$74
POKE 49151,116



RG6 256x192

$4000-$4FFF
16384-20479

$BFFF←$7C
POKE 49151,124



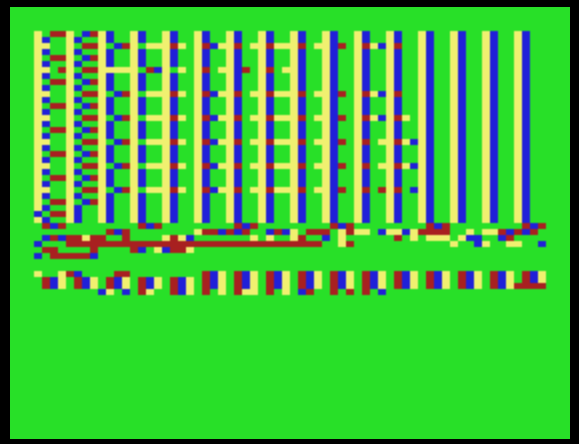
Major Mode 2

store at $BFFF (49151)

CG1 64x64

$4000-$43FF
16384-17407

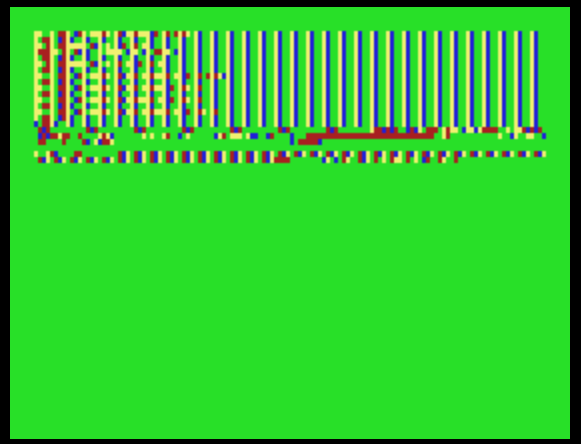
$BFFF←$20
POKE 49151,32



CG2 128x64

$4000-$47FF
16384-18431

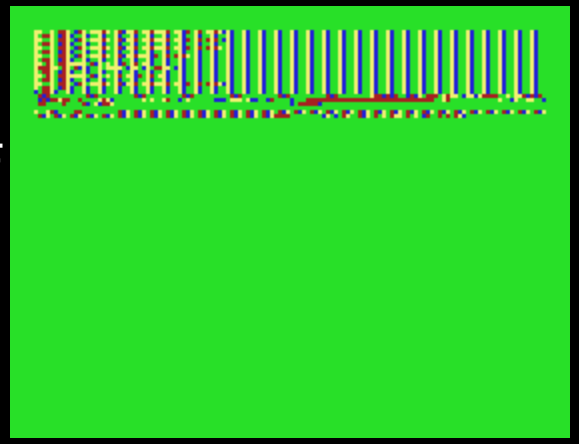
$BFFF←$28
POKE 49151,40



CG3 128x96

$4000-$4BFF
16384-19455

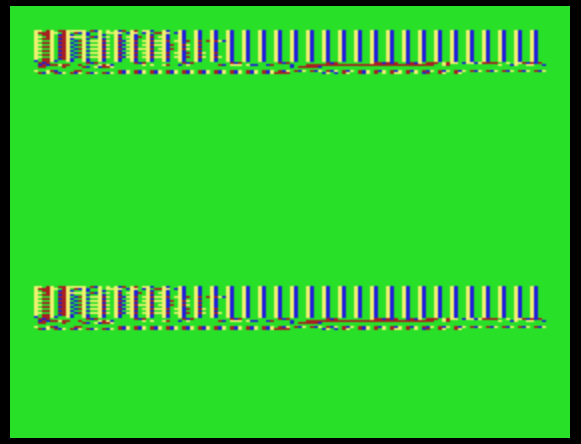
$BFFF←$24
POKE 49151,36



CG6 128x192

$4000-$4FFF
16384-20479

$BFFF←$2C
POKE 49151,44



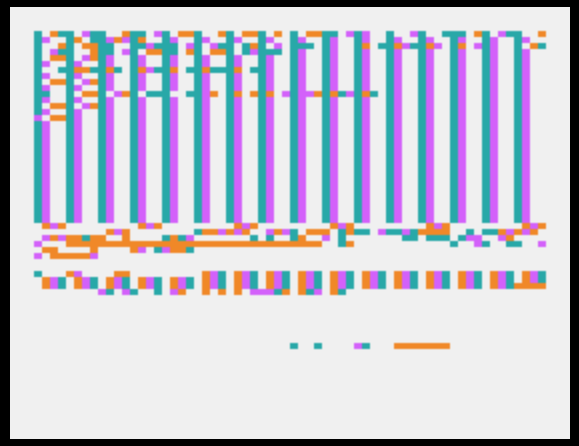
Major Mode 2

store at $BFFF (49151)

CG1 64x64

$4000-$43FF
16384-17407

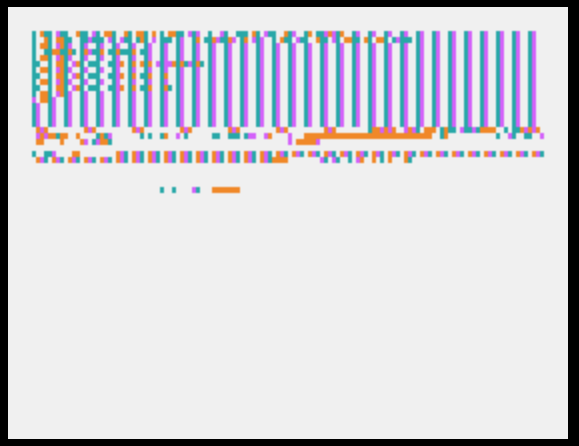
$BFFF←$60
POKE 49151,96



CG2 128x64

$4000-$47FF
16384-18431

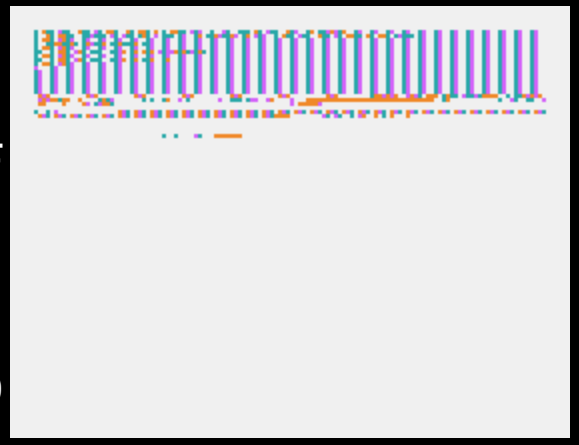
$BFFF←$68
POKE 49151,104



CG3 128x96

$4000-$4BFF
16384-19455

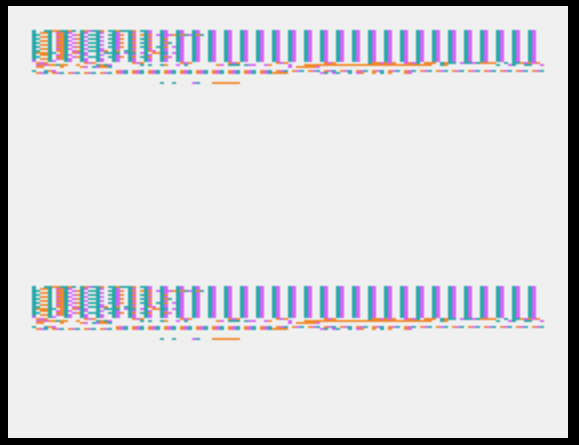
$BFFF←$64
POKE 49151,100



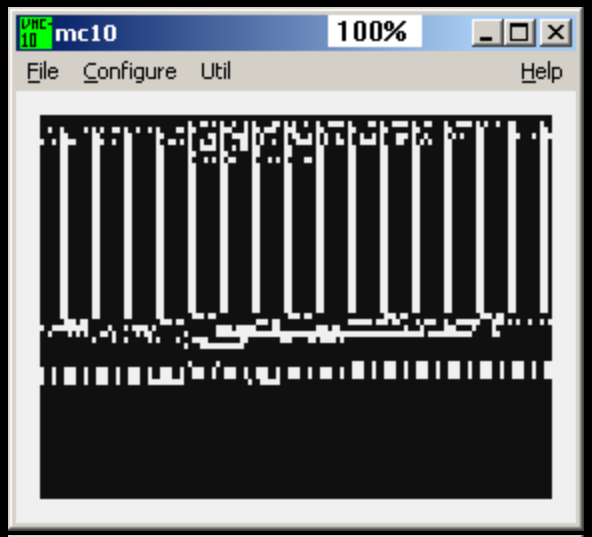
CG6 128x192

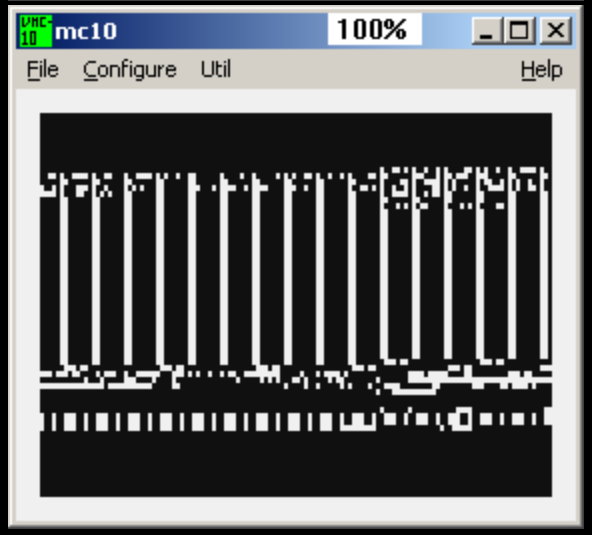
$4000-$4FFF
16384-20479

$BFFF←$6C
POKE 49151,108



Rotating the screen





tut8b.txt - Notepad

File Edit Format View Help

```
temp    .equ 32766
        .org 32768      ;tell tasm to start at 32768

        ldaa #112       ;put MC10 into low res graphics mode
        staa 49151      ;
        ldd  #8193      ;1024 bytes * 8 bits/byte + 1 carry bit
        std  temp       ;store into temp location
rstart  ldx  #16384     ;start of screen
        ldd  #1024      ;1024 bytes for this mode
again   ror  ,x         ;rotate right with carry
        inx             ;move to next screen position
        decb            ;decrement lower byte of count
        bne again       ;branch if not done yet
        deca            ;decrement upper byte of count
        bne again       ;branch if not done yet
        ldx temp        ;decrement temp counter
        dex
        stx temp
        bne rstart
        clr 49151       ;restore SG4 graphics mode
        rts

        .end
```

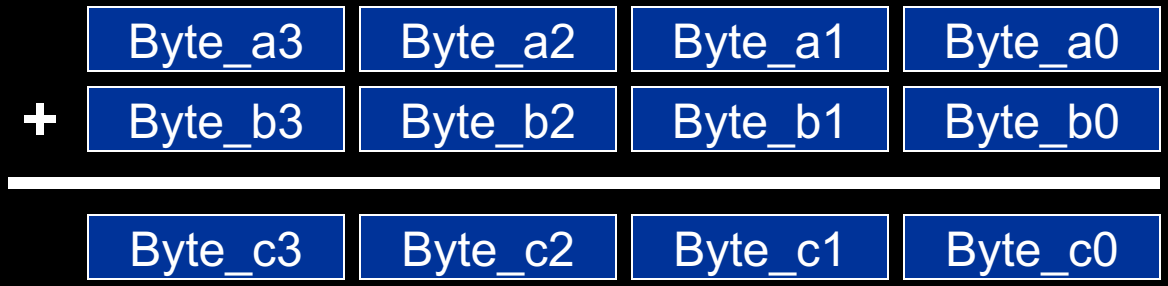
Tutorial #9
Multi-byte math

~~aba~~	~~beq~~	~~bra~~	daa	~~jmp~~	mul	~~rol~~	~~staa~~	~~tst~~
~~abx~~	~~bne~~	~~brn~~	~~dec~~	jsr		~~rola~~	~~stab~~	~~tsta~~
adca	~~bcc~~		~~deca~~		~~neg~~	~~rolb~~	~~std~~	~~tstb~~
adcb	~~bcs~~	~~cba~~	~~decb~~	~~ldaa~~	~~nega~~	~~ror~~	~~sts~~	
~~adda~~	~~bhi~~	clc	~~des~~	~~ldab~~	~~negb~~	~~rora~~	~~stx~~	wai
~~addb~~	~~bhs~~	cli	~~dex~~	~~ldd~~	nop	~~rorb~~	~~suba~~	
~~addd~~	~~blo~~	~~clr~~		~~lds~~		rti	~~subb~~	
~~anda~~	~~bls~~	~~clra~~	~~eora~~	~~ldx~~	~~oraa~~	rts	~~subd~~	
~~andb~~	~~bge~~	~~clrb~~	~~eorb~~	lsl	~~orab~~		swi	
~~asr~~	~~bgt~~	clv		lsla		~~sba~~		
~~asra~~	~~ble~~	~~cmpa~~	~~inc~~	lslb	psha	sbca	tab	
~~asrb~~	~~blt~~	~~cmpb~~	~~inca~~	lsld	pshb	sbcb	tap	
	~~bmi~~	~~cpx~~	~~incb~~	lsr	pshx	sec	tba	
~~bita~~	~~bpl~~	~~com~~	~~ins~~	lsra	pula	sei	tpa	
~~bitb~~	~~bvc~~	~~coma~~	~~inx~~	lsrb	pulb	sev	tsx	
bsr	~~bvs~~	~~comb~~		lsrd	pulx		txs	

Adding with Carry

- The add with carry instructions (adca, adcb) will add the A or B register with the carry bit.
- Since the carry is set when the result of an addition instruction doesn’t fit in 8 or 16 bits, you can use the adc instructions to continue adding the larger bytes.

Adding with Carry

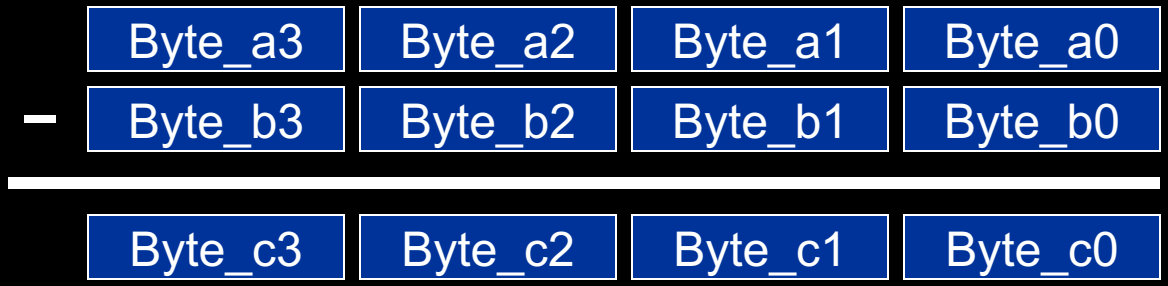



```
ldaa Byte_a0 ;get least significant byte
adda Byte_b0 ;add with corresponding byte (carry set if result rolled over)
staa Byte_c0 ;store result

ldaa Byte_a1 ;get next significant byte
adca Byte_b1 ;add with corresponding byte (+1 if carry bit set)
staa Byte_c1 ;store result

ldaa Byte_a2 ;carrying forward...
adca Byte_b2
staa Byte_c2

ldaa Byte_a3 ;carrying forward...
adca Byte_b3
staa Byte_c3
```

Subtracting with Carry

- The subtract with carry instructions (sbca, sbcb) will subtract from the A or B register along with the carry bit.
- Since the carry is set when the result of an subtraction instruction doesn’t fit in 8 or 16 bits, you can use the sbc instructions to continue subtracting the larger bytes.

Subtracting with Carry




```
ldaa Byte_a0 ;get least significant byte
suba Byte_b0 ;subtract corresponding byte (carry set if result rolled over)
staa Byte_c0 ;store result

ldaa Byte_a1 ;get next significant byte
sbca Byte_b1 ;subtract corresponding byte (additional -1 if carry bit set)
staa Byte_c1 ;store result

ldaa Byte_a2 ;carrying forward...
sbca Byte_b2
staa Byte_c2

ldaa Byte_a3 ;carrying forward...
sbca Byte_b3
staa Byte_c3
```

Multiply Instruction

- Multiplies register A with B and stores the result back into the D register.
- It assumes A and B are unsigned.
- Since the largest possible multiplication is 255 x 255 = 65025, the result will always fit into the D register

Multi-byte Multiplication

- Can be performed similar to long multiplication
  - Compute product of pairs of bytes
  - Add the results together

Example: Multiply the two 16-bit numbers stored at M and N and save the product at location P.

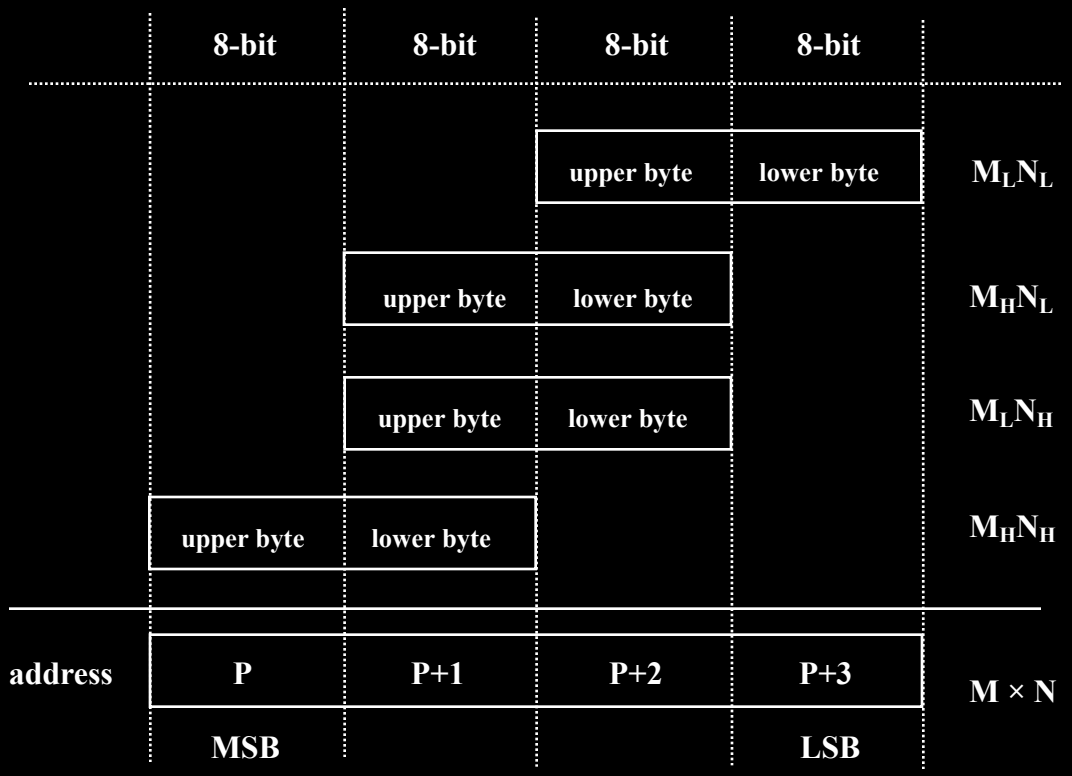
(from http://www.cs.ucf.edu/~tkocak/eel4767/lec4.ppt)

- First, rewrite M and N as $M_H M_L$ and $N_H N_L$ where
  - M_H and N_H are upper 8 bits of M and N respectively
  - M_L and N_L are lower 8 bits of M and N respectively

- M_H and M_L are stored at M and M+1 respectively
- N_H and N_L are stored at N and N+1 respectively

Illustration
16-bit by 16-bit multiplication

(from http://www.cs.ucf.edu/~tkocak/eel4767/lec4.ppt)



Program: Multiplying Two 16-bit Numbers

(from http://www.cs.ucf.edu/~tkocak/eel4767/lec4.ppt)

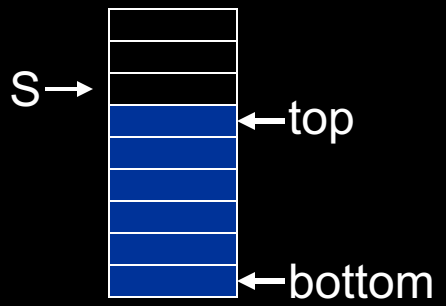
```
ldaa  M+1             ; place ML in A
ldab  N+1             ; place NL in B
mul                   ; compute ML × NL
std   P+2             ; save ML × NL to memory locations P+2 and P+3
ldaa  M               ; place MH in A
ldab  N               ; place NH in B
mul                   ; compute MH × NH
std   P               ; save MH × NH to memory locations P and P+1
ldaa  M               ; place MH in A
ldab  N+1             ; place NL in B
mul                   ; compute MH × NL
addd  P+1             ; add MH × NL to memory locations P+1 and P+2
std   P+1             ;
ldaa  P               ; add the C flag to memory location P
adca  #0              ;
staa  P               ;
ldaa  M+1             ; place ML in A
ldab  N               ; place NH in B
mul                   ; compute ML × NH
addd  P+1             ; add ML × NH to memory locations P+1 and P+2
ldaa  P               ; add the C flag to memory location P
adca  #0              ;
staa  P               ;
```

Tutorial #10
Stack Operations

~~aba~~	~~beq~~	~~bra~~	daa	~~jmp~~	mul	~~rol~~	~~staa~~	~~tst~~
~~abx~~	~~bne~~	~~brn~~	~~dec~~	jsr		~~rola~~	~~stab~~	~~tsta~~
~~adca~~	~~bcc~~		~~deca~~		~~neg~~	~~rolb~~	~~std~~	~~tstb~~
~~adcb~~	~~bcs~~	~~cba~~	~~decb~~	~~ldaa~~	~~nega~~	~~ror~~	~~sts~~	
~~adda~~	~~bhi~~	clc	~~des~~	~~ldab~~	~~negb~~	~~rora~~	~~stx~~	wai
~~addb~~	~~bhs~~	cli	~~dex~~	~~ldd~~	nop	~~rorb~~	~~suba~~	
~~addd~~	~~blo~~	~~clr~~		~~lds~~		rti	~~subb~~	
~~anda~~	~~bls~~	~~clra~~	~~eora~~	~~ldx~~	~~oraa~~	rts	~~subd~~	
~~andb~~	~~bge~~	~~clrb~~	~~eorb~~	lsl	~~orab~~		swi	
~~asr~~	~~bgt~~	clv		lsla		~~sba~~		
~~asra~~	~~ble~~	~~cmpa~~	~~inc~~	lslb	psha	~~sbca~~	tab	
~~asrb~~	~~blt~~	~~cmpb~~	~~inca~~	lsld	pshb	~~sbcb~~	tap	
	~~bmi~~	~~cpx~~	~~incb~~	lsr	pshx	sec	tba	
~~bita~~	~~bpl~~	~~com~~	~~ins~~	lsra	pula	sei	tpa	
~~bitb~~	~~bvc~~	~~coma~~	~~inx~~	lsrb	pulb	sev	tsx	
bsr	~~bvs~~	~~comb~~		lsrd	pulx		txs	

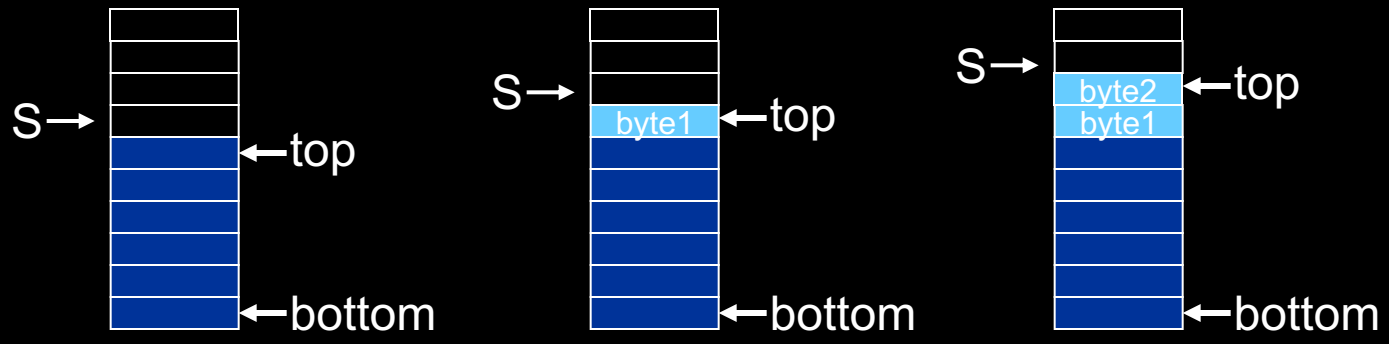
The Stack

- Contiguous area of memory used to keep track of subroutines and temporary variables.
- It grows and shrinks from one end.
- The S register points to the first empty byte at the top.



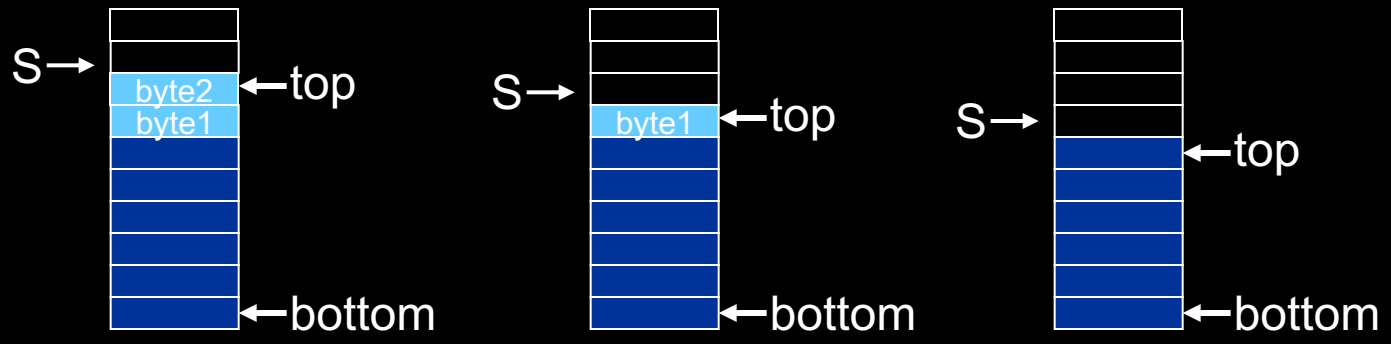
Writing to the Stack

- When you save (or “push”) bytes to the stack, the 6803 will
  - decrement the S register by the number of bytes
  - Store the bytes to the top of the stack



Reading from the Stack

- When you retrieve (or “pull”) bytes from the stack, the 6803 will
  - get the bytes to the top of the stack
  - increment the S register by the number of bytes



Basic stack instructions

- You can:
  - push/pull either accumulator to/from the stack (psha, pshb; pula, pulb)
  - push/pull the index register to/from the stack (pshx, pulx)
  - increment or decrement the stack pointer without any data transfer (ins, des)
- The condition codes are not affected by these instructions.

Subroutines

- You can either branch to a subroutine (bsr) or jump to a subroutine (jsr). The MC-10 will push the PC location of the next instruction to the stack, then jump to the specified destination.
- Subroutines are finished with a return from subroutine instruction (rts) which pulls the previously saved location from the stack and jumps to the location.

Stack Philosophy

- Use the stack
  - to store intermediate variables.
  - to protect or preserve register values between the caller and the subroutine

Tutorial #11
Transfer instructions

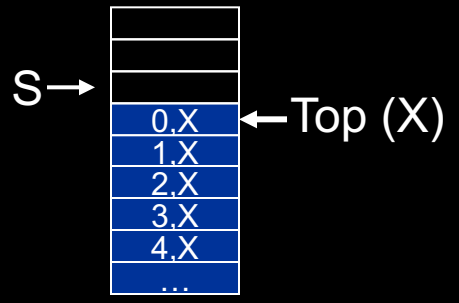
~~aba~~	~~beq~~	~~bra~~	daa	~~jmp~~	~~mul~~	~~rol~~	~~staa~~	~~tst~~
~~abx~~	~~bne~~	~~brn~~	~~dec~~	~~jsr~~		~~rola~~	~~stab~~	~~tsta~~
~~adca~~	~~bcc~~		~~deca~~		~~neg~~	~~rolb~~	~~std~~	~~tstb~~
~~adcb~~	~~bcs~~	~~cba~~	~~decb~~	~~ldaa~~	~~nega~~	~~ror~~	~~sts~~	
~~adda~~	~~bhi~~	clc	~~des~~	~~ldab~~	~~negb~~	~~rora~~	~~stx~~	wai
~~addb~~	~~bhs~~	cli	~~dex~~	~~ldd~~	nop	~~rorb~~	~~suba~~	
~~addd~~	~~blo~~	~~clr~~		~~lds~~		rti	~~subb~~	
~~anda~~	~~bls~~	~~clra~~	~~eora~~	~~ldx~~	~~oraa~~	~~rts~~	~~subd~~	
~~andb~~	~~bge~~	~~clrb~~	~~eorb~~	~~lsl~~	~~orab~~		swi	
~~asr~~	~~bgt~~	clv		~~lsla~~		~~sba~~		
~~asra~~	~~ble~~	~~cmpa~~	~~inc~~	~~lslb~~	~~psha~~	~~sbca~~	tab	
~~asrb~~	~~blt~~	~~cmpb~~	~~inca~~	~~lsld~~	~~pshb~~	~~sbcb~~	tap	
	~~bmi~~	~~cpx~~	~~incb~~	~~lsr~~	~~pshx~~	sec	tba	
~~bita~~	~~bpl~~	~~com~~	~~ins~~	~~lsra~~	~~pula~~	sei	tpa	
~~bitb~~	~~bvc~~	~~coma~~	~~inx~~	~~lsrb~~	~~pulb~~	sev	tsx	
~~bsr~~	~~bvs~~	~~comb~~		~~lsrd~~	~~pulx~~		txs	

8-bit Transfers

- You can transfer between accumulators
  - tba transfer b to a
  - tab transfer a to b
- You can transfer condition codes
  - tpa transfer condition codes to a
  - tap transfer a to condition codes

16-bit Transfers

- You can transfer the index/stack pointers
  - tsx transfer (S+1) to X
  - txs transfer (X-1) to S
- The +1 and -1 occurs so that X will point to the top of stack, and S will point to the first empty byte.



There is no direct way to transfer between the X and D registers

- You can push and pull the values of the X and D registers on the stack to load between them:

Transfer X to D:

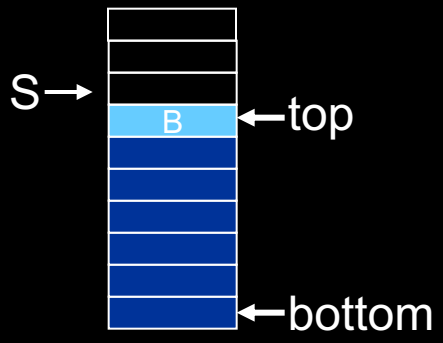
```
pshx ;put X on stack
pula ;load A with X's high byte
pulb ;load B with X's low byte
```

Transfer D to X:

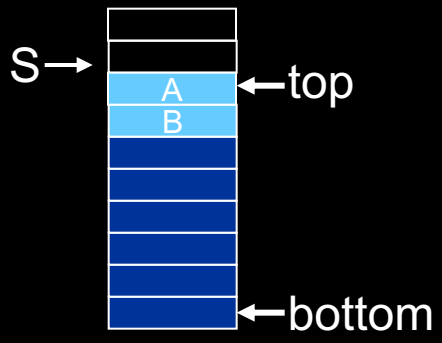
```
pshb ;put low-byte on stack
psha ;put high-byte on stack
pulx ;load X from stack
```

Transferring D to X

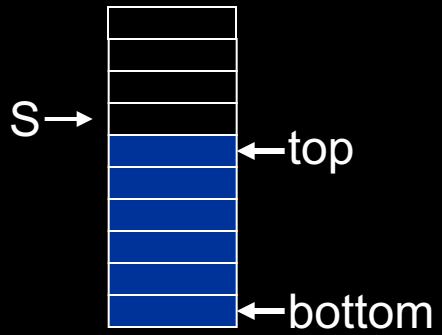
pshb



psha

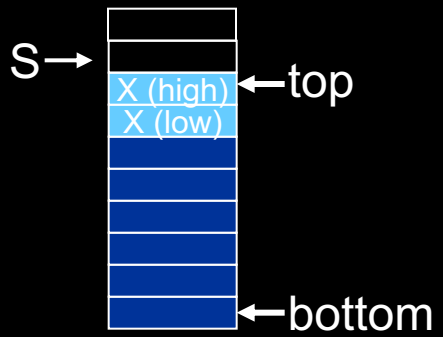


pulx

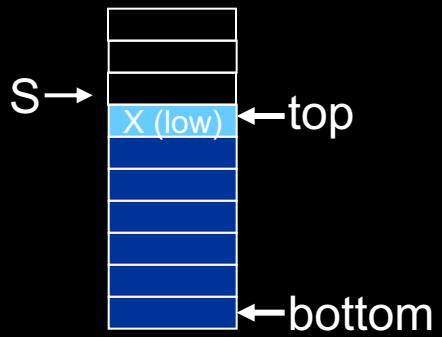


Transferring X to D

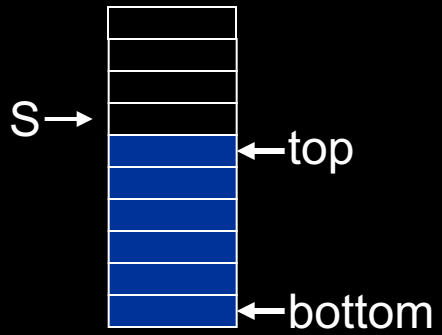
pshx



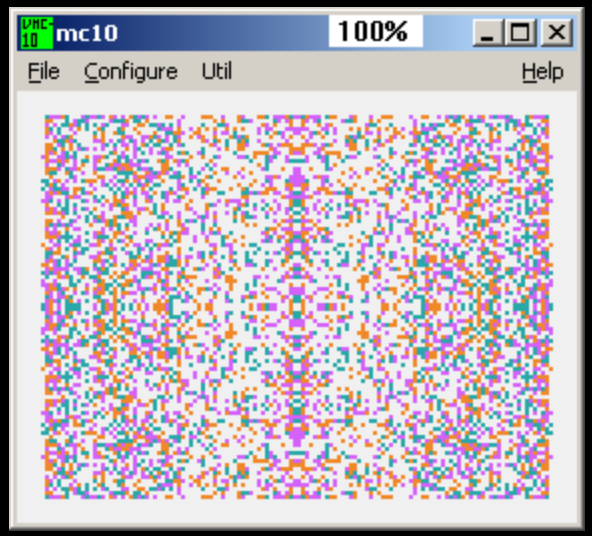
pula

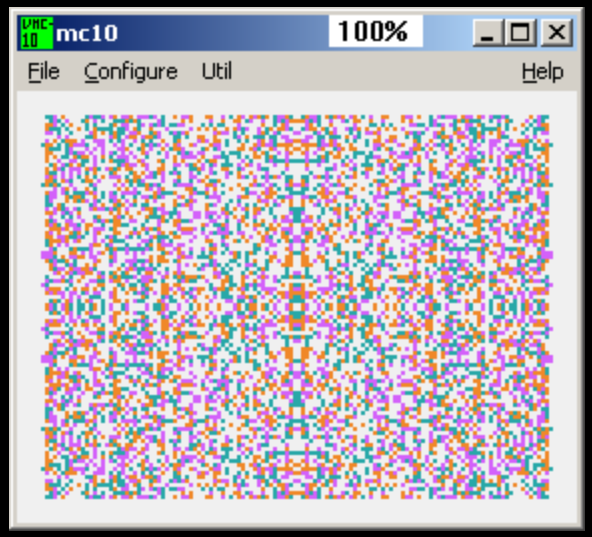


pulb



Example hi-res graphics (128x96) program similar to the “SPARKLE” program in the MC-10 BASIC manual





kscope.txt - Notepad

File Edit Format View Help

```
        .org 32768

        ldaa #100       ; set hi-res color graphics mode
        staa 49151
        ldx #16384      ; load x with start of screen address
clrnxt  clr ,x          ; clear it
        inx             ; bump address
        cpx #19456      ; compare with end of screen
        bne clrnxt
        ldx #49152      ; load x with start address of ROM table
nextpt
        ldaa ,x         ; get the color
        anda #3         ; force it between 0 and 3
        staa 255        ; store in on-chip memory at location 255
        addd 1,x        ; try to randomize the A,B coords...
        lsrd            ; keep A between 0-128
        eorb ,x         ; try to randomize B
        bsr plot        ; plot (    A,   B) with color at loc 255.
        negb
        addb #95
        bsr  plot       ; plot (    A,95-B) with color at loc 255.
        nega
        adda #127
        bsr  plot       ; plot (127-A,95-B) with color at loc 255.
        negb
        addb #95
        bsr plot        ; plot (127-A,   B) with color at loc 255.
        inx             ; bump x (will wraparound automatically at 65535)
        bra nextpt
; ENTRY A holds Xcoord, B holds Ycoord.
;       location 255 holds the color value.
plot
        pshx            ; preserve caller's x value
        pshb            ; preserve caller's b value
        psha            ; preserve caller's a value
        psha            ; push Xcoord on stack
        ldaa #32        ; 32 bytes per Y coordinate
        mul             ; D holds result.
        adda #64        ; add 64*256 = 16384 to D (start of screen).
        pshb            ; transfer D into X register
        psha            ;
        pulx            ;
        pulb            ; pull back the Xcoordinte
        tba             ; copy it also into the A register
        lsrb            ; divide Xc by 4.
        lsrb            ;
        abx             ; X = 16384 + Yc*32 + (Xc>>2)
                        ; now try to get proper 2-bit location mask
        ldab #%01000000 ; start with left bit.
        anda #3         ; leave if Xcoord is divisible by 4
        beq plotpt      ;
shiftl  lsrb            ; move mask over by two bits
        lsrb            ;
        deca            ; subtract 1 until Xcoord is divisible by 4
        bne shiftl      ;
plotpt  pshb            ; "b" holds proper multiplier -> save to stack
        tba             ; set next bit
        lslb            ;
        aba
        coma            ; a = NOT 3*B
        anda ,x         ; clear off those old bits
        staa ,x         ;
        pulb            ;
        ldaa 255        ; multiply "b" by color (0, 1, 2, or 3)
        mul
        orab ,x         ; now set the bits
        stab ,x
        pula            ; restore caller's a value
        pulb            ; restore caller's b value
        pulx            ; restore caller's x value
        rts             ; return

        .end
```

Tutorial #12
Flags

aba	beq	bra	daa	jmp	mul	rol	staa	tst
abx	bne	brn	dec	jsr		rola	stab	tsta
adca	bcc		deca		neg	rolb	std	tstb
adcb	bcs	cba	decb	ldaa	nega	ror	sts	
adda	bhi	clc	des	ldab	negb	rora	stx	wai
addb	bhs	cli	dex	ldd	nop	rorb	suba	
addd	blo	clr		lds		rti	subb	
anda	bls	clra	eora	ldx	oraa	rts	subd	
andb	bge	clrb	eorb	lsl	orab		swi	
asr	bgt	clv		lsla		sba		
asra	ble	cmpa	inc	lslb	psha	sbca	tab	
asrb	blt	cmpb	inca	lsld	pshb	sbcb	tap	
	bmi	cpx	incb	lsr	pshx	sec	tba	
bita	bpl	com	ins	lsra	pula	sei	tpa	
bitb	bvc	coma	inx	lsrb	pulb	sev	tsx	
bsr	bvs	comb		lsrd	pulx		txs	

Implicit Carry Flag Clearing

- The carry is cleared by
  - clr, clra, clrb
  - tst, tsta, tstb
- The carry is set by
  - com, coma, comb

Implicit oVerflow clearing

- The oVerflow flag is cleared by
  - ldaa, ldab, ldd, ldx, lds
  - staa, stab, std, stx, sts
  - anda, oraa, eora, bita
  - andb, orab, eorb, bitb
  - clr, clra, clrb
  - tst, tsta, tstb
  - com, coma, comb
  - tab, tba

Explicit Flag Clearing

- You can explicitly set and clear certain condition code flags.
  - Carry (sec, clc)
  - oVerflow (sev, clv)
  - Interrupt (sei, cli)

Using sec, clc, sev, clv

- The C and V flags are sometimes
  - set/cleared just before leaving a subroutine,
  - then inspected by bcc, bcs, bvc and bvs instructions after returning to the calling routine
- This helps serve as a fast way to provide status information to a calling routine.

Tutorial #13
Doing Nothing

~~aba~~	~~beq~~	~~bra~~	daa	~~jmp~~	~~mul~~	~~rol~~	~~staa~~	~~tst~~
~~abx~~	~~bne~~	brn	~~dec~~	~~jsr~~		~~rola~~	~~stab~~	~~tsta~~
~~adca~~	~~bcc~~		~~deca~~		~~neg~~	~~rolb~~	~~std~~	~~tstb~~
~~adcb~~	~~bcs~~	~~cba~~	~~decb~~	~~ldaa~~	~~nega~~	~~ror~~	~~sts~~	
~~adda~~	~~bhi~~	~~clc~~	~~des~~	~~ldab~~	~~negb~~	~~rora~~	~~stx~~	wai
~~addb~~	~~bhs~~	~~cli~~	~~dex~~	~~ldd~~	nop	~~rorb~~	~~suba~~	
~~addd~~	~~blo~~	~~clr~~		~~lds~~		rti	~~subb~~	
~~anda~~	~~bls~~	~~clra~~	~~eora~~	~~ldx~~	~~oraa~~	~~rts~~	~~subd~~	
~~andb~~	~~bge~~	~~clrb~~	~~eorb~~	~~lsl~~	~~orab~~		swi	
~~asr~~	~~bgt~~	~~clv~~		~~lsla~~		~~sba~~		
~~asra~~	~~ble~~	~~cmpa~~	~~inc~~	~~lslb~~	~~psha~~	~~sbca~~	~~tab~~	
~~asrb~~	~~blt~~	~~cmpb~~	~~inca~~	~~lsld~~	~~pshb~~	~~sbcb~~	~~tap~~	
	~~bmi~~	~~cpx~~	~~incb~~	~~lsr~~	~~pshx~~	~~sec~~	~~tba~~	
~~bita~~	~~bpl~~	~~com~~	~~ins~~	~~lsra~~	~~pula~~	~~sei~~	~~tpa~~	
~~bitb~~	~~bvc~~	~~coma~~	~~inx~~	~~lsrb~~	~~pulb~~	~~sev~~	~~tsx~~	
~~bsr~~	~~bvs~~	~~comb~~		~~lsrd~~	~~pulx~~		~~txs~~	

The NOP instruction

- This instruction does no operation
- It occupies one byte
- It is used as
  - a timewaster
  - padding for self-modifying code

The BRN instruction

- This instruction never branches.
- It is used as
  - a timewaster
  - padding for self-modifying code
  - a convenient way to hide an instruction in the place of the single-byte offset.
- It occupies two bytes.

Tutorial #14
Interrupts

~~aba~~	~~beq~~	~~bra~~	daa	~~jmp~~	~~mul~~	~~rol~~	~~staa~~	~~tst~~
~~abx~~	~~bne~~	~~brn~~	~~dec~~	~~jsr~~		~~rola~~	~~stab~~	~~tsta~~
~~adca~~	~~bcc~~		~~deca~~		~~neg~~	~~rolb~~	~~std~~	~~tstb~~
~~adcb~~	~~bcs~~	~~cba~~	~~decb~~	~~ldaa~~	~~nega~~	~~ror~~	~~sts~~	
~~adda~~	~~bhi~~	~~clc~~	~~des~~	~~ldab~~	~~negb~~	~~rora~~	~~stx~~	wai
~~addb~~	~~bhs~~	~~cli~~	~~dex~~	~~ldd~~	~~nop~~	~~rorb~~	~~suba~~	
~~addd~~	~~blo~~	~~clr~~		~~lds~~		rti	~~subb~~	
~~anda~~	~~bls~~	~~clra~~	~~eora~~	~~ldx~~	~~oraa~~	~~rts~~	~~subd~~	
~~andb~~	~~bge~~	~~clrb~~	~~eorb~~	~~lsl~~	~~orab~~		swi	
~~asr~~	~~bgt~~	~~clv~~		~~lsla~~		~~sba~~		
~~asra~~	~~ble~~	~~cmpa~~	~~inc~~	~~lslb~~	~~psha~~	~~sbca~~	~~tab~~	
~~asrb~~	~~blt~~	~~cmpb~~	~~inca~~	~~lsld~~	~~pshb~~	~~sbcb~~	~~tap~~	
	~~bmi~~	~~cpx~~	~~incb~~	~~lsr~~	~~pshx~~	~~sec~~	~~tba~~	
~~bita~~	~~bpl~~	~~com~~	~~ins~~	~~lsra~~	~~pula~~	~~sei~~	~~tpa~~	
~~bitb~~	~~bvc~~	~~coma~~	~~inx~~	~~lsrb~~	~~pulb~~	~~sev~~	~~tsx~~	
~~bsr~~	~~bvs~~	~~comb~~		~~lsrd~~	~~pulx~~		~~txs~~	

Interrupts

- Caused by various conditions
- All but two can be disabled by setting the Interrupt condition code flag.
- When an interrupt occurs
  - the Interrupt flag is set
  - all registers are pushed on the stack
  - program execution jumps to a specified location in memory

Interrupts

- There are seven kinds of interrupts (in ascending order of priority).

Type	Address	Description
– SCI	16896	serial communication interface
– TOF	16899	timer overflow
– OCF	16902	output compare flag
– ICF	16905	input capture flag
– IRQ1	16908	interrupt-request 1
– SWI	16911	software interrupt
– NMI	16914	non-maskable interrupt

- The Virtual MC-10 only emulates the SWI, OCF and TOF interrupts

SCI

- The Serial Communications Interface is not used by the MC-10.
- It is not emulated by the Virtual MC-10.

TOF

- The timer overflow flag is set when the counter contains all ones (65535), and will jump to its address if the Interrupt flag is clear.
- TOF is cleared by reading from memory location 9 (the counter).
- The Virtual MC-10 emulates this interrupt (I think).

OCF

- The output capture flag IS used by the MC-10 and Virtual MC-10.
- It gets set when the output compare register matches the free-running counter and will jump to its address if the Interrupt flag is clear.
- You can enable it by setting bit 2 of the TCSR register (address 8)
- OCF is cleared by reading the TCSR and then writing to the output compare register (locations 11 or 12) or during reset.
- The Virtual MC-10 emulates this interrupt.

ICF

- This interrupt is used when bit 0 of port 2 is configured as an input.
- The MC-10 uses it instead as an output to the RS232C communication port, so this isn't really used.
- The Virtual MC-10 doesn't use this interrupt.

IRQ1

- This interrupt is not implemented by the MC-10.
- The IRQ1 pin is tied through a pull-up resistor to +5V.

SWI

- This is known as the software interrupt.
- It is explicitly called by the user and will always jump to its address and it cannot be disabled by the interrupt flag
- It is emulated by the Virtual MC-10

NMI

- The non-maskable interrupt can be triggered from the expansion slot in the back of the MC-10.
- It is not disabled by the Interrupt flag.
- It jumps to location 16896.
- The Virtual MC-10 does not make use of this interrupt

The WAI instruction

- This instruction will save the registers to the stack, then wait for an interrupt.
- This is good for waiting for a timer to expire if you have nothing else to do

The RTI instruction

- When an interrupt is called it saves all the registers on the stack.
- The RTI instruction will restore the registers and resume control back to the main program.
- It is equivalent to the following series of instructions: pula, tap, pulb, pula, pulx, rts.

Sample Program

- An audio recording was taken from the “The Princess Bride” and run through a low-pass filter.
- The resulting waveform was truncated so that the speaker would be energized when the audio went “above zero” and de-energized when the audio goes “below zero” – not great for sound quality, but that’s all the MC-10 can do.
- The results were encoded differentially, by recording how long it took the audio to cross zero – providing reasonable compression for use with the MC-10.

Sample Program

(Plays audio in the background)

tut14a.txt - Notepad

File Edit Format View Help

```
sbyte   .equ 20480
sound   .equ 20481
        .org 20483      ;tell tasm to start at 20483

        clr sbyte       ;clear sound byte
        ldx #taunt      ;load start of movie quote
        stx sound       ;store into sound pointer
        ldaa #126       ;load 'jmp' instruction opcode
        staa 16902      ;store into TOF vector
        ldx #inter      ;load interrupt location
        stx 16903       ;store as address to jump to
        cli             ;enable interrupts
        ldaa #8         ;enable TOF
        staa 8
        rts             ;go back to BASIC...

inter   ldx sound       ;increment sound pointer
        inx             ;
        cpx #taunte     ;see if at end of sound segment
        bls savsnd      ;
        ldx #taunt      ;restart the taunt if at end
savsnd  stx sound       ;
        ldaa sbyte      ;get the sound byte
        eora #%10000000 ;flip the sound bit
        staa sbyte      ;store for safekeeping
        staa 49151      ;store into speaker/video select
        ldaa ,x         ;get amount to sleep by
        ldab #75        ;multiply by delay value (11.8kHz sample rate = 0.89MHz / 75)
        mul
        addd 9          ;add the offset to the timer
        bita 8          ;read to clear the TOF (but don't do anything with it).
        std 11          ;store into output compare register
        rti             ;return from interrupt (back to BASIC)
```

Note: This program is large, due to the amount of data, so don't forget to set the load and EXEC addresses to 20483 If you try this program.