

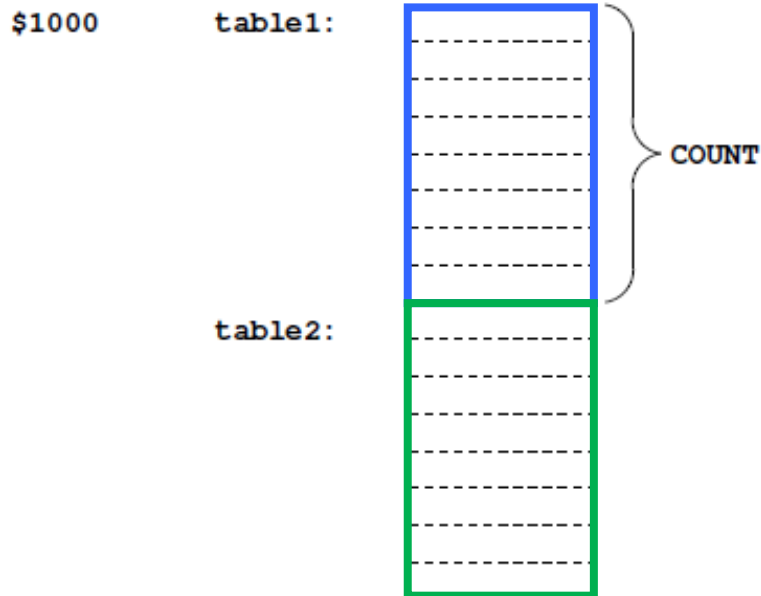
- **Writing Assembly Language Programs**
 - Use flow charts to lay out structure of program
 - Use common flow structures
 - If-then
 - If-then-else
 - Do-while
 - While
 - Plan structure of data in memory
 - Top-down design
 - Plan overall structure of program
 - Work down to more detailed program structure
 - Implement structure with instructions
 - Optimize program to make use of instruction efficiencies
 - Do not sacrifice clarity for efficiency or speed

- **Input and Output Ports**
 - How to get data into and out of the MC9S12

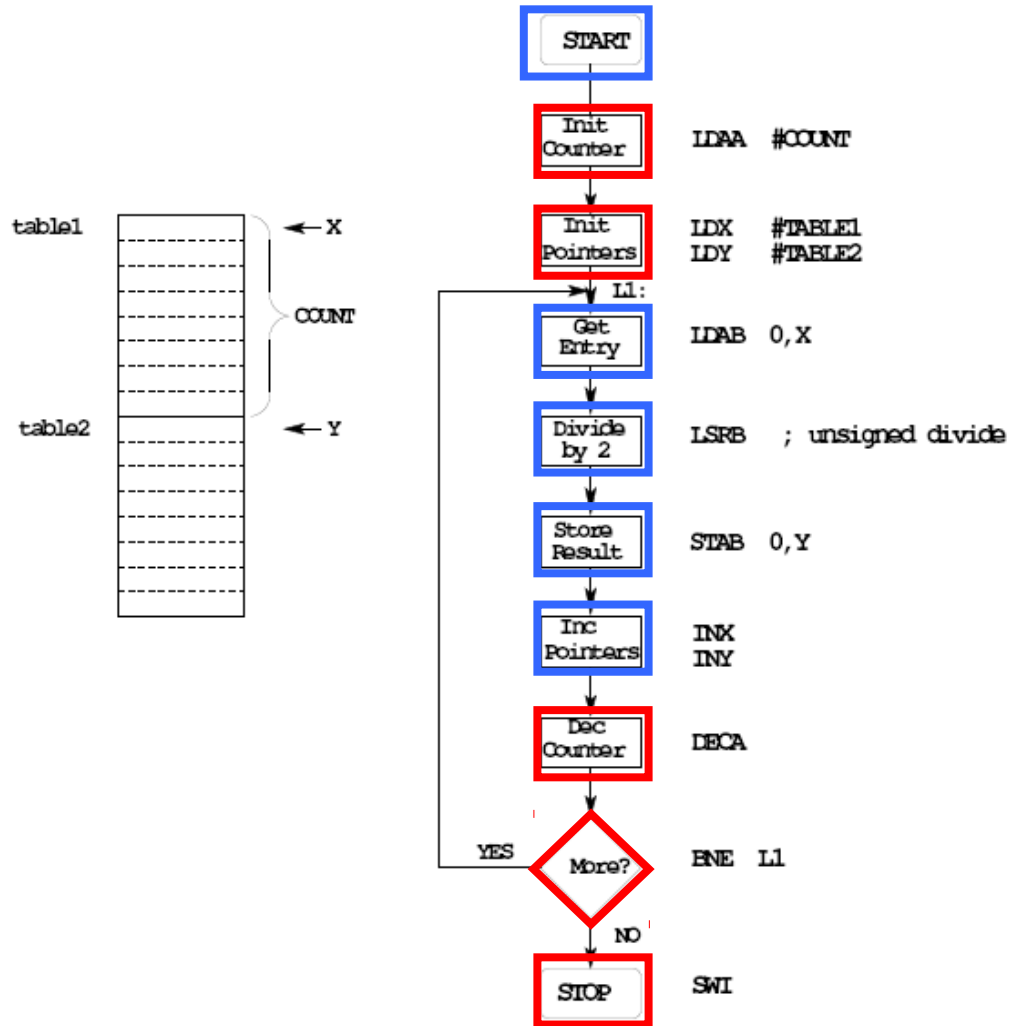
Example Program: Divide a table of data by 2

Problem: Start with a table of data. The table consists of 5 values. Each value is between 0 and 255. Create a new table whose contents are the original table divided by 2.

1. Determine where code and data will go in memory.
Code at \$2000, data at \$1000.
2. Determine type of variables to use.
Because data will be between 0 and 255, can use unsigned 8-bit numbers.
3. Draw a picture of the data structures in memory:



4-7. Add code to implement blocks:



8. Write the program:

; Program to divide a table by two
; and store the results in memory

prog: equ \$2000
data: equ \$1000

count: equ 5

org prog ; *Set program counter to 0x2000*
ldaa #count ; *Use A as counter*
ldx #table1 ; *Use X as data pointer to table1*
ldy #table2 ; *Use Y as data pointer to table2*
l1: ldab 0,x ; *Get entry from table1*
lsrb ; *Divide by two (unsigned)*
stab 0,y ; *Save in table2*
inx ; *Increment table1 pointer*
iny ; *Increment table2 pointer*
deca ; *Decrement counter*
bne l1 ; *Counter != 0 => more entries*
 ; *to divide*
swi ; *Done*

org data
table1: dc.b \$07,\$c2,\$3a,\$68,\$f3
table2: ds.b count

9. Advanced: Optimize program to make use of instructions set efficiencies:

*; Program to divide a table by two
; and store the results in memory*

```
prog:    equ  $1000
data:    equ  $2000

count:   equ  5

                org  prog      ; Set program counter to 0x1000
                ldaa #count    ; Use A as counter
                ldx  #table1    ; Use X as data pointer to table1
                ldy  #table2    ; Use Y as data pointer to table2
l1:        ldab  1,x+          ; Get entry from table1; then inc
                ; pointer
                lsrb              ; Divide by two (unsigned)
                stab  1,y+      ; Save in table2; then inc ptr.
                dbne a,l1      ; Decrement counter; if not 0,
                ; more to do
                swi              ; Done

                org  data
table1:    dc.b  $07,$c2,$3a,$68,$f3
table2:    ds.b  count
```

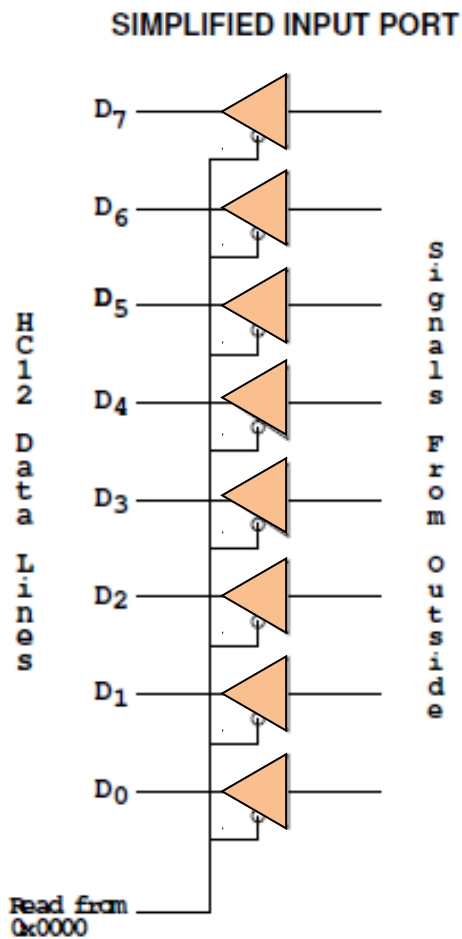
TOP-DOWN PROGRAM DESIGN

- PLAN DATA STRUCTURES IN MEMORY
- START WITH A LARGE PICTURE OF THE PROGRAM STRUCTURE
- WORK DOWN TO MORE DETAILED STRUCTURE
- TRANSLATE STRUCTURE INTO CODE
- OPTIMIZE FOR EFFICIENCY

DO NOT SACRIFICE CLARITY FOR EFFICIENCY

Input and Output Ports

- How do you get data into a computer from the outside?

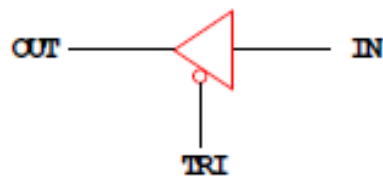


Any read from address \$0000 gets signals from outside

LDAA \$00

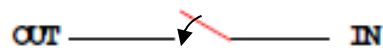
Puts data from outside into accumulator A.

Data from outside looks like a memory location.

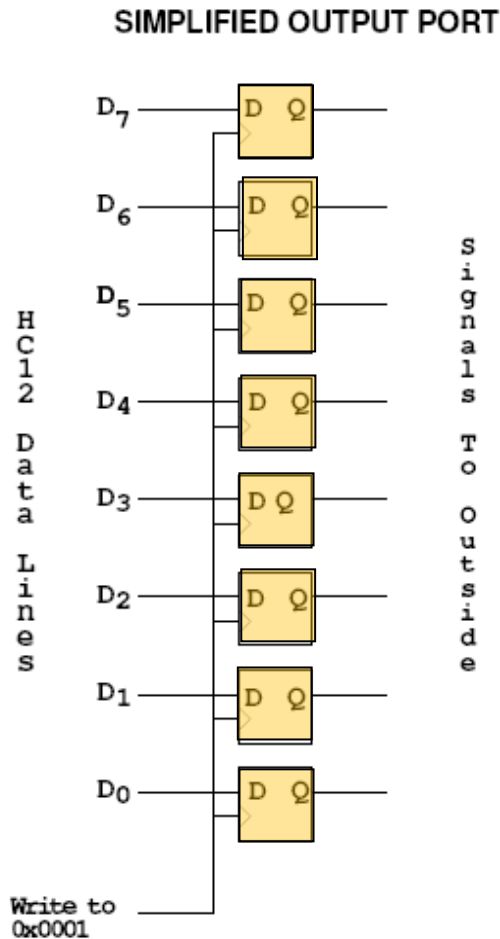


A Tri-State Buffer acts like a switch

If TRI is not active, the switch is open: OUT will not be driven by IN
Some other device can drive OUT



- How do you get data out of computer to the outside?



Any write to address \$01 latches data into FF, so data goes to external pins

MOVB #\$AA,\$01

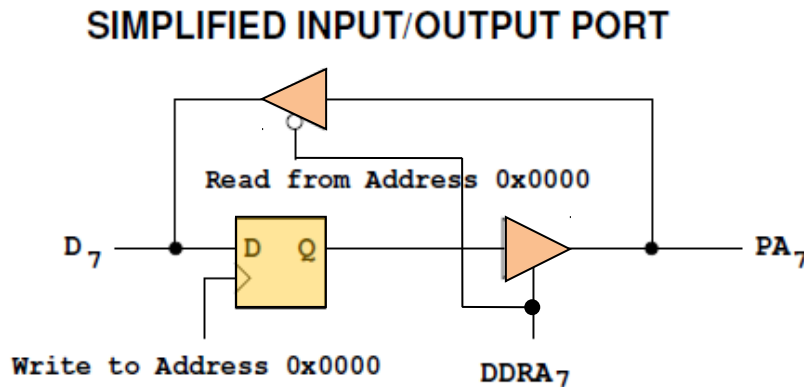
Puts \$AA on the external pins

When a port is configured as output and you read from that port, the data you read is the data which was written to that port:

MOVB #\$AA, \$01
LDAA \$01

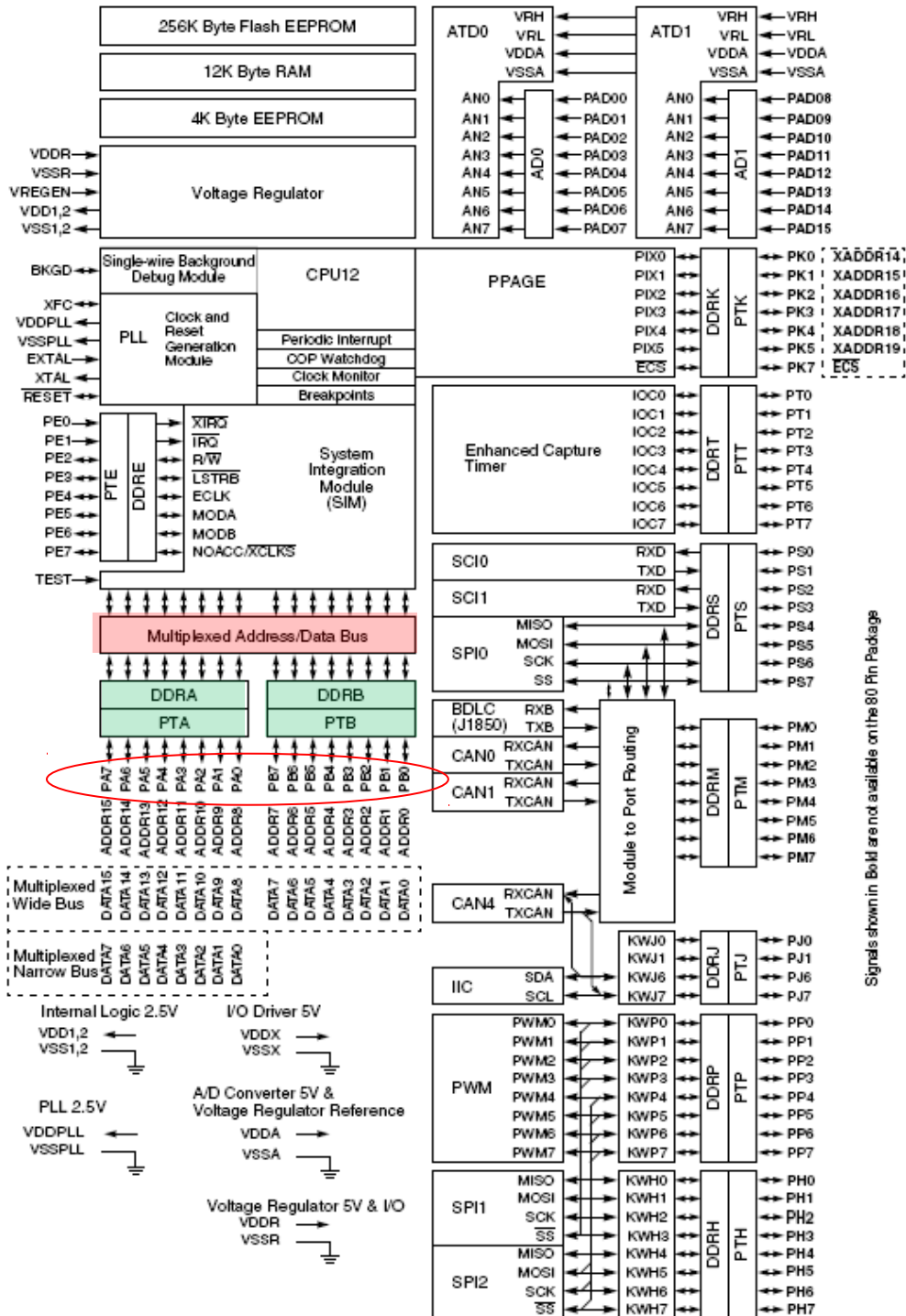
Accumulator A will have \$AA after this

- Most I/O ports on MC9S12 can be configured as either input or output



- A write to address 0x0000 writes data to the flip-flop
A read from address 0x0000 reads data on pin
- If Bit 7 of DDRA is 0, the port is an input port. Data written to flip-flop does not get to pin though tri-state buffer
- If Bit 7 of DDRA is 1, the port is an output port. Data written to flip-flop does get to pin though tri-state buffer
- DDRA (Data Direction Register A) is located at 0x0002

Figure 1-1 MC9S12DT256 Block Diagram



Ports on the HC12

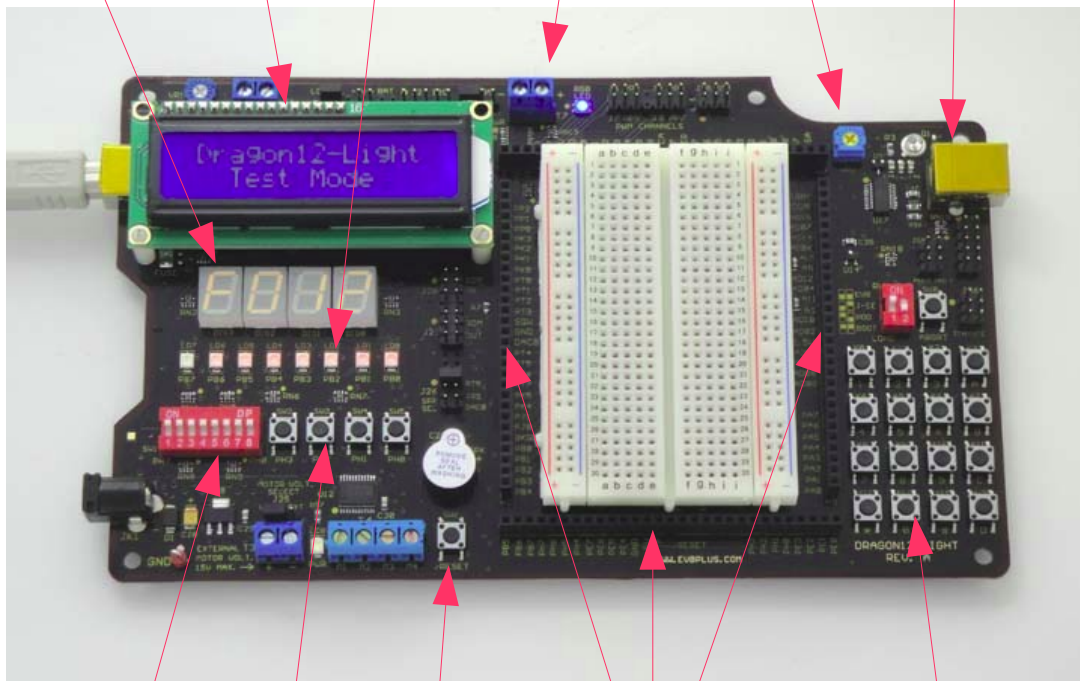
- How do you get data out of computer to the outside?
- A Port on the MC9S12 is a device that the MC9S12 uses to control some hardware.
- Many of the MC9S12 ports are used to communicate with hardware outside of the MC9S12.
- The MC9S12 ports are accessed by the MC9S12 by reading and writing memory locations **\$0000** to **\$03FF**.
- Some of the ports we will use in this course are **PORTA**, **PORTB**, **PTJ** and **PTP**:
 - PORTA is accessed by reading and writing address \$0000.
 - DDRA is accessed by reading and writing address \$0002.
 - PORTB is accessed by reading and writing address \$0001.
 - DDRB is accessed by reading and writing address \$0003.
 - PTJ is accessed by reading and writing address \$0268.
 - DDRJ is accessed by reading and writing address \$026A.
 - PTP is accessed by reading and writing address \$0258.
 - DDRP is accessed by reading and writing address \$025A.
- On the DRAGON12-Plus EVB, eight LEDs and four seven-segment LEDs are connected to PTB.

16x2 Character LCD Motor Driver Potentiometer connected to ATD

7-Segment LEDs

8 LEDs

Serial Comm. Port



8 DIP Switches

Reset Button

Debounced Keyboard

4 Push buttons

Connections to Ports

-Before you can use the eight individual LEDs or the seven-segment LEDs, you need to enable them:

- Bit 1 of PTJ must be low to enable the eight individual LEDs.

* To make Bit 1 of PTJ low, you must first make Bit 1 of PTJ an output by writing a 1 to Bit 1 of DDRJ.

* Next, write a 0 to Bit 1 of PTJ.

- Bits 3-0 of PTP are used to enable the four seven-segment LEDs.

- To use the seven-segment LEDs, first write 1's to Bits 3-0 of DDRP to make Bits 3-0 of PTP outputs.

* A low PTP0 enables the left-most (Digit 3) seven-segment LED

* A low PTP1 enables the second from the left (Digit 2) seven-segment LED

* A low PTP2 enables the third from the left (Digit 1) seven-segment LED

* A low PTP3 enables the right-most (Digit 0) seven-segment LED

– To use the eight individual LEDs and turn off the seven-segment LEDs, write ones to Bits 3-0 of PTP, and write a 0 to Bit 1 of PTJ:

```
BSET DDRP,#$0F ; Make PTP3 through PTP0 outputs  
BSET PTP,#$0F ; Turn off seven-segment LEDs  
BSET DDRJ,#$02 ; Make PTJ1 output  
BCLR PTJ,#$02 ; Turn on individual LEDs
```

- On the DRAGON12-Plus EVB, the LCD display is connected to PTK

- When you power up or reset the MC9S12, PORTA, PORTB, PTJ and PTP are input ports(!).

- You can make any or all bits of PORTA, PORTB PTP and PTJ outputs by writing a 1 to the corresponding bits of their *Data Direction Registers (DDRs)*.

– You can use DDebug-12 to manipulate the IO ports on the 68HCS12

- * To make PTB an output, use MM to change the contents of address \$0003 (DDRB) to an \$FF.

- * You can now use MM to change contents of address \$0001 (PORTB), which changes the logic levels on the PORTB pins.

- * If the data direction register makes the port an input, you can use MD to display the values on the external pins.

Using Port A of the 68HC12

To make a bit of Port A an **output** port, write a 1 to the corresponding bit of DDRA (address 0x0002).

To make a bit of Port A an **input** port, write a 0 to the corresponding bit of DDRA.

On reset, DDRA is set to \$00, so Port A is an input port(!).

DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0	
Reset	0	0	0	0	0	0	0	\$0002

For example, to make bits 3–0 of Port A inputs, and bits 7 – 4 outputs, write a *0xF0* to DDRA.

To send data to the output pins, write to PORTA (address 0x0000). When you read from PORTA input pins will return the value of the signals on them (0 ⇒ 0V, 1 ⇒ 5V); output pins will return the value written to them.

	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	
Reset	-	-	-	-	-	-	-	-	\$0000

Port B works the same, except DDRB is at address 0x0003 and PORTB is at address 0x0001.

*; A simple program to make PORTA output and PORTB
; input, then read the signals on PORTB and write these
; values out to PORTA*

prog: equ \$2000

PORTA: equ \$00

PORTB: equ \$01

DDRA: equ \$02

DDRB: equ \$03

org prog

movb #\$ff,DDRA ; *Make PORTA output*

movb #\$00,DDRB ; *Make PORTB input*

ldaa PORTB

staa PORTA

swi

- Because DDRA and DDRB are in consecutive address locations you could make PORTA an output and PORTB and input in one instruction:

movw #\$ff00,DDRA ; *FF -> DDRA, 00 -> DDRB*

GOOD PROGRAMMING STYLE

1. Make programs easy to read and understand.
 - Use comments
 - Do not use tricks
2. Make programs easy to modify
 - Top-down design
 - Structured programming – no spaghetti code
 - Self contained subroutines
3. Keep programs short BUT do not sacrifice items 1 and 2 to do so

TIPS FOR WRITING PROGRAMS

1. Think about how data will be stored in memory.
 - Draw a picture
2. Think about how to process data
 - Draw a flowchart
3. Start with big picture. Break into smaller parts until reduced to individual instructions
 - Top-down design
4. Use names instead of numbers

Another Example of an Assembly Language Program

- Find the average of the numbers in an array of data.
- The numbers are 8-bit unsigned numbers.
- The address of the first number is \$E000 and the address of the final number is \$E01F. There are 32 numbers.
- Save the result in a variable called *answer* at address \$1000.

Start by drawing a picture of the data structure in memory:

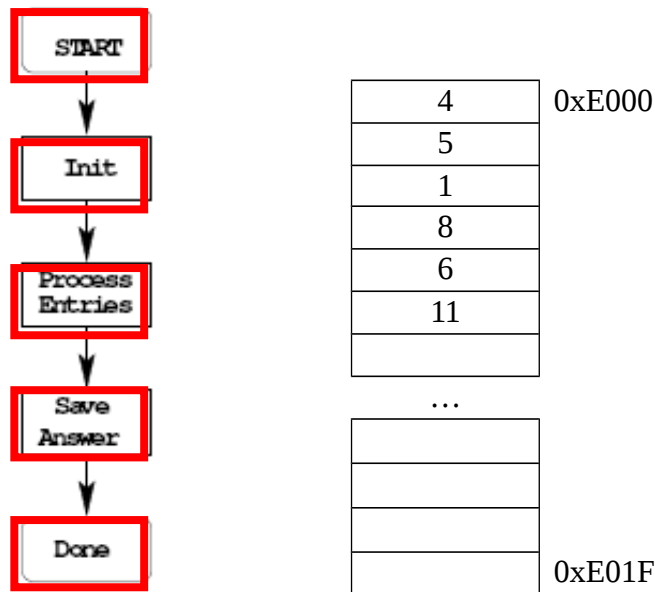
**FIND THE AVERAGE OF NUMBERS IN ARRAY FROM
0XE000 TO 0XE01F**

Treat numbers as 8-bit unsigned numbers

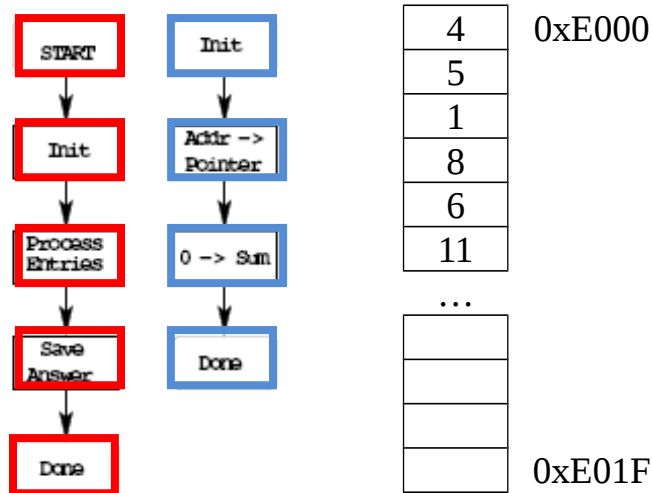
4	0xE000
5	
1	
8	
6	
11	
...	
	0xE01F

Start with the big picture

Find average of 8-bit numbers in array from 0xE000 to 0xE01F



Add details to blocks



Decide on how to use CPU registers for processing data

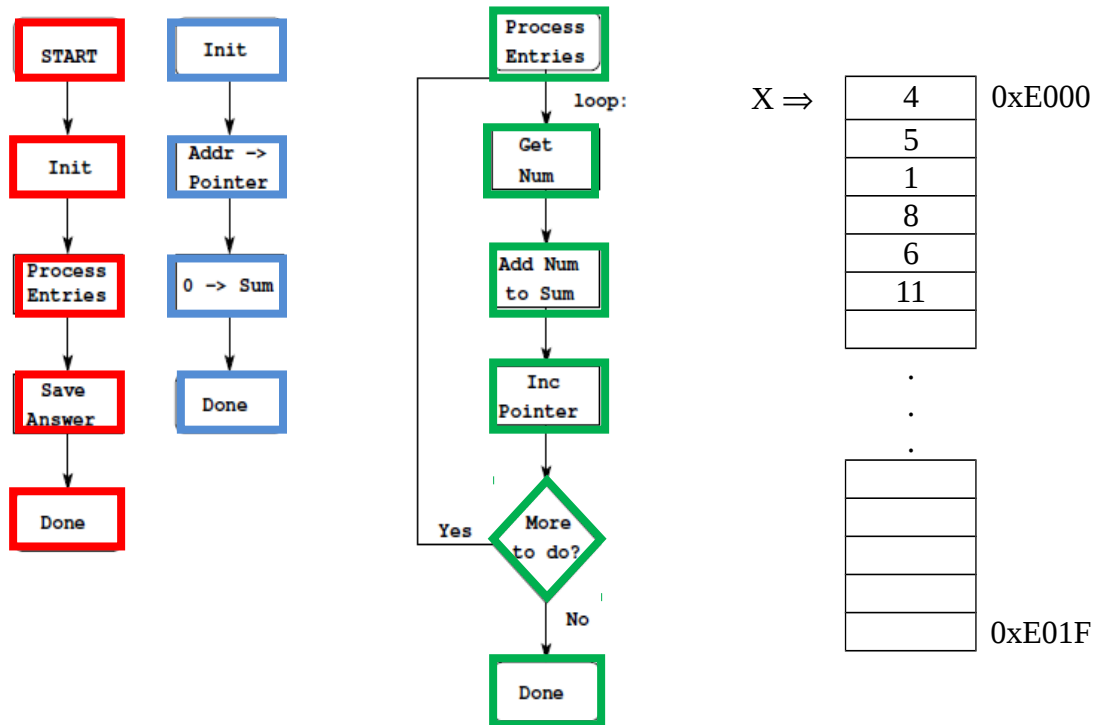
Find average of 8-bit numbers in array from 0xE000 to 0xE01f

Sum: 16-bit register. Can use D or Y

No way to add 8-bit number to D

Can use ABY to add 8-bit number to Y: (B)+(Y) ⇒ Y

Add more details: Expand another block



More details: How to tell when program reaches end of array

How to check if more to do?
If $X < 0xE020$, more to do.

BLT or BLO; Addresses are unsigned, so BLO

How to find average? Divide by LEN

To divide, use IDIV

```
TFR Y,D      ; transfer Y to D
LDX #LEN     ; load divisor in X
IDIV         ; (D)/(X) => X
```

Write program

;Program to average 32 numbers in a memory array

prog: equ \$2000

data: equ \$1000

array: equ \$E000

len: equ 32

```
    org prog
    ldx #array    ; initialize pointer
    ldy #0        ; initialize sum to 0
loop: ldab 0,x    ; get number
    aby          ; add to sum
    inx          ; point to next entry
    cpx #(array+len) ; more to process?
    blo loop     ; if so, process
    tfr y,d      ; to divide, need dividend in D
    ldx #len     ; to divide, need divisor in X
    idiv         ; D/X quotient in x, remainder in D
    stx answer   ; done – save answer
    swi

    org data
    answer: ds.w 1 ; reserve 16-bit word for answer
```

- Important: Comment program so it is easy to understand.

The assembler output for the above program

Freescale HC12-Assembler

(c) Copyright Freescale 1987-2009

Abs.	Rel.	Loc	Obj. code	Source line
1	1			;Program to average 32 numbers in a memory array
2	2			
3	3	0000 2000		prog: equ \$2000
4	4	0000 1000		data: equ \$1000
5	5			
6	6	0000 E000		array: equ \$E000
7	7	0000 0020		len: equ 32
8	8			
9	9			org prog
10	10			
11	11	a002000 CEE0 00		ldx #array ; initialize pointer
12	12	a002003 CD00 00		ldy #0 ; initialize sum to 0
13	13	a002006 E600	loop:	ldab 0,x ; get number
14	14	a002008 19ED		aby ; odd - add to sum
15	15	a00200A 08		inx ; point to next entry
16	16	a00200B 8EE0 20		cpx #(array+len) ; more to process?
17	17	a00200E 25F6		blo loop ; if so, process
18	18			
19	19	a002010 B764		tfr y,d ; To divide, need dividend
20	20	a002012 CE00 20		ldx #len ; To divide, need divisor
21	21	a002015 1810		idiv ; D/X quotient in X,
remainder				
22	22	a002017 7E10 00		stx answer ; done -- save answer
23	23	a00201A 3F		swi
24	24			
25	25			org data
26	26	a001000		answer: ds.w 1 ; reserve 16-bit word for 27
27				
28	28			

Here is the .s19 file:

```
S11E2000CEE000CD0000E60019ED088EE02025F6B764CE002018107E10003FAB  
S9030000FC
```