Lecture 10

February 8, 2012

Using the Stack and the Stack Pointer

- The Stack and the Stack Pointer

- The stack is an area of memory used for temporary storage

- The stack pointer points to the last byte pushed onto the stack

- Some instructions which use the stack

- How data is pushed onto and pulled off of the stack.

- Subroutines and the Stack

- An example of a simple subroutine

- Using a subroutine with PORTA to make a binary counter on LEDs
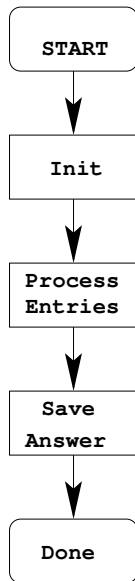
GOOD PROGRAMMING STYLE

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 $2000.

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

FIND 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

START

Init

Process
Entries

Save
Answer

Done

|	
4	0xE000
5	
1	
8	
6	
11	
	0xE01F

<u>Add details to blocks</u>

SUM ODD 8-BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f

START → Init → Process Entries → Save Answer → Done

Init → Addr -> Pointer → 0 -> Sum → Done
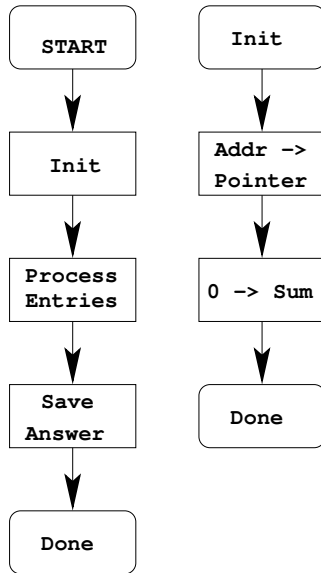
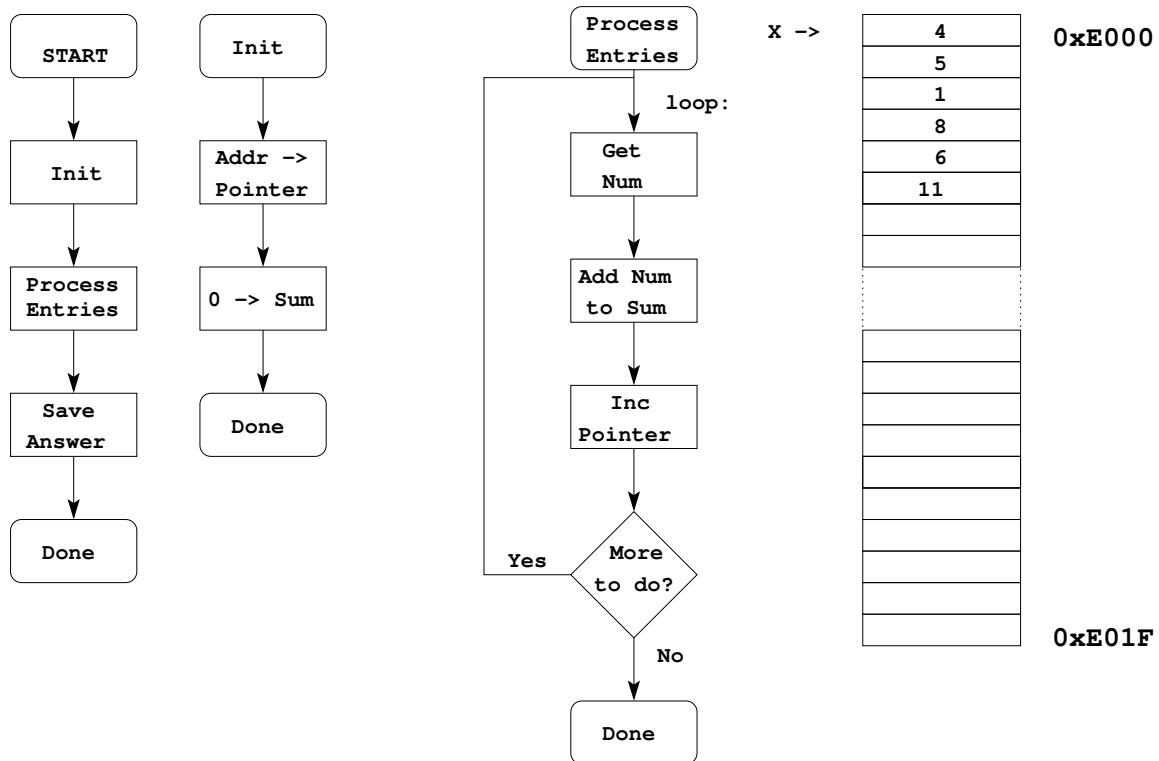|	
4	0xE000
5	
1	
8	
6	
11	
	0xE01F

Decide on how to use CPU registers for processing data

FIND AVERAGE OF 8-BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f

START	Init
Init	Addr -> Pointer
Process Entries	0 -> Sum
Save Answer	Done
Done	
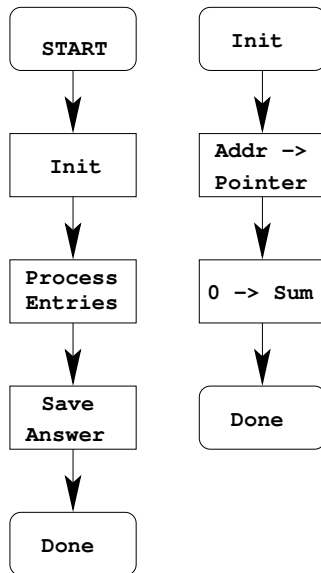
4	0xE000
5	
1	
8	
6	
11	
	0xE01F

Pointer: X or Y -- use X

**Sum: 16-bit register
        D or Y**

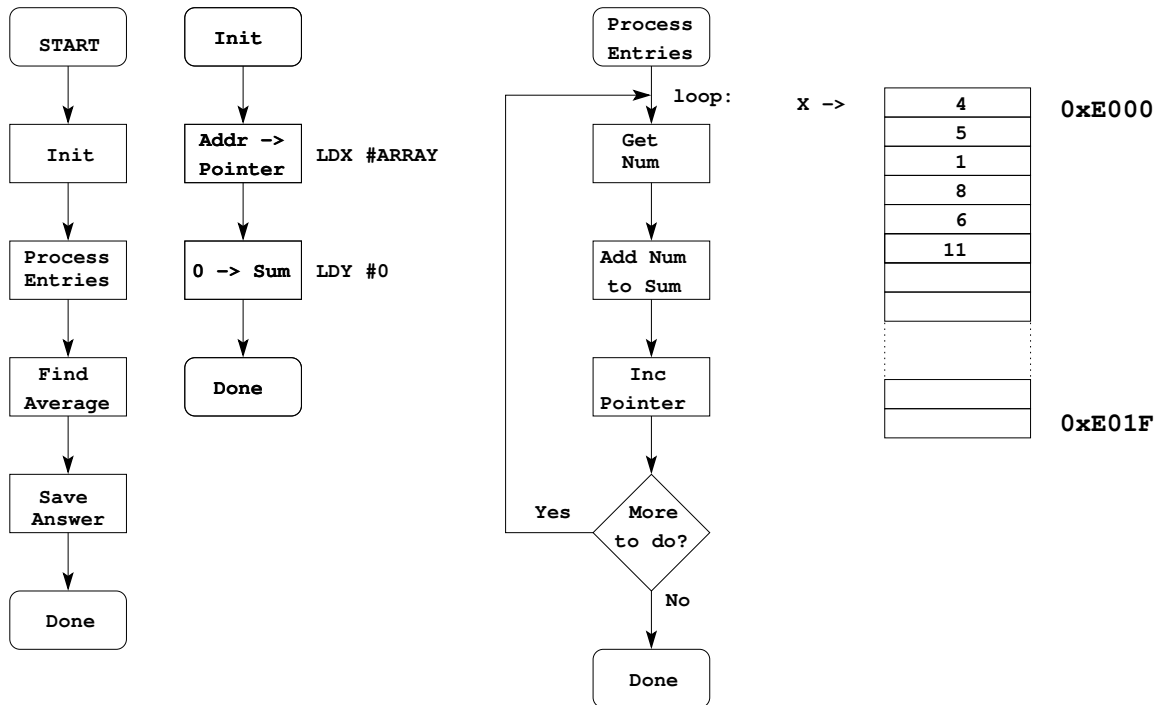**No way to add 8-bit number to D
Can use ABY to add 8-bit number to Y**

6

Add more details: Expand another block

FIND AVERAGE OF 8-BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f

```
  ┌─────────┐        ┌─────────┐              ┌─────────┐     X ->  ┌──────┐
  │  START  │        │  Init   │              │ Process │          │   4  │  0xE000
  └────┬────┘        └────┬────┘              │ Entries │          ├──────┤
       │                  │                   └────┬────┘          │   5  │
       ▼                  ▼                        │  loop:        ├──────┤
  ┌─────────┐        ┌─────────┐              ┌─────▼───┐          │   1  │
  │  Init   │        │ Addr -> │              │  Get    │          ├──────┤
  └────┬────┘        │ Pointer │              │  Num    │          │   8  │
       │             └────┬────┘              └────┬────┘          ├──────┤
       ▼                  │                        │               │   6  │
  ┌─────────┐             ▼                        ▼               ├──────┤
  │ Process │        ┌─────────┐              ┌─────────┐          │  11  │
  │ Entries │        │ 0 -> Sum│              │ Add Num │          ├──────┤
  └────┬────┘        └────┬────┘              │ to Sum  │          │      │
       │                  │                   └────┬────┘          ├──────┤
       ▼                  ▼                        │               │      │
  ┌─────────┐        ┌─────────┐              ┌─────▼───┐          │ : :  │
  │  Save   │        │  Done   │              │  Inc    │          ├──────┤
  │ Answer  │        └─────────┘              │ Pointer │          │      │
  └────┬────┘                                 └────┬────┘          ├──────┤
       │                                           │               ...
       ▼                                           ▼               ├──────┤
  ┌─────────┐                          Yes  ◇ More    ◇            │      │  0xE01F
  │  Done   │                         ◄─────  to do?              └──────┘
  └─────────┘                                  ◇        ◇
                                                 │ No
                                                 ▼
                                            ┌─────────┐
                                            │  Done   │
                                            └─────────┘
```

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

FIND AVERAGE OF 8-BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f

```
START                 Init                          Process
                                                    Entries

Init              Addr ->   LDX #ARRAY              Get         loop:        X ->    | 4  |   0xE000
                  Pointer                           Num                              | 5  |
                                                                                     | 1  |
Process           0 -> Sum  LDY #0                  Add Num                          | 8  |
Entries                                             to Sum                           | 6  |
                                                                                     | 11 |
Find              Done                              Inc
Average                                             Pointer                          |    |

Save                                        Yes     More                            |    |   0xE01F
Answer                                             to do?

Done                                                 No

                                                    Done
```

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   ; dividend in D
   LDX #LEN   ; divisor in X
   IDIV

Convert blocks to assembly code

FIND AVERAGE OF 8−BIT NUMBERS IN ARRAY FROM 0xE000 TO 0xE01f
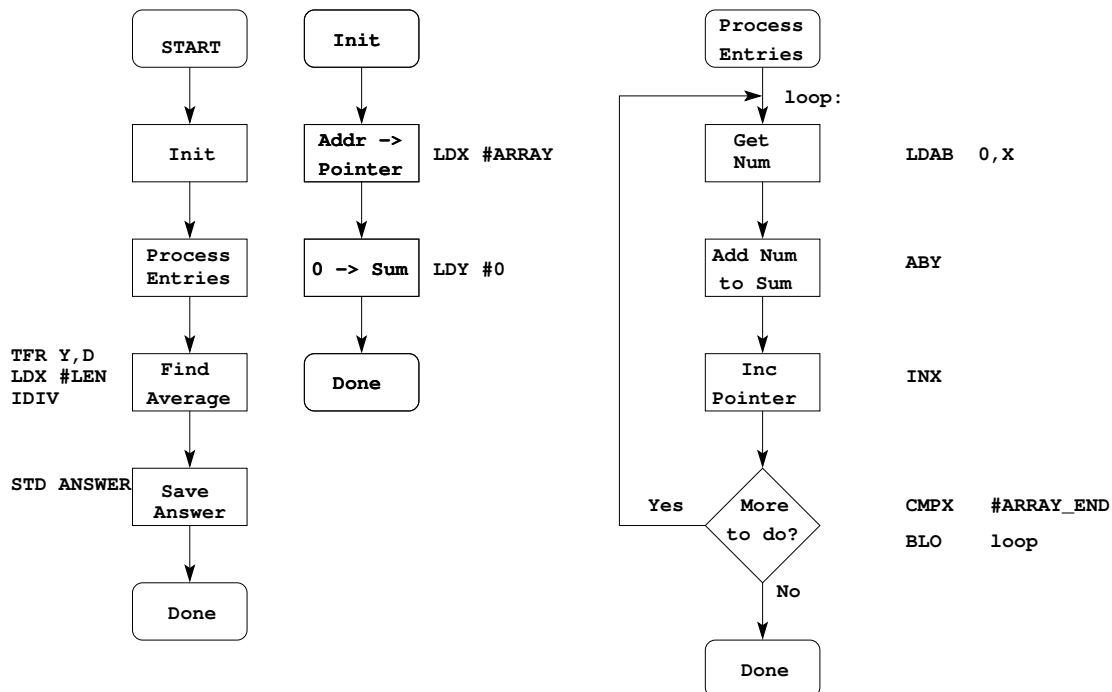
<u>Write program</u>

```
;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                     ; odd - 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 dividen
   20   20  a002012 CE00 20               ldx     #len            ; To divide, need divisor
   21   21  a002015 1810                  idiv                    ; D/X  quotient in X, rem
   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
```

And here is the .s19 file:

```
S11E2000CEE000CD0000E60019ED088EE02025F6B764CE002018107E10003FAB
S9030000FC
```

The Stack and tHe Stack Pointer

- Sometimes it is useful to have a region of memory for temporary storage, which does not have to be allocated as named variables.

- When we use subroutines and interrupts it will be essential to have such a storage region.

- Such a region is called a *Stack*.

- The *Stack Pointer* (SP) register is used to indicate the location of the last item put onto the stack.

- When you put something onto the stack (push onto the stack), the SP is decremented *before* the item is placed on the stack.

- When you take something off of the stack (pull from the stack), the SP is incremented *after* the item is pulled from the stack.

- Before you can use a stack you have to initialize the Stack Pointer to point to one value higher than the highest memory location in the stack.

- For the MC9S12 put the stack at the top of the data space

  - For most programs, use $1000 through $2000 for data.
  - For this region of memory, initialize the stack pointer to $2000.
  - If you need more space for data and the stack, and less for your program, move the program to a higher address, and use this for the initial value of the stack pointer.

  .

- Use the LDS (Load Stack Pointer) instruction to initialize the stack point.

- The LDS instruction is usually the first instruction of a program which uses the stack.

- The stack pointer is initialized only one time in the program.

- For microcontrollers such as the MC9S12, it is up to the programmer to know how much stack his/her program will need, and to make sure enough space is allocated for the stack. If not enough space is allocated the stack can overwrite data and/or code, which will cause the program to malfunction or crash.

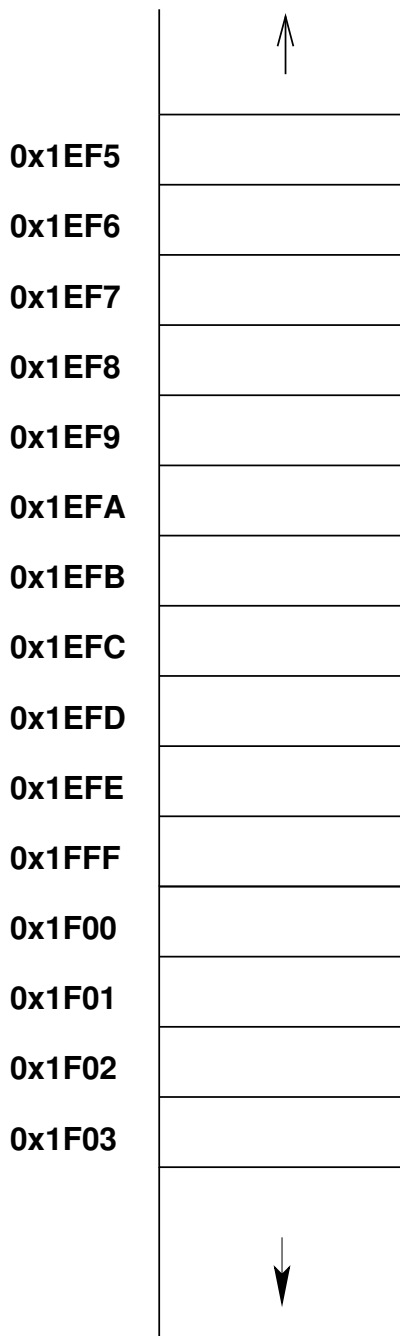The stack is an array of memory dedicated to temporary storage

SP points to location last item placed in block

SP decreases when you put item on stack

SP increases when you pull item from stack

For MC9S12, use 0x2000 as initial SP:

```
STACK:    EQU    $2000
          LDS    #STACK
```

0x1EF5

0x1EF6

0x1EF7

0x1EF8

0x1EF9

0x1EFA

0x1EFB

0x1EFC

0x1EFD

0x1EFE

0x1FFF

0x1F00

0x1F01

0x1F02

0x1F03

A   B

D

X

Y

SP

PC

CCR
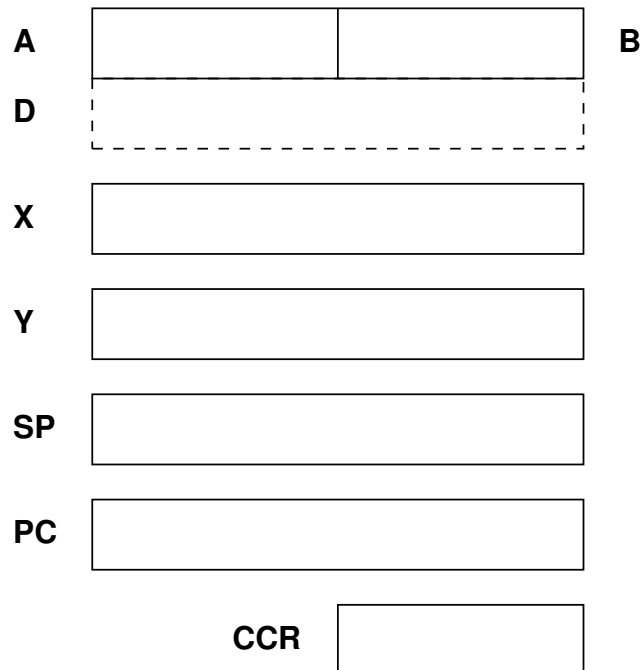
An example of some code which uses the stack

Stack Pointer:

Initialize ONCE before first use   (LDS #STACK)

Points to last used storage location

Decreases when you put something on stack

Increases when you take something off stack

0x1FF5	
0x1FF6	
0x1FF7	
0x1FF8	
0x1FF9	
0x1FFA	
0x1FFB	
0x1FFC	
0x1FFD	
0x1FFE	
0x1FFF	
0x2000	
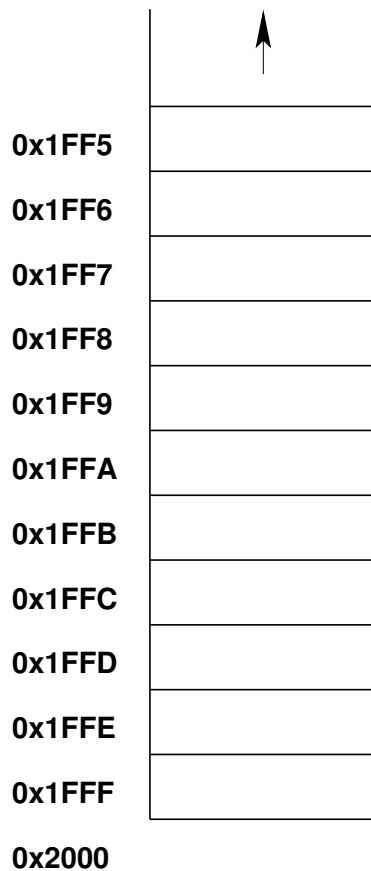
STACK: equ $2000

CODE: org $2000

```
        lds     #STACK
        ldaa    #$2e
        ldx     #$1254
        psha
        pshx
        clra
        ldx     #$ffff

        CODE THAT USES A & X

        pulx
        pula
```

A [   ]

X [   ]

SP [   ]

PSHA

Push A onto Stack

PSHA

Operation

$(SP) - \$0001 \Rightarrow SP$

$(A) \Rightarrow M_{SP}$

Decrements SP by one and loads the value in A into the address to which SP points.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PSHA	INH	36	Os

Subroutines

- A subroutine is a section of code which performs a specific task, usually a task which needs to be executed by different parts of a program.

- Example:

  - Math functions, such as *square root*

- Because a subroutine can be called from different places in a program, you cannot get out of a subroutine with an instruction such as

        bra   label

  because you would need to jump to different places depending upon which section of code called the subroutine.

- When you want to call the subroutine your code has to save the address where the subroutine should return to. It does this by saving the *return address* on the stack.

  - This is done automatically for you when you get to the subroutine by using the BSR (Branch to Subroutine) or JSR (Jump to Subroutine) instruction. This instruction pushes the address of the instruction following the JSR (BSR) instruction on the stack.

- After the subroutine is done executing its code it needs to return to the address saved on the stack.

  - This is done automatically for you when you return from the subroutine by using the RTS (Return from Subroutine) instruction. This instruction pulls the return address off of the stack and loads it into the program counter, so the program resumes execution of the program with the instruction following that which called the subroutine.

  The subroutine will probably need to use some MC9S12 registers to do its work. However, the calling code may be using its registers for some reason — the calling code may not work correctly if the subroutine changes the values of the MC9S12 registers.

  - To avoid this problem, the subroutine should save the MC9S12 registers before it uses them, and restore the MC9S12 registers after it is done with them.

BSR

Branch to Subroutine

BSR

Operation

$(SP) - \$0002 \Rightarrow SP$
$RTN_H:RTN_L \Rightarrow M_{SP}:M_{SP+1}$
$(PC) + \$0002 + rel \Rightarrow PC$

Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction after the BSR as a return address.

Decrements the SP by two, to allow the two bytes of the return address to be stacked.

Stacks the return address (the SP points to the high byte of the return address).

Branches to a location determined by the branch offset.

Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BSR *rel8*	REL	07 rr	SPPP

RTS **Return from Subroutine** RTS

Operation $(M_{SP}):(M_{SP + 1}) \Rightarrow PC_H:PC_L$
$(SP) + \$0002 \Rightarrow SP$

Restores the value of PC from the stack and increments SP by two. Program execution continues at the address restored from the stack.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
RTS	INH	3D	UfPPP

Example of a subroutine to delay for a certain amount of time

```
; Subroutine to wait for 100 ms

delay:  ldaa    #100    ; Execute outer loop 100 times
loop2:  ldx     #8000   ; Want inner loop to last 1 ms
loop1:  dbne    x,loop1 ; Inner loop  -- 3 cycles x 8000 times
        dbne    a,loop2
        rts
```

- Want inner loop to last for 1 ms. MC9S12 runs at 24,000,000 cycles/second, so 1 ms is 24,000 cycles.

- Inner loop should be 24,000 cycles/ (3 cycles/loop) = 8,000 times

- Problem: The subroutine changes the values of registers A and X

- To solve, save the values of A and X on the stack before using them, and restore them before returning.

```
; Subroutine to wait for 100 ms

delay:  psha            ; Save regs used by sub on stack
        pshx
        ldaa    #100    ; Execute outer loop 100 times
loop2:  ldx     #8000   ; Want inner loop to last 1 ms
loop1:  dbne    x,loop1 ; Inner loop  -- 3 cycles x 8000 times
        dbne    a,loop2
        pulx            ; Restore regs in opposite
        pula            ; order
        rts
```

```
; Program to make a binary counter on LEDs
;
; The program uses a subroutine to insert a delay
; between counts
;
; Does not work on Dragon12-Plus.  Need to write to PTJ
; to enable LEDs

prog:   equ     $2000
data:   equ     $1000
STACK:  equ     $2000
PORTB:  equ     $0001
DDRB:   equ     $0003


        org     prog

        lds     #STACK      ; initialize stack pointer
        ldaa    #$ff        ; put all ones into DDRA
        staa    DDRB        ; to make PORTB output
        clr     PORTB       ; put $00 into PORTB
loop:   jsr     delay       ; wait a bit
        inc     PORTB       ; add one to PORTB
        bra     loop        ; repeat forever

; Subroutine to wait for 100 ms

delay:  psha                ; Save regs used by sub on stack
        pshx
        ldaa    #100     ; Execute outer loop 100 times
loop2:  ldx     #8000    ; Want inner loop to last 1 ms
loop1:  dbne    x,loop1  ; Inner loop  -- 3 cycles x 8000 times
        dbne    a,loop2
        pulx                ; Restore regs in opposite
        pula                ; order
        rts
```

JSR and BSR place return address on stack

RTS returns to instruction after JSR or BSR

```
                            STACK:   EQU    $2000
                                     ORG    $2000
              2000 CF 20 00          LDS    #STACK
              2003 16 10 07          JSR    MY_SUB
              2006 7F               SWI
              2007 CE 12 34  MY_SUB: LDX    #$1234
              200A 3D               RTS
```

Address	
0x1EF5	
0x01E6	
0x1EF7	
0x1EF8	
0x1EF9	
0x1EFA	
0x1EFB	
0x1EFC	
0x1EFD	
0x1EFE	
0x1EFF	
0x1F00	
0x1F01	
0x1F02	
0x1F03	

A [|] B

D [   ]

X [   ]

Y [   ]

SP [   ]

PC [   ]

CCR [   ]

Another example of using a subroutine

Using a subroutine to wait for an event to occur, then take an action.

- Wait until bit 7 of address `$00C4` is set.

- Write the value in ACCA to address `$00C7`.

```
; This routine waits until the MC9S12 serial
; port is ready, then sends a byte of data
; to the MC9S12 serial port

putchar:       brclr     $00CC,#$80,putchar
               staa      $00CF
               rts
```

- Program to send the word `hello, world!` to the MC9S12 serial port

```
; Program fragment to write the word "hello, world!" to the
; MC9S12 serial port

               ldx       $str
loop:          ldaa      1,x+    ; get next char
               beq       done    ; char == 0 => no more
               jsr       putchar
               bra       loop
               swi

str:           dc.b      "hello, world!"
               fc.b      $0A,$0D,0    ; CR LF
```

Here is the complete program to write a line to the screen:

```
Freescale HC12-Assembler
(c) Copyright Freescale 1987-2009

 Abs. Rel.    Loc    Obj. code    Source line
 ---- ----   ------ ---------    -----------
    1    1           0000 2000    prog:    equ      $2000
    2    2           0000 1000    data:    equ      $1000
    3    3           0000 2000    stack:   equ      $2000
    4    4           0000 00CC    SCI0SR1: equ      $00CC  ; SCI0 status reg 1
    5    5           0000 00CF    SCI0DRL: equ      $00CF  ; SCI0 data reg low
    6    6
    7    7                                 org      prog
    8    8   a002000 CF20 00               lds      #stack
    9    9   a002003 CE10 00               ldx      #str
   10   10   a002006 A630        loop:     ldaa     1,x+ ; get next char
   11   11   a002008 2705                  beq      done ; char == 0 => no more
   12   12   a00200A 1620 10               jsr      putchar
   13   13   a00200D 20F7                  bra      loop
   14   14   a00200F 3F          done:     swi
   15   15
   16   16   a002010 4FCC 80FC   putchar: brclr     SCI0SR1,$80,putchar
   17   17   a002014 5ACF                  staa     SCI0DRL
   18   18   a002016 3D                    rts
   19   19
   20   20                                 org      data
   21   21   a001000 6865 6C6C   str:      fcc      "hello, world!"
              001004 6F2C 2077
              001008 6F72 6C64
              00100C 21
   22   22   a00100D 0A0D 00               dc.b     $0a,$0d,0 ; CR LF terminating zero
   23   23
```

Using DIP switches to get data into the MC9S12

• DIP switches make or break a connection (usually to ground)

DIP Switches on Breadboard
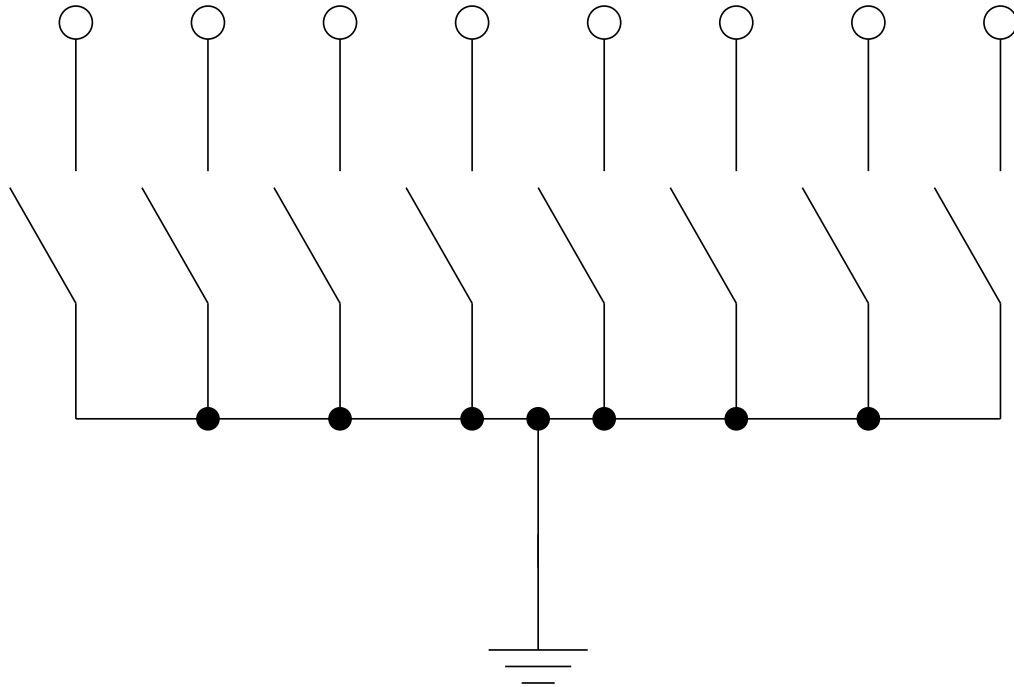
- To use DIP switches, connect one end of each switch to a resistor

- Connect the other end of the resistor to +5 V

- Connect the junction of the DIP switch and the resistor to an input port on the MC9S12

- The Dragon12-Plus has eight dip switches connected to Port H (PTH).

Using DIP Switches
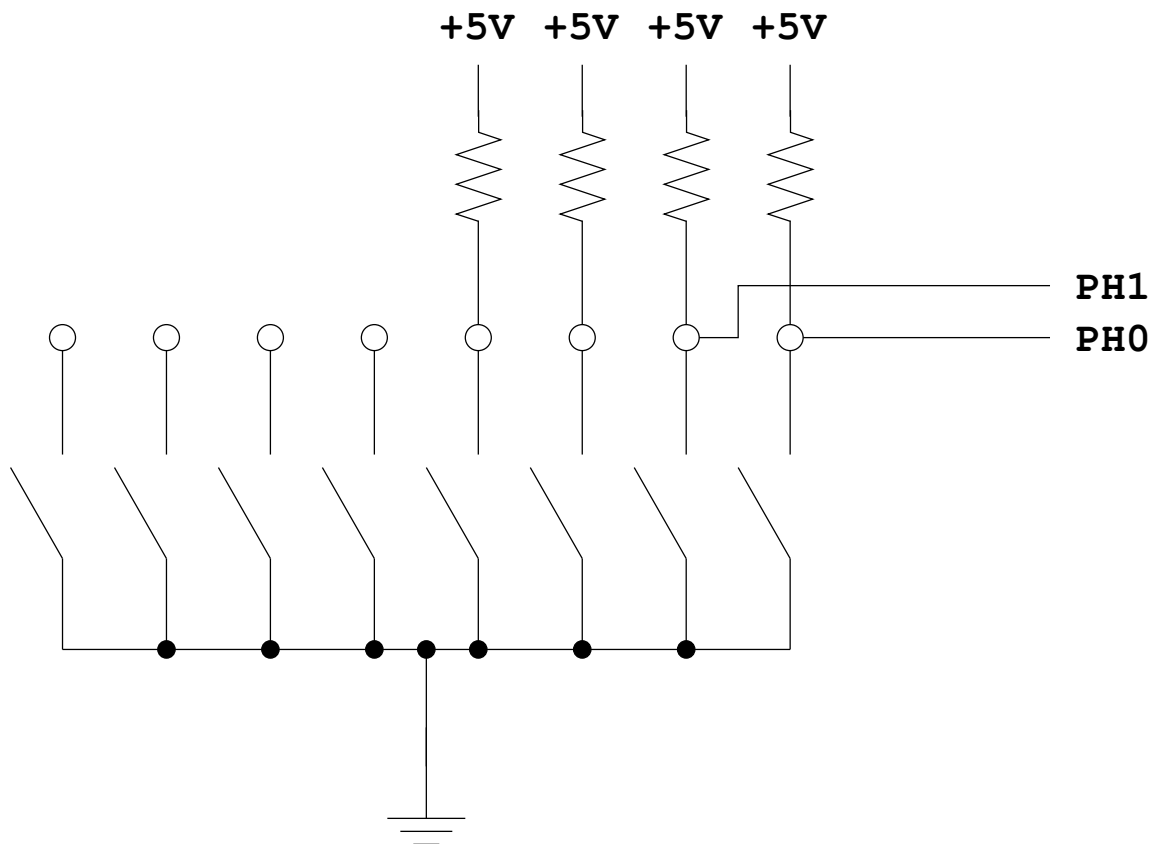
- When the switch is open, the input port sees a logic 1 (+5 V)

- When the switch is closed, the input sees a logic 0 (0 V)

Looking at the state of a few input pins

- Want to look for a particular pattern on 4 input pins

    - For example want to do something if pattern on PB3-PB0 is 0110

- Don't know or care what are on the other 4 pins (PB7-PB4)

- Here is the wrong way to do it:

```
        ldaa    PTH
        cmpa    #$06
        beq     task
```

- If PH7-PH4 are anything other than 0000, you will not execute the task.

- You need to mask out the Don't Care bits **before** checking for the pattern on the bits you are interested in
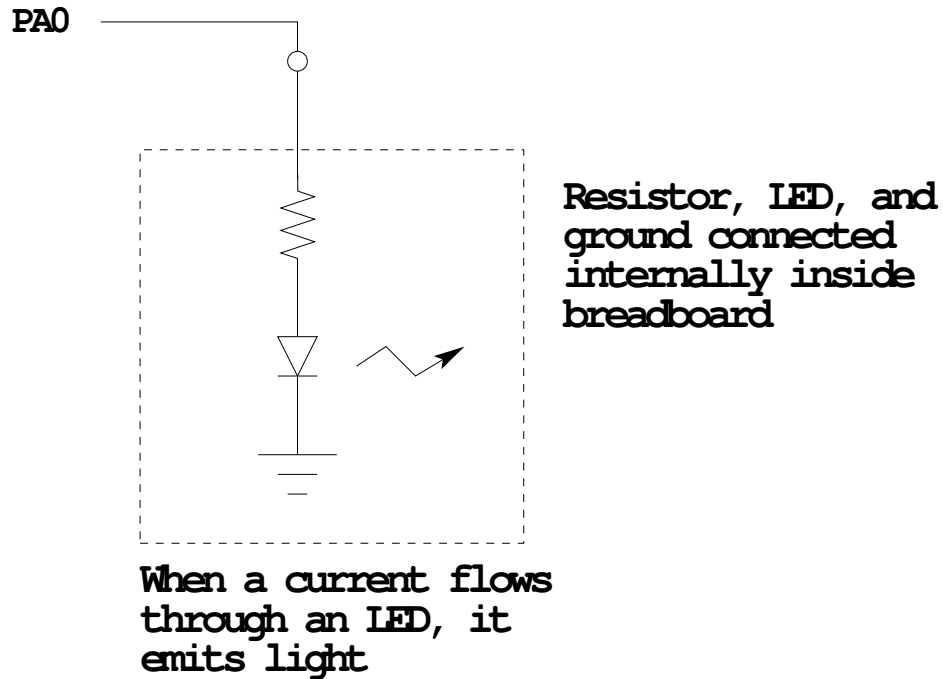
```
        ldaa    PTH
        anda    #$0F
        cmpa    #$06
        beq     task
```

- Now, whatever pattern appears on PB7-4 is ignored
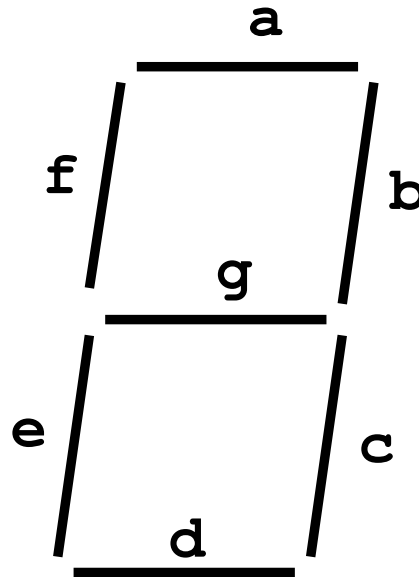
Using an MC9S12 output port to control an LED

- Connect an output port from the MC9S12 to an LED.

Using an output port to control an LED

PA0

Resistor, LED, and ground connected internally inside breadboard

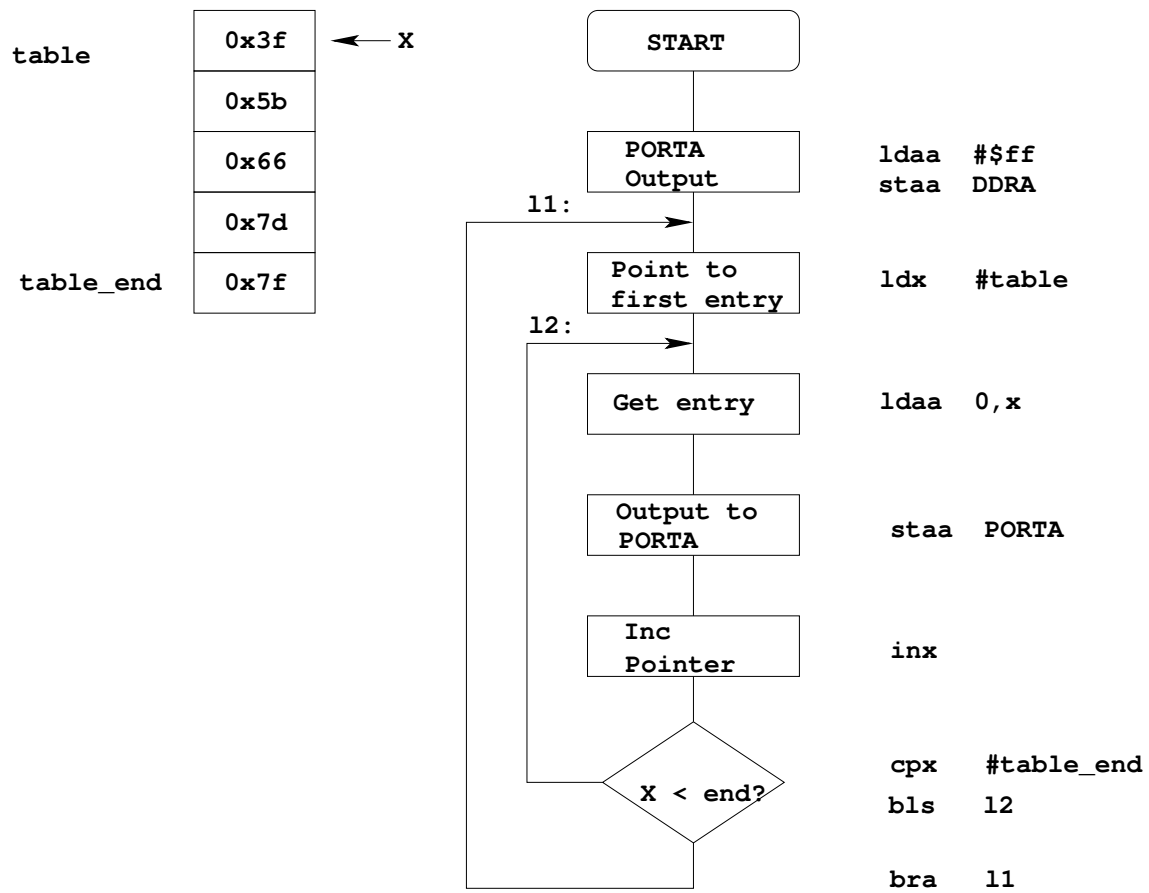**When a current flows through an LED, it emits light**

Making a pattern on a seven-segment LED

- Want to generate a particular pattern on a seven-segment LED:

- Determine a number (hex or binary) which will generate each element of the pattern

    - For example, to display a `0`, turn on segments `a, b, c, d, e and f`, or bits `0, 1, 2, 3, 4` and `5` of `PTB`. The binary pattern is `00111111`, or `$3f`.

    - To display `0 2 4 6 8`, the hex numbers are `$3f, $5b, $66, $7d, $7f`.

- Put the numbers in a table

- Go through the table one by one to display the pattern

- When you get to the last element, repeat the loop

Flowchart to display a pattern of lights on a set of LEDs

table	0x3f	← X
	0x5b	
	0x66	
	0x7d	
table_end	0x7f	

```
                START

              PORTA              ldaa   #$ff
              Output            staa   DDRA
l1:

              Point to          ldx    #table
              first entry
l2:

              Get entry         ldaa   0,x

              Output to          staa   PORTA
              PORTA

              Inc               inx
              Pointer

                                cpx    #table_end
              X < end?          bls    l2

                                bra    l1
```

```
; Program using subroutine to make a time delay

prog:        equ     $2000
data:        equ     $1000
stack:       equ     $2000
PORTB:       equ     $0001
DDRB:        equ     $0003

             org     prog

             lds     #stack      ; initialize stack pointer
             ldaa    #$ff        ; Make PORTB output
             staa    DDRB        ;   0xFF -> DDRB
l1:          ldx     #table      ; Start pointer at table
l2:          ldaa    1,x+        ; Get value; point to next
             staa    PORTB       ; Update LEDs
             jsr     delay       ; Wait a bit
             cpx     #table_end  ; More to do?
             bls     l2          ; Yes, keep going through table
             bra     l1          ; At end; reset pointer

delay:       psha
             pshx
             ldaa    #100
loop2:       ldx     #8000
loop1:       dbne    x,loop1
             dbne    a,loop2
             pulx
             pula
             rts

             org     data
table:       dc.b    $3f
             dc.b    $5b
             dc.b    $66
             dc.b    $7d
table_end:   dc.b    $7F
```