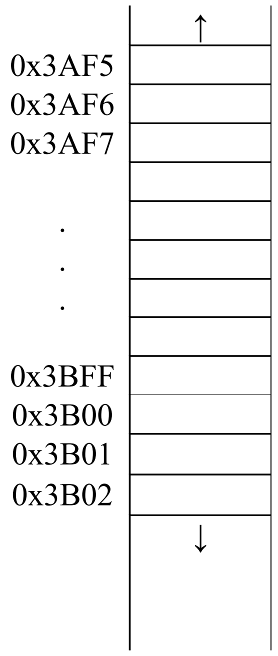
- **More on programming in assembly language**
- **Introduction to Ports on the HC12**
- **Huang Sections 7.1 through 7.5**
    - Good programming style
    - Tips for writing programs
    - Input and Output Ports
        - Simplified Input Port
        - Simplified Output Port
    - Ports on the HC12
        - PORTA, PORTB, DDRA, DDRB
        - A simple program to use PORTA and PORTB
    - Subroutines and the Stack
    - An example of a simple subroutine
    - Using a subroutine with PORTA to make a binary counter on LEDs

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 HC12 use a block of memory from about **$3B00** to **$3BFF** for the stack.
• For this region of memory, initialize the stack pointer to **$3C00**.
• 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 HC12, 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.

<u>If not enough space is allocated the stack can overwrite data and/or code, which will cause the program to malfunction or crash.</u>

The stack is an array of memory dedicated to temporary storage



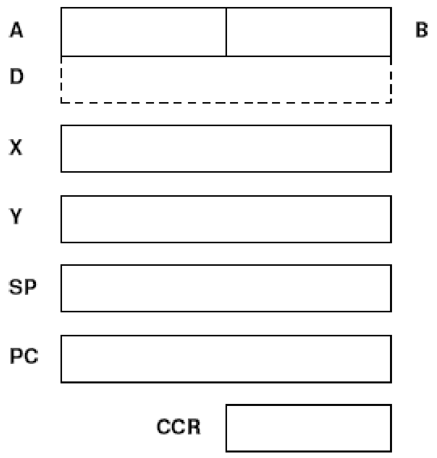
SP points to the location last item placed in block

SP **decreases** when you put an item on stack

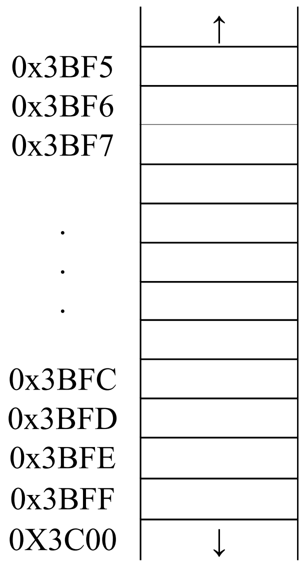
SP **increases** when you pull item from stack

For HC12 EVBU, use **0x3C00** as initial SP:

```
STACK:  EQU   $3C00
        LDS   #STACK
```



An example of some code which used the stack



Stack Pointer

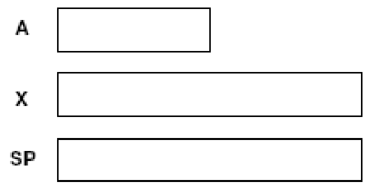
Initialize ONCE before first use (LDS #STACK)

Points to last used storage location
Decreases when you put something on stack
Increases when you take something off stack

```
STACK:  EQU   $3C00
        org    0x1000
        lds    #STACK
        ldaa    #$2e
        ldx    #$1254
        psha
        pshx
        clra
        ldx     #$ffff
```

CODE THAT USES A & X

```
        pulx
        pula
```



PSHA Push A onto Stack PSHA

Operation: $(SP) - \$0001 \Rightarrow SP$
$(A) \Rightarrow M_{(SP)}$

Description: Stacks the content of accumulator A. The stack pointer is decremented by one. The content of A is then stored at the address the SP points to.

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 Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHA	INH	36	Os	Os

PSHX Push Index Register X onto Stack PSHX

Operation: $(SP) - \$0002 \Rightarrow SP$
$(X_H : X_L) \Rightarrow M_{(SP)} : M_{(SP+1)}$

Description: Stacks the content of index register X. The stack pointer is decremented by two. The content of X is then stored at the address to which the SP points. After PSHX executes, the SP points to the stacked value of the high-order half of X.

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 Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHX	INH	34	OS	OS

PULA — Pull A from Stack — PULA

Operation: $(M_{(SP)}) \Rightarrow A$
$(SP) + \$0001 \Rightarrow SP$

Description: Accumulator A is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine, to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PULA	INH	32	ufO	ufO

PULX — Pull Index Register X from Stack — PULX

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L$
$(SP) + \$0002 \Rightarrow SP$

Description: Index register X is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PULX	INH	30	UfO	UfO

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 **jmp 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 **JSR** (Jump to Subroutine) or **BSR** (Branch 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 HC12 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 HC12 registers.

– To avoid this problem, the subroutine should save the HC12 registers before it uses them, and restore the HC12 registers after it is done with them.

JSR — Jump to Subroutine — JSR

Operation: (SP) – \$0002 ⇒ SP
$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$
Subroutine Address ⇒ PC

Description: Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction following the JSR 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 order byte of the return address.

Calculates an effective address according to the rules for extended, direct, or indexed addressing.

Jumps to the location determined by the effective address.

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

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
JSR *opr8a*	DIR	`17 dd`	`SPPP`	`PPPS`
JSR *opr16a*	EXT	`16 hh ll`	`SPPP`	`PPPS`
JSR *oprx0_xysp*	IDX	`15 xb`	`PPPS`	`PPPS`
JSR *oprx9,xysp*	IDX1	`15 xb ff`	`PPPS`	`PPPS`
JSR *oprx16,xysp*	IDX2	`15 xb ee ff`	`fPPPS`	`fPPPS`
JSR [D,*xysp*]	[D,IDX]	`15 xb`	`fIfPPPS`	`fIfPPPS`
JSR [*oprx16,xysp*]	[IDX2]	`15 xb ee ff`	`fIfPPPS`	`fIfPPPS`

RTS — Return from Subroutine — RTS

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

Description: Restores context at the end of a subroutine. Loads the program counter with a 16-bit value pulled from the stack and increments the stack pointer by two. Program execution continues at the address restored from the stack.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
RTS	INH	`3D`	`UfPPP`	`UfPPP`

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

```
delay:  ldaa   #250
loop2:  ldx    #800
loop1:  dex
        bne    loop1
        deca
        bne    loop2
        rts
```

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

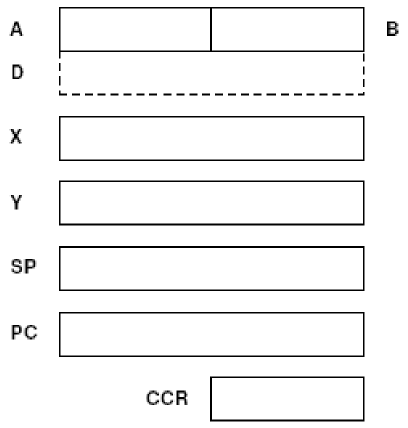
• To solve, save the values of A and X on the stack before using them, and restore them before returning.

```
delay:  psha                    ; Save registers used by sub on stack
        pshx
        ldaa   #250
loop2:  ldx    #800
loop1: dex
       bne     loop1
       deca
       bne     loop2
       pulx                     ; Restore registers in opposite order
       pula
       rts
```

```
;
; The program uses a subroutine to insert a delay
; between counts

prog:          equ     $1000
STACK:         equ     $3C00          ; Stack ends of $3BFF
PORTA:         equ     $0000
PORTB:         equ     $0001
DDRA:          equ     $0002
DDRB:          equ     $0003

               org prog

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

; Subroutine to wait for a few milliseconds

delay:         psha
               pshx
               ldaa    #250
loop2:         ldx     #800
loop1:         dex
               bne     loop1
               deca
               bne     loop2
               pulx
               pula
               rts
```

JSR and **BSR** place return address on stack
RTS returns to instruction after JSR or BSR

```
3c00                    STACK:  EQU     $3C00
1000                            ORG     $1000

1000 cf 3c 00                   LDS     #STACK
1003 16 10 07                   JSR     MY_SUB
1006 3f                         SWI
1007 ce 12 34           MY_SUB: LDX     #$1234
100a 3d                         RTS
```



Another example of using a subroutine

```
; Program fragment to write the word "hello" to the
; HC12 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"
            fc.b     $0A,$0D,0    ; CR LF
            .
            .
            .
putchar:    ...                   ; put character into the serial port
```

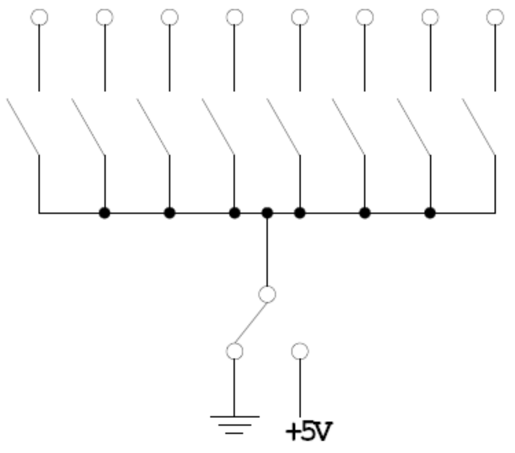
Here is the complete program to write a line to the screen:

```
prog:       equ     $1000
data:       equ     $2000
stack:      equ     $3c00

            org prog
            lds     #stack
            ldx     #str
loop:       ldaa    1,x+                ; get next char
            beq     done                ; char == 0 ⇒ no more
            jsr     putchar
            bra     loop
done:       swi
putchar:    brclr   $00CC,$80,putchar   ; check for SCI port ready
            staa    $00CF               ; put character onto SCI port
            rts

            org     data
str:        fcc     "hello"
            dc.b    $0a,$0d,0           ; CR LF
```

Using DIP switches to get data into the HC12

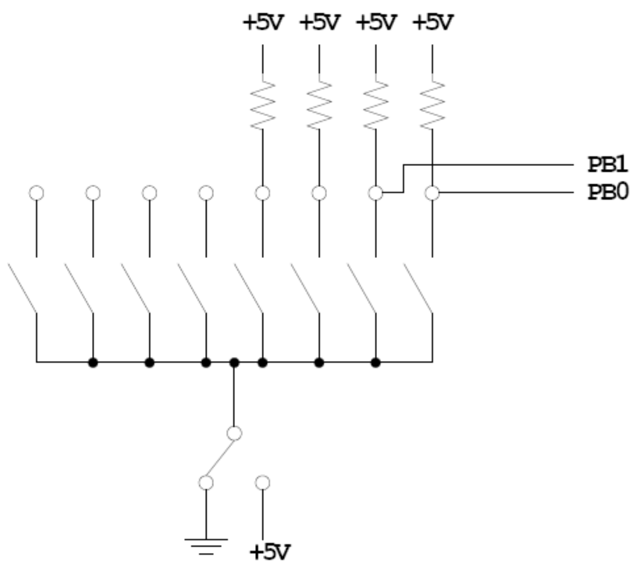
- DIP switches make or break a connection (usually to ground)

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 HC12



• 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 doing it:

```
ldaa    PORTB
cmpa    #%0110
beq     task
```

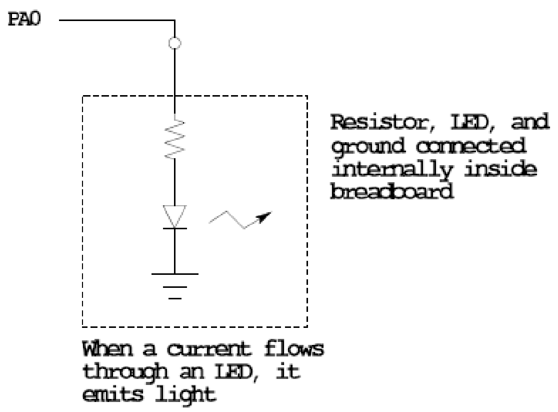
• If PB7-PB4 are anything other than 0000, you will not execute the task.
• You **need to mask out the Don't Care bits before checking** for the pattern on the bits you are interested in

```
ldaa    PORTB
anda    #%00001111
cmpa    #%00000110
beq     task
```

• Now, whatever pattern appears on PB7-4 is ignored

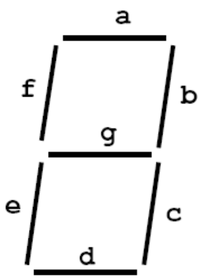
Using an HC12 output port to control an LED

• Connect an output port from the HC12 to an LED.


PA0
Resistor, LED, and ground connected internally inside breadboard
When a current flows through an LED, it emits light

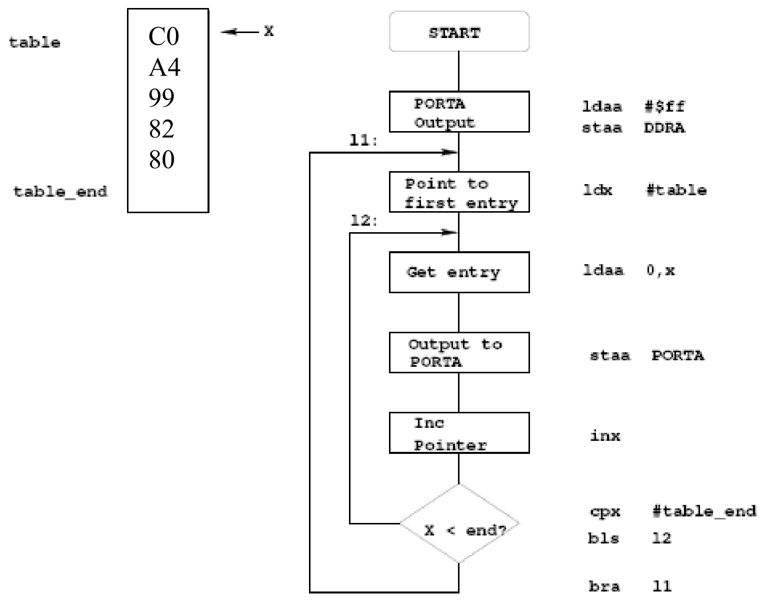
Making a pattern on a seven-segment LED

• 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 PTH. The binary pattern is 0011 1111, or $3f.
- The new board uses a common anode seven-segment LED. To display a 0, you need to use the binary pattern 1100 0000, or $C0.
– To display 0 2 4 6 8, the hex numbers are $C0, $A4, $99, $82, $80.
• 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



```
; Program using subroutine to make a time delay
prog:           equ     $1000
data:           equ     $2000
stack:          equ     $3C00
PTH:            equ     $0260
DDRH:           equ     $0262

                org prog
                lds     #stack          ; Initialize stack pointer
                ldaa    #$ff            ; Make PTH output
                staa    DDRH            ; 0xFF -> DDRH
l1:             ldx     #table          ; Start pointer at table
l2:             ldaa    1, x+           ; Get value; point to next
                staa    PTH             ; 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    #250
loop2:          ldx     #8000
```

```
loop1:      dex
            bne     loop1
            deca
            bne     loop2
            pulx
            pula
            rts

            org     data
table:      dc.b    $C0
            dc.b    $A4
            dc.b    $99
            dc.b    $82
table_end:  dc.b    $80
```