Microcomputers I – CE 320

Mohammad Ghamari, Ph.D.

Electrical and Computer Engineering

Kettering University

Announcement

Lecture 17: More on Subroutines

Today’s Topics

- Vectors, Matrices, Structures
- Return subroutine output using the stack
- Review the full structure of stack frame

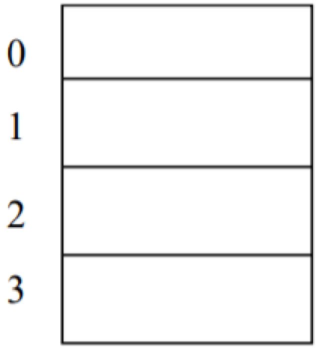
Vectors and Matrices

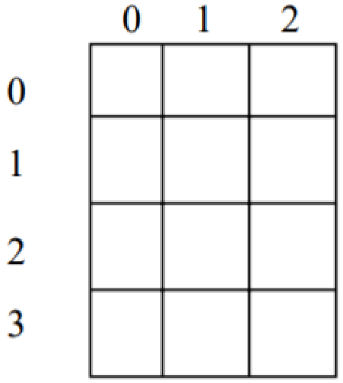
- To declare an array : Assemblers directives
  - db, dc.b, fcb for arrays of 8-bit elements
  - dw, dc.w, fdb for arrays of 16-bit elements

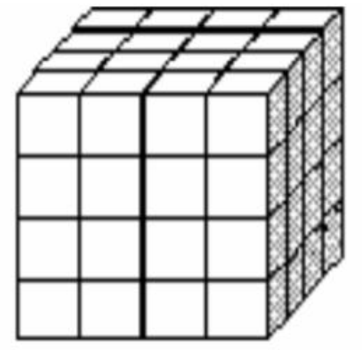
db (define byte)
dc.b (define constant byte)
fcb (form constant byte)

dw (define word)
dc.w (define constant word)
fdb (form double bytes)

- First element associated with index=0 to facilitate address calculation.


Vector/Array


2-D Matrix (4x3)


A 3-D Matrix (4x4x4)

Vectors and Matrices

Example 1

```
org $800

a1 db $11, $22, $33, $44

a2 dc.b $01

    dc.b $02

   dc.b $03

a3 rmb 2
```

C equivalent :
byte a1[4];   a1[0] = 17;
byte a2[3];   a2[1] = 2;
byte a3[2];

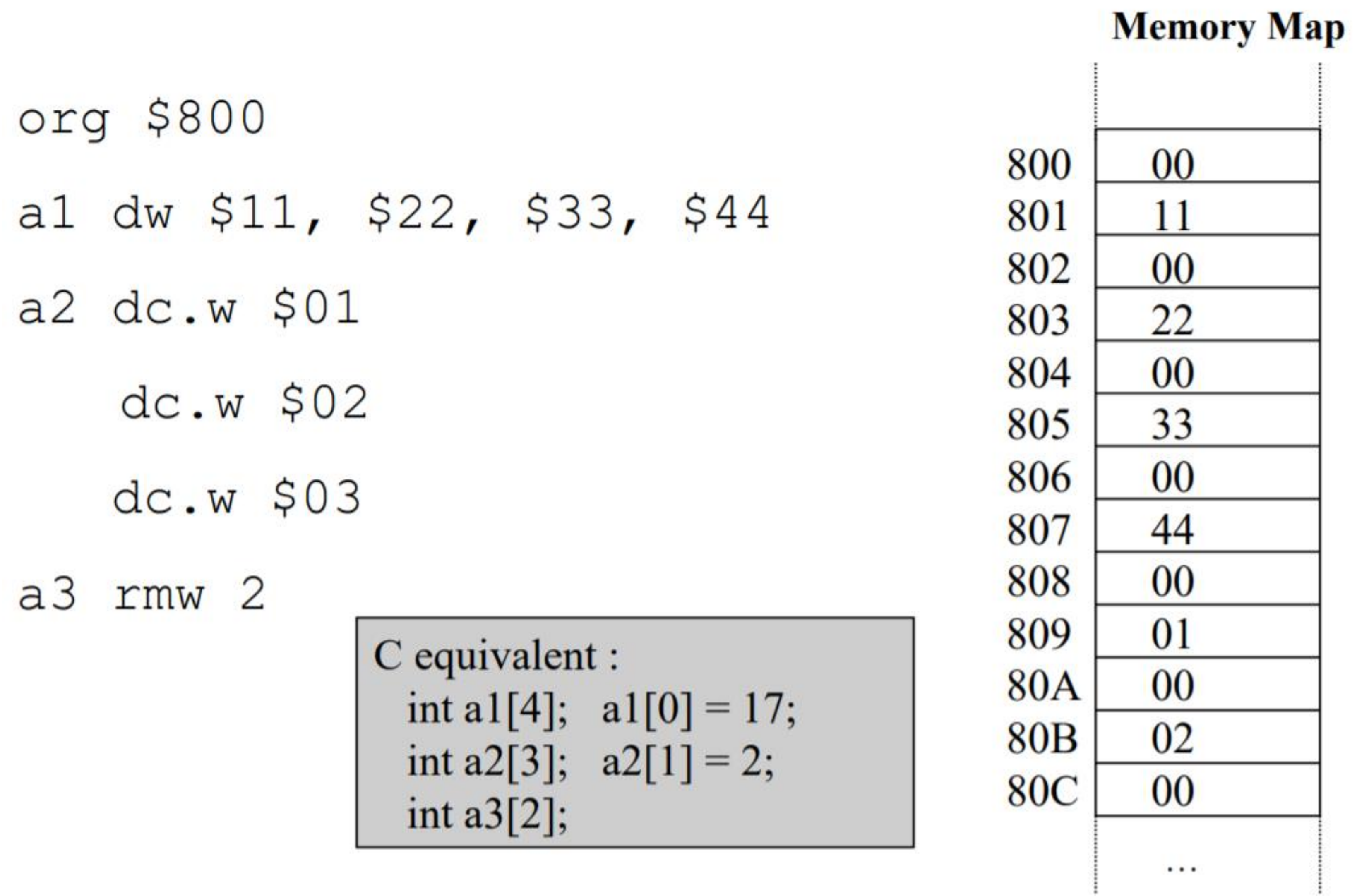
Memory Map

800	11
801	22
802	33
803	44
804	01
805	02
806	03
807	?
808	?
809	?
80A	?
80B	?

rmb (reserve memory byte)

Vectors and Matrices

Example 2



```
org $800

a1 dw $11, $22, $33, $44

a2 dc.w $01

    dc.w $02

   dc.w $03

a3 rmw 2
```

C equivalent :

```
int a1[4];   a1[0] = 17;
int a2[3];   a2[1] = 2;
int a3[2];
```

Memory Map

800	00
801	11
802	00
803	22
804	00
805	33
806	00
807	44
808	00
809	01
80A	00
80B	02
80C	00

...

Vectors and Matrices

Example 3

```
 org $800
a1  db $11, $22, $33, $44
a2  dc.b $01, $05
    dc.b $02, $06
    dc.b $03, $07
a3  rmw  2*2
```

C equivalent :

```
  byte a1[2][2];   a2[0][1] = 5;
  byte a2[3][2];   a2[1][1] = 6;
  word a3[2][2];
```

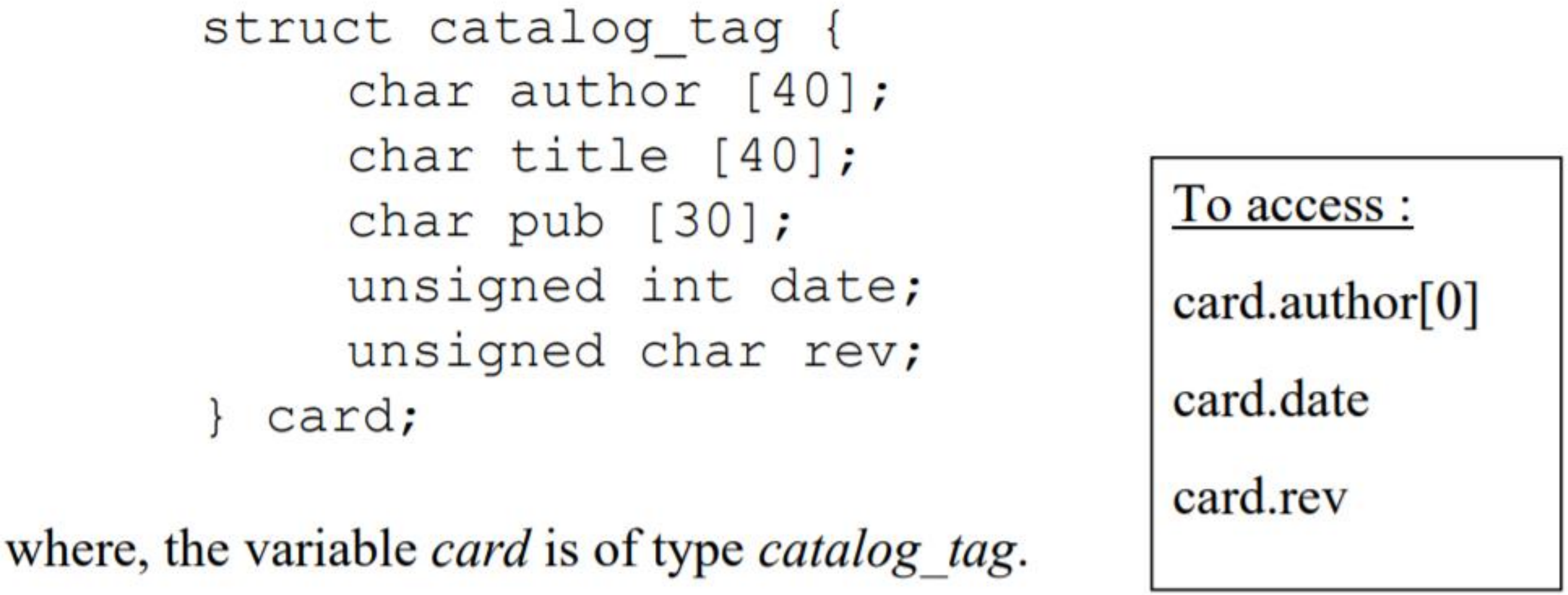
Memory Map

800	11
801	22
802	33
803	44
804	01
805	05
806	02
807	06
808	03
809	07
80A	??
80B	??
80C	??
	??

Structures

- Group of related variables that can be accessed through common name.

- Each item within structure has its own data type, which can be different.



```
struct catalog_tag {
    char author [40];
    char title [40];
    char pub [30];
    unsigned int date;
    unsigned char rev;
} card;
```

where, the variable *card* is of type *catalog_tag*.

To access :

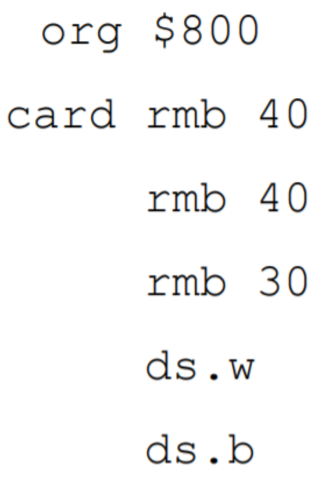
card.author[0]

card.date

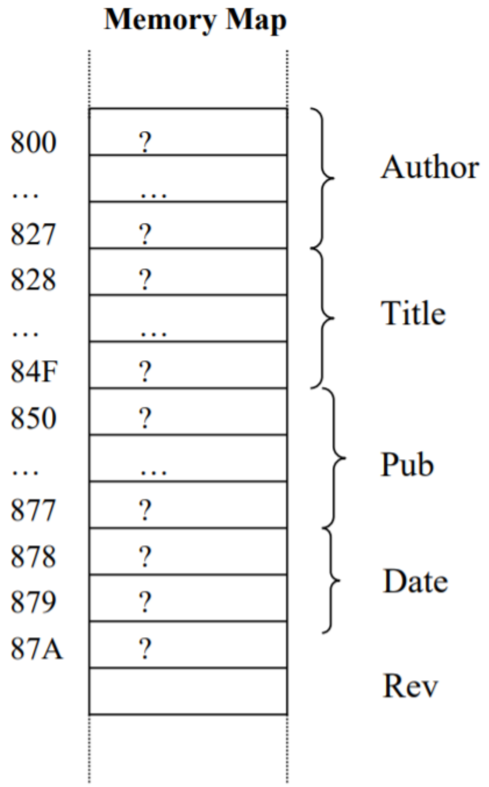
card.rev

Structures

Structures are typically implemented using **blocks of memory** where each field corresponds to a specific variable, and the structure as a whole can be accessed through a common name.

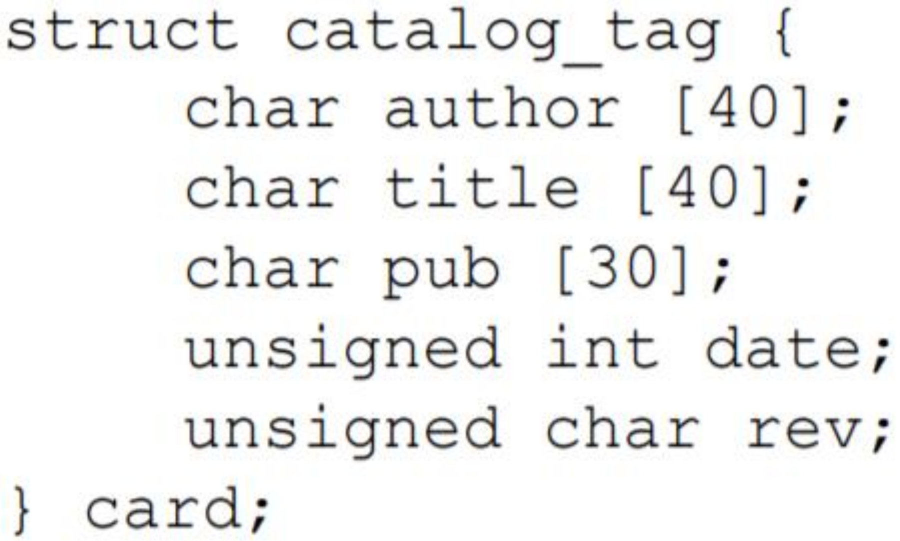


```
   org $800
card rmb 40
     rmb 40
     rmb 30
     ds.w
     ds.b
```



Memory Map

Address	Content	Field
800	?	Author
…	…	
827	?	
828	?	Title
…	…	
84F	?	
850	?	Pub
…	…	
877	?	
878	?	Date
879	?	
87A	?	Rev



```
struct catalog_tag {
    char author [40];
    char title [40];
    char pub [30];
    unsigned int date;
    unsigned char rev;
} card;
```

The Stack

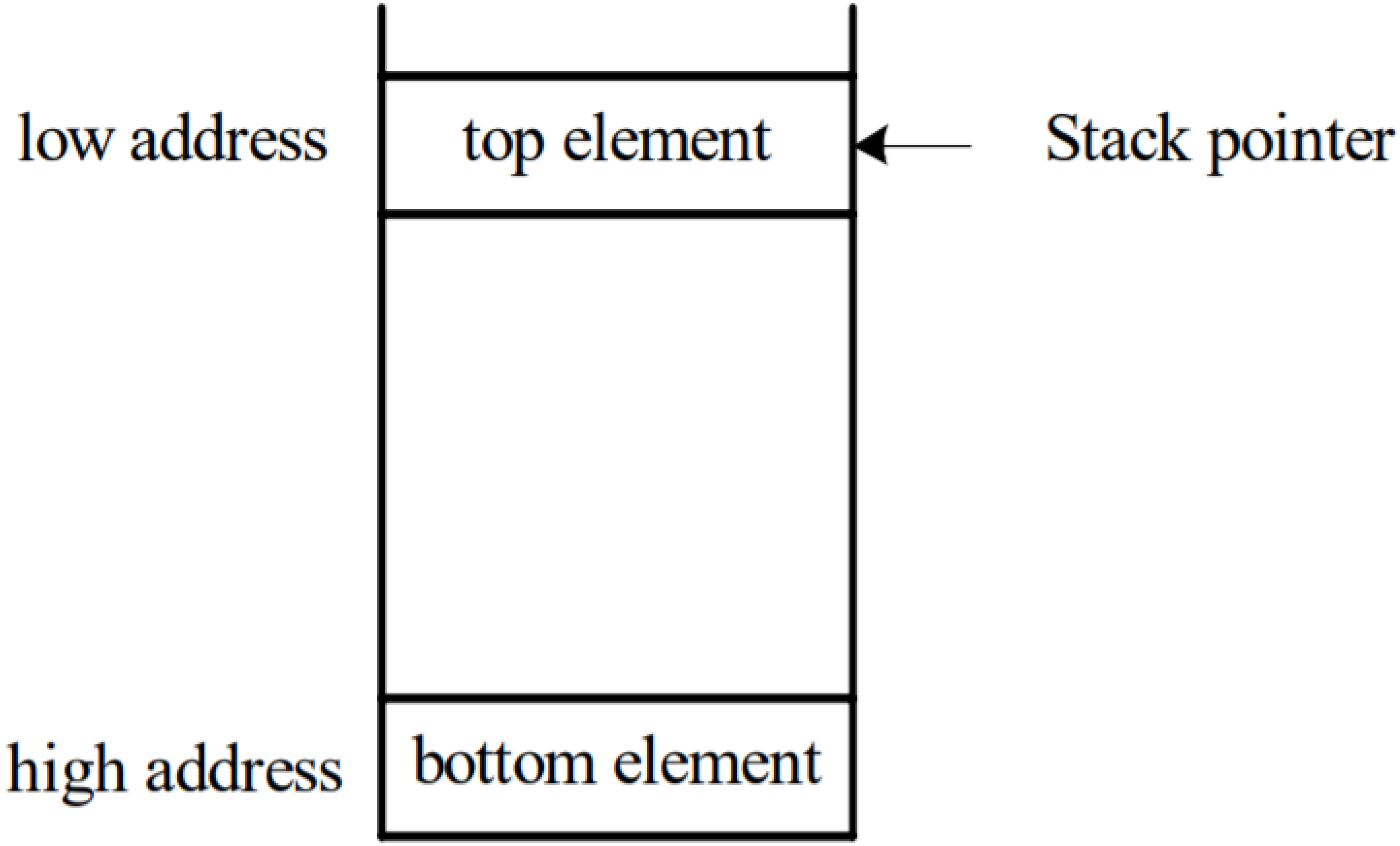


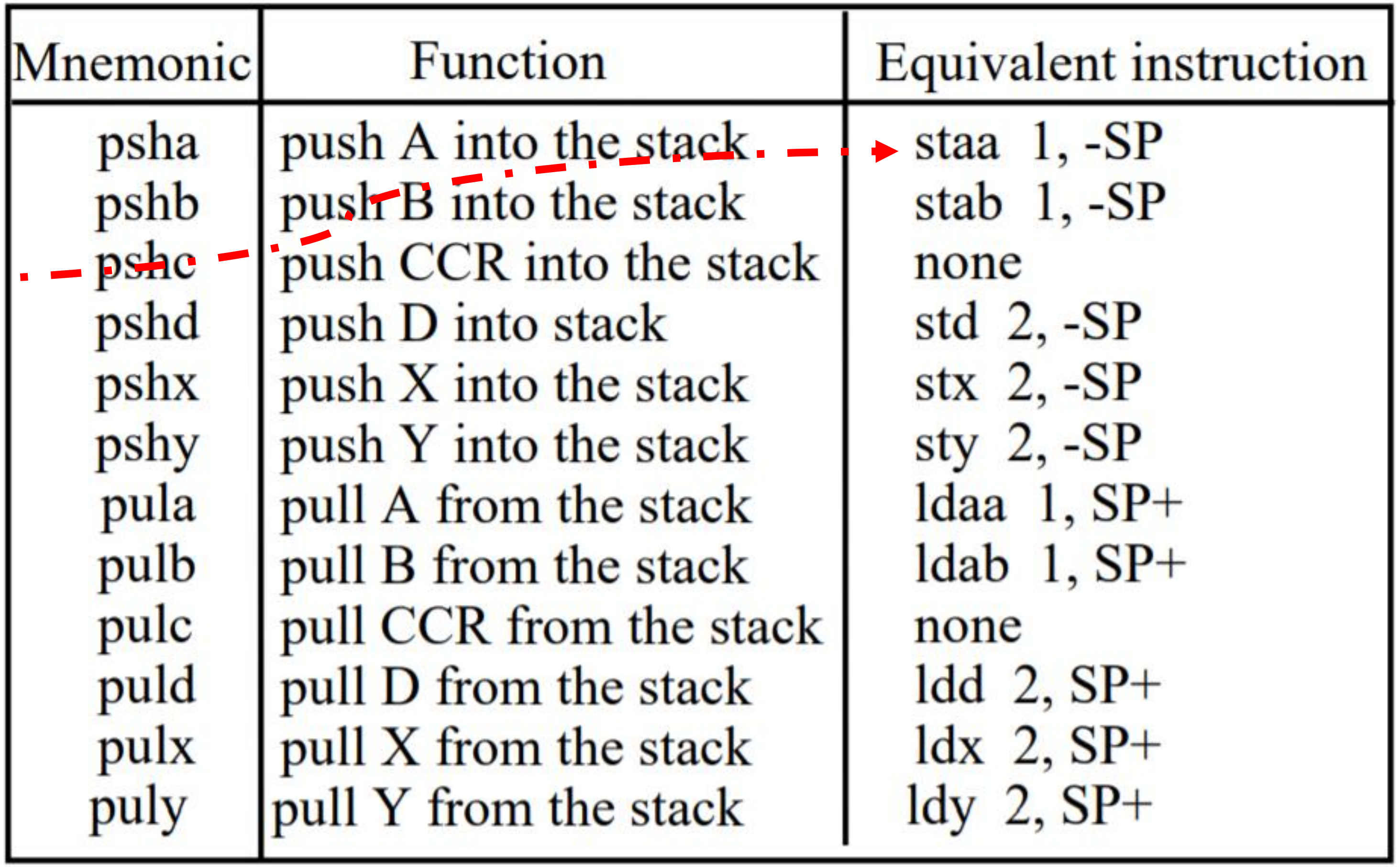
Diagram of the 68HC12 stack

HCS12 Support for the Stack Data Structure

- A 16-bit stack pointer (SP)

- **STAA 1:** Store the contents of accumulator A (A register) into the memory location with an address offset of 1.

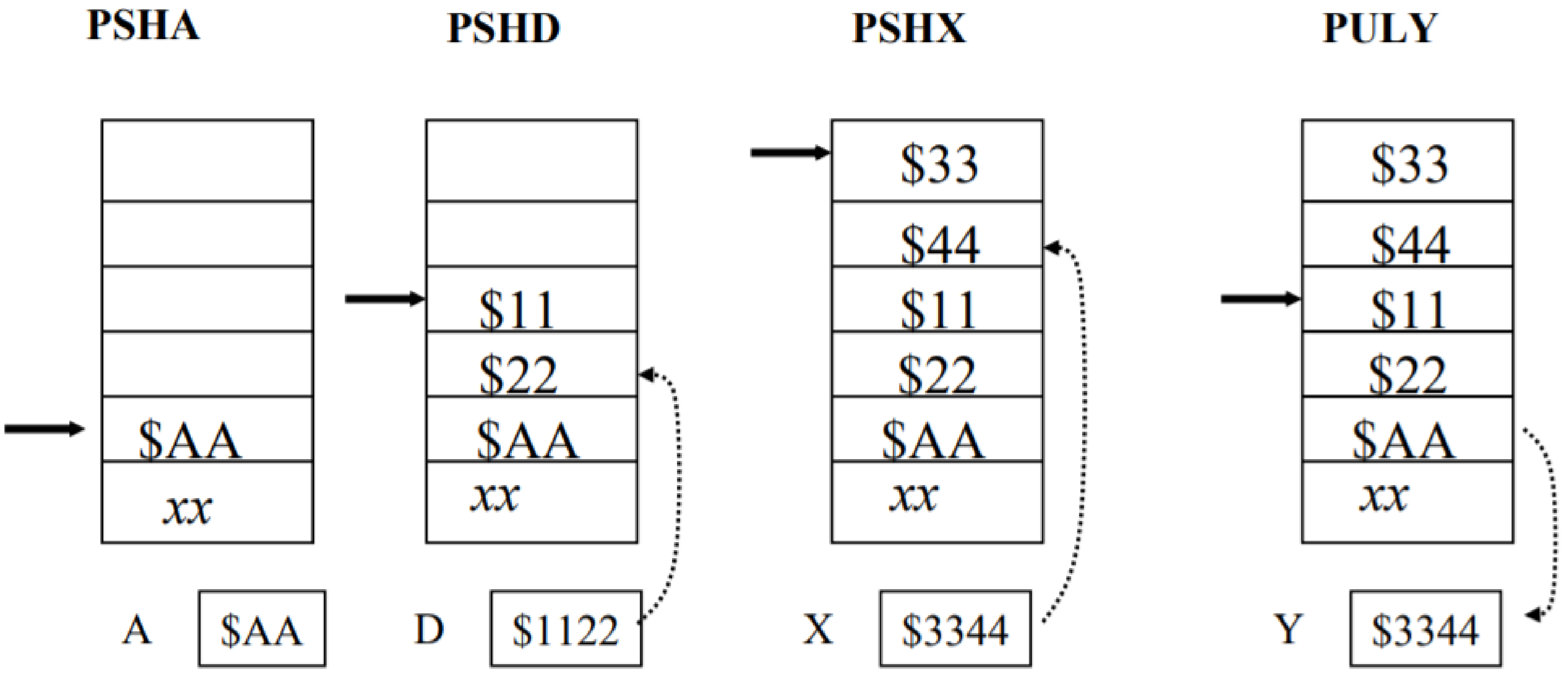
- **-SP:** Decrement the stack pointer after the store operation.



Mnemonic	Function	Equivalent instruction
psha	push A into the stack	staa 1, -SP
pshb	push B into the stack	stab 1, -SP
pshc	push CCR into the stack	none
pshd	push D into stack	std 2, -SP
pshx	push X into the stack	stx 2, -SP
pshy	push Y into the stack	sty 2, -SP
pula	pull A from the stack	ldaa 1, SP+
pulb	pull B from the stack	ldab 1, SP+
pulc	pull CCR from the stack	none
puld	pull D from the stack	ldd 2, SP+
pulx	pull X from the stack	ldx 2, SP+
puly	pull Y from the stack	ldy 2, SP+

The Stack

Example

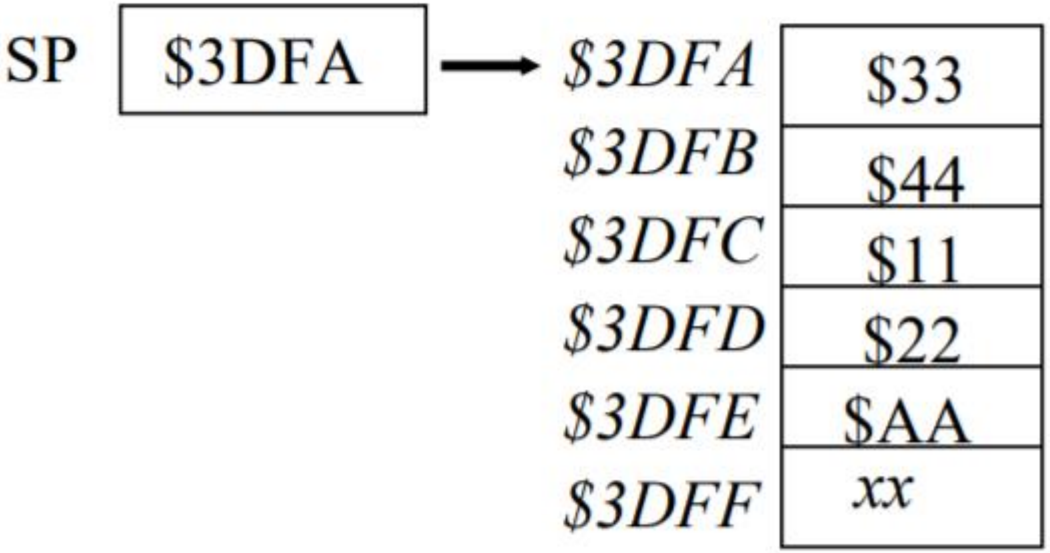


xx is Don’t care

Where is the Runtime Stack ?

- SP = address of the top element

- Before any PSH/PUL instruction, SP must be initialized.
    - LDS #$3DFF



SP $3DFA →

$3DFA	$33
$3DFB	$44
$3DFC	$11
$3DFD	$22
$3DFE	$AA
$3DFF	xx

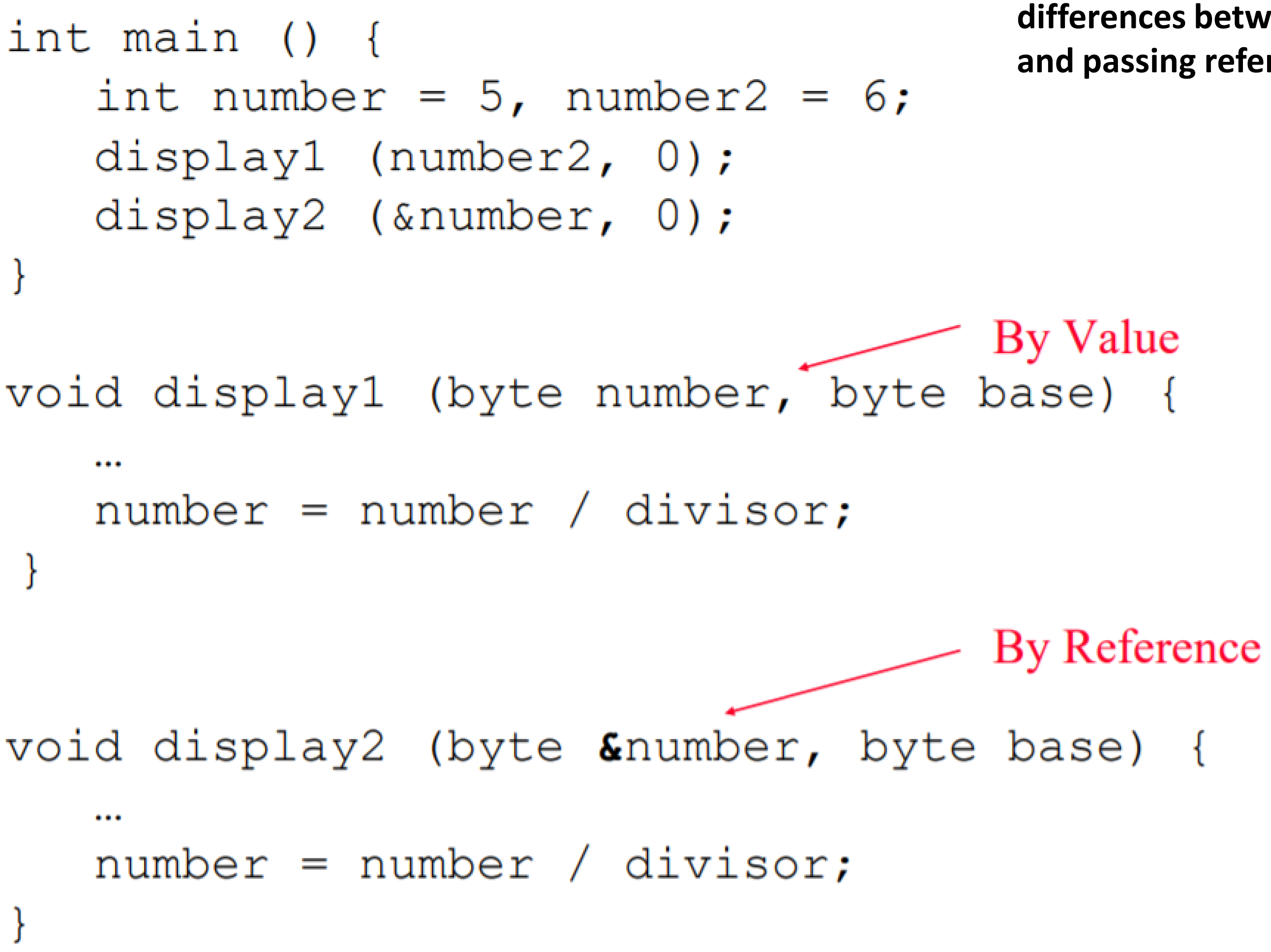
- Stack is any RAM area in main memory

- Who initializes ?
    - Simulator: Your program must use $3DFF
    - NoICE Debbugger: Auto-init's to $3DFF

Issues in Subroutine Calls: Value versus Reference

- **main** initializes two integer variables.

- Then calls two **display** functions passing **number2** and the address of **number** as arguments.



```
int main () {
   int number = 5, number2 = 6;
   display1 (number2, 0);
   display2 (&number, 0);
}

void display1 (byte number, byte base) {
   …
   number = number / divisor;
 }

void display2 (byte &number, byte base) {
   …
   number = number / divisor;
}
```

By Value

By Reference

Passing parameters to subroutines

- **This code demonstrates the differences between passing values and passing references.**

Subroutine Result Returning

- The result of a computation performed by the subroutine can be returned to the caller using **three methods**:
  - **Use registers:** This method is most convenient when there are only a few bytes to be returned to the caller.
  - **Use the stack:**
    - The caller creates a hole of a certain size in the stack before making the subroutine call.
    - The callee places the computation result in the hole before returning to the caller.
  - **Use global memory:** The callee simply places the value in the global memory and the caller will be able to access them.

Returning Data By Value Using the Stack

- The stack can be used to return an output value:

  1. The caller opens up room on the stack for the result to be returned by value (LEAS).

  2. The caller pushes any inputs that are passed on the stack.

  3. The callee stores the answer into the space created on the stack (using indexed addressing off of SP) and returns

  4. The caller removes the value with a PULL

Example

- Write a subroutine that meets the following requirements and a main program that calls it.
  1. The subroutine adds two 2-byte signed numbers.
  2. If the sum is less than -2000, the subroutine returns -2000.
  3. If the sum is greater than 3000, the subroutine returns 3000.
  4. The numbers to add and the result are all passed on the stack.

Steps:

1. The caller opens up room on the stack for the result to be returned by value (LEAS).

2. The caller pushes any inputs that are passed on the stack.

3. The callee stores the answer into the space created on the stack (using indexed addressing off of SP) and returns.

4. The caller removes the value with a PULL (Pops the result from the stack).

```
MIN        EQU        -2000
MAX        EQU        3000

           ORG        $C000
           LDS        #$3600
           LEAS       -2,SP
           LDD        #1600
           PSHD
           LDD        #700
           PSHD
           JSR        ADDRNG
           LEAS 4,SP
           PULD
           SWI
```

Pushes the first 2-byte signed number onto the stack.

Pushes the second 2-byte signed number onto the stack.

Adjusts the stack pointer to clean up the space used for the parameters and result

Load the first number from the stack

```
ADDRNG   LDD    2,SP
         ADDD   4,SP
         CPD    #MIN
         BGE    Skip
         LDD    #MIN
Skip     CPD    #MAX
         BLE    Skip2
         LDD    #MAX
Skip2    STD    6,SP
         RTS
```

Adds the second number (popped from the stack) to the first

Store the result on the stack

Returning Data By Reference

- As a standard rule, subroutines **only return one object** (if they return a value at all).

- While this may seem a little limiting, the one object returned may have multiple pieces.

- To do this, the result is passed by reference so that, technically, the subroutine still only returns one item… sort of.

- Usually, the **caller is responsible for creating space for the result**.
  - The caller then passes the address of the result as an input.
  - The subroutine changes values in the allocated space.
  - This means that although the subroutine has an effect, it doesn’t technically return a value.

Example

Write a subroutine that meets the following requirements, and a main program that calls it.

1. The subroutine finds the minimum and maximum values in an array of unsigned numbers.

2. The address of the array is the **first input** parameter passed on the stack.

3. The length of the array is a one-byte value passed as the **second input** parameter on the stack.

4. The subroutine **returns** a two-byte value by reference on the stack, where the first byte is the minimum value and the second byte is the maximum value.

5. The subroutine returns a minimum value higher than the maximum value if the length is zero.

- **Caller is responsible for creating space for the result.**
  - **The caller passes the address of the result as an input.**
- **The address of the array is the first input parameter passed on the stack.**
- **The length of the array is a one-byte value passed as the second input parameter on the stack.**
- **Then, the return address is pushed on the stack.**
- **The subroutine returns a two-byte value by reference on the stack, where the first byte is the minimum value and the second byte is the maximum value.**

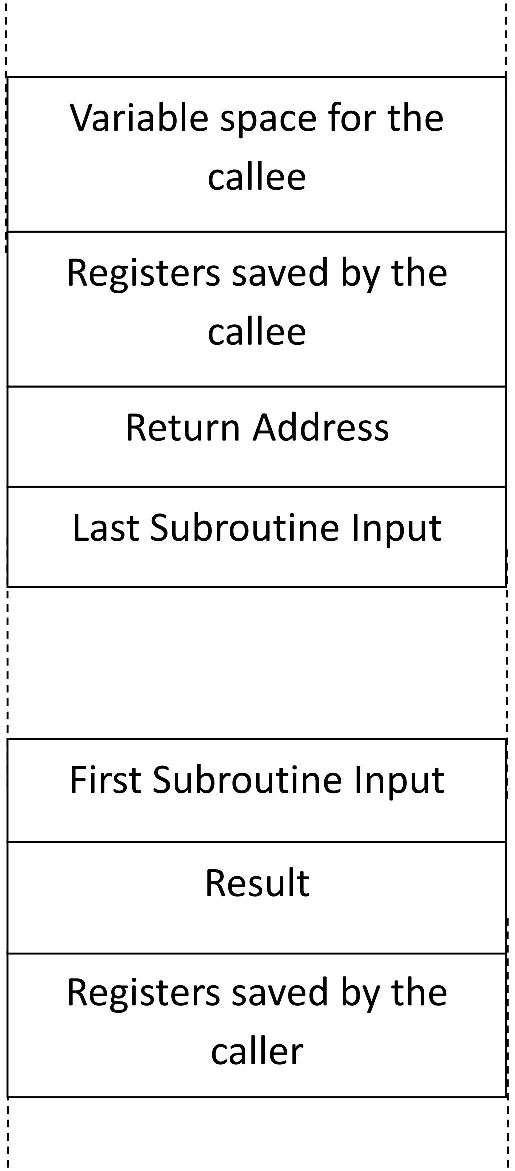
```
              ORG         $3000

Array   DC.B    $34, $98, $11, $DF
Length  DC.B    4
Result  DS.B    1         ; minimum value
        DS.B    1         ; maximum value

        ORG     $C000

        LDS     #$3600

        LDD     #Result
        PSHD
        LDD     #Array
        PSHD
        LDAB    Length
        PSHB
        JSR     MinMax
        LEAS    3,SP
        PULX
        SWI
```

Stack Frame
RAH
RAL
Len
ArrH
ArrL
ResH
ResL

```
MinMax  LDX     3,SP
        LDY     5,SP
        MOVB    #$FF,0,Y
        MOVB    #$00,1,Y
        LDAB    2,SP
Loop    BEQ     EndMM
        LDAA    0,X
        CMPA    0,Y
        BHS     skip1
        STAA    0,Y
Skip1   CMPA    1,Y
        BLS     Skip2
        STAA    1,Y
Skip2   INX
        DECB
        BRA     Loop
EndMM   RTS
```

The Stack Frame

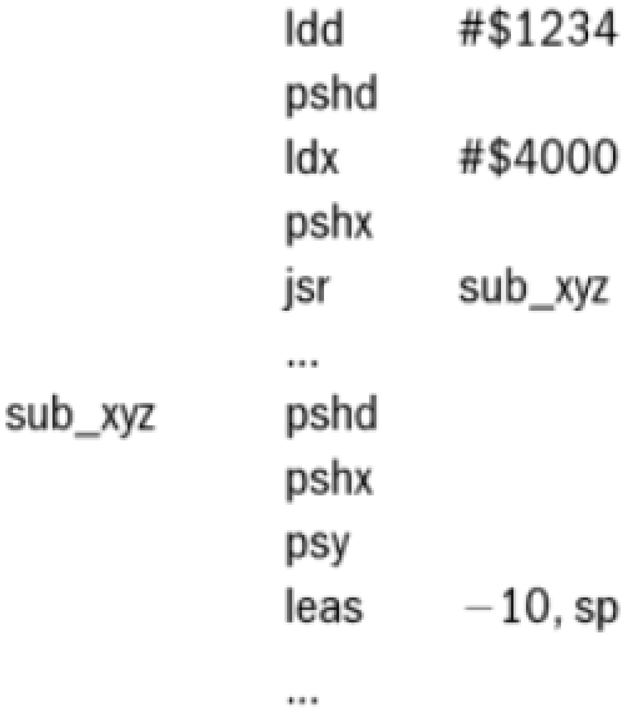
- With all the things that may end up on the stack in calling a subroutine, it’s a good idea to have a clear **order**.
- The diagram here shows the relative position of items in a **stack frame**.


Variable space for the callee
Registers saved by the callee
Return Address
Last Subroutine Input
First Subroutine Input
Result
Registers saved by the caller

The Stack Frame

Homework Example

- Draw the stack frame for the following program segment after the last **leas -10,sp** instruction is executed:



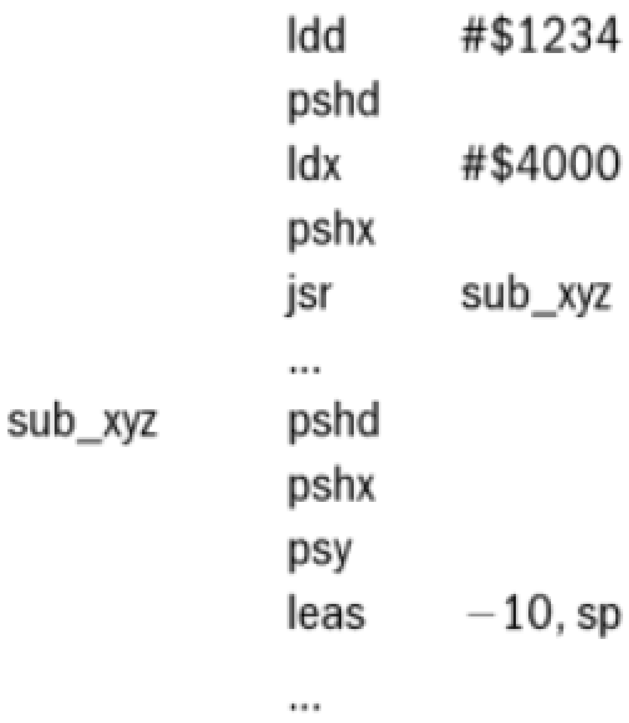
```
                ldd     #$1234
                pshd
                ldx     #$4000
                pshx
                jsr     sub_xyz
                ...
sub_xyz         pshd
                pshx
                psy
                leas    −10, sp
                ...
```

The Stack Frame

Homework Example

Question:

- Draw the stack frame for the following program segment after the last **leas -10,sp** instruction is executed.



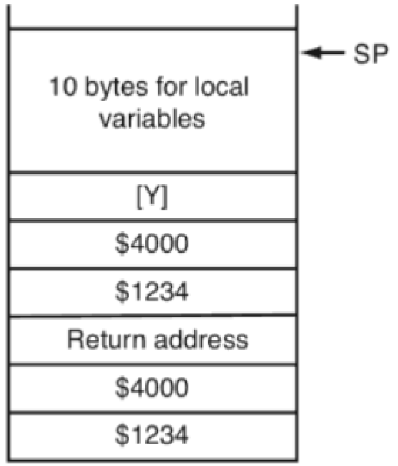
```
         ldd    #$1234
         pshd
         ldx    #$4000
         pshx
         jsr    sub_xyz
         ...
sub_xyz  pshd
         pshx
         psy
         leas   −10, sp
         ...
```

Solution:

- The **caller** pushes two l6-bit words into the stack.

- The **subroutine** sub_xyz saves three l6-bit registers in the stack and allocates 10 bytes in the stack.

- The resultant stack frame is shown here.



Questions?