In This Set of Slides:

• Data Structures (Stack, Arrays, Strings)
• Search of Sorted and Unsorted arrays
• Strings
• Subroutines
  – Usage Rules
  – Stack
  – Leas
  – Stack Frame
• Bubble Sort Example
• D-Bug12 I/O Functions
  – Printf function
Program = data structures + algorithm

Three Data structures to be discussed

1. Stack: a last-in-first-out data structure
2. Array: a set of elements of the same type
3. String: a sequence of characters terminated by a special character

• Stack:

![Diagram of the HCS12 stack](image)

Figure 4.1 Diagram of the HCS12 stack
Stack cont’d, Push and Pull Instructions

- The stack grows down in memory
- *pushes* pre-decrement while *pulls* post-increment.
- Note the equivalent instructions, which help explain what’s happening
- CCR push and pull have no equivalent instructions, so CCR can only be accessed via the stack

Table 4.1 HCS12 push and pull instructions and their equivalent load and store instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Equivalent instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>psha</td>
<td>push A into the stack</td>
<td>staa 1, -SP</td>
</tr>
<tr>
<td>pshb</td>
<td>push B into the stack</td>
<td>stab 1, -SP</td>
</tr>
<tr>
<td>pshc</td>
<td>push CCR into the stack</td>
<td>none</td>
</tr>
<tr>
<td>pshd</td>
<td>push D into stack</td>
<td>std 2, -SP</td>
</tr>
<tr>
<td>pshx</td>
<td>push X into the stack</td>
<td>stx 2, -SP</td>
</tr>
<tr>
<td>pshy</td>
<td>push Y into the stack</td>
<td>sty 2, -SP</td>
</tr>
<tr>
<td>pula</td>
<td>pull A from the stack</td>
<td>ldaa 1, SP+</td>
</tr>
<tr>
<td>pulb</td>
<td>pull B from the stack</td>
<td>ldab 1, SP+</td>
</tr>
<tr>
<td>pulc</td>
<td>pull CCR from the stack</td>
<td>none</td>
</tr>
<tr>
<td>puld</td>
<td>pull D from the stack</td>
<td>ldd 2, SP+</td>
</tr>
<tr>
<td>pulx</td>
<td>pull X from the stack</td>
<td>ldx 2, SP+</td>
</tr>
<tr>
<td>puly</td>
<td>pull Y from the stack</td>
<td>ldy 2, SP+</td>
</tr>
</tbody>
</table>
Indexable Data Structures

- Vectors (one dimension) and matrices (multi-dimensioned) are indexable data structures.

- First element of a vector is associated with the index 0 to facilitate the address calculation.

- Directives `db`, `dc.b`, `fcb` define arrays of 8-bit elements.

- Directives `dw`, `dc.w`, and `fdb` define arrays of 16-bit elements.
Example 4.2
Write a program to find out if the array vec_x contains a value, key. The array has 16-bit elements and is not sorted.

Figure 4.3 Flowchart for sequential search
Code for Search of an Unsorted Array

; looks for 16-bit key and if found stores the address at result, otherwise –1 is stored at result
; this program contains a true do until C loop
N equ 15 ; array count
notfound equ -1
key equ 190 ; define the searching key

org $1500
result rmw 1 ; reserve a word for result

org $2000
ldy #N ; set up loop count
ldd #notfound
std result ; initialize the search result with default of notfound, -1 or $FFFF
ldd #key
ldx #vec_x ; place the starting address of vec_x in X
loop cpd 2,X+ ; compare the key with array element & update pointer
beq found
dbne Y,loop ; have we gone through the whole array?
bra done ; only get to here if key is not found
found dex ; need to restore the value of X to point to the
dex ; matched element
stx result
done swi
vec_x dw 13,15,320,980,42,86,130,319,430,4, 90,20,18,55,30

Q. What will be the value in result after above has executed?
Binary Search of a Sorted Array

(Takes advantage of the fact array is sorted to increase efficiency/decrease execution time)

Algorithm: Compare key with middle element, if equal then done, if key > middle element then continue search in upper half of array, if key < middle element then continue search in lower half of array

For following: max, min, mean are pointers, not actual data

• Step 1: Initialize variables max and min to n -1 and 0, respectively.

• Step 2: If max < min, then stop since no element matches the key.

• Step 3: Let mean = (max + min)/2

• Step 4: If key = arr[mean], then key is found in the array, exit.

• Step 5: If key < arr[mean], then set max to mean - 1 and go to step 2.

• Step 6: If key > arr[mean], then set min to mean + 1 and go to step 2.
**Example 4.3** Write a program to implement the binary search algorithm for a sorted array and also a sequence of instructions to test it. (longer program than last but more efficient if you have a sorted array)

n equ 15 ; array count
key equ 83 ; key to be searched

org $1500
max rmb 1 ; maximum index value for comparison
min rmb 1 ; minimum index value for comparison
mean rmb 1 ; the average of max and min
result rmb 1 ; search result

org $2000
clda
staa min ; initialize min to 0 (i.e., point to first number in array)
staa result ; initialize result to 0
ldaa #n-1
staa max ; initialize max to n-1 (i.e., point to last number in array)
ldx #arr ; use X as the pointer to the array
loop ldamx
cmpb max ;Long Branch to notfound if min > max
lph notfound
addb max ; compute mean
lsrb ; “ (max + min)/2

;Continued on next slide
Binary Search continued

```
stab mean ; save mean
ldaa b,x ; A ← element arr[mean] uses B, mean, as offset
cmpa #key
beq found
bhi search_lo
ldaa mean
inca
staa min ; place mean+1 in min to continue
bra loop
search_lo ldaa mean
deca
staa max
bra loop
found ldaa #1
staa result
notfound swi
arr db 1,3,6,9,11
db 61,63,64,65,67
db 80,83,85,88,90
end
```
Strings

- **String def.**: A sequence of characters terminated by a NULL (ASCII code 0) or other special character such as EOT (ASCII code 4).

- To be understood, a binary number must be converted to ASCII

- Conversion method: divide the binary number by 10 repeatedly until the quotient is zero. $30$ is added to each remainder.

- **Example 4.4** Write a program to convert the unsigned 8-bit binary number in accumulator A into BCD digits terminated by a NULL character. Each digit is represented in ASCII code.

**Solution:**
- For 8 bits, the largest number would be 255, thus 4 bytes, *including* the null, are needed to hold the converted BCD digits.
- Repeated division by 10 method is used.
- See program on next page.
1. test_dat equ 34
2. org $1000
3. buf db 4 ; to hold the decimal string
4. temp db 2 ;
5. org $2000
6. lds #$2000 ; initialize SP (recall stack goes down in memory)
7. ldab #test_dat
8. ldy #buf ; use Y to point to decimal string
9. tstb
10. bne normal
11. movb #$30,buf ; store ascii 0 (30) but get here only if test_dat = 0
12. clr buf+1 ; terminate the string with an actual, not ascii, zero
13. bra done
14. normal movb #0,1,-sp ; store the NULL delimiter in the stack
15. clra
16. loop ldx #10
17. idiv
18. addb #$30 ; convert to ASCII code (rem in D but no bigger than B)
19. pshb ; push into stack
20. cpx #0 ; get out of loop when quotient is finally 0
21. beq reverse ;
22. xgdx ; otherwise, put quotient back in B for next division
23. bra loop
24. reverse tst 0,sp ; move numbers in reverse order into buf
25. beq done ; done when NULL byte reached
26. movb 1,sp+,1,y+
27. bra reverse
28. done swi
29. end
Example 4.6: Convert an ASCII String Representing a BCD Number Into a Signed Binary Number

• Algorithm

Step 1
sign ← 0
error ← 0
number ← 0

Step 2
If the character pointed to by in_ptr is the minus sign, then
   sign ← 1
   in_ptr ← in_ptr + 1

Step 3
If the character pointed to by in_ptr is the NULL character,
   then go to step 4.
else if the character is not a BCD digit, then
   error ← 1; go to step 4;
else
   number ← number * 10 + m[in_ptr] - $30;
   in_ptr ← in_ptr + 1;
   go to step 3;

Step 4
If sign = 1 and error = 0 , then
   number ← two’s complement of number;
else
   stop;

See program on next slide
ASCII String to Signed Binary

1. minus equ $2D ; ASCII code of minus sign
2. org $1000
3. in_buf fcc "9889" ; input ASCII to be converted
4. dB 0 ; null character to terminate ASCII
5. out_buf db 2 ; holds the converted binary value
6. buf2 db 1 ; holds a zero
7. buf1 db 1 ; holds the current digit value
8. sign db 1 ; holds the sign of the number
9. error db 1 ; indicates the occurrence of illegal character
10. org $1500
11. clr sign
12. clr error
13. clr out_buf
14. clr out_buf+1
15. clr buf2
16. ldx #in_buf
17. ldaa 0,x
18. cmpa #minus ; is the first character a minus sign?
19. bne continue ; branch if not minus
20. inc sign ; set the sign to 1
21. inx ; move the pointer
22. continue ldaa 1,x+ ; is the current character a NULL character?
23. lbeq done ; yes, we reach the end of the string
24. cmpa #$30 ; is the character not between 0 to 9?
ASCII String to Signed Binary Cont’d.

1. lblo in_error ; get out if number not valid, below 0
2. cmpa #$39 ; 
3. lbhi in_error ; get out if number not valid, above 9
4. suba #$30 ; convert to the BCD digit value
5. staa buf1 ; save the digit temporarily
6. ldd out_buf
7. ldy #10
8. emul ; Y:D ← D * Y
9. addd buf2 ; add the current digit value
10. std out_buf ; Y holds 0 and should be ignored
11. bra continue
12. in_error ld aa #1
13. staa error
14. done ld aa sign ; check to see if the original number is negative
15. beq positive
16. ld aa out_buf ; if negative, compute its two’s complement
17. ldab out_buf+1 ; 
18. coma ; 
19. comb ; 
• addd #1 ; 
• std out_buf
• positive swi 
• end
Subroutines

- A sequence of instructions called from various places in the program
- Allows the same operation to be performed with different parameters
- Simplifies the design of complex program by using ‘divide and conquer’

Instructions related to subroutine calls:
- `bsr <rel>` ; branch to subroutine
- `jsr <opr>` ; jump to subroutine
- `rts` ; return from subroutine
- `call <opr>` ; used for expanded memory
- `rtc` ; return from subroutine
Program Structure w/Subroutines

Notes:
1. Main will call various subroutines but also a subroutine can call another, for example subroutine 2.1 could call 3.1
2. A subroutine calling itself is ‘recursion’ but you’ve got to know what you are doing!

Figure 4.7 A structured program
General Subroutine Processing

Figure 4.8 Program flow during a subroutine call
Important Subroutine Issues

• **Keep subroutines independent/portable**
  - *Do not* use direct or extended addressing
  - Keep in mind the subroutine may be called from numerous locations including other subroutines

• **Know how a subroutine affects registers or make sure that it doesn’t**
  - Comments should be used at beginning of routine to aid in writing the caller
  - If needed, push registers on stack at beginning of a subroutine and pull them just before *rts* or *rtc*

• **Parameter passing, to or from subroutine**
  - By registers: send/receive actual data and/or address pointers
  - By stack: send/receive actual data and/or address pointers via the stack but make sure SP points to the return address when *rts* is executed
Two Ways of Preserving Registers
Discuss: advantages and disadvantages of each

1. Incorporate **saving in subx**:  
   
   bsr subx
   
   bsr subx
   
   swi
   
   subx psha ;saving a and x
   pshx
   
   pulx restoring a and x
   pula
   
   rts
   
   end

2. Incorporate **saving in main**:  
   
   psha
   pshx
   
   bsr subx
   pulx
   pula
   
   psha
   pshx
   
   bsr subx
   pulx
   pula
   
   swi
   
   subx
   
   rts
   
   end
In Class Exercise Regarding the Stack
List file is on next slide
Show stack values as program is executed
What would happen if a *psha* were placed between lines 12 and 13?

1. ;web_stackex.asm
2. org $1500
3. sum rmb 1
4. org $2000
5. lds #$2000
6. ldaa #12
7. ldab #15
8. jsr subra
9. staa sum
10. swi
11. subra aba
12. jsr subrb
13. rts
14. subrb clc
15. sbca #11 ;there is no immediate subtraction w/o carry
16. rts
17. end
In Class Exercise Regarding the Stack Cont’d.

1. as12, an absolute assembler for Motorola MCU's, version 1.2e

2. ;web_stackex.asm

3. 1500 org $1500

4. 1500 sum rmb 1

5. 2000 org $2000

6. 2000 cf 20 00 lds #$2000

7. 2003 86 0c ldaa #12

8. 2005 c6 0f ldab #15

9. 2007 16 20 0e jsr subra

10. 200a 7a 15 00 staa sum

11. 200d 3f swi

12. 200e 18 06 subra aba

13. 2010 16 20 14 jsr subrb

14. 2013 3d rts

15. 2014 10 fe subrb clc

16. 2016 82 0b sbca #11 ;there is no immediate subtraction w/o carry

17. 2018 3d rts

18. end
1. ;webex4.asm, using eg04rev as a subroutine
2. ;given three arrays of 8-bit numbers, look for a key value in each
3. ;if found place address in result, if key not found, store –1 in result
4. ;based on modifying ex. 4.2 in text and making it a subroutine
5. N1 equ 4
6. N2 equ 5
7. N3 equ 4
8. key equ 100
9. org $1500
10. result1 rmw 1 ;storing addresses, thus need to reserve words
11. result2 rmw 1
12. result3 rmw 1
13. org $2000
14. lds #$2000 ;code grows up in memory and stack grows down/not interfering
15. ldx #array1 ;preparing to call search for the first time
16. ldaa #N1 ;“
17. ldab #key ;“ (not incorporating into search to keep the subr. Independent)
18. bsr search ; using relative addressing because subroutine is close
19. stx result1
20. ldx #array2 ;array2 search starts here
21. ldaa #N2
22. ldab #key
23. bsr search
24. stx result2
25. ldx #array3 ;array3 search starts here
26. ldaa #N3
27. ldab #key
28. bsr search
29. stx result3
30. swi ; Actual exit from program

;Test Data and Subroutine SEARCH IS ON NEXT SLIDE
;-------------------subroutine search-------------------------------
;on entry: x contains pointer to array
; a contains N
; b contains key value
; on return: x contains result (Address or -1 if not found)

search  nop
loop    cmpb 1,x+ ;x(i) = key?
        beq found
        dbne a,loop ;if not, decrement counter and continue
        ldx #$fff ; only executed if key not in array
        rts ;
found  dex ;restore X so it points to matched value
        rts ;this rts is executed if key is found in data

array1  db 3,66, 100,44
array2  db 2,150,30,55,88
array3  db 200,100,56,109
end

NOTE the print screen on the next page which shows:
1. Disassembly to see where data is stored: starting at $2036
2. Program execution and memory display of results: 20 38, FF FF (-1), and 20 40
3. The stack showing the last address which was stored in the stack: 20 28
   (address following last bsr)
Using **leas** (**Load Effective Address into SP**) 

– Local variables **allocation** (by caller)  
  • **leas**  \(-n,sp\)  ; efficiently allocates \(n\) bytes in the stack for local variables by decrementing SP

– Local variables **de-allocation** (by subroutine)  
  • **Leas**  \(n,sp\)  ; efficiently de-allocates \(n\) bytes from the stack
Stack Frame
(also called activation record)

- Def: The region in the stack that holds incoming parameters, the subroutine return address, local variables, and saved registers

![Figure 4.9 Structure of the 68HC12 stack frame](image-url)
Example 4.10 rev’d. Draw the stack frame for the following program segment after the leas –10,sp instruction is executed.

1. ldd #$1234 ;1st onto stack
2. pshd
3. ldx #$4000 ;2nd onto stack
4. pshx
5. jsr sub_xyz
6. ...
7. sub_xyz pshd
8. pshx
9. pshy
10. leas  -10,sp ; allocate space
11. ...
12. ; now sp can be used as a pointer
13. ; such as stab 2,sp; stores B at 1 of the 10 locations
     ...
14. leas  +10,sp ;de-allocate space
15. puly, etc.
   rts

Figure 4.10 Stack frame of example 4.10

10 bytes for local variables

contents of y

$1234

return address

$4000

$1234
Bubble Sort Algorithm
for sorting N elements in ascending order
(inefficient but straightforward)

1. If x[i] > x[i+1], then switch
2. Inc i
3. If n-1 comparisons, go to 4, otherwise return to step 1
4. n <= N – 1 (last element guaranteed to be max and no need to examine again)
5. If n = 0, exit, otherwise return to

Enhancement: Use a flag, for each pass, which is tested to see if any exchange made and, if not, discontinue because array is sorted
n equ 10
org $1500
array db $ED,$33,$44,$22,$00,$75,$15,$5A,$12,$AA
org $2000
ldx #array
ldy #n
jsr bublsort
swi
• ;***subroutine bublsort***
• ;on entry>
• ; x points to array (assumes unsigned numbers)
• ; y contains N
• ;on return>
• ; array has been sorted into ascending order
• ; registers A, B, X, and Y are changed

1. bublsort pshx
2. dey ; n-1 comparisons
3. pshy
4. beq done ; depends on pshy not affecting flags
5. loop ldaa 0,x
6. cmpa 1,x
7. bls contin
8. ldab 1,x
9. stab 0,x
10. staa 1,x
11. contin inx
12. dbne y,loop
13. swi ; stops here for testing purposes
14. puly
15. pulx
16. bra bublsort
17. done leas 4,sp ; reset SP because of two pushes
18. rts

NOTE Execution on next slide: maximum values bubbling to top and the final X value less each time because Y is decremented after each loop
User Ekpt Encountered

PP  PC  SP  X   Y   D = Å: B  CCR = SXHI NZVC
38 2020 3BFA 1509 0000 ED: AA  1001 1000
xx: 2020 31       PULY

>ad 1500

1500 33 44 22 00 - 75 15 5A 12 - Å: ED 74 6E - 23 94 7A 63 3D". u.Z...tk#.zc
>g

User Ekpt Encountered

PP  PC  SP  X   Y   D = Å: B  CCR = SXHI NZVC
38 2020 3BFA 1508 0000 75:12 1001 1011
xx: 2020 31       PULY

>ad 1500

1500 33 22 00 44 - 15 5A 12 75 - Å: ED 74 6E - 23 94 7A 63 3".D.Z.u..tk#.zc
>g

User Ekpt Encountered

PP  PC  SP  X   Y   D = Å: B  CCR = SXHI NZVC
38 2020 3BFA 1507 0000 5A:12 1001 1001
xx: 2020 31       PULY

>g

User Ekpt Encountered

PP  PC  SP  X   Y   D = Å: B  CCR = SXHI NZVC
38 2020 3BFA 1506 0000 44:12 1001 1001
xx: 2020 31       PULY

>ad 1500

1500 00 22 15 33 - 12 44 5A 75 - Å: ED 74 6E - 23 94 7A 63 3".DZu..tk#.zc
>
### Using the D-Bug12 Functions for I/O

Table 4.2 D-Bug12 monitor (version 4.x.x) routines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Function</th>
<th>pointer address</th>
</tr>
</thead>
<tbody>
<tr>
<td>far main ()</td>
<td>Start of D-Bug12</td>
<td>$EE80</td>
</tr>
<tr>
<td>getchar ()</td>
<td>Get a character from SCI0 or SCI1</td>
<td>$EE84</td>
</tr>
<tr>
<td>putchar ()</td>
<td>Send a character out to SCI0 or SCI1</td>
<td>$EE86</td>
</tr>
<tr>
<td>printf ()</td>
<td>Formatted string output-translates binary values to string</td>
<td>$EE88</td>
</tr>
<tr>
<td>farGetCmdLine ()</td>
<td>Get a line of input from the user</td>
<td>$EE8A</td>
</tr>
<tr>
<td>far sscanfhex ()</td>
<td>Convert ASCII hex string to a binary integer</td>
<td>$EE8E</td>
</tr>
<tr>
<td>isxdigit ()</td>
<td>Check if a character (in B) is a hex digit</td>
<td>$EE92</td>
</tr>
<tr>
<td>toupper ()</td>
<td>Convert lower-case characters to upper-case</td>
<td>$EE94</td>
</tr>
<tr>
<td>isalpha ()</td>
<td>Check if a character is alphabetic</td>
<td>$EE96</td>
</tr>
<tr>
<td>strlen ()</td>
<td>Returns the length of a NULL-terminated string</td>
<td>$EE98</td>
</tr>
<tr>
<td>strcpy ()</td>
<td>Copy a NULL-terminated string</td>
<td>$EE9A</td>
</tr>
<tr>
<td>far out2hex ()</td>
<td>Output 8-bit number as 2 ASCII hex characters</td>
<td>$EE9C</td>
</tr>
<tr>
<td>far out4hex ()</td>
<td>Output a 16-bit number as 4 ASCII hex characters</td>
<td>$EEA0</td>
</tr>
<tr>
<td>SetUserVector ()</td>
<td>Setup a vector to a user's interrupt service routine</td>
<td>$EEA4</td>
</tr>
<tr>
<td>farWriteEEByte ()</td>
<td>Write a byte to the on-chip EEPROM memory</td>
<td>$EEA6</td>
</tr>
<tr>
<td>farEraseEE ()</td>
<td>Bulk erase the on-chip EEPROM memory</td>
<td>$EEAA</td>
</tr>
<tr>
<td>far ReadMem ()</td>
<td>Read data from the HCS12 memory map</td>
<td>$EEAE</td>
</tr>
<tr>
<td>far WriteMem ()</td>
<td>Write data to the HCS12 memory map</td>
<td>$EEB2</td>
</tr>
</tbody>
</table>
Rules for using D-Bug12 I/O Functions

(All functions listed in Table 4.2 are written in C language.)

- The first parameter to the function is passed in accumulator D. The rest are pushed onto the stack in the reverse order they are listed in the function declaration.

- Parameters of type `char` will occupy the lower order byte of a word pushed onto the stack and must be converted to type `int`.

- Parameters pushed onto the stack before the function is called remain on the stack when the function returns. The caller “removes” passed parameters from the stack using the LEAS instruction.

- All 8- and 16-bit values are returned in accumulator D. A returned value of type `char` is returned in accumulator B. Boolean function results are 0 for false and non-zero for true.

- Registers are not preserved and, if needed, must be saved on the stack before calling the function.
Using the `printf` function

Notes on Next Slide:

- Uses the `printf` to send a message and data to the terminal

- By putting `printf` in a loop, one can print an array of numbers

- As required the last number (num2) printed is the first pushed on the stack

- An error occurred in assembling: “delimiter missing” due to improper quotes at beginning of string, retyped and was ok.

- Extra line feeds and carriage returns were added to provide space after output

- The values are converted to decimal before printing.
```assembly
printf equ $1E88 ; function call to output a character
CR equ $0D
LF equ $0A
num1 equ $23
num2 equ $4A
org $1500
msg "The value of num1 is %d and num2 is %d."
fcb CR,LF,CR,LF,CR,LF
fcb 0 ; end of string character
org $2000
lds #$2000
ldd #num2 ; note last must go in stack first
pshd
lab $num1 ; now the first can go on the stack
pshd
ldd #msg
ldx printf
jsr 0,x
leas 4,sp ; remove data pushed on stack (reset stack pointer)
swi
end
```