CPE/EE 421/521
Fall 2004
Chapter 3 – Assembly Language and C

Dr. Rhonda Kay Gaede

UAH

3.1 Parameter Passing

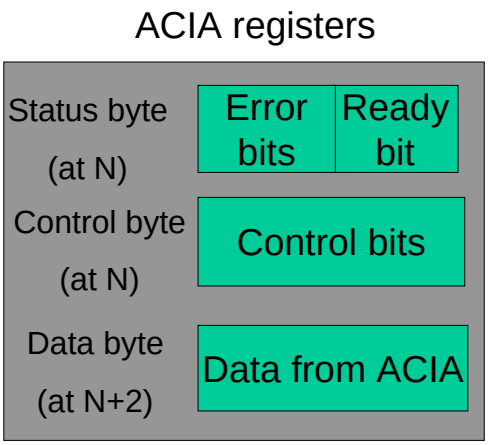
- We are interested in:
  - _____
  - _____
  - _____
  - _____
  - _____

3.2 Parameter Passing – ACIA Example

To initialize:
- write #03 to CR
- write conf. byte to CR

To read:
- polling on Ready bit

If no input:
- poll for a specified number of times before exit with an error



3.1 Parameter Passing – ACIA Subroutine in Pseudocode

```
Character_Input(Func, Dev_loc, Input_Char, Error_St)
  Error_St = 0
  IF Func = 0
  THEN Initialize Input_Dev
  ELSE
    Read status of Input_Dev
     IF status OK
    THEN
       Set Cycle_Count to max value
       REPEAT
         Read status of Input_Dev
         Decrement Cycle_Count
       UNTIL Input_Dev is ready OR Cycle_Count = 0
       Input_Char = input from Input_Device
       IF Cycle_Count = 0
       THEN Error_St = $FF
   ELSE Error_St = status from Input_Dev
End Character_Input
```

3.1 Parameter Passing – ACIA Subroutine in 68000 Assembly

```
* ACIA_Initialize and Character_Input routine
* Data register D0 contains Function
* (zero = initialize, non-zero = get a character)
* Data register D0 is re-used for the Cycle_Count
* (a timeout mechanism)
* Data register D1 returns Error_Status
* Data register D2 returns the character from the ACIA
* Data register D3 is temporary storage for the ACIA's status
* Data register D4 is temporary storage for the masked ACIA's
  status (error bits)
* Address register A0 contains the address of the ACIA's
  control/status register
*
Char_In MOVEM.W D3-D4,-(A7)   Push working registers on the stack
        CLR.B   D1            Start with Error_Status clear
        CMP.B   #0,D0         IF Function not zero THEN get input
        BNE     InPut         ELSE initialize ACIA
        MOVE.B  #3,(A0)       Reset the ACIA
        MOVE.B  #$19,(A0)     Configure the ACIA
        BRA     Exit_2        Return after initialization
```


3.1 Parameter Passing – ACIA Subroutine in 68K Assembly (continued)

```
*
InPut   MOVE.W  #$FFFF,D0     Set up Cycle_Count for time-out
*                             (reuse D0)
InPut1  MOVE.B  (A0),D3       Read the ACIA's status register
        MOVE.B  D3,D4         Copy status to D4
        AND.B   #%01111100,D4 Mask status bits to error conditions
        BNE     Exit_1        IF status indicates error, set error
                              flags & return
        BTST    #0,D3         Test data_ready bit of status
        BNE     Data_Ok       IF data_ready THEN get data
        SUBQ.W  #1,D0                       ELSE Cycle_Count--
        BNE     InPut1        IF not timed out THEN repeat
        MOVE.B  #$FF,D1                        ELSE Set error flag
        BRA     Exit_2                              and return
*
Data_Ok MOVE.B  (2,A0),D2     Read the data from the ACIA
        BRA     Exit_2        and return
*
Exit_1  MOVE.B  D4,D1         Return Error_Status
Exit_2  MOVEM.W (A7)+,D3-D4   Restore working registers
        RTS                   Return
```

3.1 Parameter Passing – ACIA Example Passed Parameters via Registers

- Two registers are used in the subroutine and have to be saved on the stack:

  `MOVE.W D3-D4,-(A7)`

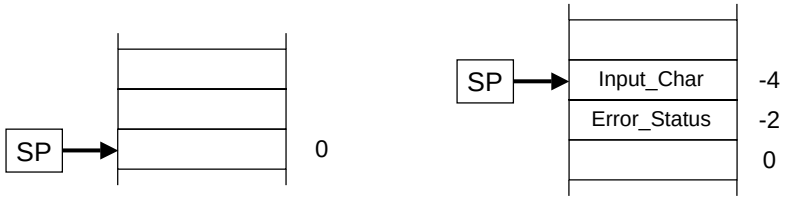
  (otherwise, data would be lost)
- D0 is simply reused without saving, because the old data is not needed
- PROS:
  - _____
  - _____
- CONS:
  - _____
  - _____
    _____

3.1 Parameter Passing – Mechanisms for Passing Parameters via the Stack

- Passing parameters by value
  - _____
  - _____
    _____
- Passing parameters by reference
  - _____
  - _____
  - _____

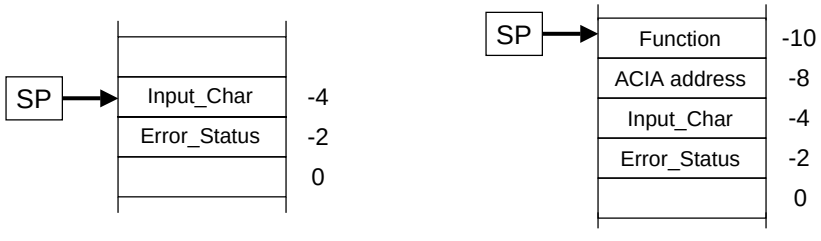
3.1 Passing Value Parameters via the Stack – Getting Ready to Call (Phase 1)

```
LEA     (-4,A7),A7   Save space on stack for
                     Error_Status and Input_Char
```



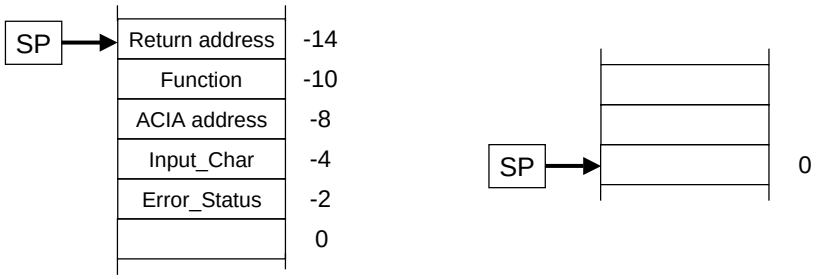
3.1 Passing Value Parameters via the Stack – Getting Ready to Call (Phase 2)

```
MOVE.L  #ACIA,-(A7)     Push ACIA address on the stack
MOVE.W  Func,-(A7)      Push function code on the stack
```



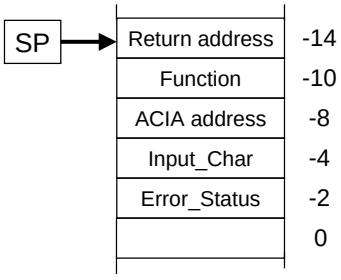
3.1 Passing Value Parameters via the Stack – Cleaning up After Call

```
BSR     Char_In    Call subroutine
LEA     (6,A7),A7  Clean up stack - remove parameters
                   Function/ACIA
MOVE.W  (A7)+,Char Pull the input character off the stack
MOVE.W  (A7)+,Err  Pull the Error_Status off the stack
```



3.1 Passing Value Parameters via the Stack – During Subroutine

```
* Character_Input and ACIA_Initialize routine
* D3 is temporary storage for the ACIA's status
* D4 is temporary storage for the Cycle_Count
* A0 contains the address of the ACIA's control/status register
*
Char_In MOVEM.L A0/D3-D4,-(A7) Push working registers onto stack
        MOVE.L  (18,A7),A0     Read address of ACIA from stack
        CLR.B   (24,A7)        Start w/ Error_Status clear
```

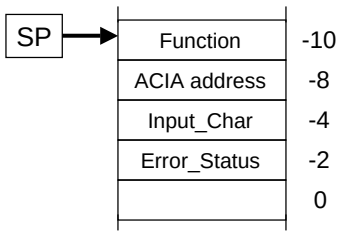


3.1 Passing Value Parameters via the Stack – During Subroutine

```
        CMPI.B  #0,(16,A7)      IF Function not zero THEN get input
        BNE     InPut                   ELSE initialize ACIA
        MOVE.B  #3,(A0)
        MOVE.B  #$19,(A0)
        BRA     Exit_2          Return after initialization
*
InPut   MOVE.W  #$FFFF,D0       Set up Cycle_Count for time-out
                                        (reuse D0)
InPut1  MOVE.B  (A0),D3         Read the ACIA's status register
        MOVE.B  D3,D4           Copy status to D4
        AND.B   #%01111100,D4   Mask status bits to error conditions
        BNE     Exit_1          IF error indication, deal with it
        BTST    #0,D3           Test data_ready bit of saved status
        BNE     Data_OK         IF data_ready THEN get data
        SUBQ.W  #1,D0                       ELSE Cycle_count--
        BNE     InPut1          IF not timed out THEN repeat
        MOVE.B  #$FF,(24,A7)                ELSE Set error flag
        BRA     Exit_2                           and return
```


3.1 Passing Value Parameters via the Stack – During Subroutine

```
Data_OK MOVE.W  (2,A0),(22,A7) Read the data from the ACIA
*                              and put on the stack
        BRA     Exit_2         and return
*
Exit_1  MOVE.B  D4,(24,A7)     Return Error_Status
Exit_2  MOVEM.L (A7)+,A0/D3-D4 Restore working registers
        RTS                    Return
```



3.1 Passing Reference Parameters via the Stack – Readying, Calling, Cleaning

```
PEA     Func            Push Function address on the
                         stack
PEA     ACIA            Push ACIA address on the stack
PEA     Error_Status    Push address of Error_Status
PEA     Char            Push address of input data
BSR     Char_In         Call subroutine
LEA     (16,A7),A7      Clean up the stack - remove the
                         four addresses
```


3.1 Passing Reference Parameters via the Stack – Starting the Subroutine

```
*
* D0 is temporary storage for the timeout counter
* D3 is temporary storage for the ACIA's status
* D4 is temporary storage for the Cycle_Count
* A0 points at the location of the character input from the ACIA
* A1 points at the location of the Error_Status
* A2 points at the location of the ACIA
* A3 points at the location of the Function code
*
Char_In MOVEM.L A0-A3/D0/D3-D4,-(A7) Push working regs on the stack
```

3.1 Passing Reference Parameters via the Stack – More of the Subroutine

```
        MOVEA.L (32,A7),A0      Read address of Char from the stack
        MOVEA.L (36,A7),A1      Read address of Error_Status
        MOVEA.L (40,A7),A2      Read address of ACIA from the stack
        MOVEA.L (44,A7),A3      Read address of Function
        CLR.B   (A1)            Start with Error_Status clear
        CMPI.B  #0,(A3)         IF Function not zero THEN get input
        BNE     InPut                   ELSE initialize ACIA
        MOVE.B  #3,(A2)
        MOVE.B  #$19,(A2)
        BRA     Exit_2          Return after initialization
*
InPut   MOVE.W  #$FFFF,D0       Set up Cycle_Count for timeout
InPut1  MOVE.B  (A2),D3         Read the ACIA's status register
        MOVE.B  D3,D4           Copy status to D4
        AND.B   #%01111100,D4   Mask status bits to error conditions
        BNE     Exit_1          IF error, set flags and return
        BTST    #0,D3           Test data_ready bit of status
        BNE     Data_OK         IF data_ready THEN get data
        SUBQ.W  #1,D0                  ELSE decrement Cycle_Count
        BNE     InPut1          IF not timed out THEN repeat
        MOVE.B  #$FF,(A1)                      ELSE Set error flag
        BRA     Exit_2                  and return
```

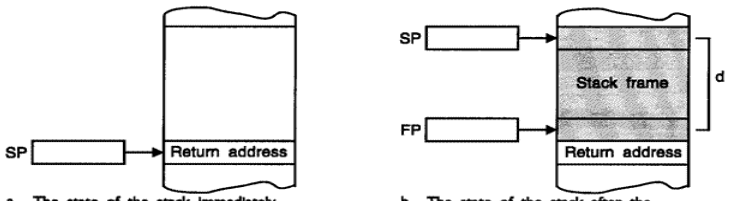

3.1 Passing Reference Parameters via the Stack – Finishing the Subroutine

```
Data_OK MOVE.W (2,A2),(A0)              Read data from ACIA
        BRA    Exit_2
*
Exit_1  MOVE.B D4,(A1)                  Return Error_Status
Exit_2  MOVEM.L (A7)+,A0-A3/D0/D3-D4    Restore working
*                                       registers
        RTS
```

3.2 The Stack and Local Variables

- Subroutines often need _____
- We can use a fixed block of memory space – _____ _____ – but:
  - The code will not be _____
  - The code will not be _____
  - The code will not be _____
- Better solution: _____
  - Allocate all local variables _____
  - **STACK FRAME** = _____ _____
  - **FRAME POINTER** = _____ _____

3.2 The Stack and Local Variables – Stack Frame via SP



a. The state of the stack immediately after a subroutine call.

b. The state of the stack after the allocation of a stack frame by executing LEA (-d,SP),SP.

It can be done simply by modifying the stack pointer:

```
AnySub  LEA   (-4,A7),A6     Set up A6 as the frame pointer
        LEA   (-200,A7),A7   Create the stack frame
        .
        .                    The subroutine proper
        .
        LEA   (200,A7),A7    Collapse the stack frame
        RTS                  and return from subroutine
```

3.2 The Stack and Local Variables – Stack Frame via LINK and UNLK

- **LINK** and **UNLK** automate the creation and removal of the stack frame

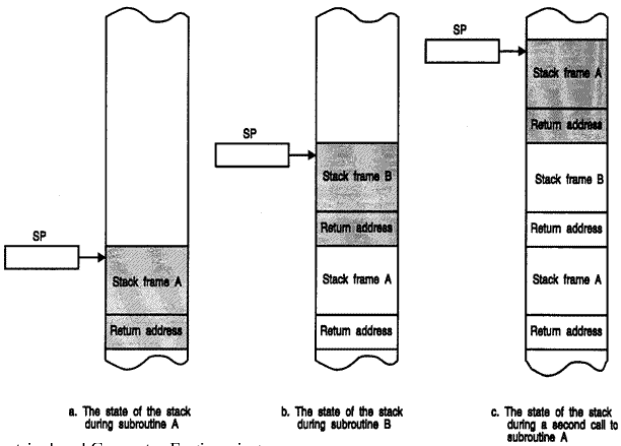
```
Sub1  LINK   A1,#-64    Allocate 64 bytes (16 long words) of storage
      .                 in this stack frame - use A1 as frame pointer
      .
      .                 Body of the subroutine
      .
      UNLK   A1         De-allocate Subroutine 1's stack frame
      RTS               Return to calling point.
```

- Implementation

```
LINK: [SP]      ← [SP] - 4     Decrement the stack pointer by 4
      [M([SP])] ← [A1]         Push the contents of address register A1
      [A1]      ← [SP]         Save stack pointer in A1
      [SP]      ← [SP] - 64    Move stack pointer up by 64 locations

UNLK:   [SP]  ← [A1]
        [A1]  ← [M([SP])]
        [SP]  ← [SP] + 4
```


3.2 The Stack and Local Variables – Multiple Stack Frames

Nested subroutines: A calls B, then B calls A



a. The state of the stack during subroutine A

b. The state of the stack during subroutine B

c. The state of the stack during a second call to subroutine A

3.2 The Stack and Local Variables – Readying, Calling, Cleaning Up

```
PEA     Char           Push address of dest. for the input
PEA     Error_Status   Push address of Error_Status message
PEA     ACIA           Push ACIA's address on the stack
MOVE.W  Function,-(A7) Push value of function code on the stack
BSR     Char_In        Call subroutine
LEA     (14,A7),A7     Clean up the stack - remove the four
                          parameters
```


3.2 The Stack and Local Variables – ACIA Subroutine

```
* Character_Input and ACIA_Initialize routine
* SF location A6 - 6 holds the ACIA's status
* SF location A6 - 4 holds the ACIA's masked status (error bits
                            only)
* SF location A6 - 2 holds the Cycle_Count
* A1 contains the address of the Error_Status
* A2 contains the address of the ACIA's control/status register
*
Char_In LINK    A6,#-6         Create a stack frame for three words
        MOVEM.L A1-A2,-(A7)    Push working registers on the stack
        MOVEA.L (14,A6),A1     Read address of Error_Status from
*                               the stack
        MOVEA.L (10,A6),A2     Read address of ACIA from the stack
        CLR.B   (A1)           Clear Error_Status
        MOVE.W  #$FFFF,(-2,A6) Set up Cycle_Count for timeout
        CMPI.B  #0,(8,A6)      IF Function not zero THEN get input
        BNE     InPut             ELSE initialize ACIA
        MOVE.B  #3,(A2)        Reset ACIA
        MOVE.B  #$19,(A2)      Configure ACIA
        BRA     Exit_2         Return after initialization
```

3.2 The Stack and Local Variables – ACIA Subroutine (concluded)

```
InPut   MOVE.B  (A2),(-4,A6)   Read the ACIA's status register –
*                              save in Temp1
        MOVE.B  (-4,A6),(-6,A6) Copy status to Temp2
        ANDI.B  #%01111100,(-6,A6) Mask status bits to error
        BNE     Exit_1         IF status indicates error,
                                      set flag and exit
        BTST    #0,(-4,A6)     ELSE Test data_ready bit of status
        BNE     Data_OK        IF data_ready THEN get data
        SUBQ.W  #1,(-2,A6)            ELSE decrement Cycle_Count
        BNE     InPut          IF not timed out THEN repeat
        MOVE.B  #$FF,(A1)                    ELSE Set error flag
        BRA     Exit_2                       and return
Data_OK MOVE.L  (18,A6),(A1)    Get address for data dest
        MOVE.B  (2,A2), (A1)    Read data from ACIA
        BRA     Exit_2
*
Exit_1  MOVE.B (-6,A6),(A1)     Return Error_Status
Exit_2  MOVEM.L (A7)+,A1-A2     Restore working registers
        UNLK A6
        RTS
```


3.3 C and the 68000

- Compiler and 68000 instruction set
- C data types and implementation
- Storage classes
- Functions and parameters
- Pointers

3.3 C and the 68000 – Compiling a C Program

```
void main (void)
{
   int i;
   int j;
   i = 1;
   j = 2;
   i = i + j;
}
```

```
* Comments
SECTION S_main,,"code"
XREF __main
   * Variable i is at -2(A6)
   * Variable j is at -4(A6)
XDEF _main
_main
LINK A6,#-4     *2 {
                *3 int i;
                *4 int j;
MOVE #1,-2(A6)  *5 i = 1;
MOVE #2,-4(A6)  *6 j = 2;
MOVEQ.L #1,D1
ADDQ #2,D1      *7 i = i + j;
MOVE D1,-2(A6)  *8 }
UNLK A6
RTS
```


3.3 C and the 68000 – C Data Types

- The 68000 family supports three basic data types:
  - _____, _____, _____
  - Each can be interpreted as _____ or _____
- C built-in types:
  - _____, _____, _____, _____
  - Void – refers to the null data type
  - Implementation dependent!

Data type	C name	Width (b)	Range
integer	int	16	-32768 to 32767
short integer	short int	8	-128 to 127
long integer	long int	32	-2147483648 **to** 2147483647
unsigned integer	unsigned int	16	0 to 65535
character	char	8	0 to 255
single-precision floating point	float	32	10^{-38} to 10^{+38}
double-precision floating point	double	64	10^{-300} to 10^{+300}

3.3 C and the 68000 – C Data Types (continued)

- Local variables
  - Defined inside a function
  - Cannot be accessed from outside the function
  - Normally lost when a return from the function is made
- Global variables
  - Defined outside a function
  - Can be accessed both from inside and outside the function
- Variables defined in a block exist only within that block

```
int i; /*global variable, visible to everything from this point*/
void function_1(void) /*A function with no parameters*/
{   int k; /*Integer k is local to function_1*/
    {    int q; /*Integer q exists only in this block*/
         int j; /*Integer j is local and not the same as j in main*/
    }
}
void main(void)
{   int j; /*Integer j is local to this block within function main*/
} /*This is the point at which integer j ceases to exist*/
```


3.3 C and the 68000 – Storage Classes

- Storage class specifiers
  - **`auto`**
    - Variable is no longer required once a block has been left; _____
  - **`register`**
    - Ask compiler to allocate the variable to _____
    - Also is _____
    - Cannot be accessed by means of _____
  - **`static`**
    - Allows local variable to _____ when a block is reentered
    - Initialized only _____
  - **`extern`**
    - Indicates that the variable is defined _____
    - The same global variable can be defined in _____ module

3.3 C and the 68000 – Storage Classes (continued)

- Access Modifiers
  - **`volatile`**
    - To define variables that can be changed externally
    - Compiler will not put them in registers
    - Think about Status Registers!
  - **`const`**
    - Variable may not be changed during the execution of a program
    - Cannot be changed unintentionally, but CAN be changed externally (as a result of an I/O, or OS operations external to the C program)
- Type conversion
  - In C, done either _____ or _____ (_____)

```
X   DS.L  1   Reserve a longword for X
Y   DS.W  1   Reserve a word for Y
```

```
USUALY WRONG
    MOVE.L   X,D0
    ADD.W    Y,D0
```

```
CORRECT
    MOVE.W   Y,D0
    EXT      D0
    ADD.L    X,D0
```


3.3 C and the 68000 – Returning a Value from a Function

- Example: **`main`** calls function **`adder`**
  - **`adder`** function has 2 _____ parameters (x and y)
  - Formal parameters behave like local variables within the function
  - When the function is called, _____ parameters are replaced by the values of the _____ parameters (a and b)

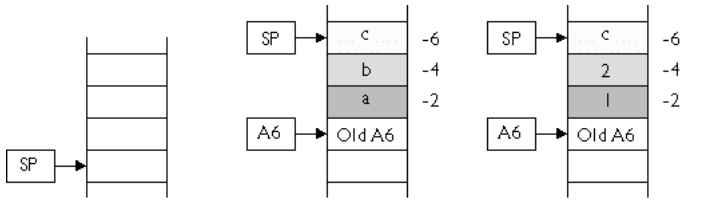
```
int adder(int x, int y) /* returns an integer */
{
  return x + y; /* return sum of x and y to the calling program */
}

void main (void)
{
  register int a, b, c; /* assign variables a, b, and c to regs */
  a = 1; b = 2;       /* provide some dummy values for a and b */
  c = adder(a, b); /* c is assigned the integer returned by adder */
}
```

3.3 C and the 68000 – Returning a Value from a Function (continued)

```
*1 int adder(int x, int y)
* Parameter x is at 8(A6)
* Parameter y is at 10(A6)
_adder
    LINK A6,#0
*2 {
*3 return x + y;
    MOVE 8(A6),D1
    ADD 10(A6),D1
    MOVE D1,D0
*4 }
    UNLK A6
    RTS
```

```
*5 void main (void)
* Variable a is at -2(A6)
* Variable b is at -4(A6)
* Variable c is at -6(A6)
_main
    LINK A6,#-6
*6 {
*7 int a, b, c;
*8 a = 1, b = 2;
    MOVE #1,-2(A6)
    MOVE #2,-4(A6)
*9 c = adder(a, b);
    MOVE #2,-(A7)
    MOVE #1,-(A7)
    JSR _adder
    MOVE D0,-6(A6)
*10 }
    UNLK A6
    RTS
```


3.3 C and the 68000 – Returning a Value from a Function Using the Stack

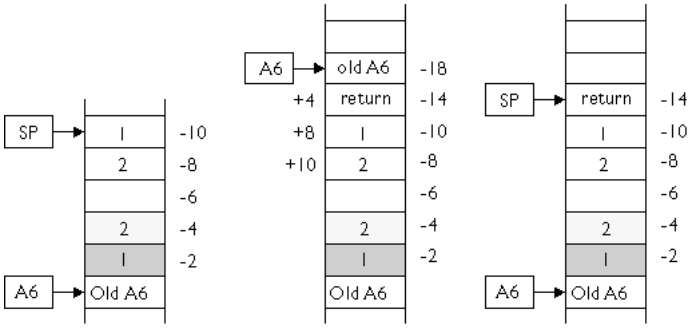


a. Initial state of the stack

b. The stack after LINK A6,#-6

c. Stack after setting up data
MOVE #1,-2(A6)
MOVE #2,-4(A6)

3.3 C and the 68000 – Returning a Value from a Function



d. Stack after copying data to stack with
MOVE #2,-(A7)
MOVE #1,-(A7)

e. Stack after calling subroutine and creating a stack frame with
JSR _adder
LINK A6,#0

f. Stack after collapsing stack frame with
UNLK A6

3.3 C and the 68000 – Arrays

```
void main(void)
{
  int x[10];
  register int i;
  for (i=0; i<10; i++)
      x[i]=0;
}
```

```
* Variable x is at -20(A6)
* Variable i is in the D7 Register
_main
  LINK A6,#-20
*2    {
*3      int x[10];
*4      register int i;
*5      for (i = 0; i < 10; i++)
  CLR  D7
  BRA  L1
L2
*6        x[i] = 0;
  MOVE D7,D1
  ADD  D1,D1
  CLR  -20(A6,D1.W)
*(see line 5)
  ADDQ #1,D7
L1CMPI #10,D7
  BLT.S  L2
*7    }
  UNLK A6
  RTS
  END
```

3.3 C and the 68000 – Pointers and C

- C is _____
- _____ in 68000 assembly language: (Ai)
- Example:

C Code	68000 code	comment
x=*y;	MOVE (A0),D0	x is in D0, y is in A0
a=&b;	LEA B,A0	a is in A0

- Pointer Arithmetic

```
char x='A';
int y=0;
register char *P_x=&x;
register int *P_y=&y;
P_xx++;
P_yy++;
```

```
LINK A6,#-4 /*x:-1(A6),y:-4(A6)*/
MOVE.B #65,-1(A6)
CLR -4(A6)
LEA.L -1(A6),A3
LEA.L -4(A6),A2
ADDQ #1,A3
ADDQ #2,A2
UNLK A6
RTS
```


3.3 C and the 68000 – Pointers and C (continued)

```
void main(void)
{
int x;
int *P_port; /*pointer*/

P_port = (int*) 0x4000;

/* wait for port ready */
do { }
while
   ((*P_port&0x0001)==0);

x = *(P_port + 1);
/* read from 2 bytes
   beyond port */
}
```

```
   *1 main()
   * Variable x is at -2(A6)
   * Variable P_port is at -6(A6)
_main
   LINK A6,#-6
   *2 {
   *3 int x;
   *4 int *P_port;
   *5 P_port = (int*) 0x4000;
   MOVE.L #16384,-6(A6)
   *6 do { }
   *7   while ((*P_port&0x0001)==0);
L1 MOVEA.L -6(A6),A4
   MOVE (A4),D1
   ANDI #1,D1
   BEQ.S L1
   *7 x = *(P_port + 1);
   MOVE 2(A4),-2(A6)
   *8 }
   UNLK A6
   RTS
```

3.3 C and the 68000 – C *call-by-value*

- Passing parameters to a function
- Passing by value/reference
- Is this going to work?

```
/* this function swaps the values of a and b */
void swap (int a, int b) {
   int temp;
   /* copy a to temp, b to a, and temp to b */
   temp = a;
   a = b;
   b = temp;
}
void main (void) {
   int x = 2, y = 3;
   swap (x, y); /* let's swap a and b */
}
```


3.3 C and the 68000 – 68000 *call-by-value*

```
*1 void swap (int a, int b)
* Parameter a is at 8(A6)
* Parameter b is at 10(A6)
* Variable temp is at -2(A6)
_swap
LINK A6,#-2
*2 {
*3 int temp;
*4 temp = a;
MOVE 8(A6),-2(A6)
*5 a = b;
MOVE 10(A6),8(A6)
*6 b = temp;
MOVE -2(A6),10(A6)
*7 }
UNLK A6
RTS
```

```
*8 void main (void)
* Variable x is at -2(A6)
* Variable y is at -4(A6)
_main
LINK A6,#-4
*9 {
*10 int x = 2, y = 3;
MOVE #2,-2(A6)
MOVE #3,-4(A6)
*11 swap (x, y);
MOVE #3,-(A7)
MOVE #2,-(A7)
JSR _swap
*12 }
UNLK A6
RTS
```

3.3 C and the 68000 – Stack Use: *call-by-value*

3.3 C and the 68000 – C *call-by-reference*

- To permit the function to modify the parameters, pass the address of the parameters
- This will work…

```
/* swap two parameters in the calling program */
void swap (int *a, int *b) {
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
void main (void) {
   int x = 2, y = 3;
    /* call swap and pass the addresses of the parameters */
   swap(&x, &y);
}
```

3.3 C and the 68000 – 68000 *call-by-reference*

```
*1 void swap (int *a, int *b)
* Parameter a is at 8(A6)
* Parameter b is at 12(A6)
* Variable temp is at -2(A6)
_swap
LINK A6,#-2
*2 {
*3 int temp;
*4 temp = *a;
MOVEA.L 8(A6),A4
MOVE (A4),-2(A6)
*5 *a = *b;
MOVEA.L 12(A6),A0
MOVE (A0),(A4)
*6 *b = temp;
MOVEA.L 12(A6),A4
MOVE -2(A6),(A4)
*7 }
UNLK A6
RTS
```

```
*8 main ()
* Variable x is at -2(A6)
* Variable y is at -4(A6)
_main
LINK A6,#-4
*9 {
*10 int x = 2, y = 3;
MOVE #2,-2(A6)
MOVE #3,-4(A6)
*11 swap (&x, &y);
PEA.L -4(A6)
PEA.L -2(A6)
JSR _swap
*12 }
UNLK A6
RTS
```


3.3 C and the 68000 – Stack Use: *call-by-reference*

3.3 C and the 68000 – Problem Time

Main memory

007000	AE	007001	F2	007002	32	007003	77	007004	89	007005	90
007006	1A	007007	AE	007008	EE	007009	F1	00700A	F2	00700B	A4
00700C	AE	00700D	88	00700E	AA	00700F	E4	007010	7E	007011	8D
007012	9C	007013	C4	007014	B2	007015	12	007016	39	007017	90
007018	00	007019	89	00701A	14	00701B	01	00701C	3D	00701D	77
00701E	89	00701F	9A	007020	5A	007021	AD	007022	99	007023	92
007024	79	007025	33	007026	97	007027	14	007028	79	007029	E7
00702A	00	00702B	0A	00702C	88	00702D	18	00702E	82	00702F	79
007030	23	007031	17	007032	46	007033	9E	007034	FC	007035	FF
007036	77	007037	60	007038	21	007039	42	00703A	55	00703B	EA
00703C	61	00703D	81	00703E	C	00703F	AA				
100000	DD	010001	B2	010002	00	010003	15	010004	76	010005	19
010006	92	010007	26	010008	17	010009	14	01000A	E7	01000B	E8
01000C	19	01000D	92	01000E	19	01000F	54	010010	45	010011	99
010012	15	010013	43	010014	25	010015	76	010016	89	010017	17
010018	81	010019	17	01001A	4E	01001B	72	01001C	33	01001D	23
01001E	E1	01001F	CD	010020	DC	010021	25	010022	15	010023	17
010024	29	010025	39	010026	49	010027	2D	010028	B2	010029	62
01002A	81	01002B	21	01002C	45	01002D	18	01002E	31	01002F	D9
010030	AA	010031	77	010032	78	010033	AE	010034	EA	010035	34
010036	25	010037	17	010038	15	010039	14	01003A	17	01003B	F9
01003C	8A	01003D	0F	01003E	F2	01003F	E5				

3.3 C and the 68000 – Problem Time (continued)

What is the effect of applying each of the following 68000 instructions assuming the initial condition shown below? Represent modified internal registers, memory locations and conditions.

68000 Registers

D0	01234567	D1	89ABCDEF	D2	0001002D	D3	ABCD7FFF
D4	33449127	D5	AAAAAAAA	D6	ABCD0003	D7	55555555
A0	00007020	A1	00007000	A2	00007010	A3	00007030
A4	00010020	A5	00FF789A	A6	00010000	A7	00010010

Status register 2700

a)
```
   ORG  $9000
   LEA  VEC1(PC),A5
        Assembly listing
   1    00009000              ORG     $9000
   2    00009000 4BFA0F0F     LEA     TABLE1(PC),A5
```

3.3 C and the 68000 – Problem Time (continued again)

b) `LEA 6(A0,D6.W),A2`

c) `BSET #2,(A3)`

d) `MOVE.L D2,(A6)+`

3.3 C and the 68000 – Problem Time (concluded)

e) `ADD.L #$DA7D,(A3)+`

f) `MOVE.L $10020,$600`

g) `MOVEP.L 2(A1),D4`