Outline

- ACIA Example: Pseudo-code + Assembly
- Passing parameters
  - In registers
  - Passing by value
  - Passing by reference
- Stack and Local Variables
- C and the M68000

Assembly Language and C

- We are interested in:
  - How a high-level language uses low-level language features?
  - C: System programming, device drivers, …
  - Use of addressing modes by compilers
  - Parameter passing in assembly language
  - Local storage

Programmer’s view of ACIA

- 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

ACIA registers

- Status byte (at N)
  - Error bits
  - Ready bit
- Control byte (at N)
  - Control bits
- Data byte (at N+2)
  - Data from ACIA
Assembly Language and C, ACIA example

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

BEGIN

set Cycle_Count to max value

REPEAT

Read status of Input_Dev

Decrement Cycle_Count

UNTIL Input_Dev is ready OR Cycle_Count = 0

THEN Error_St = $FF END_IF

ELSE Error_St = status from Input_Dev

END_IF

END_IF

End Character_Input

ACIA example, 68000 assembly language version

* ACIA_Initialize and Character_Input routine
* Data register D0 contains Function (zero=initialize, non-zero = get a character)
* Data register D1 is re-used for the Cycle_Count (as a timeout mechanism)
* Data register D2 contains Error_Status
* 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

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  (2,A0),D2     Read the data from the ACIA

BRA Exit_2                  and return

Data_Ok Data_Ok MOVE.B  (A0),D2     Read the data from the ACIA

BRA Exit_2                  and return

Error_St Exit_1  MOVE.B  #0,D0       Return Error_Status

Exit_2   MOVE.B  #159,(A0)     Configure the ACIA

BRA Exit_2                              and return

Outline

- ACIA Example: Pseudo-code + Assembly
- Passing parameters
  - In registers
  - Passing by value
  - Passing by reference
- Stack and Local Variables
- C and the M68000
Passing Parameters via Registers
- Two registers are used in subroutine and have to be saved on the stack:
  \[ \text{MOVE.W D3-D4,-(A7)} \]
  (otherwise, data would be lost)
- D0 is simply reused without saving, because the old data will not be needed
- PROS:
  - Position independent code
  - Re-entrancy (subroutine has to save registers before they are reused)
- CONS:
  - Reduces number of registers available to programmer
  - Number of parameters limited by the number of registers

Mechanisms for Parameter Passing
- **Passing parameters by value**
  - Actual parameter is transferred
  - If the parameter is modified by the subroutine, the "new value" does not affect the "old value"
- **Passing parameters by reference**
  - The address of the parameter is passed
  - There is only one copy of parameter
  - If parameter is modified, it is modified globally

Passing Parameters by Value
- **LEA \((-4,A7),A7\)** Save space on stack for Error_Status and Input_Char
- **MOVE.L \#ACIA,-(A7)** Push ACIA address on the stack
- **MOVE.W Func,-(A7)** Push function code on the stack

State of stack after executing this instruction
<table>
<thead>
<tr>
<th>Address with respect to the initial stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error_Status</td>
</tr>
<tr>
<td>-2</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Function
<table>
<thead>
<tr>
<th>Address with respect to the initial stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error_Status</td>
</tr>
<tr>
<td>-2</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

ACIA address
<table>
<thead>
<tr>
<th>Address with respect to the initial stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error_Status</td>
</tr>
<tr>
<td>-2</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

ACIA address
<table>
<thead>
<tr>
<th>Address with respect to the initial stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error_Status</td>
</tr>
<tr>
<td>-2</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Passing Parameters by Value

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

ACIA example:

Passing Parameters by Value

* Character_Input and ACIA_Initialize routine
* Data register D3 is temporary storage for the ACIA's status
* Data register D4 is temporary storage for the Cycle_Count
* Address register A0 contains the address of the ACIA's control/status register

Char_In MOVEM.L A0/D3-D4,-(A7) Push working registers on the stack
MOVE.L (18,A7),A0 Read address of ACIA from the stack
CLR.B (24,A7) Start with Error_Status clear

* InPut MOVEM.L @0,(A0,D3) Push working registers on the stack
InPut MOVEM.W #00000000,(A1,D0) Read the ACIA's status register
InPut1 MOVEM.B (A0,D3) Copy status to D4
AND.B #%01111100,D4 Mask status bits to error conditions
BTST #0,D3 Test data_ready bit of saved 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,(24,A7) ELSE Set error_flag 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
ACIA example:
Passing Parameters by Value

- BACK TO MAIN PROGRAM:
- BSR Char_In
- CALL subroutine
- LEA (6, A7), A7
- Clean up stack - remove parameters
- MOVE.W (A7)+, Char
- Pull the input character off the stack
- MOVE.W (A7)+, Err
- Pull the Error_Status off the stack
ACIA Example: Passing Parameters by Reference

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

ACIA Example: Passing Parameters by Reference

* Back to main program
  
BSR  Char_In  Call subroutine
LEA  (16,A7),A7  Clean up the stack - remove the 4 addr

Char  Error_Status  ACIA address  Function  SP
| 16 | 12 | -8 | -4 | 0 |

Outline
- ACIA Example: Pseudo-code + Assembly
- Passing parameters
  - In registers
  - Passing by value
  - Passing by reference
- Stack and Local Variables
- C and the M68000

The Stack and Local Variables
- Subroutines often need local workspace
- We can use a fixed block of memory space – static allocation – but:
  - The code will not be relocatable
  - The code will not be reentrant
  - The code will not be able to be called recursively
- Better solution: dynamic allocation
  - Allocate all local variables on the stack
  - STACK FRAME = a block of memory allocated by a subroutine to be used for local variables
  - FRAME POINTER = an address register used to point to the stack frame
The Stack and Local Variables

It can be done simply by modifying the stack pointer:

```
AnySub  
LEA (-2, A7)  ; Set up A7 as the frame pointer: create the stack frame.
        ; The subroutine proper.
LEA (100, A7) ; Collapse the stack frame and return from the subroutine.
```

---

The Stack and Local Variables

```
LINK   A1, @A4  ; Allocate 4 bytes (2 long words) of storage in this stack frame - use A1 as frame pointer.
        ; Body of the subroutine.
UNLK   @A1  ; Deallocate Subroutine's stack frame.
RTS    ; Return to calling point.
```

---

The Stack and Local Variables

**Implementation**

```
LINK       ; Decrease the stack pointer by 4
(SP) ← (SP) - 4

FAIL       ; Save stack pointer in A1:
(SP) ← (SP) - 60

UNLK       ; Move stack pointer up by 64 locations
(SP) ← (SP) + 60

(sp) ← (sp) + 4
```

---

The Stack and Local Variables

```
ACIA   
LEA (14, A7) ; Clean up the stack frame - remove the four parameters.
```

---

ACIA Example: Local Variables

```
PFA    ; Push address of dest. for the input
PFA    ; Push address of Error_Status message
PFA    ; Push ACIA's address on the stack
PFA    ; Push ACIA's address on the stack
MOVE.W (A7), -(A7) ; Push the contents of address register A1.
FAIL   ; Set up the stack pointer A1.
(SP) ← (SP) - 60

LEA (14, A7) ; Clean up the stack frame - remove the four parameters.
```

---

ACIA Example: Local Variables

```
(SP) ← (SP) + 4
```

---

ACIA Example: Local Variables
**ACIA Example: Local Variables**

- 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
MOVER.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

ACIA Example:
Local Variables
```

**ACIA Example: Local Variables**

```
InPut MOVE.B (A2), (-4, A6)  Read the save in T
MOVE.B (-4, A6), (-6, A6)   Copy at
ANDL.W #$01111100, (-6, A6) Mask condition
BNE Exit_1                 IF status
BSTR #0, (-4, A6)          ELSE Test data_ready bit of status
BNE Data_OK                IF data_ready THEN get data
SUBQ.W #1, (-2, A6)        ELSE decrement Cycle_Count
BNE InPut                  IF not timed out THEN repeat
MOVE.B #$FF, (A1)          ELSE Set error flag
BRA Exit_2                 and return
```

<table>
<thead>
<tr>
<th>SF</th>
<th>Saved_registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Temp 2</td>
</tr>
<tr>
<td>0</td>
<td>Temp 1</td>
</tr>
<tr>
<td>1</td>
<td>Cyckl. Count</td>
</tr>
<tr>
<td>A1</td>
<td>Old M</td>
</tr>
<tr>
<td>A2</td>
<td>M</td>
</tr>
<tr>
<td>A5</td>
<td>4</td>
</tr>
<tr>
<td>A4</td>
<td>-16</td>
</tr>
<tr>
<td>A3</td>
<td>-18</td>
</tr>
<tr>
<td>A6</td>
<td>-19</td>
</tr>
<tr>
<td>A7</td>
<td>-20</td>
</tr>
<tr>
<td>A8</td>
<td>-21</td>
</tr>
<tr>
<td>A9</td>
<td>Stack frame</td>
</tr>
</tbody>
</table>

**ACIA Example: Local Variables**

```
Local Variables

- Data_dest MOVE.L (18, A6), (A1)  Get address for data dest. (reuse A1)
- MOVE.L (2, A2), (A1)           Read data from ACIA
- BRA Exit_2                    Return Error_Status
- Exit_1 MOVE.B (-4, A6), (A1)   Restore Error_Status
- Exit_2 MOVE.L (A7), A1-A2      Restore working registers
- UNLK A6
```

**Outline**

- ACIA Example: Pseudo-code + Assembly
  - Passing parameters
    - In registers
    - Passing by value
    - Passing by reference
  - Stack and Local Variables
  - C and the M68000
C and The 68000
- Compiler and 68000 instruction set
- C data types and implementation
- Storage classes
- Functions and parameters
- Pointers

Compiling a C Program
```c
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)
XREF_main
main
LINK A6,#-4
* Variable i is at -2(A6)
* Variable j is at -4(A6)
MOVE $1,-2(A6)
ADD $2,-2
MOVE $2,-4(A6)
ADD $1,#1
MOVEQ.L $1,D1
UNLK A6
RTS

C Data Types
- The 68000 family supports three basic data types:
  - Byte, Word, Longword
- Each can be interpreted as signed or unsigned
- C built-in types:
  - Integer, character, floating point, double-precision
  - Void – refers to the null data type
- Implementation dependant!

<table>
<thead>
<tr>
<th>Data type</th>
<th>C name</th>
<th>Width (b)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>int</td>
<td>16</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>short integer</td>
<td>short int</td>
<td>8</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>long integer</td>
<td>long int</td>
<td>32</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned integer</td>
<td>unsigned int</td>
<td>16</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>character</td>
<td>char</td>
<td>8</td>
<td>0 to 255</td>
</tr>
<tr>
<td>single-precision floating point</td>
<td>float</td>
<td>32</td>
<td>10^-38 to 10^38</td>
</tr>
<tr>
<td>double-precision floating point</td>
<td>double</td>
<td>64</td>
<td>10^-308 to 10^308</td>
</tr>
</tbody>
</table>

C Data Types, cont’d
- 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

```c
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*/
    }
```
Storage Class

- **Storage class specifiers**
  - **auto**
    - Variable is no longer required once a block has been left; Default
  - **register**
    - Ask compiler to allocate the variable to a register
    - Also is automatic
    - Cannot be accessed by means of pointers
  - **static**
    - Allows local variable to retain its value when a block is reentered
    - Initialized only once, by the compiler!
  - **extern**
    - Indicated that the variable is defined outside the block
    - The same global variable can be defined in more than one module

Storage Class, cont’d

- **Access Modifiers**
  - **volatile**
    - To define variables that can be changed externally
    - Compiler will not put them in 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 automatically or explicitly (casting)

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

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

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

Returning a Value from a Function

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

```
Parameters accessed from the main’s stack frame:
```
```
a and b are pushed on stack prior to the function call
```

Returning a Value from a Function, cont’d

```
int adder(int x, int y)
{
    Parameter y is at 10(A6)
    _adder
    LINK A6,#0
    *2 { return x + y; /* return x + y */
    *3 return x + y;
    *6 } LINK A6,#-6
    MOVE 8(A6),D1
    ADD 10(A6),D1
    MOVE D1,D0
    *4 } UNLK A6
    RTS

void main (void)
{
    Parameter x is at -2(A6)
    Parameter y is at -4(A6)
    Parameter c is at -6(A6)
    main
    LINK A6,#-6
    *5 { int a, b, c;
    *7 a = 1, b = 2;
    *8 } MOVE #1,-2(A6)
    MOVE #2,-4(A6)
    MOVE D0,-6(A6)
    *9 c = adder(a, b); /* c is assigned the integer returned by adder */
    UNLK adder
    MOVE D0,-6(A6)
    RTS

Parameters accessed from the main’s stack frame:
```
```
a and b are pushed on stack prior to the function call
```

Not taken from the stack frame:
```
```
```
Functions and Parameters

- 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 */
}

No, because this program is using a call-by-value mechanism.

USE OF STACK – call-by-value

1. State of the stack after the allocation of the stack frame of swap
2. The stack after the push of the return address, the function pointer, and the parameters
3. The stack after the creation of a stack frame in main

Figure 3.11
Functions and Parameters
Call-by-reference

To permit the function to modify the parameters, pass the address of the parameters.

This will work...

```c
/* 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);
}
```

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

Functions and Parameters
Call-by-reference, cont’d

```c
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);
}
```

![Diagram](image1)

![Diagram](image2)

Returning a Value from a Function
USE OF STACK

![Diagram](image3)
Returning a Value from a Function
USE OF STACK, cont'd

Figure 3.9

Pointers and C
- C is pointer-oriented
- Pointers in 68000 assembly language: (A1)
  Example:
  ```c
  #include <stdio.h>

  void main(void)
  {
    int x = 10, y = 20;
    int *p = &x;  
    *p = y;      
    printf("%d\n", *p);  
  }
  
  ;
  ```

  ```assembly
  ; Example code
  ```

  - Pointer Arithmetic
    ```c
    char x = 'A';
    int y = 0;
    register char *p_x = &x;
    register int *p_y = &y;
    `p_x++;
    *p_y++;
    ```

    ```assembly
    ; Example code
    ```

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

    ```assembly
    ; Example code
    ```
Speed and Performance of Microprocessors

Why is difficult to compare the speed of two microprocessors?
- Clock speed
- Meaningless MIPS
- Memory access times
- Are registers used optimally?
- Special addressing modes (not generally useful)
- Misleading benchmarks
- Use of cache
- Pipeline
- Carefully interpret benchmarks!
- Clock Cycles/Bus Cycles

Example: Interpret the high-level language construct
IF COUNT[class[i]] <> 0 THEN ...

Clock Cycles/Bus Cycles

Execution Time: An Example

For the given assembly language program:
LEA TABLE,A0
CLR.W D1
LOOP MOVE.B D0,(A0)+
ADDQ.W #1,D1
CMP1.W #9.D1
BNE LOOP

a) Find the total execution time of the given program on a 12.5 MHz 68000 microprocessor.
b) What is the average CPI (number of clocks per instructions)?
c) What is the MIPS rate?
a) Find the total execution time of the given program on a 12.5 MHz 68000 microprocessor.

- **Cycle time**: \( T_{cycle} = \frac{1}{12.5 \, MHz} = 80 \, ns \)
- **Clock cycles**: \( C = 1 \times (8+4) + 8 \times (8+4+8+10) + 1 \times (8+4+8+8) = 280 \) cycles
- **Number of instructions**: \( N = 2 + 9 \times 4 = 38 \) instructions
- **Execution time**: \( T_{exe} = C \times T_{cycle} = 22.4 \, ms \)

b) What is the average CPI (number of clocks per instructions)?

- **Number of clocks/instruction** CPI = \( C / N = 280 / 38 = 7.37 \)

b) What is the MIPS rate?

- **MIPS rate**: \( 10^6 \times \frac{f}{CPI} = 12.5 / 7.37 = 1.7 \, MIPS \)