CPE/EE 421
Microcomputers
Instructor: Dr Aleksandar Milenkovic
Lecture Note
S06

*Material used is in part developed by
Dr. D. Raskovic and Dr. E. Jovanov

Course Administration

- Instructor: Aleksandar Milenkovic
  milenka@ece.uah.edu
  www.ece.uah.edu/~milenka
  EB 217-L
  Mon. 5:30 PM – 6:30 PM,
  Wen. 12:30 – 13:30 PM
- URL: http://www.ece.uah.edu/~milenka/cpe421-05F
- TA: Joel Wilder
- Labs: Lab#1 is on. First session 9/12. Due 9/14.
- Hws: Hw #1 is on. Due 9/21/05, 2:20.
- Text: Microprocessor Systems Design:
  68000 Hardware, Software, and Interfacing
- Review: Chapter 1, Chapter 2;
- Today: Passing Parameters; C and the M68K.
Course Administration

NEW LAB SUBMISSION POLICY (by Joel Wilder)

- Turn in a printout of your code to the instructor personally, or slide it under his office door. If the instructor has to print out your file, you will have 10 points subtracted from your grade.
- Demonstrate the code for the instructor in the lab hour. If this is not possible before the deadline, email the code to the instructor by the due date.
- Assignments are due by 8pm on the due date (assuming this is on a Wednesday. Make it by 3pm if this is a Friday)

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 (-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
```

The Stack and Local Variables

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

```
Sub1  LINK  A1,0-64  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`
The Stack and Local Variables

Nested subroutines: A calls B, then B calls A

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

(a) initial state of the stack
(b) state of stack after pushing parameters
(c) state of stack after calling the subroutine
* Character_Input and ACIA_Init routine
* SF location A6 - 6 holds the ACIA's status
* SF location A6 - 4 holds the ACIA's masked status (error bits only)
* A1 contains the address of the Error_Status
* A2 contains the address of the ACIA's control register

```
Char_In LINK A6,#-6
   MOVE.L A1-A2,-(A7)        ; Push working registers on the stack
   MOVEA.L (14,A6),A1       ; Read address of Error_Status from stack
   MOVEA.L (10,A6),A2      ; Read address of ACIA
   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
   MOVE.B #3,(A2)          ; Reset ACIA
   MOVE.B #$19,(A2)        ; Configure ACIA
   BRA Exit_2
```

```
InPut  MOVE.B (A2),(-4,A6)  ; Read the ACIA's status register - save in Temp1
   MOVE.B (-4,A6),(-6,A6)  ; Copy status
   ANDI.B #$01111100,(-6,A6)  ; Mask status bits to error conditions
   BNE Exit_1  ; IF status indicates error, set flag and exit
   BTST #0,(-4,A6)          ; ELSE Test data_ready bit of status
   BNE Data_OK            ; IF data_ready THEN get data
   SUBQ.W #1,(-2,A6)     ; ELSE decrement Cycle_Count
   BNE InPut             ; IF not timed out THEN repeat
   MOVE.B #$FF,(A1)       ; ELSE Set error flag
   BRA Exit_2             ; and return
```

(c) state of stack after calling the subroutine
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)
XDEF __main
main
    LINK A6,-4
    *2 {
    *3 int i;
    *4 int j;
    *5 i = 1;
    MOVE #1,-2(A6)
    *6 j = 2;
    MOVE #2,-4(A6)
    *7 i = i + j;
    MOVEQ.L #1,D1
    ADDQ #2,D1
    MOVE D1,-2(A6)
    *8 }
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>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 float point</td>
<td>float</td>
<td>32</td>
<td>10^-38 to 10^+38</td>
</tr>
<tr>
<td>double-precision float point</td>
<td>double</td>
<td>64</td>
<td>10^-300 to 10^+300</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
#include <stdio.h>

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
    - Indicates that the variable is defined outside the block
    - The same global variable can be defined in more than one modul
Storage Class, cont’d

- **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 automatically or explicitly (casting)
  - `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 automatically or explicitly (casting)

<table>
<thead>
<tr>
<th>X</th>
<th>DS.L 1 Reserve a longword for X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>DS.W 1 Reserve a word for Y</td>
</tr>
</tbody>
</table>

**USUALLY WRONG**

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

**CORRECT**

- `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`)

```c
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 */
}
```
Returning a Value from a Function, cont’d

```
*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
```

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

---

Returning a Value from a Function

USE OF STACK

![Diagram of stack usage](image)

- **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)**

*Figure 3.9*
Returning a Value from a Function

USE OF STACK, cont’d

Figure 3.9

Returning a Value from a Function

Array

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

- C is pointer-oriented
- Pointers in 68000 assembly language: (Ai)

Example:

<table>
<thead>
<tr>
<th>C Code</th>
<th>68000 Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=*y;</td>
<td>MOVE (A0),D0</td>
<td>x is in D0, y is in A0</td>
</tr>
<tr>
<td>a=&amp;b;</td>
<td>LEA B,A0</td>
<td>a is in A0</td>
</tr>
</tbody>
</table>

- Pointer Arithmetic

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

```assembly
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
```

Pointers and C, cont’d

```c
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 */
}
```

```assembly
*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
```
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

Functions and Parameters, cont’d

*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;
  *5 MOVE 8(A6),-2(A6)
  *6 a = b;
  *7 MOVE 10(A6),8(A6)
  *8 b = temp;
  *7 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

CPE/EE 421/521 Microcomputers

Alex Milenkovich
Functions and Parameters
USE OF STACK – call-by-value

a. State of the stack after LINK A6, N-4 in _main_. Addresses are specified with respect to A6.

b. The stack after MOVE #3, (A7) MOVE #2, (A7)

The values of parameters x and y are pushed on the stack.

Figure 3.11

Functions and Parameters
USE OF STACK – call-by-value, cont’d

c. The stack after the subroutine call. The return address is on top of the stack.

d. The stack after the creation of a stack frame in _swap_. The new stack frame is 2 bytes deep and holds the variable temp.

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...

```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);
}
```

Functions and Parameters

Call-by-reference, cont’d

```c
*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 (4x, 4y);
        PEA.L -4(A6)
        PEA.L -2(A6)
        JSR _swap
        *12 }
    UNLK A6
    RTS
```
Functions and Parameters
USE OF STACK, Call-by-reference

![Diagram of stack operations]

Speed and Performance of Microprocessors

- Why is difficult to compare the speed of two microprocessors?
  1. Clock speed
  2. Meaningless MIPS
  3. Memory access times
  4. Are registers used optimally?
  5. Special addressing modes (not generally useful)
  6. Misleading benchmarks
  7. Use of cache
  8. Pipeline

- Carefully interpret benchmarks!
- Clock Cycles/Bus Cycles
Speed and Performance of Microprocessors, cont’d

- Example: Interpret the high-level language construct

  \[ \text{IF COUNT\{CLASS[I]\} <> 0 THEN ...} \]

### 68000 Version 68020 Version

<table>
<thead>
<tr>
<th>Clock Cycles</th>
<th>Bus Cycles</th>
<th>Clock Cycles</th>
<th>Bus Cycles</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>MOVE.W D1,D3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>LSL.W #1,D3</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>LEA 0(A5,D3.W),A2</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>MOVE.W CLASS(A2),D3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>LSL.W #1,D3</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>LEA 0(A5,D3.W),A2</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>TST.W COUNT(A2)</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>BEQ ELSE</td>
</tr>
<tr>
<td>79</td>
<td>15</td>
<td>42</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

For the previous example, 68000:
- Execution time = 6.32 µs
- \( \Rightarrow \) 8 instructions / 6.32 µs = \(1.27\) MIPS

For the previous example, 68020 using the same code:
- Execution time = 2.52 µs
- \( \Rightarrow \) 8 instructions / 2.52 µs = \(3.17\) MIPS

For the previous example, 68020 using special features:
- Execution time = 1.44 µs
- \( \Rightarrow \) 3 instructions / 1.44 µs = \(2.08\) MIPS

**MIPS = Million Instructions Per Second**
Example

For the given assembly language program:

```
LEA    TABLE, A0
CLR.W  D1
LOOP   MOVE.B D0, (A0)+
       ADDQ.W #1, D1
       CMP.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?

---

**#of cycles**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Single execution</th>
<th>Loop (9 iterations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>CLR</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>MOVE.B</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>ADDQ.W</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMP.W</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>BNE LOOP</td>
<td>10 (taken)/8</td>
<td></td>
</tr>
</tbody>
</table>

---

a) Find the total execution time of the given program on a **12.5 MHz** 68000 microprocessor.

- **Cycle time** \( T_{cycle} = 1 / 12.5 \text{ MHz} = 80 \text{ ns} \)
- **Clock cycles** \( C = 1 \times (8+4) + 8 \times (8+4+8+10) + 1 \times (8+4+8+8) = 280 \text{ cycles} \)
- **Number of instructions** \( N = 2 + 9 \times 4 = 38 \text{ instructions} \)
- **Execution time** \( T_{exe} = C \times T_{cycle} = 22.4 \text{ ms} \)
b) What is the average CPI (number of clocks per instructions)?

Number of clocks/instruction \[ \text{CPI} = \frac{C}{N} = \frac{280}{38} = 7.37 \]

Total number of clock cycles to execute the program

Total number of instructions in the program (loops)

b) What is the MIPS rate?

\[ \text{MIPS rate} = 10^{-6} \times \frac{f}{\text{CPI}} = 12.5 / 7.37 = 1.7 \text{ MIPS} \]