CPE/EE 421 Microcomputers
Instructor: Dr. Aleksandar Milenkovic
Lecture Note S04

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

The 68000 Family Instruction Set

- Assumption: Students are familiar with the fundamentals of microprocessor architecture
- Groups of instructions:
  - Data movement
  - Arithmetic operations
  - Logical operations
  - Shift operations
  - Bit Manipulation
  - Program Control

Important NOTE: The contents of the CC byte of the SR are updated after the execution of an instruction. Refer to Table 2.2

Data Movement Operations

- MOVE/MOVEA
- MOVE to CCR
- MOVE to/from SR
- MOVE USP
- MOVEP
- LEA

Data Movement Operations, cont’d

Moving data from a 32-bit register to memory using the MOVEP instruction

Bytes from the register are stored in every other memory byte

NOTE: The instruction takes 24 clock cycles to execute.

Data Movement Operations, cont’d

LEA

Calculates an effective address and loads it into an address register – LEA <ea>,An
Can be used only with 32-bit operands

Assembly language

RTL

LEA $0010FFFF,A5 [A5] ← $0010FFFF
Load the address $0010 FFFF into register A5.

LEA $12(A8,D4),A5 [A5] ← $12 + [$A8] + [$D4]
Load contents of A8 plus contents of D4 plus $12 into A5.

NOTE: If the instruction MOVE.L $12(A8,D4),A5 had been used, the contents of that address would have been deposited in A5.

Why use it? FASTER!
ADD.W $1C(A3,D2),D0 vs. LEA $1C(A3,D2),A5 ADD.W (A5),D0

Course Administration

- Instructor: Aleksandar Milenkovic
  - milenka@ece.uah.edu
  - www.ece.uah.edu/~milenka
- EB 217-L
  - Mon. 5:30 PM – 6:30 PM
  - Wed. 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.
- Hw #1 is posted (due 9/21/05, 2:20);
- Text: Microprocessor Systems Design: 68000 Hardware, Software, and Interfacing
- Review: Chapter 1, Chapter 2;
- Today: Instruction Set, Subroutines, Assembly and C;

- 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.
- Hw #1 is posted (due 9/21/05, 2:20);
- Text: Microprocessor Systems Design: 68000 Hardware, Software, and Interfacing
- Review: Chapter 1, Chapter 2;
- Today: Instruction Set, Subroutines, Assembly and C;
Example:

**Main Memory**

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>007000</td>
<td>E4</td>
</tr>
<tr>
<td>007001</td>
<td>79</td>
</tr>
<tr>
<td>007002</td>
<td>1F</td>
</tr>
<tr>
<td>007003</td>
<td>9A</td>
</tr>
<tr>
<td>007004</td>
<td>00</td>
</tr>
<tr>
<td>007005</td>
<td>70</td>
</tr>
<tr>
<td>007006</td>
<td>3F</td>
</tr>
<tr>
<td>007007</td>
<td>AA</td>
</tr>
<tr>
<td>007008</td>
<td>01</td>
</tr>
<tr>
<td>007009</td>
<td>00</td>
</tr>
<tr>
<td>00700A</td>
<td>70</td>
</tr>
<tr>
<td>00700B</td>
<td>1F</td>
</tr>
<tr>
<td>00700C</td>
<td>9A</td>
</tr>
<tr>
<td>00700D</td>
<td>00</td>
</tr>
<tr>
<td>00700E</td>
<td>70</td>
</tr>
<tr>
<td>00700F</td>
<td>3F</td>
</tr>
</tbody>
</table>

**Status Register**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>00</td>
</tr>
<tr>
<td>A1</td>
<td>00</td>
</tr>
<tr>
<td>A2</td>
<td>00</td>
</tr>
<tr>
<td>A3</td>
<td>00</td>
</tr>
<tr>
<td>A4</td>
<td>00</td>
</tr>
<tr>
<td>A5</td>
<td>00</td>
</tr>
<tr>
<td>A6</td>
<td>FF</td>
</tr>
<tr>
<td>A7</td>
<td>00</td>
</tr>
</tbody>
</table>

Integer Arithmetic Operations, cont’d

- **DIVU/DIVS** – unsigned/2’s-complement numbers
  - Divide unsigned numbers
  - Divides the specified operand from the dividend
- **ADD/ADDA**
  - Add a literal value to the destination
  - Adds two packed BCD digits together with the carry flag
- **SBCD**
  - Subtracts the specified operand from zero together with the carry flag
  - Subtracts two packed BCD digits
- **NEG**
  - Negates an operand
  - Forms the 2’s complement of the operand
  - Negate with Extend, used for multi-precision arith.
- **EXT**
  - Extends the specified operand
  - Copies bit 7 to bits 8-15
  - Copies bit 15 to bits 16-31

BCD Arithmetic Operations

- **SBCD**
  - Subtracts the specified operand from zero together with the carry flag
  - Subtracts two packed BCD digits
- **ADD**
  - Adds two packed BCD digits together with the carry flag
  - Adds two packed BCD digits
- **NEG**
  - Negates an operand
  - Forms the 2’s complement of the operand
  - Negate with Extend, used for multi-precision arith.
- **EXT**
  - Extends the specified operand
  - Copies bit 7 to bits 8-15
  - Copies bit 15 to bits 16-31
Logical Operations
- Standard AND, OR, EOR, and NOT
- Immediate operand versions: ANDI, ORI, EORI
- AND a bit with 0 = mask
- OR a bit with 1 = set
- EOR a bit with 1 = toggle
- Logical operations affect the CCR in the same way as MOVE instructions

Shift Operations
- Logical Shift
  - LSL – Logical Shift Left
  - LSR – Logical Shift Right

Shift Operations, cont’d
- Arithmetic Shift
  - ASL – Arithmetic Shift Left
  - ASR – Arithmetic Shift Right

Shift Operations, cont’d
- Rotate
  - ROL – Rotate Left
  - ROR – Rotate Right

Shift Operations, cont’d
- Rotate Through Extend
  - ROXL – Rotate Left Through Extend
  - ROXR – Rotate Right Through Extend

Effect of the Shift Instructions

<table>
<thead>
<tr>
<th>Initial Value</th>
<th>After First Shift</th>
<th>After Second Shift</th>
<th>CCR X'820C</th>
<th>CCR X'800C</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASL</td>
<td>011110110</td>
<td>011110000</td>
<td>11001</td>
<td>11000</td>
</tr>
<tr>
<td>ASR</td>
<td>011111100</td>
<td>000111111</td>
<td>00000</td>
<td>10001</td>
</tr>
<tr>
<td>LSL</td>
<td>111101101</td>
<td>111110100</td>
<td>01100</td>
<td>11110001</td>
</tr>
<tr>
<td>LSR</td>
<td>111111110</td>
<td>011111111</td>
<td>00000</td>
<td>10001</td>
</tr>
<tr>
<td>ROL</td>
<td>111010111</td>
<td>110101010</td>
<td>11001</td>
<td>11001</td>
</tr>
<tr>
<td>ROR</td>
<td>011111110</td>
<td>001111111</td>
<td>00000</td>
<td>11001</td>
</tr>
<tr>
<td>ROXL</td>
<td>111010111</td>
<td>111010111</td>
<td>11001</td>
<td>11001</td>
</tr>
<tr>
<td>ROXR</td>
<td>011111110</td>
<td>001111111</td>
<td>00000</td>
<td>11001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial Value</th>
<th>After First Shift</th>
<th>After Second Shift</th>
<th>CCR X'820C</th>
<th>CCR X'800C</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROXL</td>
<td>111010111</td>
<td>111010101</td>
<td>11001</td>
<td>11001</td>
</tr>
<tr>
<td>ROXR</td>
<td>011111110</td>
<td>001111111</td>
<td>00000</td>
<td>11001</td>
</tr>
</tbody>
</table>
Forms of Shift Operations

- **Mode 1**
  ASL Dx,Dy  
  Shift Dy by Dx bits

- **Mode 2**
  ASL #<data>,Dy  
  Shift Dy by #data bits

- **Mode 3**
  ASL <ea>  
  Shift the contents at the effective address by one place

All three modes apply to all eight shift instructions

Bit Manipulation Operations

- Act on a single bit of an operand:
  1. The complement of the selected bit is moved to the Z bit (Z set if specified bit is zero)
  2. The bit is either unchanged, set, cleared, or toggled
- NVCX bits are not affected
- May be applied to a bit within byte or longword
- BTST – Bit Test only
- BSET – Bit Test and Set (specified bit set)
- BCLR – Bit Test and Clear (specified bit cleared)
- BCHG – Bit Test and Change (specified bit toggled)

Bit Manipulation Operations, cont’d

- All 4 have the same assembly language forms:
  BTST Dn, <ea> or BTST #<data>,<ea>

Program Control Operations

- Examine bits in CCR and chose between two courses of action
- CCR bits are either:
  - Updated after certain instruction have been executed, or
  - Explicitly updated (bit test, compare, or test instructions)
- **Compare instructions:** CMP, CMPA, CMPI, CMPM
  - Subtract the contents of one register (or mem. location) from another register (or mem. location)
  - Update NZVC bits of the CCR
  - X bit of the CCR is unaffected
  - The result of subtraction is ignored

Program Control Operations, cont’d

- **CMP:** CMP <ea1>,<ea2>
  [<ea2>]–[<ea1]>
- **CMPI:** CMP #<data>,<ea>
  comparison with a literal
- **CMPA:** CMP <ea>,An
  used for addresses, operates only on word and longword operands
- **CMPM:** CMP (AI)+,(Aj)+
  compares memory with memory, one of few that works only with operands located in memory
- **TST:** TST <ea>
  zero is subtracted from specified operand; N and Z are set accordingly, V and C are cleared, X is unchanged
  Except CMPA, all take byte, word, or longword operands

Program Control Operations, cont’d

- **Branch Instructions**
  - Branch Conditionally
  - Branch Unconditionally
  - Test Condition, Decrement, and Branch
- **BRANCH CONDITIONALLY**
  Bcc <label>
  - cc stands for one of 14 logical conditions (Table 2.4)
  - Automatically calculated displacement can be d8 or d16
  - Displacement is 2’s complement signed number
  - 8-bit displacement can be forced by adding .S extension
  - ZNVC bits are used to decide
Program Control Operations, cont’d

- **BRANCH UNCONDITIONALLY**
  - BRA <label> or JMP (An)
  - JMP d16(An)
  - JMP d8(An,Xi)
  - JMP Absolute_address
  - JMP d16(PC)
  - JMP d8(PC,Xi)

- **TEST CONDITION, DECREMENT, and BRANCH**
  - DBcc Dn,<label> (16 bit displacement only)
  - One of 14 values from Table 2.4, plus T, plus F
  - If test is TRUE, branch is NOT taken!
  - If cc is NOT TRUE, Dn is decremented by 1;
    - If Dn is now equal to –1 next instruction is executed
    - if not, branch to <label> is taken

Subroutines

- **BRANCH TO SUBROUTINE**
  - BSR <label> = [A7] ← [A7] – 4
  - M([A7]) ← [PC]
  - [PC] ← [PC] + d8

- **RETURN FROM SUBROUTINE**
  - RTS = [PC] ← [M([A7])]
  - [A7] ← [A7] + 4

Subroutines, cont’d

- **BRANCH TO SUBROUTINE**
  - BSR d8 d8=? (or d16, to specify d8 use BSR.S)
  - d8 = \[00001208 – (00001000 + 2) = 00000206\]

Subroutines, cont’d

- **RETURN FROM SUBROUTINE**
  - RTS

Nested Subroutines

- **Returning directly to a higher-level subroutine**
  - Sub2
    - BEQ Exit
    - RTS

- **RTR (Return and restore condition codes)**
  - Save the condition code register on the stack:
    - MOVE CCR, -(A7)
  - Use RTR instead of RTS
Miscellaneous Instructions

- **Scc**: Set byte conditionally
  
  (cc same as in DBcc)

  If the condition is TRUE, all the bits of the byte specified by <ea> are SET, if the condition is FALSE, bits are CLEARED

- **NOP**: No Operation

- **RTS**: Return from Subroutine

- **STOP**: Stop

  Stop and load n into Status Register; n is 16-bit number; Privileged instruction

- **CHK, RESET, RTE, TAS, TRAPV** - later

Example: Linked List

- Adding an element to the end of a linked list
  
  + HEAD points to the first element, NEW contains the address of the new item to be inserted
  + Longwords

  - **LEA HEAD, A0**: A0 initially points to the start of the linked list
  - **LOOP TST.L (A0)**: IF the address field = 0
  - **BEQ EXIT**: THEN exit
  - **MOV.L (A0), (A0)**: ELSE read the address of the next element
  - **BRA LOOP**: Continue
  - **EXIT LEA NEW, A1**: Pick up address of new element
  - **MOVE.L A1, (A0)**: Add new entry to end of list
  - **CLR.L (A1)**: Insert the new terminator

Example: Linked List, cont’d

- Initial linked list:

  ```
  LEA HEAD, A0
  LOOP TST.L (A0)
  IF the address field = 0
  THEN exit
  ELSE read the address of the next element
  Continue
  EXIT LEA NEW, A1
  MOVE.L A1, (A0)
  Add new entry to end of list
  CLR.L (A1)
  Insert the new terminator
  ```

Example: Linked List, Memory Map

<table>
<thead>
<tr>
<th>Memory map of linked list before inserting an element</th>
<th>Memory map of linked list after inserting an element</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001300 00001100 00002100 00003200 00004300</td>
<td>00003100 00004200 00005300</td>
</tr>
<tr>
<td>00001204 00001108 00002108 00003208 00004308</td>
<td>00002104 00003204 00004304</td>
</tr>
<tr>
<td>0000120c 00001110 00002110 00003210 00004310</td>
<td>00002110 00003210 00004310</td>
</tr>
<tr>
<td>0000120d 00001111 00002111 00003211 00004311</td>
<td>00002111 00003211 00004311</td>
</tr>
<tr>
<td>00001214 00001120 00002120 00003220 00004320</td>
<td>00002120 00003220</td>
</tr>
<tr>
<td>00001218 00001130 00002130 00003230 00004330</td>
<td>00003230 00004330</td>
</tr>
<tr>
<td>0000121c 00001220 00002220 00003320 00004420</td>
<td>00002220 00003320 00004420</td>
</tr>
<tr>
<td>00001220 00001220 00002220 00003320 00004420</td>
<td>00002220 00003320 00004420</td>
</tr>
</tbody>
</table>

Note: The shaded memory locations represent data values

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

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 via Registers

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

Passing Parameters by Value

0

LEA  [-4, A7], A0     Save space on stack for Error_Status and Input_Char
Passing Parameters by Value

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

Error_Status  Input_Char  ACIA address
-2          -4          -8

Function   ACIA address
-10

Input_Char  Error_Status
-2

SP

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 MOVE.L A0,D3,-(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  MOVE.W #$FFFF,D0  Set up Cycle_Count for time-out
                 (reuse D0)
InPut  MOVE.B (A0),D3  Read the ACIA's status register
InPut  MOVE.B D3,D4  Copy status to D4
InPut  AND.B #%01111100,D4  Mask status bits to error conditions
InPut  BNE Exit_1  IF status indicates error, deal with it
InPut  BTST #0,D3  Test data_ready bit of status
InPut  BNE Data_OK  IF data_ready THEN get data
InPut  SUBQ.W #1,D0  ELSE decrement Cycle_count
InPut  BNE InPut  IF not timed out THEN repeat
InPut  MOVE.B #$FF,(24,A7)  ELSE Set error flag
                 and return

Data_OK  MOVE.W (2,A0),(22,A7)  Read the data from the ACIA 
                 and put on the stack
                 and return

Exit_1  MOVE.L D4,(24,A7)  Return Error_Status
Exit_2  MOVE.L A0,D0-D4  Restore working registers
                 RTS  Return

Passing Parameters by Value

BACK TO MAIN PROGRAM:

Char_In  Call subroutine  
LDA (6,A7),A7  Clean up stack - remove parameters
Function/ACIA
Char_In  Pull the input character off the stack
Error_Status  Pull the Error_Status off the stack

Input_Char  Error_Status
-4          -2

SP