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.

- Hws: 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;
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

- Copy information from source to destination
- Comprises 70% of the average program

- MOVE/MOVEA

- MOVE to CCR
  MOVE <ea>,CCR – word instruction

- MOVE to/from SR
  MOVE <ea>,SR – in supervisor mode only;
  MOVE #$2700,SR – sets the 68K in supervisor mode

- MOVE USP
  MOVE.L USP,A3 – transfer the USP to A3

- MOVEQ
  MOVE. L USP,A3 – Move Quick(8b #value to 32b reg)

- MOVEM
  MOVEM.L D0-D5/A0-A5, -(A7)
  MOVEM.L (A7)+,D0-D5/A0-A5

- MOVEP
  MOVE Peripheral
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

<table>
<thead>
<tr>
<th>Assembly language</th>
<th>RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA $0010FFFF,A5</td>
<td>[A5] ← $0010FFFF</td>
</tr>
<tr>
<td>Load the address $0010 FFFF into register A5.</td>
<td></td>
</tr>
<tr>
<td>LEA $12(A0,D4.L),A5</td>
<td>[A5] ← $12 + [A0] + [D4]</td>
</tr>
<tr>
<td>Load contents of A0 plus contents of D4 plus $12 into A5.</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: If the instruction MOVEA.L $12(A0,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

CPE/EE 421/521 Microcomputers
Example:

68000 Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>00007020</td>
</tr>
<tr>
<td>A1</td>
<td>00007000</td>
</tr>
<tr>
<td>A2</td>
<td>00007010</td>
</tr>
<tr>
<td>A3</td>
<td>00007030</td>
</tr>
<tr>
<td>A4</td>
<td>00007028</td>
</tr>
<tr>
<td>A5</td>
<td>000070FA</td>
</tr>
<tr>
<td>A6</td>
<td>00010000</td>
</tr>
<tr>
<td>A7</td>
<td>00010010</td>
</tr>
</tbody>
</table>

Main memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>4BFA</td>
</tr>
<tr>
<td>0000</td>
<td>00FF</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

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

a) ORG $9000
LEA  TABLE1(PC),A5

Assembly listing

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>4BFA</td>
</tr>
</tbody>
</table>

EA = $00000000 + 2 + $00FF = $00000A00 → A5=$0000A000, CC: Not affected (NA)

current PC value

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

EA = 6 + $00007020 + $0003 = $00007029 → A2=$00007029 CC: NA

offset A0 D6.W
Data Movement Operations, cont’d

PEA
- Push Effective Address
- Calculates an effective address and pushes it onto the stack pointed at by A7 – PEA <ea>
- Can be used only with 32-bit operands

EXG
- Exchanges the entire 32-bit contents of two registers
- EXG Xi,Xj

SWAP
- Exchanges the upper- and lower-order words of a DATA register
- SWAP Di

Integer Arithmetic Operations
- Floating-point operations not directly supported
- Except for division, multiplication, and if destination is Ai, all act on 8-, 16-, and 32-bit values
- ADD/ADDA (no mem-to-mem additions, if destination is Ai, use ADDA)
- ADDQ (adds a small 3-bit literal quickly)
- ADDI (adds a literal value to the destination)
- ADDX (adds also the contents of X bit to the sum) used for multi-precision addition
- CLR (clear specified data register or memory location) equivalent to MOVE #0, <ea>
  for address registers use SUB.L An,An
**Integer Arithmetic Operations, cont’d**

- **DIVU/DIVS** – unsigned/2’s-complement numbers
  - \[\text{DIVU} \text{<ea>}, \text{Dn} \] or \[\text{DIVS} <\text{ea}>, \text{Dn}\]
  - 32-bit longword in Dn is divided by the 16-bit word at <ea>
  - 16-bit quotient is deposited in the lower-order word of Dn
  - The remainder is stored in the upper-order word of Dn

- **MULU/MULS** – unsigned/2’s-complement numbers
  - Low-order 16-bit word in Dn is multiplied by the 16-bit word at <ea>
  - 32-bit product is deposited in Dn

- **SUB, SUBA, SUBQ, SUBI, SUBX**

- **NEG** – forms the 2’s complement of an operand
  - \[\text{NEG} <\text{ea}>\]

- **NEGX** – Negate with Extend, used for multi-prec. arith.

- **EXT** – Sign Extend
  - \[\text{EXT.W Dn}\]
  - \[\text{EXT.L Dn}\]

**BCD Arithmetic Operations**

- Only 3 instructions support BCD
  - \[\text{ABCD \text{Di}, \text{Dj} or} \quad \text{ABCD} -(\text{Ai}), -(\text{Aj})\]
    - Add BCD with extend – adds two packed BCD digits together with X bit from the CCR
  - \[\text{SBCD}}\]
    - similar
      - \[\text{destination} \leftarrow \text{destination} - \text{source} - \text{X}]]
  - \[\text{NBCD} <\text{ea}>\]
    - subtracts the specified operand from zero together with X bit and forms the 10’s complement of the operand if X = 0, or 9’s complement if X = 1

- Involve X because they are intended to be used in operations on a string of BCD digits
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

Logical Shifts diagram: [Diagram showing LSL and LSR operations]
Shift Operations, cont’d

Arithmetic Shift

- ASL – Arithmetic Shift Left
- ASR – Arithmetic Shift Right

[Diagram showing ASL and ASR]

Shift Operations, cont’d

Rotate

- ROL – Rotate Left
- ROR – Rotate Right

[Diagram showing ROL and ROR]
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>Instruction</th>
<th>Initial Value</th>
<th>After First Shift</th>
<th>CCR</th>
<th>After Second Shift</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASL</td>
<td>11101011</td>
<td>11010110</td>
<td>11001</td>
<td>10101110</td>
<td>11001</td>
</tr>
<tr>
<td>ASL</td>
<td>01111110</td>
<td>11111100</td>
<td>01010</td>
<td>11111000</td>
<td>11111</td>
</tr>
<tr>
<td>ASR</td>
<td>11101011</td>
<td>11110101</td>
<td>11001</td>
<td>11110110</td>
<td>11001</td>
</tr>
<tr>
<td>ASR</td>
<td>01111110</td>
<td>00111111</td>
<td>00000</td>
<td>00011111</td>
<td>10001</td>
</tr>
<tr>
<td>LSL</td>
<td>11101011</td>
<td>11101110</td>
<td>11001</td>
<td>11011100</td>
<td>11001</td>
</tr>
<tr>
<td>LSL</td>
<td>01111110</td>
<td>11111100</td>
<td>01000</td>
<td>11111000</td>
<td>11111</td>
</tr>
<tr>
<td>LSR</td>
<td>11101011</td>
<td>01110101</td>
<td>10001</td>
<td>00111010</td>
<td>10001</td>
</tr>
<tr>
<td>LSR</td>
<td>01111110</td>
<td>00111111</td>
<td>00000</td>
<td>00011111</td>
<td>10001</td>
</tr>
<tr>
<td>ROL</td>
<td>11101011</td>
<td>11010111</td>
<td>?1001</td>
<td>10101111</td>
<td>?1001</td>
</tr>
<tr>
<td>ROL</td>
<td>01111110</td>
<td>11111100</td>
<td>?1000</td>
<td>11111001</td>
<td>?1001</td>
</tr>
<tr>
<td>ROR</td>
<td>11101011</td>
<td>11110101</td>
<td>?1001</td>
<td>11110101</td>
<td>?1001</td>
</tr>
<tr>
<td>ROR</td>
<td>01111110</td>
<td>00111111</td>
<td>?0000</td>
<td>10011111</td>
<td>?1001</td>
</tr>
</tbody>
</table>
Forms of Shift Operations

- **Mode 1**
  \[ \text{ASL } Dx, Dy \quad \text{Shift } Dy \text{ by } Dx \text{ bits} \]

- **Mode 2**
  \[ \text{ASL } \#<\text{data}>, Dy \quad \text{Shift } Dy \text{ by } \#\text{data} \text{ bits} \]

- **Mode 3**
  \[ \text{ASL } <\text{ea}> \quad \text{Shift the contents} \]
  \[ \text{at the effective address} \]
  \[ \text{by one place} \]

**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  
  - ZNCV 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**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000FFA</td>
<td>LEA TABLE, A0</td>
<td>( LEA )</td>
</tr>
<tr>
<td>001000</td>
<td>BSR GetChar</td>
<td>( BSR )</td>
</tr>
<tr>
<td>001004</td>
<td>MOVE.B D0, (A0)</td>
<td>( MOVE )</td>
</tr>
<tr>
<td>001006</td>
<td>CMP.B #$0D,D0</td>
<td>( CMP )</td>
</tr>
<tr>
<td>00100A</td>
<td>BNE NextChr</td>
<td>( BNE )</td>
</tr>
<tr>
<td>001102</td>
<td>BSR GetChr</td>
<td>( BSR )</td>
</tr>
<tr>
<td>001106</td>
<td>CMP.B #$'Q',D0</td>
<td>( CMP )</td>
</tr>
<tr>
<td>00110A</td>
<td>BEQ QUIT</td>
<td>( BEQ )</td>
</tr>
<tr>
<td>001208</td>
<td>MOVE.B ACIAC,D0</td>
<td>( MOVE )</td>
</tr>
</tbody>
</table>

**BSR d8**

\[ d8 = \$00001208 - (\$00001000 + 2) = \$00000206 \]

current PC value

Nested Subroutines
Nested Subroutines, cont’d

- Returning directly to a higher-level subroutine

Sub2

- BEQ Exit
- RTS

Exit

LEA 4(A7),A7
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
  
  \[ \text{Scc } \langle \text{ea} \rangle \]  
  (\text{cc same as in DBcc})

  If the condition is TRUE, all the bits of the byte specified by \(<\text{ea}>\) are SET, if the condition is FALSE, bits are CLEARED

- **NOP**: No Operation

- **RTS**: Return from Subroutine

- **STOP**:  
  \[ \text{STOP } \#n \]

  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
  MOVEA.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
A0 initially points to the start of the linked list
LOOP TST.L (A0)
IF the address field = 0
BEQ EXIT
THEN exit
MOVEA.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

➢ Linked list after inserting an element at the end:

```
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
MOVEA.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
```
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
    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 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
ACIA example, 68000 assembly language version

* 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 #$01111100,D4 Mask status bits to error conditions
BNE Exit_1 IF status indicates error, set error
flags & return
BTST #0,D3 Test data_ready bit of status
BNE Data_Ok IF data_ready THEN get data
SUBQ.W #1,D0 ELSE decrement Cycle_Count
BNE InPut1 IF not timed out THEN repeat
MOVE.B #$FF,D1 ELSE Set error flag
BRA Exit_2 and return

* Data_Ok 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 MOVE.W (A7)+,D3-D4 Restore working registers
RTS Return

Two registers are used in 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
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

![Diagram of passing parameters by value]

**LEA (-4,A7),A7**

Save space on stack for `Error_Status` and `Input_Char`

- **State of stack after executing this instruction**
- **Address with respect to the initial stack pointer**

CPE/EE 421/521 Microcomputers
Passing Parameters by Value

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

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

CPE/EE 421/521 Microcomputers 43
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.M 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

Passing Parameters by Value

CMPI.B #0,(16,A7) IF Function not zero THEN get input
BNE InPut ELSE initialize ACIA

InPut MOVE.B #3,(A0) 
MOVE.B #$19,(A0) 
BRA Exit_2 

InPut1 MOVE.B (A0),D3 
MOVE.B D3,D4 
AND.B #$01111100,D4 
BTST #0,D3 Test data_ready bit of saved status
BNE Data_OK IF data_ready THEN get data
BNE InPut1 ELSE decrement Cycle_count
BNE InPut1 IF not timed out THEN repeat
SUBQ.W #1,D0 
BNE InPut1 ELSE Set error_flag
BRA Exit_2

Saved registers

Address with respect to the initial stack pointer

Address with respect to the final value of stack pointer
Passing Parameters by Value

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

BACK TO MAIN PROGRAM :
* 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