CPE/EE 421 Microcomputers: Motorola 68000: Architecture & Assembly Programming

Instructor: Dr Aleksandar Milenkovic
Lecture Notes

Outline

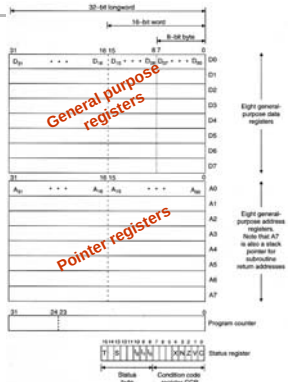
- Programmer's Model
- Assembly Language Directives
- Addressing Modes
- Instruction Set

Motorola 68000

- CISC processor
- sixteen 32-bit registers
  - eight general purpose *data* registers
  - eight general purpose *address* registers
- User/supervisor space
- 64-pin package
- Clock 8MHz, 12.5 MHz

Programming Model of the 68000*

- *Registers, Addressing Modes, Instruction Set
- NOTE: The 68000 architecture forms a subset of the 68020's architecture (i.e., 68020 is backward compatible)
- NOTE:
  - D_{31} – subscripted 2 digits mean bit location
  - D0 – unsubscripted one digit means register name



Memory Organization

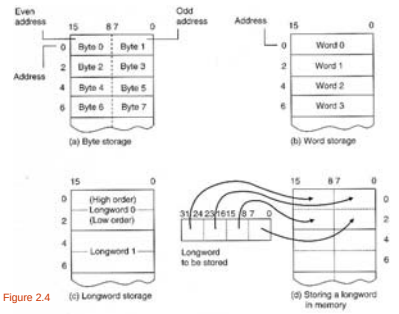
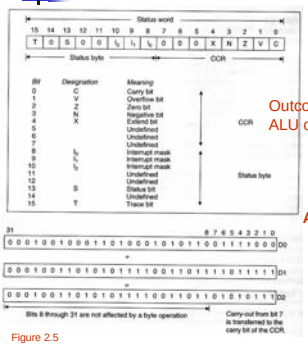


Figure 2.4

Long word address **= Address of the high order 16 bits of the longword**
Big-Endian **– The most significant unit is stored at the lowest address**

Special Purpose Registers
Status Register



Outcome of ALU operation

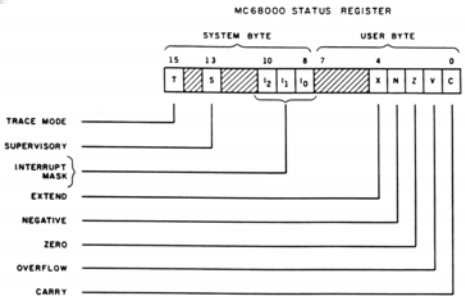
ADD.B D0,D1
78
DF
157
Carry

Figure 2.5

PC **– *Program Counter*:**

32 bits, contains the address of the next instruction to be executed

The Bits of the MC68000 Status Register



C, V and X Bits of Status Register

ADD.B

78
+DF
57
Carry 157
C - set

40
+70
B0
1011 0000
sign → V - set

```
int a;
char b;

a=(int)b;
```

X - extend
… 11111111 10110000

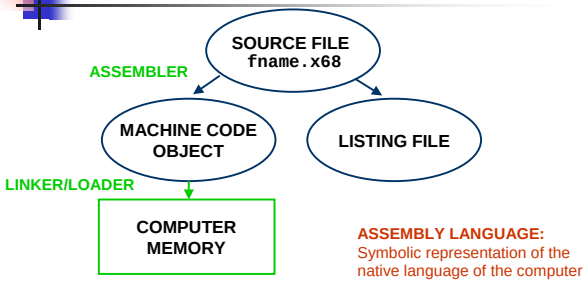
Outline

- Programmer's Model
- Assembly Language Directives
- Addressing Modes
- Instruction Set

Assembly Language Programming

- Machine code/Assembly language
  - A form of the native language of a computer
- Development environment
- Assembly program structure
- Assembly directives

Assembly Language Programming



ASSEMBLY LANGUAGE:
Symbolic representation of the native language of the computer

MACHINE INSTRUCTIONS → MNEMONICS

ADDRESSES & CONSTANTS → SYMBOLS

Assembly Language Program: Example

```
BACK-SP   EQU    $08          ASCII code for backspace
DELETE    EQU    $01          ASCII code for delete
CAR-RET   EQU    $OD          ASCII code for carriage return
          ORG    $004000      Data origin
LINE-BUF  DS.B   64           Reserve 64 bytes for line buffer
*
* This procedure inputs a character and stores it in a buffer
          ORG    $001000      Program origin
          LEA    LINE-BUF,A2  A2 points to line buffer

NEXT      BSR    GET_DATA     Call subroutine to get input
          CMP.B  #BACK_SP, D0 Test for backspace
          BEQ    MOVE_LFT     If backspace then deal with it
          CMP.B  #DELETE      Test for delete
          BEQ    CANCEL       If delete then deal with it
          CMP.B  #CAR-RET     Test for carriage return
          BEQ    EXIT         If carriage return then exit
          MOVE.B DO,(A2)+     Else store input in memory
          BRA    NEXT         Repeat
          .
          .                   Remainder of program
          END    $001000
```

$ represents HEX
% represents BIN

 # indicates a literal or immediate value (i.e. not an address)

LABEL FIELD **INSTRUCTION FIELD** **COMMENT FIELD**

Assembly Language Program

- 3 fields associated with each line:
  - LABELS
    - Start in the first column of a line
    - Refers to the address of the line it labels
    - Must be 8 or less characters
    - Start with a non-number
  - INSTRUCTION
    - Mnemonic (op code) and 0 or more parameters (operands)
    - Parameters separated by commas
  - COMMENTS
    - Can appear after instruction (many assemblers require a ; or ‘)
    - Can also be used in label field

Assembly Language Program (cont’d)

- Macroassembler
  - A MACRO: a unit of inline code that is given a name by the programmer
  - Example:
    - Instruction to push data on the stack:
    - MOVE.W D0, -(A7)
    - Define the macro:
    - PUSH D0
to replace it
  - Can define a macro for more than one instruction

Assembler Directives

- EQU – The equate directive
- DC – The define a constant directive
- DS – The define a storage directive
- ORG – The origin directive
- END – The end directive

The DC Directive

```
        ORG     $001000     Start of data region
First   DC.B    10,66       The values 10 and 66 are stored in consecutive bytes
        DC.L    $0A1234     The value $000A1234 is stored as a longword
Date    DC.B    'April 8 1985'  The ASCII characters as stored as a sequence of 12 bytes
        DC.L    1,2         Two longwords are set up with the values 1 and 2
```

address	Mem. contents		
001000	0A	42	DC.B 10,66
001002	00	0A	DC.L $0A1234
001004	12	34	
001006	41	70	DC.B ‘April 8 1985’
001008	72	69	
00100A	6C	20	
00100C	38	20	
00100E	31	39	
001010	38	35	
001012	00	00	DC.L 1,2
001014	00	01	
001016	00	00	
001018	00	02	
00101A			

The DC Directive (cont'd)

Assembler listing

```
1   00001000                        ORG     $001000
2   00001000  0A42          FIRST:  DC.B    10,66
3   00001002  000A1234              DC.L    $0A1234
4   00001006  417072696C20 DATE:   DC.B    'April 8 1995'
              382031393835
5   00001012  000000010000          DC.L    1,2
              0002
```

NOTE: Assemblers automatically align word and longword constants on a word boundary

- DC – *define a constant*
  - .B, .W, .L – specify 8, 16, or 32-bit constants
  - Normally preceded by a label to enable referring
  - Prefix:
    - Decimal
    - $ - Hexadecimal
    - % - Binary

DS – The *Define Storage* Directive

- Reserves the specified amount of storage
- Label DS.<size> <operand>

.B, .W, or .L — Number of elements

List1	DS.B	4	Reserve	4	bytes of memory
Array4	DS.B	$80	Reserve	128	bytes of memory
Pointer	DSLB	16	Reserve	16	longwords (64 bytes)
VOLTS	DS.W	1	Reserve	1	word (2 bytes)
TABLE	DS.W	256	Reserve	256	words

- Unlike DC does not initialize the values
- Useful to reserve areas of memory that will be used during run time
- Label is set to equal the first address of storage

ORG – The Origin Assembler Directive

- Defines the value of the *location counter*
- ORG <operand> ➡ Absolute value of the origin

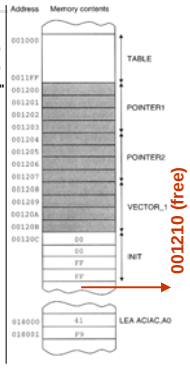
```
          ORG     $001000  Origin for data
TABLE     DS.W    256      Save 256 words for "TABLE"
POINTER1  DS.L    1        Save one longword for "POINTER1"
POINTER2  DS.L    1        Save one longword for "POINTER2"
VECTOR_1  DS.L    1        Save one longword for "VECTOR_1"
INIT      DC.W    0,$FFFF  Store two constants ($0000, $FFFF)
SETUP1    EQU     $03      Equate "SETUP1" to the value 3
SETUP2    EQU     $55      Equate "SETUP2" to the value $55
ACIAC     EQU     $008000  Equate "ACIAC" to the value $8000
RDRF      EQU     0        RDRF = Receiver Data Register Full
PIA       EQU     ACIAC+4  Equate "PIA" to the value $8004
```


Assembler Directives: Example

```
          ORG     $001000  Origin for data
TABLE     DS.W    256      Save 256 words for "TABLE"
POINTER1  DS.L    1        Save one longword for "POINTER1"
POINTER2  DS.L    1        Save one longword for "POINTER2"
VECTOR_1  DS.L    1        Save one longword for "VECTOR_1"
INIT      DC.W    0,$FFFF  Store two constants ($0000, $FFFF)
SETUP1    EQU     $03      Equate "SETUP1" to the value 3
SETUP2    EQU     $55      Equate "SETUP2" to the value $55
ACIAC     EQU     $008000  Equate "ACIAC" to the value $8000
RDRF      EQU     0        RDRF = Receiver Data Register Full
PIA       EQU     ACIAC+4  Equate "PIA" to the value $8004

          ORG     $018000  Origin for program
ENTRY     LEA     ACIAC,A0A0  points to the ACIA
          MOVE.B #SETUP2,(A0)  Write initialization
                               constant into ACIA

GET_DATA  BTST.B  #RDRF,(A0)  Any data received?
          BNE     GET_DATA    Repeat until data ready
          MOVE.B 2(A0),D0     Read data from ACIA
          END     $001000
```



Assembler Directives: Example

```
 1  00001000                           ORG     $001000
 2  00001000  00000200      TABLE:     DS.W    256
 3  00001200  00000004      POINTER1:  DS.L    1
 4  00001204  00000004      POINTER2:  DS.L    1
 5  00001208  00000004      VECTOR_1:  DS.L    1
 6  0000120C  0000FFFF      INIT:      DC.W    0,$FFFF
 7            00000003      SETUP1:    EQU     $03
 8            00000055      SETUP2:    EQU     $55
 9            00008000      ACIAC:     EQU     $008000
10            00000000      RDRF:      EQU     0
11            00008004      PIA:       EQU     ACIAC+4
12                          *
13  00018000                           ORG     $018000
14  00018000  41F900008000  ENTRY:     LEA     ACIAC,A0
15  00018006  10BC0055                 MOVE.B  #SETUP2,(A0)
16                          *
17  0001800A  08100000      GET_DATA:  BTST.B  #RDRF,(A0)
18  0001800E  66FA                     BNE     GET_DATA
19  00018010  10280002                 MOVE.B  2(A0),D0
```


Outline

- Programmer's Model
- Assembly Language Directives
- Addressing Modes
- Instruction Set

Addressing Modes

Addressing modes are concerned with how the CPU accesses the operands used by its instructions

Register Transfer Language (RTL)

- Unambiguous notation to describe information manipulation
- Registers are denoted by their names (eg. D1-D7, A0-A7)
- Square brackets mean "the contents of"
- Base number noted by a prefix (%-binary, $-hex)
- Backward arrow indicates a transfer of information (←)

[D4] ← 50	Put 50 into register D4
[D4] ← $1234	Put $1234 into register D4
[D3] ← $FE 1234	Put $FE 1234 into register D3

Register Transfer Language (RTL)

SYMBOL	Meaning
M	Location (i.e., address) M in the main store
Ai	Address register i (i = 0 to 7)
Di	Data register i (i = 0 to 7)
Xi	General register i
[M]	The contents of memory location M
[X]	The contents of register X
[Di(0:7)]	Bits 0 to 7 inclusive of register Di
<>	Enclose a parameter required by an expression
ea	The effective address of an operand
[M(ea)]	The contents of a memory location specified by ea
d8	An 8-bit signed offset (-128 to 127)
d16	A 16-bit signed offset (-32K to 32K -1)
d32	A 32-bit signed offset (-2G to 2G- 1)

ADD <source>,<destination>
[destination] ← [source] + [destination]

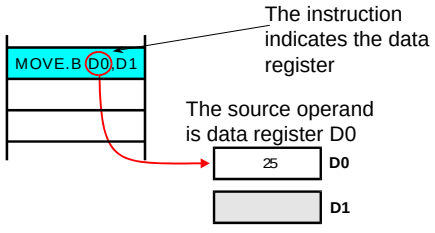
MOVE <source>,<destination>
[destination] ← [source]

Register Direct Addressing

Register direct addressing is the simplest addressing mode in which the source or destination of an operand is a data register or an address register. The contents of the specified source register provide the source operand. Similarly, if a register is a destination operand, it is loaded with the value specified by the instruction. The following examples all use register direct addressing for source and destination operands.

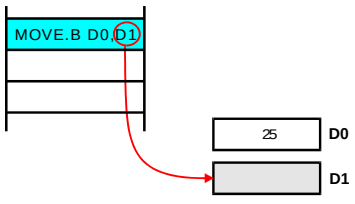
```
MOVE.B D0,D3  D3[0:7] <- D0[0:7]
SUB.L  A0,D3  Subtract the source operand in register A0 from register D3
CMP.W  D2,D0  Compare the source operand in register D2 with register D0
ADD    D3,D4  Add the source operand in register D3 to register D4
```


Register Direct Addressing



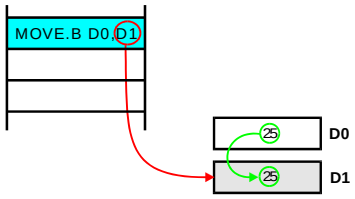
The MOVE.B D0,D1 instruction uses data registers for both source and destination operands

Register Direct Addressing



The destination operand is data register D1

Register Direct Addressing



The effect of this instruction is **TO COPY** the contents of data register D0 in to data register D1

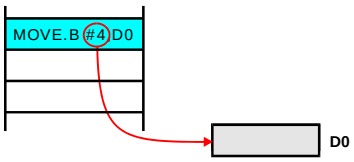
Register Direct Addressing

- Register direct addressing uses short instructions because it takes only three bits to specify one of eight data registers.
- Register direct addressing is fast because the external memory does not have to be accessed.
- Programmers use register direct addressing to hold variables that are frequently accessed (i.e., scratchpad storage).

Immediate Addressing

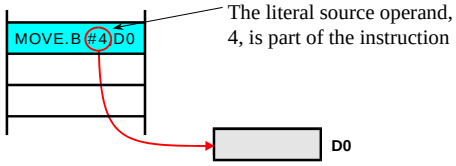
- In immediate addressing the actual operand forms part of the instruction. An immediate operand is also called a literal operand. Immediate addressing can be used only to specify a source operand.
- Immediate addressing is indicated by a # symbol in front of the source operand.
- For example, MOVE.B #24,D0 uses the immediate source operand 24.

Immediate Addressing

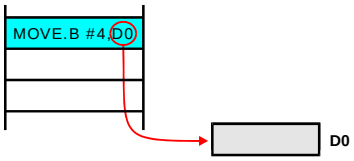


The instruction MOVE.B #4,D0 uses a literal source operand and a register direct destination operand

Immediate Addressing

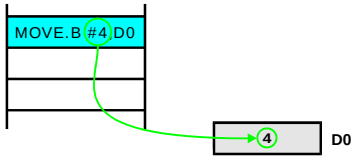


Immediate Addressing



The destination operand is a data register

Immediate Addressing



The effect of this instruction is to copy the literal value 4 to data register D0

Immediate Addressing Example

- Typical application is in setting up control loops:

```
for(i=0; i<128; i++)
  A(i) = 0xFF;
```

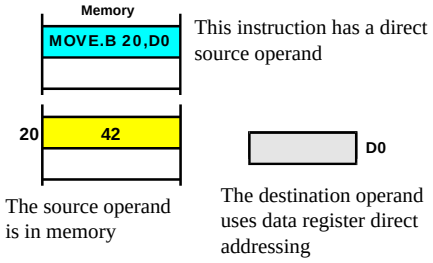
- 68000 assembly language implementation:

```
MOVE.L #$001000,A0 Load A0 with the address of the array
MOVE.B #128, D0          D0 is the element counter
LOOP   MOVE.B #$FF,(A0)+  Store $FF in this elem. and incr. pointer
       SUBQ.B #1,D0       Decrement element counter
       BNE LOOP     Repeat until all the elements are set
```

Direct (or Absolute) Addressing

- In direct or absolute addressing, the instruction provides the address of the operand in memory.
- Direct addressing requires two memory accesses. The first is to access the instruction and the second is to access the actual operand.
- For example, CLR.B 1234 clears the contents of memory location 1234.

Direct (or Absolute) Addressing

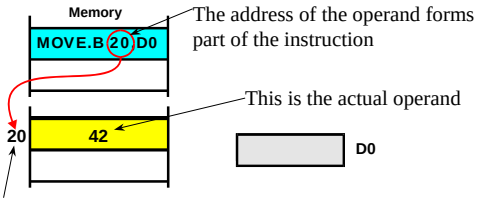


This instruction has a direct source operand

The source operand is in memory

The destination operand uses data register direct addressing

Direct (or Absolute) Addressing

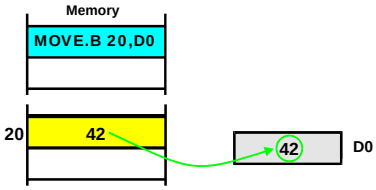


The address of the operand forms part of the instruction

This is the actual operand

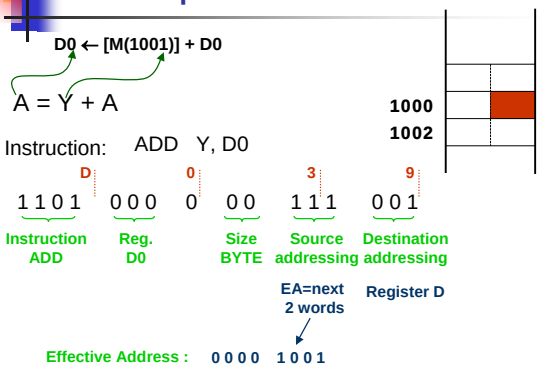
Once the CPU has read the operand address from the instruction, the CPU accesses the actual operand

Direct (or Absolute) Addressing

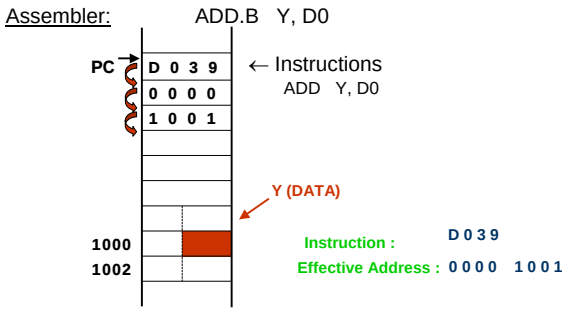


The effect of MOVE.B 20,D0 is to read the contents of memory location 20 and copy them to D0

An Example



An Example



Summary of Fundamental Addressing Modes

- Consider the high-level language example: Z = Y + 4
- The following fragment of code implements this construct

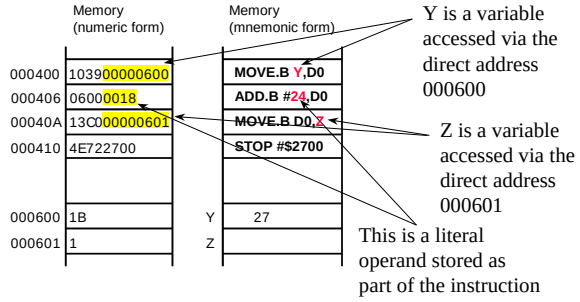
```
        ORG     $400    Start of code
        MOVE.B  Y,D0
        ADD     #4,D0
        MOVE.B  D0,Z

        ORG     $600    Start of data area
Y       DC.B    27      Store the constant 27 in memory
Z       DS.B    1       Reserve a byte for Z
```


The Assembled Program

```
1  00000400                     ORG     $400
2  00000400 103900000600        MOVE.B  Y,D0
3  00000406 06000018            ADD.B   #24,D0
4  0000040A 13C000000601        MOVE.B  D0,Z
5  00000410 4E722700            STOP    #$2700
6                          *
7  00000600                     ORG     $600
8  00000600 1B            Y:    DC.B    27
9  00000601 00000001      Z:    DS.B    1
10          00000400            END     $400
```

Memory Map of the Program



Summary

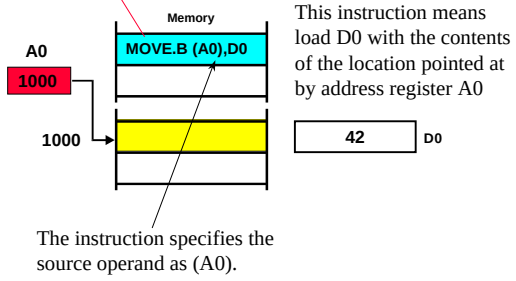
- Register direct addressing is used for variables that can be held in registers
- Literal (immediate) addressing is used for constants that do not change
- Direct (absolute) addressing is used for variables that reside in memory
- The only difference between register direct addressing and direct addressing is that the former uses registers to store operands and the latter uses memory

Address Register Indirect Addressing

- In address register indirect addressing, the instruction specifies one of the 68000's address registers;
  for example, MOVE.B (A0),D0.
- The specified address register contains the address of the operand.
- The processor then accesses the operand pointed at by the address register.

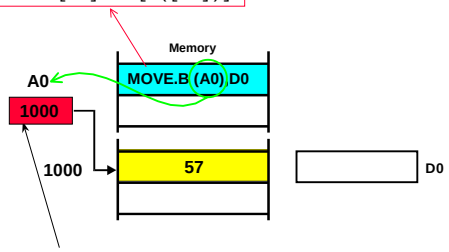
Address Register Indirect Addressing

RTL Form: `[D0] ← [M([A0])]`



Address Register Indirect Addressing

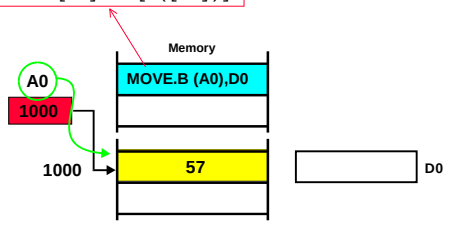
RTL Form: [D0] ← [M([A0])]



The address register in the instruction specifies an address register that holds the address of the operand

Address Register Indirect Addressing

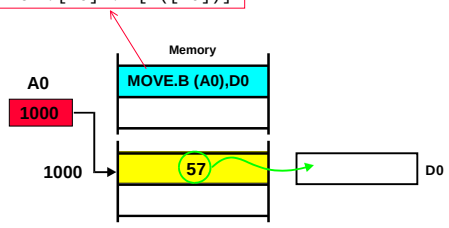
RTL Form: [D0] ← [M([A0])]



The address register is used to access the operand in memory

Address Register Indirect Addressing

RTL Form: [D0] ← [M([A0])]

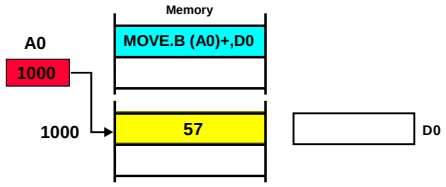


Finally, the contents of the address register pointed at by A0 are copied to the data register

Auto-incrementing

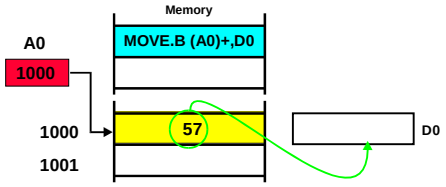
If the addressing mode is specified as (A0)+, the contents of the address register are incremented after they have been used.

Auto-incrementing



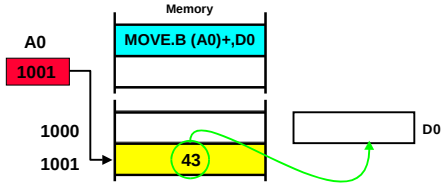
The address register contains 1000 and points at location 1000

Auto-incrementing



Address register A0 is used to access memory location 1000 and the contents of this location (i.e., 57) are added to D0

Auto-incrementing



After the instruction has been executed, the contents of A0 are incremented to point at the next location

Use of Address Register Indirect Addressing

The following fragment of code uses address register indirect addressing with post-incrementing to add together five numbers stored in consecutive memory locations.

```
        MOVE.B  #5,D0           Five numbers to add
        LEA     Table,A0        A0 points at the numbers
        CLR.B   D1              Clear the sum
Loop    ADD.B   (A0)+,D1        REPEAT Add number to total
        SUB.B   #1,D0
        BNE     Loop            UNTIL all numbers added
        STOP    #$2700
*
Table   DC.B    1,4,2,6,5       Some dummy data
```

We are now going to trace through part of this program, instruction by instruction.

Use of Address Register Indirect Addressing

```
>DF
PC=000400 SR=2000 SS=00A00000 US=00000000       X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.B    #$05,D0

>TR
PC=000404 SR=2000 SS=00A00000 US=00000000       X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000005 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->LEA.L     $0416,A0

Trace>
PC=00040A SR=2000 SS=00A00000 US=00000000       X=0
A0=00000416 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000005 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->CLR.B     D1
```

The first instruction loads D0 with the literal value 5

D0 has been loaded with 5

This instruction loads A0 with the value $0416

A0 contains $0416

Use of Address Register Indirect Addressing

```
Trace>
PC=00040C SR=2004 SS=00A00000 US=00000000       X=0
A0=00000416 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=1
D0=00000005 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->ADD.B     (A0)+,D1

Trace>
PC=00040E SR=2000 SS=00A00000 US=00000000       X=0
A0=00000417 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000005 D1=00000001 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->SUBQ.B    #$01,D0

Trace>
PC=000410 SR=2000 SS=00A00000 US=00000000       X=0
A0=00000417 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000004 D1=00000001 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->BNE.S     $040C
```

This instruction adds the contents of the location pointed at by A0 to D1

Because the operand was (A0)+, the contents of A0 are incremented

`ADD.B (A0)+,D1` adds the source operand to D1

Use of Address Register Indirect Addressing

```
Trace>
PC=00040C SR=2000 SS=00A00000 US=00000000       X=0
A0=00000417 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000004 D1=00000001 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->ADD.B     (A0)+,D1

Trace>
PC=00040E SR=2000 SS=00A00000 US=00000000       X=0
A0=00000418 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000004 D1=00000005 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->SUBQ.B    #$01,D0

Trace>

PC=000410 SR=2000 SS=00A00000 US=00000000       X=0
A0=00000418 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0
D0=00000003 D1=00000005 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->BNE.S     $040C
```

On the next cycle the instruction `ADD.B (A0)+,D1` uses A0 as a source operand and then increments the contents of A0

Problem

Identify the source addressing mode used by each of the following instructions.

`ADD.B (A5),(A4)`

Address register indirect addressing. The address of the source operand is in A5.

`MOVE.B #12,D2`

Literal addressing. The source operand is the literal value 12.

`ADD.W TIME,D4`

Memory direct addressing. The source operand is the contents of the memory location whose symbolic name is “TIME”.

`MOVE.B D6,D4`

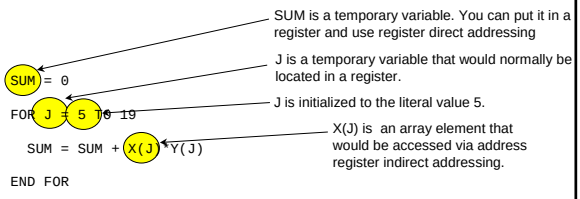
Data register direct. The source operand is the contents to D6.

`MOVE.B (A6)+,TEST`

Address register indirect with post-incrementing. The address of the source operand is in A6. The contents of A6 are incremented after the instruction.

Problem

If you were translating the following fragment of pseudocode into assembly language, what addressing modes are you most likely to use?



Other ARI Addressing Modes

- Address Register Indirect with Predecrement Addressing

  `MOVE.L –(A0),D3` (A0 is first decremented by **4**!)
  - `Combination: MOVE.B (A0)+,(A1)+`
    `MOVE.B –(A1),(A0)+`
- Register Indirect with Displacement Addressing

  `d16(Ai)  RTL: ea=d16+[Ai]`
- Register Indirect with Index Addressing

  `d8(Ai,Xj.W) or d8(Ai,Xj.L)`
  `RTL: ea=d8+[Ai]+[Xj]`

Other ARI Addressing Modes

- Program Counter Relative Addressing
  - Program Counter With Displacement

    `d16(PC)  RTL: ea=[PC]+d16`
  - Program Counter With Index

    `d16(PC)  RTL: ea=[PC]+[Xn]+d16`
- **PC can be used only for SOURCE OPERANDS**

```
        MOVE.B      TABLE(PC),D2
        …
TABLE   DC.B        Value1
        DC.B        Value2
```


Summary Addressing Modes

- Register direct addressing is used for variables that can be held in registers: **`ADD.B D1,D0`**
- Literal (immediate) addressing is used for constants that do not change: **`ADD.B #24,D0`**
- Direct (absolute) addressing is used for variables that reside in memory: **`ADD.B 1000,D0`**
- Address Register Indirect: **`ADD.B (A0),D0`**
- Autoincrement: **`ADD.B (A0)+,D0`**

Summary
Addressing Modes

- Address Register Indirect with Pre-decrement Addressing

  `MOVE.L –(A0),D3` (A0 is first decremented by **4**!)
  - `Combination: MOVE.B (A0)+,(A1)+`
    `MOVE.B –(A1),(A0)+`
- Register Indirect with Displacement Addressing

  `d16(Ai)  RTL: ea=d16+[Ai]`
- Register Indirect with Index Addressing

  `d8(Ai,Xj.W) or d8(Ai,Xj.L)`
  `RTL: ea=d8+[Ai]+[Xj]`

Summary
Addressing Modes

- Program Counter Relative Addressing
  - Program Counter With Displacement

    `d16(PC)  RTL: ea=[PC]+d16`
  - Program Counter With Index

    `d16(PC)  RTL: ea=[PC]+[Xn]+d16`
- **PC can be used only for SOURCE OPERANDS**

```
        MOVE.B      TABLE(PC),D2
        …
TABLE   DC.B        Value1
        DC.B        Value2
```


Outline

- Programmer's Model
- Assembly Language Directives
- Addressing Modes
- Instruction Set

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 – to/from User Stack Pointer
  `MOVE.L USP,A3 - transfer the USP to A3`
- MOVEQ – Move Quick(8b #value to 32b reg)
- MOVEM – to/from multiple registers (W/L)
  e.g., `MOVEM.L D0-D5/A0-A5, -(A7)`
  `MOVEM.L (A7)+,D0-D5/A0-A5`
- MOVEP – Move Peripheral

Data Movement Operations, LEA

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

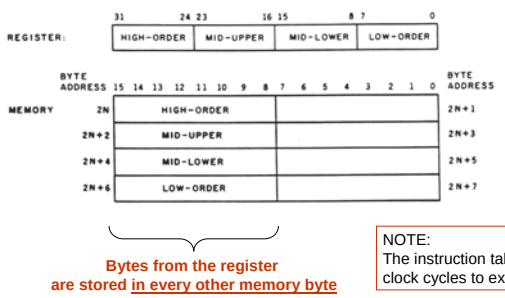
Assembly language	RTL
`LEA $0010FFFF,A5`	`[A5] ← $0010FFFF`
Load the address $0010 FFFF into register A5.	
`LEA $12(A0,D4.L),A5`	`[A5] ← $12 + [A0] + [D4]`
Load contents of A0 plus contents of D4 plus $12 into A5.	

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
```


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

An Example

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 register2700

Main memory

007000	AE	007020	5A	010000	DD	010020	DC
007001	F2	007021	A0	010001	B2	010021	25
007002	32	007022	99	010002	00	010022	15
007003	77	007023	92	010003	15	010023	17
007004	89	007024	79	010004	76	010024	29
007005	90	007025	33	010005	19	010025	39
007006	1A	007026	97	010006	92	010026	49
007007	AE	007027	14	010007	26	010027	2D
007008	EE	007028	79	010008	17	010028	B2
007009	F1	007029	E7	010009	14	010029	62
00700A	F2	00702A	00	01000A	E7	01002A	81
00700B	A4	00702B	0A	01000B	EB	01002B	21
00700C	AE	00702C	B8	01000C	19	01002C	45
00700D	B8	00702D	18	01000D	92	01002D	18
00700E	AA	00702E	B2	01000E	19	01002E	31
00700F	E4	00702F	79	01000F	54	01002F	D9
007010	7E	007030	23	010010	45	010030	AA
007011	8D	007031	17	010011	99	010031	77
007012	9C	007032	46	010012	15	010032	78
007013	C4	007033	9E	010013	43	010033	AE
007014	B2	007034	FC	010014	25	010034	EA
007015	12	007035	FF	010015	76	010035	34
007016	39	007036	77	010016	89	010036	25
007017	90	007037	60	010017	17	010037	17
007018	00	007038	21	010018	B1	010038	15
007019	89	007039	42	010019	17	010039	14
00701A	14	00703A	55	01001A	4E	01003A	17
00701B	01	00703B	EA	01001B	72	01003B	F9
00701C	3D	00703C	61	01001C	33	01003C	8A
00701D	77	00703D	B1	01001D	23	01003D	0F
00701E	89	00703E	C9	01001E	E1	01003E	F2
00701F	9A	00703F	AA	01001F	CD	01003F	E5

An Example

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
1 00009000 ORG $9000
2 00009000 4BFA0FFE LEA TABLE1(PC),A5

EA = $00009000 + 2 + $0FFE = $0000A000 → 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 (EXG Xi,Xj)
  - Exchanges the entire 32-bit contents of two registers
- SWAP (SWAP Di)
  - Exchanges the upper- and lower-order words of a DATA register

Integer Arithmetic Operations

- Float-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
  - DIVU <ea>,Dn or DIVS <ea>,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
  NEG <ea>
- NEGX – Negate with Extend, used for multi-prec. arith.
- EXT – Sign Extend
  EXT.W Dn copies bit 7 to bits 8-15
  EXT.L Dn copies bit 15 to bits 16-31

BCD Arithmetic Operations

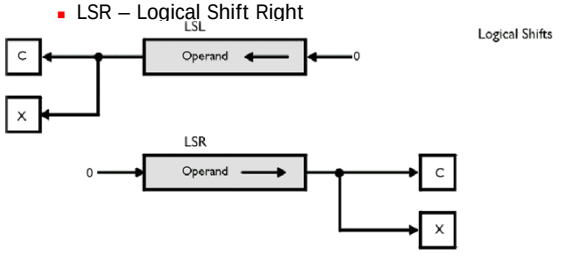
- Only 3 instructions support BCD
  - ABCD Di,Dj or ABCD –(Ai),-(Aj)
    Add BCD with extend – adds two packed BCD digits together with X bit from the CCR
  - SBCD – similar
    [destination]←[destination]-[source]-[X]
  - NBCD <ea>
    subtracts the specified operand from zero together with X bit and 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

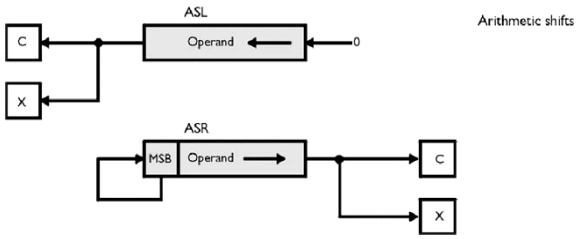
Shift Operations

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



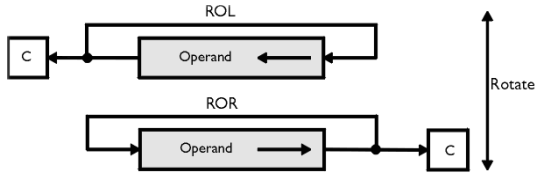
Shift Operations, cont'd

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



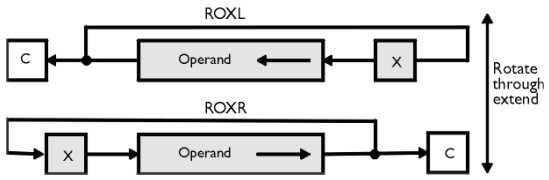
Shift Operations, cont'd

- 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

	Initial Value	After First Shift	CCR XNZVC	After Second Shift	CCR XNZVC
ASL	11101011	11010110	11001	10101100	11001
ASL	01111110	11111100	01010	11111000	11011
ASR	11101011	11110101	11001	11111010	11001
ASR	01111110	00111111	00000	00011111	10001
LSL	11101011	11010110	11001	10101100	11001
LSL	01111110	11111100	01000	11111000	11001
LSR	11101011	01110101	10001	00111010	10001
LSR	01111110	00111111	00000	00011111	10001
ROL	11101011	11010111	?1001	10101111	?1001
ROL	01111110	11111100	?1000	11111001	?1001
ROR	11101011	11110101	?1001	11111010	?1001
ROR	01111110	00111111	?0000	10011111	?1001

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

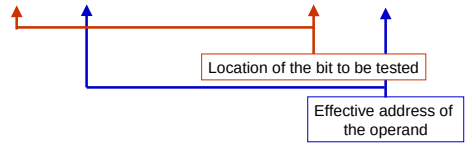
Bit Manipulation Operations

- 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
  - 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>

Stack Pointer

- First-in-last-out
- SP points to the element at the top of the stack
- Up to eight stacks simultaneously
- A7 used for subroutines
- A7 automatically adjusted by 2 or 4 for L or W ops.
- Push/pull implementation:

```
MOVE.W  Dn,-(A7)  <-PUSH
MOVE.W  (A7)+,Dn  <-PULL
```

- SSP/USP

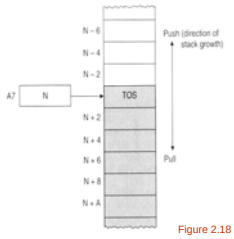


Figure 2.18

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

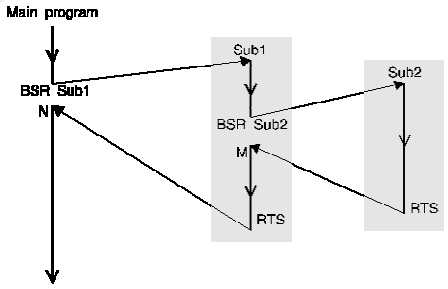
```
000FFA    41F900004000                    LEA   TABLE,
A0
001000    61000206    NextChr     BSR   GetChar
001004    10C0                    MOVE.B
    D0,(A0)
001006    0C00000D                CMP.B #$0D,D0
00100A    66F4                    BNE   NextChr
001102    61000104                BSR   GetChr
001106    0C000051                CMP.B #'Q',D0
00110A    67000EF4                BEQ   QUIT
001208    1239000080000     GetChr            MOVE.B
    ACIAC,D0
```

BSR d8 **d8=? (or d16, to specify d8 use BSR.S)**

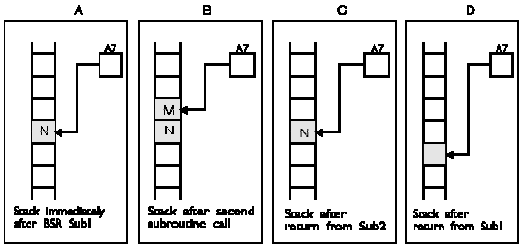
d8 = $00001208 – ($00001000 + 2) = $00000206

current PC value

Nested Subroutines



Nested Subroutines, cont'd



Nested Subroutines, cont'd

- Returning directly to a higher-level subroutine

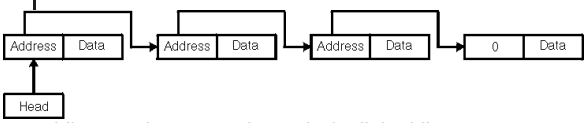
```
Sub2  .
      .
      BEQ   Exit
      .
      .
      RTS
Exit  LEA   4(A7),A7
      RTS
```

- 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
  Scc <ea> (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 #n
  Stop and load n into Status Register; n is 16-bit number; Privileged instruction
- CHK, RESET, RTE, TAS, TRAPV - later

Example: Linked List

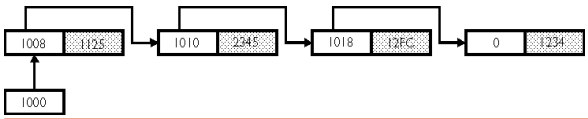


- Adding an element to the end of a linked list
  - HEAD points to the first element, NEW contains the address of the new item to be inserted
  - Longwords

```
        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

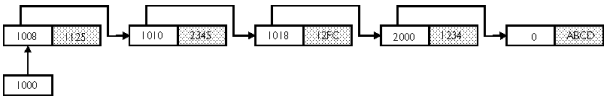
- Initial linked list:




```
        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

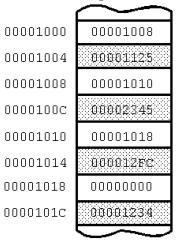
- Linked list after inserting an element at the end:



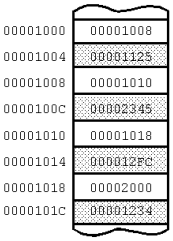

```
        LEA    HEAD,A0      A0 initially points to the start of the
*                           linked list
LOOP    TST.L (A0)          IF the address field = 0
        BEQ EXIT             THEN exit
        MOVEA.L (A0),A0      ELSE read the address of the next element
        BRA LOOP            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
```

Example: Linked List , Memory Map

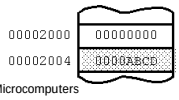
Memory map of linked list before inserting an element



Memory map of linked list after inserting an element



Note: The shaded memory elements represent data values



CPE/EE 421/521 Microcomputers 101