CPE/EE 421
Microcomputers
Instructor: Dr Aleksandar Milenkovic
Lecture Notes
S03

*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: Accounts ready! Lab#1 is on. First session 9/12

- Text: Microprocessor Systems Design:
  68000 Hardware, Software, and Interfacing

- Review: Introduction, M68000 Prog. Registers,

- Today: 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 (e.g., 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 (←)

\[ D4 \leftarrow 50 \]  
Put 50 into register D4

\[ D4 \leftarrow $1234 \]  
Put $1234 into register D4

\[ D3 \leftarrow $FE\ 1234 \]  
Put $FE\ 1234 into register D3

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

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  \( D3[0:7] \leftarrow D0[0:7] \)
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

The instruction indicates the data register

The source operand is data register D0

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

The effect of this instruction is TO COPY the contents of data register D0 into 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.

The literal source operand, 4, is part of the instruction.
Immediate Addressing

MOVE.B #4,D0

The destination operand is a data register

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:

```c
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

MOVE.B 20,D0

Memory

20

42

D0

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

This is the actual operand

The address of the operand forms part of the instruction

CPE/EE 421/521 Microcomputers

19

20
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.

Problem: \[ \text{[D0]} \leftarrow [M(1001)] + \text{[D0]} \]

\[ A = Y + A \]

Instruction:

1 1 0 1 0 0 0 0 0 0 1 1 1 0 0 1

- Instruction: ADD
- Reg. D0
- Size BYTE
- Source addressing
- Destination addressing
- Effective Address: 0 0 0 0 1 0 0 1

EA = next 2 words

Register D
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. ORG $400
2. MOV Y, D0
3. ADD #24, D0
4. MOV D0, Z
5. STOP "$2700"
6. *
7. ORG $600
8. DC B 27
9. DS B 1
10. END $400

### Memory Map of the Program

<table>
<thead>
<tr>
<th>Memory (numeric form)</th>
<th>Memory (mnemonic form)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000400</td>
<td>MOV Y, D0</td>
</tr>
<tr>
<td>000406</td>
<td>ADD #24, D0</td>
</tr>
<tr>
<td>00040A</td>
<td>MOV D0, Z</td>
</tr>
<tr>
<td>000410</td>
<td>STOP &quot;$2700&quot;</td>
</tr>
<tr>
<td>000600</td>
<td></td>
</tr>
<tr>
<td>000601</td>
<td></td>
</tr>
</tbody>
</table>

Y is a variable accessed via the direct address 000600

Z is a variable accessed via the direct address 000601

This is a literal operand stored as part of the instruction
Moving data from a 32-bit register to memory using the **MOVEP** instruction

NOTE:
The instruction takes 24 clock cycles to execute

---

**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
### The Assembled Program

<table>
<thead>
<tr>
<th></th>
<th>Address</th>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Operand</th>
<th>Address Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00000400</td>
<td>ORG</td>
<td>$400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>00000400</td>
<td>MOVE.B</td>
<td>Y, D0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00000406</td>
<td>ADD.B</td>
<td>#24, D0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0000040A</td>
<td>MOVE.B</td>
<td>D0, Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00000410</td>
<td>STOP</td>
<td>#$2700</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>00000600</td>
<td>ORG</td>
<td>$600</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00000600</td>
<td>DC.B</td>
<td>Y: 27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>00000601</td>
<td>DS.B</td>
<td>Z: 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00000400</td>
<td>END</td>
<td>$400</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Memory Map of the Program

- **Y** is a variable accessed via the direct address 000600.
- **Z** is a variable accessed via the direct address 000601.
- This is a literal operand stored as part of the instruction.
Moving data from a 32-bit register to memory using the **MOVEP** instruction

![Diagram showing the layout of bytes in a register and memory, with an explanation that bytes from the register are stored in every other memory byte.]

**NOTE:** The instruction takes 24 clock cycles to execute.

---

**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

This instruction means load D0 with the contents of the location pointed at by address register A0.

RTL Form: \([D0] \leftarrow [M([A0])]\)

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

RTL Form: \([D0] \leftarrow [M([A0])]\)

The instruction specifies the source operand as (A0).
Address Register Indirect Addressing

RTL Form: \([D0] \leftarrow [M([A0])]\)

The address register is used to access the operand in memory

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.

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

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      UNTIL all numbers added
         BNE     Loop
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

Trace>
PC=00040C SR=2004 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=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=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

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?

**SUM** is a temporary variable. You can put it in a register and use register direct addressing.

- **SUM = 0**
  - SUM is initialized to 0.

- **FOR J = 5 TO 19**
  - J is a temporary variable that would normally be located in a register.

- **SUM = SUM + X(J) * Y(J)**
  - X(J) is an array element that would be accessed via address register indirect addressing.

- **END FOR**
  - J is initialized to the literal value 5.

Other ARI Addressing Modes

- **Address Register Indirect with Predecrement Addressing**
  
  MOVE.L - (A0), D3 \( (A0 \text{ is first decremented by } 4!) \)
  
  - Combination: MOVE.B (A0)+, (A1)+
  - MOVE.B -(A1), (A0)+

- **Register Indirect with Displacement Addressing**

  \(d_{16}(Ai)\) RTL: \(ea=d_{16}+[Ai]\)

- **Register Indirect with Index Addressing**

  \(d_{8}(Ai,Xj.W)\) or \(d_{8}(Ai,Xj.L)\)
  RTL: \(ea=d_{8}+[Ai]+[Xj]\)

---

Other ARI Addressing Modes

- **Program Counter Relative Addressing**

  - Program Counter With Displacement
    \(d_{16}(PC)\) RTL: \(ea=[PC]+d_{16}\)

  - Program Counter With Index
    \(d_{16}(PC)\) RTL: \(ea=[PC]+[Xn]+d_{16}\)

- **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
    \[ d16(PC) \text{ RTL: } ea = [PC] + d16 \]
  - Program Counter With Index
    \[ d16(PC) \text{ RTL: } ea = [PC] + [Xn] + d16 \]

- **PC can be used only for SOURCE OPERANDS**

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

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
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 Quick(8b #value to 32b reg)

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

- MOVEP
  MOVE Peripheral
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

<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></td>
<td>Load the address $0010 FFFF into register A5.</td>
</tr>
<tr>
<td>LEA $12(A0,D4.L),A5</td>
<td>[A5] ← $12 + [A0] + [D4]</td>
</tr>
<tr>
<td></td>
<td>Load contents of A0 plus contents of D4 plus $12 into A5.</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
Example:

### 68000 Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>D1</td>
<td>00000100</td>
<td>1</td>
</tr>
<tr>
<td>D2</td>
<td>00000200</td>
<td>2</td>
</tr>
<tr>
<td>D3</td>
<td>00000300</td>
<td>3</td>
</tr>
<tr>
<td>D4</td>
<td>00000400</td>
<td>4</td>
</tr>
<tr>
<td>D5</td>
<td>00000500</td>
<td>5</td>
</tr>
<tr>
<td>D6</td>
<td>00000600</td>
<td>6</td>
</tr>
<tr>
<td>D7</td>
<td>00000700</td>
<td>7</td>
</tr>
</tbody>
</table>

### Main Memory

```
<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00000100</td>
<td>1</td>
</tr>
<tr>
<td>00000200</td>
<td>2</td>
</tr>
<tr>
<td>00000300</td>
<td>3</td>
</tr>
<tr>
<td>00000400</td>
<td>4</td>
</tr>
<tr>
<td>00000500</td>
<td>5</td>
</tr>
<tr>
<td>00000600</td>
<td>6</td>
</tr>
<tr>
<td>00000700</td>
<td>7</td>
</tr>
</tbody>
</table>
```

**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 00000000 ORG $9000
2 00000100 4FAD0FFE LEA TABLE1(PC),A5
```

**EA = \( \text{current PC value} + 2 + 0\text{FFE} = \text{00000A000} \) → \( A5=\text{00000A000}, \) CC: Not affected (NA)**

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

```
offset A0 D6.W
```

**EA = \( 6 + \text{000007020} + 0\text{003} = \text{000007029} \) → \( A2=\text{000007029}, \) CC: NA**
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
- 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

![Arithmetic Shift Diagram]

Shift Operations, cont’d

- **Rotate**
  - ROL – Rotate Left
  - ROR – Rotate Right

![Rotate Diagram]
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></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>10101100</td>
<td>11001</td>
</tr>
<tr>
<td>ASL</td>
<td>01111110</td>
<td>11111100</td>
<td>01010</td>
<td>11111000</td>
<td>11011</td>
</tr>
<tr>
<td>ASR</td>
<td>11101011</td>
<td>11110101</td>
<td>11001</td>
<td>11111010</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>11010110</td>
<td>11001</td>
<td>10101100</td>
<td>11001</td>
</tr>
<tr>
<td>LSL</td>
<td>01111110</td>
<td>11111100</td>
<td>01000</td>
<td>11111000</td>
<td>11001</td>
</tr>
<tr>
<td>LSR</td>
<td>11101011</td>
<td>01110101</td>
<td>10001</td>
<td>00111101</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>11111010</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**
  - 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>

  ![Diagram showing effective address of the operand and location of the bit to be tested]

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**
  - \( \text{BRA} \ <\text{label}> \) or \( \text{JMP} \ (\text{An}) \)
  - \( \text{JMP} \ d16(\text{An}) \)
  - \( \text{JMP} \ d8(\text{An},\text{Xi}) \)
  - \( \text{JMP} \ \text{Absolute} \ _\text{address} \)
  - \( \text{JMP} \ d16(\text{PC}) \)
  - \( \text{JMP} \ d8(\text{PC},\text{Xi}) \)

- **TEST CONDITION, DECREMENT, and BRANCH**
  - \( \text{DBcc} \ Dn,\ <\text{label}> \) (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**
  - \( \text{BSR} \ <\text{label}> = [\text{A7}] \leftarrow [\text{A7}] - 4 \)
  - \( M([\text{A7}]) \leftarrow [\text{PC}] \)
  - \( [\text{PC}] \leftarrow [\text{PC}] + d8 \)

- **RETURN FROM SUBROUTINE**
  - \( \text{RTS} \ = [\text{PC}] \leftarrow M([\text{A7}]) \)
  - \( [\text{A7}] \leftarrow [\text{A7}] + 4 \)
Subroutines, cont’d

- BRANCH TO SUBROUTINE
  
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000FFA</td>
<td>41F900004000</td>
<td>LEA TABLE, A0</td>
</tr>
<tr>
<td>001000</td>
<td>61000206</td>
<td>NextChr BSR GetChar</td>
</tr>
<tr>
<td>001004</td>
<td>10C0</td>
<td>MOVE.B D0, (A0)</td>
</tr>
<tr>
<td>001006</td>
<td>0C00000D</td>
<td>CMP.B #$0D,D0</td>
</tr>
<tr>
<td>00100A</td>
<td>66F4</td>
<td>BNE NextChr</td>
</tr>
<tr>
<td>001102</td>
<td>61000104</td>
<td>BSR GetChr</td>
</tr>
<tr>
<td>001106</td>
<td>0C000051</td>
<td>CMP.B #$'Q',D0</td>
</tr>
<tr>
<td>00110A</td>
<td>67000EF4</td>
<td>BEQ QUIT</td>
</tr>
<tr>
<td>001208</td>
<td>1239000080000</td>
<td>GetChr MOVE.B ACIAC,D0</td>
</tr>
</tbody>
</table>

BSR d8  

\[ d8 = \text{ current PC value } - \left( \text{ current PC value } + 2 \right) \]

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
  
  Scc <ea> \hspace{1cm} \text{(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

  MOV A, 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

  MOV A, 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

<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>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>00001004</td>
<td>00001012</td>
</tr>
<tr>
<td>00001008</td>
<td>00001010</td>
</tr>
<tr>
<td>0000100c</td>
<td>00001018</td>
</tr>
<tr>
<td>00001010</td>
<td>00001018</td>
</tr>
<tr>
<td>00001014</td>
<td>0000101c</td>
</tr>
<tr>
<td>00001013</td>
<td>00000000</td>
</tr>
<tr>
<td>0000101c</td>
<td>00001234</td>
</tr>
</tbody>
</table>

**Note:** The shaded memory elements represent data values.