CPE/EE 421/521
Fall 2004
Chapter 2 – Programming the 68000 Family

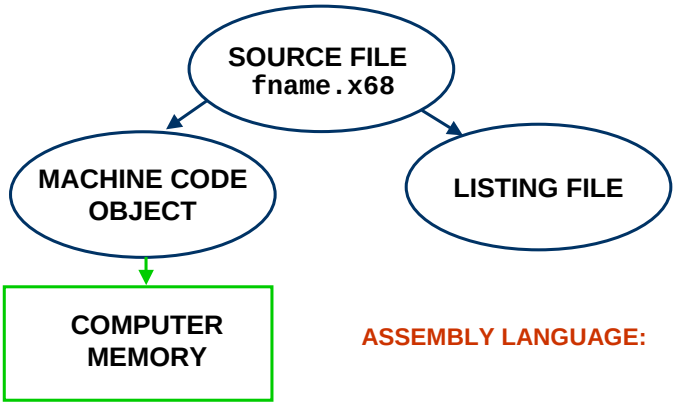
Dr. Rhonda Kay Gaede

UAH

Motorola 68000

- CISC processor – supports many _____
- 16 _____ registers
  - 8 general purpose _____ registers
  - 8 general purpose _____ registers
- Two levels of privilege
  - _____
  - _____
- Three types of data are supported
  - Byte - _____
  - Word - _____
  - Longword - _____
- 64-pin package
- Clock 8MHz, 12.5 MHz

2.1 Assembly Language Programming and the 68000 – Assembly Process



ASSEMBLY LANGUAGE:

MACHINE INSTRUCTIONS → MNEMONICS
ADDRESSES & CONSTANTS → SYMBOLS

2.1 Assembly Language Programming and the 68000 - Program Structure

- 3 fields associated with each line:
  - LABELS
    - _____
    - _____
    - _____
    - _____
  - INSTRUCTION
    - _____
    - _____
  - COMMENTS
    - _____
    - _____

2.1 Assembly Language Programming and the 68000 - 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
```


2.1 Assembly Language Programming and the 68000 – Assembler Directives

- **EQU** – The *equate* directive binds a name to a value – _____
  - Example:
- **ORG** – The *origin* directive defines the value of a location counter that tells _____.
  - Example:
- **DC** – The *define a constant* directive loads a constant into memory.
- **DS** – The *define a storage* directive reserves memory space for variables - _____
- **END** – The *end* directive tells the assembler that the program is finished.

2.1 Assembly Language Programming and the 68000 – **ORG** and **DC** Directives

```
        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
```


2.1 Assembly Language Programming and the 68000 – **DS** Directive

– Label DS.<size> <operand>

```
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
```

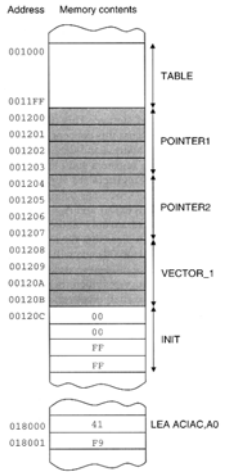
– Unlike DC _____

– Used for _____

– Label is set to equal _____

2.1 Assembly Language Programming and the 68000 – Directives Example

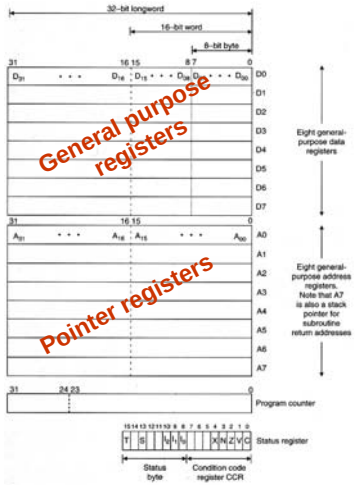
```
                ORG     $001000     Origin for data
TABLE           DS.W    256         Save 256 words for "TABLE"
POINTER1        DS.L    1           Save one longword for "POINTER1"
POINTER2        DS.L    1           Save one longword for "POINTER2"
VECTOR_1        DS.L    1           Save one longword for "VECTOR_1"
INIT            DC.W    0,$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,A0    A0      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
```



2.1 Assembly Language Prog. and the 68000 – Directives Example (cont'd)

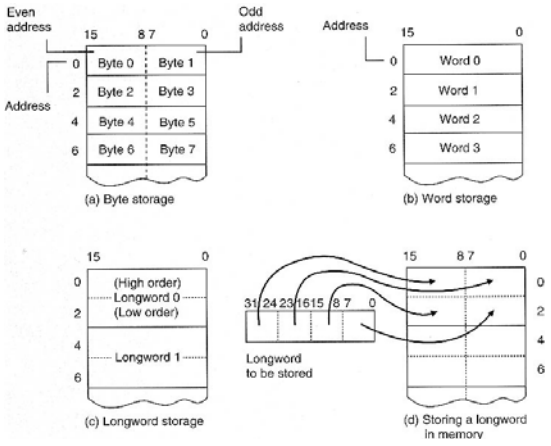
```
 1  00001000                           ORG     $001000
 2  00001000  00000200      TABLE:     DS.W    256
 3  00001200  00000004      POINTER1:  DS.L    1
 4  00001204  00000004      POINTER2:  DS.L    1
 5  00001208  00000004      VECTOR_1:  DS.L    1
 6  0000120C  0000FFFF      INIT:      DC.W    0,$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,AO
15  00018006  10BC0055                 MOVE.B  #SETUP2,(AO)
16                          *
17  0001800A  08100000      GET_DATA:  BTST.B  #RDRF,(AO)
18  0001800E  66FA                     BNE     GET_DATA
19  00018010  10280002                 MOVE.B  2(AO),DO
```

2.2 Programmer's Model of the 68000 - Registers



- Consists of Accessible 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

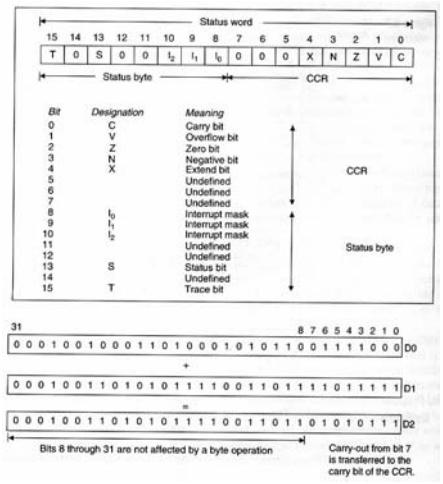
2.2 Programmer's Model of the 68000 – Memory Organization



Long word address =

Big-Endian –

2.2 Programmer's Model of the 68000 – Special Purpose Registers



2.3 Addressing Modes of the 68000

- Immediate
- Absolute or Direct
- Register Direct
- Address Register Indirect
  - with Postincrement
  - with Predecrement
  - with Displacement Addressing
  - with Index Addressing
- Program Counter Relative

2.3 Addressing Modes of the 68000 – 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
```


2.3 Addressing Modes of the 68000 – RTL (continued)

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)

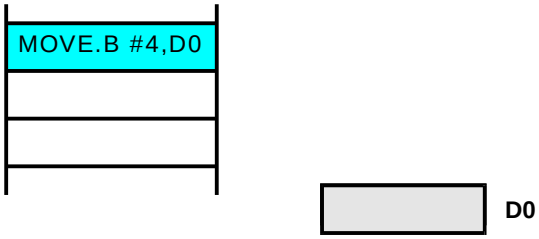
ADD <source>,<destination>
[destination] ← [source] + [destination]

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

2.3 Addressing Modes of the 68000 – Immediate Addressing

- In *immediate addressing* the actual operand is part of _____. An immediate operand is also called a literal operand. Immediate addressing can be used only to specify a _____ 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.

2.3 Addressing Modes of the 68000 – Immediate Addressing Example



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

2.3 Addressing Modes of the 68000 – Immediate Addressing Application

- Typical application is in setting up control loops:

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

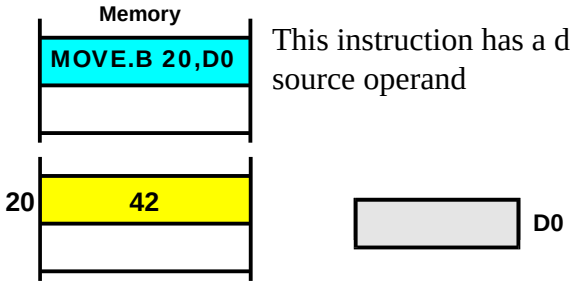
- 68000 assembly language implementation:

```
      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 here and increment pointer
      SUBQ.B  #1,D0        Decrement element counter
      BNE     LOOP         Repeat until all the elements are set
```


2.3 Addressing Modes of the 68000 – Absolute or Direct Addressing

- In direct or absolute addressing, the instruction provides the _____ in memory.
- Direct addressing requires _____ 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.

2.3 Addressing Modes of the 68000 – Direct Addressing Example



This instruction has a direct source operand

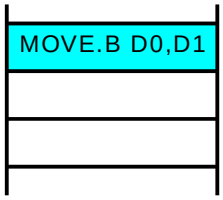
The source operand is in memory

2.3 Addressing Modes of the 68000 – Register Direct Addressing

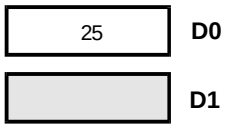
Register direct addressing is the simplest addressing mode in which the source or destination of an operand is _____. The contents of the specified source register provide the _____. Similarly, if a register is a _____, it is loaded with the value specified by the instruction. The following examples 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
```

2.3 Addressing Modes of the 68000 – Register Direct Addressing Example



The source operand is data register D0,



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

2.3 Addressing Modes of the 68000 – Register Direct Ramifications

- Register direct addressing uses _____ instructions because it takes only _____ bits to specify one of _____ data registers.
- Register direct addressing is _____ because _____ does not have to be accessed.
- Programmers use register direct addressing to hold variables that are frequently accessed _____.

2.3 Addressing Modes of the 68000 – An Example with Multiple Modes

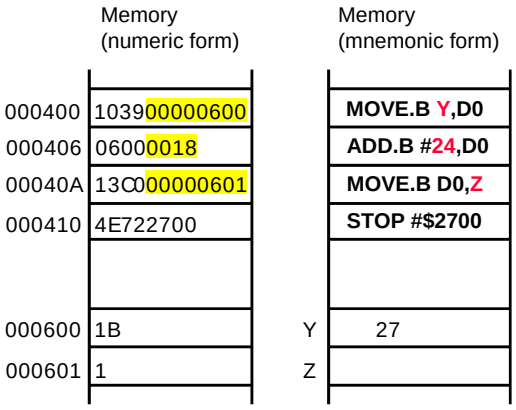
- Consider the high-level language example:
  `Z = Y + 4;`
- The following fragment of code implements this construct

```
        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
```


2.3 Addressing Modes of the 68000 – An Example with Multiple Modes

```
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
```

2.3 Addressing Modes of the 68000 – An Example with Multiple Modes

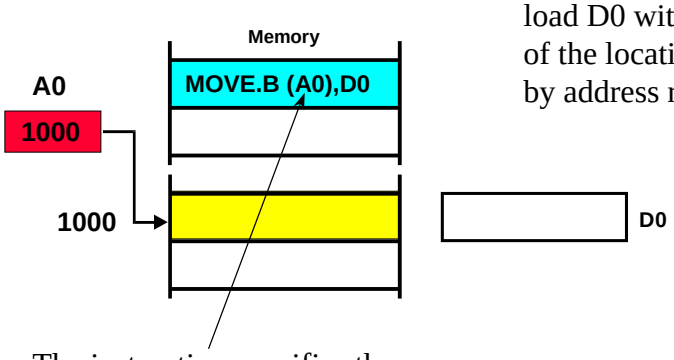


2.3 Addressing Modes of the 68000 – Address Register Indirect (ARI)

- 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 _____ _____.
- The processor then accesses the operand pointed at by the _____.

2.3 Addressing Modes of the 68000 – ARI Example

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

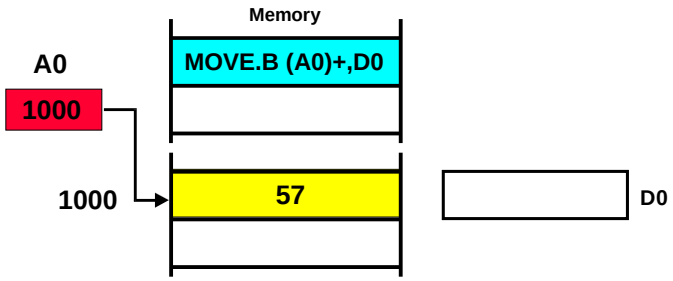


The instruction specifies the source operand as (A0).

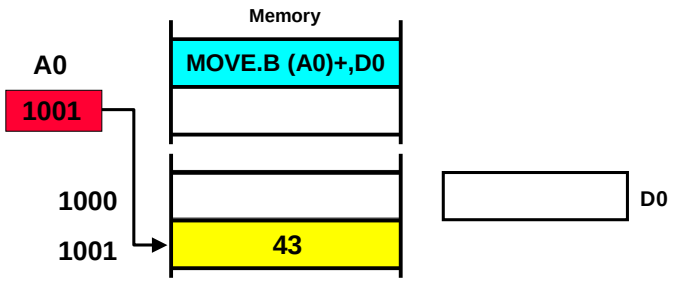
2.3 Addressing Modes of the 68000 – ARI with Update

- There are two flavors of update
  - Post-increment
  - Pre-decrement
- The high level constructs these resemble are **`++`** and **`--`**
  - **`i++`** – i is used, then changed
  - **`--i`** – i is changed, then used
- Think of **`i`** as a pointer moving through an array.

2.3 Addressing Modes of the 68000 – Post-Increment ARI (Step 1)



2.3 Addressing Modes of the 68000 – Post-Increment ARI (Step 2)



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

2.3 Addressing Modes of the 68000 – ARI Application

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

```
        ORG     $400
        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
        SUBQ.B  #1,D0
        BNE.S   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.

2.3 Addressing Modes of the 68000 – ARI Application Memory Map

```
0400        00000005          MOVE.B #5,D0
0404        41FC00000416      LEA Table,A0
040A        4201              CLR.B D1
040C        D218              ADD.B (A0)+,D1     Loop
040E        9001              SUBQ.B #1,D0
0410        66F6              BNE.S Loop
0412        4E722700          STOP #$2700
0416        0102              Table
0418        0206
041A        05
```

2.3 Addressing Modes of the 68000 – ARI Application Explained

```
>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
```


2.3 Addressing Modes of the 68000 – ARI Application Explained

```
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
```

2.3 Addressing Modes of the 68000 – ARI Application Explained

```
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
```


2.3 Addressing Modes of the 68000 – Addressing Modes Quiz

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

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

MOVE.B #12,D2

ADD.W  TIME,D4

MOVE.B D6,D4

MOVE.B (A6)+,TEST
```

2.3 Addressing Modes of the 68000 – Which Mode to Use?

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

```
SUM = 0
FOR J = 5 TO 19
  SUM = SUM + X(J)*Y(J)
END FOR
```


2.3 Addressing Modes of the 68000 – 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]`

2.3 Addressing Modes of the 68000 – 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
```


2.3 Addressing Modes of the 68000 – The Stack

- 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/pop implementation:

  `MOVE.W  Dn,-(A7)` PUSH

  `MOVE.W (A7)+,Dn` POP
- SSP/USP

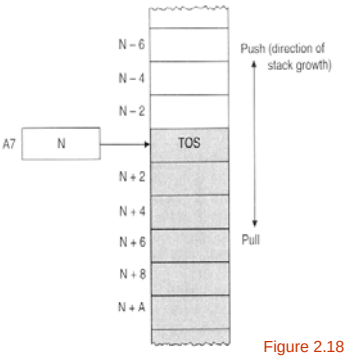


Figure 2.18

2.4 An Introduction to the 68000 Family Instruction Set

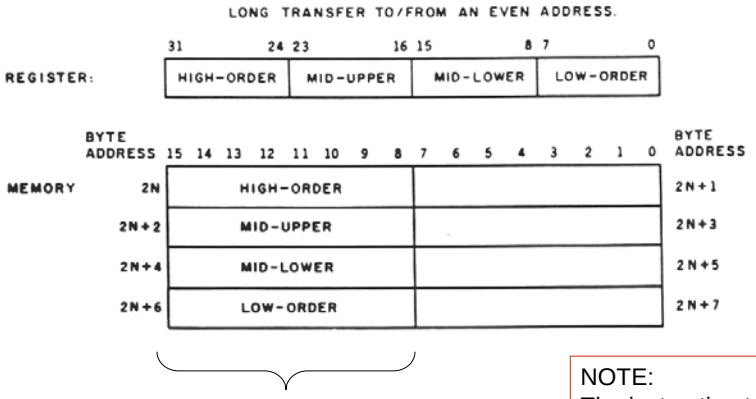
- Assumption: Students are familiar with the fundamentals of microprocessor architecture
- Groups of instructions:
  - _____
  - _____
  - _____
  - _____
  - _____
  - _____

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

2.4 An Introduction to the 68000 Family Instruction Set – Data Movement

- Copy information from source to destination - 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` – puts 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 register)
- 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

2.4 An Introduction to the 68000 Family Instruction Set – **MOVEP**



Bytes from the register are stored in every other memory byte

NOTE:
The instruction takes 24 clock cycles to execute

2.4 An Introduction to the 68000 Family Instruction Set – **LEA**

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

Assembly language	**RTL**
`LEA $0010FFFF,A5`	`[A5]` ← `$0010FFFF`
Load _____ *into register A5.*	
`LEA $12(A0,D4.L),A5`	`[A5]` ← `$12 + [A0] + [D4]`
Load _____ *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 if _____!

```
LEA $1C(A3,D2),A5 (12) vs. ADD.W $1C(A3,D2),D0 (14)
ADD.W(A5),D0      (8)
```

2.4 An Introduction to the 68000 Family Instruction Set – **LEA** Examples

```
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 register     2700
```

Given the initial condition shown above, what is the effect of applying each of the following 68000 instructions?

a)
```
ORG    $9000
LEA    TABLE1(PC),A5
Assembly listing
1      00009000          ORG      $9000
2      00009000 4BFA0FFE LEA      TABLE1(PC),A5
```

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


2.4 An Introduction to the 68000 Family Instruction Set – **PEA, EXG, SWAP**

- **PEA**
  - **P**ush **E**ffective **A**ddress
  - Calculates an effective address and pushes it onto the _____ – **`PEA <ea>`**
  - Can be used only with _____ operands
- **EXG**
  - Exchanges the entire 32-bit contents of _____
  - **`EXG Xi,Xj`**
- **SWAP**
  - Exchanges the upper- and lower-order words of a DATA register
  - **`SWAP Di`**

2.4 The 68000 Family Instruction Set – Integer Arithmetic Instructions

- _____ operations are not directly supported
- Except for division, multiplication, and if destination is **Ai**, all act on 8-, 16-, and 32-bit values
- **ADD/ADDA/SUB/SUBA** (no *mem-to-mem* additions, if destination is **Ai**, use **ADDA** or **SUBA**)
- **ADDQ/SUBQ** (adds a small 3-bit literal quickly)
- **ADDI/SUBI** (adds a literal value to the destination)
- **ADDX/SUBX** (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**

2.4 The 68000 Family Instruction Set – More Integer Arithmetic Instructions

- **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**
- **NEG** – forms the _____ of an operand **NEG <ea>**
- **NEGX** – Negate with Extend, used for multi-precision arithmetic
- **EXT** – Sign Extend

2.4 The 68000 Family Instruction Set – BCD Arithmetic Instructions

- **`ABCD Di,Dj`** and **`ABCD -(Ai),-(Aj)`**
  - Add BCD with extend – adds two packed BCD digits together with **X** bit from the CCR
- **`SBCD`** – similar except subtract `[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

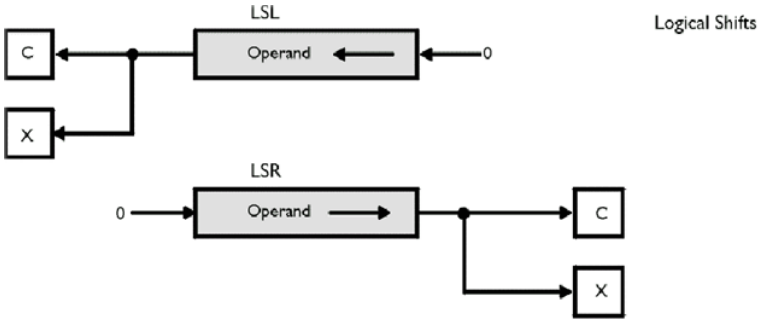
2.4 The 68000 Family Instruction Set – Logical Instructions

- 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*

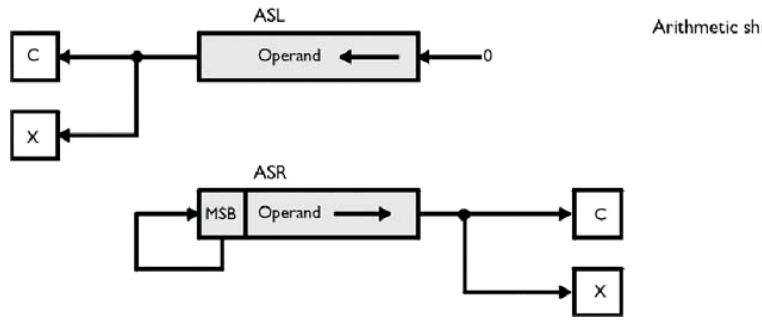
2.4 The 68000 Family Instruction Set – Logical Shift Instructions

- _____ – Logical Shift Left
- _____ – Logical Shift Right



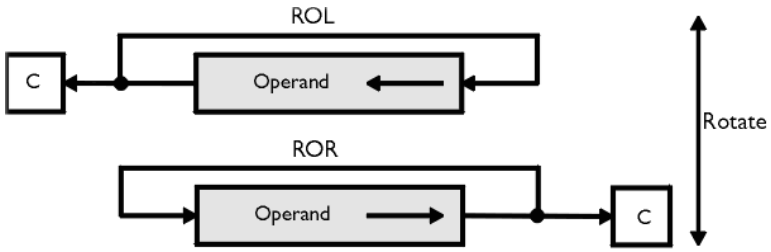
2.4 The 68000 Family Instruction Set – Arithmetic Shift Instructions

- _____ – Arithmetic Shift Left
- _____ – Arithmetic Shift Right



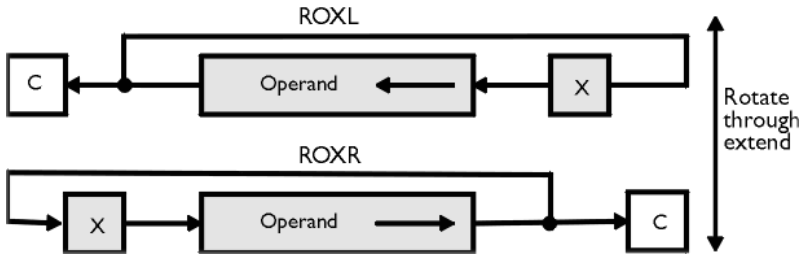
2.4 The 68000 Family Instruction Set – Rotate Instructions

- _____ – Rotate Left
- _____ – Rotate Right



2.4 The 68000 Family Instruction Set – Rotate Through Extend Instructions

- _____ – Rotate Left Through Extend
- _____ – Rotate Right Through Extend



2.4 The 68000 Family Instruction Set – Shift Instruction Examples

	Initial Value	After First Shift	CCR XNZVC	After Second Shift	CCR XNZVC
ASL	11101011				
ASL	01111110				
ASR	11101011				
ASR	01111110				
LSL	11101011				
LSL	01111110				
LSR	11101011				
LSR	01111110				
ROL	11101011				
ROL	01111110				
ROR	11101011				
ROR	01111110				

2.4 The 68000 Family Instruction Set – Shift Instruction Modes

- 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 _____

All three ____ apply to all _____ shift instructions

2.4 The 68000 Family Instruction Set – Bit Manipulation Instructions

- 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)

`BTST Dn, <ea>` **or** **`BTST #<data>,<ea>`**

2.5 Program Control and the 68000 – Compare Instructions

- **CMP**
  - **`CMP <ea1>,<ea2>`** **`[<ea2>]-[<ea1>]`**
- **CMPI**
  - **`CMP #<data>,<ea>`** - compares 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
- Update **NZVC** bits of the CCR

2.5 Program Control and the 68000 – Branch Instructions

- Branch Instructions
  - _____
  - _____
  - _____

2.5 Program Control and the 68000 – Conditional Branch Instructions

`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

2.5 Program Control and the 68000 – Unconditional Branch Instructions

```
BRA <label>
JMP (An)
JMP d16(An)
JMP d8(An,Xi)
JMP Absolute_address
JMP d16(PC)
JMP d8(PC,Xi)
```


2.5 Program Control and the 68000 – Test Condition, Decrement and Branch

```
DBcc Dn,<label>  (16 bit displacement only)

IF cc FALSE
   [Dn] := [Dn] - 1
   IF [Dn] != -1
       [PC] ← label
   END_IF
END_IF
```

2.6 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

2.7 Subroutines and the 68000 – Branch to and Return from Subroutine

- Branch to Subroutine

```
BSR <label> =   [A7] ← [A7] - 4
             M([A7])] ← [PC]
             [PC] ← [PC] + d16(d8)
```

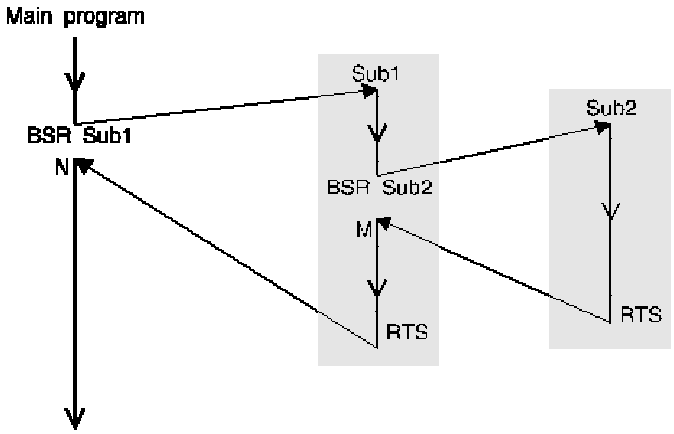
- Return from Subroutine

```
RTS          =    [PC] ← [M([A7])]
                  [A7]← [A7] + 4
```

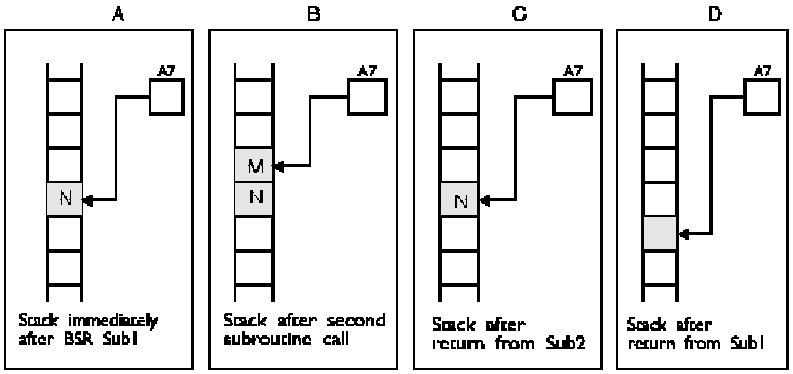
2.7 Subroutines and the 68000 – Branch to Subroutine Example

```
000FFA   41F900004000          LEA   TABLE, A0
001000   61000206     NextChr  BSR   GetChar
001004   10C0                  MOVE.B D0,(A0)
001006   0C00000D              CMP.B #$0D,D
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
```


2.7 Subroutines and the 68000 – Nested Subroutine Calls



2.7 Subroutines and the 68000 – Subroutines and the Stack



2.7 Subroutines and the 68000 – Skipping Levels of Return

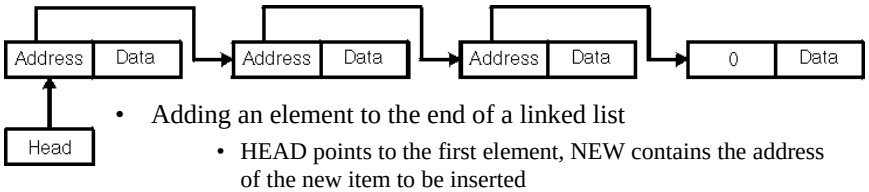
- Skipping one level of return

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

2.7 Subroutines and the 68000 – An Alternative to **RTS**

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

68000 Programming – Linked List Example

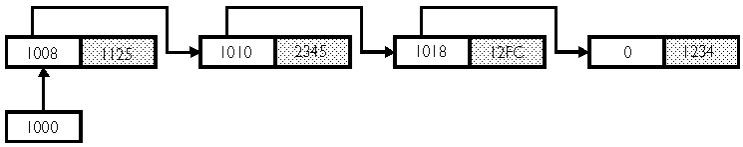


- Adding an element to the end of a linked list
  - HEAD points to the first element, NEW contains the address of the new item to be inserted
  - Longwords

```
        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
```

68000 Programming – Linked List Example (continued)

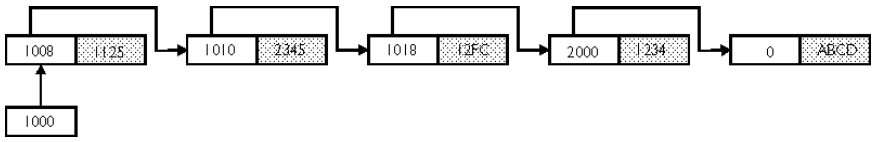
- Initial linked list:




```
        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
```

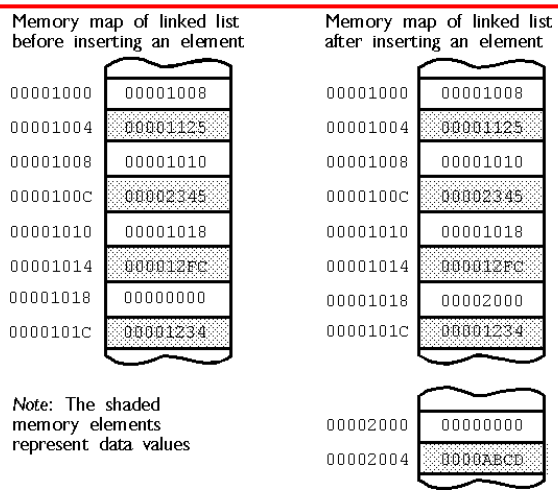

68000 Programming – Linked List Example (concluded)

- 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
```

68000 Programming – Memory Map for Linked List Example



2.9 Speed and Performance of Microprocessors – Comparison Difficulty

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

2.9 Speed and Performance of Micro-processors – 68000 vs. 68020

- Example: Interpret the high-level language construct

```
IF COUNT[CLASS[I]] <> 0 THEN …
```

68000 Version Clock Cycles	68000 Version Bus Cycles	68020 Version Clock Cycles	68020 Version Bus Cycles	Code
4	1	2	0	MOVE.W D1,D3
8	1	4	0	LSL.W #1,D3
12	2	6	0	LEA 0(A5,D3.W),A2
12	3	7	1	MOVE.WCLASS(A2),D3
8	1	4	0	LSL.W #1,D3
12	2	6	0	LEA 0(A5,D3.W),A2
12	3	7	1	TST.W COUNT(A2)
11	2	6	0	BEQ ELSE
79	15	42	2	

2.9 Speed and Performance of Micro-processors – more 68000 vs. 68020

MIPS = Million Instructions Per Second

- For the previous example, 68000:
  - Execution time = 6.32 μs
  - => 8 instructions / 6.32 μs = **1.27** MIPS
- 68020 using the same code
  - Execution time = 2.52 μs
  - => 8 instructions / 2.52 μs = **3.17** MIPS
- 68020 using special features
  - Execution time = 1.44 μs
  - => 3 instructions / 1.44 μs = **2.08** MIPS

```
MOVE.W (CLASS,A5,D1.W*2),D3
TST.W (COUNT,A5,D3.W*2)
BEQ ELSE
```

2.9 Speed and Performance of Microprocessors - Example

For the given assembly language program:

```
      LEA     TABLE,A0
      CLR.W   D1
LOOP  MOVE.B  D0,(A0)+
      ADDQ.W  #1,D1
      CMPI.W  #9,D1
      BNE     LOOP
```

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

b) What is the *average* CPI (number of clocks per instructions)?

c) What is the MIPS rate?

2.9 Speed and Performance of Microprocessors – Example

```
                          #of cycles
      LEA     TABLE,A0        8
      CLR.W   D1              4
LOOP  MOVE.B  D0,(A0)+        8
      ADDQ.W  #1,D1           4
      CMPI.W  #9,D1           8
      BNE     LOOP      10(taken)/8
```

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

- ***Cycle time***
- ***Clock cycles***
- ***Number of instructions***
- ***Execution time***

2.9 Speed and Performance of Microprocessors – Example

			# of cycles	
	LEA	TABLE,A0	8	→ Singe execution
	CLR.W	D1	4	
LOOP	MOVE.B	D0,(A0)+	8	→ Loop (9 iterations)
	ADDQ.W	#1,D1	4	
	CMPI.W	#9,D1	8	
	BNE	LOOP	10(taken)/8	

b) What is the average CPI (number of clocks per instructions)?

2.9 Speed and Performance of Microprocessors – Example

			# of cycles	
	LEA	TABLE,A0	8	→ Singe execution
	CLR.W	D1	4	
LOOP	MOVE.B	D0,(A0)+	8	→ Loop (9 iterations)
	ADDQ.W	#1,D1	4	
	CMPI.W	#9,D1	8	
	BNE	LOOP	10(taken)/8	

b) What is the MIPS rate?