68000 Stack-Related Instructions

PEA <EA> Push Effective Address

- Calculates an effective address <ea> and pushes it onto the stack pointed to by address register A7 (the stack pointer, SP).
- The difference between PEA and LEA
  - LEA loads an effective address in any address register.
  - PEA pushes an effective address onto the stack.
- PEA <EA> is equivalent to:

```
LEA        <EA>,Ai
MOVEA.L    Ai,-(A7)
```

Where Ai is an address register other than A7 (A0-A6)

The MOVE Multiple: MOVEM Instruction

- This instruction saves or restores multiple registers.
- Useful in subroutines to save the values of registers not used to pass parameters. MOVEM has two forms:

  MOVEM register_list,<ea>
  MOVEM <ea>,register_list

- No effect on CCR

Example: Saving/restoring registers to from memory

```
SUBR1   MOVEM   D0-D7/A0-A6,SAVEBLOCK        SAVE D0-D7/A0-A6
        . . .
        MOVEM   SAVEBLOCK,D0-D7/A0-A6        Restore D0-D7/A0-A6
        RTS
```

Example: Saving/restoring registers using the stack (preferred method).

```
SUBR1   MOVEM   D0-D7/A0-A6,-(SP)       Push D0-D7/A0-A6 onto the stack
         . . .
        MOVEM   (SP)+,D0-D7/A0-A6       Restore D0-D7/A0-A6  from the stack
        RTS
```

The Stack and Local Subroutine Variables: Stack Frames

- **In order for a subroutine to be *recursive* or *re-entrant* , the subroutine's local workspace must be *attached* to each use or call of the subroutine.**
- **A stack frame (SF) of size *d* bytes is defined as a region of temporary storage in memory of size *d* bytes at the top of the current stack.**
- **Upon creating a stack frame:**
  - **The frame pointer (FP) points to the bottom of the stack frame. Register A6 is normally used as the frame pointer.**
  - **The stack pointer, SP is updated to point to the top of the frame.**
- **In 68000 assembly, the LINK and UNLK instructions are used to facilitate the creation/destruction of local subroutine storage using stack frames.**

LINK Instruction

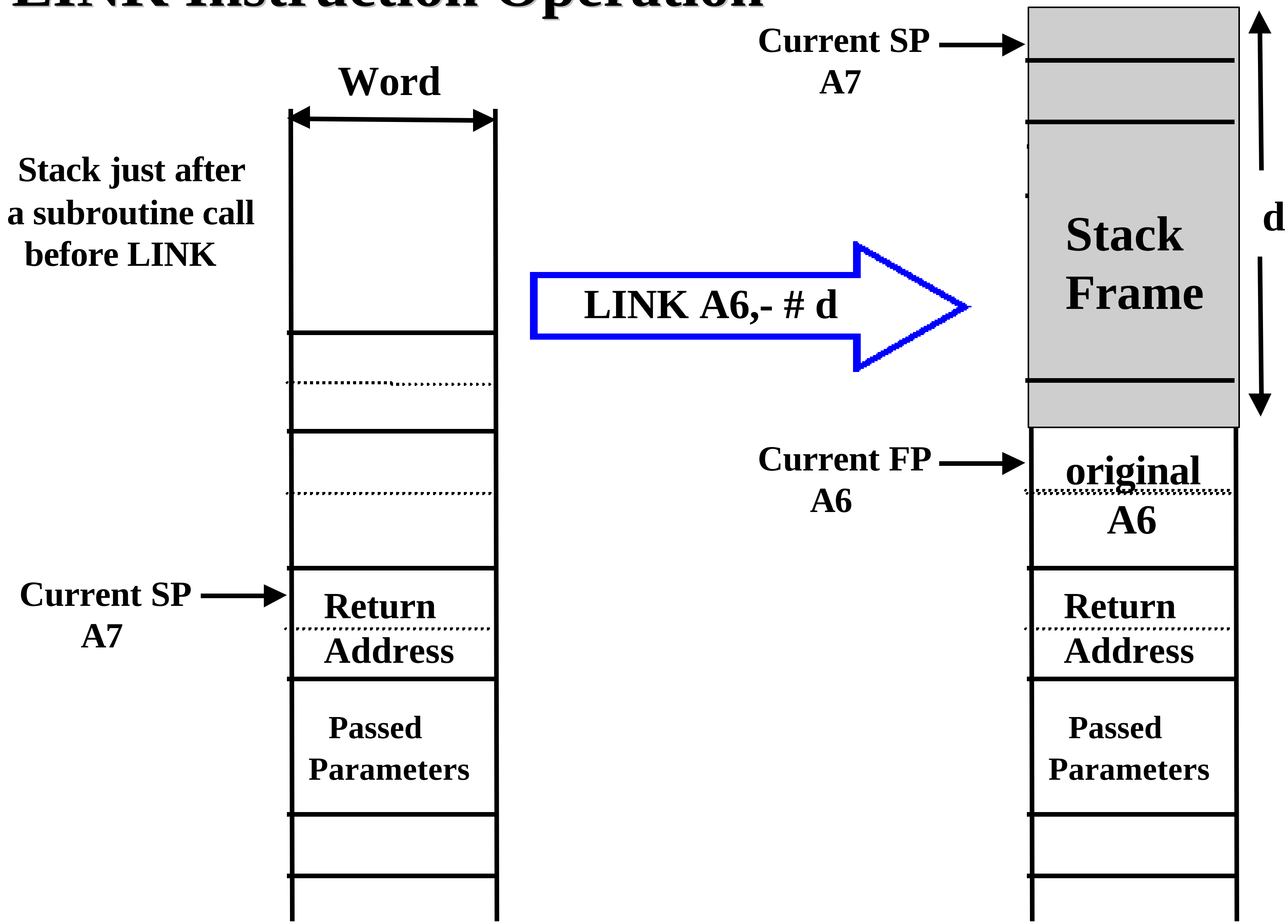
LINK An,-# d

- **Allocates or creates a frame in the stack for local use by the subroutine of size *d* bytes.**
- **An is an address register serving as the frame pointer (FP); A6 is used.**
- **Function:**
  - **Push the contents of address register An onto the stack. (includes pre-decrementing SP by 4).**
  - **Save the stack pointer in An (An points to bottom of frame)**
  - **Decrement the stack pointer by *d* (points to the top of the frame)**
  - **Similar in functionality to the following instruction sequence:**

```
MOVEA.L    A6,-(SP)
LEA        (SP),A6
LEA        -d(SP),SP
```

- **After creating the frame:**
  - **Passed parameters are accessed with a positive displacement with respect to FP, A6 i.e MOVE.W 8(A6),D0**
  - **Local temporary storage variables are accessed with negative displacement with respect to A6 i.e. MOVE.L D2,-10(A6)**

LINK Instruction Operation



UNLK UNLinK Instruction

UNLK An

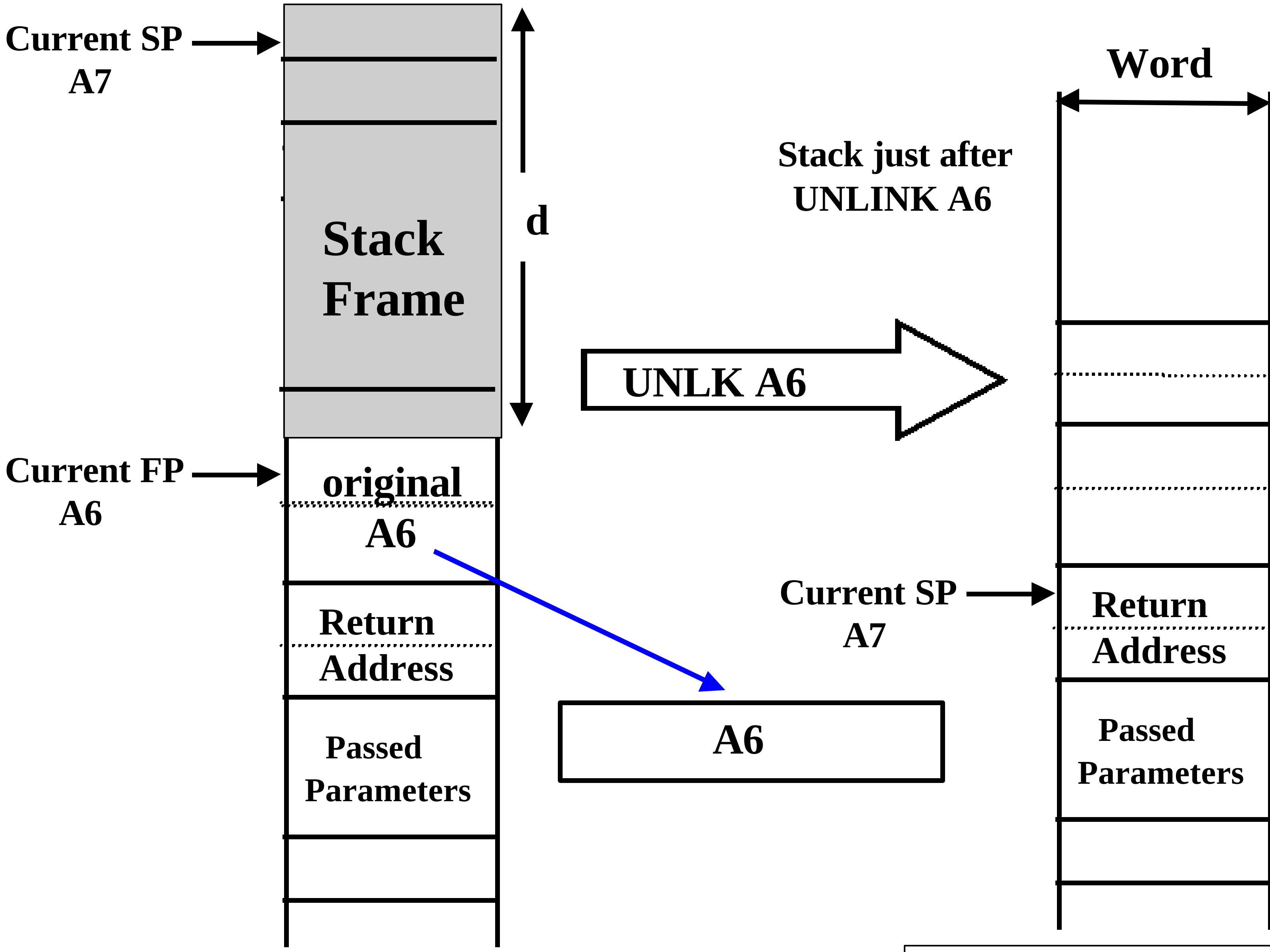
- Deallocates or destroys a stack frame. Where An is the address register used as frame pointer (FP); usually A6
- Function:
  - Restore the stack pointer to the value in address register An

    i.e SP = An or SP = SP + d
  - Restore register An by popping its value from the stack. (includes post-incrementing SP by 4).

Similar in functionality to the following instruction sequence:

```
LEA        d(SP),SP
MOVEA.L    (SP)+,An
```

UNLK Instruction Operation



Recursive Subroutine Calls Example

The purpose of this example is to examine how all parameters, local variables, return addresses, and frame pointers are stored on the stack when a main program calls a procedure "Process" as well as when the procedure calls itself again in a recursion. We assume the following:

- **The stack pointer initially has the value value $00000F00 just before Process is invoked (before any parameters are pushed onto the stack).**
- **Array "X", "Y", "Z" and "ALPHA" are passed by reference.**
- **Parameter "N" is passed by value (both ways - i.e. into the called procedure and also copied by value back into the calling routine).**
- **A6 is used as the frame pointer (assumed to have initial value $00002000).**
- **Procedure "Process" uses registers D0 - D4 as well as registers A0 - A4.**
- **Array X starts at location $1800, Y starts at $17F8, Z is at $17FC, ALPHA is at $17FD, and N is at $17FE.**

Recursive Subroutine Calls Example

Problem specification (continued):

{main routine}

X: array [0..30] of words

Y: longword

Z, ALPHA, N: byte

Process(var: X, var: Y, var: Z, var: ALPHA, N)

- **We are to show all the M68000 assembly language instructions necessary to pass these parameters as well as to copy the return value N into its regular storage location (off the stack) (at $17FE).**

Recursive Subroutine Calls Example

Problem specification (continued):

Procedure Process (A, B, C, D, E)

- **A: array [0..?] of words {passed by reference}**
- **B: longword {passed by reference}**
- **C, D: byte {passed by reference}**
- **E: byte {passed both ways by value}**
- **local variables -**
  - **T: longword**
  - **U: word**
  - **V: byte**

{ some place within the first invocation of "Process" it calls itself as follows:}

Process(var: A, var: T, var: C, var: V, E) {Note that some input parameters are passed through to the next iteration.}

Recursive Subroutine Calls Example Solution

The main program is assumed to allocate the original storage for:

```
            ORG  $17F8
Y           DS.L   1          This will resolve to address $000017F8
Z           DS.B   1          This will resolve to address $000017FC
ALPHA       DS.B   1          This will resolve to address $000017FD
N           DS.B   1          This will resolve to address $000017FE
*
            ORG  $1800
X           DS.W  31          an array of longwords 0..30
```

Recursive Subroutine Calls Example
Solution (Continued)

```
    ORG  $1000                  (assumed where main program starts - not critical)
*
*   In main program the procedure (subroutine) is called in HLL:
*
*    Process ( var:X, var:Y, var:Z, var:ALPHA, N) where N is the only one passed by value
*   The assembly language version/translation of this invocation is:
*
    CLR.W  D2                   zeroes out an entire word for pushing on stack
    MOVE.B N,D2                 copies value of byte N into lowest byte of D2
    MOVE.W D2,-(A7)             pushes that word containing value of N on stack
    PEA    ALPHA                pushes pointers to other arguments in reverse
    PEA    Z                    order
    PEA    Y
    PEA    X
    JSR    Process              actually call the subroutine here
    MOVE.B 17(A7),N             copy returned value back into N
    ADDA.L #18,A7               fix up stack from all parameters pushed for
*                               subroutine call.
```

Recursive Subroutine Calls Example

Solution (Continued)   Stack Utilization Diagram

```
0E5E |  not used     |
0E60 |               |
     |               |
0E64 |  not used     |
     |               |
0E68 |  not used     |
     |               |
0E6C |  D0 (high) 2  |
     |  D0 (low)     |
0E70 |  D1    2      |
     |               |
0E74 |  D2    2      |
     |               |
0E78 |  D3    2      |
     |               |
0E7C |  D4    2      |
     |               |
0E80 |  A0    2      |
     |               |
0E84 |  A1    2      |
     |               |
0E88 |  A2    2      |
     |               |
0E8C |  A3    2      |
     |               |
0E90 |  A4    2      |
     |               |
```

```
     0E94 | local 2 "T"  |
          | (longword)   |
     0E98 | local 2 "U"  |
     0E9A | - -  | "V" 2 |
**   0E9C | link reg val |
          | = $00000EE6  |
     0EA0 | return addr  |
          | into Process |
     0EA4 | Addr of "X"  |
          | ="A" in Proc |
     0EA8 | Addr of "T"1 |
          | = $00000EDE  |
     0EAC | Addr of "Z"  |
          | equiv "C" 1  |
     0EB0 | Addr of "V"1 |
          | = $00000EE5  |
     0EB4 | $00   | "E"2 |
     0EB6 |  D0 (high) 1 |
          |  D0 (low)    |
     0EBA |  D1    1     |
          |              |
     0EBE |  D2    1     |
          |              |
     0EC2 |  D3    1     |
          |              |
     0EC6 |  D4    1     |
          |              |
```

```
 0ECA |  A0          |
      |              |
 0ECE |  A1 (high)   |
      |  A1 (low)    |
 0ED2 |  A2          |
      |              |
 0ED6 |  A3          |
      |              |
 0EDA |  A4          |
      |              |
 0EDE | local 1 "T"  |
      | (longword)   |
 0EE2 | local 1 "U"  |
 0EE4 | - -  | "V" 1 |
*0EE6 | orig linkreg |
      | = $00002000  |
 0EEA | return addr  |
      | into main pr |
 0EEE | Addr of "X"  |
      | = $00001800  |
 0EF2 | Addr of "Y"  |
      | = $000017F8  |
 0EF6 | Addr of "Z"  |
      | = $000017FC  |
 0EFA | Addr "ALPHA" |
      | = $000017FD  |
 0EFE | $00  |"N"val |
```

*** indicates the value of link register A6 during first call of Process**

**** indicates the value of link register A6 during the second call to Process**

Recursive Subroutine Calls Example Solution (Continued) procedure Process

- **The coding of procedure Process would be something like this:**

Procedure Process (var:A, var:B, var:C, var:D, E)

```
* where A:  is an array of words [0.. ?]   passed by reference
*      B:   longword passed by reference
*      C, D:   byte passed by reference
*      E:  byte passed by value (in BOTH directions)
* and local variables:
*      T:  longword
*      U:  word
 *     V:  byte
Aptr    equ    8     displacements for finding pass by reference
Bptr    equ    12    addresses from the frame pointer: A6
Cptr    equ    16
Dptr    equ    20
E       equ    25    this one is a byte which is passed by value
V       equ    -1
U       equ    -4
T       equ    -8
```

Recursive Subroutine Calls Example
Solution (Continued) procedure Process

```
* The start of the code of Process looks like this:
*
Process   LINK      A6,#-8
          MOVEM.L  D0-D4/A0-A4,-(A7)        save registers as required
*
* The invocation of Process from within Process:
*
* Process ( A, T, C, V, E)
*
          CLR.W     D0
          MOVE.B    E(A6),D0           note how we access "E" - we could have
          MOVE.W    D0,-(A7)           modified "E" before sending it
          PEA       V(A6)              this is basically how we can use "V" too
          MOVE.L    Cptr(A6),-(A7)     we push the pointer to "Z" on stack
          PEA       T(A6),A0           push pointer to local variable "T" on stack
          MOVE.L    Aptr(A6),-(A7)     push pointer to "X" ("A" in Process)
          JSR       Process
          MOVE.B    17(A7),E(A6)       copy return value of "E" to local copy
          ADDA.L    #18,A7             fix up stack from all parameters pushed
*
```

Recursive Subroutine Calls Example

Solution (Continued) procedure Process

```
*   This is how we'd access some of the variables in the subroutine:
*
        MOVEA.L   Aptr(A6),A0      This is how we'd copy the first array
        MOVE.L    (A0),U(A6)       element of X ("A" in procedure) into "U"
*
        MOVEA.L  Bptr(A6),A1       This is how we'd copy input parameter "B"
        MOVE.W   (A1),T(A6)        into local word "T"
*
        MOVEA.L  Cptr(A6),A2       This is how we actually reference "C"
        MOVE.B   (A2),D1
*
        MOVEA.L  Dptr(A6),A3       This is how we could access/change
        CLR.B      (A3)            "D" in procedure = "ALPHA" in main
*
*   Before leaving the procedure we'd need to restore registers and destroy stack frame:
*
        MOVEM.L (A7)+,D0-D4/A0-A4
        UNLK    A6
        RTS
```

68000 Binary Coded Decimal (BCD) Arithmetic

- Binary Coded Decimal (BCD) is a way to store decimal numbers in binary. This number representation uses 4 bits to store each digit from 0 to 9. For example:

$$1998_{10} = 0001\ 1001\ 1001\ 1000 \quad \text{in BCD}$$

- BCD wastes storage space since 4 bits are used to store 10 combinations rather than the maximum possible 16.
- BCD is often used in business applications and calculators.
- The 68000 instruction set includes three instructions that offer some support for BCD arithmetic:
  - ABCD Add BCD with extend
  - SBCD Subtract BCD with extend
  - NBCD Negate BCD
- BCD instructions use and affect the X-bit because they are intended to be used in chained calculations where arithmetic is done on strings of BCD digits.
  - For addition: the X-bit records the carry
  - For subtraction: the X-bit records the borrow

ABCD | **Add Decimal with Extend (M68000 Family)** | **ABCD**

ABCD — Add Decimal with Extend (M68000 Family)

Operation: Source10 + Destination10 + X → Destination

Assembler Syntax:
ABCD Dy,Dx
ABCD – (Ay), – (Ax)

Attributes: Size = (Byte)

Description: Adds the source operand to the destination operand along with the extend bit, and stores the result in the destination location. The addition is performed using binary-coded decimal arithmetic. The operands, which are packed binary-coded decimal numbers, can be addressed in two different ways:

1. Data Register to Data Register: The operands are contained in the data registers specified in the instruction.
2. Memory to Memory: The operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

This operation is a byte operation only.

ABCD | Add Decimal with Extend (M68000 Family) | ABCD

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X — Set the same as the carry bit.
N — Undefined.
Z — Cleared if the result is nonzero; unchanged otherwise.
V — Undefined.
C — Set if a decimal carry was generated; cleared otherwise.

NOTE

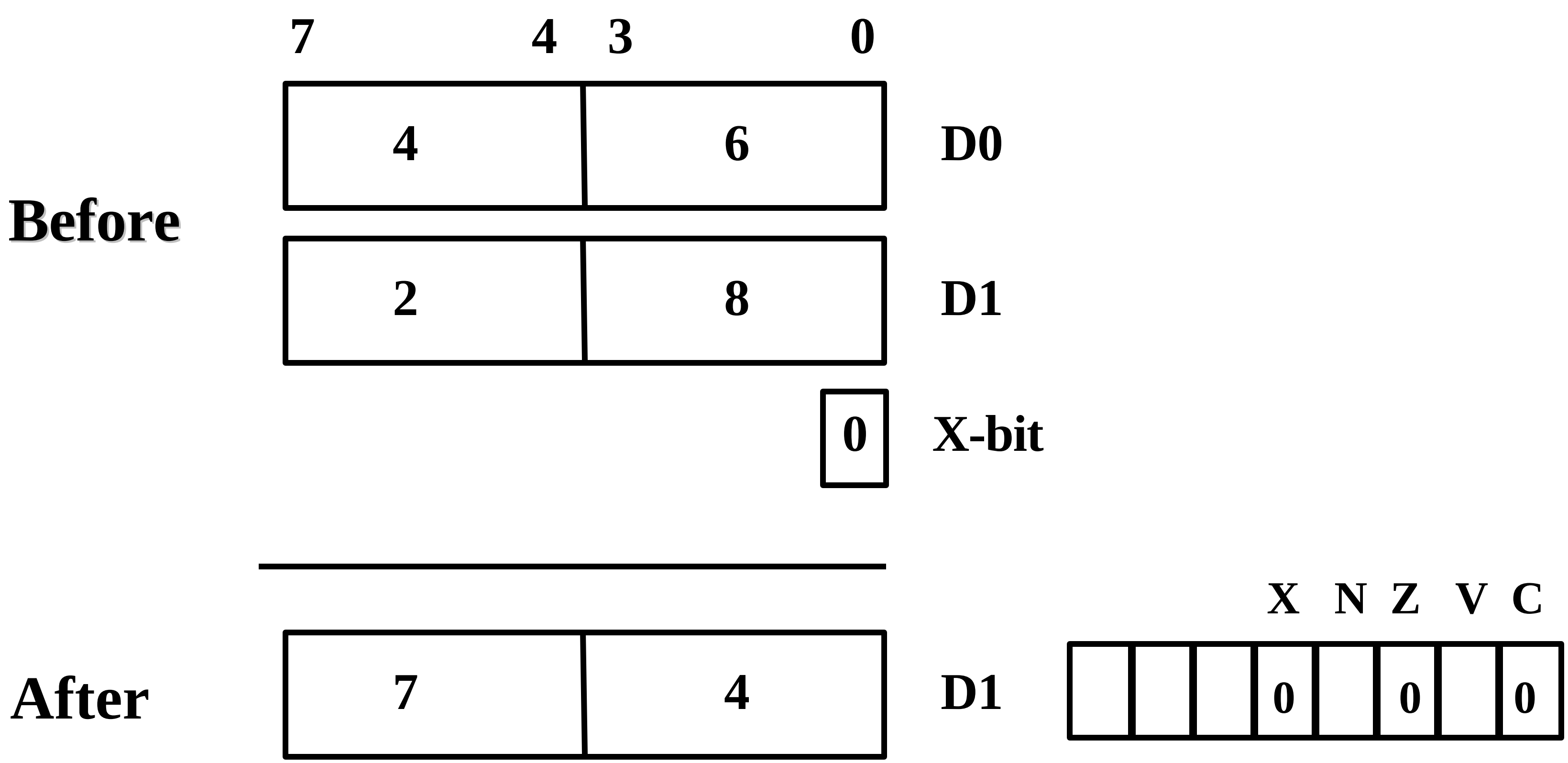
Normally, the Z condition code bit is set via programming before the start of an operation. This allows successful tests for zero results upon completion of multiple-precision operations.

Effect of ABCD

When X = 0 initially

ABCD D0,D1

Add D0 to D1 with the X-bit

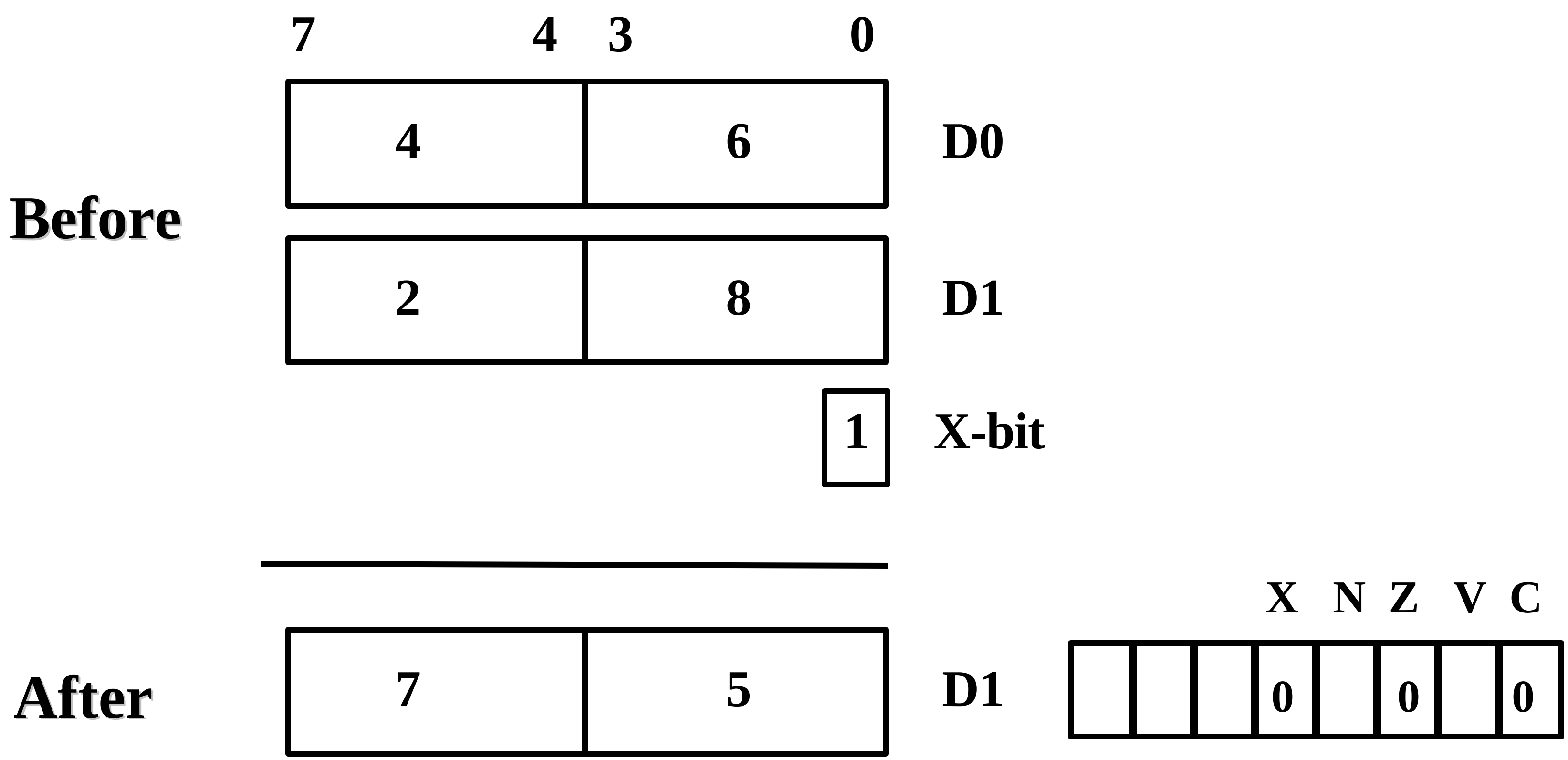


Effect of ABCD

When X = 1 initially

ABCD D0,D1

Add D0 to D1 with the X-bit



SBCD — Subtract Decimal with Extend (M68000 Family) — SBCD

Operation: Destination10 – Source10 – X → Destination

Assembler Syntax:
SBCD Dx,Dy
SBCD – (Ax), – (Ay)

Attributes: Size = (Byte)

Description: Subtracts the source operand and the extend bit from the destination operand and stores the result in the destination location. The subtraction is performed using binary-coded decimal arithmetic; the operands are packed binary-coded decimal numbers. The instruction has two modes:

1. Data register to data register—the data registers specified in the instruction contain the operands.
2. Memory to memory—the address registers specified in the instruction access the operands from memory using the predecrement addressing mode.

This operation is a byte operation only.

SBCD Subtract Decimal with Extend (M68000 Family) SBCD

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X — Set the same as the carry bit.
N — Undefined.
Z — Cleared if the result is nonzero; unchanged otherwise.
V — Undefined.
C — Set if a borrow (decimal) is generated; cleared otherwise.

NOTE

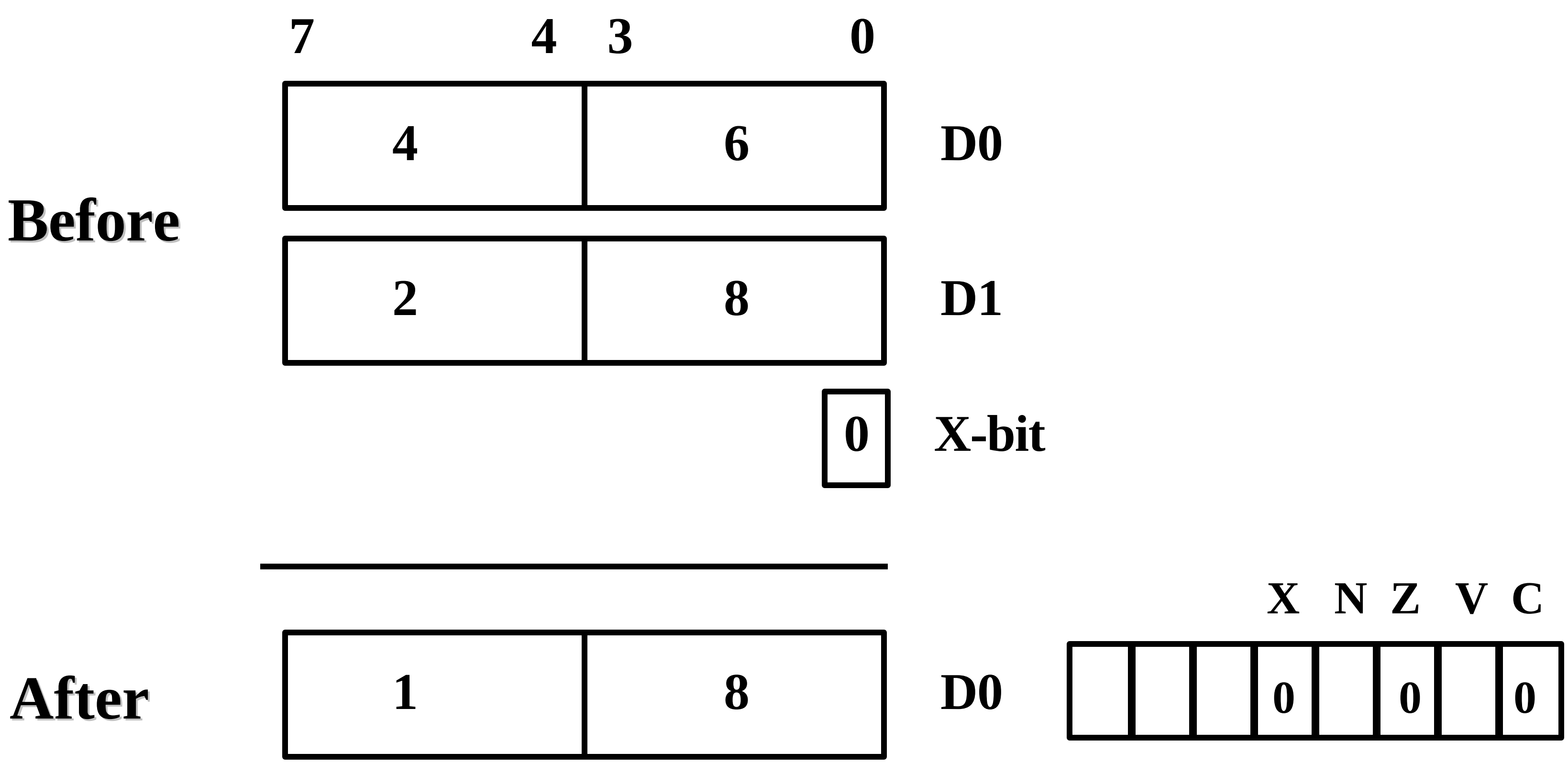
Normally the Z condition code bit is set via programming before the start of an operation. This allows successful tests for zero results upon completion of multiple-precision operations.

Effect of SBCD

When X = 0 initially

SBCD D1,D0

Subtract D1 from D0 with the X-bit



EECC250 - Shaaban

Effect of SBCD

When X = 1 initially

SBCD D1,D0

Subtract D1 from D0 with the X-bit

Before

7 4	3 0	
4	6	D0
2	8	D1

1	X-bit

After

1	7	D0

			X	N	Z	V	C
			0		0		0

NBCD — Negate Decimal with Extend (M68000 Family) — NBCD

Operation: $0 - \text{Destination}_{10} - X \rightarrow \text{Destination}$

Assembler Syntax: NBCD < ea >

Attributes: Size = (Byte)

Description: Subtracts the destination operand and the extend bit from zero. The operation is performed using binary-coded decimal arithmetic. The packed binary-coded decimal result is saved in the destination location. This instruction produces the tens complement of the destination if the extend bit is zero or the nines complement if the extend bit is one. This is a byte operation only.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X — Set the same as the carry bit.
N — Undefined.
Z — Cleared if the result is nonzero; unchanged otherwise.
V — Undefined.
C — Set if a decimal borrow occurs; cleared otherwise.

Effect of NBCD

When X = 0 initially

NBCD D0

Subtract D0 from 0 with the X-bit

Before

7 ... 4	3 ... 0	
0	0	
2	8	D0

X-bit: 0

After

7 ... 4	3 ... 0	
7	2	D0

			X	N	Z	V	C
			1		0		1

Effect of NBCD

When X = 1 initially

NBCD D0

Subtract D0 from 0 with the X-bit

	7 4	3 0	
Before	0	0	
	2	8	D0
		1	X-bit
After	7	1	D0

			X	N	Z	V	C
			1		0		1

BCD Addition Example

- Two BCD strings each with 12 BCD digits (six bytes) and stored in memory starting at locations: String1, String2, are to be added together with the result to be stored in memory starting at String2

```
        ORG       $1000
ADDBCD  MOVE.W    #5,D0           Loop counter, six bytes to be added
        ANDI      #$EF,CCR        Clear X-bit in CCR
        LEA       String1+6,A0    A0 points at end of source string +1
        LEA       String2+6,A1    A0 points at end of destination string +1
LOOP    ABCD      -(A0),-(A1)     Add pair of digits with carry-in
        DBRA      D0,LOOP         Repeat until 12 digits are added
        RTS
         .
         .
String1 DS.B      6
String2 DS.B      6
```

DBRA used here because it does not affect the X-bit needed in BCD arithmetic

68000 Multiple-Precision Arithmetic

- **For numerical values, *precision* refers to the number of significant digits in the numerical value.**
  - →**If more precision is needed in a numerical value, more significant digits must be used to yield a more precise result.**
- **The maximum single-precision operand length supported by the 68000 is 32 bits. Thus, values with greater length cannot be handled as a single arithmetic operand by the CPU.**
- **To extend the precision, several 32-bit operands can be used and considered mathematically as a single value.**
- **The 68000 offers three special instructions to facilitate addition, subtraction, and negation of multiple-precision integers:**
  - **ADDX ADD with eXtend**
  - **SUBX SUBtract with eXtend**
  - **NEGX NEGate with eXtend**

ADDX — Add Extended (M68000 Family) — ADDX

Operation: Source + Destination + X → Destination

Assembler Syntax:
ADDX Dy,Dx
ADDX – (Ay), – (Ax)

Attributes: Size = (Byte, Word, Long)

Description: Adds the source operand and the extend bit to the destination operand and stores the result in the destination location.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X — Set the same as the carry bit.
N — Set if the result is negative; cleared otherwise.
Z — Cleared if the result is nonzero; unchanged otherwise.
V — Set if an overflow occurs; cleared otherwise.
C — Set if a carry is generated; cleared otherwise.

SUBX — Subtract with Extend (M68000 Family) — SUBX

Operation: Destination – Source – X $\rightarrow$ Destination

Assembler Syntax:
SUBX Dx,Dy
SUBX – (Ax), – (Ay)

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the source operand and the extend bit from the destination operand and stores the result in the destination

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X — Set to the value of the carry bit.
N — Set if the result is negative; cleared otherwise.
Z — Cleared if the result is nonzero; unchanged otherwise.
V — Set if an overflow occurs; cleared otherwise.
C — Set if a borrow occurs; cleared otherwise.

NEGX — Negate with Extend (M68000 Family) — NEGX

Operation: 0 – Destination – X → Destination

Assembler Syntax: NEGX < ea >

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the destination operand and the extend bit from zero. Stores the result in the destination location. The size of the operation is specified as byte, word, or long.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X — Set the same as the carry bit.
N — Set if the result is negative; cleared otherwise.
Z — Cleared if the result is nonzero; unchanged otherwise.
V — Set if an overflow occurs; cleared otherwise.
C — Set if a borrow occurs; cleared otherwise.

Multiple-Precision Addition Example

- **Two unsigned binary numbers each with 128 bits (16 bytes) and stored in memory starting at locations Num1, Num2 are to be added together with the result to be stored in memory starting at Num2**

```
          ORG      $1000
MPADD     MOVE.W   #3,D0          Four long words to be added
          ANDI     #$EF,CCR       Clear X-bit  in CCR
          LEA      Num1,A0        A0 points at start of source
          ADDA     #16,A0         A0 points to end of source  + 1
          LEA      Num2,A1        A1 points at start of destination
          ADDA     #16,A1         A1 points to end of destination  + 1
LOOP      ADDX.L   -(A0),-(A1)    Add pair of long words with carry-in
          DBRA     D0,LOOP        Repeat until 4 long words are added
          RTS
           .
           .
Num1      DS.L     4
Num2      DS.L     4
```

DBRA is used here because it does not affect the X-bit needed in multiple-precision arithmetic

Estimation of Assembly Programs Execution Time

- For a CPU running at a constant clock rate:

  clock rate = 1 / clock cycle time

- Every machine or assembly instruction takes one or more clock cycles to complete.
- The total time an assembly program requires to run is given by:

Execution time = Total number of cycles X Clock cycle time
= Instruction count X cycles per instruction X clock cycle time
= Instruction count X cycles per instruction / clock rate

Example:

For a CPU running at 8MHZ is executing a program with a total of 100 000 instructions. Assuming that each instruction takes 10 clock cycles to complete:

Execution time = 100 000 X 10 / 8 000 000 = 0.125 seconds

68000 Cycles For MOVE Instructions

Operand Size	Addressing Mode								
.b.w/.l	dn	an	(an)	(an)+	-(an)	d(an)	d(an,dn)	abs.s	abs.l
dn	4/4	4/4	8/12	8/12	8/14	12/16	14/18	12/16	16/20
an	4/4	4/4	8/12	8/12	8/14	12/16	14/18	12/16	16/20
(an)	8/12	8/12	12/20	12/20	12/20	16/24	18/26	16/24	20/28
(an)+	8/12	8/12	12/20	12/20	12/20	16/24	18/26	16/24	20/28
-(an)	10/14	10/14	14/22	14/22	14/22	18/26	20/28	18/26	22/30
d(an)	12/16	12/16	16/24	16/24	16/24	20/28	22/30	20/28	24/32
d(an,dn)	14/18	14/18	18/26	18/26	18/26	22/30	24/32	22/30	26/34
abs.s	12/16	12/16	16/24	16/24	16/24	20/28	22/30	20/28	24/32
abs.l	16/20	16/20	20/28	20/28	20/28	24/32	26/34	24/32	28/36
d(pc)	12/16	12/16	16/24	16/24	16/24	20/28	22/30	20/28	24/32
d(pc,dn)	14/18	14/18	18/26	18/26	18/26	22/30	24/32	22/30	26/34
Immediate	8/12	8/12	12/20	12/20	12/20	16/24	18/26	16/24	20/28

Clock Cycles

Time to Calculate Effective Addresses

	Addressing Mode				
	(an)	(an)+	-(an)	d(an)	d(an,dn)
.b.w/.l	4/8	4/8	6/10	8/12	10/14

Operand Size

	Addressing Mode				
	abs.s	abs.l	d(pc)	d(pc,dn)	Imm
.b.w/.l	8/12	12/16	8/12	10/14	4/8

Operand Size

The time taken to calculate the effective address must be added to instructions that affect a memory address.

68000 Cycles For Standard Instructions

Operand Size .b.w/.l	Addressing Mode ea,an	ea,dn	dn,mem
add	8/6(8)	4/6(8)	8/12
and	-	4/6(8)	8/12
cmp	6/6	4/6	-
divs	-	158max	-
divu	-	140max	-
eor	-	4/8	8/12
muls	-	70max	-
mulu	-	70max	-
or	-	4/6(8)	8/12
sub	8/6(8)	4/6(8)	8/12

Clock Cycles

(8) time if effective address is direct

Add effective address times from above for mem addresses

Cycles For Immediate Instructions

Operand Size .b.w/.l	Addressing Mode #,dn	#,an	#,mem
addi	8/16	-	12/20
addq	4/8	8/8	8/12
andi	8/16	-	12/20
cmpi	8/14	8/14	8/12
eori	8/16	-	12/20
moveq	4	-	-
ori	8/16	-	12/20
subi	8/16	-	12/20
subq	4/8	8/8	8/12

Clock Cycles

Moveq.l only

nbcd+tas.b only

scc false/true

Add effective address times from above for mem addresses

Cycles for Single-Operand Instructions

Operand Size / **Addressing Mode**

.b.w/.l	#,dn	#,an	#,mem
clr	4/6	4/6	8/12
nbcd	6	6	8
neg	4/6	4/6	8/12
negx	4/6	4/6	8/12
not	4/6	4/6	8/12
scc	4/6	4/6	8/8
tas	4	4	10
tst	4/4	4/4	4/4

Clock Cycles

Add effective address times from above for mem addresses

Cycles for Shift/Rotate Instructions

Operand Size .b.w/.l	Addressing Mode dn	an	mem
asr,asl	6/8	6/8	8
lsr,lsl	6/8	6/8	8
ror,rol	6/8	6/8	8
roxr,roxl	6/8	6/8	8

Clock Cycles

Memory is byte only
For register add 2x the shift count

Misc. Instructions

Addressing Mode

	(an)	(an)+	-(an)	d(an)	d(an ,dn)	abs.s	abs.l	d(pc)	d(pc ,dn)
jmp	8	-	-	10	14	10	12	10	14
jsr	16	-	-	18	22	18	20	18	22
lea	4	-	-	8	12	8	12	8	12
pea	12	-	-	16	20	16	20	16	20
movem	t=4								
m>r	12	12	-	16	18	16	20	16	18
movem	t=5								
r>m	8	-	8	12	14	12	16	-	-
movem	add t x number of registers for .w								
movem	add 2t x number of registers for .l								

Clock Cycles

Cycles for Bit Manipulation Instructions

Operand Size .b/.l	Addressing Mode register .l only	memory .b only
bchg	8/12	8/12
bclr	10/14	8/12
bset	8/12	8/12
btst	6/10	4/8

Clock Cycles

Cycles To Process Exceptions

Address Error	50
Bus Error	50
Interrupt	44
Illegal Instr.	34
Privilege Viol.	34
Trace	34

Cycles for Other Instructions

Operand Size .b.w/.l	dn,dn	m,m	Addressing Mode
addx	4/8	18/30	
cmpm	-	12/20	
subx	4/8	18/30	
abcd	6	18	.b only
sbcd	6	18	.b only
Bcc	.b/.w	10/10	8/12
bra	.b/.w	10/10	-
bsr	.b/.w	18/18	-
DBcc	t/f	10	12/14
chk	-	40 max	8
trap	-	34	-
trapv	-	34	4

Clock Cycles

Add effective address times from above for mem addresses

Cycles for Other Instructions

reg<>mem

movep .w/.l 16/24

	Reg	Mem
andi to ccr	20	-
andi to sr	20	-
eori to ccr	20	-
eori to sr	20	-
exg	6	-
ext	4	-
link	18	-
move to ccr	12	12
move to sr	12	12
move from sr	6	8
move to usp	4	-

	Reg
move from usp	4
nop	4
ori to ccr	20
ori to sr	20
reset	132
rte	20
rtr	20
rts	16
stop	4
swap	4
unlk	12

Addressing Mode

Clock Cycles

Timing Example 1

	Instruction		Clock Cycles
RANDOM	ADDI.B	#17,D0	8
	LSL.B	#3,D0	12
	NOT.B	D0	4
	RTS		16
	Total Cycles needed:		40 cycles

For a 68000 running at 8MHZ:

Clock cycle = 125 nsec

Execution time = 40 X 125 nsec = 5 μs = 5 x 10^{-6} second

Timing Example 2

	Instruction		Clock Cycles Overhead	Clock Cycles Loop
	MOVE.B	#255,D0	8	
READ	ADD.W	(A0)+,D1		8
	SUBQ.B	#1,D0		4
	BNE	READ		10

Total Cycles Needed = 8 + 255 (8 + 4 + 10)

= 8 + 255 x 22

= 5618 cycles

Execution time for 8MHZ 68000 = 5618 x 125 nsec

= 0.00070225 Seconds = .702 msec

Timing Example 3

- TOBIN converts a four-digit BCD number in the lower word of D0 into a binary number returned in D2

Label	Instructions		Clock Cycles overhead	Clock Cycles outer loop	Clock Cycles inner loop
TOBIN	CLR.L	D2	6		
	MOVEQ	#3,D6	4		
NEXTDIGIT	MOVEQ	#3,D5		4	
	CLR.W	D1		4	
GETNUM	LSL.W	#1,D0			8
	ROXL.W	#1,D1			8
	DBRA	D5,GETNUM			10
	MULU	#10,D2		42	
	ADD.W	D1,D2		4	
	DBRA	D6,NEXTDIGIT		10	
	RTS		16		

Total Clock cycles = overhead + ((inner loop cycles x 4) + outer loop cycles) x 4

= 26 + ((26 x 4) + 64) x 4

= 26 + 168 x 4 = 698 cycles

= 698 x 125 nsec = 87.25 ms

or over 11 400 BCD numbers converted to binary every second.

Representation of Floating Point Numbers in

Single Precision *IEEE 754 Standard*

$$\text{Value} = N = (-1)^{S} \times 2^{E-127} \times (1.M)$$

	1	8	23
sign	S	E	M

$0 < E < 255$
Actual exponent is:
$e = E - 127$

exponent:
excess 127
binary integer
added

mantissa:
sign + magnitude, normalized
binary significand with
a hidden integer bit: 1.M

Example: 0 = 0 00000000 0 . . . 0 -1.5 = 1 01111111 10 . . . 0

Magnitude of numbers that
can be represented is in the range: $2^{-126}(1.0)$ to $2^{127}(2 - 2^{-23})$

Which is approximately: 1.8×10^{-38} to 3.40×10^{38}

Floating Point Conversion Example

- The decimal number $.75_{10}$ is to be represented in the *IEEE 754* 32-bit single precision format:

$$.75_{10} = 0.11_2 \quad \text{(converted to a binary number)}$$

$$= 1.1 \times 2^{-1} \quad \text{(normalized a binary number)}$$

Hidden

- The mantissa is positive so the sign S is given by:

$\boxed{S = 0}$

- The biased exponent E is given by $E = e + 127$

$$E = -1 + 127 = 126_{10} = \boxed{01111110_2}$$

- Fractional part of mantissa M:

$$M = \boxed{.10000000000000000000000} \text{ (in 23 bits)}$$

The *IEEE 754* single precision representation is given by:

S	E	M
0	01111110	10000000000000000000000
1 bit	8 bits	23 bits

Floating Point Conversion Example

- The decimal number -2345.125_{10} is to be represented in the *IEEE 754* 32-bit single precision format:

$-2345.125_{10} = -100100101001.001_2$ (converted to binary)

$= -1.00100101001001 \times 2^{11}$ (normalized binary)

Hidden

- The mantissa is negative so the sign S is given by:

$\boxed{S = 1}$

- The biased exponent E is given by E = e + 127

$E = 11 + 127 = 138_{10} = \boxed{10001010_2}$

- Fractional part of mantissa M:

M = $\boxed{.00100101001001000000000}$ (in 23 bits)

The *IEEE 754* single precision representation is given by:

1	10001010	00100101001001000000000

S	E	M
1 bit	8 bits	23 bits

Basic Floating Point Addition Algorithm

Assuming that the operands are already in the IEEE 754 format, performing floating point addition: Result = X + Y = $(Xm \times 2^{Xe}) + (Ym \times 2^{Ye})$

involves the following steps:

(1) Align binary point:

- Initial result exponent: the larger of Xe, Ye
- Compute exponent difference: Ye - Xe
- If Ye > Xe Right shift Xm that many positions to form $Xm\ 2^{Xe-Ye}$
- If Xe > Ye Right shift Ym that many positions to form $Ym\ 2^{Ye-Xe}$

(2) Compute sum of aligned *mantissas*:

i.e $Xm2^{Xe-Ye} + Ym$ or $Xm + Xm2^{Ye-Xe}$

(3) If normalization of result is needed, then a normalization step follows:

- Left shift result, decrement result exponent (e.g., if result is 0.001xx…) or
- Right shift result, increment result exponent (e.g., if result is 10.1xx…)

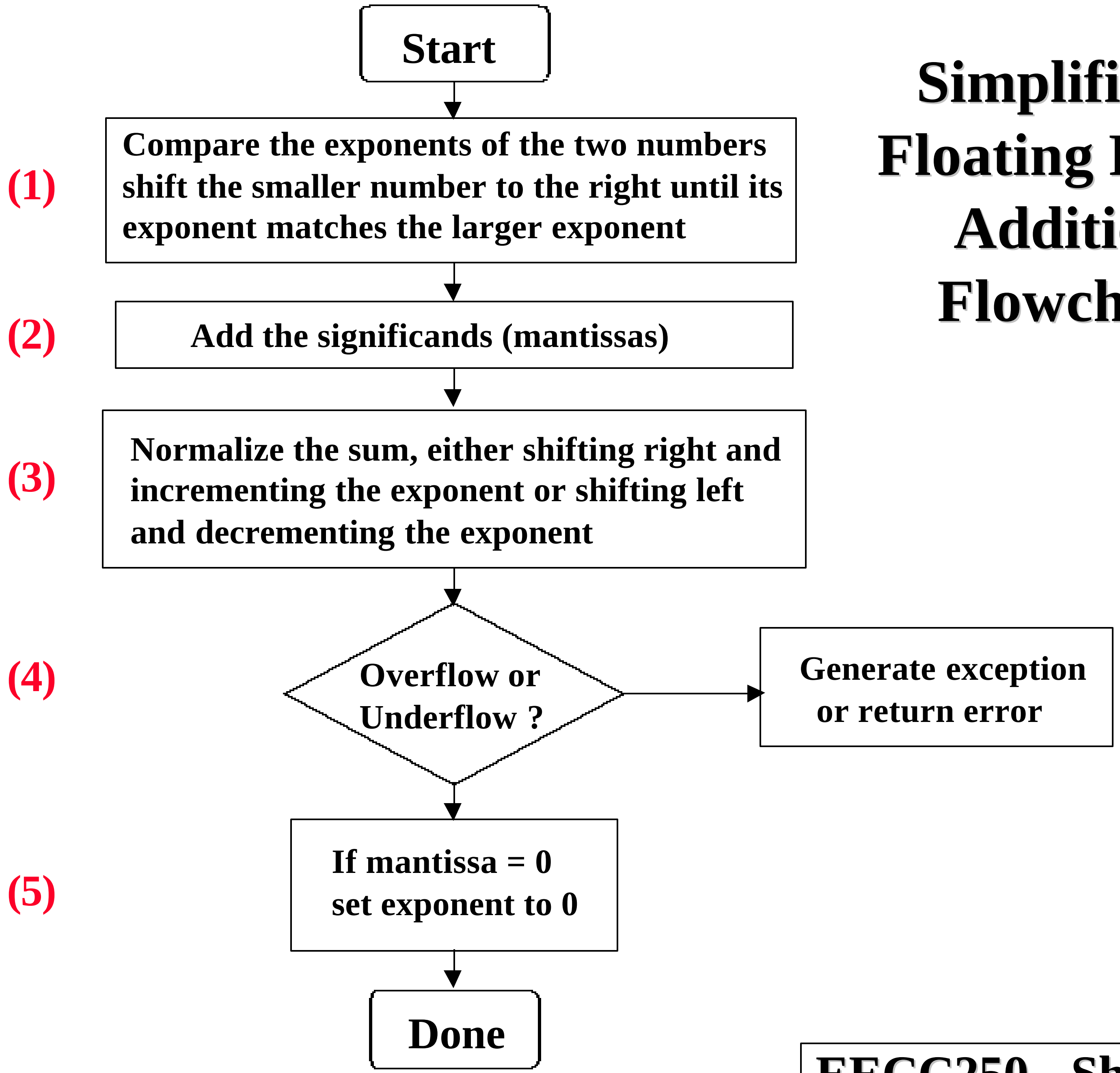
Continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard)

(4) Check result exponent:

- If larger than maximum exponent allowed return exponent overflow
- If smaller than minimum exponent allowed return exponent underflow

(5) If result mantissa is 0, may need to set the exponent to zero by a special step to return a proper zero.

Simplified Floating Point Addition Flowchart



Floating Point Addition Example

- **Add the following two numbers represented in the *IEEE 754* single precision format: X = 2345.125_{10} represented as:**

0	10001010	00100101001001000000000

to Y = $.75_{10}$ represented as:

0	01111110	10000000000000000000000

(1) Align binary point:

- **Xe > Ye initial result exponent = Ye = 10001010 = 138_{10}**
- **Xe - Ye = 10001010 - 01111110 = 00000110 = 12_{10}**
- **Shift Ym 12_{10} postions to the right to form**

$$Ym\ 2^{Ye-Xe} = Ym\ 2^{-12} = 0.00000000000110000000000$$

(2) Add mantissas:

$$Xm + Ym\ 2^{-12} = 1.00100101001001000000000$$

$$+\ 0.00000000000110000000000 =$$

$$1.\boxed{00100101001111000000000}$$

(3) Normailzed? Yes

(4) Overflow? No. Underflow? No (5) zero result? No

Result

0	10001010	00100101001111000000000

IEEE 754 Single precision Addition Notes

- **If the exponents differ by more than 24, the smaller number will be shifted right entirely out of the mantissa field, producing a zero mantissa.**
  - **The sum will then equal the larger number.**
  - **Such truncation errors occur when the numbers differ by a factor of more than 2^{24} , which is approximately 1.6×10^7 .**
  - **Thus, the precision of IEEE single precision floating point arithmetic is approximately 7 decimal digits.**
- **Negative mantissas are handled by first converting to 2's complement and then performing the addition.**
  - **After the addition is performed, the result is converted back to sign-magnitude form.**
- **When adding numbers of opposite sign, cancellation may occur, resulting in a sum which is arbitrarily small, or even zero if the numbers are equal in magnitude.**
  - **Normalization in this case may require shifting by the total number of bits in the mantissa, resulting in a large loss of accuracy.**
- **Floating point subtraction is achieved simply by inverting the sign bit and performing addition of signed mantissas as outlined above.**

Assembly Language Macros

- Most assemblers include support for macros. The term macro refers to a word that stands for an entire group of instructions.
- Using macros in an assembly program involves two steps:

1. Defining a macro:

   The definition of a macro consists of three parts: the header, body, and terminator:

<label>	MACRO	The header
. . . .		The body: instructions to be executed
ENDM		The terminator

2. Invoking a macro by using its given <label> on a separate line followed by the list of parameters used if any:

```
<label>   [parameter list]
```

Differences Between Macros and Subroutines

- **Both permit a group of instructions to be defined as a single entity with a unique given label or name called up when needed.**
- **A subroutine is called by the BSR or JSR instructions, while a macro is called by simply using its name.**
- **Macros are not a substitute for subroutines:**
  - **Since the macro is substituted with the code which constitutes the body of the macro into the code, very long macros that are used many times in a program will result in an enormous expansion of the code size.**
  - **In this case, a subroutine would be a better choice, since the code in the body of the subroutine is not inserted into source code many when called.**
- **Support for subroutines is provided by the CPU --here, the 68000-- as part of the instruction set, while support for macros is part of the assembler (similar to assembler directives).**

A Macro Example

Defining the macro:

```
AddMul   MACRO                     Macro definition
         ADD.B     #7,D0           D0 = D0 + 7
         AND.W     #00FF,D0        Mask D0 to a byte
         MULU      #12,D0          D0 = D0 x 12
         ENDM                      End of macro  def.
```

Invoking the macro:

```
         MOVE.B    X,D0            Get  X
         AddMul                    Call the macro
           .  .  .
         MOVE.B    Y,D0            Get  Y
         AddMul                    Call the macro
```

Macros and Parameters

- **A macro parameter is designated within the body of the macro by a backslash "\" followed by a single digit or capital letter:**

```
\1,\2,\3 . . . \A,\B,\C ... \Z
```

- **Thus, up to 35 different, substitutable arguments may used in the body of a macro definition.**
- **The enumerated sequence corresponds to the sequence of parameters passed on invocation.**
  - **The first parameter corresponds to \1 and the 10th parameter corresponds to \A.**
  - **At the time of invocation, these arguments are replaced by the parameters given in the parameter list.**

Macro Example with Parameter Substitution

Defining the macro:

```
AddMul   MACRO                    Macro definition
         ADD.B    #7,\1           Reg = Reg + 7
         AND.W    #00FF,\1        Mask Reg to a byte
         MULU     #12,\1          Reg = Reg x 12
         ENDM                     End of macro def.
```

Invoking the macro:

```
         MOVE.B   X,D0            Get X
         AddMul   D0              Call the macro
          . . .
         MOVE.B   Y,D1            Get Y
         AddMul   D1              Call the macro
```

Labels Within Macros

- **Since a macro may be invoked multiple times within the same program, it is essential that there are no conflicting labels result from the multiple invocation.**
- **The special designator "\@" is used to request unique labels from the assembler macro preprocessor.**
- **For each macro invocation, the "\@" designator is replaced by a number unique to that particular invocation.**
- **The "\@" is appended to the end of a label, and the preprocessor replaces it with a unique number.**

Internal Macro Label Example

Macro SUM adds the sequence of integers in the range: i, i+1, …., n

Macro Definition:

```
SUM       MACRO                      \1 = start   \2 = stop   \3 = sum
          CLR.W      \3              sum  = 0
          ADDQ.W     #1,\2           stop = stop +1
SUM1\@    ADD.W      \1,\3           For i = start  to  stop
          ADD.W      #1,\1           sum = sum + i
          CMP.W      \1,\2
          BNE        SUM1\@
          ENDM
```

Sample macro SUM invocation:

```
          SUM        D1,D2,D3        D1 = start   D2 = stop  D3 = sum
```

Macro Example:
ToUpper, A String Conversion Macro

```
*       ToUpper Address-Register
*       This macro converts a string from lower case to upper case.
*       The argument is an address register.  The string MUST be
*       terminated with $0
*
ToUpper        macro
convert\@      cmpi.b   #0,(\1)        test for end of string
               beq      done\@
               cmpi.b   #'a',(\1)      if < 'a' not lower case
               blt      increment\@
               cmpi.b   #'z',(\1)      if <= 'z' is a lower case
               ble      process\@
increment\@    adda.w   #1,\1
               bra      convert\@
process\@      subi.b   #32,(\1)+      convert to upper case
               bra      convert\@
done\@         NOP
               endm                    End of macro
```