UNIVERSITY *of* WASHINGTON

CSS 422 Hardware and Computer Organization

68K Instruction Set

Professor: Yang Peng

The slides are re-produced by the courtesy of Dr. Arnie Berger and Dr. Wooyoung Kim

Today's Topic

- MC68000 instruction set
  - Chapter 3, 5 by Clements (Available online)
  - 68K manual
  - Online source, http://www.engineering.uiowa.edu/~carch/lectsupp09/68kSimplifiedHandout.pdf

68000 Instruction Set

- **Assembler Directives (Pseudo Op Code)**
  - **ORG, EQU, END, DC, DCB, DS, etc.**
- **Instructions (Op Code)**
  - **Data Movement and Arithmetic operations**
  - **Logical, Shift and Bit operations**
  - **Program Control**
  - **System Control**

Pseudo OP Code

Pseudo Op Codes – Assembler Directives

- **Data Storage Directives**
  - Tells the assembler to allocate blocks of memory for data storage
  - **DC - Define Constant**
    - Format: <label>	DC.<size>	<item>,<item>,<item>…..
    - Can store more than one byte
    - The <size> specifies the **boundary** ***but not the size*** of each item

    Example: date_msg DC.W ‘April 8’   *stored as 8 but not 7 bytes

  - **DCB - Define Constant Block**
    - Format: <label>	DCB.<size>	<length>,<value>
    - Initializes a block of memory **to the same value**
    - The length is the number of bytes, words, or long words

    Example: Repeat_msg   DCB.B   4, $54

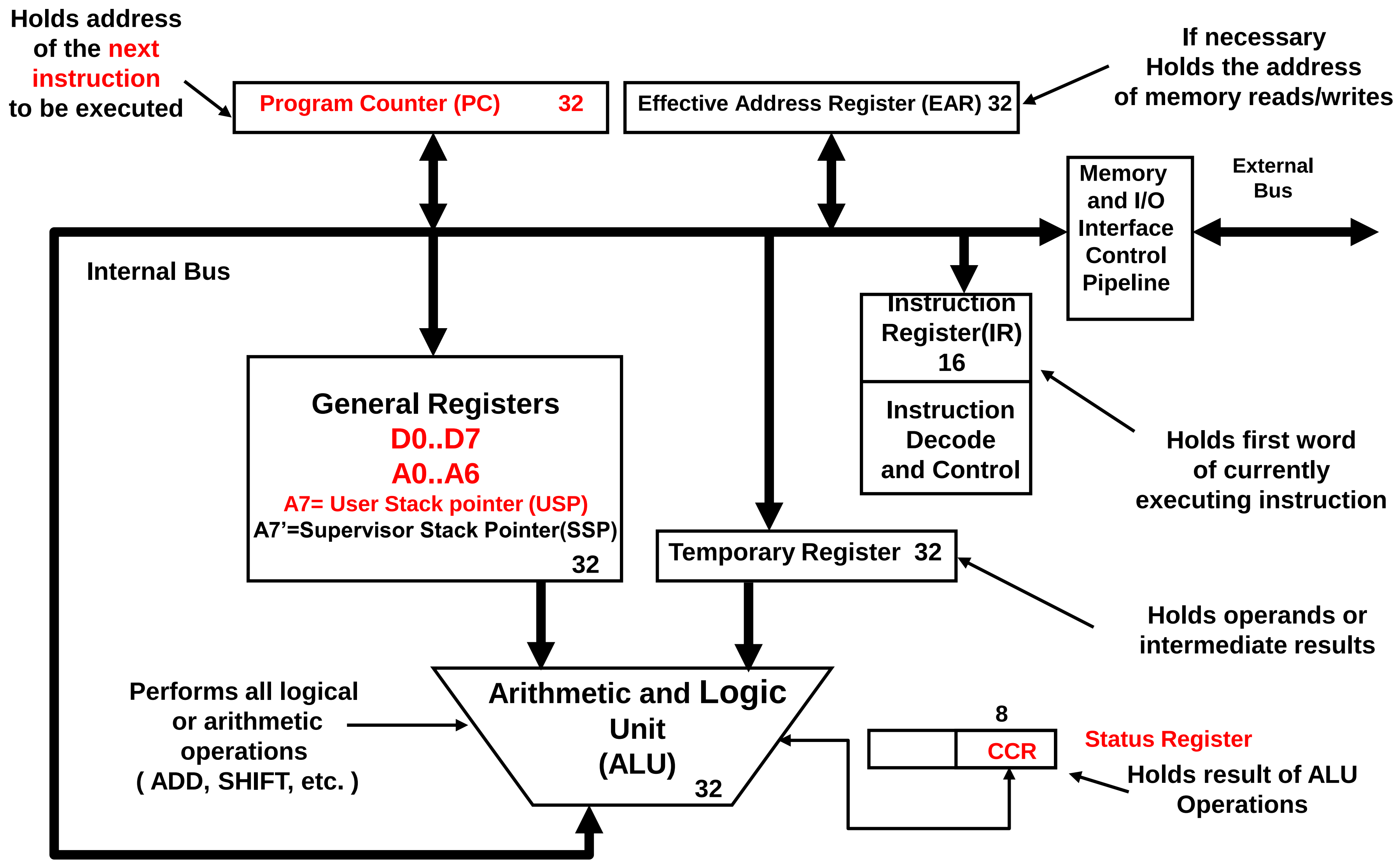
Pseudo Op Codes – Assembler Directives

- **DS - Define Storage**
  - Generates an **uninitialized block of memory**
  - Allocate a space for the result, most of cases
  - Format: <label> DS.<size>   <length>

- **END - End of Source file**
  - **Everything after END is ignored**
  - Format: END   <address>
  - The END pseudo-op also instructs the simulator where to load the program in memory
  - The <address> is typically the address of ORG

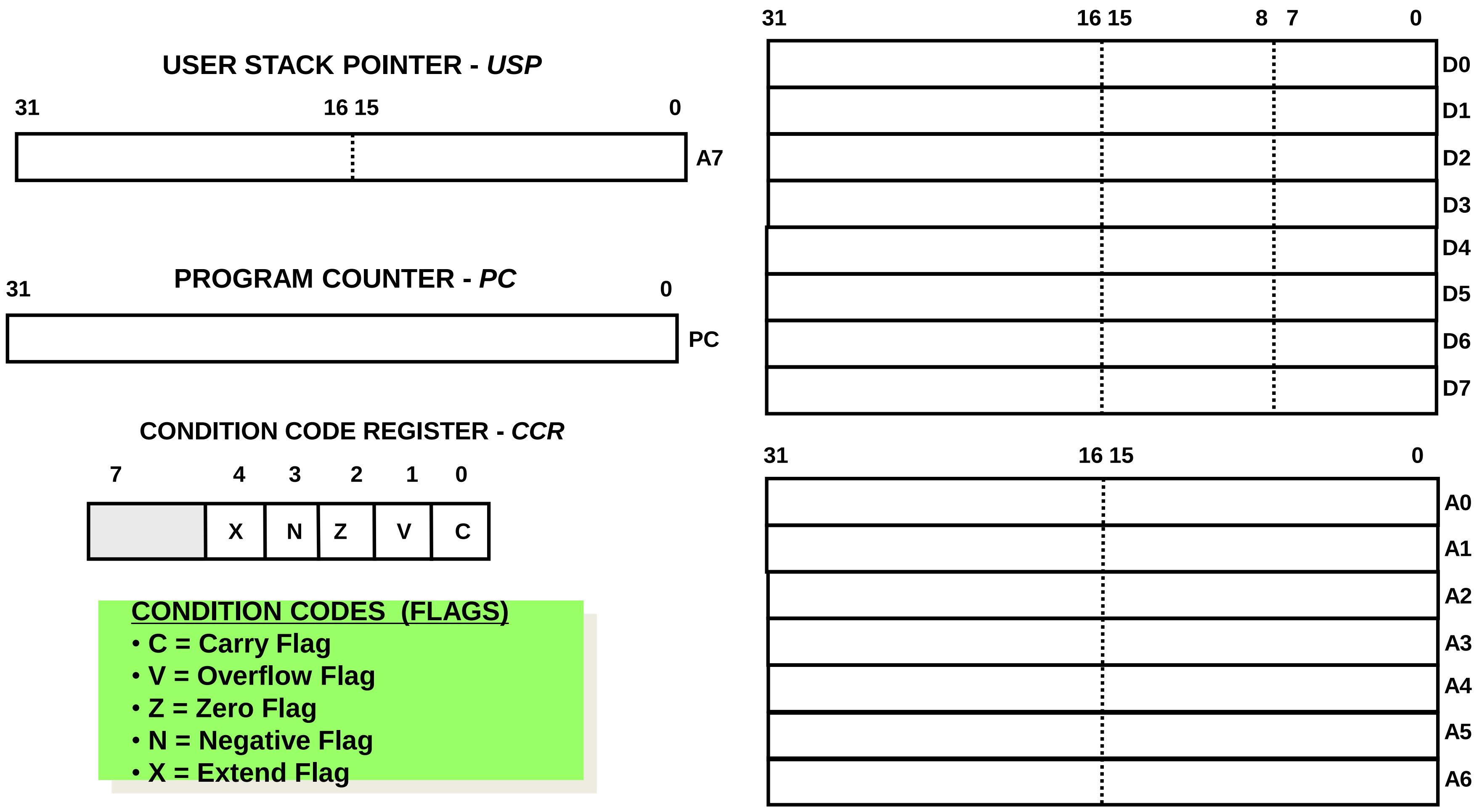
- Note: **ORG and END are complementary**
  - ORG instructs the assembler how to resolve address references
  - END instructs the loader where to put the code in memory
  - They don’t necessarily to be the same!
  - But, we always want to make them using the same address in our code in this class

Hardware Organization of the MC68000


Holds address of the next instruction to be executed
Program Counter (PC) 32
Effective Address Register (EAR) 32
If necessary Holds the address of memory reads/writes
Memory and I/O Interface Control Pipeline
External Bus
Internal Bus
Instruction Register(IR) 16
Instruction Decode and Control
Holds first word of currently executing instruction
General Registers
D0..D7
A0..A6
A7= User Stack pointer (USP)
A7'=Supervisor Stack Pointer(SSP)
32
Temporary Register 32
Holds operands or intermediate results
Performs all logical or arithmetic operations (ADD, SHIFT, etc.)
Arithmetic and Logic Unit (ALU)
32
8
CCR
Status Register
Holds result of ALU Operations

User Programming Model

- For most applications the architectural model of the 68000 is the ***User Programmer's Model***



CONDITION CODES (FLAGS)
- **C = Carry Flag**
- **V = Overflow Flag**
- **Z = Zero Flag**
- **N = Negative Flag**
- **X = Extend Flag**

TRAP #15 task

http://www.easy68k.com/QuickStart/TrapTasks.htm

- TRAP #15 is used for Input/Output
- How to use? Practice with Lab1!
  - MOVE.B #number, D0
  - TRAP #15
  - #number 14 – Display the NULL terminated string at (A1) without CR, LF.
  - #number 4 – Read a number from the keyboard into D1**.L**.
- Based on what you put in the D0, you will have different I/O tasks
- **For your project, you cannot go beyond #number 14!**

Data Movement Instructions

- Move operands (data) among memory or registers

INSTR.	DESCRIPTION	EXAMPLE	

MOVE	Copies an 8-, 16- or 32-bit value from one memory location or register to another memory location or register	`MOVE.B #$8C,D0`
		`MOVE.W #$8C,D0`
		`MOVE.L #$8C,D0`
			[D0]←$XXXXXX8C
			[D0]←$XXXX008C
			[D0]←$0000008C

MOVEA	Copies a source operand to an **address register.** MOVEA operates only on words or longwords. MOVEA.W sign-extends the 16-bit operand to 32 bits	`MOVEA.W #$8C00,A0`
		`MOVEA.L #$8C00,AO`
			[A0]←$FFFF8C00
			[A0]←$00008C00

MOVEQ	Copies a 8-bit signed value in the range –128 to +127 to one of the eight data registers. The data to be moved is sign-extended before it is copied to its destination	`MOVEQ #-3,D0`
		`MOVEQ #4,D0`
			[D0]←$FFFFFFFD
			[D0]←$00000004

MOVEM	Transfers the contents of a **group** of registers specified by a list. The list of registers is defined as A_i-A_j/D_p-D_q. MOVEM operates only on words or longwords.	`MOVEM.L A0-A3/D0-D7,-(A7)`	;copies all working ;registers to stack

MOVE <ea>, <ea>

- Copies a byte, word or long from one location to another
- Source effective address: All the addressing modes
- Destination effective address: All **except** **immediate**, **address register direct** and **program counter relative addressing**
- CCR bit: V and C are cleared, N and Z are updated according to the value of the destination operand, X is unaffected. (**XNZVC = ?**00**)
- For example,

  MOVE.B #$8C, D0 results in XNZVC set to → ?1000

  MOVE.B #0, D0 results in XNZVC set to → ?0100
- *Bug report for easy68K simulator!*
  - *MOVE.L #(data<5bits), D0→ assembled to MOVEQ instruction*
  - *Address register as the destination is considered valid*

MOVEA <ea>, An

- Copy an Address value from source to destination
- Source effective address: All the addressing modes
- Destination effective address: only Direct Address Register
- Can only take **.W** and **.L**
- **Does not affect the state of the CCR**
- **Words are sign extended** when moved into the register

MOVEA.W #$8C00, A0	[A0] ← $FFFF8C00
MOVEA.W #$4F00, A0	[A0] ← $00004F00

- **Longwords are just an absolute longword address**

MOVEA.L #$8C00, A0	[A0] ← $00008C00

MOVEA

- What value will you see in A1 after the following operations?

MOVEA.W #$7FFF, A1

$7FFF = % 0111 1111 1111 1111 ➔ % 0000 0000 0000 0000 0111 1111 1111 1111
= $ 0000 7 FFF

MOVEA.W #$8000, A1

$8000 = % 1000 0000 0000 0000 ➔ % 1111 1111 1111 1111 1000 0000 0000 0000
= $ FFFF8000

MOVEA.L #$8000, A1

- Then A1 contains $00008000

More about MOVEA

- **Addresses are signed numbers!**
  - Why need negative number?
  - For the **reverse direction**, instead of forwarding

- Why not just use Long-word addresses only?
  - **Saves memory space**
  - Requires one less extension word fetch
  - ROM code is usually placed in low or high memory

- **MOVEA** and some other Address Operation instructions can
  - explicitly specify word and long address (MOVEA.W, MOVEA.L)
  - sign extend the address automatically
- **What about other instructions**?

LEA <ea>, An

- Source → An
- Load effective address
- **Does not affect the state of the CCR**
- **Size = Longword**
- Source effective address:
  - (An), An, Absolute addressing (Word, Long)

LEA $A000, A0	[A0] ← $0000A000
LEA (A2), A0	[A0] ← [(A2)]

Is “LEA $A000, A0” the same as “MOVEA.W #$A000, A0” ?

No. LEA does not allow word-address. So, all values are considered longword addresses → no sign extension!

After “MOVEA.W #$A000, A0”, A0 has FFFFA000

MOVEQ #<data>, Dn

- Move Quick instruction copies **one byte immediate data as** **8-bit signed value** (-128~127) to a **data register**
- ***Cannot* move the data out of range**
- **The data to be moved is sign-extended** before it is copied to its destination
- CCR bit: V and C are cleared, N and Z are updated according to the value of the destination operand, X is unaffected. (**XNZVC = ?**00**)
- Data is saved into data register as Longword (smaller values are sign-extended)

```
MOVEQ    #-3, D0                  [D0] ← $FFFFFFFD
(-3 = 11111101 with 8-bits)
MOVEQ    #4, D0                   [D0] ← $00000004
```

MOVEM <register list>, <ea>
MOVEM <ea>, <register list>

- Transfers the contents of a **group of registers** specified by a list. The list of registers is defined as **Ai-Aj/Dp-Dq**.
- MOVEM operates **only** on **words** or **longwords**.
- Use assembler directive REG to bind a name to the list

```
MOVEM.L A0-A6/D0-D7, -(SP)
```

```
GROUP REG A0-A6/D0-D7
MOVEM.L GROUP, -(SP)
```

Pseudo Op Codes – Assembler Directives

- **REG - Register Range**
  - Allows a list of registers to be defined
  - Format: <label> REG <register list>
  - Registers may be specified as a single register, An or Dn, separated by slashes, i.e. A1/A5/A7/D1/D3
  - Register ranges may also be specified, i.e., A0-A3
  - Thus:

```
save_reg        REG     A0-A3/A5/D0-D7
```

  - Refer to the MOVEM instruction in your programmer’s manuals
  - Will go over again for subroutine

MOVEM <register list>, <ea>
MOVEM <ea>, <register list>

- Mostly used with **pre-decrementing** (registers to memory) and **post-incrementing** (memory to registers)
- Frequently used to **save working registers** on entering a subroutine and to **retrieve** them on exiting the subroutine

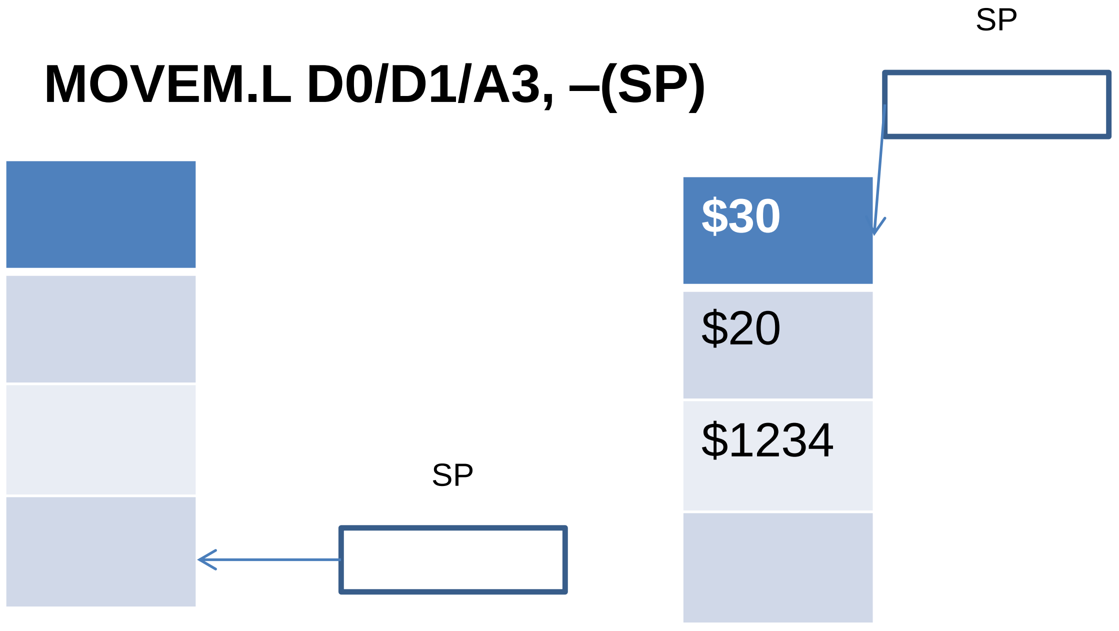
```
MOVEM.L  A0-A3/D0-D7, -(A7)  *copies all working registers to stack
MOVEM.L (A7)+, A0-A3/D0-D7  *Restore the registers
```

WHY do you need to save the registers to the stack when entering a subroutine??

- 68K Always **writes** to memory in the order of **A7 to A0, D7 to D0**
- 68K Always **reads** from memory in the order of **D0 to D7, A0 to A7**

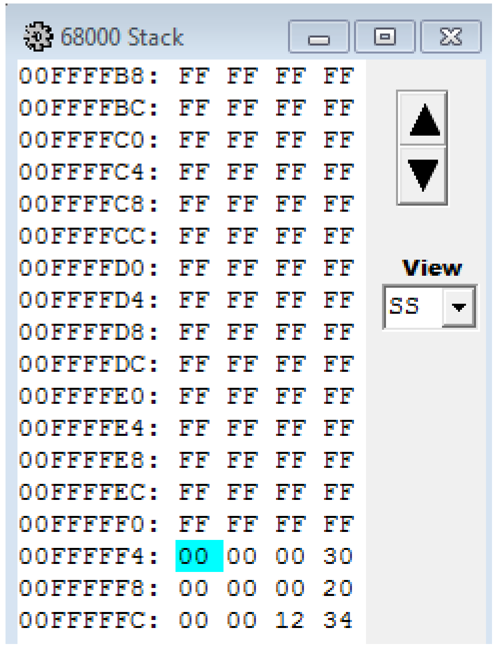
E.g., D0 has $30, D1 has $20, A3 had $1234

MOVEM.L D0/D1/A3, –(SP)



Before executing

After executing

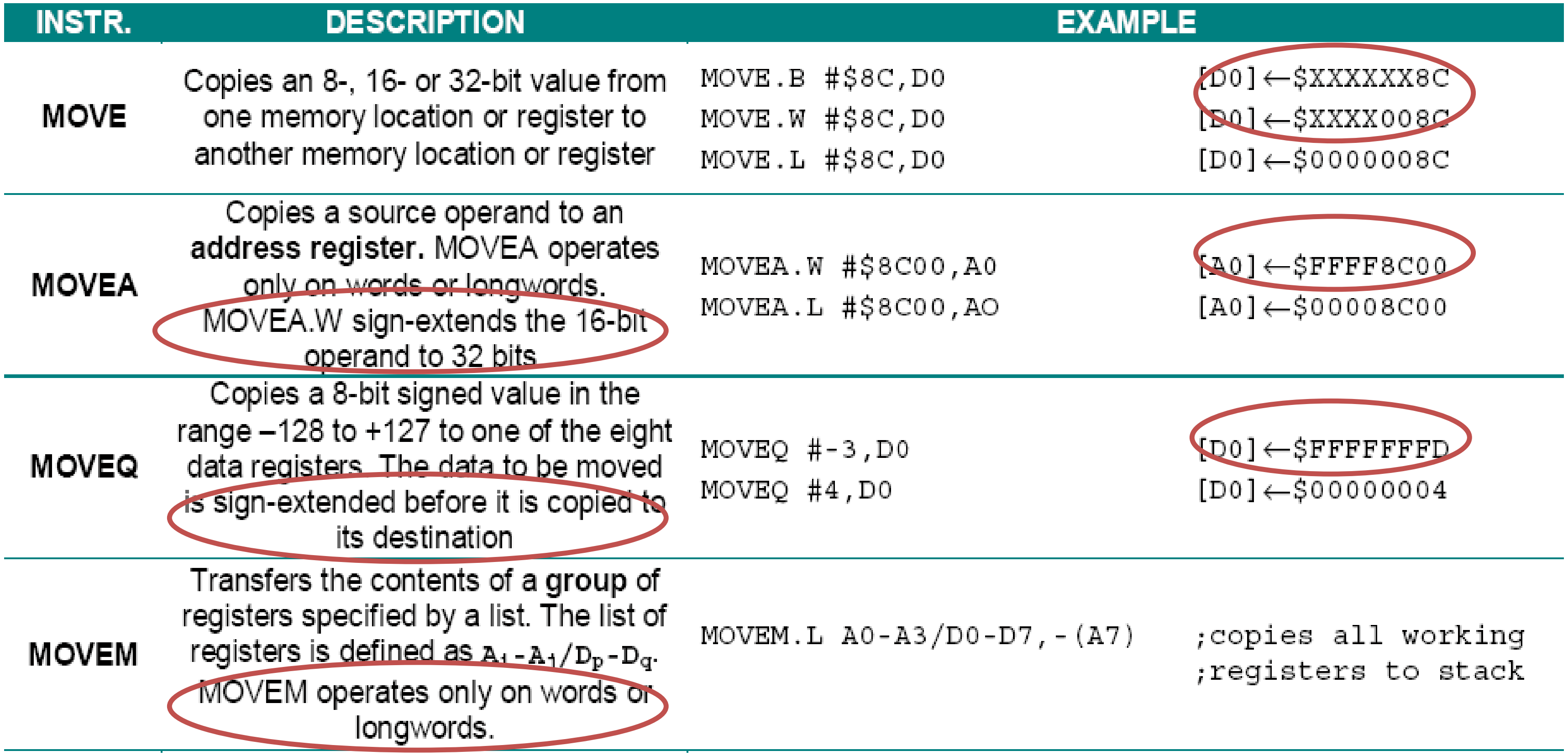


```
00FFFFB8:  FF FF FF FF
00FFFFBC:  FF FF FF FF
00FFFFC0:  FF FF FF FF
00FFFFC4:  FF FF FF FF
00FFFFC8:  FF FF FF FF
00FFFFCC:  FF FF FF FF
00FFFFD0:  FF FF FF FF
00FFFFD4:  FF FF FF FF
00FFFFD8:  FF FF FF FF
00FFFFDC:  FF FF FF FF
00FFFFE0:  FF FF FF FF
00FFFFE4:  FF FF FF FF
00FFFFE8:  FF FF FF FF
00FFFFEC:  FF FF FF FF
00FFFFF0:  FF FF FF FF
00FFFFF4:  00 00 00 30
00FFFFF8:  00 00 00 20
00FFFFFC:  00 00 12 34
```

```
00001000                        * Put program code here
00001000  303C 0030                 MOVE #$30, D0
00001004  323C 0020                 MOVE #$20, D1
00001008  367C 1234                 MOVEA #$1234, A3
0000100C  48E7 C010                 MOVEM.L D0/D1/A3, -(SP)
```

Data Movement Instructions

- Move operands (data) among memory or registers



INSTR.	DESCRIPTION	EXAMPLE	

MOVE	Copies an 8-, 16- or 32-bit value from one memory location or register to another memory location or register	`MOVE.B #$8C,D0`
		`MOVE.W #$8C,D0`
		`MOVE.L #$8C,D0`
			`[D0]←$XXXXXX8C`
			`[D0]←$XXXX008C`
			`[D0]←$0000008C`

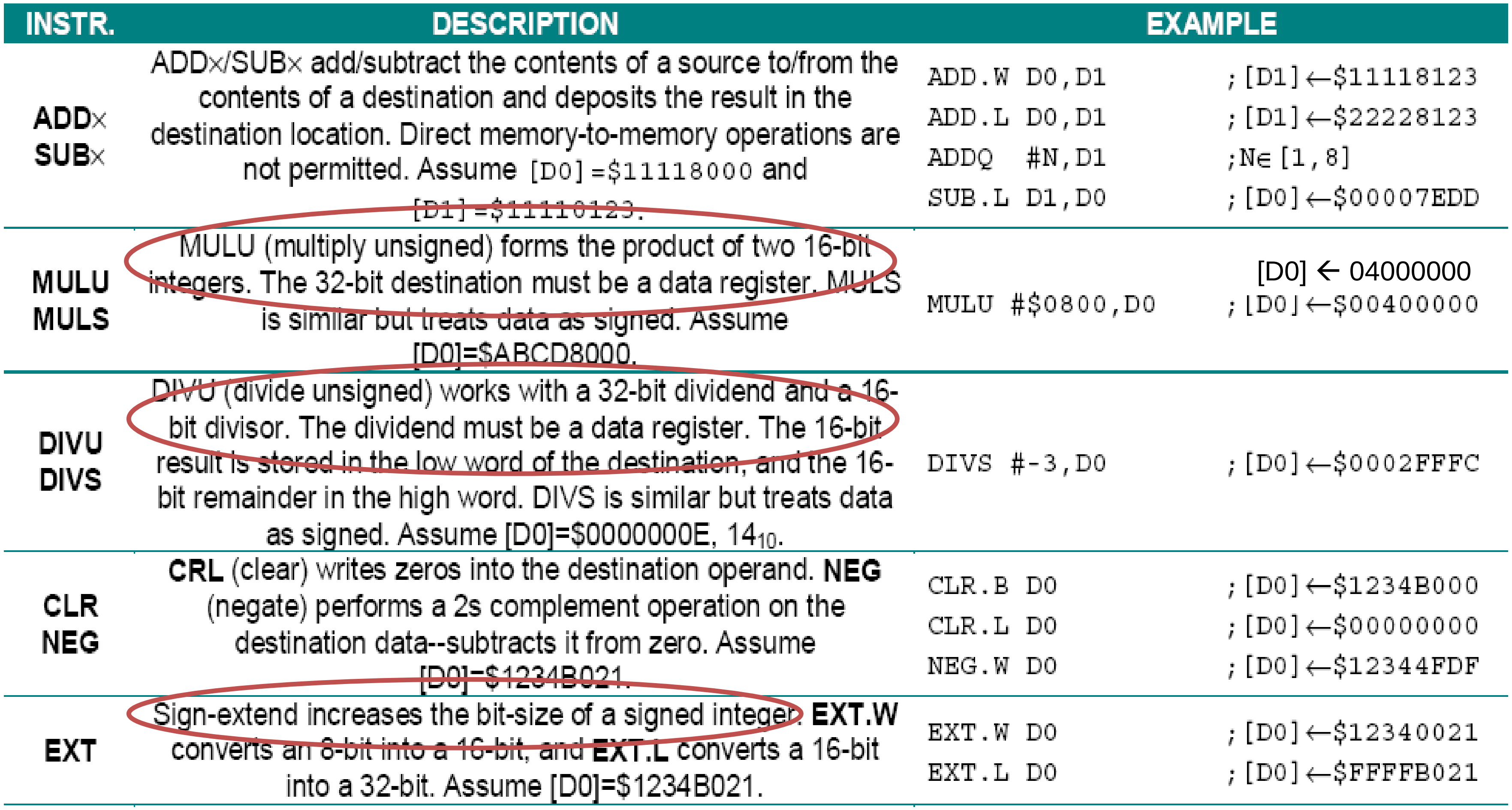
MOVEA	Copies a source operand to an **address register.** MOVEA operates only on words or longwords. MOVEA.W sign-extends the 16-bit operand to 32 bits	`MOVEA.W #$8C00,A0`
		`MOVEA.L #$8C00,AO`
			`[A0]←$FFFF8C00`
			`[A0]←$00008C00`

MOVEQ	Copies a 8-bit signed value in the range –128 to +127 to one of the eight data registers. The data to be moved is sign-extended before it is copied to its destination	`MOVEQ #-3,D0`
		`MOVEQ #4,D0`
			`[D0]←$FFFFFFFD`
			`[D0]←$00000004`

MOVEM	Transfers the contents of a **group** of registers specified by a list. The list of registers is defined as A_i-A_j/D_p-D_q. MOVEM operates only on words or longwords.	`MOVEM.L A0-A3/D0-D7,-(A7)`	`;copies all working`
			`;registers to stack`

68000 Arithmetic Instructions

- 68K has a conventional set of integer arithmetic operations.
- 68K does not support floating-point operations.



INSTR.	DESCRIPTION	EXAMPLE	

ADD×
SUB×
	ADD×/SUB× add/subtract the contents of a source to/from the contents of a destination and deposits the result in the destination location. Direct memory-to-memory operations are not permitted. Assume `[D0]=$11118000` and `[D1]=$11110123`.	`ADD.W D0,D1`
		`ADD.L D0,D1`
		`ADDQ #N,D1`
		`SUB.L D1,D0`
			;[D1]←$11118123
			;[D1]←$22228123
			;N∈[1,8]
			;[D0]←$00007EDD

MULU **MULS**	MULU (multiply unsigned) forms the product of two 16-bit integers. The 32-bit destination must be a data register. MULS is similar but treats data as signed. Assume [D0]=$ABCD8000.	`MULU #$0800,D0`	[D0] ← 04000000 ;[D0]←$00400000
DIVU **DIVS**	DIVU (divide unsigned) works with a 32-bit dividend and a 16-bit divisor. The dividend must be a data register. The 16-bit result is stored in the low word of the destination, and the 16-bit remainder in the high word. DIVS is similar but treats data as signed. Assume [D0]=$0000000E, 14_{10}.	`DIVS #-3,D0`	;[D0]←$0002FFFC

CLR
NEG
	CRL (clear) writes zeros into the destination operand. **NEG** (negate) performs a 2s complement operation on the destination data--subtracts it from zero. Assume [D0]=$1234B021.	`CLR.B D0`
		`CLR.L D0`
		`NEG.W D0`
			;[D0]←$1234B000
			;[D0]←$00000000
			;[D0]←$12344FDF

EXT	Sign-extend increases the bit-size of a signed integer. **EXT.W** converts an 8-bit into a 16-bit, and **EXT.L** converts a 16-bit into a 32-bit. Assume [D0]=$1234B021.	`EXT.W D0`
		`EXT.L D0`
			;[D0]←$12340021
			;[D0]←$FFFFB021

ADD

- **ADD** (Add binary)
  - A **data register** must be the source or destination of an ADD instruction
  - ADD <ea>, Dn   or ADD Dn, <ea>
  - **Direct memory-to-memory additions are not permitted**
  - Example:
    - Before: <D2> = $12345678, <D3> = $5F02C332
    - **ADD.B   D2,D3**
    - After: <D2> = $123456**78**, <D3> = $5F02C3**AA** **(XNZVC = 01010)**
  - CCR bits are changed based on the result
    - X : EXTEND flag: Used in certain operations, generally affected like CARRY
    - N : NEGATIVE flag: if **Bit 7** is 1, then N=1
    - Z : ZERO flag: if zero, Z=1
    - V : OVERFLOW flag
    - C : CARRY flag

ADDA and ADDQ

- **ADDA** (Add Address): Adds data to an **address register**
  - Only **word** and **longword** operations are allowed
  - ADDA.W #$8122, A0 ; [A0] ← $FFFF8122 + A0 (**sign-extended**)
  - CCR is not affected

- **ADDQ** (Add Quick): Adds a number in the range **1 to 8**
  - 3-bits can represent 1 - 8, (Note: **8 being 000 here**)
  - **Word** and **longword** size when the destination is an **address register**
  - If destination is address register, CCR is not affected
  - Faster as the immediate data is just 3 bits

- ADDQ, SUBQ support operands 1 to 8 only and not sign-extended

ADDI

- **ADDI** (Add Immediate): Adds number to **data register or memory**
  - Byte, word or longword operations
  - ADDI.W #$1234, (A0) is **valid**
  - **ADD**.W #$1234, (A0) is **invalid**

- ***Question: If <D2> = $047128E8 what's the difference between***

  - ADDI.B #$20,D2 * <D2> = ??, C = ??

    $04712808

    C=1

  - ADDI.W #$20,D2 * <D2> = ??, C = ??

    $04712908

    C=0

MULU and MULS

- MULU (multiply unsigned) , MULS (multiply signed)
  - MULU <ea>, Dn
  - MULS <ea>, Dn
  - Product of **two 16-bit integers** (Only Word is allowed)
  - ***Only lower order word*** is used in multiplication: upper word is unused
  - [D0] = $ABCD**8000**
  - MULU #$0800, D0 ; [D0] ← 04000000
  - CCR Bits
    - X: unchanged
    - V = C = 0
    - N and Z depend on the result

DIVU and DIVS

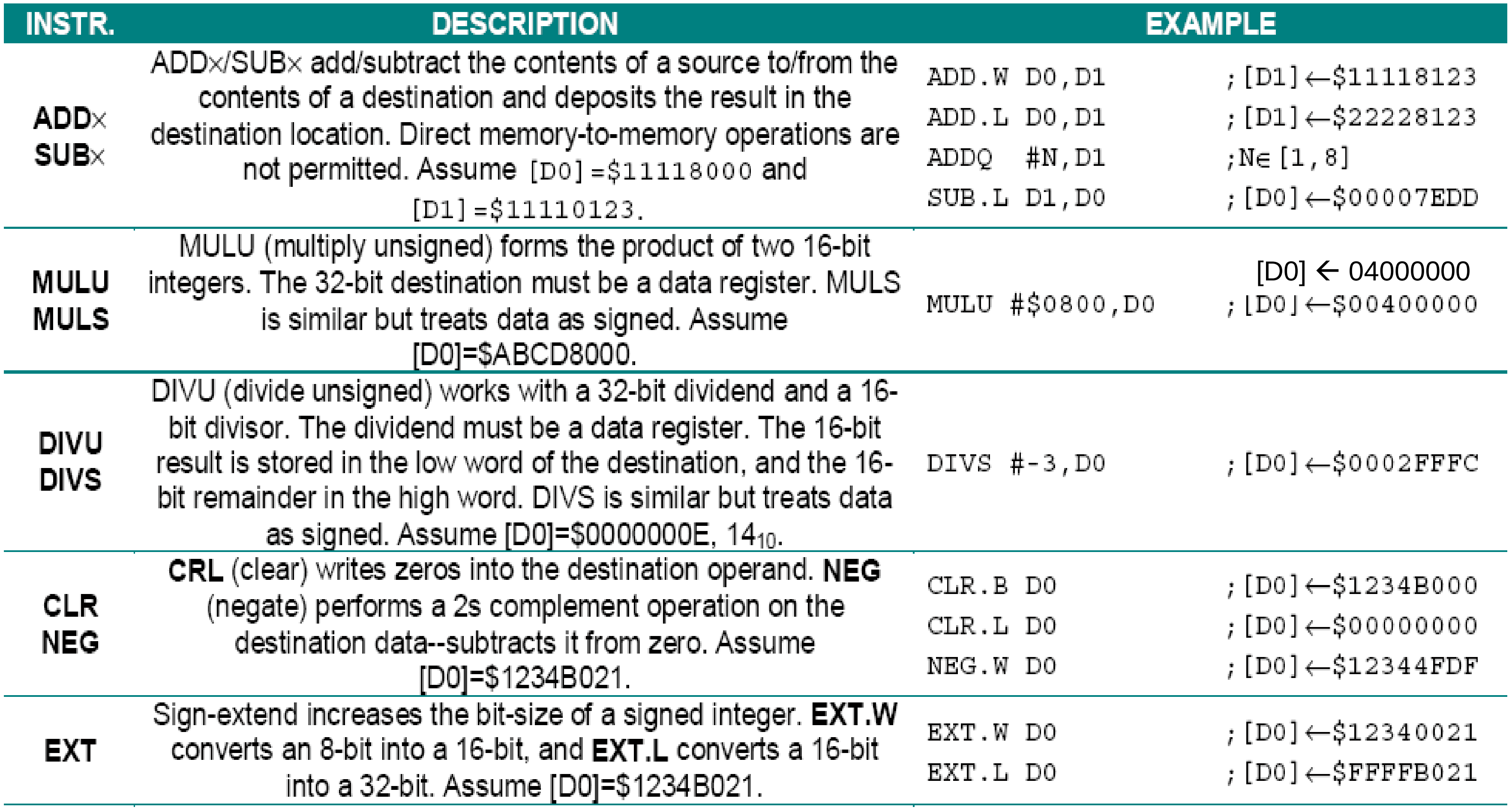
- **DIVU** (divide with unsigned) , **DIVS** (divide with signed)
  - DIVU <ea>, Dn
  - DIVS <ea>, Dn
  - Destination ÷ Source → Destination
  - **Destination** is **longword, Source** is **word**
  - The **quotient** is in the **lower-order word**
  - The **remainder** is in the **upper-order word**
  - CCR bits
    - X: unchanged
    - N, Z: depends on the **quotient**, but if V=1 or divide by zero then undetermined
    - V: 1 if overflow occurs. If divide zero, then undetermined
    - C: 0

CLR, CMP and EXT

- **CLR** (Clear an operand)
  - May be used on **bytes**, **words** and **longwords**
  - All condition codes except X are affected
  - **Faster than MOVE.L $0, D0**
  - Z = 1, V=C=N = 0, X is unchanged
- **CMP** (Compare data with a **data register**)
  - The **second operand must be a data register**
  - Sets the condition codes accordingly
  - Subtracts the source operand from the destination operand
    - Only the condition code flags are affected!
    - Source and destination operands are not affected!
- **EXT** (extend byte to word or word to longword)
  - EXT.W Dn: extend Dn(7) to Dn(8:15) (extend byte to word)
  - EXT.L Dn: extend Dn(15) to Dn(16:31) (extend word to longword)

68000 Arithmetic Instructions

- 68K has a conventional set of integer arithmetic operations.
- 68K does not support floating-point operations.



INSTR.	DESCRIPTION	EXAMPLE	

ADD×
SUB×
	ADD×/SUB× add/subtract the contents of a source to/from the contents of a destination and deposits the result in the destination location. Direct memory-to-memory operations are not permitted. Assume `[D0]=$11118000` and `[D1]=$11110123`.	`ADD.W D0,D1`
		`ADD.L D0,D1`
		`ADDQ  #N,D1`
		`SUB.L D1,D0`
			`;[D1]←$11118123`
			`;[D1]←$22228123`
			`;N∈[1,8]`
			`;[D0]←$00007EDD`

MULU MULS	MULU (multiply unsigned) forms the product of two 16-bit integers. The 32-bit destination must be a data register. MULS is similar but treats data as signed. Assume [D0]=$ABCD8000.	`MULU #$0800,D0`	[D0] ← 04000000 `;[D0]←$00400000`
DIVU DIVS	DIVU (divide unsigned) works with a 32-bit dividend and a 16-bit divisor. The dividend must be a data register. The 16-bit result is stored in the low word of the destination, and the 16-bit remainder in the high word. DIVS is similar but treats data as signed. Assume [D0]=$0000000E, 14_{10}.	`DIVS #-3,D0`	`;[D0]←$0002FFFC`

CLR
NEG
	CRL (clear) writes zeros into the destination operand. **NEG** (negate) performs a 2s complement operation on the destination data--subtracts it from zero. Assume [D0]=$1234B021.	`CLR.B D0`
		`CLR.L D0`
		`NEG.W D0`
			`;[D0]←$1234B000`
			`;[D0]←$00000000`
			`;[D0]←$12344FDF`

EXT	Sign-extend increases the bit-size of a signed integer. **EXT.W** converts an 8-bit into a 16-bit, and **EXT.L** converts a 16-bit into a 32-bit. Assume [D0]=$1234B021.	`EXT.W D0`
		`EXT.L D0`
			`;[D0]←$12340021`
			`;[D0]←$FFFFB021`

68000 Logical Operations

INSTR.	DESCRIPTION	EXAMPLE	

AND
ANDI
	Bit-wise logical AND operation. Normally used to **clear**, or **mask**, certain bits in a destination operand.	`ANDI.B #%0111111,D0`	`;clear the 8th least`
			`;significant bit of D0`

OR
ORI
	Bit-wise logical OR operation. Normally used to **set** certain bits in a destination operand.	`ORI.B #%10101010,D0`	`;set even bits of D0`
			`;lowest byte`

EOR EORI	Bit-wise logical XOR operation.	`EOR.B #%11111111,D0`	`;XOR of the lowest byte of D0`
NOT	Bit-wise NOT operation. Assume `[D0]=$1234F0F0.`	`NOT.W D0`	`;[D0]←$12340F0F`
TST	Similar to `CMP #0, operand`	`TST D0`	`;update N,Z and clear V,C`

68000 Logical Operations

- **AND** (Performs bitwise logical AND)
  - Example
    - **AND.W A0,D4**
    - "AND.W": only the lower 16-bits of each register is used in a word operation
  - Example

    Before: <D0> = $3795AC5F and <D1> = $B6D34B9D

    AND.W D0,D1

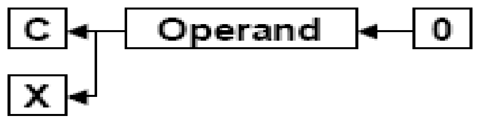
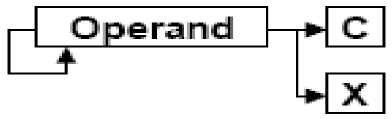
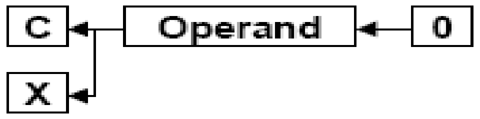
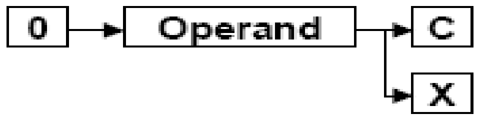
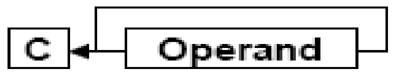
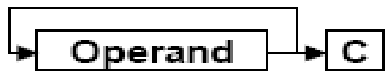
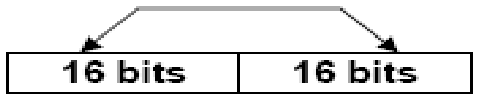
    After: <D0> = $3795AC5F and <D1> = $B6D3081D
  - Shortcut to remember: Any hex digit AND'ed with F returns the digit

    Any hex digit AND'ed with 0 returns 0
- **ANDI** (AND immediate data)
  - Example: **ANDI.B #$5A,D7**

68000 Logical Operations (2)

- **OR** (Perform bit-wise logical inclusive OR of the operands)
- **ORI** (Immediate bit-wise OR with destination operand)
- **EOR** (Perform bit-wise logical Exclusive OR of the operands)
- **EORI** (Immediate bit-wise Exclusive OR with destination operand)
- **NOT** (Bit-wise logical complement)
  - Example
    - Before: <D3> = $FF4567FF
    - **NOT.B D3**
    - After: <D3> = $FF456700

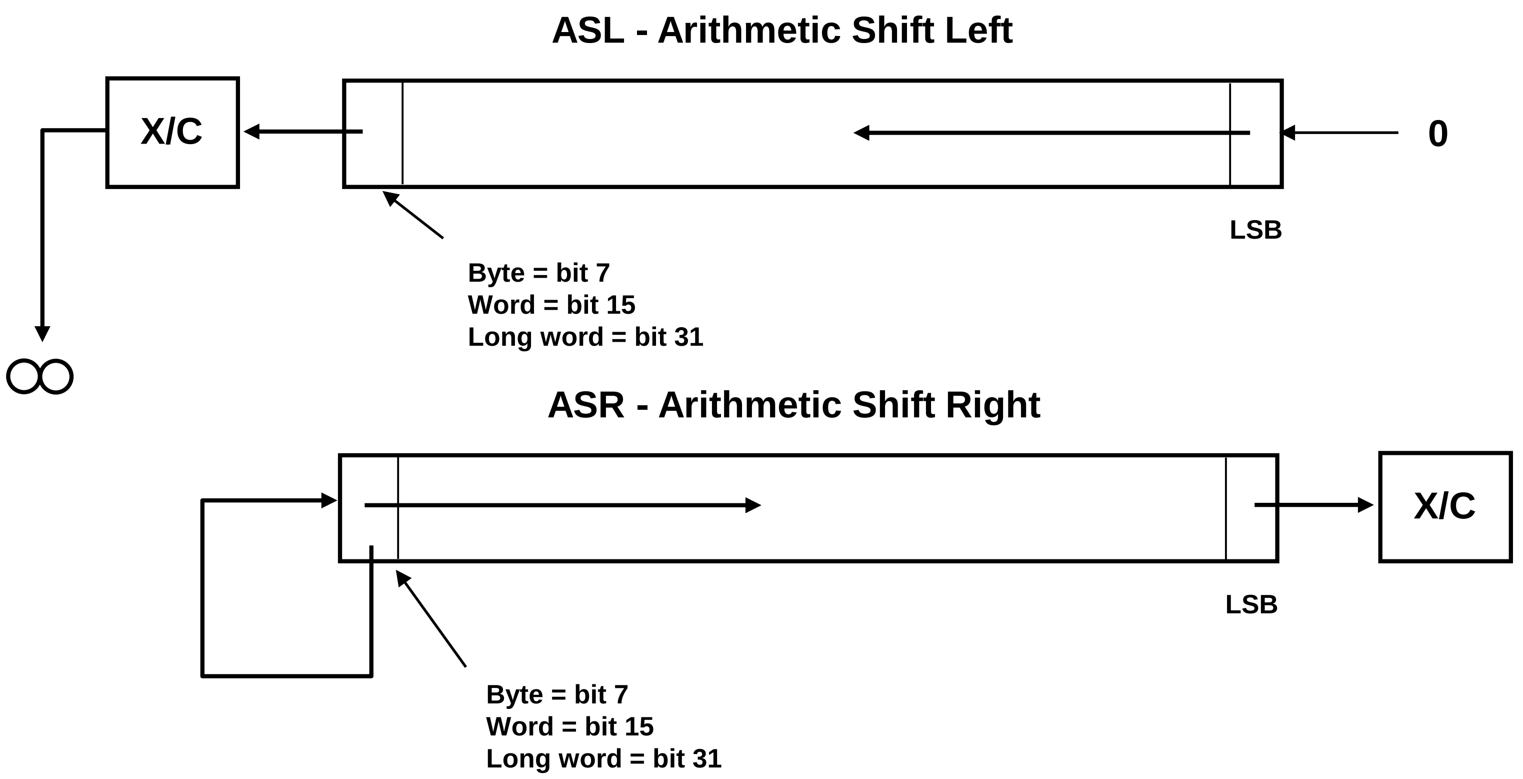
68000 Shift and Rotate Instructions

INSTR.	OPERATION	BIT MOVEMENT
ASL	Arithmetic shift left	
ASR	Arithmetic shift right	
LSL	Logic shift left	
LSR	Logic shift right	
ROL	Rotate left	
ROR	Rotate right	
SWAP	Swap words of a longword	

Arithmetic Shift

- Byte, Word, Longword (careful for the data size)
- **V = 1 if MSB changed at any time during the shift operation**
- X/C = the last bit shifted out of the operand
- N,Z: based on the result
- **ASL**
  - Bits shift to left
  - **New bits, 0’s, are added at the end**
  - A bit which is shifted out goes to C-bit and X-bit
  - If there are multiple bits out of the end, then the last bit goes to C and X
  - If the sign bit has changed at the end, then V bit set
- **ASR**
  - Bits shift to right
  - The **most significant bit is preserved** (to preserve the sign bit)
  - The bit shifted out of the end goes to C-bit and X-bit

ASL and ASR

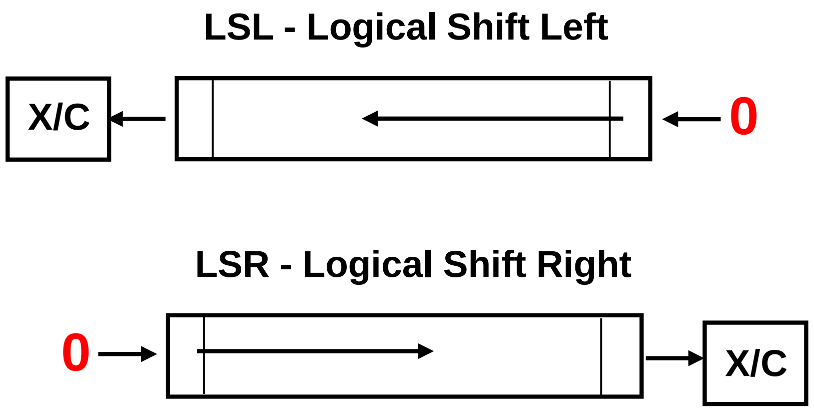


Question:

- Before: <D0> = $3456ABCF and X=0, C=0
- ASL.W #3,D0
- After: <D0> = ???, X/C = ???

Logical Shift

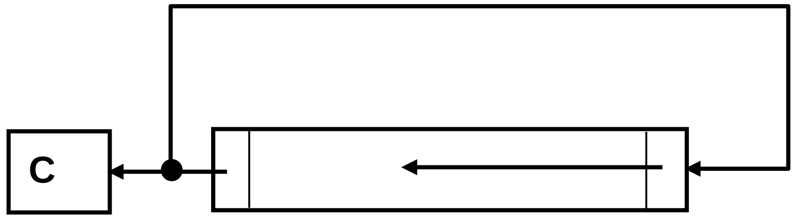
- Byte, Word, Longword
- V = 0 always
- X/C = the last bit shifted out of the operand
- N,Z: based on the result



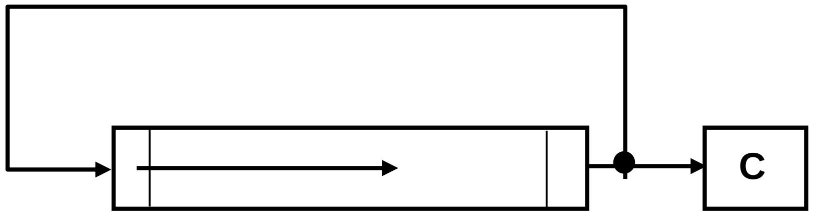
0's will be added, and sign bits cannot be preserved!

Rotate (Circular Shift)

- Byte, Word, Longword
- The bit shifted out of the operand moves to the position of the bit shifted in.
- **No bit is lost (all bits are preserved, but in different places)**
- V = 0 always
- X: not affected
- C = the last bit shifted out of the operand
- N,Z: based on the result



ROL - Roll Left



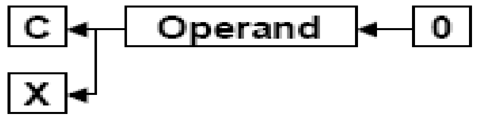
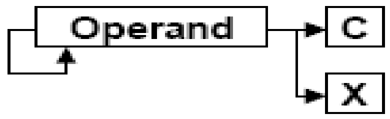
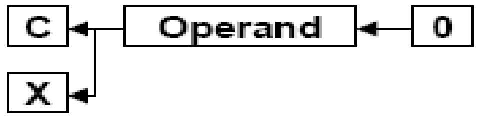
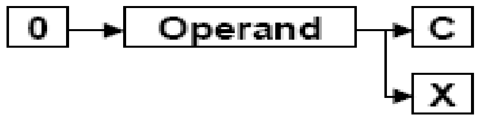
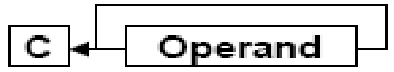
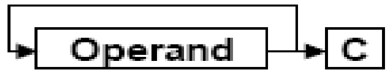
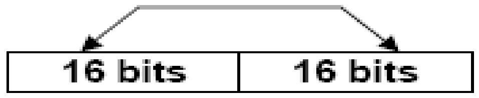
ROR - Roll Right

Question:
- Before: <D0> = $3456CBCF
- ROL.W #3,D0
- After: <D0> = ???, XNZVC = ???

68000 Shift and Rotate Instructions

- Byte, word or long word operands
  - Contents of data registers D0-D7
  - Contents of memory locations
- Shifting **data in a data register**: **ASd #data,Dn** or **ASd Dx,Dy**
  - Instruction must supply the **direction**, and **total number of shifts/rotations**
  - **If number of shifts ≤ 8, use the immediate form**
  - **If number of shifts > 8, number must be in another data register**
  - Example: `<D0> = $0000000A`
    - `ASL.B  #4,D2   *Arithmetic shift left 4 bits`
    - `LSR.W  #6,D3   *Logical shift right 6 bits`
    - `ROL.L  #7,D4   *Rotate left 7 bits`
    - `LSR.W  D0,D5   *Logical shift right 10 bits`
- Shifting **data in a memory location**: **ASd <ea>**
  - **Only one bit** is shift at a time and **only word operand**

68000 Shift and Rotate Instructions

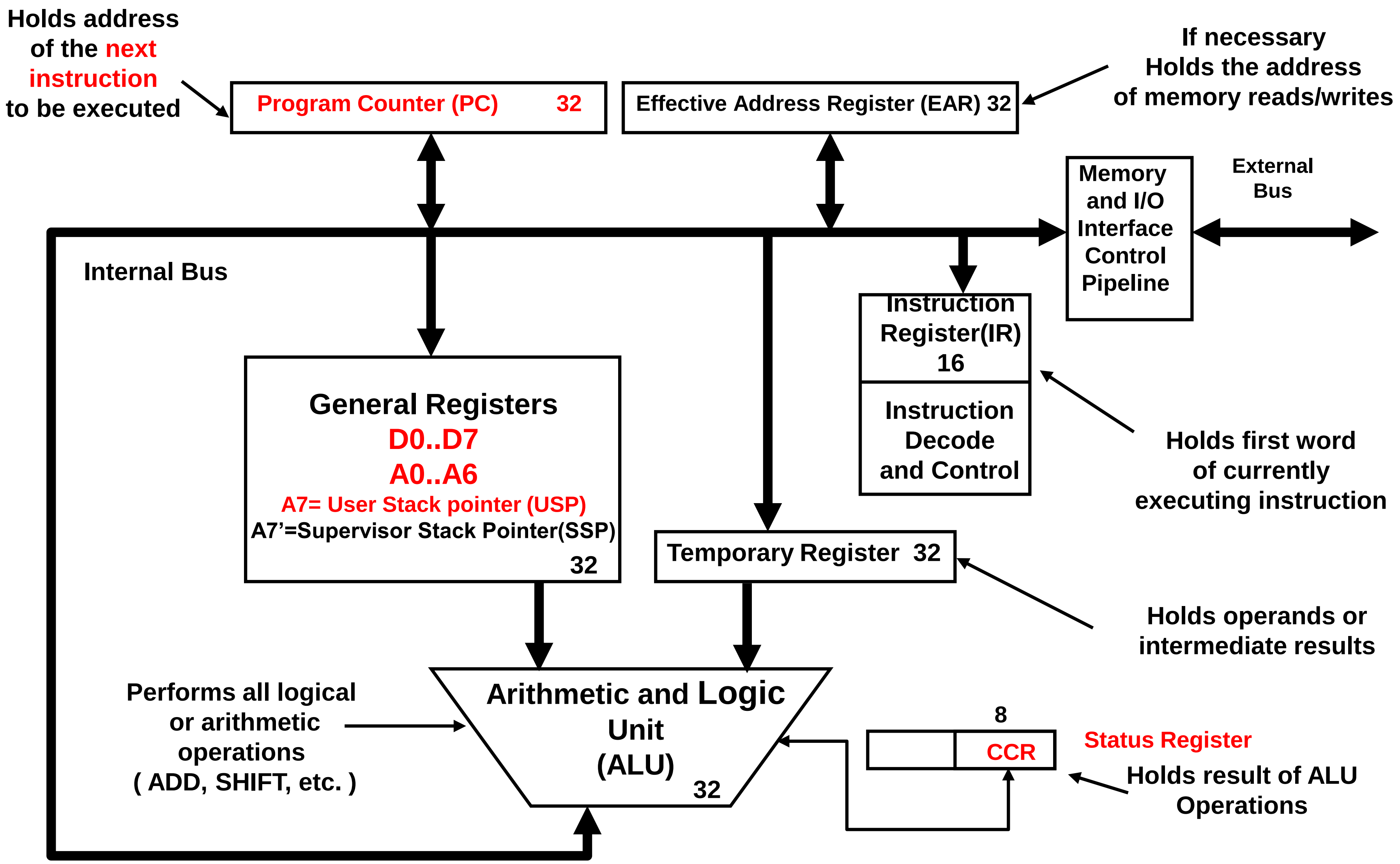
INSTR.	OPERATION	BIT MOVEMENT
ASL	Arithmetic shift left	
ASR	Arithmetic shift right	
LSL	Logic shift left	
LSR	Logic shift right	
ROL	Rotate left	
ROR	Rotate right	
SWAP	Swap words of a longword	

68000 Bit Manipulation

- **Only Z-bit in CCR is affected**
- Test one bit of a byte if in memory (source is modulo 8)
- Test one bit of a Longword if in data register

INSTR.	DESCRIPTION	EXAMPLE (Assume [D0] =$00000009)	
BSET	**Bit test and set** Causes the Z-bit to be set if the specified bit is zero and then forces the specified bit of the operand to be set to one	`BSET #2, D0`	`;[D0]←$0000000D and [Z]←1`
BCLR	**Bit test and clear** works like BSET except that the specified bit is cleared (forced to zero) after it has been tested	`BCLR #0, D0`	`;[D0]←$00000008 and [Z]←0`
BCHG	**Bit test and change** causes the value of the specified bit to be reflected in the Z-bit and then toggles (inverts) the state of the specified bit	`BCHG #4, D0`	`;[D0]←$00000019 and [Z]←1`
BTST	**Bit test** reflects the value of the specified bit in the Z-bit	`BTST #2, D0`	`;[Z]←1`

Hardware Organization of the MC68000


Holds address of the next instruction to be executed
Program Counter (PC) 32
Effective Address Register (EAR) 32
If necessary Holds the address of memory reads/writes
Memory and I/O Interface Control Pipeline
External Bus
Internal Bus
Instruction Register(IR) 16
Instruction Decode and Control
Holds first word of currently executing instruction
General Registers
D0..D7
A0..A6
A7= User Stack pointer (USP)
A7'=Supervisor Stack Pointer(SSP)
32
Temporary Register 32
Holds operands or intermediate results
Performs all logical or arithmetic operations (ADD, SHIFT, etc.)
Arithmetic and Logic Unit (ALU)
32
8
CCR
Status Register
Holds result of ALU Operations

Program Control

INSTR.	DESCRIPTION
BRA	BRA (branch always) implements an unconditional branch, relative to the PC. The offset is expressed as an 8- or 16-bit signed integer. If the destination is outside of a 16-bit signed integer, BRA **cannot** be used.
B*cc*	B*cc* (branch conditional) is used whenever program execution must follow one of two paths depending on a condition. The condition is specified by the mnemonic *cc*. The offset is expressed as an 8- or 16-bit signed integer. If the destination is outside of a 16-bit signed integer, B*cc* **cannot** be used.
BSR **RTS**	BSR branches to a subroutine. The PC is saved on the stack before loading the PC with the new value. RTS is use to return from the subroutine by restoring the PC from the stack.
JMP	JMP (jump) is similar to BRA. The only difference is that BRA uses only relative addressing, whereas JMP has more addressing modes, including absolute address (see reference manual).
JSR **RTS**	**Similar to BSR and RTS. But, JSR has more addressing modes than BSR. See reference manual.**

cc	CONDITION	BRANCH TAKEN IF
CC	Carry clear	C=0
CS	Carry set	C=1
NE	Not equal	Z=0
EQ	Equal	Z=1
PL	Plus	N=0
MI	Minus	N=1
HI	Higher than	$\overline{C}\overline{Z} = 1$
LS	Lower than or same as	C+Z=1
GT	Greater than	$NV\overline{Z} + \overline{NVZ} = 1$
LT	Less than	$N\overline{V} + \overline{N}V = 1$
GE	Greater than or equal to	$N\overline{V} + \overline{N}V = 0$
LE	Less than or equal to	$Z + (NV + NV) = 1$
VC	Overflow clear	V=0
VS	Overflow set	V=1
T	Always true	Always
F	Always false	Never

Program Control (1)

- ***Conditional branches*** are due to tests within the program
  - Logical combinations of the states of the flags

- Branches are examples of ***PC relative addressing***
  - **2's complement displacement value plus <PC>** determines the destination of the branch
  - **Relative addressing** does not require a programmer to specify an absolute memory location

- Unconditional jumps and jumps to **subroutines**
  - JMP <expression>: value of expression is new PC value (EA→PC)
  - JSR <expression> and RTS provide method of accessing frequently used code and returning to starting point

    JSR automatically does the following three things for you!

    SP-4→SP, PC→(SP), EA→PC

Program Control (2)

- **SUB vs. CMP**
  - Both instructions subtract the source from the destination
  - SUB places the result in the destination
  - CMP does not change the operands, but change CCR flags

- Programmer’s Reference Manual describes which Condition Code Flags are affected by each instruction and how they are modified

C++ example:

```
int a = 3, b = 5 ;
if (a == b)
  { Execute this code };
```

Assembly language example:

```
                move.l   #3,D0
                move.l   #5,D1
                cmp.l    D0,D1
                beq      equal
not_equal        {This code executes}
equal            {This code executes}
```

Condition Codes

- Most machine instructions will cause a change to condition code flags
  - A branch is taken when the logical combination of the appropriate condition code flags evaluates to TRUE

- The class of instructions that change program flow are called ***branch instructions***
  - **Bcc** means branch if the appropriate condition code flag is set to 1 Examples:
    - BEQ: branch if the ZERO FLAG = 1 (result is zero)
    - BNE: branch if the ZERO FLAG = 0 (non-zero result)
    - BGE: branch if the result is greater or equal : NV + ~N~V
      - **CMP.B #10, D0 when <D0> = 20, then 20-10 is not negative and not overflow (~N~V)**
      - **CMP.B #$-7F, D0 when <D0> = 6F, then, 6F-(-7F) is overflow and negative. (NV)**

Branch Instructions

- The following table summarizes the branch conditions and how they are tested:

Bcc	MEANING	LOGICAL TEST
BCC	Branch if CARRY is clear	$C = 0$
BCS	Branch if CARRY is set	$C = 1$
BEQ	Branch if result equals zero	$Z = 1$
BNE	Branch if result does not equal zero	$Z = 0$
BGE	Branch if result is greater or equal	$N * V + \overline{N} * \overline{V} = 1$
BGT	Branch if result is greater than	$N * V * \overline{Z} + \overline{N} * \overline{V} * \overline{Z} = 1$
BHI	Branch if result is HI	$\overline{C} * \overline{Z} = 1$
BLE	Branch if result is less than or equal	$Z + N * \overline{V} + \overline{N} * V = 1$
BLS	Branch if result is low or the same	$C + Z = 1$
BLT	Branch if result is less than	$N * \overline{V} + \overline{N} * V = 1$
BMI	Branch if result is negative	$N = 1$
BPL	Branch if result is positive	$N = 0$
BVS	Branch if the resulted caused an overflow	$V = 1$
BVC	Branch if no overflow resulted	$V = 0$

Branch Instructions (2)

- How can you keep track of the sense of branches?
- Let’s consider the possible relationships between registers D0 and D1 for

  CMP D0,D1 (D1 – D0→ CCR flag)

- The following table shows which branch ***will be taken:****

Relationship	Signed numbers	Unsigned numbers	Comments
`D1 < D0`	BLT	BCS	BCS = Branch on carry set
`D1 <= D0`	BLE	BLS	BLS = Branch on Low or same
`D1 = D0`	BEQ	BEQ	
`D1 <> D0`	BNE	BNE	
`D1 > D0`	BGT	BHI	BHI = Branch on High
`D1 >= D0`	BGE	BCC	BCC = Branch on carry clear

*** http://www.easy68k.com/paulrsm/doc/trick68k.htm**

Thanks to spring quarter 2009 CSS 422 student Mike Kromarek for finding this

Calculating Branches

- How does a processor “take a branch?”
  - Program counter (PC) contains the address of the ***next instruction to be executed*** as the current instruction is being decoded
- Form of instruction: **Bcc** ***displacement***
- Operand of a branch instruction is a **signed** **8-bit** or **16-bit** **offset**
  - Called a ***displacement***
  - If the displacement value is more than a word (two bytes), then you cannot use branches
- If the branch test condition evaluates to be true
  - <PC> + displacement --> PC
  - Becomes the address of the next instruction

0	**1**	**1**	**0**	**condition**	**8-bit displacement**
If 8-bit = 00, then it is an 16-bit displacement					

Analyzing the Branch Code

```
34  00000432 2080          TEST_LOOP:  MOVE.L   D0,(A0)
35  00000434 B090                      CMP.L    (A0),D0
36  00000436 6700000E                  BEQ      ADDR_OK
37  0000043A 3888          NOT_OK:     MOVE.W   A0,(A4)
38  0000043C 5213                      ADDQ.B   #1,(A3)
39  0000043E 0C130004                  CMPI.B   #MAX_CNT,(A3)
40  00000442 6700000A                  BEQ      DONE
41  00000446 5888          ADDR_OK:    ADDQ.L   #INC_ADDR,A0
42  00000448 B2C8                      CMPA     A0,A1
43  0000044A 6CE6                      BGE      TEST_LOOP
44  0000044C 60DA                      BRA      NEXT_TEST
45  0000044E 4E722700      DONE:       STOP     #EXIT_PGM
```

Instruction code in memory

Instruction address in memory

Line numbers added by listfile

- The instruction, BGE TEST_LOOP has the instruction code: 6CE6
- <PC>=0000044C (not 0000044A!!)
  - The displacement = E6
  - 044C + E6 = 432
- Why? E6 = 1A's 2's complement
- Therefore, 44C - 1A = 432
- If the code is 6C00 and follows 00E6, then it is 16-bit displacement, and it is positive (please check the manual for more info).

Loop Constructs in Assembly

- C++ has built-in constructs for changing the program flow
  - If/Else, While, Do/While, Switch, For, Function calls
- Assembly language requires that we **build our own constructs**
  - Branch, Jump, Jump to Subroutine (Function call)
- General rule
  - Pair an instruction that either TESTS A CONDITION or CAUSES A VARIABLE TO CHANGE with the proper ***branch on result*** instruction
- Objective
  - **Set the condition code flags and then test their value** with the branch instruction

"FOR" Loop Example

- C++ example:

  for (int counter = 1 ; counter < 10 ; counter++)

  { Execute these statements }

- Assembly language example:

```
                move.l  #1,D0           *D0 is the counter
                move.l  #10,D1          *D1 holds terminal value
for_loop        cmp.b   D0,D1           *Do the test
                beq    next_code        *Are we done yet?
                { Execute some other loop instructions}
                addq.l  #1,D0           *Increment the counter
                bra     for_loop        *Go back
next_code       { Execute the instructions after the loop }
```

Stack-based Operations – Review

- 68k architecture automatically manages the memory stack as a Last-in/First-out (LIFO) data structure
  - Registers A7 or SP or A7' implement the stack pointer
- **Stack pointer should always be initialized as one of the first operations of setting-up the environment, otherwise it crashes**
  - Stack is necessary if the program includes subroutines
- Stack is normally initialized to the highest value in RAM
  - "**Grows**" from higher memory **down** towards lower memory
- PUSH and POP operations use auto-incrementing or auto-decrementing
  - **PUSH**
    - Decrease the stack pointer and place an item on the stack
    - Address register indirect with **pre-decrement**
  - **POP**
    - Remove an item from the stack and increase the stack pointer
    - Address register indirect with **post-increment**

Stack-based Operations – Review

During a PUSH operation the stack grows towards lower memory addresses (decrease)

Example:

    MOVE.B   D0,-(SP)

will place a byte of data on the stack.

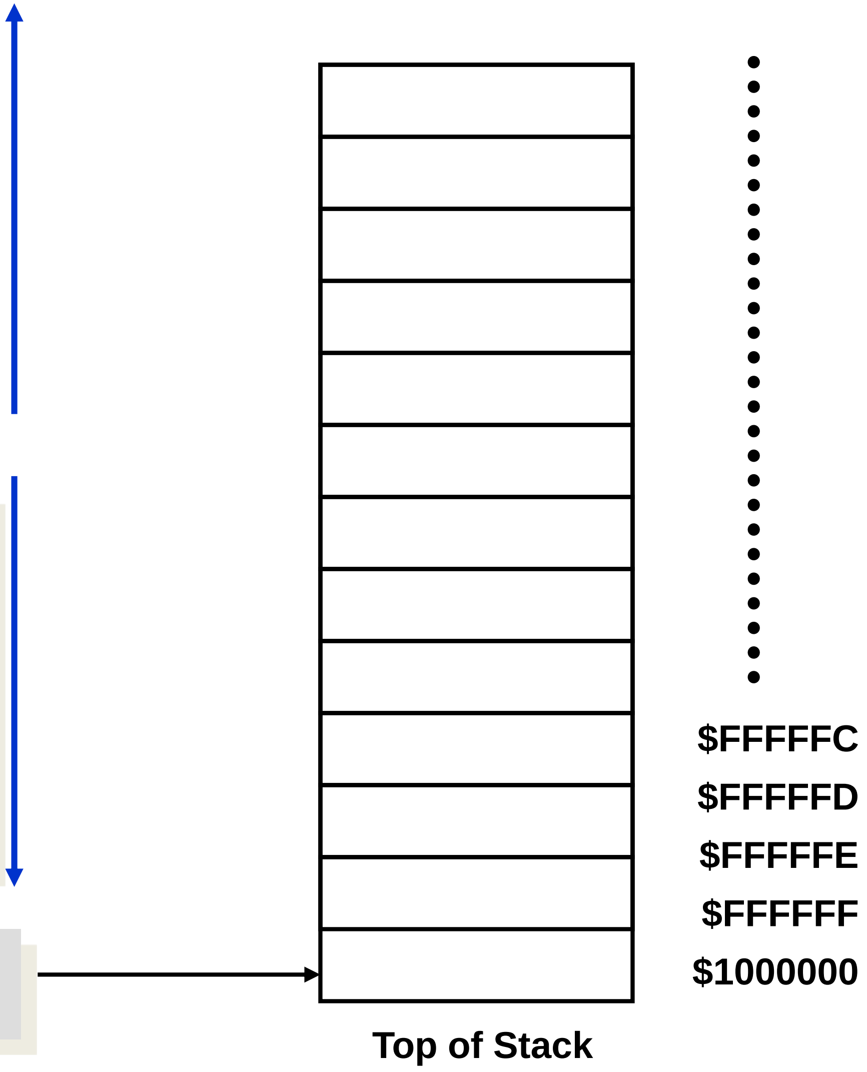
During a POP operation the stack shrinks towards higher memory addresses

Example:

    MOVE.B   (SP)+,D0

will move a byte of data from the stack to D0.

Initial value of the stack pointer, SP
The TOP of the stack



Subroutines

- Subroutines in assembly language ***are procedures or function calls in C***

- JSR instruction
  1. **PUSHES** the **long word address of the next instruction** onto the stack in three steps: **SP-4→SP, PC→(SP), EA→PC**
  2. Current PC value is stored in SP (PC→(SP))
  3. Jumps to the location specified in the operand (EA→PC)

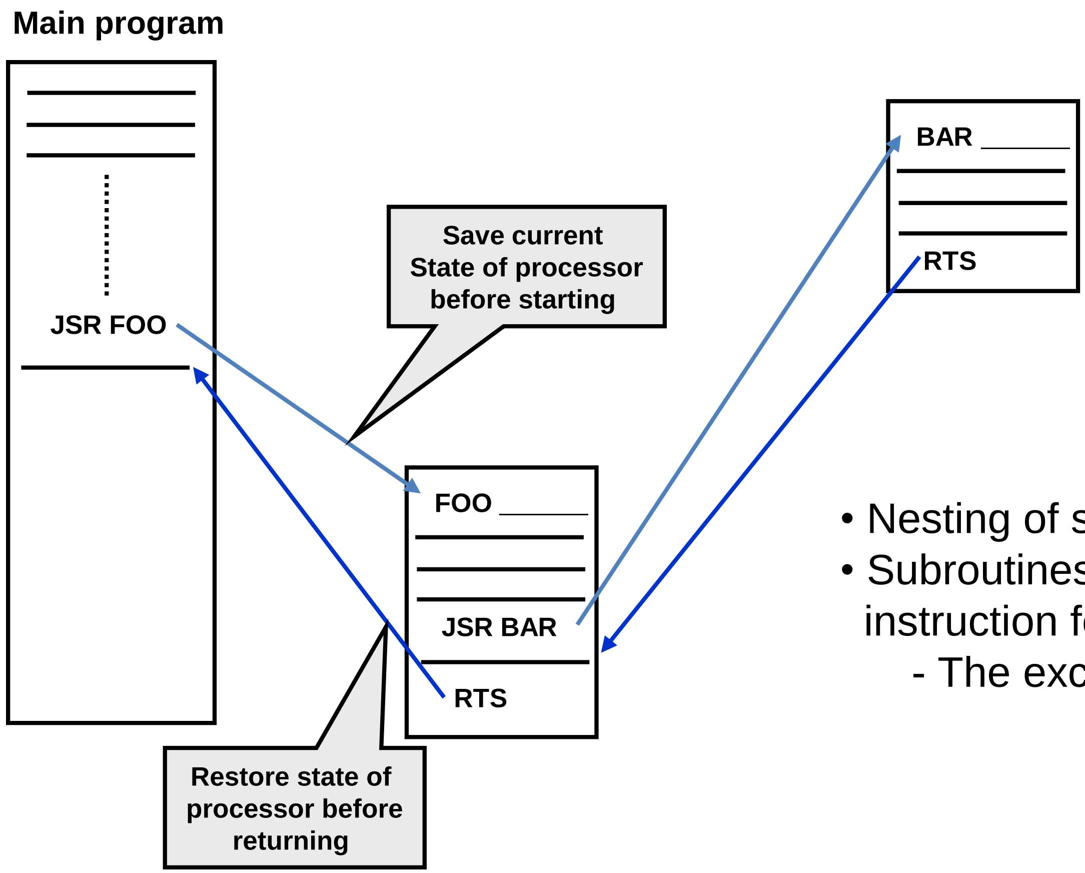
- RTS instruction
  1. Must be used at the end of the subroutine
  2. Causes the stack to **POP** the return address back into the PC (SP)→PC, SP+4→SP

Subroutines (2)

- It is ***up to the programmer (the assembler will not do this for you)*** to
  1. Make sure that **all stack pushes and pops line up**
  2. All **resources** used by the subroutine (registers and memory) are properly **saved before** the subroutine uses them and **restored when** the subroutine **returns**
  3. Decide on a mechanism for **parameter passing** between the subroutines and the main program

- It is generally unnecessary to save everything, but it won't hurt to save all
  - Can eliminate saving all the registers once you have written the subroutine
  - It is a good idea to devise a consistent method of parameter passing
- To PUSH registers onto the stack on entry to the subroutine:
  - `MOVEM   <register list>,-(SP)`
- To POP registers from the stack on exit from the subroutine:
  - `MOVEM    (SP)+,<register list>`

Subroutines (3)



Main program
JSR FOO
Save current State of processor before starting
FOO
JSR BAR
RTS
Restore state of processor before returning
BAR
RTS

- Nesting of subroutines is OK
- Subroutines should return to the instruction following the JSR
  - The exception is for jump tables

- Subroutines require careful stack management
  - Return address must be at the top of the stack just prior to the RTS

Tips on Subroutine Design

- Why need?
  - Same block of code can be reused many times
  - Alternative is to have all the code in-lined
  - Efficient coding
- Subroutine guidelines
  1. Must **have the stack established (SP has a default value)**
  2. Locate the subroutines after the main program
  3. Each subroutine should have a comment block header listing
     - Subroutine name
     - What it does
     - Registers used and saved
     - Parameters input and returned
  4. **First instruction line of the subroutine must have label with name**
  5. **Save registers used on entry**
  6. **Restore registers on exit**
  7. Return to the point in program where subroutine was called from
     - Don’t jump somewhere else. **Always RTS**
     - Nesting of subroutines is permitted

MOVEM <register list>, <ea>
MOVEM <ea>, <register list>

- Transfers the contents of a **group of registers** specified by a list. The list of registers is defined as **Ai-Aj/Dp-Dq**.
- MOVEM operates **only** on **words** or **longwords**.
- Use assembler directive REG to bind a name to the list

```
MOVEM.L A0-A6/D0-D7, -(SP)
```

```
GROUP REG A0-A6/D0-D7
MOVEM.L GROUP, -(SP)
```

Pseudo Op Codes – Assembler Directives

- **REG - Register Range**
  - Allows a list of registers to be defined
  - Format: <label> REG <register list>
  - Registers may be specified as a single register, An or Dn, separated by slashes, i.e. A1/A5/A7/D1/D3
  - Register ranges may also be specified, i.e., A0-A3
  - Thus:

```
save_reg        REG     A0-A3/A5/D0-D7
```

  - Refer to the MOVEM instruction in your programmer's manuals
  - Will go over again for subroutine

MOVEM <register list>, <ea>
MOVEM <ea>, <register list>

- Mostly used with **pre-decrementing** (registers to memory) and **post-incrementing** (memory to registers)
- Frequently used to **save working registers** on entering a subroutine and to **retrieve** them on exiting the subroutine

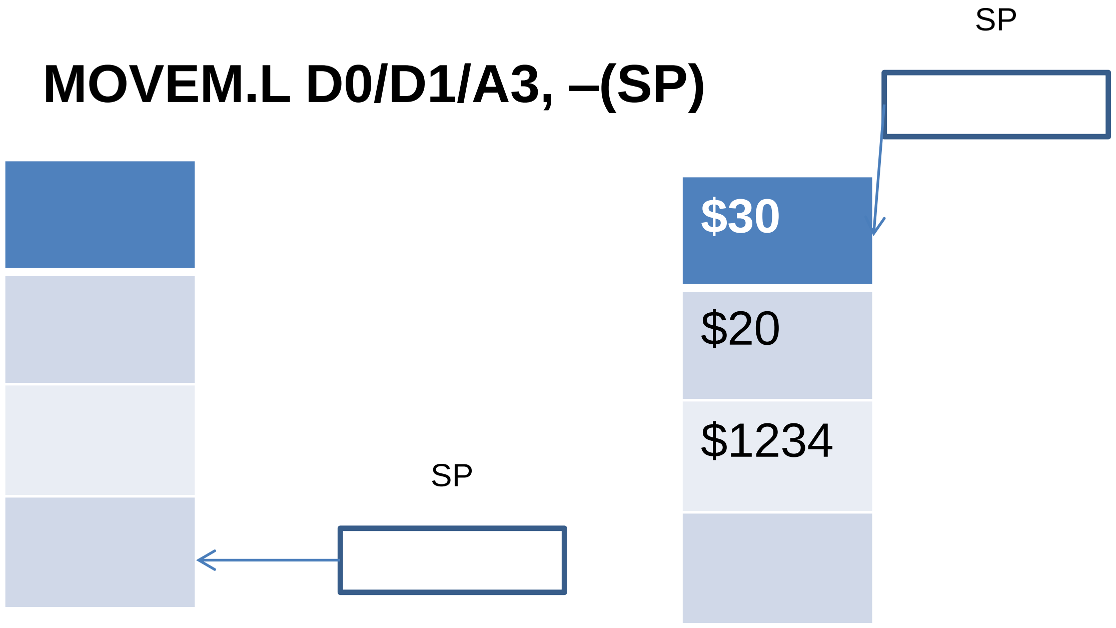
```
MOVEM.L  A0-A3/D0-D7, -(A7)  *copies all working registers to stack
MOVEM.L (A7)+, A0-A3/D0-D7  *Restore the registers
```

WHY do you need to save the registers to the stack when entering a subroutine??

- 68K Always **writes** to memory in the order of **A7 to A0, D7 to D0**
- 68K Always **reads** from memory in the order of **D0 to D7, A0 to A7**

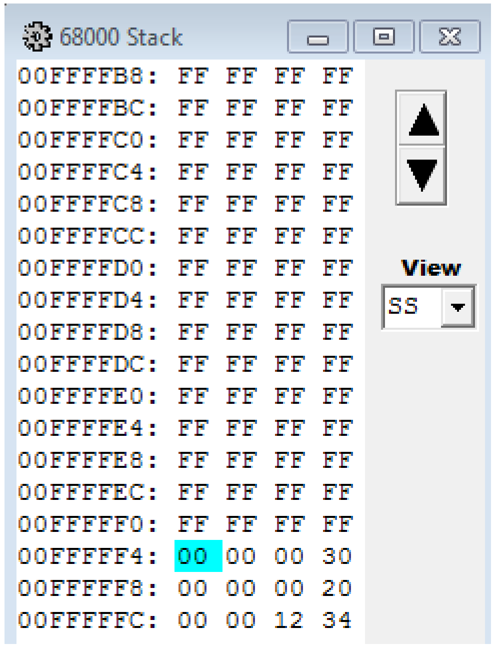
E.g., D0 has $30, D1 has $20, A3 had $1234

MOVEM.L D0/D1/A3, –(SP)



Before executing

After executing



```
00FFFFB8:  FF FF FF FF
00FFFFBC:  FF FF FF FF
00FFFFC0:  FF FF FF FF
00FFFFC4:  FF FF FF FF
00FFFFC8:  FF FF FF FF
00FFFFCC:  FF FF FF FF
00FFFFD0:  FF FF FF FF
00FFFFD4:  FF FF FF FF
00FFFFD8:  FF FF FF FF
00FFFFDC:  FF FF FF FF
00FFFFE0:  FF FF FF FF
00FFFFE4:  FF FF FF FF
00FFFFE8:  FF FF FF FF
00FFFFEC:  FF FF FF FF
00FFFFF0:  FF FF FF FF
00FFFFF4:  00 00 00 30
00FFFFF8:  00 00 00 20
00FFFFFC:  00 00 12 34
```

```
00001000                       * Put program code here
00001000  303C 0030                MOVE #$30, D0
00001004  323C 0020                MOVE #$20, D1
00001008  367C 1234                MOVEA #$1234, A3
0000100C  48E7 C010                MOVEM.L D0/D1/A3, -(SP)
```

Subroutine Example (1)

Main

```
…
stack   EQU $7000   *sp initial
…
        ORG   $400
start   LEA   stack, SP
…
$ 424   JSR do_test
$ 426   MOVE.B  …
```

PC = $ 424 → $**426**

SP= $7000

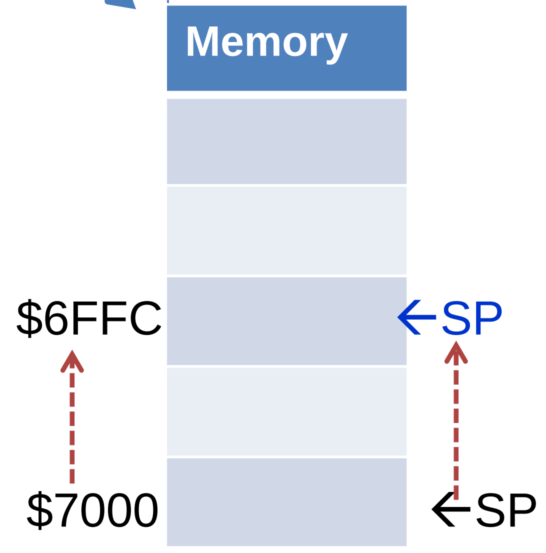
A3= $4500

D1= $0012

SP-4→SP
PC→(SP)
EA→PC

Subroutine

```
*******************************************
Subroutine do_test ($480)
Description: ______________
*******************************************
do_test MOVEM.W A3/D1, -(SP)
        MOVE.B….
…
        MOVEM.W (SP)+, A3/D1
exit    RTS
```



Subroutine Example (1)

Main

```
…
stack   EQU $7000   *sp initial
…
        ORG  $400
start   LEA  stack, SP
…
$ 424   JSR do_test
$ 426   MOVE.B …
```

Subroutine

```
*************************************************
Subroutine do_test ($480)
Description: _______________
*************************************************
do_test MOVEM.W A3/D1, -(SP)
        MOVE.B….
…
        MOVEM.W (SP)+, A3/D1
exit    RTS
```

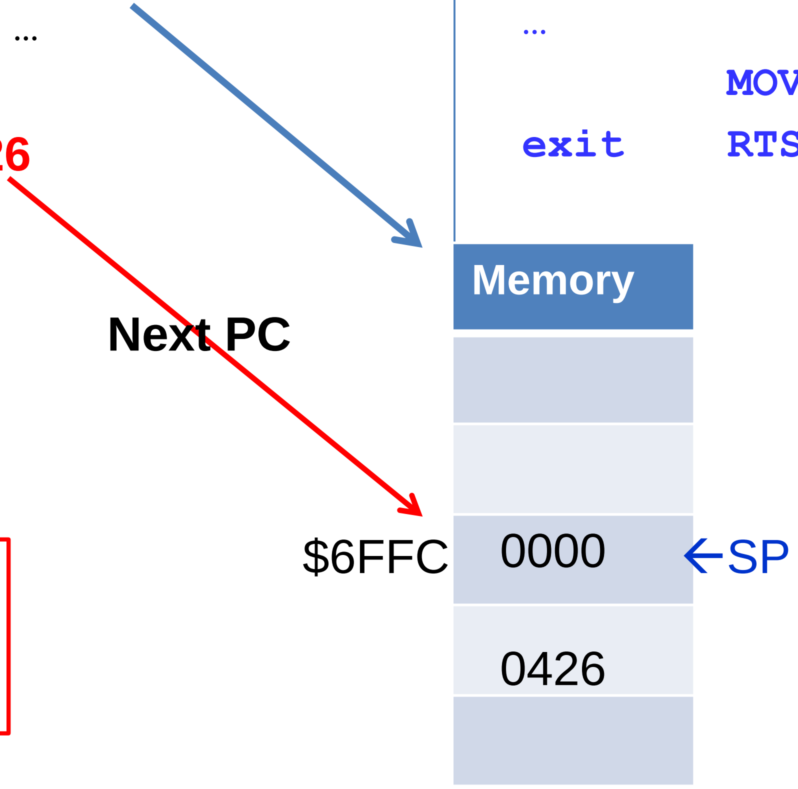
PC = \$ 424 → \$**426**

SP= \$6FFC

A3= \$4500

D1= \$0012

SP-4→SP
PC→(SP)
EA→PC



Subroutine Example (1)

Main

```
…
stack   EQU $7000   *sp initial
…
        ORG   $400
start   LEA   stack, SP
…
$ 424   JSR do_test
$ 426   MOVE.B  …
```

PC = $ 480

SP= $6FFC

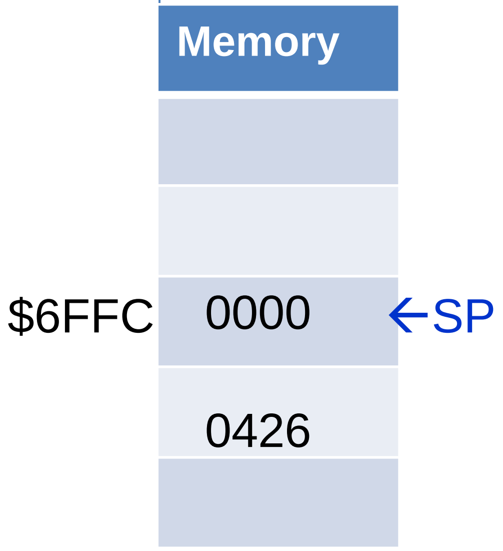
A3= $4500

D1= $0012

SP-4→SP
PC→(SP)
EA→PC

Subroutine

```
*******************************************
Subroutine do_test ($480)
Description: ______________
*******************************************
do_test MOVEM.W A3/D1, -(SP)
        MOVE.B….
…
        MOVEM.W (SP)+, A3/D1
exit    RTS
```



Subroutine Example (2)

Main

```
…
stack   EQU $7000   *sp initial
…
        ORG   $400
start   LEA   stack, SP
…
$ 424   JSR do_test
$ 426   MOVE.B  …
```

PC = $ 480

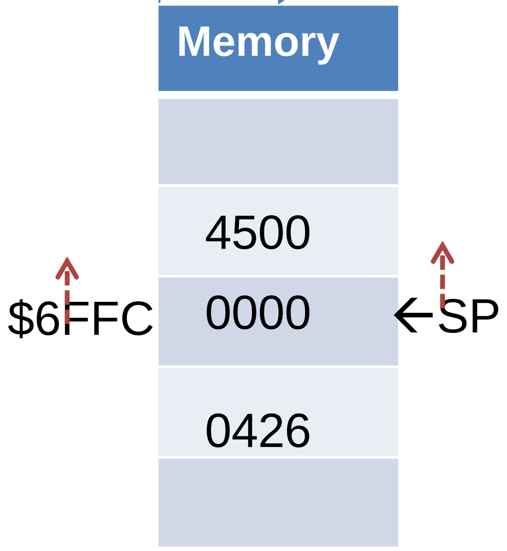
SP= $6FFC

A3= $4500   A3= free

D1= $0012

Subroutine

```
*******************************************
Subroutine do_test ($480)
Description: _______________
*******************************************
do_test MOVEM.W A3/D1, -(SP)
        MOVE.B… .
…
        MOVEM.W (SP)+, A3/D1
exit    RTS
```



Subroutine Example (2)

Main

```
…
stack   EQU $7000   *sp initial
…
        ORG   $400
start   LEA   stack, SP
…
$ 424   JSR do_test
$ 426   MOVE.B …
```

PC = $ 480

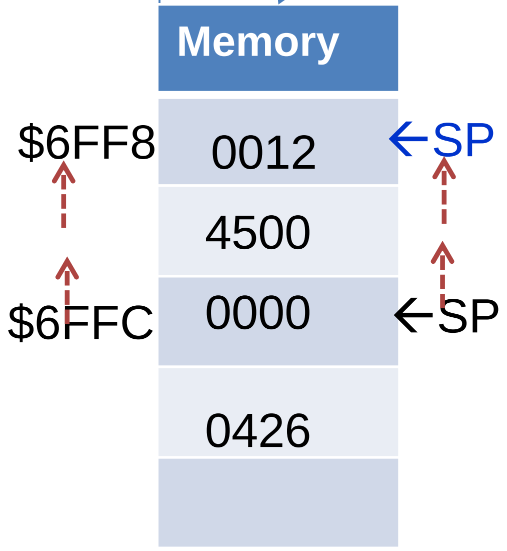
SP= $6FFA

A3= free

D1= $0012   D1= free

Subroutine

```
*******************************************
Subroutine do_test ($480)
Description: ______________
*******************************************
do_test MOVEM.W A3/D1, -(SP)
        MOVE.B… .
…
        MOVEM.W (SP)+, A3/D1
exit    RTS
```



Subroutine Example (2)

Main

```
…
stack   EQU $7000  *sp initial
…
        ORG  $400
start   LEA  stack, SP
…
$ 424   JSR do_test
$ 426   MOVE.B  …
```

PC = $ 480

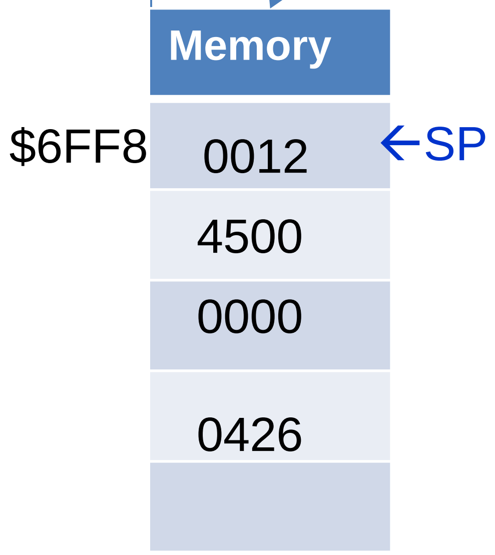
SP= $6FF8

A3= free

D1= free

Subroutine

```
*******************************************
Subroutine do_test ($480)
Description: ______________
*******************************************
do_test MOVEM.W A3/D1, -(SP)
        MOVE.B… .
…
        MOVEM.W (SP)+, A3/D1
exit    RTS
```



Subroutine Example (3)

Main

```
…
stack   EQU $7000   *sp initial
…
        ORG   $400
start   LEA   stack, SP
…
$ 424   JSR do_test
$ 426   MOVE.B  …
```

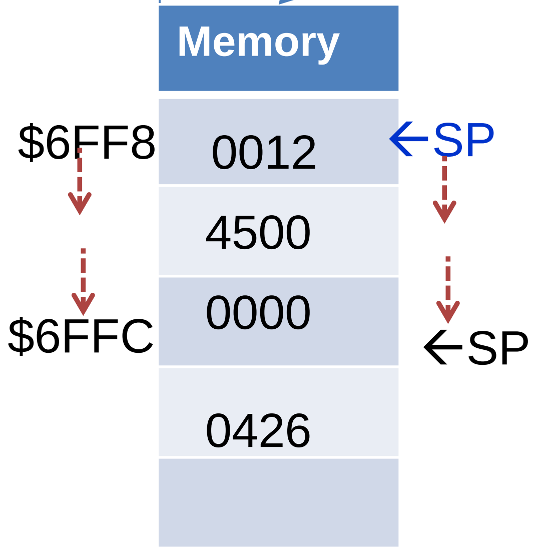
Subroutine

```
*******************************************
Subroutine do_test ($480)
Description: _______________
*******************************************
do_test MOVEM.W A3/D1, -(SP)
        MOVE.B….
…
        MOVEM.W (SP)+, A3/D1
exit    RTS
```

SP= $6FF8

A3= $4500 A3= …

D1= $0012 D1= …



Subroutine Example (3)

Main

```
…
stack   EQU $7000   *sp initial
…
        ORG  $400
start   LEA  stack, SP
…
$ 424   JSR do_test
$ 426   MOVE.B  …
```

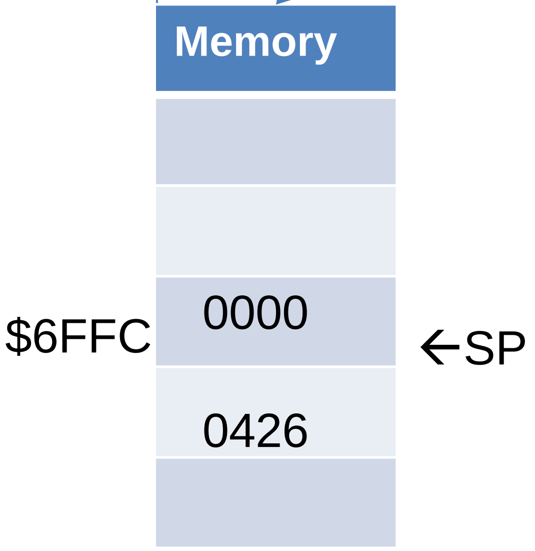
Subroutine

```
*************************************************
Subroutine do_test ($480)
Description: ______________
*************************************************
do_test MOVEM.W A3/D1, -(SP)
        MOVE.B….
…
        MOVEM.W (SP)+, A3/D1
exit    RTS
```

SP= $6FFC

A3= $4500

D1= $0012



Subroutine Example (4)

Main

```
…
stack   EQU $7000  *sp initial
…
        ORG  $400
start   LEA  stack, SP
…
$ 424   JSR do_test
$ 426   MOVE.B …
```

PC = $ 426

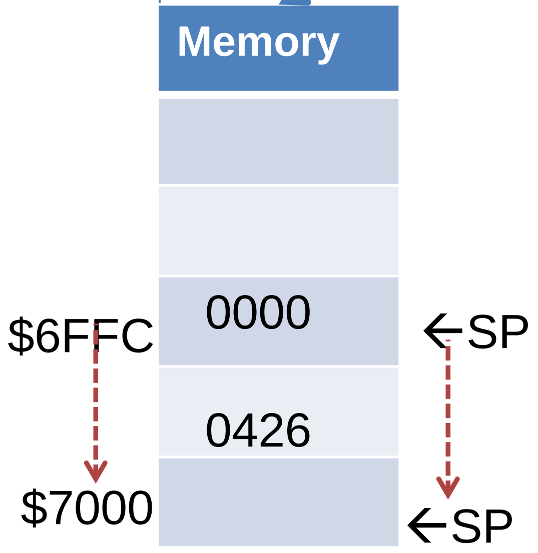
SP= $6FFC

A3= $4500

D1= $0012

Subroutine

```
******************************************
Subroutine do_test ($480)
Description: _______________
******************************************
do_test MOVEM.W A3/D1, -(SP)
        MOVE.B… .
…
        MOVEM.W (SP)+, A3/D1
exit    RTS
```



(SP)→PC
SP+4→SP

Subroutine Example (4)

Main

```
…
stack   EQU $7000  *sp initial
…
        ORG  $400
start   LEA  stack, SP
…
$ 424   JSR do_test
$ 426   MOVE.B  …
```

PC = $ 426

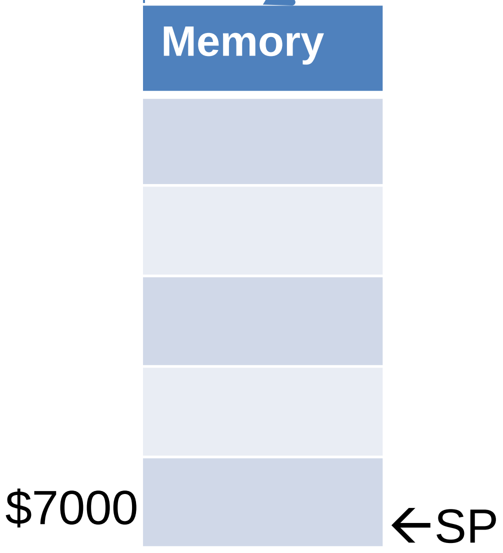
SP= $7000

A3= $4500

D1= $0012

Subroutine

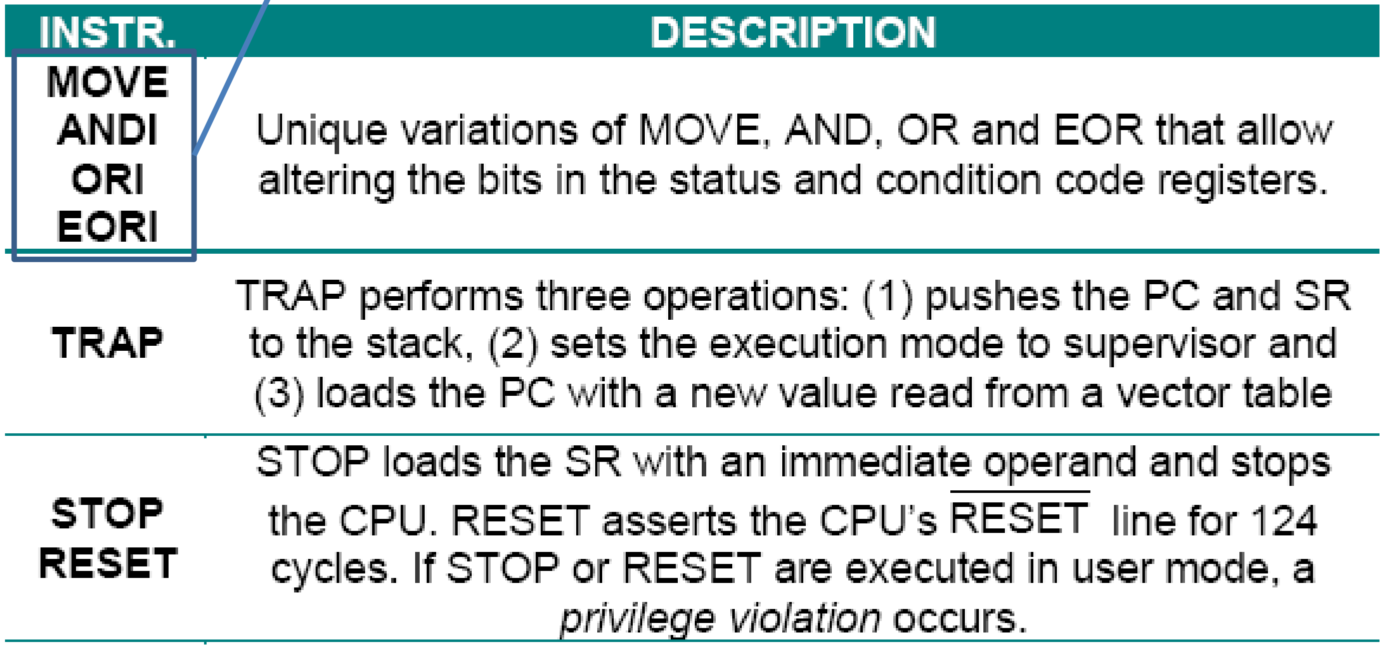
```
************************************************
Subroutine do_test ($480)
Description: _______________
************************************************
do_test MOVEM.W A3/D1, -(SP)
        MOVE.B….
…
        MOVEM.W (SP)+, A3/D1
exit    RTS
```



(SP)→PC
SP+4→SP

68000 System Control

to CCR, to SR



INSTR.	DESCRIPTION
MOVE ANDI ORI EORI	Unique variations of MOVE, AND, OR and EOR that allow altering the bits in the status and condition code registers.
TRAP	TRAP performs three operations: (1) pushes the PC and SR to the stack, (2) sets the execution mode to supervisor and (3) loads the PC with a new value read from a vector table
STOP RESET	STOP loads the SR with an immediate operand and stops the CPU. RESET asserts the CPU's $\overline{\text{RESET}}$ line for 124 cycles. If STOP or RESET are executed in user mode, a *privilege violation* occurs.

http://easy68k.com/easy68kexamples.htm

Summary of Data Transfer Instructions

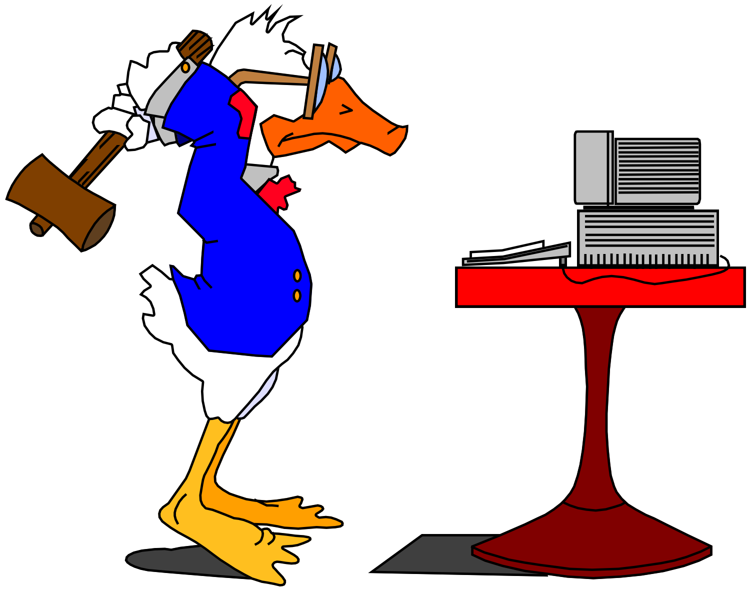
LEA	Load effective address
MOVE	Move data
MOVEA	Move address
MOVEM	Move multiple registers
MOVEP	Move peripheral data
MOVEQ	Move quick
SWAP	Swap register halves

Summary of Arithmetic Instructions

ADD	Add binary
ADDA	Add address
ADDI	Add immediate
ADDQ	Add quick
CLR	Clear operand
CMP	Compare
CMPI	Compare immediate
DIVS	Divide signed number
DIVU	Divide unsigned
MULS	Multiply signed number
NEG	Negate
SUB	Subtract binary
SUBA	Subtract Address
SUBI	Subtract immediate
SUBQ	Subtract quick
EXT	Extend sign

Summary of Logical and Shift Instructions

AND	Logical AND
ANDI	AND immediate
OR	Logical OR
ORI	OR immediate
EOR	Exclusive OR
EORI	Exclusive OR immediate
NOT	Logical complement
ASL	Arithmetic shift left
ASR	Arithmetic shift right
LSL	Logical shift left
LSR	Logical shift right
ROL	Rotate left
ROR	Rotate right



Summary of Bit Manipulation Instructions

BCHG	Bit change
BCLR	Bit clear
BSET	Set bit
BTST	Test bit

Summary of Program Control Instructions

Bcc	Branch on state of conditional code (cc) flag
BRA	Branch always
BSR	Branch to subroutine
JMP	Unconditional jump
JSR	Jump to subroutine
RTR	Return and restore
RTS	Return from subroutine

Summary of Privileged Instructions

ANDI SR	And immediate to Status Register
EORI SR	Exclusive OR immediate to Status Register
MOVE SR	Move to/from the Status Register
MOVE USP	Move to/from USP
RESET	Reset the processor
RTE	Return from exception
STOP	Stop the processor
CHK	Check register
ILLEGAL	Force an illegal instruction exception
TRAP	Trap call
TRAPV	Trap on overflow
ANDI CCR	AND immediate to condition code register
ORI CCR	OR immediate to condition code register
EORI CCR	Exclusive OR immediate to CCR
MOVE CCR	Move to/from CCR
NOP	No operation - Do nothing