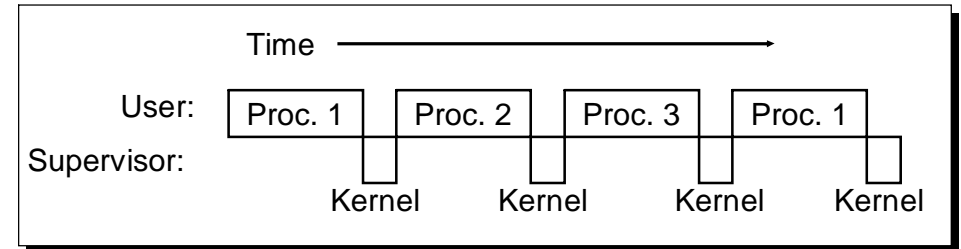


# 68000 Programming Model

Supervisor, user modes,  
registers, execution of instructions

# Multitasking Operating System

CPU time is shared among processes. In order to switch from one process to another, operating system kernel (task switcher) is activated. It performs the switching and restarts the next task.

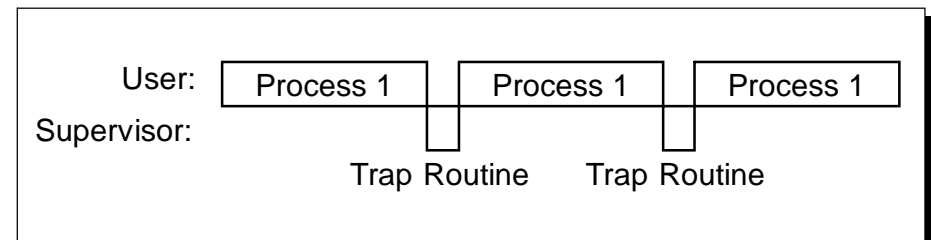


# User and Supervisor Modes

- Building complex software systems, it is necessary to have some built-in hardware security features. In 68K, two modes exist:
  - User Mode: This privilege mode is used by ordinary users. It has only restricted access to the CPU's registers and/or instructions.
  - Supervisor Mode: it is intended for expert users or the operating system. It has greater or unrestricted access to the CPU.

# Single Task Calling OS Functions

Trap routines are special functions provided by the operating system. Calling these routines means switching from one operating mode to another. They usually handle I/O devices.



## CPU Registers

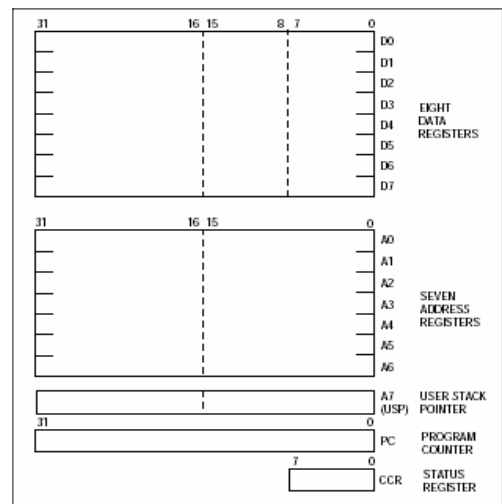
- To do its work, CPU needs registers. All CPU need registers to operate.
- Registers are used to
  - store temporary information (like memory)
  - have associated operations (like accessing some memory via an index register).
  - perform some operations inside te CPU in a fast way.
  - store internal data inside the CPU without showing the data to users.

## Program Counter

- PC is a register inside CPU
- PC holds address of next instruction to be fetched and executed.
- Contents of PC can not be directly accessed by programs. The content is manipulated by
  - jump, branch and jump subroutine instructions
  - interrupts (traps, exceptions)

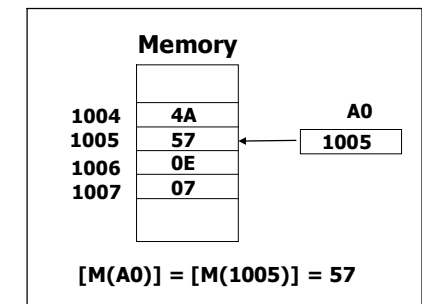
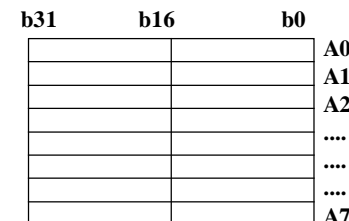
## 68000 Programming Model

- There are 8 general purpose data registers D0-D7
- There 7 general purpose address registers A0-A6 and 2 stack pointers (USP and SSP) for user and supervisor modes.
- Status register contains operating flags such as carry, negative flags etc.

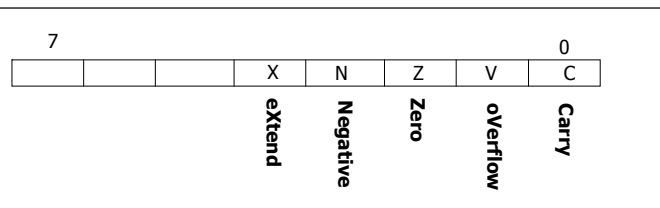


## Address Registers

- Eight (8) address registers of 32 bits.
- Information in an address register points to a location in memory.
- Special one: A7 is used as stack pointer.



# Condition Code Register



The CCR is updated to reflect the result of the operation.

- Z=1 if the result is 0
- C=1 if there is carry-out from MSB
- V=1 if there is overflow
- N=1 if the result is negative

# Addressing Modes

- Addressing mode refers to the way in which an operand address can be specified. The MC68000 has the following address modes
  1. Data Register Direct
  2. Address Register Direct
  3. Register Indirect
  4. Post-increment Register Indirect
  5. Pre-decrement Register Indirect
  6. Register Indirect with Offset
  7. Indexed Register Indirect with Offset
  8. Absolute Short
  9. Absolute Long
  10. Immediate
  11. PC Relative with Offset
  12. PC Relative with Index and Offset

# Addressing Modes

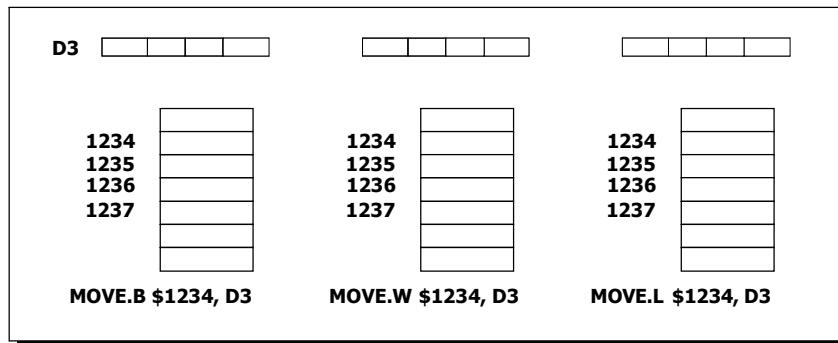
How to access memory?

# Bytes, Words and Longwords

- MC68000 supports 8-, 16-, and 32-bit operations.
- The memory is byte-addressed: when MC68000 reads a word, it gets two bytes from the consecutive locations.
- All word and longword accesses must be to an even address

`MOVE.L 11,D1` is invalid

## Bytes, Words and Longwords



**MOVE.B \$1235, D3** instruction is invalid. Destination address must be on even boundary.

## Unsigned Integers

- Larger range of integers than 2's complement
- Range of a 16-bit unsigned integer:  
 $0 - 2^{16} - 1$  (65535)
- Question: How are arithmetic operations performed on unsigned numbers?
- Answer: No difference between signed and unsigned arithmetic. In other words, the sign bit is interpreted by the programmer.

## Byte Arithmetic Applied to Longword Register

```
D0:12345678 00010010001101000101011001111000
D1:13579BDF 00010011010101111001011011011111
ADD.B D0,D1 00010011010101111001011001010111
```

Unaffected

X N Z V C = ?

## Unsigned Integers

Question: How do we check whether the result is out of range?

- Signed: check V bit in CCR
- Unsigned: check C bit in CCR

Unsigned Arithmetic

ADD.B D1, D2

BCS ERROR

:

ERROR

Signed Arithmetic

ADD.B D1, D2

BVS ERROR

:

ERROR

## The 68000 Address Bus

- The 68000 has 32-bit address registers and program counter (PC)
- Because of packaging, Address Bus lines A24 to A31 are not used
- The A00 is used to select the byte or word addressing
- The 68000 has actually 24-bit addressing !  
68K can access only  $2^{24}$  (16M) bytes in memory
- 68020 has a full 32-bit address bus
  - How many bytes in memory can be accessed?

## Using Address Registers


- Data registers support byte, word, and longword operations.
- Address registers support word and longword operations.
- 32-bit address stored in an address register is a “single entity”
- Effect of a word operation to the content of an address register is a longword operation.

## Using Address Registers

- Address Registers: A0 to A7.
- Instructions that modify and use Address Registers are
  - MOVEA
  - ADDA
  - SUBA
  - CMPPA
- Important: They do not affect the CCR.

## Using Address Registers

- All MC68000's addresses are sign-extended to 32 bits for word operations.

`ADDA.L # $FFF4, A0`  `A0 = A0 + $0000FFF4`

words operation sign extends the value to 32bit.

`ADDA.W # $FFF4, A0`  `A0 = A0 + $FFFFFFF4`

## Data Register Direct

BEFORE

D1	\$12345678
D2	\$FEDCBA98

`move.b d1,d2`



AFTER

D1	\$12345678
D2	\$FEDCBA78

`move.w d1,d2`



D1	\$12345678
D2	\$FEDC5678

`move.l d1,d2`



D1	\$12345678
D2	\$12345678

68000 lecture notes

21

## Register Indirect

BEFORE

D1	\$12345678
D2	\$FEDCBA98
A2	\$00010000
A3	\$00011000

\$00010000	\$33	\$44
\$00010002	\$55	\$66

`move.b (a2),d2`



D2	\$FEDCBA33
----	------------

`move.w (a2),d2`



D2	\$FEDC3344
----	------------

`move.l (a2),d2`



D2	\$33445566
----	------------

68000 lecture notes

23

## Address Register Direct

BEFORE

D1	\$12345678
D2	\$FEDCBA98
A2	\$00010000
A3	\$00011000

`move.b d1,a3`



ILLEGAL (WHY?)

`move.w d1,a3`



A3	\$00015678
----	------------

`move.l d1,a3`



A3	\$12345678
----	------------

68000 lecture notes

22

## Post-increment Register Indirect

BEFORE

D1	\$12345678
D2	\$FEDCBA98
A2	\$00010000
A3	\$00011000

\$00010000	\$33	\$44
\$00010002	\$55	\$66

`move.b (a2)+,d2`



D2	\$FEDCBA33
A2	\$00010001

`move.w (a2)+,d2`



D2	\$FEDC3344
A2	\$00010002

`move.l (a2)+,d2`



D2	\$33445566
A2	\$00010004

68000 lecture notes

24

# Pre-decrement Register Indirect

BEFORE

D1	\$12345678	\$0000FFFE	\$11	\$22
D2	\$FEDCBA98	\$00010000	\$33	\$44
A2	\$00010000	\$00010002	\$55	\$66
A3	\$00011000			

```
move.b -(a2), d2
```

D2	\$FEDCBA22
A2	\$0000FFFF

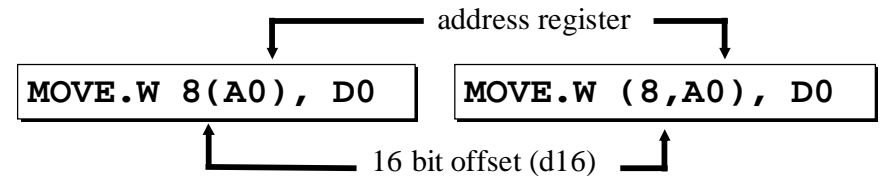
```
move.w -(a2), d2
```

D2	\$FEDC1122
A2	\$0000FFFE

Same as register indirect, except the address register is automatically decremented by the data size (in bytes) before it is used to do the data operation specified in the instruction.

# Register Indirect With Displacement

- Extension of address register indirect with specification of displacement or offset



$$EA \text{ (Effective Address)} = [\text{Address Register}] + d16$$

# Absolute Addressing

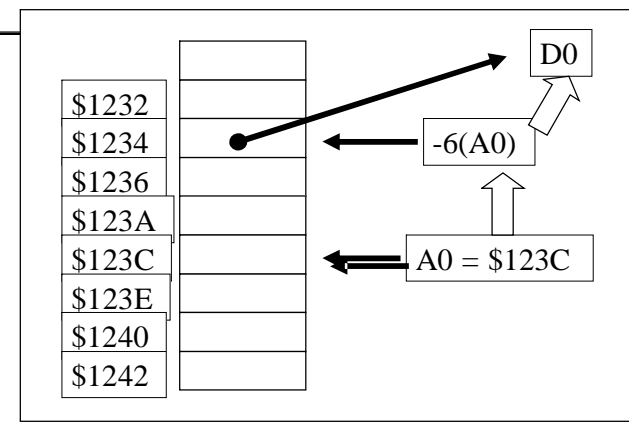
Accesses directly to a memory location

```
MOVE.L D3, $1234 [M($1234)] B [D3(0:31)]
MOVE.W $1234, D3 [D3(0:15)] B [M($1234)]
```

Is "absolute addressing" a good idea?  
 What about program "relocation" in the memory?  
 What about multiprogramming system?

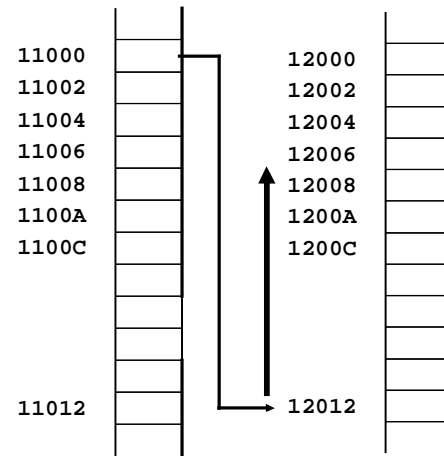
# Register Indirect With Displacement

If A0 = \$123C, what does the following instruction do?  
**MOVE.W -6(A0), D0**



## Example

- Write a program to move 10 consecutive words (word means 2 bytes) of data starting at memory location \$011000 to a consecutive block of memory starting at memory location \$012000, but in reverse order. The program should start at \$01000.



## Stack (FILO) Operations

- Post-increment and pre-decrement modes are used for maintaining stacks.
- A stack is a sequence of memory locations that supports the following two operations
  - Place a new value on the top of a queue of saved values.
  - Take the value at the top of stack of saved values.

## Solution

```
Start      move.b  #10,d0      ; counter=0
           movea.l #11000,a0 ; initialize a0 to point block1
           movea.l #12014,a1 ; initialize a1 to point block2
loop       move.w  (a0)+,-(a1) ; move from block1 to block2
           sub.b   #1,d0      ; decrement loop counter
           bne    loop       ; if d0<>0 then goto loop
```

## Why Stack?

- Stacks are widely used in microprocessor programming
  - for storing subroutine/exception return addresses
  - for passing parameters to/from subroutines
  - for creating space for local variables in subroutines

## Stack in 68K

- Any address register can be a stack pointer

**PUSH**

`move.w d2, -(a3)`

**PULL**

`move.w (a3)+, d2`

a3 points to the last filled location,  
i.e. the top of the stack.

## Register Indirect with Offset d16(An)

D2	\$FEDCBA98
A2	\$00010000

\$00011000	\$77	\$88
\$00011002		

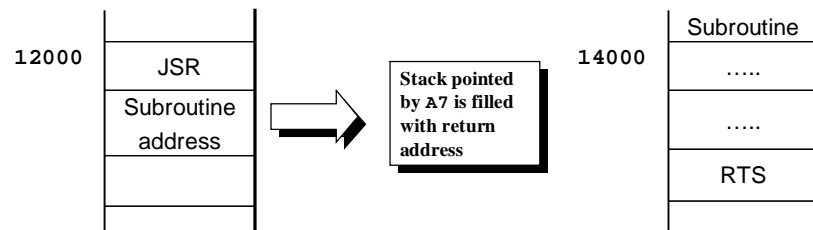
`move.w $1000(a2), d2`

D2	\$FEDC7788
A2	\$00010000

- Sixteen-bit offset (in two's complement) is stored in an additional instruction word following the operand word.
- Offset is added automatically to the contents of **An** to form the effective address of the operand.

## Stack in 68K

- A7** is the Supervisory Stack Pointer (SSP)
- A7'** is the User Stack Pointer (USP)
- These stack pointers are used to store the return address on a `jsr` (jump to subroutine) and a `bsr` (branch to subroutine)



## Indexed Register Indirect with Offset d8(An,Xn)

- Index register **xn** can be a data or an address register, of either `.w` (which will be sign-extended) or `.L` size.
- Offset is limited to 8-bit two's complement numbers.
- This mode is very useful for implementing:
  - Lists of records with multiple data fields
  - Two-dimensional arrays of data elements.
- Eight-bit offset and index register are stored in an extension word that follows the operand word.

## Indexed Register Indirect with Offset d8(An,Xn)

D1	\$12345678
A2	\$00010000

\$00010000	\$33	\$44
\$00010002	\$55	\$66

```
move.l #$1243FFFE,d0
move.w $4(a2,d0.w),d1
```

```
e.a. = [a2] + [d0.w] + $4
      = $10000 + $FFFFFFFE + $4
      = $10002
```

D1	\$33445566
----	------------

## Solution

```
init  movea.l #ARRAY, a0 ; a0 points to ARRAY
      move.w #0,d0 ; d0 points to 1st col
      move.l #0,d1 ; sum of column 1 is 0
      clr.l d2 ; sum of column 2 is 0
      clr.l d3 ; sum of column 3 is 0

loop  add.b 0(a0,d0.w),d1 ; add 1st element
      add.b 1(a0,d0.w),d2 ; add next
      add.b 2(a0,d0.w),d3 ; add last
      addi.w #3,d0 ; next column
      cmpi.w #6,d0 ; end of column ?
      ble loop
```

## Example

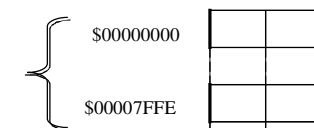
- Sum the columns of the array and compute d1,d2 and d3.

ARRAY	0	1	2	Memory	0	1
	3	4	5		2	3
	6	7	8		4	5
					6	7
				8		

We set **A0** to point at **ARRAY** and **D0** to offset the rows

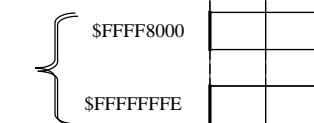
## Absolute Short Addressing Mode

```
move.l $1200,d0
```



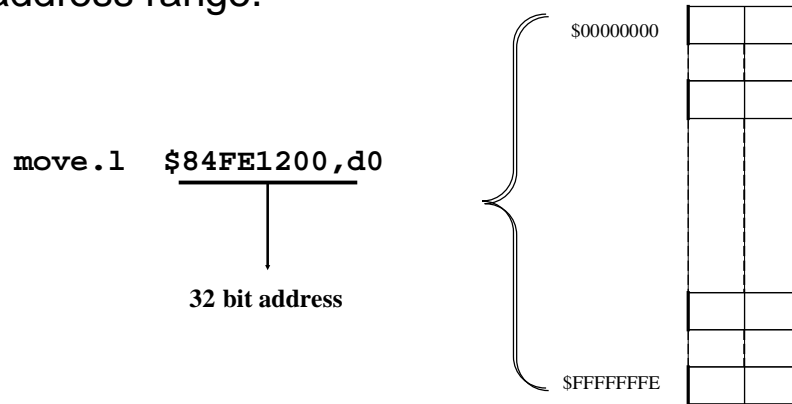
Accesses upper and lower 32Kbytes of memory.  
It occupies less bytes than other memory access instructions.

```
move.l -$1200,d0
```



## Absolute Long Addressing Mode

- It accesses any memory location within the address range.



## PC Relative with Offset $d_{16}$ (PC)

- Operand address is formed by adding the contents of the PC to an offset in the instruction.
- Operand address is determined relative to the instruction being executed.
- For the assembler write
 

```
move.w label(pc), d1
```
- And the assembler computes the offset. Offset can be between `[-32768, 32767]`
- Effective address is `[PC]+offset`
- Cannot use as destination, e.g
 

```
move.b d0, label(pc)
```

## Quick Immediate

```
addq.l #5, d0
```

- The immediate operand occupies three bits embedded inside the operand word. Therefore 001 to 111 represent values 0 to 7, respectively. 000 represents a value of 8 since the value 0 is not useful.
- Quick immediate mode is used with certain instructions:
 

```
ADDQ.W    #1, D1
SUBQ.L    #8, A1
MOVEQ.L   #5, D2
```
- The quick instructions execute faster because they require only one word in memory, and the operation can proceed immediately without requiring any further operand values.
- Many (but not all) assemblers automatically select the quick instead of the regular versions of ADD, SUB, and MOVE.

## Example

```
CLR.L    D1                2 bytes
MOVE.W   BIRTHDATE(PC), D1 4 bytes
ADD.W    AGE(PC), D1       4 bytes
MOVE.L   D1, -(SP)        2 bytes

BIRTHDATE:
    dc.w  1959                2 bytes
AGE:
    dc.w  40                   2 bytes
```

## PC Relative with Indexed Offset $d_8(\text{PC}, X_n.s)$

- The implementation of this mode is similar to indexed register indirect with offset, except that the PC is used instead of an address register **An**.
- Difference: The offset is given as a label in the code.
- The assembler calculates the offset value automatically.
- Restriction: PC relative with offset can only be used to specify source operand addresses. This restriction is to avoid the possibility of self-modifying code.
- Offset can be between  $[-128, +127]$

```
move.w incr(pc,a1.w),d1 * Where offset=label-[PC]
.....
incr: dc.b 16
```

## SPIC: Statically Position Independent Code

```
org $400400
*****
* FIND LARGER OF TWO NUMBERS
*****
* code is not SPIC
* self-modifying
* not reentrant
*****
larger move.l arg1,d1 1st argument
        cmp.l  arg2,d1  compare arguments
        bgt.s  xx      skip if arg1 larger
        move.l arg2,d1  else put larger_arg in d1
xx      lea    biggr,a1  put larger_arg in "bigger"
        move.l d1,(a1)  "
        rts      quit subroutine
arg1 ds.l 1 first argument
arg2 ds.l 1 second argument
Biggr ds.l 1 answer = bigger_argument
```

## Program Counter Relative Addressing

- Program counter relative addressing is useful for writing Position Independent Code (PIC).
- Program can be moved anywhere in memory and will still run. Very useful in multitasking environments.
- All data must be located relative to the program.

```
move.w table(pc),d0
.....
table dc.w $abcd,$1234,$1024,$cdef
```

- As long as **table** maintains the same relative position, the program will execute properly no matter where in memory the program and table are located

## SPIC: Statically Position Independent Code

```
org $400400
*****
* FIND LARGER OF TWO NUMBERS
*****
* code is SPIC
* self modifying
* not reentrant
*****
larger move.l arg1(pc),d1 1st argument
        cmp.l  arg2(pc),d1  compare arguments
        bgt.s  xx      skip if arg1 larger
        move.l arg2(pc),d1  else larger_arg to d1
xx      lea    bigger(pc),a1  put larger_arg
        move.l d1,(a1)  in "bigger"
        rts      quit subroutine
arg1 ds.l 1 first argument
arg2 ds.l 1 second argument
biggr ds.l 1 answer = bigger_argument
```